PROGRAMMING SYSTEMS FOR SPARSE ACCELERATORS

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Olivia Weiya Hsu

December 2025

Abstract

Generations of hardware development have centered around executing dense-data computations on general-purpose (von Neumann) architectures. However, with the end of Moore's law and slowdown of performance scaling, interest is shifting toward designing hardware tailored to specific computational domains. Focusing within one domain allows for efficient software and hardware by leveraging domain knowledge, applying targeted optimizations, and hardening computations to fit specific data and control patterns. This trend has fueled a new surge of hardware designs and proposed architectures. Yet, comparatively little attention has been given to how such new accelerators should be programmed effectively, if at all.

Narrowing in, a class of recent domain-specific hardware targets computations on sparse data stored in compressed data structures. The motivation for accelerating sparse operations is threefold: first, sparse computations are central to a wide range of real-world applications; second, their irregular memory access patterns and control flow make them particularly inefficient on traditional hardware; third, their dynamic computational performance (i.e., depends on the exact data) leads to an extremely complex design search space when compared to their dense counterparts. Therefore, this dissertation focuses on programming systems that effectively program sparse accelerators.

Specifically, this thesis lays a foundation for how to program hardware accelerators for sparse computations from high-level sparse languages—or sparse domain-specific languages (DSLs). From this foundation, it introduces a novel reconfigurable hardware architecture designed to support the full generality of sparse tensor algebra. The research culminates in four programming systems and one fabricated system-on-chip (SoC) accelerator: The compiler for generalized sparse tensor programming (formerly sparse array programming), the sparse abstract machine (SAM) and its compiler, the Stardust compiler, and the Onyx accelerator.

First, a new **compilation theory for generalized sparse tensor programming (formerly sparse array programming)** introduces *set abstractions* that are essential for targeting sparse accelerators that follow a dataflow execution model. Second, the **Sparse Abstract Machine (SAM)** presents a *dataflow machine abstraction* for sparse tensor algebra computations, incorporating these set abstractions. SAM bridges traditional imperative programming models in higher-level software with hardware-oriented dataflow execution. This abstraction further enables reuse across multiple sparse accelerator backends, unifying the space of sparse dataflow accelerators. A front-end compiler translates the sparse DSLs into SAM, forming a concrete programming system.

This foundation enabled the design of **Onyx**, the first fabricated SoC based on a coarse-grained reconfigurable array (CGRA) architecture capable of computing both sparse and dense tensor operations within the same fabric. We co-designed Onyx with SAM using an agile hardware design approach. This work provides a back-end compiler that translates SAM to Onyx bitstreams, forming a complete programming system from sparse DSLs to fabricated accelerator hardware.

Stardust provides an alternative end-to-end programming system from sparse DSLs to real accelerator hardware. Stardust's compilation path lowers imperative loop-based abstractions directly to a pre-existing sparse dataflow architecture. Much of Stardust's novelty lies in its ability to transform imperative loops to *explicit memory hierarchies*. Finally, **Mosaic** scales out the programming stack to support multiple sparse accelerators by compiling the sparse DSLs to *external kernel library interfaces* provided for each backend, like those generated by Stardust and SAM.

Taken together, these contributions provide both the theoretical underpinnings and practical systems necessary for scaling the programming of sparse accelerators, thus transforming how we approach performance and programmability in this era of domain-specific computing.

Acknowledgments

"No man will make a great leader who wants to do it all himself or get all the credit for doing it."

Andrew Carnegie

The ideas and systems presented here are the work and effort of many people, whom I would like to thank. Without them, my career and life would not be what they are today. Rawn Henry, Stephen Chou, Rohan Yadav, and I refined and published a compiler for sparse array programming that was built from the foundations of Rawn's master's research. Alex Rucker was extremely instrumental in my first attempt at a sparse compiler to dataflow hardware (*Stardust*), and Tian Zhao and Varun Desai helped us benchmark this work. Maxwell Strange, Jaeyeon Won, and Ritvik Sharma supported me through the work that I am most proud of: *The Sparse Abstract Machine (SAM)*. Max, Kalhan Koul, Bo-Wun Cheng, Jack Melchert, Po-Han Chen, Qiaoyi Li, Keyi Zhang, Taeyoung Kong, Jake Ke, Michael Oduoza, Zhouhua Xie, members of the Agile Hardware (AHA) project, and I used the abstractions and compiler developed in SAM to create *Onyx*, the first fabricated reconfigurable accelerator for sparse tensor algebra.

Max deserves his own acknowledgment because much of the work presented in the thesis on SAM and Onyx is as much his as it is mine. Additionally, Kalhan was key in integrating the ideas and work of many collaborators together for Onyx's fabrication, testing, and publication. Rubens Lacouture, Nathan Zhang, Marco Siracusa, Ritvik Sharma, and I expanded upon the abstract machine and compiler ideas from SAM through FuseFlow, which is still in progress.

While developing ideas and compilers from sparse tensor algebra to dataflow accelerators during my PhD, I was also inspired by the limitations of sparse compilation theory. Manya Bansal and I developed ideas in *Mosaic* that allow us to systematically target multiple high-performance libraries for more general tensor computations. Much of the work presented in this dissertation on *Mosaic* is Manya's contribution, and I thank her for her generosity in letting me include it. Genghan Zhang and I developed a theory on generating CPU code with sparse temporary tensors. Although I do not talk about this work in the dissertation, it completed some key missing pieces in generating high-performance sparse code from prior compilers. I especially want to thank the people whom

I have been able to mentor during my PhD, including Manya, Genghan, Rubens, Gina Sohn, Sai Gautham Ravipati, Varun, Anderson Truong, Zhouhua, Gloria Tumushabe, Jayashree Adivarahan, Parthiv Krishna, and Akhilesh Balasingam. Everyone is truly a powerhouse in their own way, and seeing them grow as researchers and succeed in their academic careers has been extremely fulfilling. They are the reason I became a professor.

I am extremely grateful to have learned from and been guided by my advisors, Kunle Olukotun and Fredrik Kjolstad. For the past six and a half years, they have both supported me fully. Together, Fred and Kunle bring a productive dichotomy that has been critical to my success: combining new perspectives with experienced wisdom, hands-on technical exploration with high-level strategic direction, and interesting ideas with grounded systems performance. The many ideas and projects described in this thesis would not be possible without them. To Kunle, thank you for always having my back and for letting me disagree with you; you are usually correct. To Fred, thank you for your time and for helping me with the details. I feel like I can come to you for advice on anything. Although my PhD is ending, I know they will continue to shape my academic career.

There are also many other mentors and collaborators who supported me during my PhD whom I would like to thank. Joel Emer has been one of the best mentors I could ask for. Joel is warm and kind, and he will make your research better with his precision, technical rigor, taxonomies, and inquisitive nature. I am honored to be a member of Team Einsum and the Joel Emer School of Architecture.[1] Next, Mark Horowitz is another mentor who has helped me get to where I am today. I am grateful to be one of the students who has gotten the full-Horowitz experience. Thank you for being interested in my technical projects and ideas (once you understand them, of course), for having technical disagreements with me and others that deepen our own understanding of our work, and for providing your perspective over the years. Saman Amarasinghe was my first faculty collaborator outside of Stanford and taught me a lot about the MIT way. Saman is truly unique, and I strive to have a career in compiler research that is even remotely as prolific as his. Zachary Myers and Yaqi Zhang were some of my first student mentors when I started at Stanford. They helped me navigate a new place, new research, and a new community. I would also like to thank Ardavan Pedram, Luigi Nardi, Priyanka Raina, Doug Joseph, Michel Steuwer, Caroline Trippel, Thierry Tambe, Jonathan Balkind, Miquel Moreto, Muhammad Shahbaz, and other mentors who helped me along the way.

Being co-advised, I am deeply grateful to have been a part of two labs at Stanford: the self-proclaimed Compiler Group[2] and the Pervasive Parallelism Lab. The first couple of years of my PhD were pretty lonely given COVID, and Rohan Yadav and Scott Kovach were some of the first friends who made me feel like I had a larger community at Stanford. I would like to thank: the rest of the Compiler Group, including Shiv Sundram, Marco Siracusa, Alexander (AJ) Root, Christophe Gyurgyik, Rubens, Manya, Gautham, Bobby Yan, Trevor Gale, James Dong, Haoran Xu, Jacob Torring, Timothy Gu, Usman Tariq, Advay Pal, Matthew Lee, Praneeth Kolichala, and Bala

[1]Credits to Bobbie Manne.
[2]The self-proclaimed Stanford Compiler Group, despite Fred's wishes.

Vinaithirthan; the Pervasive Parallelism Lab, including Yaqi Zhang, Alex Rucker, Tian Zhao, Tushar Swamy, Matthew Feldman, Matthew Villim, Nathan Zhang, Rubens Lacouture, Gina Sohn, Qizheng Zhang, Paul Mure, Sho Ko, Jungwoo Kim, Konstantin Hossfeld, and Nathan Sobotka; and other Stanford collaborators and friends, including Anjiang Wei, Ben Driscoll, Sam Archer, David Zhang, Kathleen Feng, Simran Arora, and Kaylee Burns.

The collaborators above only encompass my work life, and I would also like to thank my friends. First, to my childhood friends Julia Riedy (and her family), Maria Bobbett, Katie Cox, Gerry Wan, William Liu, Andrew Haymaker, and Raymond Yang, thank you for supporting me throughout the years. To my friends from undergrad, including Aviral Pandey, Gloria Tumushabe, Travis Brashears, Nealah Lee, Malhar Patel, Dalton Omens, Tavor Baharav, Janny Zhang, Celine Veys, Barbara Shvareva, Harrison Zheng, Hersh Sanghvi, Greg Balke, Kevin Zheng, David Zhang, Avinash Jois, James Zhu, Andrew Dong, Michael Weymouth, and Kevin Chen, thank you for the community during COVID and when I first started graduate school, for regular hangouts and dinners in the Bay, and for soup (Supe!). To the friends I met after coming to Stanford, including Dennis Rich, Kartik Prabhu, Maggie Skortcheva, Saarthak Sarup, Raman Vilkhu, Manj Kalkat, Alex Bergman, Kavya Sreedhar, Gary Cheng, Robert Radway, Vamshi Balanaga, Rohith Kuditipudi, Tathagata Srimani, Sam Clarke, and Anirudh Jain, thank you for movie nights, mid-day walks, outdoor lunches, dinners, getting me addicted to board games, back-to-back concerts, and much more. To my chosen family Jenny Lai, Jeffrey, Hu Mama, Lai ShuShu, and Chloe, thank you for being my support system in California. My memories of Stanford would not be complete without our holidays, picking fruits and veggies in your backyard, and our walks with Chloe. Finally, to anyone else I forgot to mention, I am deeply grateful. Any omission reflects my memory at the end of a long PhD, not the impact you have had on my life and work.

I want to give a heartfelt thank you to Jack Melchert, my partner in crime, compiler buddy, and best friend. You fill every day with goofy smiles, uncontrollable laughter, and infinite joy. You gracefully listen to me and are often my voice of reason. A small anecdote: Jack always drives me to and from the airport, no matter how late or early, whenever I need to take a business trip. This was especially meaningful during faculty interviews, when we did not see each other for months at a time. I will continue to cherish this kindness in the future. Thank you for your unconditional support during this PhD journey and as we start our next one together.

And to my parents: you both gave me the tools I needed to not only pursue but also succeed in my PhD. From a young age, you taught me to be curious, critical, and creative. My ability to deeply understand and debug technical problems, grasp the limits of what I know, and be unsatisfied with the status quo comes from you. In my parents' eyes I can do no wrong, and that, in turn, has taught me to reach for the stars. Thank you, Mom and Dad; my accomplishments are yours as well.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

"Nothing in life is to be feared, it is only to be understood."

Marie Curie

Sparse operations perform computation on sparse data, where many of the data values are empty or zero. They are an important class of computations that underlie many real-world applications, like machine learning [61], image processing (see Chapter 3), graph and data analytics [50], physical simulations, and more. Beyond their importance, this data sparsity creates the opportunity to reduce storage and avoid ineffectual operations by removing the redundant zeros through compression. A class of sparse operations (and their algorithms) natively operate on this compressed representation of the sparse data rather than on its equivalent uncompressed data, which is the focus of this dissertation. These sparse operations become fundamentally more complex than their dense counterparts, making them technically interesting to study. Their complexity stems from the fact that both the computation and its performance depend on how the sparse data is compressed and on the data values (sparsity patterns) themselves. The complexity makes them hard to program and introduces many new algorithmic optimizations. Ultimately, these sparse computations are run on hardware, and the complex nature of these operations significantly impact hardware performance and utilization.

While sparse operations traditionally run on general-purpose hardware, accelerators have also been proposed to run them. Accelerators have emerged as a hardware alternative that achieves performance—beyond that of technology scaling—by specializing the hardware design to a specific application or computation domain. In particular, sparse accelerators specialize to the domain of sparse computations. Specialized accelerator hardware can, therefore, achieve performance by leveraging assumptions and patterns specific to the domain. In the case of accelerator hardware specialized for sparse operations, designers achieve performance by focusing on patterns like compressed storage and tiling [168, 155], skipping ineffectual computation [81, 176, 45], and hardware targeted for irregular memory accesses [176].

Many of these proposed accelerators, whether fixed-function for a specific sparse kernel or reconfigurable to many sparse expressions, also take advantage of dataflow for performance. Dataflow allows these accelerator architectures to execute based on the availability and locality of data, rather than a sequential program counter as in general-purpose von Neumann architectures. Dataflow accelerators lay out computation in space, which look more like a native VLSI implementation. Beyond the complexities that arise from sparsity, sparse dataflow accelerators add more complexity due to considerations like resource constraints, low-level design optimizations, and hardware scheduling. Often, the programming of these sparse hardware accelerators has been left as an afterthought.

The use of hardware accelerators for sparse operations may offer significant performance benefits [81, 150, 176, 45, 78, 35, 157, 169, 252, 246, 202, 201, 240, 255, 128], but their broader adoption is hindered by the difficulty of programming them effectively. Without developments in their programming systems, accelerator use is not scalable. Given that the inception of sparse accelerators is relatively recent, existing approaches often rely on low-level programming techniques [43], such as hand-stitching custom hardware abstractions [81], writing in a modified assembly language, or tuning configuration files. These methods demand intimate knowledge of both the sparse algorithm and the target architecture, making them inaccessible to most users. Furthermore, writing and optimizing low-level code becomes increasingly burdensome as sparse applications scale in size, optimization opportunities, and complexity. Developers must reason about both the structure of sparse data and the nuances of accelerator design, which significantly slows down iteration and limits programmability. As a result, current practices do not scale to diverse users, applications, or optimization goals. The history of dense accelerators has shown, accelerator software often lags behind hardware design by many years [183] and requires industry-level resources [43], underscoring the need for abstractions and automation in sparse accelerator programming throughout the programming stack. Therefore, we cannot expect sparse accelerator adoption in the near future without the creation of more mature programming systems for sparse accelerators.

In order to facilitate the use of the sparse accelerators that are currently being developed, I identify in this thesis *programming abstractions for sparse accelerators* that enable *end-to-end sparse programming systems*. These abstractions enable new compiler algorithms that lower from high-level domain-specific languages (DSLs) to generate low-level code running on real hardware accelerators. In this dissertation, I demonstrate the following:

Thesis Statement: With a careful choice of abstraction, it is possible to build simple, end-to-end programming systems for complex, sparse hardware.

The abstractions proposed in this thesis not only simplify the implementation of programming systems, but also take a first step in generating unified and portable code across different sparse accelerators. In Figure 1.1, I show a generalized end-to-end domain-specific programming stack that lies atop sparse hardware accelerators. In order for an end-to-end programming system to be complete, it must contain many components—languages, compilers, runtime systems, kernels, abstractions, scheduling systems—that are intertwined. The ideas presented in this dissertation focus on intentional abstractions to produce well-designed programming systems, which allow us to simplify this programming stack. I believe that the intentional abstractions and IRs presented in this dissertation provide enough information to produce straightforward compiler lowering algorithms while leaving room for domain-specific and lower-level optimizations. Beyond simplicity, the abstractions and algorithms proposed in this dissertation also enable many hardware designs to share pieces of the programming stack. In other words, the research ideas in this dissertation decouple the software from the hardware, similar to an ISA, leading to portable software and compilers across many accelerator implementations.

Given the explosion of proposed sparse accelerator designs, I also tackle the problem of programming beyond individual accelerators in this dissertation. The programming systems I propose also consider unified abstractions that can lead to portable code across many different accelerator hardware implementations. These abstractions unify and modularize across accelerator commonalities but still allow for freedom in implementation differences. More specifically, the abstractions contain requirements on functionality across a sparse operations execution model while leaving room for the different low-level optimizations and implementation details that make accelerators performant.

1.1 Contributions

This dissertation presents a comprehensive approach to programming abstractions for sparse accelerators and its application to sparse tensor algebra compilation to dataflow accelerators. I have used the ideas behind the approach to build compilers that can generate efficient code on individual sparse accelerators and across multiple sparse accelerators for any sparse tensor algebra expression, for many data structures, with different optimizations. My contributions are:

- **A generalization of sparse tensor algebra compilation** to any user-defined function and fill value. This generalization makes explicit the set algebra of sparse iteration spaces for any set operation.

- **A sparse abstract machine representation** that lets us describe sparse tensor algebra as abstract streaming dataflow graphs. Since streaming dataflow is more aligned with the design and execution of hardware accelerators, it enables simpler transformations from the abstraction to hardware.

- **Algorithms that lower** high-level sparse tensor languages to streaming dataflow models.

- **A reconfigurable sparse accelerator design** that implements the sparse abstract machine representation in a coarse-grain reconfigurable array (CGRA) architecture.

- **Code generation algorithms** that compile sparse abstract machine representations to architectural dataflow simulations and hardware bitstreams.
- **An external function interface** that lets us compile tensor algebra expressions to multiple accelerators, and other high-performance libraries, through their existing library kernel interfaces. The interface defines the algorithm to generate code targeting a given external function and specifies the tensor algebra expressions it can compute.
- **Compiler verification, mapping, and code-generation algorithms** that automatically validate external-function mappings, rewrite eligible sub-expressions into those functions, and generate compiler-native code for all remaining unmapped computation.

These ideas were used in the design and implementation of various programming systems for sparse accelerators. Figure 1.1 shows how these contributions fit together across the entire programming stack from high-level sparse DSLs to low-level sparse specialized hardware accelerators. Figure 1.2 and Section 1.2 detail how these contributions and their various programming systems fit together conceptually within this software–hardware programming stack. My dissertation provides solid footing for programming sparse accelerators, but it only scratches the surface. There are several unexplored questions left for future work as described in Chapter 8.

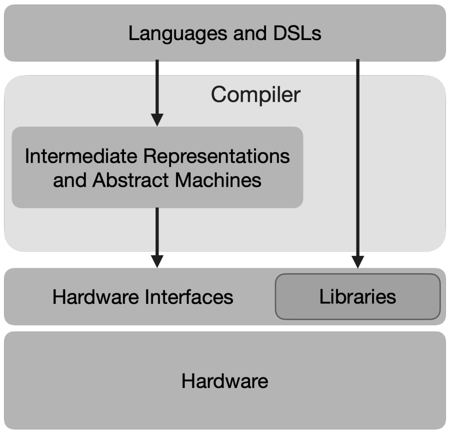


Figure 1.1: Components of a domain-specific programming stack. This dissertation focuses on developing this stack for sparse specialized hardware.

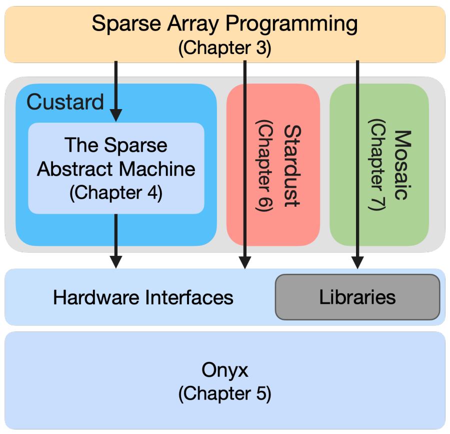


Figure 1.2: Dissertation overview. A conceptual diagram of how the systems in this dissertation fit together mirroring the components in Figure 1.1.

1.2 Dissertation Overview

This dissertation is organized as follows:

Chapter 2 - Background motivates the need for sparse tensor computation and provides broader context on the programming systems and hardware that existed prior for sparse computations.

It also provides background on how sparse hardware accelerators were programmed before the work presented in this thesis.

Chapter 3 - Generalized Sparse Tensor Programming introduces high-level domain-specific languages (DSLs) for generalized sparse tensor computation and shows how to transform them into two abstractions needed for programming sparse hardware: a loop-based index notation and explicit set expressions.

Chapter 4 - A Unified Sparse Dataflow Abstraction describes an abstract machine model that represents sparse tensor algebra as dataflow graphs and an algorithm from the high-level sparse languages presented in Chapter 3 to this sparse dataflow abstraction.

Chapter 5 - A Reconfigurable Sparse Accelerator details the architecture of the first, to our knowledge, fabricated programmable accelerator that spans all of sparse tensor algebra and the first, to our knowledge, fabricated CGRA fabric with both sparse and dense tensor computation. The accelerator was co-designed with our unified sparse dataflow abstraction (Chapter 4), leading to an end-to-end approach that compiles from high-level sparse languages to the reconfigurable sparse accelerator. The compiler introduces techniques that turn an abstract, infinite-resource dataflow machine into real hardware constraints with lower-level optimizations necessary for performant acceleration.

Chapter 6 - Alternative to Targeting Sparse Hardware describes a second end-to-end approach that compiles from high-level sparse languages to a real sparse accelerator architecture. This compiler flow lowers explicit set expressions to abstract data-parallel constructs and loop nests to target a bitvector architecture with explicit memory.

Chapter 7 - Scaling to Multiple Accelerators presents a plug-in interface to target multiple hardware accelerators and high-performance libraries. Given this novel interface for multiple libraries, this section also describes algorithms for automatically mapping expressions to these libraries, verifying that the mappings are correct, and generating code to call these libraries efficiently.

Chapter 8 - Conclusions concludes this dissertation and includes promising directions for future research.

Through these chapters we describe the following systems:

A Compiler for Generalized Sparse Tensor Programming: High-level sparse tensor languages and a compiler from those languages that generalize beyond sparse tensor algebra and lower to CPU programs.

The Sparse Abstract Machine (SAM): A sparse dataflow abstraction and its front-end compiler, Custard, from high-level sparse tensor languages

Onyx: The first fabricated reconfigurable accelerator for sparse and dense tensor computation on the same fabric and its back-end compiler from SAM to the Onyx bitstream.

Stardust: Another compiler from high-level sparse tensor languages, Stardust, to a variant of the hardware domain-specific language (DSL) Spatial that includes sparse iteration.

The Mosaic Compiler: A compiler that allows users to plug-in and target multiple external libraries, which include kernel libraries to sparse accelerators.

Figure 1.2 shows how these programming systems fit together. The rest of this dissertation motivates, describes, and evaluates the design and implementation of the various programming systems for sparse accelerators listed above.

1.3 Publication and Disclosure Statements

The following chapters of this dissertation are based on the following previously published work:

Chapter 3 Generalized Sparse Tensor Programming: Rawn Henry et al. “Compilation of Sparse Array Programming Models”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485505. URL: https://doi.org/10.1145/3485505

Chapter 4 A Unified Sparse Dataflow Abstraction: Olivia Hsu et al. “The Sparse Abstract Machine”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3.* ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 710–726. DOI: 10.1145/3582016.3582051. URL: https://doi.org/10.1145/3582016.3582051

Chapter 5 A Reconfigurable Sparse Accelerator: Kalhan Koul et al. “Onyx: A 12nm 756 GOPS/W Coarse-Grained Reconfigurable Array for Accelerating Dense and Sparse Applications”. In: *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits).* 2024, pp. 1–2. DOI: 10.1109/VLSITechnologyandCir46783.2024.10631383

Kalhan Koul et al. “Onyx: A 12-nm Programmable Accelerator for Dense and Sparse Applications”. In: *IEEE Journal of Solid-State Circuits* (2025), pp. 1–13. DOI: 10.1109/JSSC.2025.3604724

Chapter 6 Alternative to Targeting Sparse Hardware: Olivia Hsu et al. “Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture”. In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization.* CGO ’25. Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 628–643. DOI: 10.1145/3696443.3708918. URL: https://doi.org/10.1145/3696443.3708918

Chapter 7 Scaling to Multiple Accelerators: Manya Bansal et al. “Mosaic: An Interoperable Compiler for Tensor Algebra”. In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591236. URL: https://doi.org/10.1145/3591236

All previously published material incorporated appears with the full permission of my collaborators and reflects collaborative work for which the original contributors retain credit. In addition, generative AI tools were used to assist with writing edits to improve the clarity of this dissertation. All technical content, results, claims, and contributions are my own.

Chapter 2

Background

"Anything wise in these pages you should credit to the many experts who preceded me. Anything foolish, assume it is my error."

James Clear, Atomic Habits

This chapter provides the background required to understand the contributions and systems within this dissertation. First, I motivate sparse operations and define the class of sparse operations that this dissertation focuses on. By sparse operations, I specifically mean applications with sparse tensor algebra operations. Within the background of sparse tensor algebra, I provide the necessary features of any general sparse tensor algebra computing system (Section 2.1). Next, I survey various prior approaches to programming systems for generating sparse tensor algebra code. The approaches from this section mostly present sparse tensor algebra code for general-purpose hardware (i.e. CPUs and GPUs). Two major approaches include kernel libraries for sparse linear algebra (Section 2.2.1) and programming systems that can generate general sparse tensor algebra code (Section 2.2.2). Within the survey of general sparse tensor algebra programming systems, I emphasize background on sparse compiler approaches since the work described in this thesis builds upon a larger body of work on sparse compilation and sparse iteration theory [110]. I then provide background on sparse dataflow accelerators and survey the design of prior sparse accelerators and their associated programming model and stack.[1] For the descriptions above, I intentionally provide background on systems that predate the first set of publications in Section 1.3 so the background cleanly reflects the state of the field at the onset of the work in this dissertation. This chapter concludes by presenting recent developments on sparse programming systems and sparse accelerators (Section 2.4) that emerged after the work in Section 1.3 began but before the publication of this dissertation.[2]

[1]The programming and accelerator background in this chapter (Section 2.2 and Section 2.3) are differentiated by fixed-function versus general approaches for categorization purposes. However, much of the research and systems presented in this dissertation apply to both fixed-function and general/reconfigurable hardware and their programming.

[2]More detailed related work also appears in the corresponding sections of each chapter.

$$\begin{array}{|c|c|c|c|}\hline a & b & c & d \\ \hline e & f & g & h \\ \hline i & j & k & l \\ \hline \end{array}$$

(a) Example dense matrix.

$$\begin{array}{|c|c|c|c|}\hline a & b & 0 & 0 \\ \hline 0 & 0 & c & 0 \\ \hline 0 & d & 0 & e \\ \hline \end{array}$$

(b) Example sparse matrix.

i coordinates	0	0	1	2	2
j coordinates	0	1	2	1	3
values	a	b	c	d	e

(c) Compressed representation of Figure 2.1b.

Figure 2.1: Example dense matrix (Figure 2.1a), sparse matrix (Figure 2.1b), and sparse matrix stored in a coordinate list (COO) compressed data structure (Figure 2.1c).

2.1 Sparse Tensor Algebra

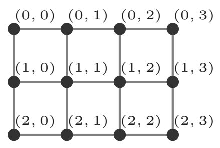
The sparse operations in this thesis focus on sparse data that is already in a compressed data structure, where the computational patterns and algorithms have to directly traverse through and operate on those compressed data structures. In this thesis, I focus on a subset of those sparse operations known as *sparse tensor algebra*. Sparse tensor algebra expressions extend linear algebra to higher-order tensors, where at least one or more of the tensors is sparse and stored using a compressed data structure. A sparse tensor has many zero (or empty values) that are compressed away. Without loss of generality, when I refer to zeros in this dissertation, it means any compressed away value. An example 4×3 sparse tensor (matrix), and how it differs from a dense tensor, is shown in Figure 2.1 along with its equivalent representation as a compressed set of arrays (i.e. its compressed data structure).

As mentioned in Chapter 1, sparse tensor algebra occurs in many real-world applications like graph analytics, machine learning, physical and circuit simulation, signal processing, and bioinformatics, just to name a few. Real-world applications often decompose into one or more sparse tensor algebra expressions. For example, machine learning models perform a series of matrix multiplications with model weights and activations. If the model weights have been sparsified and compressed, the computation must therefore perform a sparse-matrix times dense-matrix multiplication (SpMM). SpMM has become one of the most notable sparse tensor algebra expressions, but many other important sparse tensor algebra expressions exist depending on their application [61, 60, 27, 113, 101, 1, 159, 108, 110].

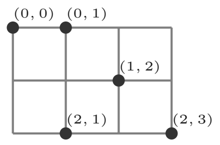
The many properties of sparse tensor algebra expressions make them tricky to compute. Beyond the challenge of sparse iteration, there exists a combinatorial problem for sparse operations that does not exist for their dense counterparts [110].

$$\underset{\texttt{(algorithm)}}{\texttt{expression}} \times \underset{\texttt{(format)}}{\texttt{data structure}} \times \underset{\texttt{(schedule)}}{\texttt{optimization}} \times \underset{\texttt{(backend)}}{\texttt{architecture}}$$

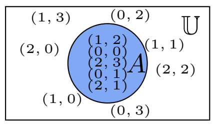
The domain of sparse tensor algebra spans the diverse expressions that appear across real-world applications, the compressed data structures that store each tensor, and the optimizations that apply to each expression–structure pair. Examples include sparse matrix–vector multiplication



(a) Dense iteration space of Figure 2.1a, with all points present.



(b) Sparse iteration space of Figure 2.1b, with some points missing.



(c) Set interpretation of Figure 2.2b.

Figure 2.2: A grid representation of iteration spaces showing a dense and example sparse iteration space tied to the two 4×3 matrices in Figure 2.1.

(SpMV), sparse–dense matrix multiplication (SpMM), and sampled dense–dense matrix multiplication (SDDMM), each interacting differently with formats such as compressed-sparse row (CSR) [211], coordinate list (COO) [179] shown in Figure 2.1c, doubly-compressed sparse row (DCSR) [26], or doubly-compressed sparse column (DCSC). Even the best-performing code optimizations—loop order, tiling, or fusion strategy—will vary with the architecture since each favors different data layouts and concurrency patterns. Furthermore, there exist many architectures to target: CPUs, GPUs, various accelerators, distributed machines of architectures, and more. Collectively, these axes define a vast, discrete design space where small changes in data representation or sparsity can alter the performance of code, and what constitutes the most efficient schedule. This combinatorial nature underpins the challenges faced by both sparse compilers and hardware designers.

To produce efficient sparse tensor code within this design space, we must understand how to iterate over sparse tensors and how the iteration differs from that of dense tensors. Whereas dense tensors admit regular, rectangular iteration domains, sparse tensors introduce holes in the grid that can be skipped as they are ineffectual. The iteration pattern is therefore data dependent, since the number and location of points visited depend on which coordinates contain explicit nonzeros. Again consider the 4×3 dense matrix in Figure 2.1a. We can view the iteration space of loops over dense tensors as a hyper-rectangular grid of points by taking the Cartesian product of the iteration domain of each loop, as in Figure 2.2a. A sparse iteration space, shown in Figure 2.2b, is a grid with missing points called holes, which take on *implicit zeros*. Any zeros that are stored in the compressed representation explicitly are called *explicit zeros*. The iteration space of compressed (sparse) tensors is fundamentally an abstract set of coordinate points, independent of any particular storage format or tensor data (the data space). Any sparse iteration space can also be thought of as an abstract set, with different iteration behavior based on the set regimes. At runtime, the iteration space becomes tied to the exact input data space, and all explicit valued points fall within the set and all implicit zero points outside of the set (see Figure 2.2c). We visualize an example sparse iteration space and set interpretation in Figures 2.2b and 2.2c anchored to the example 4×3 sparse matrix in Figure 2.1b. The figures serve only as illustrative instances of the more abstract notion of sparse iteration spaces.

The abstract set of points in the sparse iteration space must eventually be materialized on a concrete data structure. Sparse code operates not on an idealized grid but on compressed formats such as coordinate list (COO) or compressed sparse row (CSR), which store metadata arrays that index into nonzero values. Iterating over these formats requires indirect lookups through coordinate and position arrays–introducing pointer chasing and control irregularity [108]. For example, in the COO representation Figure 2.1c, the program must read the next pair of (i, j) coordinates from separate arrays to locate the corresponding value, skipping positions that do not exist in the compressed storage. Each such dereference depends on the data itself, causing unpredictable branches and cache misses. As a result, runtime behavior depends on both the tensor's dimensions and the number and distribution of nonzeros. These irregular memory accesses make sparse tensor algebra inherently data dependent and difficult to optimize in the same uniform ways as dense computation.

The combinatorial and data-dependent nature of sparse tensor algebra defines a rich and complex design space. The rest of this chapter outlines this design space and how prior systems have attempted to navigate it.

The Design Space of Sparse Tensor Algebra Having introduced the combinatorial challenges of sparse tensor algebra, we now turn to how components of the combinatorial problem are represented and the space of choices they create. Tensor algebra expressions are typically expressed using tensor index notation (or Einsum notation) [57], where tensors are indexed by index variables and combined through additions and multiplications, where results may be summed over index variables. For example, matrix multiplication can be written in tensor index notation as $X_{ij} = \sum_k B_{ik}C_{kj}$, where index variables i, j, and k range over the rows and columns that they index. Oftentimes, the explicit summation (Sigma) is omitted. Expressions may have more than two operands, such as sampled dense-dense matrix multiplication (SDDMM) $X_{ij} = \sum_k B_{ij}C_{ik}D_{jk}$. For such compound expressions, it is often beneficial to fuse the resulting computation (i.e., loop fusion or hardware pipelining to avoid materializing large temporary data structures). Tensor index notation only specifies the expression (or algorithm of computation) and does not include a description of the schedule (e.g. dataflow traversal order, tiling methodology, and parallelization) or the format of the tensor. Prior work popularizes the separation of algorithm and schedule [170, 181, 36, 216] and later format [41, 40] in both software and hardware [34].

Tensor index notation consists of five features that must be supported by any general tensor algebra computing system without reducing the computation to a simpler problem (such as factorizing into a sequence of fixed kernels): transposes and matrix multiplications. These features are:

1. a way to traverse multidimensional tensors;

2. a way to combine traversal over multiple tensors;

3. a way to repeat operands over other operands, e.g., in $x_i = \sum_j B_{ij}c_j$, c must be multiplied by each row of B;

4. a way to compute scalar additions and multiplications, including summation reductions; and

5. a way to assign results to a tensor.

Efficient sparse tensor algebra computing systems must also support

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

in order to avoid inferior worst-case asymptotic complexity [110, 4]. Together, tensor index notation, the compressed data structure language, and the optimization schedule define the key axes of the sparse tensor algebra design space: what computation is performed, how tensors are stored, and how the iteration is ordered and optimized.

2.2 Programming Systems for Sparse Code

There are two major approaches to interfacing with sparse tensor algebra expressions: libraries that provide one or more fixed-function kernels (Section 2.2.1) and general programming systems that support the entire domain (Section 2.2.2). For clarity, I define a *sparse kernel library* as a software library or API that exposes a finite set of pre-implemented sparse kernels—each corresponding to a specific algebraic expression (e.g., SpMV, SpMM, SDDMM)—and does not generalize to arbitrary or user-defined combinations of tensor operations. In contrast, a general programming system provides a language, intermediate representation, or runtime system that can express and run an unbounded number of sparse tensor algebra kernels by varying their structure, schedule, or storage format. This section first surveys these two varied approaches and then examines the subcategories of general programming systems for sparse tensor algebra in more detail.

2.2.1 Sparse Tensor Algebra Kernel Libraries

Performance is crucial in many applications with sparse tensor operations, resulting in a proliferation of libraries for CPUs [126, 95, 90, 224, 69, 98], GPUs [149, 46], vector processors [94, 38, 203], and domain-specific hardware [176, 45, 81, 99, 246, 169, 35, 202, 157, 201, 80]. Notable sparse libraries include Intel MKL [95] and MatLab Tensor Toolbox [12] for CPUs and CuSparse for GPUs [149]. Most of these libraries include a select-few fixed-function library kernels, each for a specific sparse tensor algebra expression. These libraries have been optimized at great expense and effort, often by an expert. Within the library kernel, sections of code may also be generated or compiled but each kernel function is still limited to a fixed-expression that is parameterizable. As a result, most application code that contains tensor algebra, both sparse and dense, is written as a sequence of calls to libraries that compute different sub-expressions.

There are also several library frameworks that can be categorized as sparse kernel libraries despite leveraging more general programming techniques. For instance, although the internals of Eigen [73] are built on composable expression templates that permit limited compile-time fusion, its sparse operations are still implemented as a collection of pre-defined kernel abstractions. Similarly, `scipy.sparse` [98] and PyTorch's sparse module [160] provide user-facing APIs for sparse matrices and tensors, but they only support a limited subset of operations or data structures on these sparse tensors. Their backends further delegate computation to fixed-function kernels implemented in C++ or CUDA. These frameworks enable some flexible composition at the application level, yet they ultimately rely on a small, finite set of optimized sparse kernels underneath. Consequently, developers must decompose larger sparse tensor computations into sequences of library calls, leaving performance across kernel boundaries dependent on the efficiency of data materialization and transfer between intermediate results.

2.2.2 General Sparse Tensor Algebra Programming Systems

General sparse tensor algebra programming systems, on the other hand, can generate and run code for multiple sparse tensor algebra expressions, across many data structures, with one or many optimizations to each expression (as described in Section 2.1).

Factorization

There are general programming systems that can handle any sparse tensor algebra expression by factorizing the expression into a known set of sub-computations. Although these frameworks and libraries are general, they often lead to runtime overheads when compared to sparse compilers that generate custom code as shown in Chapter 3 and other work [233, 244].

PyData/Sparse [2],[3] for example, is a Python library wrapped around dense NumPy that supports many sparse data structures. It iterates through sparse tensor data and dynamically packs them to determine which dense NumPy function to call. At its core, PyData/Sparse factorizes sparse tensor algebra computations into a reshape-and-pack phase across multiple dense kernel calls, followed by reshaping the results back into the desired sparse format. PyData/Sparse is also able to handle many data structures by defaulting to the dictionary-of-keys data structure [222, 2, 199] for any sparse tensor and reformatting to the requested compressed format. We show in Chapter 3 that this factorization introduces interpretive overhead and prevents compile-time optimization across operations.

The Cyclops Tensor Framework (CTF) [193] and MATLAB [13, 12, 137] are other examples of sparse tensor algebra frameworks that rely on algebraic factorization. CTF represents tensor algebra computations as contractions of multi-dimensional arrays and factors these contractions into distributed dense operations using communication-avoiding algorithms. CTF achieves scalability by

[3]Now the Numba backend for the Sparse library [199]

casting sparse and structured computations into block-sparse dense contractions, distributing them across processors, and mapping the computation to dense matrix multiplication (specifically the SUMMA algorithm [62]) with replication. However, this generality comes at the cost of specialization: it cannot exploit fine-grained sparsity patterns or dynamic sparsity during execution, leading to performance gaps compared to compilers that emit code specialized for each expression and data format [244, 233].

Factorization provides generality through reuse of dense kernels and numerical libraries, but at the expense of static specialization and data-dependent optimizations. In contrast, the sparse compilers introduced next analyze the algebraic structure of each expression to generate code specialized to its sparsity pattern, avoiding the overheads of runtime factorization.

Sparse Tensor Algebra Compilers

There is a long history of sparse linear and tensor algebra systems for CPUs and some GPUs, spanning early sparse linear algebra kernel libraries [140, 105, 179, 211, 178, 51, 16, 225, 20, 149, 191], sparse loop optimizations [143, 24, 116, 167, 216, 204], and sparse programming systems [65, 21, 12, 193]. Kjølstad [110] provides an in-depth analysis of this lineage in Chapter 8 of his dissertation. This body of research culminated in general sparse tensor algebra compilation through the TACO work [108, 106, 110, 41, 181]. Since then, new sparse programming systems [98, 160, 2] and sparse compilers have been developed, many directly influenced by TACO's structure and abstractions. Together, these systems provide the foundation on which we build the compilation for sparse accelerators in this dissertation.

The TACO compiler [108] compiles tensor index notation on tensors of varying compressed data structures [41] to von Neumann architectures, including CPUs, GPUs [181]. TACO supports all five features of tensor index notation (see Section 2.1), as well as compressed data structures [108, 41], iteration reordering [106], and fusion [108][4]. The TACO compiler has three separate languages that independently describe functionality, data, and optimization: tensor index notation, a data representation language, and a scheduling language. I provide more details on these languages later in Section 3.1. Since the introduction of the original TACO compiler the ideas have been implemented within MLIR [23] and extended by related systems to address tensor reuse [148], compositions of varying compressed formats [242], and targeting distributed machines of CPUs and GPUs [232, 233].

Although TACO has the generality we described in Section 2.1, it only compiles to von Neumann architectures. This limitation is due to its lowering machinery fundamentally embedding the (co-)iteration over one or more tensors into general-purpose control flow found in von Neumann machines including: indirect index accesses, `while` loops, and `if` statements. The heavy reliance on these constructs during lowering and code generation for traversing irregular structures makes it inapplicable for reconfigurable dataflow accelerators since many of these architectures remove general control

[4]Although its support for scattering into sparse results appears to be limited [106] and is fixed by [244]

flow to achieve higher performance [165, 176, 158, 28, 81, 45]. Converting the complex control flow generated by TACO into a dataflow abstraction, such as the one we describe in this dissertation, would require significant assumptions about data accesses and/or a complex synthesis system. Instead, this dissertation describes alternative compiler lowering algorithms that lower from TACO's high-level tensor index notation (or Einsum notation) [106] to the dataflow accelerators we describe in the next section.

2.3 Sparse Tensor Algebra Accelerators

In conjunction with the emergence of general programming systems for sparse operations, there has been an emergence of sparse hardware accelerator designs. To contextualize these designs, this chapter first introduces the dataflow foundations necessary for understanding sparse accelerator designs (Section 2.3.1). Then, I survey both fixed-function (Section 2.3.2) and programmable sparse accelerator architectures (Section 2.3.3), which are architectures capable of targeting multiple sparse tensor algebra expressions. Table 2.1 provides a survey of fixed-function accelerators for sparse tensor algebra from the literature, while Table 2.2 and Table 2.3 together compare prior programmable sparse accelerators.

2.3.1 Dataflow in Accelerators

Accelerator architectures often adopt dataflow execution because it exposes parallelism directly and naturally hardens computation into spatial pipelines for performance. Traditional von Neumann architectures like CPUs and traditional GPUs represent operations in time, relying on a program counter to sequence instructions and memories to communicate data, and achieves parallelism primarily through microarchitectural designs like wide superscalar dispatch, speculation, and multilevel caching. By contrast, dataflow accelerators materialize computation and operations in space with data flowing through wires before landing in memories. Operators fire when their inputs arrive, enabling high-throughput pipelines with predictable locality which can eliminate much of the control and scheduling overhead inherent in sequential execution models. This spatial form of parallelism underlies many modern accelerator designs, from RDAs [165, 218, 176, 219], CGRAs [45, 59], FPGAs [241], and modern GPU architectures [234, 33, 200] to domain-specialized accelerators like application-specific integrated circuits (ASICs) [34, 208, 246], and provides the architectural context for the abstractions introduced in later in this dissertation.

For sparse accelerators in particular, dataflow execution aligns well with the structure of sparse computations. These systems operate on streams of coordinates, references, and values produced during traversal of compressed data formats. Hardware primitives consume and produce these streams using ready–valid protocols, allowing nondeterministic progress as different operands advance at different rates. Merging (intersection/union), filtering, repetition, and reduction become explicit spatial blocks, and memory tiles convert between compressed storage and streaming tensor representations. This execution style mirrors the set-theoretic iteration spaces introduced earlier in this chapter and provides the conceptual foundation for the sparse dataflow abstractions used later in this dissertation.

Dataflow concepts also appear in programming languages, compiler IRs, and programming frameworks through examples like single-static assignment (SSA), functional dataflow graphs, and compiler ASTs and computation graphs (like PyTorch). These program-level and hardware execution manifestations of dataflow are complementary. Program-level dataflow captures dependencies between coarse operations, while accelerator-level dataflow implements (potentially finer-grained) operators physically in space with hardware. Both share the same central notion that computation proceeds when data is ready, which allows program dataflow graphs to map cleanly to dataflow hardware. Dataflow nodes become spatial primitives, edges become streams, and dependencies can be realized through ready–valid or other constraints. This structural synergy is essential to the systems developed in later chapters.

2.3.2 Prior Work on Fixed-Function Hardware

Table 2.1: Comparison of prior sparse fixed-function accelerators sorted by year. Blue bold tensors indicate operands that must be sparse; black bold tensors indicate operands that may be optionally compressed or dense. Tuples in the dataflow either represent a fusion of indices or a hybrid dataflow strategy.

Accelerator	Year	Computation Expression			Dataflow
EIE [78]	2016	x_i	$= \sum_j \mathbf{B}_{ij}\, \mathbf{c}_j$	SpMSpV	$j \to i$
Eyeriss v2 [35]	2018	x_i	$= \sum_j \mathbf{B}_{ij}\, \mathbf{c}_j$	SpMSpV	$j \to i$
SIGMA [169]	2018	X_{ij}	$= \sum_k \mathbf{B}_{ik}\, \mathbf{C}_{kj}$	SpM*SpM	$i \to j \to k$
OuterSPACE [157]	2018	$\mathbf{X}_{ij}$	$= \sum_k \mathbf{B}_{ik}\, \mathbf{C}_{kj}$	SpGEMM	$k \to (i, j)$
Cambricon-S [257]	2018	$\mathbf{X}_{ij}$	$= \sum_k \mathbf{B}_{ik}\, \mathbf{C}_{kj}$	SpM*SpM and	$i \to j \to k$
SparTen [68]	2019	$\mathbf{X}_{ij}$	$= \sum_k \mathbf{B}_{ik}\, \mathbf{C}_{kj}$	SpGEMM	$i \to j \to k$
SpArch [252]	2020	$\mathbf{X}_{ij}$	$= \sum_k \mathbf{B}_{ik}\, \mathbf{C}_{kj}$	SpGEMM	$k \to i \to j$
MatRaptor [201]	2020	$\mathbf{X}_{ij}$	$= \sum_k \mathbf{B}_{ik}\, \mathbf{C}_{kj}$	SpGEMM	$i \to k \to j$
Sparse-TPU [80]	2020	x_i	$= \sum_k \mathbf{B}_{ij}\, c_j$	(Sp)MV	$i \to j \to k$
GAMMA [246]	2021	X_{ij}	$= \sum_k \mathbf{B}_{ik}\, C_{kj}$	SpMSpM	$i \to k \to j$

There is a large body of recent work on architectures that explore point solutions in the space of hardened sparse tensor algebra expressions and algorithms [202, 159, 78, 157, 201, 80, 35, 169, 246, 7, 248], which we will call fixed-function accelerators. A list of these accelerators, along with the sparse tensor expression they compute and their dataflow (index order), is shown in Table 2.1. Many of these are convolution engines that implement sparse matrix vector multiplication [78, 35, 80, 248, 159, 7] or implement sparse matrix multiplication (SpM*SpM) [80, 169, 157, 201, 246]. For example, SIGMA [169] implements the inner-product $i \to j \to k$ iteration order. Although this order is typically preferred for dense matrix multiplications, and although it avoids scattering into the result, it has poor asymptotic performance because it iterates over all combinations of i and j before coordinates are intersected at the contracted variable k. GAMMA [246] applies Gustavson's [75] $i \to k \to j$ iteration order, which improves the asymptotic complexity at the cost of merge hardware to rearrange the reduction step to allow in-order generation of elements of X. And OuterSPACE [157] implements the outer-product $k \to i \to j$ iteration order, which for doubly-compressed sparse row (DCSR) matrices has better asymptotic complexity than the $i \to k \to j$ order but requires an additional step for merging whole matrices into X.

Using fixed-function matrix multiplication hardware to compute any expression in tensor algebra relies on factorizing general expressions into a sequence of invocations of matrix multiplication operations as presented by the approaches in Section 2.2.2. Factorizing unfuses the computation and fixes the dataflow ordering. The ideas presented in Chapter 7 allow for more general factorized code, and these fixed-function accelerator kernels can be plugged into the Mosaic system. And the other systems in this dissertation (Chapters 3, 4 and 6) remove the limitation of factorization by having sufficient power to express fused expressions on accelerators, letting us explore both the benefits and the costs of both approaches.

The number of sparse tensor algebra expressions is countably infinite. And for each expression, there is a design space of algorithms with different iteration orders, data structures, fusion, and tiling schemes that have drastically different memory and performance characteristics. In fact, two different iteration orders in a sparse tensor algebra expression are likely to have different asymptotic complexity as we will show in Chapter 4, and fusing a computation with three or more operands can also lower the asymptotic complexity. Moreover, one expression can even have two algorithms with different asymptotic complexity where neither dominates the other [4] and the best performing schedule depends on which operand is more sparse. Therefore, there is great value in general hardware that supports any expression as well as the design space of algorithms for each expression as we will see next.

2.3.3 Prior Work on Programmable Hardware

Table 2.2: Comparison of prior programmable sparse accelerators that can compute multiple tensor algebra expressions by year.

Accelerator	Year	Programming Model	Architecture	Implementation
SPU [45]	2019	Subset of C with intrinsics [223]	CGRA	RTL
Extensor [81]	2019	Hand-stitched Configurations	CGRA	RTL+Cycle Sim
Tensaurus [202]	2020	Custom Instructions	Configurable Vector-Systolic	RTL+Cycle Sim
Capstan [176]	2021	Extended Spatial DSL [111]	Vector RDA	RTL+Cycle Sim

The limitations of fixed-function libraries and hardware motivate programmable sparse tensor algebra dataflow hardware, which are implementations of an abstract machine like the one proposed in Chapter 4 of this dissertation. Tables 2.2 and 2.3 combine to describe four programmable accelerators that can compute many sparse tensor algebra expressions. We focus on three major lines of prior work on programmable sparse tensor algebra dataflow hardware due to their reconfigurability: the SPU [45], ExTensor [81], and Capstan [176]. While these designs do not support the full generality

Table 2.3: A more detailed comparison of the same prior sparse programmable accelerators as Table 2.2. Computation domains are determined by the evaluated workloads in the original publications and categorized by tensor algebra expressions (TA), graph computation (Graphs), relational queries (Rel.), and other domains. The features of the accelerator are also categorized by supported dataflows, Higher-order tensors (N-dim), and multiple sparse tensor operands (> 1-sp.).

Accelerator	**Computation Domains**				**Acceleration Pattern**	**Features**		
	TA	**Graphs**	**Rel.**	**Other**		**Dataflow**	**N-dim**	**>1-sp.**
SPU [45]	✔	✔	✔	Agnostic	Stream-joins, Indirection, Regular	All	✘	✔
Extensor [81]	✔	✘	✘	✘	Intersections	I	✔	✔
Tensaurus [202]	✔	✘	✘	✘	(Sparse-)Dense Tensor Contractions	I,R	✔	✘
Capstan [176]	✔	✔	✘	Solver	Vectorized Intersections and Unions	All	✔	✔

described in Section 2.1, they suggest the essential hardware structures and techniques in dataflow accelerators for sparse tensor algebra and greatly influenced the work in this dissertation.

SPU

The Sparse Processing Unit (SPU) [153, 45] is a spatial streaming dataflow architecture where instructions on a CPU configure a Coarse-Grained Reconfigurable Architecture (CGRA) and then stream arrays to it. It has efficient hardware both for combining streams (e.g., an intersection) and for using one stream to index into an on-chip array [45]. The SPU can be used to implement binary vector operations, relational joins, and graph algorithms [44], thus unifying domains. The SPU also includes an idiom-directed compiler [223] from pragma-annotated C loops to CGRA configurations.

Although the SPU literature describes operations that support a broad set of domains—tensor algebra, relational algebra, and graph operations—the SPU CGRA [45] only supports vector operations, while higher-order tensor algebra operations appear to be implemented as CPU loops that dispatch inner-loop vector operations to the CGRA (see Figure 6 from Nowatzki et al. [153], which the SPU extends). Thus, higher-order expressions must be broken into pieces with data flowing between the CPU and CGRA engine, reducing pipeline locality. The abstract machine we describe in Chapter 4 provides a complete dataflow model for tensor algebra. Using it, together with the compiler ideas proposed in this dissertation, to target the SPU and the TACO compiler [110] to target the CPU, provides a fruitful path towards compiling to the SPU system from a higher-level tensor algebra language.

ExTensor

The ExTensor system [81] is a CGRA-style architecture designed to hierarchically evaluate Einsum operations on sparse tensors. In ExTensor, a compute element is made up of memory that stores pieces of tensors, i.e, fibers, and hardware to perform operations on those fibers (e.g., read/stream them in/out, perform intersections/MACCs). ExTensor's design primarily considered a topology of compute elements that each operated on two fibers (from two operand tensors) at a time, and was focused on computing SpM*SpM. This topology also hardened the dataflow within the intersection units to inner-product only. To support SDDMM, which has three operands, the authors proposed an ExTensor instance with additional compute elements. However, it cannot perform all Einsum computations, such as those with union merges for addition. Extensor also did not provide a programmatic approach to map an arbitrary Einsum to a concrete ExTensor instance, which makes it hard to add new expressions that were not tested by the authors. Nor is there a discussion of how the architecture might change to better suit the needs of arbitrary Einsums. The techniques and systems presented in this dissertation address these limitations and should allow for more ExTensor configurations.

> "The complete ExTensor accelerator is configured to run a specific tensor algebra kernel [...] The accelerator is not Turing complete, but is designed to compute all algebra equations accepted by the TACO Tensor Algebra Compiler. Generating configurations directly from TACO is future work."
>
> —*Hegde et al. [81]*

Capstan

Finally, the Capstan system [176] supports hierarchical iteration over dense, compressed, bitvector, and bit-tree data structures. For some data structure types, it supports any tensor algebra expression, including additions and multiplications. Its primary limitation is that it does not support combining two or more compressed data structures (e.g., by intersection) at one iteration level. Instead, it relies on bitvectors and bit-trees, which densify the iteration. Rucker et al. [176] argue that this works well for common clustered sparse tensors, but not for arbitrary sparsity. Finally, the Capstan system does not support efficient broadcasting of a tensor over a sparse inner dimension of another tensor, as it relies on programmed counters to control broadcasting. While Capstan is programmed by Spatial [111, 251], a domain specific language for hardware accelerators based on parallel patterns [164], Spatial is at a lower-level of abstraction—more descriptive of the hardware accelerator—than the Custard input APIs. An example Spatial program for SDDMM can be found at Figure 6.4. This dissertation provides multiple opportunities for a higher-level programming interface for Capstan through SAM (Chapter 4), Stardust (Chapter 6), and Mosaic targeting kernels generated by Stardust (Chapter 7).

> "Capstan supports all sparse-iteration tensor applications with a low-level map-reduce programming model based on Spatial [41] and a clear path to highlevel programming (e.g., via TACO [40])."
>
> —*Rucker et al. [176]*

2.4 Recent Developments on Sparse Tensor Programming Systems and Hardware

Since the start of the work in this dissertation, there have been many developments in sparse programming systems and sparse hardware. I present these sparse programming systems, along the lines of extending sparse tensor algebra compilation to broader classes of computations, formats, backends, and more:

Factorization. Newer sparse tensor contraction frameworks [130] for optimizing sparse tensor algebra and subsequent analysis of sparse tensor contraction memory and computation behavior has led to optimized 2D tiling algorithms for multicore CPUs [172].

Generalizing to Other Domains. Generalization beyond classical sparse tensor algebra languages that include extensions to the Einsum language to add richer expressiveness and new operators [156]. Generalizations that encompass both language and compilation techniques to sparse shape programs [175]; machine-learning (sub-)computations on CPUs [238, 239], GPUs [74], and dataflow accelerators [122, 188]; affine-indexed sparse convolutions [132, 229]; tiled recurrences for applications like sparse direct solvers and dynamic programming [207, 209]; and relational query compilation [121, 56].

Code lifting. New techniques that lift general-purpose code (like C/C++) to sparse tensor DSLs using enumerative synthesizers [135] and E-graphs [123].

Formal verification. Verified lowering and index-variable reasoning for sparse code generation from tensor index notation [121].

Tensor compression formats. New ways to emit code for compressed data structures with dynamic updates [39], compressed data [5, 6], extend compression beyond zeros [55], and support continuous arrays [228]. In addition, compilation for tensors represented using combinations of many tensor data structures [242, 47], especially useful in targeting sparse tensor cores on GPUs, and leveraging Einsum specialization to compressed data structures with indirect indexing [227]. There has also been work on a unified IR and compiler for customizing sparse formats [131] for hardware beyond the sparse format language used by TACO and in this dissertation.

Code generation. Work has introduced new constraint-based IR to describe the space of possible fused loops on one compressed tensor format and developed a code generator from that IR to TACO [173]. Other sparse tensor algebra compilers create different code generation strategies that use a simple iterator model [175, 121], eliminate redundancy for compound (cascades of) expressions [256], support expressions with sparse temporary tensors [244], and leverage the MLIR framework [23].

(Auto)scheduling and autoformatting. Follow-on work to TACO has introduced new scheduling commands and primitives for kernel fusion and distribution [54, 122]. Beyond manual user scheduling and optimizations, recent work in sparse auto-scheduling include SMT-based sparse autoscheduling [53] that extends the scheduling work of Dias et al. [54], asymptotic sparse cost modeling [4], applying modern query optimization techniques to sparse tensor compilers [52], lightweight autoscheduling for sparse machine learning workloads [238, 239], autotuning sparse tensor compilers with Bayesian optimization [82], auto-format selection [230, 238], and leveraging sparse compiler code generation to automatically optimize the workload on other hardware [205].

Backends. Since the initial proposal for sparse tensor compilation to GPU backends, improvements have been made to GPU kernel generation by utilizing novel tensor compression techniques [242, 227]. Beyond targeting accelerators, which I present in this dissertation, sparse tensor algebra compilation now maps to distributed machines through the Legion runtime system [232, 233] and MLIR [214], integrates with higher-level languages and runtimes like Python and Legate [235].

General-Purpose to Accelerator Programming Systems. There is another line of compilers that target dataflow accelerators capable of sparse tensor algebra expressions from general-purpose code [223, 66].

Performance modeling. Compilers and frameworks with heuristic analysis of sparse tensor workloads [172, 122, 52]

On the hardware side, the landscape of sparse accelerators also expanded rapidly with new sparse performance modeling, fixed accelerator designs, and dataflow fabrics:

Hardware performance modeling. Advances in analytical performance modeling of hardware for sparse tensor workloads [231] and further developments on a unified framework for modeling sparse tensor algebra accelerator designs [150] that pushes the separation of algorithm, schedule [170, 171], and format [41] into a clearer set of independent concerns.

Sparse hardware generation. Influenced by the input languages of sparse tensor compilers and sparse accelerator design, there has been recent work on tensor algebra frameworks for fixed-function hardware generation [63].

Fixed-function sparse hardware. Novel accelerator designs that harden specific sparse kernels with increasingly complex dataflows, like adaptive dataflow selection in hardware to match the input data distributions at runtime [128, 147], ML-assisted dataflow selection [237], and specialization for both dense and single-sparse workloads [240]. Beyond single-expression accelerators, there has been ongoing work on accelerators that can handle multiple sparse expressions that are not countably infinite, for example, a tightly-coupled accelerator with a tile-based ISA design for two sparse expressions [64].

FPGA-based acceleration. There has been a line of sparse matrix-vector accelerators that use HBM-based FPGA sparse accelerator designs [196, 197, 15]. Although FPGAs are programmable, these designs are reconfigured for one expression, making them fixed-function.

Programmable Hardware. Several works introduce general reconfigurable dataflow architectures that support multiple workloads, which include sparse tensor algebra expressions [66, 163]. From there, there are various reconfigurable architectures similar to the SPU [45] that focus on generalized irregular paradigms that can target many application domains, including sparse tensor algebra [151, 10, 67]. Finally, there have been more proposed reconfigurable [255], GPU-like [161], and decoupled access–execute CPU [189] architectures for sparse tensor algebra.

These developments collectively expand the ecosystem of sparse programming systems and sparse hardware, reinforcing the need for unified abstractions and end-to-end compilation approaches such as those developed in this dissertation.

Chapter 3

Generalized Sparse Tensor Programming

"The emergence of a new theory is rarely just an incremental step forward; it is a reconstruction of the field from new fundamentals."

Albert Einstein

Sparse tensor algebra as presented in Section 2.1 is important, but it does not encompass general iteration of compressed tensor data. Traditional tensor algebra ties iteration to the arithmetic structure of addition and multiplication only, which implicitly restricts compressed tensor iteration to a small set of patterns (e.g., unions and intersections of nonzeros) and assumes fixed notions of sparsity. These assumptions fundamentally limit the computational and traversal expressiveness of sparse tensor algebra programming systems. In contrast, this chapter introduces a general iteration system that supports arbitrary value compressions and makes all set operations explicit on sparse iteration spaces. This generality completes the expressiveness for compressed tensor iteration and computation by developing theory for any possible sparse iteration space and tensor operator. This set-centric generalized view of sparse tensor iteration, which I call generalized tensor programming, is the heart of this chapter and lays the foundation for the programming abstractions used later in this dissertation. By decoupling iteration from arithmetic properties, the system represents how a computation traverses sparse data independently of the particular operators used, which is exactly the structure needed to drive the dataflow-style joins over streams that appear in sparse accelerators.

Therefore, this chapter presents the first key idea to enable the programming of dataflow accelerators using high-level sparse DSLs: *an explicit set abstraction to join multiple tensors.* Fundamentally, many of the general-purpose sparse compilers and abstractions presented in Chapter 2, specifically in Section 2.2, assume an imperative model of execution. The imperative model assumes that computation that iterates over tensors must be done using arbitrary loops and control logic (i.e. for or while-loops and if-statements). However, as mentioned in Section 2.3, most of the sparse compilers use a streaming dataflow execution model. This execution model means that iterating through tensors becomes a scan of the tensor data in memory to produce streams of data, and combining multiple tensor streams to iterate through the computation becomes a join or merge operation. In order to represent sparse tensor operations as joins, a sparse compiler must have a notion of join operations and reason about them. The set algebra presented in this chapter is that notion.

The concrete programming system implementation in this chapter presents a compiler for generalized sparse tensor programming.[1] However, this compiler still uses the explicit set algebra to generate general-purpose CPU code. Since programming systems for sparse accelerators is the focus of this dissertation, the explanation of this compiler will not include the code generation process. Full details of the process can be found at Henry et al. [83].

Beyond compiling to sparse accelerators, the compiler for generalized sparse tensor programming has been influential in the development of sparse compilers for other domains. The ideas in this chapter and the full system were essential in extending sparse tensor algebra compilation to relational algebra [56], with sparse reshape operators [175] which later led to sparse machine learning [122, 238], and to sparse recurrence relations which includes computations like sparse linear solvers and dynamic programming problems [207]. The idea of generalizing sparse tensors to with arbitrary fill values was also instrumental in a new line of compiler research related to more generalized fill values, which include compilers for lossless compression data structures [55], generalized tensor structures [6, 5], and continuous arrays [228].

The work in this chapter also motivated industrial adoption through the Python PyData/Sparse library. The bespoke code generation in this work, and its significant speeedups over factorized runtime code [2], led to this push.

3.1 Background on Sparse Tensor Algebra Languages

Although Chapter 2 surveys varying approaches that lead to high-performance sparse code, we limit our discussion in this section to sparse compilers as it is most relevant to this chapter, and the dissertation as a whole. This section provides the necessary details to understand the body of research that led to the TACO compiler [108, 106, 41, 181] before this dissertation. This section further focuses on the input APIs to the TACO compiler, three high-level DSLs that combine to describe a sparse tensor algebra expression [108, 181]. It is important to understand these input languages since many of the programming systems in this dissertation either build upon (Chapters 3

[1]In the original publication, we called the programming model "sparse array programming" to refer to sparse arrays similar to NumPy arrays [79, 222]. However, there is an overloading of the term array between abstract arrays and in-memory data structure arrays. Thus, I rename sparse abstract array programming to generalized tensor programming when referring to the work in this dissertation. However, I make a conscious effort to use the term array programming when referring to historical array languages, libraries, and frameworks [129, 30, 222, 79, 98, 2]

```
 1 // Format language: comprised of level formats and mode orderings
 2 // Below are the descriptions for CSR and dense column-major data structures
 3 Format csr({dense, sparse}, {0, 1});
 4 Format dense_cm({dense, dense}, {1, 0});
 5
 6 // Declare input and output tensors
 7 Tensor<int> A({I,J}, csr);
 8 Tensor<int> B({I,K}, csr);
 9 Tensor<int> C({K,J}, dense_cm);
10
11 // Define the SpMSpM expression in tensor index notation (algorithm)
12 IndexVar i, j, k;
13 A(i, j) = B(i, k) * C(k, j);
14
15 // Scheduling language: Inner-product dataflow (reorder),
16 IndexStmt stmt = A.getAssignment();
17 stmt = stmt.reorder({i, j, k})
```

Figure 3.1: Example SpMM code using the high-level sparse input languages of the TACO compiler.

```
15 // Scheduling language: Row-product dataflow (reorder),
16 // where the j-loop is vectorized by 8 (split, parallelize)
17 stmt = stmt.reorder({i, k, j})
18            .split(j, jo, ji, 8)
19            .parallelize(ji, CPUVector, NoRaces)
```

Figure 3.2: Another SpMM example with a different schedule, where lines 1–14 are the same as Figure 3.2.

and 7) or directly use these DSLs as their input API (Section 4.7 and chapters 5 and 6).

The TACO compiler separates the algorithm (tensor index notation) [108] from the tensor compression formats and computation transformations through the use of format [41] and scheduling [181] languages, respectively. An example input program to the TACO compiler of these three languages, and how they interact, is shown in Figure 3.2. TACO compiles sparse tensor algebra to imperative code by decomposing sparse iteration spaces into hierarchical combinations of per-dimension data structures. Sparse algorithms are expressed in concrete index notation (CIN), which encodes iteration, computation, transformations, and temporary tensors [106, 244].[2] Finally, TACO lowers CIN to generate efficient fused code that traverses irregular data structures by skipping unnecessary computation.

Algorithm Language The algorithmic language of our compiler is tensor index notation (or Einsum notation [57]) as previously introduced in Section 2.1. To illustrate tensor index notation further, the element-wise addition of two three-dimensional tensors, for instance, can be expressed as $A_{ijk} = B_{ijk} + C_{ijk}$, which specifies that each component of the result 3-order tensor A should be computed as the sum of its corresponding components in the input 3-order tensors B and C. Tensor index notation can also express computations that reduce over components of operand tensors along

[2]Although not included in this dissertation, I also helped develop techniques for compiling code with sparse temporary tensors [244] during my PhD. Prior work at the time was only capable of generating code for dense temporary tensors [106]. This work furthered sparse iteration theory by compiling high-level sparse tensor algebra expressions with **sparse** temporary tensors to efficient CPU code.

Table 3.1: The precompute command from the scheduling language of the TACO work [106, 181]. $e[x'/x]$ denotes the expression e with each occurrence of x replaced by x'.

Scheduling Commands	**Description**
$\texttt{precompute}(e, i*, i_w*, \mathcal{T})$	Inserts a where statement to precompute a sub-expression e into a temporary tensor workspace $\mathcal{T}$ with new indices i_w* on the right-hand side of the newly introduced where node.
$\forall_{i*} A \xrightarrow{\text{precompute}(e,i*,i_w*,\mathcal{T})}$	$\forall_{i*} A[\mathcal{T}(i*)/e]$ where $\forall_{i_w*} \mathcal{T}(i_w*) = e[i_w*/i*]$

one or more dimensions (often also referred to as a tensor's rank or mode). For example, $y_i = \sum_j A_{ij}$ expresses a computation that defines each component of y to be the sum of all components in the corresponding row of A. Figure 3.2 line 13 gives an example of tensor index notation written in embedded C++ code. Tensor index notation is a subset of concrete index notation presented below.

Scheduling The sparse scheduling language proposed by Senanayake et al. [181] provides a sparse iteration transformation framework. The framework modifies the sparse iteration space of an expression by taking its CIN statement and transforming it into a new CIN statement that represents a different algorithm for the same expression. The scheduling transformation framework describes optimizations to change the computation order, insert temporary tensors for partial sub-computation, exploit parallelism, and more. Figure 3.1 and Figure 3.2 provide two example schedules for sparse-matrix dense-matrix multiplication. Table 3.1 provides one example of an original scheduling command in TACO.

Format Language The format language proposed by Chou et al. [41] decomposes a sparse tensor into per-dimension (or level) formats that each describes how to store the coordinates of one dimension of a tensor. As an example, the canonical compressed sparse row (CSR) compression format (see Figure 6.8 for an example matrix) can be represented by an uncompressed (dense) dimension followed by a compressed (sparse) dimension. After the tensors have been described using level formats and scheduling transformations have been applied to the CIN, TACO generates code that iterates over the level formats of the expression.

Concrete Index Notation (CIN) The concrete index notation abstraction formalized by Kjolstad et al. [106] and later extended by Senanayake et al. [181] is an abstract loop-based IR that extends tensor index notation. CIN explicitly denotes the `forall` loops over index variables, contains temporary tensor information using `where` statements, and encodes scheduling relationships between index variables and tensors using scheduling relations. The abstraction maintains abstract tensors but contains references to the compressed data structure information provided by the format language. The CIN abstraction, in essence, unifies the algorithm, schedule, and format of a given user input into a single sparse tensor algebra expression. Figure 3.3 gives the full grammar for concrete index notation.

$$
\begin{array}{rcll}
\multicolumn{4}{c}{\textit{Index Variable}\ \ i\quad \textit{Index Variable List}\ \ i*\quad \textit{Constants}\ \ c\quad \textit{Tensors}\ \ \mathcal{T}} \\
\textit{Accesses} & a & ::= & \mathcal{T}_{i*} \\
\textit{Expressions} & e & ::= & a \mid c \mid e+e \mid e*e \mid \ldots \\
\textit{Assignment} & A & ::= & a = e \mid a \mathrel{+}= e \\
\textit{Statements} & S & ::= & \forall_{i*}\ S \ \mid\ A\ \mid \\
 & & & S\ ;\ S \mid S\ \textsf{where}\ S \mid S\ \textsf{s.t.}\ r* \\
\textit{Scheduling Relation} & r & ::= & \textsf{split}(i, i_o, i_i, c) \mid\ \textsf{fuse}(i_o, i_i, i_f) \mid \ldots
\end{array}
$$

Figure 3.3: Concrete index notation (CIN) syntax.

3.2 Generalizing to Sparse Tensor Programming

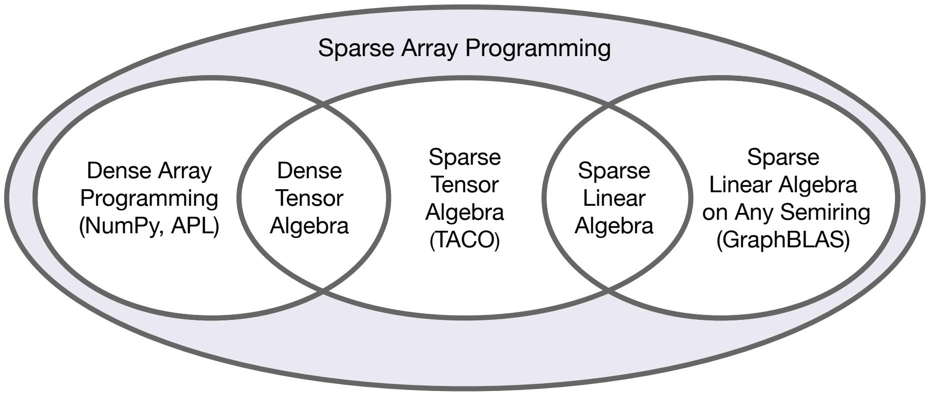

Sparse Array Programming
Dense Array Programming (NumPy, APL)
Dense Tensor Algebra
Sparse Tensor Algebra (TACO)
Sparse Linear Algebra
Sparse Linear Algebra on Any Semiring (GraphBLAS)

Figure 3.4: Generalized sparse tensor programming (i.e. sparse array programming) is a superset of other programming models.

Arrays are fundamental data structures that let us represent collections of numbers, tabular data, grids embedded in Euclidean space, tensors, and more. They naturally map to linear memory and it is unsurprising that they have been the central data structure in languages built for numerical computation since Fortran [11] and APL [96]. In fact, Python became prevalent in computational science, data analytics, and machine learning partially due to the introduction of the NumPy array programming library [222, 79].

An array programming model is a programming model whose expressions operate on arrays as a whole through element-wise operations, broadcasts, and reductions over dimensions. From APL [96] introduced in 1960 to NumPy [79] today, array programming languages have played a prominent role in our programs. For example, NumPy permits element-wise operations and reductions with any user-defined function, broadcasting, and slicing.

A generalized sparse tensor (referred to also as sparse array) is an array where many components have the same value, known as a *fill value* [98, 208, 222]. Generalized sparse tensors are becoming increasingly important as the need for numerical computation across large, sparsely populated systems increases in scientific computing, data analytics, and machine learning. They can be used to model sparse matrices and tensors [220], sparse grids [88], and even graphs [139]. For example, Generalized sparse tensors can represent the number of friends shared by every pair of people (the sparsity arises because most people share no friends), the set of nodes to exclude in each step of breadth-first search (Section 3.7.3), or black-and-white MRI images (Section 3.7.4).

Therefore, there is a need for a generalized sparse tensor programming model as a counterpart to—and generalization of—dense array programming models. In fact, at the time of writing, the roadmap [180] of the ubiquitous SciPy library [220] calls directly for a sparse NumPy as one of five goals. The PyData/Sparse project has responded with an implementation [2], but it relies on data transformation to implement the significant generality of sparse tensor programming and therefore runs significantly slower than what is possible.

Table 3.2: Features in a generalized sparse tensor programming model compared to those in related programming models.

Paradigm	Supported Functions			Data Representation				Slicing
	$(+, \times)$	Any semiring	Any	Dense	Sparse		Any # of dims.	
		$(\wedge, \vee), \ldots$	`foo`, ...		Zero fill	Any fill		
Dense Array Programming (NumPy)	✔	✔	✔	✔	✘	✘	✔	✔
Dense Tensor Algebra	✔	✘	✘	✔	✘	✘	✔	✔
Sparse Tensor Algebra (TACO)	✔	✘	✘	✔	✔	✘	✔	✘
Sparse Linear Algebra	✔	✘	✘	✔	✔	✘	✘	✘
Sparse LA on Any Semiring (GraphBLAS)	✔	✔	✘	✔	✔	✘	✘	✔
Generalized Sparse-Tensor Programming (This Work)	✔	✔	✔	✔	✔	✔	✔	✔

In this chapter, we present the first generalized sparse tensor programming model compiler that can fuse and compile any expression involving sparse and dense arrays with arbitrary (implicit) fill values, where the operators and reductions can be any function. The array expression $A_{ij} = (B_{ij}^{C_{ij}}) * \neg D_{ij}$ is an example of a computation that cannot be expressed in sparse tensor algebra (since it uses operations that are not additions or multiplications) and that cannot be expressed in dense array programming (if the inputs B, C, and D are too large to store without compression). Table 3.2 and Figure 3.4 show how our proposed sparse programming model is a superset of the programming models of NumPy dense array programming, TACO sparse tensor algebra, and the GraphBLAS [139] graph algorithm library. In order to execute arbitrary functions, we generalize the compilation theory of Kjolstad et al. [108] to support any sparse iteration space. We have also extended the sparse iteration theory to support generating code to compute on sliced windows of data, which allows for operating on subsets of sparse tensors in place. In addition, we built an API for defining these functions and for declaring their properties. Our technical contributions are:

1. A generalization of sparse iteration space theory to include any sparse iteration space, instead of only those that can be described by intersections and unions.

2. Code generation to support any sparse iteration space for arbitrary user-defined functions.

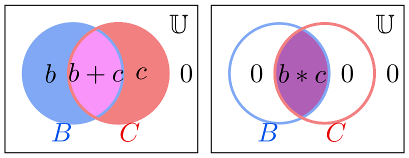
3. Derivation of sparse iteration spaces from functions decorated with mathematical properties.
4. Extension of sparse tensors to allow any fill value (not just 0) for compressed entries.
5. Generalization of iteration spaces to allow iteration over sub-array slices of sparse tensors.

We evaluate these contributions by comparing against implementations of sparse array primitives in popular and state-of-the-art generalized sparse array programming libraries like SciPy and PyData/Sparse, as well as in larger applications like image processing and graph processing. Our evaluation shows a normalized speedup of 0.98× to 5.63× compared to SciPy/Sparse for sub-array slicing and between 1.4× and 43.4× compared to PyData/Sparse for universal functions. Furthermore, we demonstrate our technique's ability to fuse computation with a performance improvement of 12.7× to 43.4× for fused universal functions when measured against PyData/Sparse. In the context of graph kernels, our system performs between 0.56× and 3.50× that of a hand-optimized application-specific baseline system, SuiteSparse:GraphBLAS. For practical array algorithms, we outperform PyData/Sparse by between 6.4× to 70.3×, and the relative performance of NumPy compared to our system is between 0.96× to 28.93× when a dense implementation is feasible.

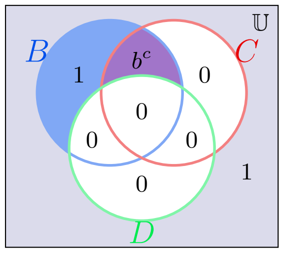
3.3 Example General Sparse Tensor Programs

Array programming is a fundamental computation model that supports a wide variety of features, including array slicing and arbitrary element-wise, reduction, and broadcasting operators. However, current dense array implementations cannot store and process the increasingly large and sparse data emerging from applications like machine learning, graph analytics, and scientific computing. Sparse tensor algebra, on the other hand, is a powerful tool that allows for multilinear computation on tensors—higher-order matrices and vectors. Multi-dimensional arrays can be represented as tensors (and vice versa), which means that sparse tensor algebra allows for computation on sparse arrays, but there are limitations to the existing sparse tensor algebra model.

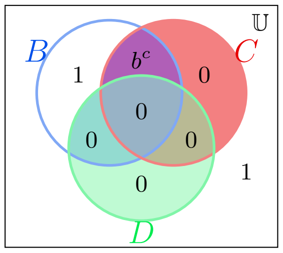
Tensor algebra computation and reductions are only defined across additions and multiplications. Element-wise addition $A = B + C$ takes the union of non-zero input values and element-wise multiplication $A = B * C$ takes the intersection, as illustrated in Figure 3.5a. However, there are situations where the user would want to perform more general computation. One example is $A_{ij} = (B_{ij}^{C_{ij}}) * \neg D_{ij}$, which raises B to the power of C (`power`) and filters the result by the logical inverse of D. Arbitrary functions like `power` are not expressible using sparse tensor algebra since they cannot be defined by combining the intersection (multiplication) or union (addition) of non-zero input values, as shown in Figure 3.5. Sparse tensor algebra also limits the definition of sparsity to having a significant number of zeros that can be compressed away (see Figure 3.5b). Our `power` example motivates the need to compress out other values instead—namely 1 since $b^0 * 1 = 1$ (see Figure 3.5c). Furthermore, the $A_{ij} = (B_{ij}^{C_{ij}}) * \neg D_{ij}$ example is motivated by applications like medical image



(a) Add (union) and multiply (intersection) computation space with 0 compression



(b) Masked `power` with 0 compression of the result A



(c) Masked `power` with 1 compression of the result A

Figure 3.5: Computation spaces of traditional tensor algebra operators (a) versus arbitrary function computation for the masked power example: $A_{ij} = (B_{ij}^{C_{ij}}) * \neg D_{ij}$ with 0-value (b) and 1-value (c) compression of A. Color-filled regions require the computation denoted with black text, and white-filled regions are ignored.

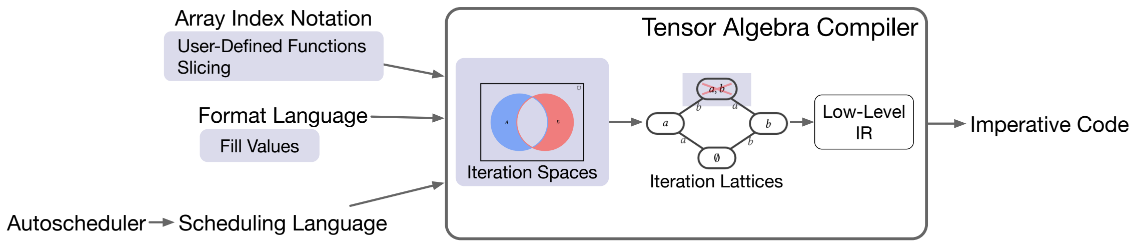


Figure 3.6: Overview of the generalized sparse tensor compiler system. Array Index Notation is called Generalized Tensor Index Notation in this dissertation. Gray components are new contributions.

processing and graph algorithms, which often perform computations that apply filters and masks (like the $*\neg D_{ij}$ sub-expression). Generalizing tensor algebra to any function requires formalizing the function's properties and computational behavior. Finally, tensor algebra expressions are also restricted to computation on entire tensors, even though it can be useful to extract and compute on sub-arrays. These limitations motivate us to generalize concepts from sparse tensor algebra and dense array programming to propose a generalized sparse tensor programming model and a compilation-based system that realizes it.

3.4 Generalized Sparse Tensor Compiler Overview

We implemented the generalized sparse tensor programming model and general sparse tensor compilation as extensions to the open-source sparse tensor algebra compiler framework TACO [108], as depicted in Figure 3.6. Our extension is open-source and publicly available at `https://github.com/tensor-compiler/taco/tree/array_algebra`. Like the TACO compiler, our generalized sparse tensor compiler takes an algorithm description, a format language [41], and a scheduling language [181].

Unlike the TACO compiler, which compiles a tensor algebra language [108], the input algorithm description for our generalized sparse tensor compiler is a generalized sparse tensor programming model, further described in Section 3.5. The programming model supports applying any functions across generalized sparse tensors through a new language we call *generalized tensor index notation* (see Section 3.5.2) and compressing out any value from the generalized sparse tensors through an extended format language (see Section 3.5.1). Generalized tensor index notation uses sparse tensors to represent sparse arrays and allows the description of any universal function along with its mathematical properties, which is detailed in Section 3.5.3. Additionally, computations in generalized tensor index notation can be performed on sparse sub-arrays using sparse tensor slicing and striding, as also detailed in Section 3.5.2. The combination of sparse tensor representations and their fill values, generalized tensor index notation, sparse tensor slicing, and user-defined functions forms the generalized sparse tensor programming model. Figure 3.7 and Figure 3.8 show how programmers can express complex computations using this programming model[3].

Arbitrary user-defined functions are specified by a description of the function's computation and iteration pattern. The iteration pattern describes how the compiler should iterate through values of the input array space, defined directly through a set algebra composed of intersections, unions, and complements of sparse tensor coordinates. Instead of providing an explicit iteration pattern, users may provide mathematical properties of the function which the generalized sparse tensor compiler uses, along with fill values of the input tensors, to automatically derive an iteration pattern (see Section 3.5.3). We describe these generalized iteration spaces and property derivations for generalized functions in Section 3.6. These generalized iteration spaces explicitly formulate the set operations needed for sparse hardware accelerators to efficiently execute their stream-join dataflow execution model—which is described in Chapter 4 and Chapter 6.

The generalized sparse tensor compiler in this chapter, however, uses the descriptions of generalized iteration spaces to generate low-level C-code for CPUs. Since the code generation path for CPUs is independent from that of sparse accelerators, we do not include detailed descriptions of the process in this dissertation. We provide an overview of the CPU code generation process for our compiler below, and detailed descriptions can be found at Henry et al. [83]. The generalized sparse tensor compiler uses the descriptions of generalized iteration spaces to create an extension of the iteration lattice intermediate representation (IR) described by Kjølstad [110] to simplify loop and case-statement generation for an input sparse tensor computation. The compiler includes necessary generalizations to the iteration lattice IR to represent iteration over any iteration space, not just those described by intersection and union expressions. The generalized sparse tensor compiler uses the generalized iteration lattice to generate low-level code that performs iteration over any iteration space. Henry et al. [83] describes how to lower an iteration lattice into low-level CPU code as well as how to generate code that operates on slices of tensors. Figure 3.9 shows an example of the final optimized CPU code that the generalized sparse tensor compiler can generate using these techniques.

Finally, in Section 3.7 we not only evaluate our compiler against an existing sparse array library that provides as much generality as our system, but also against special purpose libraries that

[3]Example code using the PyData/Sparse API can be found in Appendix A.2 in the supplemental materials[4].

```
// Define a dense vector format
// and a sparse vector format
// with fill values of 0.
Format dv({dense}, 0);
Format sv({compressed}, 0);

// Declare inputs to be sparse
// vectors and declare output
// to be a dense vector.
Tensor<int> a(N, dv);
Tensor<int> b(N, sv);
Tensor<int> c(N, sv);

// Define computation that computes
// element-wise GCD of two vectors.
// GCD is defined in Figure 3.8
IndexVar i;
a(i) = gcd(b(i), c(i));

// Perform computation by generating
// and executing code in Figure 3.9.
std::cout << a << std::endl;
```

Figure 3.7: C++ code that uses our generalized sparse tensor compiler to compute the element-wise greatest common divisor (GCD) of two sparse vectors.

```
def gcd(x,y):
  x,0 => { return abs(x); }
  0,y => { return abs(y); }
  x,y => {
    x = abs(x);
    y = abs(y);
    while (x != 0) {
      int t = x;
      x = y % x;
      y = t;
    }
    return y;
  }
  iteration_space:
    {x ≠ 0} ∪ {y ≠ 0}
```

Figure 3.8: A function that implements the GCD operation. It contains optimized implementations for the cases where `x` or `y` is 0, and the iteration space is explicitly defined using iteration algebra.

```
int pb = b_pos[0];
int pc = c_pos[0];
while (pb < b_pos[1] &&
       pc < c_pos[1]) {
  int ib = b_crd[pb];
  int ic = c_crd[pc];
  int i = min(ib, ic);
  if (ib == i && ic == i) {
    int x = b_vals[pb];
    int y = c_vals[pc];
    x = abs(x);
    y = abs(y);
    while (x != 0) {
      int t = x;
      x = y % x;
      y = t;
    }
    a_vals[i] = y;
  } else if (ib == i) {
    int x = b_vals[pb];
    a_vals[i] = abs(x);
  } else {
    int y = c_vals[pc];
    a_vals[i] = abs(y);
  }
  pb += (ib == i);
  pc += (ic == i);
}
while (pb < b_pos[1]) {
  int x = b_vals[pb];
  a_vals[i] = abs(x);
  pb++;
}
while (pc < c_pos[1]) {
  int y = c_vals[pc];
  a_vals[i] = abs(y);
  pc++;
}
```

Figure 3.9: Code that our technique generates to compute $a_i = \gcd(b_i, c_i)$, assuming b and c are one-dimensional sparse tensors with zeros compressed out.

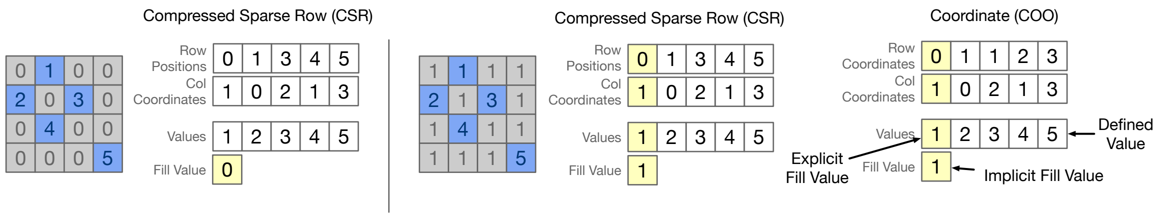
hand-code implementations of specific generalized sparse tensor programs.

3.5 Generalized Sparse Tensor Programming Model

In this section, we describe the features of a general sparse tensor programming model through a programming language we call *generalized tensor index notation* that supports complex computations on general sparse tensors. Generalized tensor index notation generalizes the conventional tensor index notation by relaxing the definition of sparse tensors and supporting a wider range of operations on sparse tensors.

3.5.1 Generalized Sparse Tensors and Fill Values

Generalized tensor index notation operates on multi-dimensional arrays. A multi-dimensional array can be viewed as a map from sets of (integer) coordinates to their corresponding values, which may



(a) CSR matrix with a fill value of 0 (b) CSR and COO matrices with a fill value of 1

Figure 3.10: Examples of varying sparse tensor formats with different fill values.

be of any data type (e.g., floating-point values, integers, etc.).

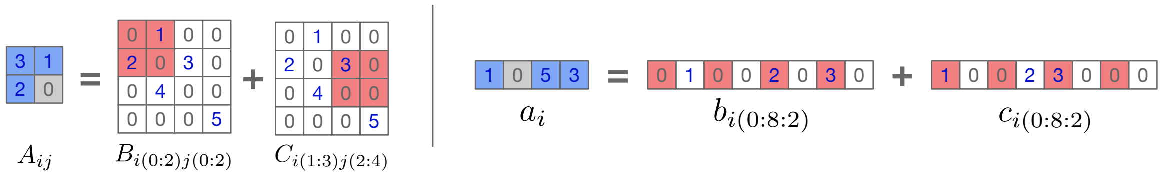
An array is sparse if many of its components have the same value, which we refer to as the array's *fill value*. For instance, an array that encodes distances between directly-connected points in a road network (with two points having a distance of ∞ if they are not directly connected by a road) is very likely sparse since most pairs of points in the network are not directly connected, meaning most components in the array would be ∞. This distance array can be said to have a fill value of ∞, while all other (i.e., non-infinite) values in the array are its *defined values*.

Generalized sparse tensors can be efficiently stored in memory using various data structures (formats) that omit all (or at least most) of the arrays' fill values. Figure 3.10 shows two examples of sparse two-dimensional array (i.e., matrix) formats. The coordinate list (COO) format stores the row/column coordinates and value of every defined value in the array, while the compressed sparse row (CSR) format additionally compresses out the row coordinates by using a positions array to track which defined values belong to each row. Chou et al. [41, 40] showed how a format language can precisely describe a wide range of sparse tensor formats in a way that lets compilers generate efficient code to compute using the arrays stored in those formats. However, this language assumes that sparse tensors always have a fill value of 0, which, as the distance array example shows, is not always true.

We generalize the data representation language to support arbitrary fill values (such as ∞ and 1) by requiring that the compressed value be specified as part of the sparse tensor format description. Figure 3.10b, for example, shows how both CSR and COO can be specified to have fill values of 1. Array components that are not explicitly stored are called *implicit fill values*, and components that are explicitly stored but also equal the fill value are called *explicit fill values*.

3.5.2 Generalized Tensor Index Notation

As with tensor index notation, computations on multi-dimensional tensors can be expressed in generalized tensor index notation by specifying how each component of the result array should be computed in terms of components of the input arrays. The full syntax of generalized tensor index notation can be found at Figure B.1 in the Appendix of this dissertation and in Appendix A.2 in



(a) Windowing example (b) Striding example

Figure 3.11: Generalized tensor index notation supports computations on slices of sparse tensors.

the supplemental materials of the original publication[4]. Generalized tensor index notation extends tensor index notation in two ways.

First, generalized tensor index notation allows programmers to define arbitrary functions (on top of addition and multiplication) and to use these functions in computations. So, for instance, a programmer can define a function `xor` that computes the exclusive or of three scalar inputs. The programmer may then use this function for element-wise computation with three-dimensional arrays, which can be expressed as $A_{ijk} = \text{xor}(B_{ijk}, C_{ijk}, D_{ijk})$. User-defined functions can also be used in reductions. For example, assuming `min` is a binary function that returns the smallest argument as output, the statement $y_i = \min_j A_{ij}$ expresses a computation that returns the minimum value in each row of a two-dimensional array. Section 3.5.3 describes how to define custom generalized tensor index notation functions.

Second, generalized tensor index notation allows users to slice and compute with subsets of sparse tensors. For instance, as Figure 3.11a shows, the statement $A_{ij} = B_{i(0:2)j(0:2)} + C_{i(1:3)j(2:4)}$ specifies a computation that extracts 2×2 sub-arrays from B and C and element-wise adds the sub-arrays, producing a 2×2 result array A. Generalized tensor index notation also supports strided accesses of sparse tensors. For instance, as Figure 3.11b shows, the statement $a_i = b_{i(0:8:2)} + c_{i(0:8:2)}$ specifies computation that extracts and element-wise adds the components with even-valued coordinates from b and c. (This slicing notation corresponds to the standard Python syntax `x[lo:hi:st]`, which accesses an array `x` from coordinate `lo` to non-inclusive coordinate `hi` with stride `st`.) Slicing operations in generalized tensor index notation can be viewed semantically as first extracting the sliced array into a new array where each dimension ranges from 0 to the size of the slice, and then using that new array in the rest of the computation. However, just as in dense array programming, slicing operations should be oblivious to the underlying data structures used and should not result in unnecessary data movement or reorganization. Slicing operations should instead adapt the implementation of the generalized tensor index notation statement to the desired slicing operation and format of the sparse tensor. Our technique to emit efficient CPU code to slice sparse tensors can be found at the full publication by Henry et al. [83]. We do not include our CPU code generation techniques in this dissertation as it is unrelated to targeting accelerator hardware.

[4]A link to the supplemental materials can be found here.

3.5.3 Generalized Functions

Programmers can define custom functions that can be used to express complex sparse tensor computations in generalized tensor index notation. Programmers specify the semantics of a custom function by providing an implementation that, given any (fixed) number of scalar inputs, computes a scalar result. Function implementations are written in a C-like intermediate language that provides standard arithmetic and logical operators, mathematical functions found in the C standard library, and imperative constructs such as `if`-statements and loops. Figs. 3.8 and 3.12 illustrate how users can specify the semantics of simpler functions like bitwise-and as well as more complex functions like the greatest common divisor (GCD) function, which is implemented using the Euclidean algorithm.

```
def bitwise_and(x,y):
  x,y => {
    return x & y;
  }
  properties:
    commutative
    annihilator=0
```

Figure 3.12: A function that implements the bitwise-and operation decorated with algebraic properties. If the fill values of `x` and `y` are 0, then the iteration space for this function will be an intersection.

A user may optionally specify, for each combination of fill value and defined value inputs, how the function can be more efficiently computed for that specific combination of inputs. For example, lines 2–3 in Figure 3.8 shows how a programmer can specify that, when either argument is zero, the `gcd` function simply has to return the value of the other argument. Using these additional specifications, our technique can generate code like in Figure 3.9, which computes the element-wise GCD of two input vectors without having to explicitly invoke the Euclidean algorithm whenever one input is guaranteed to be zero (see lines 19–25 and 29–38).

To support efficient computing on sparse tensors with a custom function, the user must also define the subset of components in the input arrays that could return a value other than the result array's fill value. This can be done explicitly in a language we define called *iteration algebra*, which we describe in Section 3.6. Figure 3.8 shows how a user can define the iteration algebra to specify that the `gcd` function may return a non-zero result only if at least one input is non-zero. Our technique can then use this iteration algebra to generate the code in Figure 3.9, as described in Henry et al. [83], which computes the element-wise GCD by strictly iterating over the defined values in vectors b and c.

Instead of explicitly specifying iteration algebras for custom functions, users may also annotate functions with any subset of four predefined properties from which our technique can infer optimized iteration algebras:

- **Commutative:** A function is commutative if the order in which arguments are passed to the function does not affect the result. Arithmetic addition is an example of a commutative function, since $x + y = y + x$ for any x and y.

- **Idempotent:** A function is idempotent if, for any x, the function evaluates to x whenever all arguments are x (i.e., $f(x, ..., x) = x$). The max function is an example of an idempotent function, since $\max(x, x) = x$ for any x.

- **Annihilator($\mathbf{x}[, \mathbf{p}]$):** A function has an annihilator x if the function evaluates to x whenever any argument is x. Arithmetic multiplication, for instance, has 0 as its annihilator since multiplying 0 by any value yields 0. If p is also specified, then the function is only guaranteed to evaluate to x if the p-th argument (as opposed to any argument) is x.
- **Identity($\mathbf{x}[, \mathbf{p}]$):** A binary function has an identity x if, for any y, the function evaluates to y whenever one argument is x and the other argument is y. Multiplication, for instance, has 1 as its identity since multiplying 1 by any y yields y. If p is also specified, then the function is only guaranteed to evaluate to y if the p-th argument (as opposed to any argument) is x.

Figure 3.12 demonstrates how a programmer can specify that the `bitwise_and` function is commutative and has 0 as its annihilator. From these properties, our technique infers that the `bitwise_and` function (with inputs x and y) has iteration algebra $x \cap y$ assuming that the input arrays have 0 as fill values, as we will explain in Section 3.6.2.

3.6 Explicit Set Algebra

Having described the desired features of a generalized sparse tensor programming model, we now explain how our general sparse tensor compiler reasons about and implements these features. In this section, we describe how our system reasons about user-defined functions iterating over any iteration space through an IR called *iteration algebra*. Then, we describe how an iteration algebra can be derived from mathematical properties of user-defined functions.

3.6.1 Iteration Algebra

We can view the iteration space of loops over dense arrays as a hyper-rectangular grid of points by taking the Cartesian product of the iteration domain of each loop, as in Figure 2.2a. A generalized sparse iteration space, shown in Figure 2.2b, is a grid with missing points called holes, which take on the *fill value* attached to the format of that array. Another way to view iteration spaces is as a Venn diagram of coordinates where the universe is the set of all points in a dense iteration space. Sparse tensors only define values at some of the possible coordinates in the dense space, forming subsets within the universe, as shown in Figure 2.2c. This view naturally leads to a set expression language for describing array iteration spaces, which we introduce, called *iteration algebra*.

Iteration algebra is defined by introducing index variables into set expressions, where the variables in the set expressions are the coordinate sets of sparse tensors. The index variables index into the sparse tensors, controlling which coordinates are compared in the set expression. For example, the iteration algebra for $c_i = \sum_j A_{ij} b_j$ (i.e., sparse matrix-vector multiplication) is $A_{ij} \cap b_j$, where the j

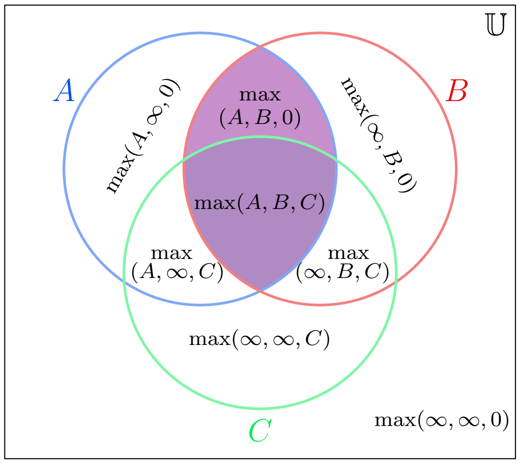


Figure 3.13: Illustration of case (1), where f is the ternary max operator, A and B have fill value ∞ and C has fill value 0.

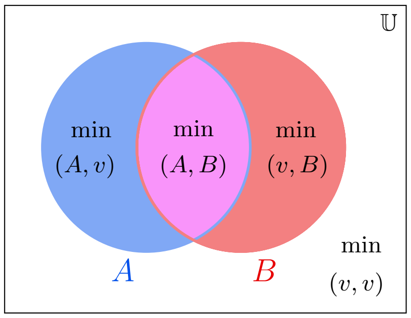


Figure 3.14: Illustration of case (2), where f is the idempotent min operator and all arguments have the same fill value v.

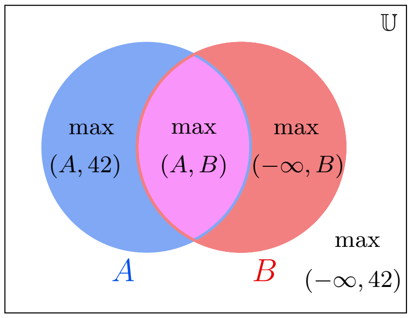


Figure 3.15: Illustration of case (3), where f is the max operator with identity $-\infty$, A has fill value 42 and B has fill value $-\infty$.

in A_{ij} indexes into the second dimension of A and the j in b indexes into the first dimension of b. Coordinate sets indexed by the same index variable are combined using the set operations. In the SpMV example, the j coordinates of A and b are combined with an intersection.

The prior work of Kjolstad et al. [108] intertwines tensor index notation and the corresponding iteration space by interpreting additions as unions and multiplications as intersections. As such, it is limited to describing and working with spaces that are represented as compositions of those intersections and unions. Our iteration algebra addresses this by adding support for *set complements*, which makes the language complete: any iteration space can be described as compositions of intersections, unions, and complements. For example, set complements can be used to express the iteration space $A_{ij} \cap \overline{B_{ij}}$, which contains only coordinates in A that are also not present in B.

Promoting iteration algebra to an explicit compiler IR has two benefits. First, it lets users directly express the iteration space of a complicated function whose space can not be derived from simple mathematical properties. Second, it detaches the compiler machinery that generates low-level loops to iterate over data structures from the unbounded number of functions that a user may define.

3.6.2 Deriving Iteration Algebras

To derive the iteration algebra for an generalized tensor index notation expression, our technique recurses on the expression and derives the algebra for each subexpression by combining the iteration algebras of its arguments. As an example, to derive the iteration algebra for the expression bitwise_and(gcd(b_i, c_i), d_i), our technique first derives the iteration algebra for gcd(b_i, c_i) and then combines it with d_i (the iteration algebra for the second argument of `bitwise_and`).

If a function f is explicitly defined with an iteration algebra alg, then our technique derives the iteration algebra for an invocation of f by replacing the terms of alg with the iteration algebras of the function arguments. In Figure 3.8, for instance, `gcd(x,y)` is defined with iteration algebra $\{x \neq 0\} \cup \{y \neq 0\}$. So to derive the iteration algebra for gcd(b_i, c_i), our technique checks that b and c have 0 as fill values and, if so, substitutes b_i for $\{x \neq 0\}$ and c_i for $\{y \neq 0\}$, yielding $b_i \cup c_i$ as the function call's iteration algebra. (If either b or c has a fill value other than 0 though, our technique instead conservatively returns the universe $\mathbb{U}$ as the function call's iteration algebra.)

If a function is instead annotated with properties, our technique attempts to construct an iteration algebra that minimizes the amount of data to iterate over. This is done by pattern matching on the cases below, in the order they are presented. In particular, assuming a function f is invoked with arguments $args$ in the target expression, we apply the cases below. For each case, we include an example of resulting iteration space on sample inputs, and visual examples for the first three cases in Figures 3.13, 3.14 and 3.15.

1. **f has an annihilator α.** When f is commutative, our technique returns the algebra $\mathbb{U}$ intersected with the algebras of all arguments in $args$ with a fill value of α. Any coordinate c where tensor arguments with fill value α are undefined will cause f to equal α at c because α annihilates f. Therefore, we can iterate only over positions where arguments with fill value α are defined.

   Example. Consider the ternary max operator max(A, B, C), where A and B have fill value ∞ (the annihilator for max, so $\alpha = \infty$) and C has fill value 0. In this case, we emit an algebra to iterate over $A \cap B$. Consider a coordinate c in C. If $c \in A \cap B$, then the max operator will return the maximum of A, B, and C. If $c \in A \cap B$, then no matter what C's value at c is, it will be annihilated by A or B having the value of ∞ (see Figure 3.13).

2. **f is idempotent and all arguments have the same fill value v.** Our technique returns the union of the algebras of all arguments. Since all arguments have fill value v and f is idempotent, f applied at all points outside the union of all arguments evaluates to v.

   Example. Consider the min operator min(A, B), where A and B have some arbitrary fill value v. Because min is idempotent, tt is correct to iterate over the union of A and B —at all coordinates $c \notin A \cup B$, the result of min is min(v, v) $= v$ (see Figure 3.14).

3. f **has an identity** i**.** If all arguments have fill value i, then our technique returns the union of the algebras of all arguments, because computation only must occur where the arguments are defined. If all but one argument have fill value i, then our technique can also return the same algebra, but marks that the resulting expression has the fill value v of the remaining argument, since f applied to i and v returns v.

   Example. Consider the max operator $\max(A, B)$ where A has fill value $-\infty$ and B has fill value 42. Here, we can infer the result tensor should have fill value 42 since the computation at any coordinate outside of $A \cup B$ is $\max(-\infty, 42) = 42$ (see Figure 3.15).

4. f **is not commutative.** When f is not commutative, cases (1) and (3) can be applied, but only to the position p where the property holds.

   Example. Let $f(a, b) = a/b$ has an annihilator 0 at position 0, so case (1) could be applied to iterate only over the defined values of the input array a if it had fill value 0.

If none of these cases match but the result array's fill value is left unspecified by the user, our technique can still return the union of the algebras of all arguments (and constant propagate through f to determine an appropriate fill value for the result). Otherwise, our technique falls back to returning $\mathbb{U}$ as the function call's iteration algebra. In the case of a function call `bitwise_and(x,y)` though, our technique can simply apply the first rule (since Figure 3.12 specifies the function is commutative and has 0 as its annihilator) to derive the iteration algebra $x \cap y$ for the function call. Thus, our technique can infer that the expression $\text{bitwise\_and}(\text{gcd}(b_i, c_i), d_i)$ has $(b_i \cup c_i) \cap d_i$ as its iteration space.

After the derivation of the iteration algebra, the compiler takes the explicit iteration algebra to generate CPU code. Again, the full code generation algorithms and CPU-specific abstractions are at [83]. Next, we evaluate our compiler to show the benefits of generating custom CPU code using explicit set algebras and generalized semirings for sparse tensor programs.

3.7 Evaluating General Sparse Tensor Programs

We evaluate our generalized sparse tensor programming compiler by comparing it to the PyData/Sparse library[5], which is the only general sparse tensor language implementation known to us. We also compare to the less general SciPy/Sparse and GraphBLAS libraries, which consist of hand-implemented functions, to demonstrate our performance against hand-optimized code. Finally, we implement a medical imaging edge detection algorithm and the Minimax algorithm from game theory to demonstrate the applicability of our system. We restrict our evaluation to multi-core CPUs, as our implementation does not yet support GPUs.

[5]Since the publication of this work, the PyData/Sparse library has become the numba backend for the Python Sparse library and can be found here [199]

Table 3.3: FROSTT tensors used in our evaluation

Tensor name	Non-zeros	Order	Shape
nips	3,101,609	4	2,482 x 2,862 x 14,036 x 17
uber-pickups	3,309,490	4	183 x 24 x 1,140 x 1,717
chicago-crime	5,330,673	4	6,186 x 24 x 77 x 32
vast	26,021,945	5	165,427 x 11,374 x 2 x 100 x 89
enron	54,202,099	4	6,066 x 5,699 x 244,268 x 1,176
nell-2	76,879,419	3	12,092 x 9,184 x 28,818

3.7.1 Methodology

All experiments are run on a dual-socket, 12-core Intel Xeon E5-2680 v3 machine @ 2.5 GHz with 30 MB of L3 cache and 128 GB of main memory. The machine runs Ubuntu 18.04.3 LTS. Our system and generated kernels are compiled with GCC 7.5.0. Python 3.6.9 is used to run all Python code. In our evaluation, we compare against PyData/Sparse [2] version 0.11.2, SciPy [220] version 1.5.4, NumPy [79] version 1.19.5, and SuiteSparse:GraphBLAS [50] version 4.0.3. We disable hyperthreading and use `numactl` to restrict execution to a single socket. All execution times, except in Section 3.7.3, are compared over an average of 10 executions.

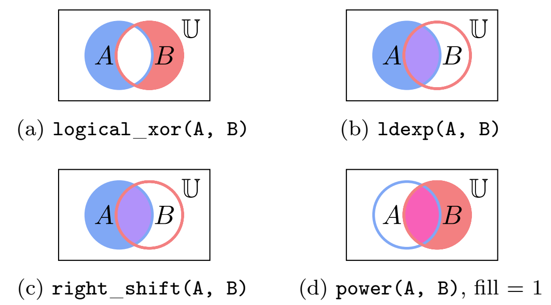
3.7.2 Comparison to Sparse Array Libraries

In the Python ecosystem, programmers have two main options for operating on sparse matrices or arrays: SciPy/Sparse and PyData/Sparse. SciPy/Sparse is a SciPy package for working with sparse matrices. It contains some common sparse matrix formats along with hand-written C implementations for many operations, but is limited in the scope of array programming features supported. For additions and multiplications, our system generates the exact same code as the TACO system [108], which performs competitively with the hand-optimized implementations like those in SciPy [41].

PyData/Sparse is a recent project that supports tensors of arbitrary dimensions in the COO format. Like the NumPy library for dense array processing, it also supports general user-defined functions. The PyData/Sparse implementation utilizes existing NumPy and SciPy/Sparse dense kernels by first transforming and transposing the data into shapes that NumPy and SciPy/Sparse can operate on. Then, the PyData/Sparse algorithm will transform the results back into COO format. While the kernels used by PyData/Sparse are heavily optimized, its data transformation-based approach adds additional data movement overhead. By contrast, our techniques for generalized sparse tensor programming can generate optimized kernels that operate on tensors of any dimension and data format, without performing unnecessary data movement.

Binary Operations

We demonstrate the flexibility and performance of our techniques by implementing a subset of the NumPy element-wise universal functions (ufuncs) that have iteration spaces different from intersection and union. We evaluate the `logical_xor`, `ldexp`, `right_shift` and `power` ufuncs, which have the



(a) `logical_xor(A, B)` (b) `ldexp(A, B)` (c) `right_shift(A, B)` (d) `power(A, B)`, fill = 1

Figure 3.16: Iteration spaces of the benchmarked ufuncs.

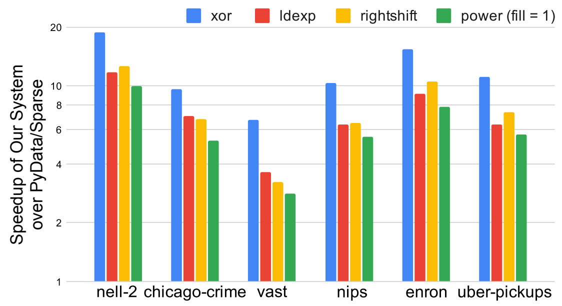


Figure 3.17: Speedup of our system over PyData/Sparse on a log scale for ufunc operations on FROSTT tensors.

iteration spaces shown in Figure 3.16. SciPy/Sparse does not support most ufuncs outside of addition and multiplication and NumPy implementations cannot materialize the tensors into a dense format, so we restrict our comparison to PyData/Sparse.

We evaluate the above ufuncs on the subset of real-valued tensors from the FROSTT tensor repository [190] and SuiteSparse sparse matrix repository [49] that PyData/Sparse could successfully load without memory issues. Characteristics about the FROSTT tensors satisfying these constraints can be found in Table 3.3. The tensors in these datasets were used as the first argument to the ufunc. We constructed synthetic inputs for the second argument by shifting the coordinates in the last tensor mode from the first argument by one position and setting the data to a constant value. Finally, since some ufuncs are sensitive to the particular value in the operand tensor (such as `ldexp`), we filled the shifted tensor with a small constant value of 2. Both PyData/Sparse and our system use a single thread for these kernels.

We show the normalized execution times of PyData/Sparse's ufunc operations on the FROSTT tensors in Figure 3.17 and the execution times of our system and PyData/Sparse on the SuiteSparse matrices in Figure 3.18. The geometric mean speedup of our system is 7.55× on the FROSTT tensors and 4.24× on the SuiteSparse matrices. The PyData/Sparse's data-movement heavy approach to generalized sparse tensor programming is much slower than our approach on all inputs, which iterates directly through the sparse data structures for the exact iteration pattern of the target ufunc.

Fusing Operations

Our approach for generalized sparse tensor programming can also fuse different functions together, which avoids doing work that is then discarded. Our technique only iterates over the spaces where defined values are produced, which avoids materializing temporaries. We evaluate the fused kernels described in Figure 3.20, which are constructed with the `logical_xor`, `logical_and`, and `logical_or` ufuncs. We use the FROSTT tensors described in Section 3.7.2 as inputs to the fused functions, with the third tensor argument created by the same shifting operation described previously. We report normalized execution times of fused operations in PyData/Sparse against our system

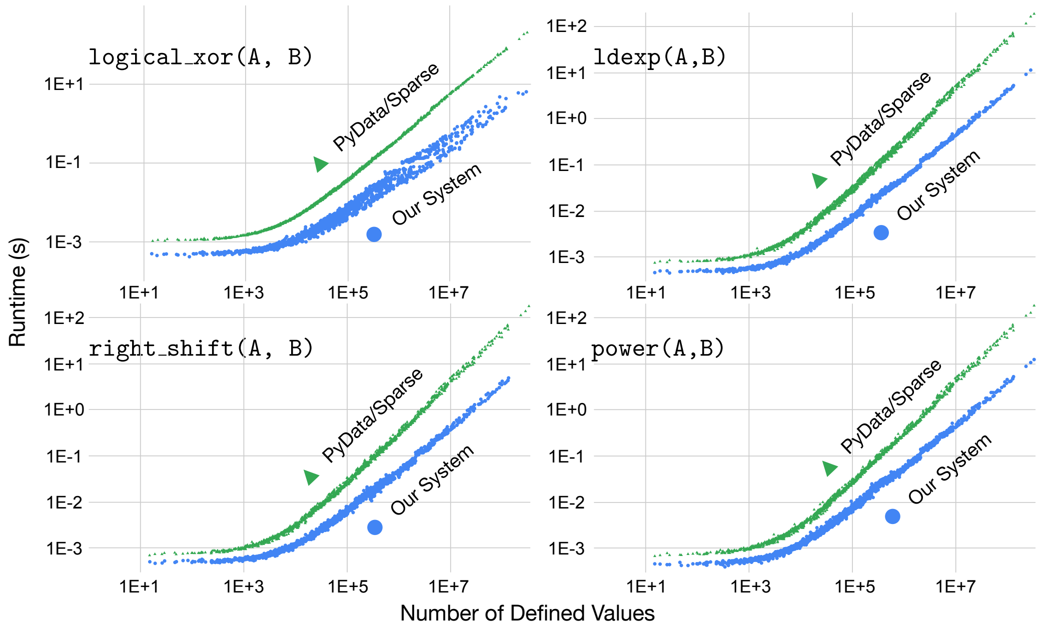


Figure 3.18: Ufunc operations on SuiteSparse arrays using a log-log scale.

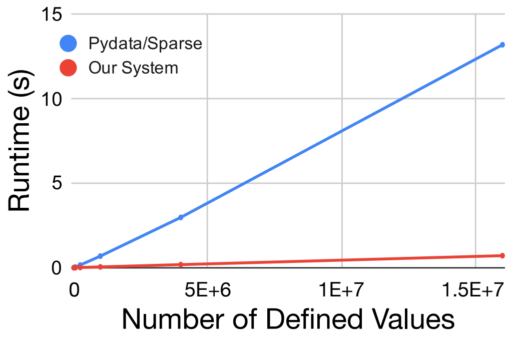


Figure 3.19: Scaling of fused and/xor operation on random arrays.

in Figure 3.21. By fusing operations, our system iterates over less data and avoids allocation of intermediate results. By contrast, PyData/Sparse's allocation of an intermediate and extra pass over the data cause it to have an even larger slowdown than for a single ufunc application.

Fusing functions can decrease the amount of work realized in the final output tensor. For example, consider the fused `and` and `xor` kernel described in Figure 3.20. Without fusing, first `xor(A, B)` must be computed, and then the result must be `and`-ed with `C`. Since the annihilator of `and` is `false`, all coordinates in the iteration space of `xor` that are not present in `C` will be `false` in the final result. If the `and` and `xor` functions are fused, then the generated code avoids computing any values for `false` coordinates in `C` in the first place. To demonstrate this effect, we compare the fused `and`–`xor` kernel of PyData/Sparse to our system on a set of square matrices of increasing size where each matrix has a uniformly random sparsity of 1%. We plot the execution time of both systems against the number of defined values in the matrix in Figure 3.19. Our system is already generally faster

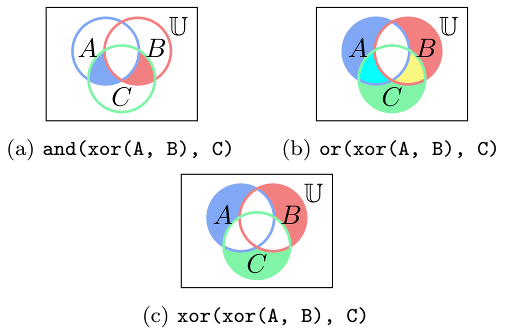


Figure 3.20: Iteration spaces of fused ufunc benchmarks.

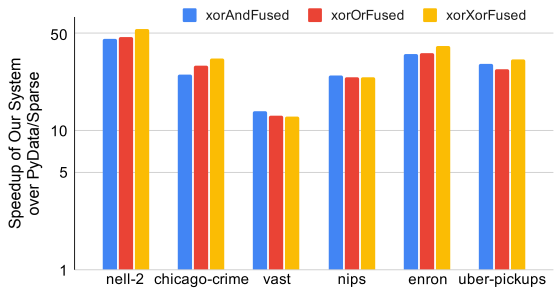


Figure 3.21: Log-scale speedup of our system over PyData/Sparse for fused ufunc operations on FROSTT tensors.

than PyData/Sparse, but our system's execution time still grows at a slower rate because it can avoid doing operations that are later ignored through fusing.

Memory Usage

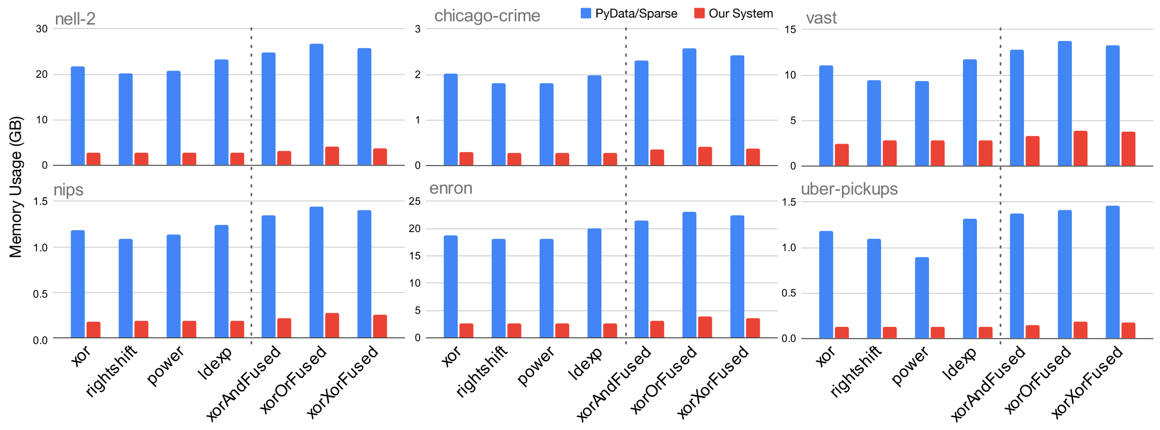


Figure 3.22: Memory usage (in GB) of our system versus PyData/Sparse on FROSTT tensors. The experiments are run for both the ufunc and fused operations, which is differentiated by a vertical dashed line.

We present memory usage results between our system and PyData/Sparse on individual and fused ufunc operations on each of the considered FROSTT tensors. To measure the memory used for our system, we count the size of allocations performed to hold input and output data structures. Since our approach does not allocate any temporary data structures, this is all of the memory used by our system. To measure the memory used by PyData/Sparse, we use the Python package `memory_profiler`, which records a line-by-line profile of the total memory allocated/released by each line of a Python program.

We present the memory usage results in Figure 3.22, which groups the memory used data by each considered tensor in the FROSTT dataset. On average, we find that PyData/Sparse uses an average of 6.55x more memory than our system to perform these ufunc operations. PyData/Sparse

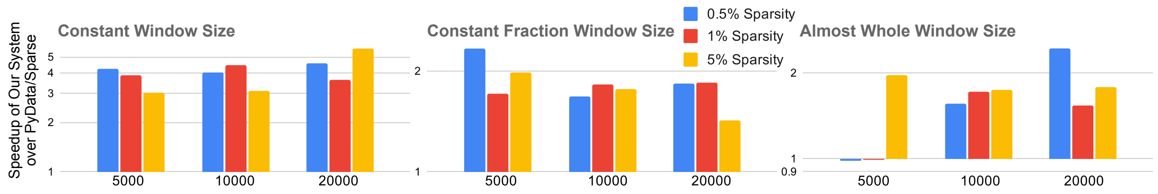


Figure 3.23: Various window size slicing operations across different square matrix sizes, parameterized by sparsity and plotted on a log-scale.

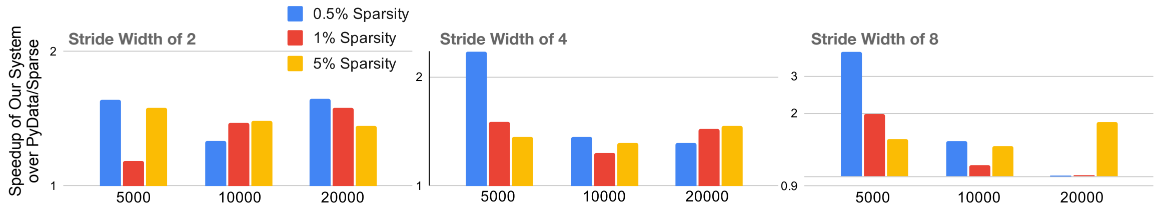


Figure 3.24: Various stride length slicing operations across different square matrix sizes, parameterized by sparsity and plotted on a log-scale.

uses a similar amount of memory as our system to store the input and output tensors, but is unable to compress as much of these tensors since they are stored in a coordinate list format. Much of the measured space overhead of PyData/Sparse comes from large amount of intermediate storage allocated during execution of the ufunc operation, which is often more than twice the space used to store the input tensors themselves.

Slicing

To evaluate the performance of code that our technique generates to perform slicing and striding, we perform an element-wise addition between uniformly random square sparse matrices with varying sparsities that have been sliced in different ways. We choose a simple kernel and input tensors to highlight the costs of data movement caused by slicing operations. We consider square slices of a constant 500×500 size, a constant fraction (1/4) of the matrices' rows, and slices that contain the entire matrix except for the first and last rows. We separately consider slices of the whole matrix with strides of widths 2, 4, and 8. Execution times normalized against our system for slicing and striding can be found in Figs. 3.23 and 3.24. We exclude results from PyData/Sparse in these figures as it was consistently an order of magnitude slower than both our system and SciPy/Sparse.

The geometric mean speedup of our system over SciPy/Sparse is 2.25× on the slicing benchmarks and 1.47× on the striding benchmarks. These speedups come from the implementation strategy of SciPy/Sparse and PyData/Sparse. In particular, SciPy/Sparse and PyData/Sparse implement slicing by first deep copying and repacking elements into their new sparse array, and then performing the desired computation. This deep copying and repacking step incurs extra cost compared to our approach, which operates directly on the slice.

Table 3.4: Performance of (complement-masked) matrix-vector multiplication (`mxv`) kernels, generated by our generalized sparse tensor compiler and implemented in SuiteSparse:GraphBLAS, on varying SuiteSparse matrices. Relative and absolute execution times (in parentheses) are shown, with the faster implementation highlighted in gray. We use input vectors that are 25% dense and mask vectors with 25% of elements being false. All matrices are stored in CSR, while mask vectors are stored using dense arrays and other vectors are stored using byte maps. Our technique generates code that is competitive with SuiteSparse:GraphBLAS in terms of performance, with the generated code being 1.13–1.26× as fast as hand-optimized code on average.

Matrix	Boolean Semiring		Tropical Semiring	
	SuiteSparse	Our System	SuiteSparse	Our System
belgium_osm	1× (1.62 ms)	1.38× (1.17 ms)	1× (2.33 ms)	1.05× (2.22 ms)
cit-Patents	1× (8.08 ms)	1.33× (6.06 ms)	1× (17.3 ms)	1.19× (14.6 ms)
coAuthorsCiteseer	1× (0.336 ms)	1.80× (0.186 ms)	1× (0.561 ms)	1.35× (0.417 ms)
coPapersDBLP	1× (1.76 ms)	1.77× (0.993 ms)	1× (9.19 ms)	1.18× (7.79 ms)
delaunay_n24	1× (27.2 ms)	0.768× (35.5 ms)	1× (68.4 ms)	0.98× (69.7 ms)
indochina-2004	1× (17.8 ms)	1.22× (14.7 ms)	1× (46.3 ms)	0.967× (47.9 ms)
rgg_n_2_24_s0	1× (47.6 ms)	0.941× (50.6 ms)	1× (138 ms)	1.18× (116 ms)
road_central	1× (25.5 ms)	1.04× (24.4 ms)	1× (52.8 ms)	1.01× (52.1 ms)
road_usa	1× (31.6 ms)	0.889× (35.6 ms)	1× (73 ms)	0.957× (76.2 ms)
roadNet-CA	1× (2.17 ms)	1.35× (1.61 ms)	1× (5.5 ms)	1.54× (3.57 ms)
ship_003	1× (0.214 ms)	2.00× (0.107 ms)	1× (1.04 ms)	1.11× (0.93 ms)
soc-LiveJournal1	1× (15.2 ms)	1.25× (12.2 ms)	1× (46.3 ms)	1.12× (41.3 ms)
Geometric mean	1×	**1.26×**	1×	**1.13×**

Table 3.5: Performance of matrix-matrix multiplication (`mxm`) kernels, generated by our generalized sparse tensor compiler and implemented in SuiteSparse:GraphBLAS, on varying SuiteSparse matrices. Relative and absolute execution times (in parentheses) are shown, with the fastest implementation highlighted in gray. All matrices are stored in CSR, and we use each matrix as both inputs to the kernel. For SuiteSparse:GraphBLAS, we report results for whichever algorithm the library chooses (Any) as well as for their implementation of Gustavson's algorithm, which uses dense arrays to store partial results. Again, on the whole, our technique generates code that is competitive with SuiteSparse:GraphBLAS in terms of performance, with the generated code being 0.836–1.02× as fast as hand-optimized code on average.

Matrix	Boolean Semiring			Tropical Semiring		
	SuiteSparse (Any)	Our System	SuiteSparse (Gustavson's)	SuiteSparse (Any)	Our System	SuiteSparse (Gustavson's)
belgium_osm	1× (0.032 s)	0.896× (0.036 s)	0.256× (0.125 s)	1× (0.040 s)	0.599× (0.067 s)	0.243× (0.166 s)
cit-Patents	1× (0.655 s)	0.804× (0.815 s)	0.506× (1.30 s)	1× (0.816 s)	0.577× (1.41 s)	0.451× (1.81 s)
coAuthorsCiteseer	1× (0.077 s)	1.40× (0.055 s)	1.02× (0.076 s)	1× (0.118 s)	1.14× (0.104 s)	0.921× (0.128 s)
coPapersDBLP	1× (2.48 s)	1.12× (2.2 s)	1.00× (2.48 s)	1× (4.27 s)	0.946× (4.51 s)	1.00× (4.27 s)
delaunay_n24	1× (1.99 s)	1.05× (1.90 s)	0.966× (2.06 s)	1× (2.49 s)	0.955× (2.61 s)	0.938× (2.65 s)
indochina-2004	1× (120 s)	0.997× (121 s)	1.00× (120 s)	1× (207 s)	0.748× (276 s)	0.999× (207 s)
rgg_n_2_24_s0	1× (7.77 s)	1.23× (6.34 s)	1.21× (6.44 s)	1× (10.8 s)	1.13× (9.59 s)	1.12× (9.69 s)
road_central	1× (0.941 s)	0.852× (1.1 s)	0.66× (1.43 s)	1× (1.11 s)	0.661× (1.68 s)	0.627× (1.77 s)
road_usa	1× (0.777 s)	0.689× (1.13 s)	0.698× (1.11 s)	1× (0.966 s)	0.558× (1.73 s)	0.681× (1.42 s)
roadNet-CA	1× (0.064 s)	0.836× (0.077 s)	0.837× (0.076 s)	1× (0.083 s)	0.687× (0.121 s)	0.832× (0.1 s)
ship_003	1× (0.14 s)	1.28× (0.109 s)	1.22× (0.116 s)	1× (0.236 s)	1.31× (0.18 s)	1.19× (0.198 s)
soc-LiveJournal1	1× (22.6 s)	1.32× (17.2 s)	1.18× (19.1 s)	1× (42.4 s)	1.17× (36.1 s)	1.10× (38.7 s)
Geomean	1×	**1.02×**	**0.815×**	1×	**0.836×**	**0.778×**

3.7.3 GraphBLAS Kernels

Many graph algorithms, such as those for performing breadth-first search and for solving the all-pairs shortest paths problem, can be expressed using linear algebra (with semirings beyond the standard $(+, \times)$ semiring) [101]. For instance, each iteration of breadth-first search on a graph can be expressed as the multiplication of the graph's adjacency matrix by a vector that represents the current frontier, which returns a new vector that represents the next set of vertices to be visited. Kepner et al. [100]

show how GraphBLAS, an API that exposes a fixed set of common linear algebra primitives like matrix-vector multiplication (`mxv`) and matrix-matrix multiplication (`mxm`), can be used to implement efficient graph applications.

Our technique can generate efficient code for many of the core primitives in GraphBLAS. To demonstrate this, we use our technique to generate code that implement `mxv`

$$y_i = \texttt{mask}(\neg m_i, \bigoplus_j (A_{ij} \otimes x_j))$$

and `mxm` ($A_{ij} = \bigoplus_k (B_{ik} \otimes C_{kj})$) for both Boolean and tropical semirings. (The tropical semiring replaces $\oplus$ with `min` and $\otimes$ with $+$, while the Boolean semiring assumes that all values are Boolean and replaces $\oplus$ with $\vee$ and $\otimes$ with $\wedge$. `mask` returns the value of the second argument only if the first argument evaluates to true.) We then measure the performance of the generated code (running with 12 threads) and compare it against that of SuiteSparse:GraphBLAS [50], a highly-optimized implementation of GraphBLAS. We report average execution times over 1000 iterations for `mxv` and over 100 iterations for `mxm`.

Table 3.4 and Table 3.5 show the results of our experiment. For `mxv`, our system is on average 1.26× faster than SuiteSparse:GraphBLAS (i.e., SuiteSparse) when computing with the Boolean semiring and 1.13× faster when computing with the tropical semiring. This is because our technique generates code that uses the same algorithm as SuiteSparse but is fully specialized to the input's types and formats. By contrast, SuiteSparse, though partially specialized using C macros, still has to perform some dynamic dispatching to handle inputs of arbitrary types and formats, which adds run-time overhead. For `mxm`, our technique is on average 1.02× faster than SuiteSparse when computing with the Boolean semiring and has performance 0.836× that of SuiteSparse when computing with the tropical semiring. Our generated code and SuiteSparse both use a linear combination of rows algorithm to compute the kernel. However, when the output matrix has relatively few defined values per row, SuiteSparse is able to use hash tables to store partial results. By contrast, our technique currently can only generate code that stores partial results using dense arrays, thus reducing cache efficiency. Table 3.5 also shows, though, that the code our technique emits has similar or better performance on average than SuiteSparse when the latter also uses dense arrays to store partial results (which is preferable when the output is relatively dense).

3.7.4 Applications

To demonstrate the usefulness of our system, we used it to implement two algorithms: an edge detection algorithm from medical imaging and the MinMax algorithm for game decision making. We compare our system to implementations using NumPy and PyData/Sparse.

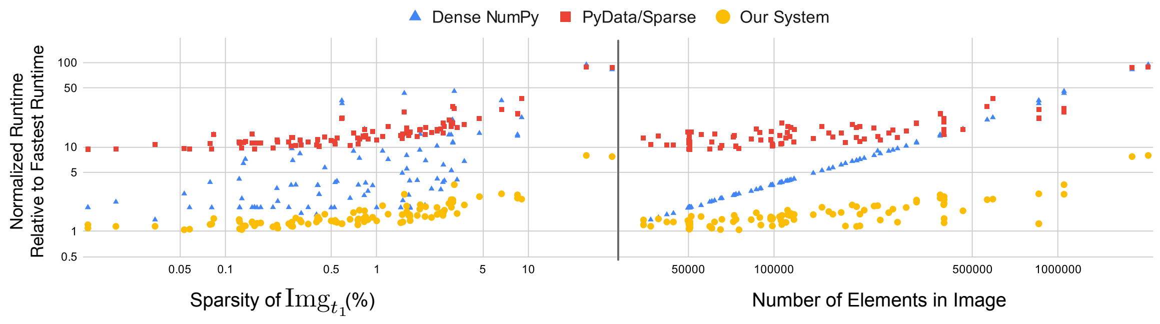


Figure 3.25: Medical imaging edge detection performance normalized by the fastest runtime point on a log-log scale plotted with respect to Img_{t_1} sparsity (left) and number of image pixels (right). The normalized runtime value of 1 corresponds to an absolute runtime of 244.7 μs

Medical Imaging Edge Detection

Image processing and computer vision approaches often use generalized tensor (array) programming. More specifically, the medical imaging field applies these processing techniques to patient images from various imaging modalities. Oftentimes, after initial measurements are taken, the measurements are post-processed for digital enhancement, diagnostic purposes, and even to create domain specific machine learning models. Many systems and libraries exist for (grayscale) medical image analysis that include functions like logical `and`, `xor`, `or`, and `not` [226, 91, 102]. We implement boundary edge detection on magnetic resonance imaging (MRI) images [195] to demonstrate the practical application of our generalized sparse tensor programming system. We implement a computer vision thresholding technique to determine the edges of an MRI image, which are then filtered using a region-of-interest (ROI) mask (see Appendix Figure 33 in the supplemental materials[4]). The masked edge detection is represented by the equation $\mathrm{Img}_{post} = (\mathrm{Img}_{t_2} \wedge \mathrm{ROI}) \oplus (\mathrm{Img}_{t_1} \wedge \mathrm{ROI})$, where Img is the original two-dimensional single-channel MRI image and Img_{t_1} and Img_{t_2} are thresholded versions of Img using $t_1 = 75\%$ and $t_2 = 80\%$ respectively.

We compare the average execution time of the masked edge detection on MRI brain images from a dataset on Kaggle [29]; an example image can be found in Appendix 33 in the supplemental materials[4]. Our generalized sparse tensor compiler has a geometric mean speedup of 2.69$\times$ (0.96$\times$ to 28.9$\times$) faster than the dense NumPy implementation and 9.41$\times$ (6.58$\times$ to 17.9$\times$) faster than the PyData/Sparse implementation, as shown in Figure 3.25. We also demonstrate the benefits of sparsity since the dense implementation scales linearly with the number of image pixels.

Game Playing Minimax Algorithm

In game theory, game choices are often represented as a decision tree where each node represents a potential state in the game and each edge represents a move decision, with leaf-node values representing a heuristic of that game state (see Figure 3.27). In addition to interpreting generalized sparse tensors as images (Section 3.7.4) or graphs (Section 3.7.3), we can use sparse tensors to represent tree-like structures. Artificial intelligence algorithms, like the Minimax algorithm, are often

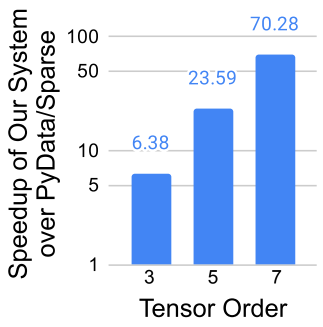
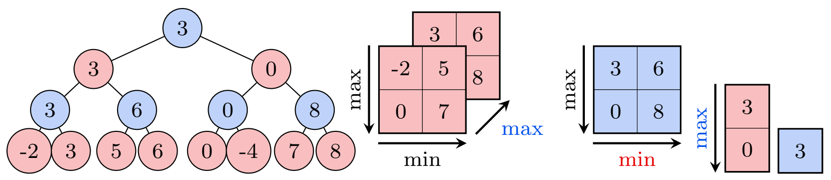


Figure 3.26: Minimax speedup plotted on a log-scale



(a) Example game decision tree (b) Higher-order tensor representation

Figure 3.27: Simple Minimax Algorithm Example

used on these game-state decision trees to calculate the optimal move given a starting game position and assuming that the opponent will also choose their moves optimally. Our system can represent the Minimax algorithm, which alternates taking the minimum and maximum value at each level of the game tree, as: opt $= \max_i \min_j \max_k \ldots (A_{ijk\ldots})$.

We implement this algorithm on our system and compare against PyData/Sparse for higher-order tensors. We cannot compare against NumPy since the tensors are too large for a dense representation, and we cannot compare against SciPy/Sparse since these tensors have greater than two dimensions. We test this for tensor orders $o = 3, 5, 7$, where the dimensions are $20 \times 20 \times 43^{(o-2)}$ to represent the first o moves of a chess game; this was chosen since chess has 20 opening moves and then 43 moves on average for a board state at any given time. The sparsity of the tensor comes from sparse sampling and pruning of the decision tree, where the fill values represent pruned nodes. Figure 3.26 illustrates that we outperform PyData/Sparse by 6.38× to 70.3× depending on the tensor order and that our performance improves significantly with increasing tensor order.

3.8 Context within Prior Sparse Array and Tensor Programming Systems

We explore four areas of related work, ordered from most to least relevant. We begin with prior work on execution of generalized sparse tensor programs, which is divided into two strategies: generating bespoke code for each operation or emulating programs by reorganizing data and then calling hand-written functions. We then survey the large body of work on array programming models (for dense arrays). And finally, we discuss prior work on generalizing sparse linear and tensor algebra for machine learning and graph algorithms.

3.8.1 Generalized Sparse Tensor Language Compilation

The work in this chapter is the first to describe how to generate bespoke code for general sparse tensor programs—any function applied across sparse and dense arrays with any fill value, including element-wise application, broadcasts, and reduction. But there exists prior work on generating code for sparse linear and tensor algebra, which are subsets of generalized sparse tensor languages where the functions must be additions and multiplications applied in linear expressions.

Most directly related to our work is the body of work on the Sparse Tensor Algebra Compiler (TACO) [108, 41, 106, 181, 40]. Our work shows how to generalize the compilation theory behind TACO [110] to the much broader class of array programs, by allowing any function to be applied to sparse tensors with any fill value. We achieve this generalization by introducing functions with annotated properties, an iteration algebra containing complements, and omitter points to iteration lattices.

Other prior work on generating bespoke code for subsets of sparse linear algebra include the MT1 [24], Bernoulli [116], and CHiLL-I/E [216] compilers, which analyze and transform imperative code that implements dense linear algebra kernels to sparse implementations. CHiLL-I/E can transform dense and sparse multiplication operations on matrices and vectors to implementations where one operand is sparse. MT1 supports those operations as well as several other built-in operators and intrinsic functions, such as +, ==, and `sqrt`. It can also generate code for operations with multiple sparse data structures by introducing dense temporaries, thus turning them into sparse-dense iteration. Bernoulli maps dense linear algebra implementations to relational algebra and then further maps the relational algebra to templated sparse implementations.

3.8.2 Sparse Array Frameworks via Emulation

The alternative explored in prior work to generating bespoke code for array computations is to emulate sparse array programs using a finite set of hand-written implementations. That approach requires data movement to reshape the data to the available functions. An early system of this sort is the MATLAB Tensor Toolbox [12], which executes high-order tensor algebra by re-organizing the tensors to look like matrices. Since this approach requires pre/post-processing to re-organize data, it is slower than bespoke generated implementations.

We know of only one prior system for the general category of generalized sparse tensor programming languages, namely the PyData/Sparse library [2]. This system executes one two-operand operation at a time in the following steps: First, a hand-written function iterates through the defined elements of the two array operands and divides them into three sets: those that both have defined values, those that only the first operand has, and those that only the second operand has. Next, it invokes NumPy's dense implementations to compute the function at hand on each of those subspaces. And finally, it re-integrates the three sets of resulting values into a result array. By contrast, our work generates bespoke implementations that do not require data pre/post-processing and therefore performs significantly better, as shown in our evaluation.

3.8.3 Array Languages

There is a large body of prior work on dense array programming models, as defined in this chapter, going back to APL [96]. Modern variants include ZPL [129, 30] and NumPy [79], but the core operations—element-wise operations, reductions, and broadcasts—remain the same. Furthermore, many compiler techniques have been developed to compile dense array programs, including the polyhedral model [124]. Our novel contribution is compiling the array programming model to sparse tensors. Another key insight, which differs from dense array programming models, is that functions applied across sparse tensors must be decorated with algebraic properties for the system to be able to generate efficient code.

3.8.4 Generalizations of Sparse Linear and Tensor Algebra

Two additional bodies of work have made steps towards the full generalization of sparse tensor programming, by generalizing sparse linear and tensor algebra to compute sparse neural networks and graph algorithms. Systems with support for sparse neural networks must support non-linear functions in addition to sparse linear algebra. For example, PyTorch [160] supports hand-implemented `softmax` and `log_softmax` on sparse tensors, while TensorFlow [1] supports max element-wise and reduction operations. We expect new non-standard operations to keep arising in the future, which motivates a comprehensive generalized sparse tensor programming model.

In addition, several researchers [139, 100, 50] have defined and implemented APIs, namely GraphBLAS, for linear algebra computations where the operations are different semirings than $(+, \times)$, such as $(\wedge, \vee)$ or $(\min, +)$. Computations in different types of semirings provide surprising features, such as the ability to compute several graph algorithms through matrix multiplications. And since all semirings behave linearly, the same implementation can be reused by just replacing the meaning addition and multiplication. Researchers have also proposed sparse tensor algebra libraries with support for multiple semirings [193]. Generalized sparse tensor programming supports operations in different semirings, but generalizes the programming model to computations with any function, whether it partakes in a semiring or not. Furthermore, our work generalizes the arrays to support any fill value.

Chapter 4

A Unified Sparse Dataflow Abstraction

“Everything should be made as simple as possible, but not simpler.”

Attributed to Albert Einstein

“It can scarcely be denied that the supreme goal of all theory is to make the irreducible basic elements as simple and as few as possible without having to surrender the adequate representation of a single datum of experience”

Albert Einstein

Stable instruction sets that span multiple generations of hardware implementations have been essential for general purpose processors. The development of stable ISAs, pioneered by the IBM360, enabled the separation of hardware designs from their compilers and system software. Furthermore, stable ISAs provide architects with a specification for their next-generation hardware design. As long as architects implement the instructions defined in the instruction set, they have guarantees about functionality and completeness of their processor.

A unified sparse dataflow abstraction will enable the evolution of families of sparse tensor engines in the same way that well-crafted, stable ISAs did for processors. Today’s sparse accelerators have one-off programming interfaces that require bespoke software. A unified sparse data flow abstraction, and more generally the idea of abstract machines for domain-specific architectures, is the key innovation towards a stable, architecturally-independent description of sparse dataflow. These abstract machines let us create a robust programming system for new classes of machines, through the stable and expressive interface between accelerators and compilers.

This work will be remembered as a unifying abstract machine for sparse tensor algebra. While many sparse tensor algebra dataflow accelerators exist, *this work is the first to unify them under a single abstraction.* To date, we have shown that our unified dataflow abstraction named the sparse abstract machine (SAM) is expressive enough to describe six architectures [246, 157, 81, 176, 252] (including the one introduced in Chapter 5). The following attributes enable the elegance, generality, and usefulness of this unifying abstraction:

A complete set of composable dataflow primitives for sparse tensor algebra Similar to RISC ISAs, a small number of simple SAM primitives is sufficient for the entire domain. This project has been referenced as a specification by architects interested in designing dataflow accelerators for sparse applications. As long as hardware designers implement the SAM primitives, their machine will be complete for the language of sparse tensor algebra. With SAM as a specification, hardware designers need not be concerned about the composibility of their modules or the completeness of their design, but can focus on micro-architecture, performance, and optimizations. SAM has already been used as an architectural guide to fabricate a coarse-grained reconfigurable array chip for sparse applications as shown in Chapter 5.

A composable representation of tensors as per-dimension hierarchical streams. *This work presents the first per-dimension streaming dataflow tensor representation.* The unique hierarchical streaming representation described in Section 4.4 makes seamless composition, and thus modularity, possible by composing per-dimension coordinate streams from separate tensors. The design decision to embed stop tokens into the streams to separate fibers (e.g., rows) simplifies the design because any primitive can process any stream, regardless of which tensor dimension it represents. The representation is also tensor format agnostic, abstracting over specific compressed data structures. Our streaming tensor representation has inspired four other follow-on works [177, 122, 192, 245] that employ a similar per-dimension streaming abstraction with control tokens embedded directly into the data plane. The four follow-on works employ a similar abstraction data model for other domains like sparse machine learning, dense machine learning with dynamism, and a model for dataflow threading of dense computation.

A simple architectural abstraction that eases the software stack The unification through SAM acts as a virtual machine and a unified compiler IR. The infinite resources within the SAM IR simplifies the compilation flow and leaves optimizations as rearrangements of the SAM IR itself, which parallels the idea of three address code compilers. The three address code IR unified multiple programming languages by introducing ideas such as an infinite register machine and optimizations simply as code rearrangements. Additionally, we designed SAM to be a modular and easily readable IR, as in LLVM IR. Thus, SAM provides a unifying programming model for future sparse architectures, enabling them to run programs that were written by programmers other than the architects. When architects follow our SAM specification, they get end-user friendly software for free. *We developed Custard (Section 4.7), a compiler from high-level Einsum notation to the sparse abstract machine.* We also show in Chapter 5 how to lower SAM to a fabricated accelerator implementation, reducing the work required to make an accelerator usable. And we envision that other sparse accelerators (from prior work and the future) will follow suit.

SAM will be foundational in the development of abstract machines for other domains. As described above, there are numerous benefits to a well-crafted, systematic abstract machine, and this type of abstract machine can influence more than just the domain of sparse tensor algebra. SAM has already impacted other such designs for sparse machine learning [122]. We believe the ideas in this work extend to other domains, including domain-specific architectures for relational algebra, graph computation, recurrence equations, general array programming, and dynamic machine learning [192, 245].

Beyond compilers and abstractions, the ideas behind the SAM have also begun to influence both sparse software systems and accelerator hardware design. In the software domain, the D2T2 system [186] came about after SAM's simulation framework showed the performance impact of different accelerators and their dataflows on different sparse tensor input data. The D2T2 sought to better tile sparse tensors to fit the fixed buffer capacities of sparse accelerators targeted by SAM (including the Onyx accelerator present in Chapter 5). In hardware, the FEASTA [255] architecture adopts many of the same SAM dataflow primitive abstraction functionality within their FEASTA ISA. For example, the SAM dataflow block analogous FEASTA instructions include: the level scanners and load fiber instructions, intersector and intersection of fibers, the ALU primitive and calculation of operations and products, and the level writers and write/store fiber instructions. Together, these extensions illustrate that SAM is not only a unifying abstraction for sparse compilation, but also a conceptual framework that informs how software and hardware systems can reason about sparse tensor algebra as a first-class execution model.

4.1 Motivating Sparse Dataflow Unification

Specialized streaming dataflow accelerators that leverage pipelining, locality, and parallelism are becoming increasingly popular as performance- and energy-efficient alternatives to CPUs and GPUs. But the efficiency comes at the cost of programmability: all have limits to their application domain, and most have limited and/or difficult programming interfaces. As a result, users often access these accelerators through library calls that are created by expert programmers [218, 219]. Recent research has proposed closing this flexibility and programmability gap by creating reconfigurable dataflow architectures or coarse-grained reconfigurable arrays [165, 153, 45, 151, 28, 81, 158], including compilation tools to map a class of user applications to these arrays [251, 111, 133, 166, 223].

Given these trends, it is not surprising that interest in general accelerators for sparse tensor algebra is increasing [81, 45, 176]. Sparse tensor algebra has applications across many fields including science, engineering, data and graph analytics, and machine learning [60, 27, 113, 101, 1, 159]. Tensor algebra generalizes linear algebra to higher-order tensors, and "sparse" indicates tensor algebra computations where one or more tensors are stored in compressed data structures that omit zeros. Sparse tensor algebra, expressed as a language using tensor index notation or Einstein summation (Einsum) notation, is an important language with a long history, starting as a mathematical notation [174]. It has recently gained traction as a computational language [12] that subsumes linear algebra. To accelerate these computations, many papers have also been published on point solutions for single-expression hardware, where the expression is often sparse matrix multiplication [157, 169, 252, 246, 202, 201].

Most sparse tensor algebra accelerators are fixed-function matrix multiply engines. Arbitrary sparse tensor contractions must therefore be reduced to sparse matrix multiplications through algebraic *factorization* [193, 194]. Factorization breaks up large expressions using transpositions, tensor-to-matrix conversions, and temporaries. However, compared to dense tensor algebra, factorization is significantly more expensive for sparse tensor algebra. In fact, a sequence of matrix multiplications is often more than an order of magnitude slower compared to bespoke generated tensor contractions. And, more importantly, the lack of fusion in sparse computations can, and often does, lead to inferior worst-case asymptotic complexity: the runtime of an unfused expression may grow with the number of tensor components while a fused expression grows with the number of nonzero components [110]. To address the limitations of fixed-function accelerators, the SPU [45], ExTensor [81], and Capstan [176] systems propose programmable sparse dataflow hardware. However, they lack full support for sparse tensor algebra.

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

We built SAM to be for sparse tensor dataflow accelerators what LLVM [125] is for instruction-based conventional processors: It defines the machine functionality and provides an interface between the compiler and hardware, allowing the end-to-end system to continue to function while both sides are independently optimized. SAM also enumerates the primitives that are needed to support all features of sparse tensor algebra. Our contributions are:

1. the first abstract machine model that expresses the whole of sparse tensor algebra computations as spatial dataflow graphs on multidimensional sparse and dense tensors,

2. cleanly defined dataflow primitives for each of the fundamental features of sparse tensor algebra,

3. a representation of multidimensional sparse/dense tensors as flattened streams with hierarchical control tokens, and
4. a compilation strategy from a high-level tensor index notation to our abstract machine model.

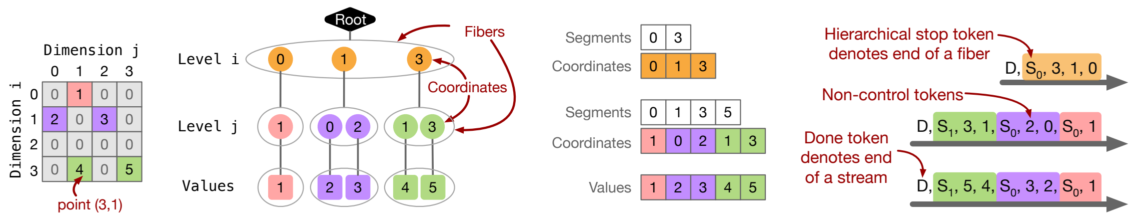
To evaluate our contributions we implemented SAM as a cycle-approximate simulator that is generated by our compiler, Custard. Using the simulations, we search the space of sparse tensor algebra architectural designs. Finally, we show that SAM can represent prior sparse dataflow accelerators.

4.2 The Core Sparse Dataflow Abstraction

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

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

SAM is sufficiently expressive to represent any dataflow for any tensor algebra expression, as it implements all the features in Section 2.1. Furthermore, SAM implements a streaming model of those features, and Kovach and Kjolstad have shown an equivalence proof of sparse tensor algebra and a formal streaming model after the original publication of this work [121].



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

Figure 4.1: The data model of the SAM models sparse tensors (Figure 4.1a) as a coordinate fibertree, (Figure 4.1b) that can be stored in memory (Figure 4.1c) as DCSR data structure, or sent through streams (Figure 4.1d) where time increases from right to left.

4.3 Tensor Data Model

In the SAM abstract data model, each tensor is a coordinate tree where each tree level represents the coordinates of a different tensor dimension. This coordinate tree abstraction was first introduced as part of the TACO system [108, 41] and further abstracted and formalized as *fibertrees* [208, 231]. Fibertrees are tries where each coordinate at one level is linked to a *fiber*—a list of child coordinates—at the next level. Crucially, only those children whose sub-trees have nonzeros are stored. Figure 4.1a shows a sparse matrix and Figure 4.1b its corresponding fibertree. The matrix is stored in row-major order, so the i coordinates (orange circles) are stored at the top fibertree level. The i coordinate 2 is not stored since its sub-tree (the third row) has only zeros. The middle level stores a j coordinate for every nonzero component and the last level stores nonzero tensor values. Fibertrees are useful for reasoning about tensors level by level without considering the exact storage representation.

Fibertrees are stored in memory and transmitted via streams. When in memory, each tree level is separately assigned a storage type that specifies its data representation. A level's data representation can be an uncompressed level that stores a single number encoding the fiber size or it may be a compressed data structure that stores only coordinates with nonempty sub-trees. Many other data representations are possible with this abstraction [41, 208, 231]. Figure 4.1c depicts one possible in-memory data structure for the fibertree in Figure 4.1b, where both levels are stored in compressed data structures. This storage format is called doubly-compressed sparse rows (DCSR), where a segment array denotes the start and stop reference positions of each segment in the coordinate array. A segment is one way to encode fiber data associated with an array representation. Concretely in Figure 4.1c, the level j segment [3, 5) refers to the green level j coordinates [1, 3] since the coordinates are located at indices [3, 4] in the level j coordinate array.

4.4 Tensor Streams

SAM streams are abstractions of physical wires that connect processing blocks and transmit data between these blocks. Each SAM stream is a sequence of tokens that transmits one level of fibertree data, along with stop tokens (S_n) denoting the hierarchical fiber boundaries within a level, and a done token (D) to mark the end of a stream. There are three types of SAM streams: coordinate streams (abbreviated as `crd`) that transmit coordinate levels, value streams (`vals`) that transmit last-level tensor values, and reference streams (`ref`) that transmit references to the location of each coordinate's child fiber in memory. Figure 4.1d shows the fibertree in Figure 4.1b as coordinate and value streams. Streams can be interpreted as variable-length nested lists where each stop token represents a parenthesis. The data closest to the arrowhead is sent first, and the done (D) token

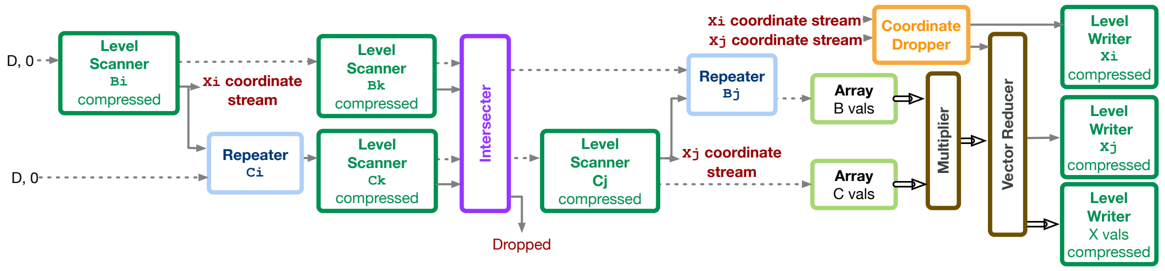


Figure 4.2: The SAM dataflow graph for sparse matrix multiplication $X_{ij} = \sum_k B_{ik}C_{kj}$, on DCSR matrices with linear combination of rows ($i \to k \to j$ order). Stipled, solid, and double arrows resemble reference, coordinate, and value streams respectively.

terminates the stream. Thus, the value stream in Figure 4.1d,

$$\overleftarrow{1,\ \mathrm{S}_0,\ 2,\ 3,\ \mathrm{S}_0,\ 4,\ \ 5,\ \mathrm{S}_1,\ \mathrm{D}}$$

represents the nested value level

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

4.5 Core Primitives of Our Sparse Dataflow Abstraction

We will use the linear combination of rows algorithm (sometimes referred to as Gustavson's algorithm [75]) for sparse-matrix sparse-matrix multiplication (SpM*SpM) to illustrate the operation of SAM blocks, and to demonstrate how their composition defines different algorithms. The Einstein summation notation for this algorithm is $X_{ij} = \sum_k B_{ik} * C_{kj}$, where the matrix multiplication is accomplished by using an index order of $i \to k \to j$ [246]. The advantage of this iteration order is that k coordinates are first intersected and only those ks that survive result in further computation.

Figure 4.2 shows the algorithm as a SAM dataflow graph. From the left, the coordinates of the two matrices are loaded from DCSR data structures in memory by level scanners. The coordinates are then transformed into a three-dimensional iteration space by chaining together the $i \to k$ coordinates of the B matrix with the $k \to j$ coordinates of the C matrix. This space requires duplicating data to fill in missing dimensions. In this example each matrix is broadcast over an index variable of the other matrix (B over j and C over i).

4.5.1 Tensor Iteration

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

Definition 4.5.1 (Level Scanner). A level scanner takes in a reference stream and outputs two streams: one of coordinates and one of references. It produces a single fibertree level on its output coordinate stream, fiber by fiber. Each non-control token on the input stream is a reference to a single fiber location for a given level in memory. The level scanner generates all coordinates in that

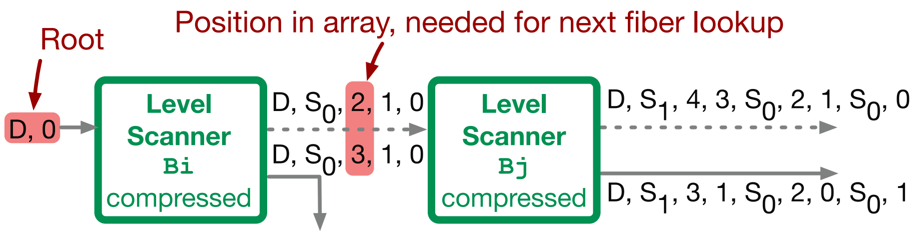

Position in array, needed for next fiber lookup
Root
D, 0
Level Scanner Bi compressed
D, S0, 2, 1, 0
D, S0, 3, 1, 0
Level Scanner Bj compressed
D, S1, 4, 3, S0, 2, 1, S0, 0
D, S1, 3, 1, S0, 2, 0, S0, 1

Figure 4.3: Composition of level scanner blocks.

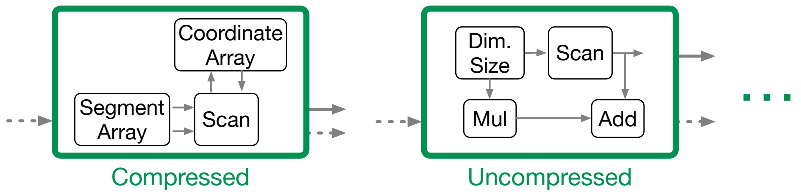


Figure 4.4: Implementations of the level scanner interface.

fiber, along with their corresponding references, and then adds an additional stop token to denote the end of the fiber.

Each SAM level scanner generates fibers for only one dimension. Therefore, multiple scanners must be composed to iterate over a multidimensional tensor. The composition uses the references emitted from each successive level scanner to locate the fibers of the next level scanner. The key to this composition is that level scanners communicate information by embedding both fiber location and coordinate hierarchy—needed by downstream level scanners—into the reference streams. Each level scanner adds a level to the hierarchy by either adding an S_0 stop token at the end of each scan or by incrementing all input stop tokens by one. They thus chain together to load an entire tensor and to convert it to per-level streams. Figure 4.3 shows two level scanners that iterate over the compressed matrix in Figure 4.1c. The reference stream emitted by the final-level scanner is sent to blocks that load values from memory, as described in Section 4.5.3. Each level scanner also connects to a memory array (Definition 4.5.5) that stores the fiber and coordinate information for the level, but these are not shown in figures to reduce clutter.

The SAM level scanners support iterating over tensors stored in various in-memory level formats presented in [41], which decouples an algorithm from the tensor formats. Thus, the interfaces of the level scanner are format agnostic and Figure 4.4 demonstrates how they remain unchanged as the level format implementation varies.

4.5.2 Stream Merging

Once the operand coordinate streams have been generated, the next task is to merge them. The index variables of a tensor index notation expression create an iteration space that we must cover, taking advantage of both the sparsity of the tensors and the mathematical properties of the operations to avoid unnecessary computation. Our design covers this sparse iteration space hierarchically by merging the coordinates of one dimension at a time, with the surviving coordinates from one dimension dictating what fibertree fibers need to be merged in the next dimension. The hierarchical merging is implemented with per-level merging blocks (intersection and union) and repetition machinery to handle the case where a tensor is broadcast [96, 79] across the dimension of another tensor, as required by our illustrative example in Figure 4.2.

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

Definition 4.5.2 (Intersecter). An intersecter has m pairs of coordinate and reference streams go in and one coordinate stream and m reference streams come out. It outputs coordinates and corresponding input references when all input coordinates are equivalent.

Definition 4.5.3 (Unioner). A unioner has the same input/output interface as the intersecter. However, it outputs coordinates and their associated input references whenever there exists at least one coordinate from any input. If the coordinate exists only on p inputs where $p < m$, the union block outputs an empty (N) token on the other $m - p$ output reference streams.

Figure 4.5 shows an example of a binary unioner that produces a coordinate stream that is the union of two input streams, along with the references from each input reference stream whose coordinates survived the union. Both emitted reference streams are augmented with empty tokens (N) to have the same shape as the emitted coordinate stream.

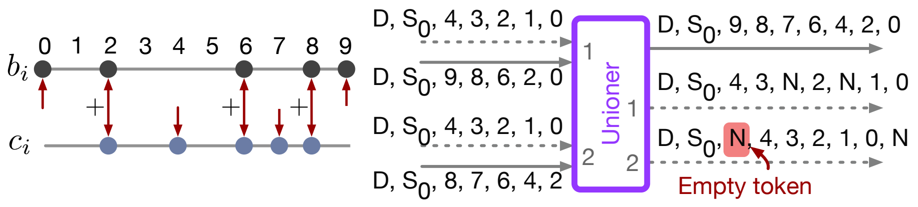


Figure 4.5: Example of union coiteration for $b_i + c_i$

As we saw in Figure 4.2, it is common for expressions to replicate one tensor across a dimension of another, often called array broadcasting. Figure 4.6 shows a simple vector scaling example. It demonstrates how the repeater block replicates a reference stream over every coordinate of the provided coordinate stream. The repeater is a new primitive that solves limitations on prior work architectures needing to pre-configure higher-order iteration counts [45, 81, 176].

Definition 4.5.4 (Repeater). Repeaters have one input coordinate stream, one input reference stream, and one output reference stream. Each non-control token in the input reference stream is repeated m number of times, where m is the number of non-control tokens from the input coordinate stream before a stop token is seen.

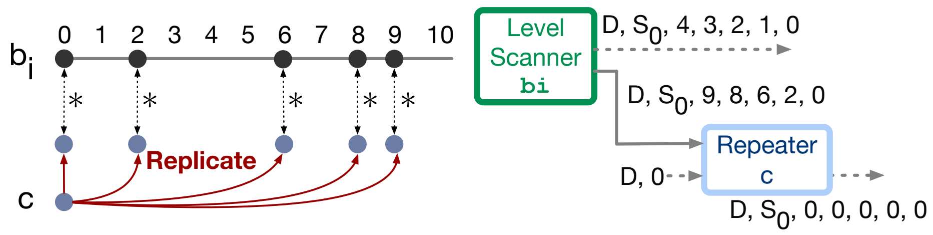


Figure 4.6: Repeating a scalar with a repeater block.

Hierarchical repeating and stream merging compose to express algorithms for multidimensional tensor contractions. Reconsider the linear combination of rows SpM*SpM algorithm from Figure 4.2. The i coordinates loaded from B are not only passed to the i level writer of X by way of the coordinate dropper, but also fed to a repeater that broadcasts all of C's k coordinates over each i.

4.5.3 Computation

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

Definition 4.5.5 (Array). An array block is a proxy for a memory interface and can be treated as a contiguous section of memory. It has two interface modes—load, which given one input reference stream fetches data to produce one output stream of any type, and store, which given one input reference stream and one input data stream of any type has a side effect that stores the data to its corresponding reference location in memory.

In SAM, arrays store values, coordinates, and references. In the computation pipeline, value streams are read from the array of each operand, with the same coordinates, and combined using streaming arithmetic-logic units (ALUs). The Figure 4.2 ALU is a multiply unit.

Definition 4.5.6 (ALU). An ALU block consumes two value streams and produces one value stream. It applies an arithmetic operator (add, subtract, or multiply) to inputs, treating empty tokens as zeros.

In addition to combining values at the same coordinate, often the algorithm needs to accumulate a tensor. In our illustrative example in Figure 4.2 this occurs at the end, where we sum over the k dimension (multiple dimensions can be summed by chaining reduction blocks). Reductions in tensor algebra may occur over any tensor dimension, independent of the order in which we choose to merge coordinates. Thus, summation reductions may occur over the coordinate level that is merged last (requiring a scalar to accumulate the result), over the coordinate-level merged second to last (requiring a vector to accumulate the results), or over coordinates merged earlier (requiring a higher-dimensional tensor to accumulate the results). SAM provides one block for reductions that must be configured for any specific dimension of accumulation.

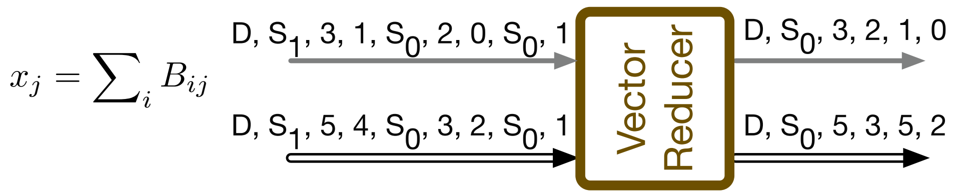


Figure 4.7: Example using the row reducer, where $n = 1$, to accumulate the columns of the matrix from Figure 4.1a.

Definition 4.5.7 (Reducer). A reducer is configured by n, the dimension of the memory needed in the reduction. It inputs and outputs n coordinate streams and one value stream. The block is sent an entire n-dimensional (sub-)tensor with repeated points/values and outputs streams that represent that tensor with unique coordinates and summed values. Specific reducers include: scalar where $n = 0$, vector where $n = 1$, and matrix where $n = 2$.

The reducer internally adds values corresponding to equivalent coordinate points and stores the results in an internal storage, which may be a dense or a sparse data structure. Finally, when an n-level reduction is completed, for example when a whole row has been processed for the Gustavson's algorithm in Figure 4.2, the reducer emits the resulting tensor as streams with deduplicated coordinates. When accumulating coordinates with empty fibers, resulting from ineffectual intersections, the reducer may be configured to either accumulate empty fibers into an explicit zero (the identity for addition) or to remove the empty fibers by removing their extra stop tokens. The choice is an implementation decision, but empty reduction behavior may affect SAM graph construction for other blocks (see Definition 4.5.9 and Table 4.3).

Like with level scanners, various implementations of the reducer are possible underneath the abstraction, including k-way merging, dense arrays, compressed data structures, and bitmaps [176, 145, 165]. Figure 4.7 shows an example of a row ($n = 1$) reducer.

4.5.4 Tensor Construction

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

Definition 4.5.8 (Level writer). Level writers take in either one value stream or one coordinate stream and store its contents to memory, internally generating reference information and auxiliary level data structures. As a result, the block is a wrapper around the store mode of a coordinate Array (and its metadata) or a value array. The level writer's internally generated references store the data tokens from the input stream in order.

In cases with at least one index-variable level above an intersection level, the result coordinate streams must be cleaned before the level writer stores it back to memory. The coordinate cleanup

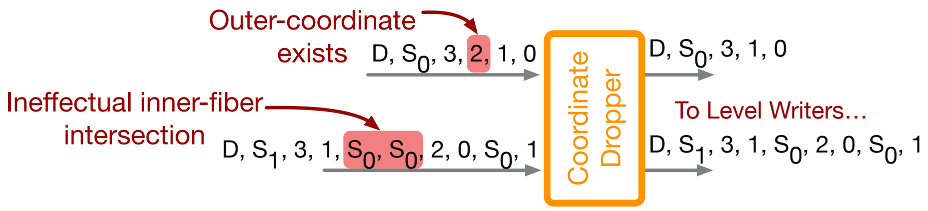

Outer-coordinate exists
D, S0, 3, 2, 1, 0
Ineffectual inner-fiber intersection
D, S1, 3, 1, S0, S0, 2, 0, S0, 1
Coordinate Dropper
D, S0, 3, 1, 0
To Level Writers…
D, S1, 3, 1, S0, 2, 0, S0, 1

Figure 4.8: Dropping coordinate 2 from the matrix in Figure 4.1a.

removes any outer-level result coordinates that have ineffectual inner-level intersections (either empty intersections or zero values) as shown in Figure 4.8. We introduce the coordinate dropper block to handle these cases. Coordinate droppers with value stream inputs are optional if explicit zeros need not be removed. In this case, other coordinate dropper blocks are also optional if the reducer block is configured to accumulate empty fibers into 0-values, since ineffectual (empty) intersections will produce explicit values.

Definition 4.5.9 (Coordinate Dropper). The coordinate dropper takes in one outer-level coordinate stream and one inner-level coordinate or value stream. It removes both the outer-level and inner-level tokens that came from ineffectual merging or computation (empty fibers or zeros) at the inner level.

4.5.5 Detailed Example of an Expression in SAM

Now that we have presented the core of SAM, we will continue to use our SpM*SpM running example to demonstrate how the primitives in SAM compose to implement arbitrary sparse tensor algebra expressions. However, for the example in this case, we will use SpM*SpM with the inner-product dataflow ($i \to j \to k$ order). As a reminder, SpM*SpM is represented in Einsum notation as $\mathbf{X}_{ij} = \sum_k \mathbf{B}_{ik}\mathbf{C}_{kj}$, and in our example $\mathbf{X}$ and $\mathbf{B}$ are stored in doubly compressed sparse row (DCSR) format and $\mathbf{C}$ is stored in doubly compressed sparse column (DCSC) format. Custard compiles the input program for this example (shown in Figure 3.1) to a SAM dataflow graph. Intermediate reference (ref) and coordinate (coord) streams are shown at each step of the SAM dataflow graph.

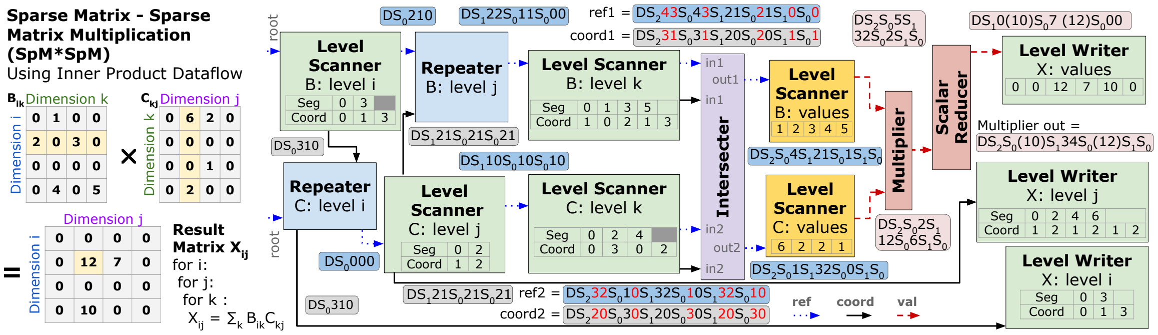


Figure 4.9: SpM*SpM ($X_{ij} = \sum_k B_{ik}C_{kj}$) with inner-product dataflow represented as a SAM graph annotated with example streams and tensor data. The SAM graph remains static for each scheduled expression, in this case inner-product SpM*SpM, even if the data changes within the streams and memory primitives.

First, we have two pairs of level scanners (B: level i and C: level j) and repeaters for the outer dimension of each matrix. The level scanner/repeater pairs produce replicated references indicating nonzero rows/columns (in matrix multiplication, B is replicated for each j and C for each i). Next, those replicated references are fed into level scanners for the shared dimension (k) of each matrix. These level scanners produce reference-coordinate pairs that are intersected to determine the values that need to be multiplied. The resulting intersected pairs are then used to read values, multiply, and reduce them, before, finally, a level writer writes the output values to memory. We will use this example again later in Chapter 5, specifically Sections 5.3 and 5.4).

4.5.6 Alternatives and Tradeoffs Discussion

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

Stream Control Tokens

Section 4.3 presents a solution for streaming dataflow where the control tokens (stop, empty, and done) are directly passed through the data plane, but alternative solutions to control flow for dataflow exist. Other valid dataflow control-flow solutions include control signaling on dedicated control-only streams, embedding control (via counters and control tokens) directly into each primitive, and having a completely separate control plane. We give examples, discuss the limitations of these approaches, and justify putting control tokens on the data plane.

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

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

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

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

Level-Based Stream Representation

Another approach to our level-based streaming tensor representation would be a less efficient point-based streaming representation. One implementation could stream flattened tensor point tuples with no control tokens. The tensor from Figure 4.1 could thus be represented as

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

In this representation, the number of processed stream tokens for identity matrices is $3 \cdot \text{nnz}_B$, where nnz_B is the number of nonzeros in B. We can compare the two representations to find when the point-based representation has more tokens using the equation $3 \cdot \text{nnz}_B > (1+c) \cdot \text{nnr}_B + 2 \cdot (1+c) \cdot \text{nnz}_B$ where c is the fraction of control tokens and nnr_B is the number of nonempty rows in B. Using worst-case numbers from our analysis in Figure 4.15, we rewrite the equation to $3 \cdot \text{nnz}_B > 1.3326 \cdot \text{dim}_{B_i} + 2 \cdot 1.3326 \cdot \text{nnz}_B \implies \text{nnz}_B > 3.98 \cdot \text{dim}_{B_i}$ where dim_{B_i} is the number of rows in B. The result demonstrates that our level-based representation, in the worst-case, processes less tokens than the point-based approach when there are on average more than 4 elements per row. Of the matrices we selected in Figure 4.15, all 5 middle 50 and 5 large 50 matrices satisfy the $4\times$ inequality and are more efficient in our level-based representation. Our approach becomes even more efficient for higher-order tensors. The coordinates at every level are expanded to the last level—proportional to roughly $O(n^N)$ instead of $O(n^2)$ for matrices, where n is a single tensor dimension and N is the tensor order—to produce the tensor point tuples.

4.6 Representing Optimizations in Sparse Dataflow

The core SAM blocks introduced in Section 4.2 are complete in the sense that they compose to express every tensor algebra expression. Moreover, they suffice to express all coordinate processing (dataflow) orders and fusion—the primary tools to construct algorithms with good asymptotic complexity [4]. To express SAM graphs that further optimize performance and deal with finite hardware, we have added additional capabilities. These capabilities let the graphs express parallelism, tiling, and more ways to represent tensor information either in memory or as streams. In this section, we discuss how SAM extends to include these additional optimizations and how they compose with the core SAM from Section 4.2.

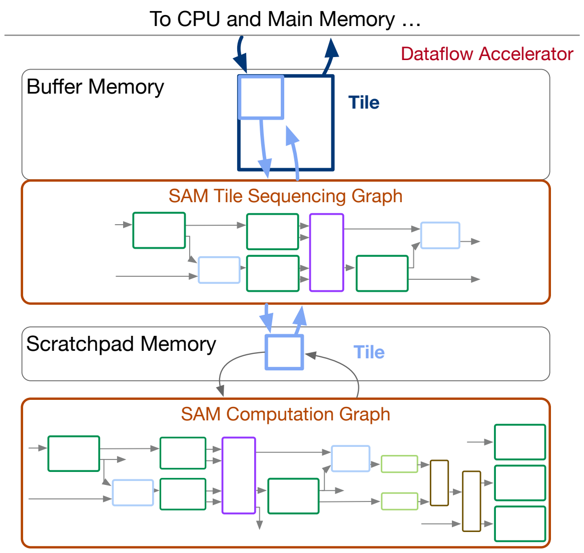


Figure 4.10: How to sequence tiled tensors (shown in blue) to fit in finite memory for SpM*SpM. The SAM computation graph is the same as in Figure 4.2, and the SAM tile-sequencing graph performs coiteration and merging of tile coordinates.

4.6.1 Tiling and Blocking

In our data model, tiling a tensor splits a single fibertree level into multiple levels and then reorders those levels to produce smaller sub-tensors (tiles). Figure 4.10 shows how SAM can sequence tiled tensors between host and accelerator devices for computation with fixed-size memories. SAM graphs are used in outer levels to sequence the tile coordinates (tile IDs) for reuse and in the inner levels to perform the computation. The tile sequencing is equivalent to tensor iteration (Section 4.5.1) and stream merging (Section 4.5.2), where tile IDs are coordinates and the values are references to the next level of tiles. As in ExTensor [81] and Capstan [176], we assume that tensors are tiled beforehand so that each tile fits in the dataflow accelerator's memory hierarchy. We demonstrate in Section 4.8.4 SAM's ability to fit tensors into finite memories and the tradeoff space of different memory configurations (dictated by architectural and implementation-specific memory configurations, like the maximal tile size and bandwidth at each level of the memory hierarchy).

In addition to tensor tiling, the techniques developed for SAM naturally extend beyond fully unstructured sparsity: they also apply to block-sparse tensors. Because SAM's primitives operate over hierarchical coordinate streams and compose at fiber boundaries, a block-sparse tensor simply appears as a higher-level sparse index that references dense sub-trees (blocks). Follow-on work extending SAM to sparse machine learning applications using a compiler for dataflow fusion implements these block-sparse ideas as performance optimizations [122].

4.6.2 Tensor Locating

In Section 4.2, all intersections are performed using coiteration, where the coordinates are intersected using a two-finger merge strategy. This is sufficient for computational correctness, however, it can be asymptotically inefficient. (The core SAM has to coiterate between an uncompressed dense counter level and a compressed level even though it is sufficient to iterate through just the sparse level.) We can often improve intersection efficiency if one tensor has far fewer elements than the other. Rather than waiting for the larger tensor to stream all its level coordinates, it can be more efficient to ask the larger tensor if it contains any of the coordinates from the smaller tensor. This operation, known as iterate-locate or leader-follower intersection, is possible with another SAM block that uses a coordinate instead of a reference to index an array.

Definition 4.6.1 (Locator). A locator takes in one coordinate and reference stream and outputs one coordinate and two reference streams. For each coordinate, the block finds the associated reference within an array block, if it exists, and outputs that reference and the input coordinate and reference. Otherwise, it emits an empty fiber on all streams.

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

Locators can also speed up intersections when used in conjunction with intersecters that communicate information back to level scanners about coordinate ranges that are no longer needed. This optimization, called coordinate skipping or galloping, is common in software and has also been proposed in hardware [81]. In coordinate skipping, the intersecter sends a signal back to the trailing level scanner (extending the interface of both blocks from the definitions in Section 4.2), informing it of the coordinate that is needed next. The level scanner, in conjunction with a locator, then skips ahead to this coordinate and avoids sending useless coordinates between its current coordinate and the coordinate sent by the intersecter.

4.6.3 Bitvectors

Bitvectors are a natural way to compress coordinate information since bits are easy to implement in hardware. Bitvectors have a 1 in positions where explicit coordinates exist and a 0 for empty (or zero) coordinates. Bitvectors have a pseudo-dense iteration space—one that iterates proportional to some constant factor of the dense dimension of each tensor level. This iteration is usually asymptotically worse in performance when compared to compressed iteration, especially with increasing sparsity. However, bitvectors may also increase efficiency since an n-bit bitvector, encoding n coordinate elements can be processed in one cycle. In some prior-work hardware like Capstan [176] and SIGMA [169], bitvectors are the only compression protocol. Bitvectors may also be offered in addition to compressed coordinates, with blocks that convert between their stream protocols. We introduce a bitvector converter that transforms a coordinate stream into a new bitvector stream protocol and describes a level scanner for the bitvector level format.

Definition 4.6.2 (Bitvector Converter). Bitvector converters transform b coordinates from the input coordinate stream into a single bitvector token of b bits on the bitvector stream output. Each bit indicates whether it has children or whether its sub-tree is empty.

The SAM bitvector level scanner is similar to Definition 4.5.1, but it outputs a bitvector stream instead of a coordinate stream. The bitvector level scanner also changes the reference stream behavior presented in Section 4.4. Consider the b vector from Figure 4.6 that produces the coordinate stream $\underrightarrow{D, S_0, 9, 8, 6, 2, 0}$ compressed as the 4-bit bitvector stream $\underrightarrow{D, S_0, 0011, 0100, 0101}$. As a reminder, the compressed level scanner would output a reference stream of $\underrightarrow{D, S_0, 4, 3, 2, 1, 0}$ to indicate contiguous references (positions) in memory. However, the bitvector level scanner instead produces the reference stream $\underrightarrow{D, S_0, 3, 2, 0}$ that sums bitcounts (popcounts) to find the positions in memory for the next level. The bitvector format thus demonstrates how SAM handles various stream types as different compression protocols on the wires (coordinates versus bitvectors), along with various reference stream protocols, while maintaining composability.

4.6.4 Parallelization

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

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

4.7 Compiling Sparse DSLs to Our Dataflow Abstraction

The Compiler for Unified Sparse Tensor Algebra Reconfigurable Dataflows (Custard) is our compiler to SAM, where SAM acts as an intermediate representation in the compiler flow. Custard compiles tensor algebra expressions with associated data structure specifications [41] and schedules [106, 181] to SAM dataflow graphs (see Figure 4.11). Custard is an open-source C++ project that utilizes the TACO front-end [108] but supplies a new lowerer and code generator. Custard uses TACO's three input APIs (tensor index notation, a format language, and a scheduling language) and the code that

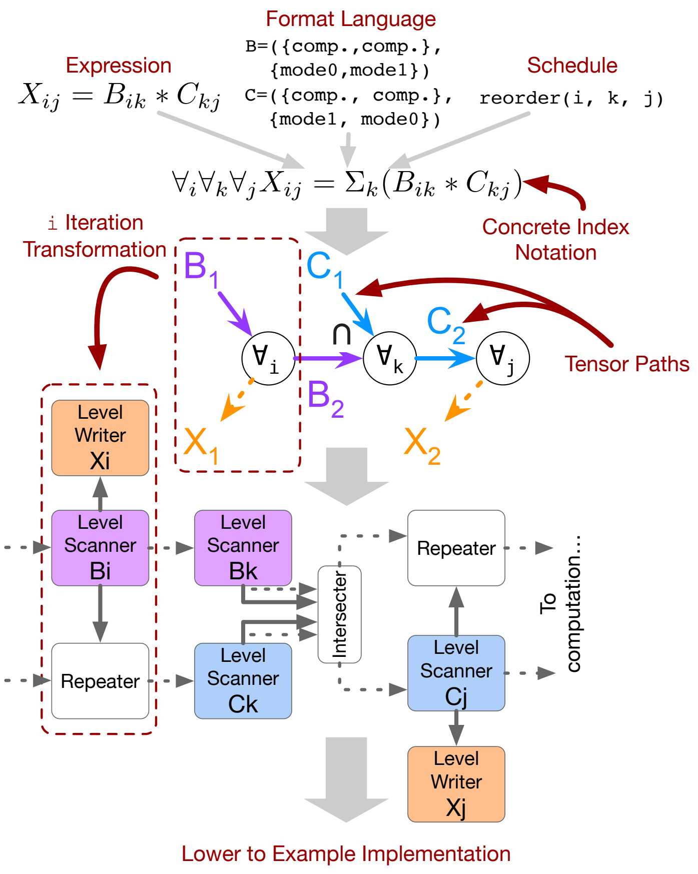


Figure 4.11: Custard's steps for compiling SAM tensor iteration, merging, and construction for the SpM*SpM example in Section 4.5. We abbreviate compressed as comp. From top to bottom, Custard uses the TACO input APIs to generate concrete index notation, creates index-variable paths for each tensor, and constructs the partial SAM graph (where the color of each block corresponds to a tensor path—purple for B, blue for C, and orange for X). Custard lowers the dotted red region of the tensor paths to the dotted red region of SAM blocks.

transforms these into the high-level concrete index notation (CIN) IR—an abstract loop nest with scheduling information shown in Figure 4.11. An example input program to Custard is at Figure 3.1, which is equivalent to TACO with a limited set of scheduling commands, and the syntax for CIN can be found in Figure 3.3. The final output of Custard is a generated SAM dataflow graph along with its cycle-approximate simulation. The automatic binding from these dataflow graphs to real hardware backends is described in Chapter 5. We leave the automatic binding from SAM to multiple hardware backends, including the prior-work accelerators described in Section 4.8.5, as future work.

Figure 4.11 illustrates a partial compilation to the SAM dataflow graph for the ikj-order SpM*SpM example from Section 4.5. Custard converts the concrete index notation to a graph that represents each tensor's path through the index variables (shown as colored arrows with tensor labels in Figure 4.11). Custard then builds the following three sections in order: tensor iteration and merging, computation, and tensor construction. It builds the tensor iteration and merging by iterating over the Cartesian product of index variables and input tensors, which in our example is $\{i, k, j\}$ by $\{B, C\}$. For every index variable in a tensor's path, Custard places and connects a level scanner, which we color corresponding to its associated tensor path. For every index variable absent from a tensor's path that does not have an outer index variable reduction, Custard inserts a repeat block. Finally, if multiple tensor paths exist for an index variable, then Custard inserts an intersecter (for multiplication) or unioner (for addition). Next, the output reference streams from the first part are connected to the compute tree, which consists of scalar operations and reductions, (extracted from the concrete index notation). Finally, the output values from the computation section and each index variable's final coordinate stream are connected to the output construction blocks (denoted by the orange in Figure 4.11) with coordinate drop blocks inserted as necessary.

4.8 Dataflow Abstraction Evaluation

We use Custard to compile disparate sparse tensor algebra algorithms into SAM graphs that are automatically lowered to a cycle-approximate functional simulator. The Custard code, used to automatically compile SAM graphs, is compiled using GCC 9.4.0. The SAM lowering and SAM simulator are written in Python 3.8. Our SAM simulator tracks each cycle iteration and models SAM graphs as fully pipelined (i.e. every primitive produces one token each cycle). It is cycle approximate since we assume for this section—except *Modeling Hardware with Finite Constraints* in Section 4.8.4—that: input queues are infinite, data fetched from arrays (memory) take only one cycle, memories are pre-initialized, and primitives are not time-shared. These assumptions do not affect our evaluation conclusions since this section only contains comparisons against simulator cycles. All benchmarks are run on a 2.2 GHz Intel Xeon Silver 4214 24-core CPU with a 16 MB LLC running Ubuntu 18.04.

4.8.1 Empirical Study of the Generality of SAM

We demonstrate the generality of SAM by generating dataflow graphs for a wide range of useful sparse tensor algebra expressions, shown in Table 4.2, including all expressions used in the TACO paper [108]. These real-world applications consist of algorithms such as factor analysis (e.g. alternating least squares), graph convolutional networks, and tensor factorization and decomposition for domains like machine learning, data analytics, and scientific computing. Table 4.2 lists the sparse tensor algebra features used by each expression and the number of primitives it uses (empty cells denote a primitive is not used). We see the primitive counts for level scanners and writers are higher, since they are used for tensor iteration and correspond to the tensor orders of all inputs and outputs respectively. The primitive composition counts also show that most blocks are uniformly used. It is interesting to note that two expressions use all primitive types. In addition, we automatically lowered all graphs to our simulator and checked for functional correctness on the set of all real and integer SuiteSparse matrices [49] and FROSTT tensors [190] that fit into memory and the Facebook tensor [221].

Table 4.1: A wide range of real-world expressions used to evaluate SAM. Each expression also contains a breakdown of the sparse tensor algebra features it contains. All features are obtained assuming the expression uses an alphabetical dataflow index-variable ordering except in SpM*SpM.[b]

Name	Expression	Sparse Tensor Algebra Feature					
		Out order	Input order	Num. inputs	Reduce order	Broadcast	Op
SpMV	$x_i = \sum_j B_{ij} c_j$	1	1,2	2	0	✓	*
SpM*SpM	$X_{ij} = \sum_k B_{ik} C_{kj}$	2	2	2	0–2 [b]	✓	*
SDDMM	$X_{ij} = \sum_k B_{ij} C_{ik} D_{jk}$	2	2	3	0	✓	*
InnerProd	$\chi = \sum_{ijk} B_{ijk} C_{ijk}$	0	3	2	0	×	*
TTV	$X_{ij} = \sum_k B_{ijk} c_k$	2	1,3	2	0	✓	*
TTM	$X_{ijk} = \sum_l B_{ijl} C_{kl}$	3	2,3	2	0	✓	*
MTTKRP	$X_{ij} = \sum_{kl} B_{ikl} C_{jk} D_{jl}$	2	2,3	3	0	✓	*
Residual	$x_i = b_i - \sum_j C_{ij} d_j$	1	1,2	3	0	✓	*,-
MatTransMul	$x_i = \sum_j \alpha B^T_{ij} c_j + \beta d_i$	1	0-2	5	1	✓	*,+
MMAdd	$X_{ij} = B_{ij} + C_{ij}$	2	2	2	×	×	+
Plus3	$X_{ij} = B_{ij} + C_{ij} + D_{ij}$	2	2	3	×	×	+
Plus2	$X_{ijk} = B_{ijk} + C_{ijk}$	3	3	2	×	×	+

[b] In SpM*SpM we show the features and primitive counts for all dataflow orderings: inner product, linear combination of rows, and outer product.

4.8.2 Ablation Study on the Utility of SAM Blocks

Each SAM block in Chapter 4 is essential for expressing the domain of sparse tensor algebra for dataflow. In order to demonstrate the usefulness of each primitive, we conduct an analysis on which sparse tensor algebra algorithms cannot be computed without each SAM primitive. Concretely, we analyze the entire set of algorithms input by users into the TACO website, provided by the TACO authors, and show which algorithms are not expressible if a given SAM primitive does not exist. We use the TACO website as our dataset since it is representative of real-world sparse tensor algebra computations. From the website, users have successfully compiled 23,794 sparse tensor algebra algorithms to date—of which 3,839 were distinct algorithms (unique combinations of expression and format) and 1,745 were unique solely in expression. Table 4.3 shows that removing any SAM primitive limits the expressible algorithms in the domain of sparse tensor algebra. Most blocks are used for most applications, and, moreover, full algorithmic generality requires all the primitives presented in Chapter 4.

4.8.3 Asymptotic Tradeoff Analysis

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

Dataflow Ordering

The index-variable order avoids different data-dependent asymptotic behaviors as shown by Section 3.7 and others [4, 233] and allows for generality in the execution of a particular dataflow algorithm. We

Table 4.2: SAM primitive counts for a range of real-world expressions in Table 4.1, demonstrating the expressiveness and generality of SAM. All primitive counts are obtained assuming the expression uses an alphabetical dataflow index-variable ordering except in SpM*SpM.[b]

		SAM Primitive Composition (count)								
Name	**Expression**	**Lvl Scan**	**Repeat**	**$\cap$**	**$\cup$**	**ALU**	**Reduce**	**Crd Drop[a]**	**Lvl Wr.**	**Array**
SpMV	$x_i = \sum_j B_{ij} c_j$	3	1	1		1	1	1	2	2
SpM*SpM	$X_{ij} = \sum_k B_{ik} C_{kj}$	4	2	1		1	1	0–2 [b]	3	2
SDDMM	$X_{ij} = \sum_k B_{ij} C_{ik} D_{jk}$	6	3	3		2	1	2	3	3
InnerProd	$\chi = \sum_{ijk} B_{ijk} C_{ijk}$	6		3		1	3		1	2
TTV	$X_{ij} = \sum_k B_{ijk} c_k$	4	2	1		1	1	2	3	2
TTM	$X_{ijk} = \sum_l B_{ijl} C_{kl}$	5	3	1		1	1	3	4	2
MTTKRP	$X_{ij} = \sum_{kl} B_{ikl} C_{jk} D_{jl}$	7	5	3		2	2	3	3	3
Residual	$x_i = b_i - \sum_j C_{ij} d_j$	4	1	1	1	2	1	1	2	3
MatTransMul	$x_i = \sum_j \alpha B^T_{ij} c_j + \beta d_i$	4	4	1	1	4	1	1	2	5
MMAdd	$X_{ij} = B_{ij} + C_{ij}$	4			2	1			3	2
Plus3	$X_{ij} = B_{ij} + C_{ij} + D_{ij}$	6			2	2			3	3
Plus2	$X_{ijk} = B_{ijk} + C_{ijk}$	6			3	1			4	2

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

Table 4.3: The number of sparse tensor algebra algorithms from the TACO website that are not expressible if a SAM primitive is removed. The **All** and **Unique** columns analyze all input algorithms and distinct algorithms, respectively.

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

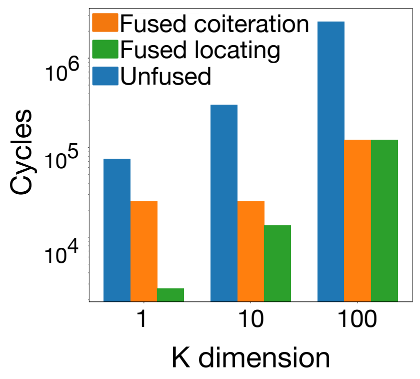


Figure 4.12: Performance of fused and unfused SDDMM algorithms.

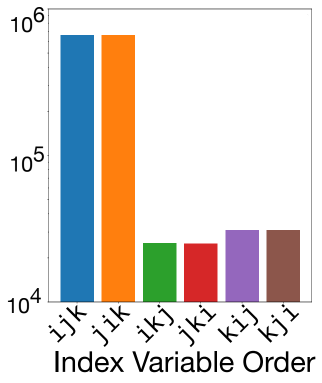


Figure 4.13: Performance of SpM*SpM dataflows.

simulate all six permutation orders of ijk for the SpM*SpM expression using two distinct 95% sparse uniformly random matrices with different dimensions of sizes $I = J = 250$ and $K = 100$. Figure 4.13 shows the inner-product algorithms (ijk, jik) perform the worst for matrix multiply. The linear combination of rows (ikj, jki) and outer product (kij, kji) algorithms perform at least an order of magnitude better. The performance is dictated by the order of k since coordinates are filtered out (intersected) at k earlier in the dataflow before repeating along the other dimensions i, j. These algorithms differ in their asymptotic complexity [110, 4], so performance differences will increase with increases in sparsity. However, the inner-product algorithm may be more efficient with other data and uses asymptotically less memory for the reduction (a scalar instead of a row). Since the efficiency choice is a tradeoff, sparse hardware should support many processing orders.

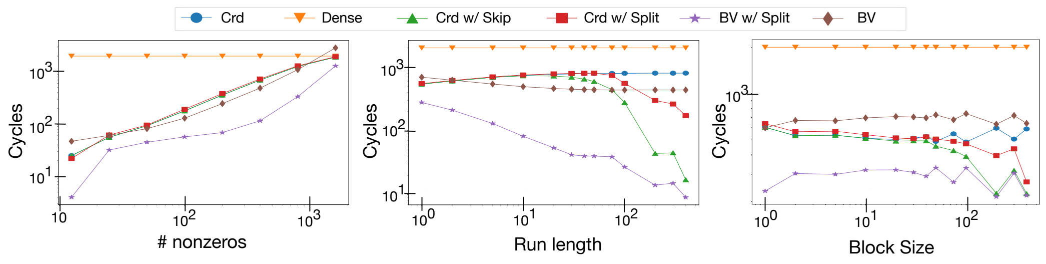
Fusion

We demonstrate the algorithmic performance advantage of fusion using a common expression from machine learning, the ijk-ordered SDDMM $X_{ij} = \sum_k B_{ij}C_{ik}D_{jk}$ [61, 22]. We generate a 95% sparse uniformly random matrix along with two dense matrices of dimensions $I = J = 250$ with a sweep of $K = \{1, 10, 100\}$. Figure 4.12 shows that the unfused implementation performs far worse, since calculating the entire dense matrix multiplication is costly with mostly wasted work. Given the number of nonzeros in B as nnz_B, the unfused computation complexity is proportional to $\max(\text{nnz}_B * K, \text{locate}(\text{nnz}_B))$, while the cost of factorization becomes $I * J * K + \text{locate}(\text{nnz}_B)$. The only case where we would want to factorize this expression is when the matrix B is almost fully dense and we have very efficient dense matrix multiplication hardware. But for a sufficiently sparse matrix, a fused expression will perform far better. Efficient sparse hardware should therefore support fused expressions.

We further enhance performance by using locator blocks (Section 4.6.2) to find the sampled i, j values, which is trivial in a dense array. Interestingly, Figure 4.12 shows that this advantage becomes negligible as K increases: iteration costs of the dense inner-product dimension k will dominate the computation time, hiding the benefits of locating during intersection. But locating provides significant performance gains when the amount of computation is modest, which is often true in sparse computations.

Accelerator Structures

We next explore different iteration acceleration techniques by comparing various configurations of coordinate-skipping (Section 4.6.2), bitvector iteration (Section 4.6.3), and iteration-splitting (Section 4.6.1). Figure 4.14 compares the performance when both vectors are in the following formats: one uncompressed level (Dense), one compressed coordinate level (Crd), one compressed coordinate level with coordinate-skipping (Crd w/ skip), two compressed coordinate levels (Crd w/ split), one pseudo-dense bitvector level (BV), and two bitvector levels (BV w/ split), also known as a bit-tree.



(a) Performance vs. sparsity of uniformly random synthetic vectors on a log-log scale

(b) Performance vs. run length of synthetic vectors with runs on a log-log scale

(c) Performance vs. block size of blocked synthetic vectors on a log-log scale

Figure 4.14: Simulated performance of various optimization techniques (compression, splitting, skipping, and bitvectors) for sparse vector sparse vector element-wise multiplication where the vectors have a dense dimension size of 2000.

For this set of experiments, we assume the coordinates were already split before this operation[1] and use the vector-vector element-wise multiply expression $x_i = b_i * c_i$ with both b and c as single dimensional vectors of size 2000. We use three types of synthetic vectors, namely *urandom*, *runs*, and *blocks*; runs and blocks are shown in Figure A.1. Vectors with *runs* are pairs of vectors where one vector will have longer stretches of nonzeros between the nonzeros of the other vector. Similarly, *blocks* are vectors which have dense blocks of nonzeros placed throughout the vector. For both these vectors, the number of nonzeros is 400 (20%) with the index indicating the size of the runs/blocks in each vector.

Figure 4.14a shows the performance as a function of sparsity for *urandom* data with bitvector bitwidth $b = 64$ and split factor (how many chunks the vector is divided up into) $s = 64$, where applicable, and shows the limitations of a single-level bitvector. As the sparsity increases, the compressed coordinate format becomes better than the bitvectors, since bitvectors are still a dense representation. The coordinate-skipping behaves exactly the same as the compressed coordinate format since *urandom* tensors on average have small (around 1.5) run lengths.

Figure 4.14b shows the utility of coordinate skipping and splitting. As run lengths increase, there are more opportunities to skip invalid input coordinates or avoid computation at the outer-level intersection. The bitvector remains flat since the number of nonzeros remains about the same for various run lengths. This advantage of skipping and splitting remains in the *blocks* case, without the dependence on block size, since intersections can also be dense. Overall, these results show the advantage of the implicit parallelism of bitvectors, but show that they need to be organized hierarchically for robust performance.

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

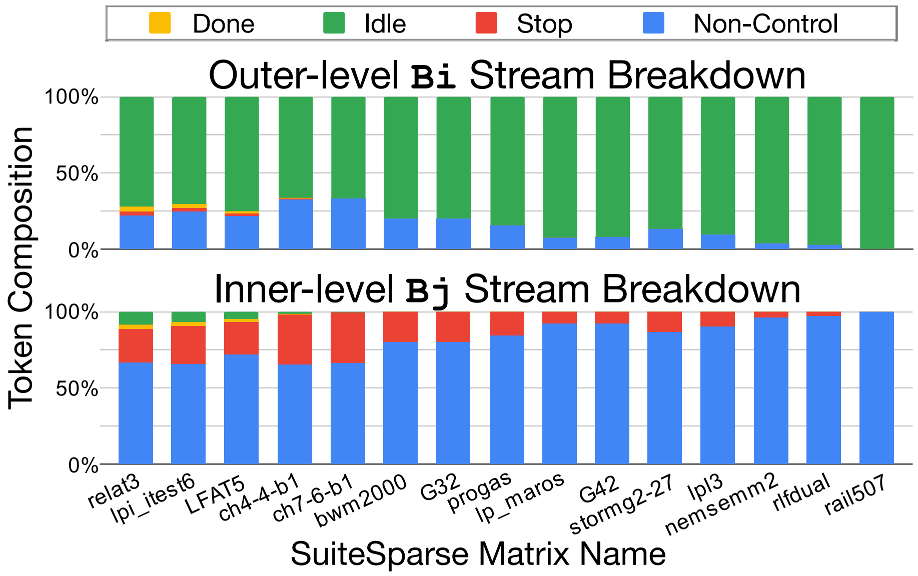


Figure 4.15: Breakdown of the outer B_i level stream and inner B_j level stream by token type for the matrix identity expression $X_{ij} = B_{ij}$ where B is a sparse DCSR matrix

4.8.4 Modeling Exploration

Stream Analysis

We analyze the token breakdown of the SAM flattened stream representation and identify that the stream control overhead is modest. We use Custard to compile the SAM graph for the matrix identity expression $X_{ij} = B_{ij}$, where B is a sparse DCSR matrix, and count the token types for each coordinate stream at the output of each level scanner. In our simulator, we model streams as Python lists and all control tokens as strings.[2] We run the expression on 15 matrices of various sizes from the SuiteSparse matrix collection [49] (see Table A.1 in the Appendix for matrix characteristics and selection criteria).

The control token overhead of our representation is reasonable, with an average non-idle control overhead reaching 0.95% for outer levels and 16.20% for inner levels as shown in Figure 4.15. (Section 4.5.6 shows the control overhead of the alternative of using non-flattened point streams would be higher.) The average inner-level percentage means that rows have an average of 5 nonzeros, an appropriate number of coordinates for this set of matrices. The outer-level B_i stream and inner-level B_j stream refer to the coordinate stream outputs of the first B_i level scanner and the second B_j level scanner, respectively. We do not show the B_{vals} breakdown since it is the same as B_j. Most tokens, on average 83.32%, on the B_i stream are idle since the B_i level scanner is in the done state while the inner-level iterates through its coordinates. This behavior occurs in compressed arrays,

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

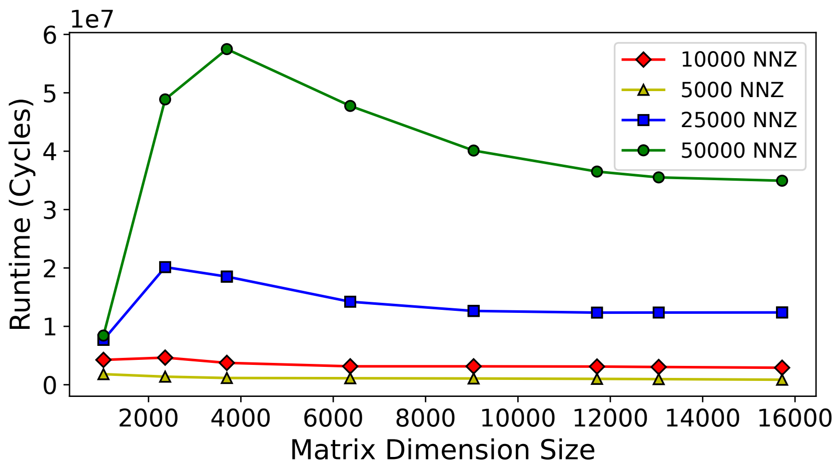


Figure 4.16: Recreation of ExTensor's "SpM*SpM performance across varying dimension sizes with a constant number of nonzeros per matrix" study using our SAM simulator.

as in Figure 4.1c, because there are exponentially more coordinates for each lower level of a tensor. The done state of the primitive is efficient as it is idle and avoids computation activity. Improving efficiency and utilization of idle primitives could include switching the outer level scanner to other tasks through time multiplexing, which we leave as future work. At the lower level of the matrix, control overhead is dominated by stop tokens. The stop token overhead ranges from 0.12% (for `rail507`) to 33.26% (for `ch7-6-b1`). Again, these breakdowns are reasonable since higher percentages of stop tokens occur only in small matrices.

Modeling Hardware with Finite Constraints

Although SAM is an abstract machine with infinite resources, it can also represent finite hardware with finite memory. ExTensor [81] is one design point in the space of sparse tensor algebra accelerators, and SAM is sufficiently expressive to model it. We find that SAM can recreate the performance characteristics of ExTensor's evaluation. We recreate the synthetic data study, Figure 19 Section 8.4, in the ExTensor paper that measures "SpM*SpM performance across varying dimension sizes with a constant number of nonzeros per matrix" as shown in Figure 4.16. Our SAM model contains the hierarchical coordinate skipping, fixed-memory tiling, sparse tile skipping, and n-buffering techniques of ExTensor. Our performance matches the three performance regions of the ExTensor study: increasing runtime due to more non-empty tiles at small dimensions, decreasing runtime due to sparse tile skipping, and decreasing runtime with saturating performance. Concretely, we model two levels of memory hierarchy, a last-level buffer (LLB) and a processing element buffer (PEB). SAM is used to sequence the sparse tile coordinates including the CPU loop nests and using a DRAM bandwidth of 68.256 GB/s, an LLB size of 17MB, and a processing element (PE) tile size of 128×128 (configured using implementation-specific information).

4.8.5 Backend Case Studies

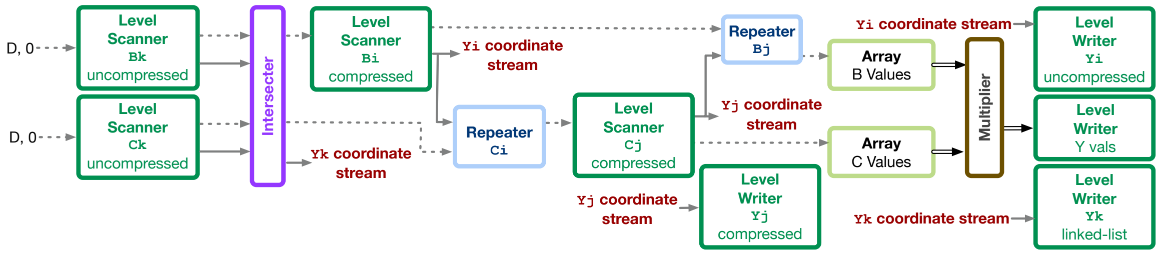


Figure 4.17: The SAM dataflow graph for SpM*SpM that represents the OuterSPACE multiply phase, which is followed by the OuterSPACE merge phase. Compare this outer product graph to the linear combination graph in Figure 4.2.

By construction, we designed SAM to easily represent dataflow hardware. To evaluate its likeness and ability to bind to hardware, we qualitatively analyze how SAM is able to represent fixed-function and reconfigurable dataflow backends including Gamma [246], OuterSPACE [157], ExTensor [81], and Capstan [176]. For example, Gamma's dataflow is similar to Figure 4.2. The main difference is that Gamma adds a parallelizer after the intersection unit and then uses a multi-input vector reducer to rejoin the parallel threads.

For space reasons, we only provide a concrete SAM graph for OuterSPACE (see Figure 4.17), which leverages an outer-product dataflow ($k \to i \to j$). We chose OuterSPACE because it factorizes SpM*SpM into two stages: a multiply phase ($Y_{ikj} = B_{ik}C_{kj}$) and a merge phase ($X_{ij} = Y_{ikj}$), thus showing how SAM supports factorization. For efficiency, B_{ik} and C_{kj} are respectively stored in column-major and row-major order. The first phase computes outer products between all columns of B and all rows of C and stores the partial result into a 3-dimensional tensor Y_{ikj}, as shown in Figure 4.17. To efficiently merge in the next phase, the intermediate result Y is stored in ikj-order, which is discordant with the dataflow kij. To efficiently support a discordant write of the tensor streams, OuterSPACE utilizes a linked-list representation as the level-format for k. Because our level writer is not restricted to a specific representation, SAM supports this dataflow. The merge phase (not shown to conserve space) then accumulates the partial product Y_{ikj} from the previous phase into a final result X_{ij}. This dataflow consists of three cascaded level scanners to generate the values Y_{ikj} that need to be summed, a vector reducer to sum the k dimension, and three level writers to store the X_{ij} results.

ExTensor's Stream Coordinator [81] is naturally representable with SAM. It is a hardware unit consisting of, in order, their Stream Sequencer, two of their Scanners, their Intersection unit, and two Data Storage units. The Stream Sequencer (and its `Configure()` command) is representable with two SAM repeater primitives, with the added benefit that SAM's repeaters do not have to be pre-configured. We represent their Scanner as a composition of n SAM level scanners with coordinate-skipping since their Scanner can scan through n levels of a fibertree. Finally, their

Intersect and Data Storage units are equivalent to our intersecter and array primitives. We note that each ExTensor Coordinator is fixed for two input tensors (hence two Scanners with Data Storage) with the intersection always occurring at the last level of each unit. These implementation choices limit Extensor-like SAM graphs to a subset of the entire space producible by SAM.

Finally, we analyze the Capstan [176] specialized loop-header hardware, since it is one of the main contributions of that work. Capstan can represent this specialized hardware as a two-operand bitvector scanner that can be configured with `or` or `and`. Using SAM, the hardware is equivalent to two bitvector level scanners (one for each tensor) followed by an intersecter for `and` or a unioner for `or`. The output values produced by the Capstan hardware: addresses for both tensors, a store address, and a dense index correspond to SAM's post-intersect/union reference streams, result level writer address generation, and post-intersect/union coordinate stream, respectively. The vectorized loop bodies in Capstan are representable using n-lane stream buses (bundles), SAM arrays to get the data, a single SAM ALU, and level writers. Again, the class of SAM graphs that represent a single Capstan loop-header is fixed to two sparse operands with only a subset of SAM's expressibility. Additionally, the Capstan-like SAM only iterates through bitvectors, which is great for vectorization but is fixed to a pseudo-dense iteration space.

Chapter 5

A Reconfigurable Accelerator for Sparse Tensor Algebra

"Facts do not cease to exist because they are ignored."

Aldous Huxley

The power of an abstraction lies in what it lets you ignore. In compiler design, a well-chosen intermediate representation (IR) does exactly this. Three-address code, for instance, allowed early CPU compilers to decompose complex instructions into a sequence of simpler ones operating on three operands. By assuming infinite registers, the IR abstracted away resource constraints and enabled clean, modular transformations. However, the assumptions made by abstracting away details need to be addressed eventually. Only at a later stage did the compiler resolve these assumptions through register allocation, the renaming of an infinite set of registers onto a finite number of them. This separation of concerns through abstraction simplified compiler design: the compiler can first focus on correctness and optimizing code transformations and later focus on enforcing resource limits.

Similarly, the Sparse Abstract Machine (SAM) ignores hardware resource constraints and assumes an unbounded set of dataflow units. This abstraction allows the Custard compiler to map any sparse tensor algebra expression into an infinite, composable dataflow graph. Yet, just as the illusion of infinite registers eventually meets real hardware, an end-to-end compiler must ultimately reconcile SAM to finite hardware. This chapter presents that reconciliation: a full compilation path from the SAM abstraction to a fabricated reconfigurable accelerator for sparse tensor algebra.

Programming abstractions and systems for hardware must ultimately tie back to real implementations. Before this dissertation, however, there existed no fully fabricated sparse accelerator. Prior architectures were either simulated, partially implemented, or specialized to narrow workloads (as described in Section 2.3). To validate SAM and the broader programming model on real hardware, we therefore developed the first fabricated accelerator for sparse tensor algebra computation **Onyx**. This accelerator is, to our knowledge, the first coarse-grain reconfigurable array (CGRA) capable of executing both sparse and dense tensor operations on the same fabric. As the first programmable accelerator to support arbitrary dense or sparse applications, Onyx also demonstrates an approach to accelerating several application domains on a unified hardware fabric.

Since SAM provides a composable language for describing sparse tensor algebra, it became a natural specification for hardware design. Rather than beginning from an arbitrary collection of features, SAM guided us toward a principled, minimal set of dataflow operations sufficient to express the infinitely large space of kernels. This generality aligned well with reconfigurable architectures. Leveraging existing CGRA infrastructure from prior work [14, 118, 59, 28], we evolved the design to support the domain of sparse tensor algebra with minimal manpower. The Onyx architecture was completed within a few months by only two designers.

The fabrication of Onyx has since influenced several generations of sparse accelerator design. Its architectural principles and sparse dataflow primitives have informed new work on VLSI implementations [120, 32], and have extended reconfigurable fabrics to sparse machine learning pipelines [32]. Two subsequent accelerators—**Opal** and **Zircon**—build directly on these ideas. Opal implements a VLSI realization of SAM's row-wise reduction dataflow, refines the sparse primitives introduced in Onyx, and extends the processing elements to support operations found in sparse neural networks. Zircon generalizes further: a heterogeneous CGRA integrating dense and sparse arrays to support hybrid sparse and dense machine learning workloads. Example workloads include computations found in multimodal pipelines, learning on visual data, graph learning, and sparse machine learning. Zircon's sparse compute fabric adds overlapping tile execution and hardware graph unrolling, further realizing key optimizations within the SAM abstraction.

Beyond its impact on hardware design, Onyx's expansion into the domain of sparse tensor algebra shaped research on CGRA hardware compilers and tooling. Its streaming sparse dataflow model and ready-valid interconnect directly influenced the design of Cascade [141], a CGRA pipelining compiler that generalizes to broader reconfigurable fabrics and (both static and dynamic) interconnects. The sparse workloads developed for Onyx likewise served as early drivers for Cascade's CGRA pipelining, timing, and debugging infrastructure.

Together, these systems demonstrate how the abstractions introduced in SAM and realized in Onyx have shaped both software and hardware for sparse computation. This chapter details the Onyx accelerator, the compilation path from SAM to Onyx, and the hardware design decisions that made the abstraction physically realizable.

5.1 The Need for a Fabricated Sparse CGRA

As discussed in Chapter 4, existing sparse accelerators target input data sparsity to bypass ineffectual computations [81, 176, 240]. However, real-world end-to-end applications consist of computations with both dense and sparse data regimes [48]. For example, a block-sparse implementation of a neural network layer must handle sparse data management in the outer dimensions and dense computation in the inner dimensions. A useful tensor algebra accelerator, therefore, must handle both dense and sparse tensor computations.

Previous approaches typically integrate distinct specialized (fixed-function) accelerators for different application regions on a single chip to target end-to-end applications. For instance, [92] incorporates distinct accelerators for convolutional neural networks (CNNs) and graph convolutional networks (GCNs). While these accelerators deliver high performance and efficiency for their multiple target applications, their lack of reusable building blocks means they support only a limited number of algorithms and often require hardware area that grows proportionally with the number of supported algorithms. Thus, they prove inefficient or incapable when applied to other workloads and become obsolete when improved algorithms emerge.

Programmable accelerators, conversely, can adapt to applications as they advance [70, 217]. These systems provide substantial performance and energy-efficiency gains compared to general-purpose computing platforms, yet have predominantly concentrated on dense data processing [59, 165]. Furthermore, of the numerous hardware accelerators referenced above that present architectural innovations, they often lack actual hardware implementations [81, 176, 240] and none have been physically fabricated [165]. The comparison table in Table 5.4 demonstrates that no fabricated programmable accelerator currently exists for sparse applications.

Consequently, Onyx addresses the shortcomings of existing accelerators by enabling both dense and sparse kernel execution on a single fabricated, programmable accelerator platform, as illustrated in Figure 5.1. End-to-end applications can be partitioned into sparse and dense regions of computation, with each region subsequently accelerated using Onyx. The system incorporates a coarse-grained reconfigurable array (CGRA) featuring composable tiles that enable support for tensor algebra expressions involving sparse and dense tensors, multiple inputs, higher-order tensors, and operation fusion. Beyond sparse tensor algebra, Onyx provides specialized support for image processing and machine learning workloads. This versatility is achieved through the following key contributions:

1. Composable memory primitives designed to store compressed representations of tensors with arbitrary dimensionality.

2. Composable computational primitives that eliminate ineffectual operations while supporting the complete spectrum of sparse tensor algebra operations.

3. A fully automated end-to-end sparse compiler that translates high-level tensor index notation, commonly known as Einstein summation (Einsum) notation, directly to hardware execution.

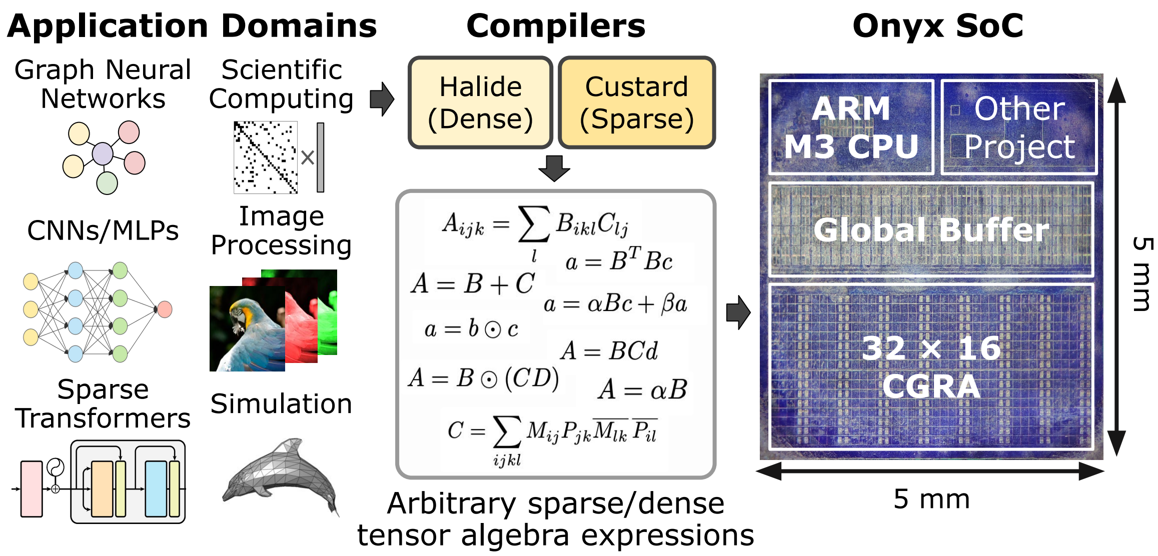


Figure 5.1: End-to-end applications are partitioned into dense and sparse kernels according to data sparsity characteristics. These kernels are subsequently mapped onto Onyx, which accelerates both dense and sparse tensor computation kernels. Onyx is, to the best of our knowledge, the first fabricated accelerator for both sparse and dense tensor algebra on a single fabric.

4. Compiler optimizations encompassing software pipelining, kernel unrolling, and tensor tiling to enhance the performance mapping of sparse tensor expressions.

5. Compute and memory optimizations for the dense tensor applications, which we do not include in this dissertation. For full details on these contributions refer to Kalhan Koul et al. "Onyx: A 12nm 756 GOPS/W Coarse-Grained Reconfigurable Array for Accelerating Dense and Sparse Applications". In: *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. 2024, pp. 1–2. DOI: `10.1109/VLSITechnologyandCir46783.2024.10631383`.

Onyx demonstrates up to 76% improvement in energy-delay product (EDP) compared to state-of-the-art programmable accelerators on dense applications and achieves up to 565× superior EDP performance versus CPUs equipped with dedicated sparse libraries on sparse workloads. This work establishes a comprehensive approach for accelerating entire *application domains* on unified programmable hardware platforms.

5.2 The Onyx Architecture

Onyx is a system-on-chip (SoC) illustrated in Figure 5.2 that integrates a 32×16 coarse-grained reconfigurable array (CGRA), a 4 MB global buffer (GLB), and an ARM Cortex-M3 processor [9]. The CGRA comprises 384 processing element (PE) tiles and 128 memory (MEM) tiles arranged in a three-to-one column ratio. Each PE tile consists of an arithmetic logic unit (ALU), a 64-byte register file (RF), and specialized primitives for sparse computation detailed in Section 5.3.3. Each MEM tile

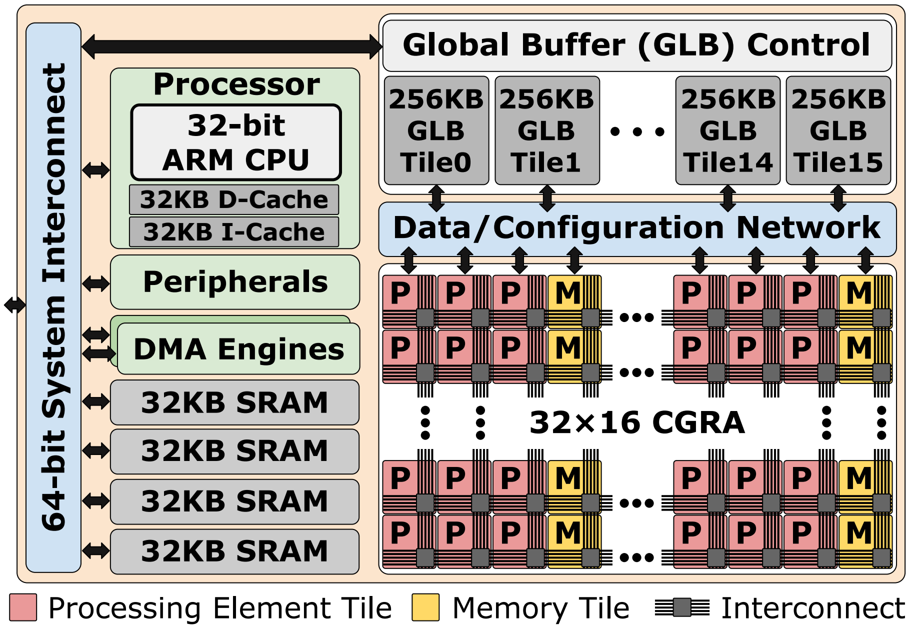


Figure 5.2: The Onyx SoC architecture.

incorporates a 4 KB SRAM alongside two memory controllers: one optimized for dense applications [118] and another for sparse applications, as described in Section 5.3.2. The tiles communicate through a statically configured interconnect that accommodates both static data movement for dense applications and dynamic data movement for sparse applications [142]. This interconnect routes 17-bit data signals via switch boxes (SBs) and connection boxes (CBs) embedded within each PE and MEM tile. Each tile's switch boxes direct outgoing data flow, while its connection boxes handle incoming data routing. The GLB consists of 16 GLB tiles, each featuring a bidirectional 16-bit connection to the CGRA. Within each GLB tile are two 128 KB SRAM banks, load and store units, and a specialized configuration network [114].

5.3 Sparse Reconfigurable Hardware

To accelerate arbitrary sparse tensor algebra, Onyx contains hardware implementations of the sparse acceleration primitives defined by the sparse abstract machine (SAM) described in Chapter 4. We design VLSI implementations of

1. sparse memory primitives that convert between compressed data structures in storage to compressed streams,

2. sparse processing (iteration and compute) primitives that eliminate ineffectual computation and support tensor algebra operations, and

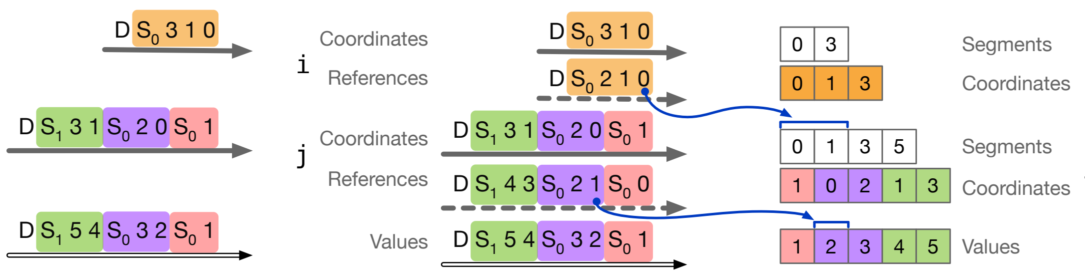


Figure 5.3: The Onyx data model: a recapitulation of the SAM streaming abstraction from Figure 4.1 augmented with reference streams and how they point (blue arrows) into arrays in memory.

3. a ready-valid interconnect that supports our compressed streams.

As a reminder, Figure 4.9 in Chapter 4 shows an example of how Onyx computes SpM*SpM. Similarly, using our composable primitives, we can accelerate any sparse tensor algebra expression on our CGRA.

5.3.1 Sparse Abstraction

We employ the same streaming tensor abstraction presented in Chapter 4 to represent tensors in Onyx. As a reminder, Figure 4.1 illustrates an example tensor alongside its fibertree format, SAM stream representation, and storage format as a doubly-compressed sparse row (DCSR) data structure.

A fibertree organizes tensors hierarchically, where each level corresponds to a tensor dimension and consists of fibers. Fibers represent groups of nonzero coordinates and are lists of one or more nonzero coordinates (highlighted in gray) paired with references to their payloads. The payload can be either another fiber at the next level or, at the lowest level, the actual tensor values. In the example shown, the fiber [0, 1, 3] at level i indicates that rows 0, 1, and 3 contain nonzero entries. Each coordinate in this fiber references its corresponding child fiber at level j.

Onyx hardware implements fibertrees as streams and storage structures, as detailed in Section 4.3 and Section 4.4. Like the sparse abstract machine, the system uses three stream types: coordinates, references, and values. To encode the hierarchical structure, streams incorporate special tokens: hierarchical stop tokens (S_n) mark boundaries between fibers at different levels, done tokens (D) signal stream termination, and maybe tokens (M) denote empty fibers. In storage, the fibertree abstraction is implemented as a data structure selected by the format language provided by the user. In Figure 5.3, the fibertree abstraction is implemented using a compressed sparse fiber (CSF) storage format, where fiber boundaries are implemented as segments that denote begin and end locations in the coordinate array. Figure 5.3 also provides more detail on how the references in the SAM stream representation, shown in blue, correspond to the next-level segments in the memory storage format.

5.3.2 Memory Tile Primitives

Each memory tile in Onyx stores one level of a fibertree. Memory tiles include three new sparse memory controller primitives as shown in Figure 5.4: the level writer, level buffer, and level scanner. Level writer and level scanner controller primitives are exact VLSI implementations of the abstract level writer and level scanner primitive interfaces and behavior defined in SAM in Section 4.2, respectively. The level buffer controller sequences reads and writes of varying sizes to the same memory tile, which is a simplified VLSI implementation of the buffet storage idiom [162]. To motivate the need for level buffers, we must remember that the SAM abstraction assumes infinite memory resources. Fundamentally, SAM level scanner and writer primitives assume the memories that they have access to are infinite with independent address spaces. However, real accelerator memories are finite and memories have both read and write capabilities. Furthermore, reads and writes to those memories (for example, when the output tensor for one kernel is used as the input to the next kernel) must be sequenced and synchronized. The level buffer, therefore, provides the necessary functionality to orchestrate data using finite scratchpad memories in an explicitly decoupled manner within the Onyx memory tiles.

The level writer receives coordinate streams and forwards them to the level buffer for storage. The level writer contains a coordinate address generator and segment address generator to compute storage addresses, along with a coordinate counter to track segment sizes. The coordinate counter and coordinate address increment with every data token input, while the segment address increments with every stop token input. A multiplexer, controlled by FSM logic, selects whether to write segments or coordinates into arrays in memory, with the address generators producing the appropriate write addresses for each array. As an example, consider the stream $\underrightarrow{D, S_0, 3, 1, 0}$ in Figure 5.4. The level writer routes these coordinates to the coordinate array while the coordinate counter simultaneously counts three data tokens and produces $[0, 3]$ for the segment array. When storing the last-level values of a fibertree, values replace coordinates in transmissions to the level buffer. In other words, all coordinate streams shown in Figure 5.4 are instead value streams.

The level buffer coordinates explicit-decoupled data orchestration (EDDO [162]) between the writes from the level writer and reads from the level scanner. Again, the level buffer either arbitrates both segment and coordinate arrays for most fibertree levels or a single value array when the input stream represents the last-level of a fibertree. The level buffer read–write arbitration uses a cache line of four-element words to speed up redundant and sequential accesses. A write state machine accumulates array data elements in a cache line during writes, executing grouped writes when the line fills. For incomplete lines at stream end, the state machine flushes the cache line with zero-padding. On read requests, the read state machine verifies whether the target address was accessed previously, utilizing the cached data if available or initiating a fresh read otherwise. Since the sparse memory tile is designed with one 512x64 single-port SRAM, an arbiter sequences all read and write operations to that port in request order.

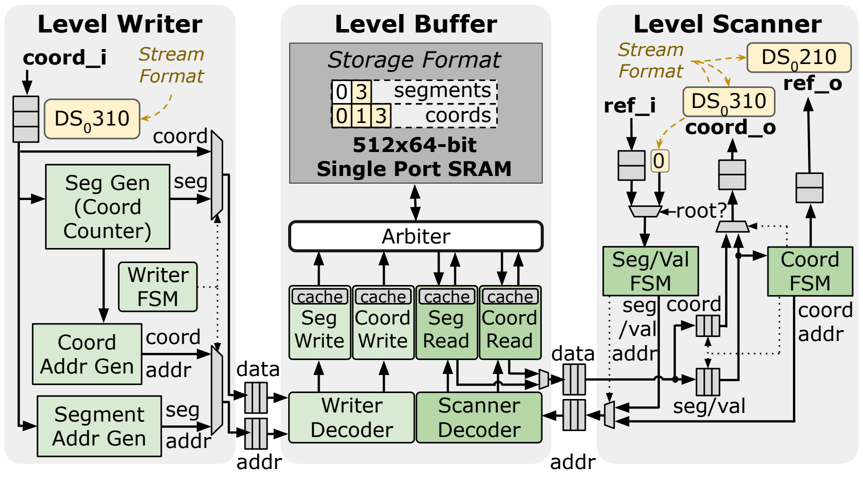


Figure 5.4: The sparse memory tiles in Onyx comprise three interconnected primitives: a level writer, a level buffer, and a level scanner. Dotted lines denote control and yellow boxes are example streams.

The level scanner accepts as input a reference stream identifying fiber(s) locations and outputs coordinate and reference streams for the next level. Each data token in the input reference stream indexes a fiber stored in the memory tile, pointing to the target segment location for retrieval. Upon receiving the reference stream, the Seg/Val state machine fetches the segment size from the level buffer and forwards that to the Coord state machine to retrieve the coordinates for each fiber. More specifically, for each data token in the reference stream, the Seg/Val state machine executes two segment reads and reserves a stop token for the output coordinate stream in a reservation buffer. For the reserved stop token, the Seg/Val state machine increments the stream level by one (i.e., an input stop token at level i becomes a reserved stop token at level $i + 1$). The Coord state machine, then, receives the segment values and uses a counter to iterate between them, generating read addresses for the coordinate array and queueing them in the coordinate FIFO. These coordinates form the level scanner output coordinate stream, with an address generator creating the matching output reference stream. At the highest tensor level, the system uses a root stream [0, D] to kick off the tensor streams. For other levels, upstream primitives provide the input reference stream. In the same example in Figure 5.4, the level scanner retrieves the segment array$[0, 3]$, determines that the fiber length is 3, then fetches coordinates $[0, 1, 3]$ from the level buffer for the coordinate stream. The reserved control tokens get inserted, leading to the final output coordinate stream of $\underrightarrow{D, S_0, 3, 1, 0}$, and the address generator simultaneously produces references for each coordinate, creating the reference stream $\underrightarrow{D, S_0, 2, 1, 0}$.

5.3.3 Processing-Element Tile Primitives

We added VLSI implementations of the coordinate merger (combined intersector and unioner), reducer, repeater, and coordinate dropper primitives from Chapter 4 to the processing-element tiles in Onyx. These primitives perform computation on streams of coordinates, references, and values to

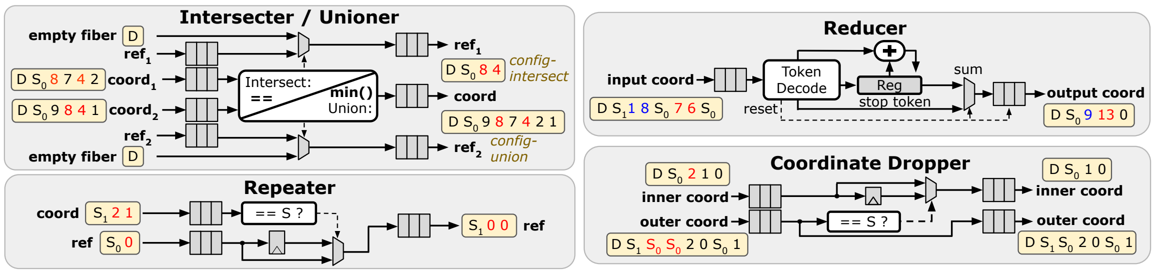


Figure 5.5: PE tile primitives: coordinate joiner (intersecter/unioner), coordinate dropper, reducer, and repeater.

eliminate ineffectual computation from the implicit zeros in sparse tensor computation. Figure 5.5 provides the implementation of each primitive along with example input and output streams to understand their behavior. The coordinate merger calculates the intersection or union of coordinate streams, depending on its configuration, of two tensor streams to produce only non-zero output indices. Specifically, a tensor stream refers to a coordinate and reference stream pair for a given level of a tensor. The merger in intersection mode performs a two-finger merge of the input streams based on the coordinates by dequeuing the FIFO with the lower coordinate value. It only emits coordinates, and their corresponding references, when both streams have equivalent coordinates. The merger in union mode uses the same hardware but collects and outputs all coordinates (merging duplicate coordinates). For any coordinates that only exist in one of the input coordinate streams, the merger in union mode also inserts empty-fiber tokens into the corresponding output reference stream which does not contain that coordinate. While complete, these primitives may produce empty fibers due to empty-fiber tokens or back-to-back stop tokens. The coordinate dropper, therefore, removes the coordinates associated with empty fibers by registering, looking for, and dropping contiguous stop tokens in the inner coordinate stream and its corresponding outer coordinate as well. The reducer primitive performs a sum over a stream, outputting a single value, and the repeater primitive duplicates streams for tensor broadcasting.

5.3.4 Ready-Valid Interconnect

The primitives above generate data-dependent access patterns that exhibit variable latency. To support these behaviors, we extended the baseline CGRA architecture [59] with ready–valid signaling and FIFOs using the hardware generation infrastructure from Melchert et al. [142]. Instead of assuming infinite input and output FIFOs for each primitive as in SAM, we performed an analysis and testing to ensure that every primitive can always make forward progress with a minimum of two-element FIFOs. A naive implementation would replace every pipeline register with a two-element FIFO, but this would double the register count. Instead, Onyx has a *SplitFIFO* architecture, which reuses pipeline registers in adjacent tiles by sharing control signals. The SplitFIFO avoids doubling the number of registers and saves 13% in interconnect area. In addition, the 16-bit interconnect has its most significant bit reserved to encode special tokens (e.g., `stop`, `done`, and `empty`). Finally, the SAM dataflow graphs may contain producer primitives that broadcast to multiple consumers. In hardware, this corresponds to one producer FIFO driving several consumer FIFOs over the interconnect. When any consumer is not ready, the producer must stall. At each switch box (SB) broadcast point, a logical `and` across all incoming ready signals will generate a single `enable` signal for each corresponding producer FIFO.

5.4 Compiling Abstract Sparse Dataflow to Hardware

Since SAM is designed to abstract away hardware implementation details, a back-end compiler from SAM to any real hardware must:

1. Perform SAM graph transformations based on hardware constraints.
2. Lower SAM graphs to hardware-executable dataflow graphs. This lowering step includes elaborating abstract SAM streams to multi-wire signals and expanding or collapsing abstract SAM primitives to a dataflow graph of hardware-defined primitives.
3. Map hardware-defined architectural primitives to CGRA tiles.
4. Generate tiled data to meet buffer capacity constraints.

Figure 5.6 provides an overview of our sparse application compiler, which accomplishes the above steps for the Onyx CGRA.

Our compiler builds on the Custard compiler presented in Section 4.7. As a reminder, Custard accepts as input the three high-level DSLs presented in Section 3.1 and lowers them to SAM dataflow graphs. The work in this chapter extends that flow to hardware by introducing an end-to-end compilation path from these high-level languages to a fabricated accelerator. We develop a new backend that lowers SAM graphs to CGRA bitstreams (Figure 5.6) through an Onyx-aware sparse dataflow IR that aligns with the hardware's implementation. The compiler maps sparse primitives from this hardware-oriented IR onto Onyx's tiled CGRA [118], performs placement, routing, and pipelining using the algorithm in Section 5.4.4 and presented by Melchert et al. [141], and emits the configuration bitstream. Finally, the compiler tiles and loads the input data as described in Section 5.4.2 before dispatching the tiled data to the configured array for execution.

5.4.1 Hardware-aware Transformations and Lowering

To produce dataflow graphs that execute correctly on Onyx, the backend compiler performs a series of dataflow transformations in two phases. First, the compiler applies a series of rewrites that transforms the SAM graph to fit hardware resource constraints. Then, the compiler lowers SAM to a lower-level Onyx-aware dataflow IR. This two-part structure is analogous to register allocation in traditional

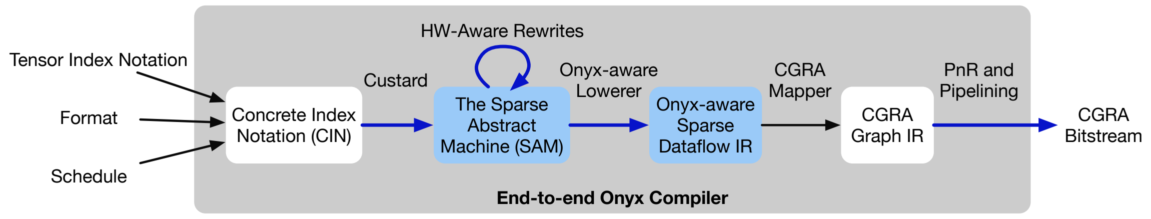


Figure 5.6: Our end-to-end sparse compiler that starts with a sparse tensor algebra application and outputs a CGRA bitstream. Blue denotes new contributions of this dissertation.

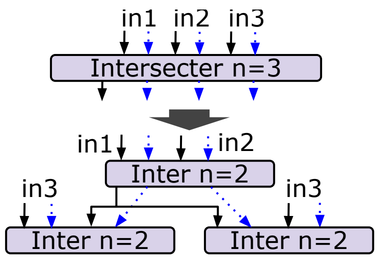
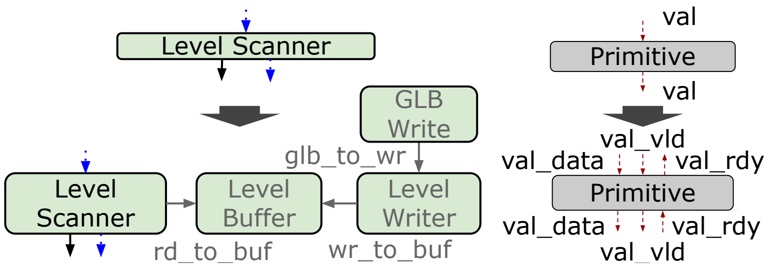


Figure 5.7: Hardware-constrained compiler rewrites in SAM: N-ary to binary joiner transformation.



(a) Memory primitive expansion (b) Signal elaboration

Figure 5.8: SAM to Onyx-aware lowering rewrites.

compilers. Register allocation is often designed as a variable renaming pass on three-address code that stays within the same IR, before lowering to a lower-level representation closer to machine code. In the same way, our first phase reshapes and normalizes the SAM graph under Onyx's resource constraints while remaining in the SAM abstraction, and only the second phase lowers that normalized graph into Onyx-compatible dataflow code.

All transformations occur through an ordered traversal of the dataflow graph, matching on sub-graph patterns. The ordered traversal starts from the SAM graph root primitive and follows the directed-acyclic graph (DAG) edges from each primitive to each neighboring primitive, marking them as visited along the way. During this traversal, the compiler pattern matches on the input sub-graphs for each rewrite and connects in the output sub-graph appropriately. Figure 5.7 and Figure 5.8 illustrate three types of dataflow rewrites from these two phases.

The compiler first rewrites the SAM graph to satisfy structural constraints that are resolved entirely within the SAM abstraction. These transformations preserve SAM's semantics but reshape the dataflow to patterns compatible with downstream hardware. The primary transformation is a primitive remapping for joiners. SAM allows arbitrary N-ary joiners, but we only implement binary intersect and union primitives in Onyx. The compiler therefore decomposes any N-ary joiner into a tree of binary joiners, as shown in Figure 5.7. The compiler also performs broadcast elimination at this stage. SAM has implicit broadcast points where streams fan out to multiple primitives. The compiler replaces these points with multiple point-to-point connections because broadcasting is handled in Onyx's interconnect and router by default.

After these transformations, the compiler specializes dataflow graphs specifically for the Onyx

Table 5.1: An example Onyx-aware dataflow IR neighboring constraint and signal elaboration for the level buffer (`buffer`) primitive.

Neighbor type	**Neighboring constraint**	**Signal elaboration (src.port → sink.port [width])**
SRAM Memory (`mem`)	Must connect the level buffer to exactly one on-chip SRAM *memory* tile. Provides address, data, and read/write-enable lines, plus response data and ready–valid handshake signals.	`buffer.addr_to_mem` → `mem.input_num_1/2` [16] `buffer.data_to_mem` → `mem.input_num_0` [16] `buffer.wen_to_mem` → `mem.input_num_1` [1] `buffer.ren_to_mem` → `mem.input_num_0` [1] `mem.output_num_0` → `buffer.data_from_mem` [16] `mem.output_num_1` → `buffer.valid_from_mem` [1] `mem.output_num_0` → `buffer.ready_from_mem` [1]
Level Scanner (`lvl_scan`)	At most one upstream level scanner may drive the buffer. The buffer consumes read-response streams and is driven by the read address, op, and ID streams from the read scanner.	`buffer.rd_rsp_data` → `lvl_scan.rd_rsp_data_in` [17] `lvl_scan.addr_out` → `buffer.rd_addr` [17] `lvl_scan.op_out` → `buffer.rd_op` [17] `lvl_scan.ID_out` → `buffer.rd_ID` [17]
Level Writer (`lvl_wr`)	At most one downstream write scanner may consume buffered writes. The write scanner drives data, address, and ID streams into the buffer.	`lvl_wr.data_out` → `buffer.wr_data` [17] `lvl_wr.addr_out` → `buffer.wr_addr` [17] `lvl_wr.ID_out` → `buffer.wr_ID` [17]
Other primitives	Disallowed: attempting to connect a level buffer directly to any of these node types raises a `NotImplementedError`.	

CGRA through lowering. The Onyx-aware IR closely resembles SAM but restricts the graph to the subset of dataflow programs that are realizable on Onyx. Each primitive in this IR carries explicit type constraints between neighboring primitives and defines those exact wiring requirements. This refinement is necessary because SAM abstracts both primitive behavior and stream connectivity, while the hardware requires concrete mappings from abstract operators to physical modules and fully elaborated multi-wire signals. We provide the primitive neighborhood constraints, their description, and their concrete signal elaboration for the Onyx-aware dataflow IR level buffer primitive in Table 5.1.

Our compiler lowering, therefore, resolves three things. First, SAM primitives are remapped to Onyx's hardware primitive set presented in Section 5.3, where one SAM operator may expand into several hardware nodes or collapse into a single one. Because Onyx was co-designed with the SAM abstraction, the only rewrite that requires primitive expansion/collapsing is memory primitive expansion: SAM models level scanners and level writers independently, but we implement each sparse memory tile in Onyx as a coordinated level scanner–buffer–writer triplet. This expansion is required in real hardware because every memory interface must sequence both reads from and writes to that memory. The lowerer therefore rewrites each SAM memory primitive into this triplet, as illustrated for the level scanner in Figure 5.7a. The compiler further elaborates all abstract streams into concrete wire bundles during lowering and constraint checking. For Onyx's ready–valid signaling, each SAM edge becomes a set of ready, valid, data, and optionally address wires with fixed bit widths. Figure 5.7b shows how an abstract `values` stream of an abstract primitive becomes the full ready–valid–data interface required by the CGRA interconnect and Table 5.1 provides exact source port and sink port signal expansions. Different primitives will have varying constraints types and exact signal elaborations depending on the pair of source primitive type and sink primitive type.

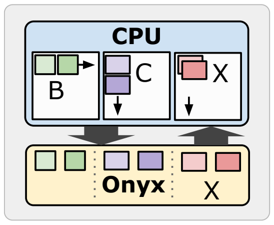


Figure 5.9: Sparse tiling.

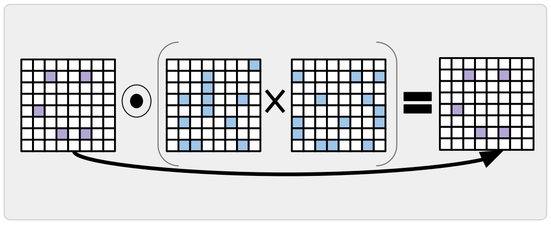


Figure 5.10: Expression fusion.

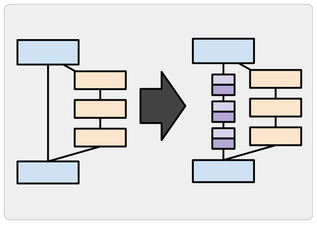


Figure 5.11: Pipeline balancing.

5.4.2 Sparse Data Tiling

Similar to other push-memory accelerators, programs mapped to Onyx must operate within the capacity of its global buffer and per-tile memory. Our sparse application compiler automates this process by generating tiled input data for the global buffer and further partitions that data into sub-tiles for each sparse memory tile (Figure 5.9). For every sparse tensor algebra expression compiled through Custard, our backend compiler also emits CPU-side code that constructs these tiles and sub-tiles for each dataset. The compiler also generates the SoC routines for moving tiled data from the ARM host to the CGRA fabric. This tiling flow supports both standard and unrolled executions.

To improve performance, the compiler omits empty tiles and does not transmit them to the accelerator. The generated code forms $N \times N$ tiles and $M \times M$ coordinate-space sub-tiles [181] and only enumerates the sub-tile pairs that share work for the given expression. Because shared coordinates vary across both expressions and datasets, the specialized sub-tile matching logic needs to be generated by our compiler. Consider our matrix multiplication example in Section 4.5.5 which pairs sub-tiles of **B** and **C** along the k dimension, whereas element-wise multiplication would pair the same sub-tiles along both the i and j dimensions. During evaluation, we sweep the tile sizes N and M to ensure all tiles fit within the global and per-tile memory limits. The compiler also produces reference result sub-tiles, calculated using PyTorch [160] and NumPy, as part of the collateral we use to validate Onyx's outputs.

5.4.3 Fusion

Onyx supports higher-order tensors, multiple sparse inputs, and, critically, expression fusion. Fusion allows the accelerator to avoid materializing large intermediate tensors and eliminates ineffectual work that would otherwise propagate across inputs. To illustrate this capability, we extend the running matrix multiplication example from Section 4.5.5 to a sampled SpM*SpM,

$$X_{ij} = \sum_k B_{ij} C_{ik} D_{kj}.$$

Many existing sparse accelerators—both fixed-function and reconfigurable—only admit unfused two-input kernels [246, 157, 252, 128, 197]. These systems must decompose the sampled matrix multiplication into

$$T_{ij} = \sum_k C_{ik} D_{kj}, \qquad X_{ij} = B_{ij} T_{ij},$$

which requires fully materializing the temporary tensor T_{ij}. This unfused schedule may also generate nonzeros in T that never contribute to the final output, as illustrated in Figure 5.11.

In contrast, our sparse application compiler composes Onyx's memory and PE tiles to evaluate the full three-input expression in a single invocation of the CGRA. This fused execution eliminates the temporary tensor and skips work on noncontributing coordinates, allowing the computation to take advantage of the true sparsity structure of the expression and avoid unnecessary memory traffic. These fusion capabilities arise directly from the SAM abstraction, whose dataflow formulation naturally exposes multi-input fusion opportunities that the backend can map onto the hardware. When requested by the user, our compiler and Onyx can still execute the unfused schedules, ensuring generality across schedules which is important for sparse tensor algebra due to its data-dependent nature (see Chapters 2, 4 and 7).

5.4.4 Pipelining

Our CGRA supports registering data at every hop of the interconnect, which shortens critical paths and enables high operating frequencies. We perform exhaustive pipelining along all routed connections using the SplitFIFOs introduced in Section 5.3 to achieve the maximum clock rate for each application. Exhaustive pipelining alone, however, does not address imbalances in the SAM graph. Uneven path lengths that merge can introduce backpressure, manifesting as bubbles during application execution.

To address this inefficiency, the compiler runs a path-balancing pipelining pass before exhaustive pipelining and tile placement. The algorithm identifies reconvergent paths in the dataflow graph, which typically occur at joiner nodes, and inserts the appropriate number of SplitFIFOs along the shorter branch. This ensures that all inputs to a joiner arrive at a uniform pace. Figure 5.10 demonstrates the effects of pipeline balancing where joiners are shown in blue, SplitFIFOs are shown in purple, and all other primitives are yellow. After balancing, the compiler maps the updated dataflow graph to the accelerator and applies additional exhaustive pipelining to all interconnect routes. The remaining backend compiler steps after this point leverage prior work [118, 141]: the compiler first maps the lowered dataflow graph onto an abstract model of CGRA resources, then places and routes onto the physical array, and finally generates the configuration bitstream that programs Onyx.

We provide an example mapping of our running matrix multiplication example from Section 4.5.5 onto a simplified Onyx CGRA array, as produced by our full backend compiler flow, in Appendix C.

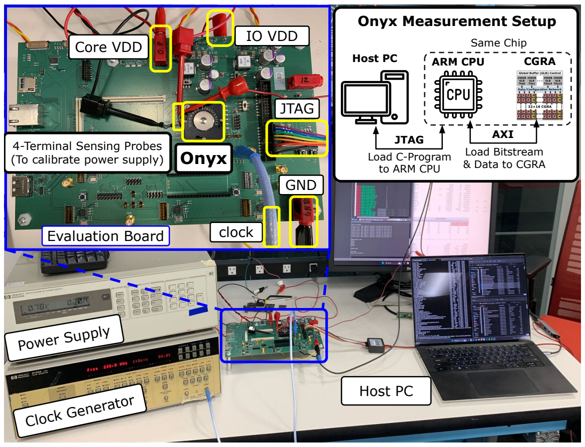


Figure 5.12: Experimental setup for testing the fabricated Onyx chip.

Table 5.2: Onyx specifications.

Technology	GlobalFoundries 12nm FinFet
Project Area	23 mm^2
Processor	ARM-M3 CPU
Voltage	0.78 V
Frequency (Processor)	500 MHz
Frequency (CGRA)	970 MHz
Peak Performance	571 INT16 GOPS
Peak Energy Efficiency	756 GOPS/W

Table 5.3: Onyx layout area.

Description	**Area (mm^2)**
Onyx	23.00 mm^2
Global Buffer	4.45 mm^2
GLB Tile (each)	0.249 mm^2
CGRA	8.28 mm^2
PE Tile (each)	0.011 mm^2
MEM Tile (each)	0.023 mm^2

5.5 Accelerator Evaluation and Results

Onyx was fabricated in GlobalFoundries' 12 nm process. Figure 5.1 provides an annotated die photograph, and Figure 5.12 shows our evaluation setup along with a close-up of the measurement board. The host machine generates the processor code, application inputs, and bitstreams, and transfers these artifacts to Onyx. Once received, the on-chip ARM CPU configures both the global buffer and the CGRA fabric to run the accelerated application. We drive the board using an external power supply and use an external clock generator to sweep application frequencies. Unless otherwise specified, Onyx operates at an I/O voltage of 1.8 V and a core voltage of 0.78 V, as summarized in Table 5.2. Onyx reaches a maximum frequency of 500 MHz for the processor subsystem and 970 MHz for the CGRA fabric. The chip delivers a peak performance of 571 INT16 GOPS and a peak energy efficiency of 756 GOPS/W. We compare all results in this section using reported numbers from prior work without normalizing for process technology.

5.5.1 Area Breakdown

We show the layout area breakdown of Onyx in Table 5.3. To understand how these areas compare with prior work, Figure 5.13 further illustrates the changes in Onyx cell areas of the PE and memory tiles when compared to the Amber CGRA[59]. We report all area numbers from unflattened synthesis results. Adding sparse primitives to the memory tile increases the memory tile size by 27%. For the

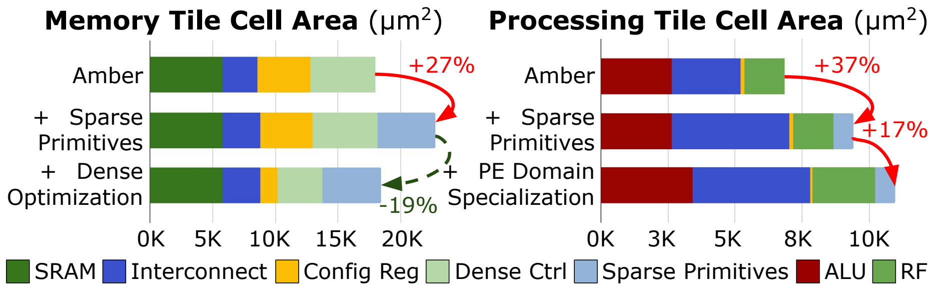


Figure 5.13: Memory and PE tile synthesis areas showing the changes after optimizations and added features.

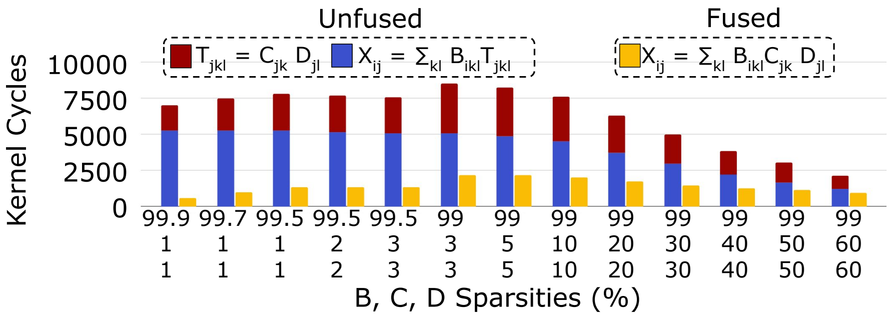


Figure 5.14: MTTKRP fusion runtime results for different input sparsities.

PE tile, adding sparse primitives increases the area by 37% with most of the area increase coming from the interconnect. Onyx's other memory tile and PE specialization optimizations decrease the memory area by 19% and increase the PE tile area by 17%, respectively. These optimizations are not presented in this dissertation, as they are unrelated to the sparse features in the architecture; further details can be found in the original Onyx journal publication by Koul et al. [117].

5.5.2 Sparse Application Results

This section quantifies the performance contributions of the compiler optimizations described in Section 5.4 and then presents end-to-end sparse application results across a range of kernels. For 2D expressions, we use matrices drawn from the SuiteSparse collection [49] (sparsities between 99.0–99.95% and dimensions ranging from 1813×1813 to 2021×2021) as well as uniformly generated random sparse matrices. For expressions involving 3D tensors, we generate uniform random tensors with 67% sparsity and dimensions spanning $8 \times 37 \times 10$ to $28 \times 35 \times 54$, unless otherwise specified. When tensor operands exceed the capacity of the global buffer, we stream tiles onto the chip as described in Section 5.4. All intermediate data needed to compute a tiled result is held on-chip.

Compiler Optimizations

First, we start with expression fusion. As described in Section 5.4.3, we utilize fusion to eliminate temporary tensors and ineffectual compute across multi-input expressions. Figure 5.14 shows the runtime of unfused graphs versus a fused graph for matricized-tensor times Khatri-Rao product

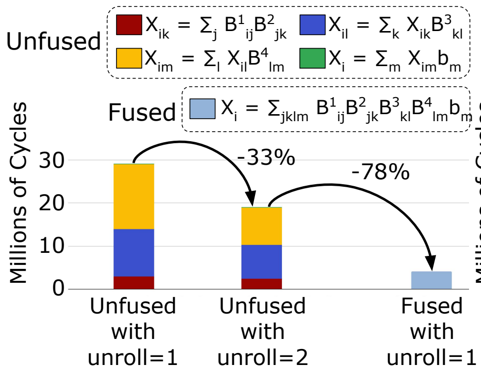


Figure 5.15: Iterative matrix-matrix/-vector multiplication fusion results on SuiteSparse matrices (*qiulp, west2021, watt_2, tols2000, tols2000-vectorized*) truncated to a shared dimension of 2100.

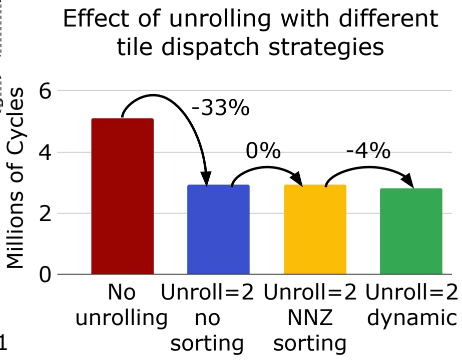


Figure 5.16: SpM*SpM runtime with three unrolling strategies: no unrolling, NNZ sorting, and dynamic dispatching.

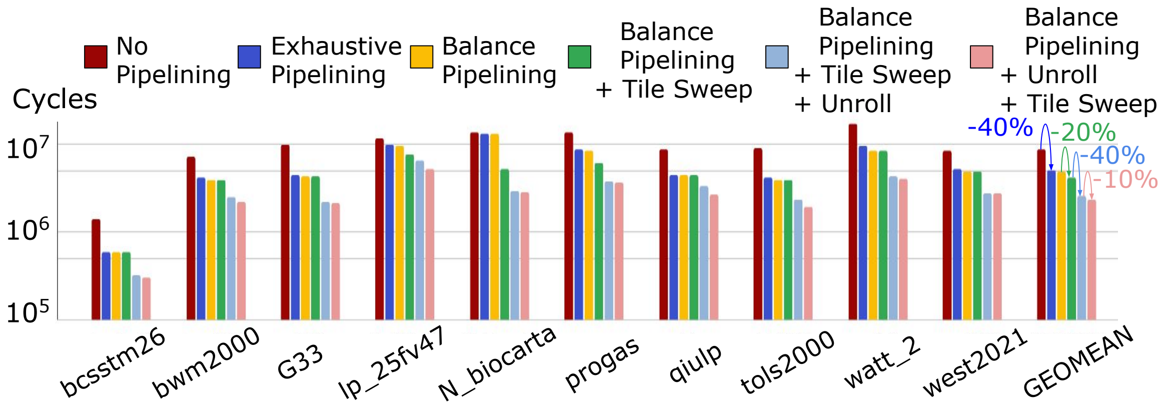


Figure 5.17: SpM*SpM performance improvement results after applying exhaustive pipelining, balance pipelining, a tile sweep, and unrolling.

(MTTKRP) with different input sparsities (on 10×10×10 and 10×10 tensors). As B gets sparser and C and D get denser, the speedup increases up to 11.8×. Figure 5.15 shows the runtime for a four-iteration iterative sparse matrix-vector multiplication. Since unfused kernels have fewer inputs, we can benefit from unrolling. Even with this optimization, the fused kernel performs 4.6× faster than the unfused kernels.

For the pipelining ablation study below, the Onyx compiler performs sparse application pipelining as described in Section 5.4.4. Figure 5.17 shows matrix-multiplication runtime results with no pipelining, when only exhaustive pipelining is applied, and when balancing is applied before exhaustive pipelining. Balancing ensures that there are no imbalanced paths in the mapped application graph and exhaustive pipelining ensures high application frequency.

Since our CGRA is homogeneous, we can *unroll* or parallelize a bitstream over the array to improve performance. Unfortunately, feeding tiles into two parallel matrix multiplications in order can lead to load imbalance and result in suboptimal performance. In Figure 5.16, we compare the

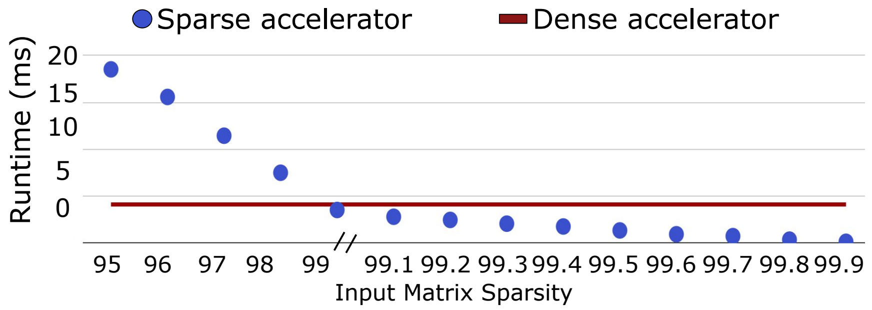


Figure 5.18: Onyx's sparse accelerator fabric versus dense accelerator fabric runtime results on a 512×512 matrix multiplication of varying sparsity.

results for three tile dispatch strategies. The first approach is in order and is the baseline. The second strategy sorts the tile pairs based on the sum of the number of non-zeros in the two inputs. Finally, the third approach dynamically dispatches data whenever a region of the accelerator is finished. This requires duplicating input data in our global buffer, but achieves the highest performance, 1.76× faster than no unrolling. Figure 5.17 shows a 1.6× improvement in performance for sparse matrix multiplication with unrolling.

As described in Section 5.4.2, tiling can have a significant effect on performance. Smaller tiles perform less computation on the array but are faster and reduce the ratio of kernel acceleration to the processor overhead of reconfiguring the global buffer. Figure 5.17 shows our most performant results when we run an exhaustive tiling sweep before and after unrolling the application. Performing the tiling sweep after unrolling is slightly better since the ratio of kernel time to processor time changes with unrolling.

Sparse Versus Dense Matrix Multiplication

For a 512×512 matrix multiplication with varying input sparsity, we compare our dense accelerator configuration versus its sparse counterpart in Figure 5.18. After 99% matrix sparsity, we should configure our accelerator as a sparse accelerator. At 99.9% input sparsity, we perform 30× better than the dense accelerator configuration.

Given the above data in Figure 5.18, future work includes improving primitive performance, supporting other dataflows (for example, Gustavson's algorithm [75]), and adding new primitives to increase accelerator utilization. With these improvements, the sparse accelerator will perform better on less sparse matrices.

Final Sparse Results

We evaluate our sparse tensor algebra kernels using energy-delay product (EDP) versus a baseline CPU using the Intel Math Kernel Library (MKL) [95] and code generated by TACO [108]. All CPU results are run on a 12-core Intel Xeon CPU. We used the Intel oneAPI Math Kernel Library

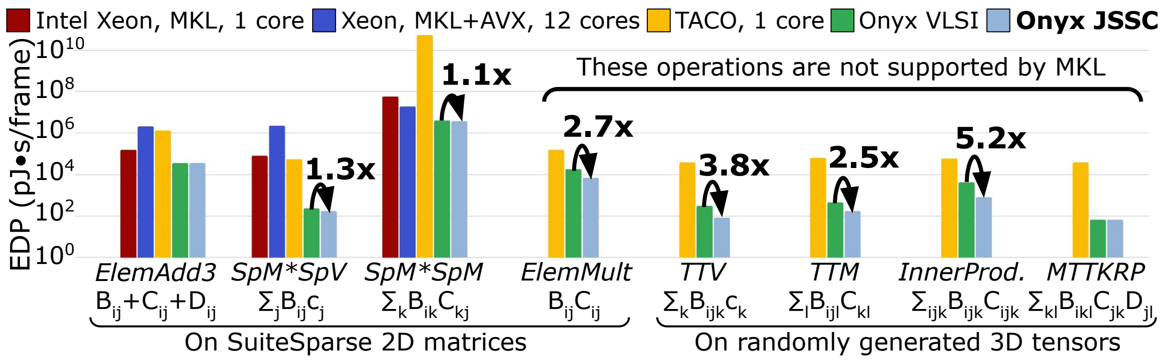


Figure 5.19: Energy-delay-product (EDP) compared with CPUs with sparse acceleration libraries for a variety of sparse 2D and 3D kernels.

version 2023.2.0 2023 1.0. For single-core MKL baselines, we ran the library with no optimizations enabled. For 12-core MKL+AVX, we configured MKL with AVX512 and OMP enabled. For the TACO baseline, we used the GitHub master branch compiled with GCC 8.5.0 and default schedules for each expression. In all baseline configurations, we take the median runtime across 10 iterations.

Utilizing all scheduling optimizations described above, we perform up to 5.2× better in EDP on sparse tensor algebra expressions versus our initial VLSI publication [119] and up to 4.4 to 565× better in EDP versus the CPU baselines as shown in Figure 5.19. Onyx excels in expressions with higher-order tensors and multiple inputs, where we can leverage fusion to outperform the baselines.

5.6 Onyx in the Landscape of Fabricated Accelerators

Typically, fixed-function accelerators are used to accelerate sparse workloads. For example, Huang et al. [92] target augmented reality with dedicated sparse-CNN and sparse-GCN engines but only support accelerating these two operations. Similarly, Song et al. [198] design several fixed-function accelerators for 3D navigation, including a sparse matrix multiplication engine, but do not accelerate full application domains or exploit sparsity in other kernels.

Several other works focus on accelerating sparse kernels. For example, [240, 45, 255, 134] optimize matrix-multiplications over different levels of input sparsity, while [81, 202] support operations on high-order tensors such as MTTKRP. Additionally, [176] supports tensor addition, but does not take full advantage of sparse iteration for intersects/unions. Symphony [161] is a sparse architecture that supports all of sparse tensor algebra but has no hardware implementation. As opposed to Onyx, none of these works have silicon results.

On the other hand, there are programmable accelerators that support application domains, such as Feng et al. [59] and Zhang et al. [247] (shown in Table 5.4). However, [59] and [247] are limited to dense applications only and achieve lower peak performance. There are several other works that explore CGRA architectures [165, 212], [37], [70], [217], but they focus solely on dense applications and only have simulation or FPGA-based implementations.

Table 5.4: Fabricated hardware comparison table.

	This Work	Amber JSSC 22 [59]	Zhang VLSI 22 [247]	Huang VLSI 22 [92]
Architecture	SoC w/CGRA	SoC w/CGRA	CNN/Image Processing PE Array	Sparsity-Aware CNN-GCN
Programmability	✓	✓	✓	×
Sparse/Dense	Sparse/Dense	Dense Only	Dense Only	Sparse/Dense
Technology/Area	12 nm/23 mm2	16 nm/20.1 mm2	22 nm/8.8 mm2	28 nm/8.3 mm2
Datatypes	BFloat16, INT16	BFloat16, INT16	INT16	INT8
# of Cores	384 PEs, 1 M3	384 PEs, 1 M3	576 PEs, RISC Core, Custom M33	1024 8-bit MACs
Total SRAM	4.5 MB	4.5 MB	1428 KB, 2MB MRAM	292 KB
Voltage & Frequency	0.78 V @970 MHz	0.84 V @580 MHz 1.29 V @955 MHz	0.5 - 1.0 V 56 KHz - 190 MHz	10 - 200 MHz
Peak-Performance[a]	571 INT16 GOPS @(0.78 V, 850 MHz)	367 INT16 GOPS @(1.29 V, 955 MHz)	414 INT16 GOPS @(1.0 V, 180 MHz)	Dense: 410 GOPS Sparse: 3.3 TOPS
Peak Energy-Efficiency[a]	756 INT16 GOPS/W @(0.66 V, 500 MHz)	538 INT16 GOPS/W @(0.84 V, 580 MHz)	7.0 INT16 TOPS/W @(0.5 V, 16 MHz)	Dense: 3.1 TOPS/W Sparse: 25.1 TOPS/W

[a] Normalized to MAC=2OPs for all.

Chapter 6

An Alternative Approach to Targeting Sparse Dataflow Hardware

“Simplicity does not precede complexity, but follows it”

Alan J. Perlis

Up to this point, this dissertation develops fundamental ideas that transform a loop-based execution model to a dataflow execution model that is closer to accelerator VLSI implementations. However, an alternative approach would be to solely leverage imperative loops as the abstraction within these programming systems for sparse accelerators. In this chapter, we explore this contrasting approach and use it to develop another end-to-end compiler, Stardust, from the sparse DSLs presented in this thesis to another sparse reconfigurable dataflow accelerator. Instead of targeting a dataflow execution model, Stardust transforms sparse iteration loop nests to imperative parallel-patterns (like map and reduce) augmented with accelerator memories and sparse computation patterns and then leverages prior compilers [251, 111, 176] to target one prior sparse accelerator [176]. It is important to understand how the system and contributions in this chapter fit into that of the previous chapters. Therefore, Figure 6.1 shows a diagram of the two contrasting programming systems for sparse accelerators and how they relate.

In order to understand the full picture, you should understand the historical context for these contrasting compiler techniques. Only a few prior reconfigurable dataflow accelerator architectures (RDAs) for sparse applications existed Section 2.3, one of them being the Capstan RDA. The goal to build programming systems for sparse accelerators from high-level DSLs started with building that system for Capstan. And Capstan already had a programming model from a hardware DSL, a sparse iteration extension of Spatial [111], and multiple years of compiler research to target Capstan [164, 111, 251]. Spatial is a hardware DSL that is based on parallel patterns [164], which are functional map-reduce paradigms that are still fundamentally based on imperative loops. Then, the output of the Spatial compiler is fed to SARA, a compiler from imperative loops to dataflow graphs on logical and physical dataflow hardware. Therefore, we decided to leverage the existing software infrastructure of Capstan when creating our programming system.

Although targeting a loop-based representation (i.e. Spatial code) simplified parts of our compiler system, it also complicated a lot. To compensate for dataflow accelerators, the Spatial programming model had other incompatibilities with the sparse DSLs described in Section 3.1. Most of the incompatibilities arise from the memory hierarchy and design of dataflow accelerators. For efficiency and performance reasons, most accelerators strip away caching mechanisms and use explicitly managed accelerator memories [162, 176, 161, 133, 165]. These memories are also more similar to their physical hardware implementations. Therefore, in explicitly managed memories, programs must emit explicit reads and writes of data decoupled from the computation of that data, manage movement between multiple levels of memory, and contend with memory addresses and limited sizes. Unlike SAM, which assumes infinite resources and ideal memories, the work in this chapter makes strides towards generating sparse code with realistic and explicit memories (which include types, hierarchies, and resource constraints).

Stardust, thus, became the first end-to-end system built from sparse DSLs to real sparse accelerator hardware. The ideas in Stardust are valuable for targeting explicit memories from high-level representations and for transforming loop-based iteration into hardware-like bitvector iteration.[1] And the ideas can generalize to other systems and domains of computation. Ultimately, the Stardust system could only target one specific accelerator implementation. Motivated by the goal of portable systems and code across multiple accelerators, the limitations of Stardust was a key catalyst in the inception of SAM (Chapter 4), and later Onyx (Chapter 5), and Mosaic (Chapter 7).

In the next chapter (Chapter 7), we will also see how the kernel libraries generated by Stardust allow users to run sparse tensor programs across multiple hardware backends, including RDA and GPU accelerator architectures.

6.1 Background on Capstan Hardware

Our work in this chapter builds on two lines of prior work: sparse tensor algebra compilation techniques for CPUs [108, 181, 41] and the extended Spatial DSL [111] that targets the Capstan RDA [176]. Since we provide detailed sparse tensor algebra compilation background in Sections 2.2.2 and 3.1, we provide more details beyond Section 2.3 on the Capstan RDA and its Spatial programming model in this section.

[1]Bitvector iteration is also a promising avenue for generating acceleration data structures for sparse operations akin to mipmaps and spatial query trees in graphics that allow for significant pruning or down-sampling.

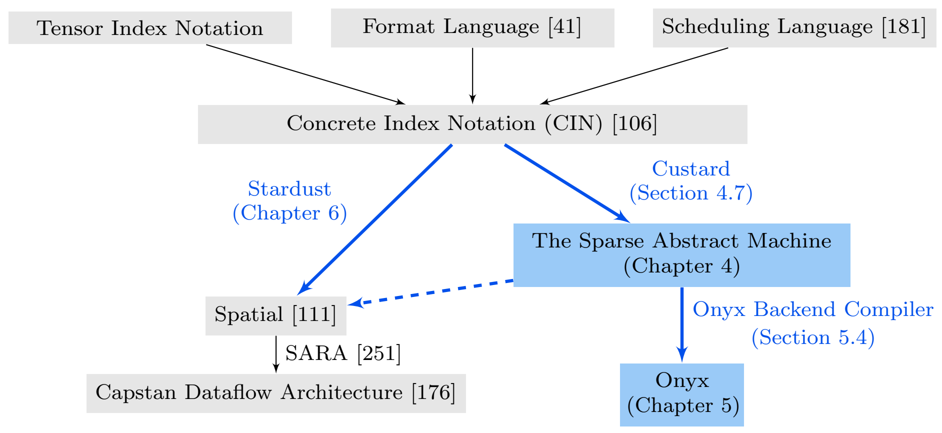


Figure 6.1: How two compiler paths that achieve the same end-to-end programming system, SAM and Stardust, compare and fit together. Arrows denote compiler paths, and blue denotes new contributions of this work. Dashed arrows are paths that we demonstrated conceptually without a full compiler implementation (where we argued for the feasibility and lowered to the target execution model).

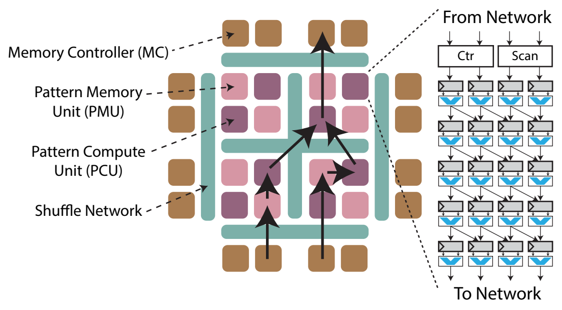


Figure 6.2: A high-level overview of the Capstan architecture, showing the opportunities for high-level parallelism across PCUs and vectorized parallelism within a PCU.

Reconfigurable dataflow architectures (RDAs) improve performance and efficiency by removing overhead found in CPUs and GPUs. RDAs map programs in space, meaning multiple data elements are processed in the same clock cycle by pipelined and parallel compute units. Capstan [176], shown in Figure 6.2, derives from Plasticine [165] with support for sparse operations. A notable Capstan contribution is its ability to iterate over sparse tensors using scanners and bitvectors, which is enabled by its microarchitecture and apparent in its programming model. In order to program Capstan, sparse iterations must be split into pattern headers and pattern bodies, where headers determine which (un)compressed iterations to run, and bodies use header iteration information to load, compute, and store data.

Users program Capstan with Spatial [111].[2] Compilers that handle low-level optimizations and insert memory-consistency logic [111, 251] automatically lower Spatial to a streaming on-chip dataflow graph and a cycle-accurate simulator.

Spatial uses a map-reduce abstraction. Each `Foreach` or `Reduce` pattern is counter-indexed with

[2]A full description of Spatial can be found at `spatial-lang.org`.

an explicit parallelization factor; multiple levels of nested loops can be independently parallelized to exploit different program-level parallelism opportunities. Typically, the innermost loop is vectorized, and the outermost loop is replicated across pattern compute units (PCUs). Capstan provides sparse iterator patterns—including union and intersection combinations—in addition to dense ones. Sparse patterns iterate by running on non-zero bit-vector elements using the index of the non-zero element instead of a counter. These sparse patterns are shown in Figure 6.9 and described later in Section 6.6.

Spatial has an explicit, decoupled, programmer-managed memory hierarchy. In a CPU, memory is managed using caches and demand misses; however, Spatial requires manually partitioning data into chunks that fit on-chip and controlling the corresponding data movement. Specifically, there are four programmer-controlled memory types, ranging from far to near: DRAM, SRAM, FIFOs, and registers, with the middle two mapping to Capstan's pattern memory units (PMUs).

6.2 Motivation for Managing Explicit Memories

There are several challenges when compiling to RDAs like Capstan: managing different types of RDA memories, mapping computation to different accelerator units (which are parallel patterns in the case of Spatial), and controlling combinations of sparse coordinate–value streams between those units. Imperative languages like C present the programmer with a convenient *pull* memory model—when you need data, you ask for it—as CPUs and GPUs separate control logic from memories. In RDAs, however, programmers must explicitly manage data movement through the memory hierarchy, as the control logic is attached to memories in a *push* memory model [162, 133, 165, 176, 28, 59]. These challenges with RDAs are inevitable and arise as complexity in the Spatial programming model. Spatial, specifically, has parallel patterns which may look like imperative loops, but their programming abstraction is different. The patterns represent scanners, producing variables in a fixed manner over time, rather than temporally modifying variables in place as in imperative code. This way of representing scanners (and their parallelism in space) severely limits what Spatial code is valid, and these programming and compilation challenges are further exacerbated by the inherent complexity in sparse kernels.

To illustrate the fundamental difference between compiling sparse expressions to imperative C-like code versus a parallel-patterns programming model, we introduce a common sparse linear algebra kernel in machine learning [61], sampled dense-dense matrix multiplication (SDDMM), as a running example. SDDMM produces a result by performing a dense matrix multiplication sampled by a sparse mask. The tensor index notation for SDDMM is $A_{ij} = \sum_k B_{ij} C_{ik} D_{kj}$ where A and B are compressed sparse row (CSR) matrices. However, index notation is declarative and does not specify any low-level control flow. We can expand the index notation expression with three loops to describe control flow (over a scalar expression): $\forall_i \forall_j \forall_k \left(A_{ij} \mathrel{+}= B_{ij} C_{ik} D_{kj}\right)$.

Again, this notation is called concrete index notation (CIN) [106], and we provide its syntax in Figure 3.3. Many compiler decisions in prior work presuppose an imperative target language. Figure 6.3 shows the C-like code generated from the CIN statement by one such compiler [110]. The following code locations in Figure 6.3 describe how prior work compiles this example to imperative code and shows why it is more straightforward than compiling to parallel patterns: ❶ the ∀ nodes are exactly converted into for-loops (highlighted in red), ❷ tensor elements are loaded/stored one element at a time through indirect accesses that syntactically match the index expression, ❸ tensor computation occurs only in the innermost loop, and ❹ tensor accumulations may be implemented as temporally-repeated variable modifications.

To target the parallel-patterns programming model of Spatial, on the other hand, a compiler cannot depend on the assumptions of an imperative programming model. For example, the compiler to imperative code can load elements where tensor accesses syntactically appear in the index expression, whereas most of the generated Spatial code in Figure 6.4 manages data movement. Specifically, Stardust must address the following issues when compiling to Spatial code as shown in Figure 6.4: ❺ the ∀ nodes are converted to *different* parallel patterns, which may include sparse patterns that scan through data without temporal counters, ❻ tensor elements are transferred in chunks parallelized across pipelines, ❼ tensor data must be retrieved whenever the data arrives not just in the innermost loop (at line 32), and ❽ tensor computation (like accumulations) cannot temporally modify variables so they are mapped to patterns (in this case the Reduce pattern) that represent computation in space.

The lines highlighted in blue in Figure 6.4 show the code complexity required to manage memories and data movement in the Spatial programming model. The complexity stems from the explicit, decoupled push data movement of RDA accelerators. This memory management has two parts:

1. Explicit mapping of tensor arrays to different memory types, such as `FIFO`, appearing on the right-hand-side of the immutable variable `val` declarations.

2. Bulk data transfers between these memory types, demonstrated by the many `load` and `store` keywords.

The Spatial programming model represents an accelerator memory hierarchy, where the different memory types have different capacities, locality, access constraints, and properties. We do not expect a performance engineer who is familiar with CPU code to write such memory management code for three reasons: it is abstracted away on CPUs, it requires intimate knowledge of the accelerator memory hierarchy design and memory types, and it is tedious since the memory management takes up a majority of the Spatial program. Therefore, Stardust automatically generates this memory management code for usability and productivity, raising the programming abstraction of RDAs.

```
❶for (i = 0; i < C1_dim; i++)
  for (jB = B2_pos[i]; jB < B2_pos[i+1]; jB++) {
    j = B2_crd[jB];❷
    for (k = 0; k < D1_dim; k++)
      ❸A[jB] ❹+= B[jB]*C[i,k]*D[k,j]; }
```

Figure 6.3: C implementation of SDDMM with CSR matrices, generated by the TACO compiler.

```
// Spatial header code ...
// Initialize all DRAM arrays as <name>_d
val A2_pos_d = DRAM[T](nnz_max)
...
Accel {
  val B2_pos = SRAM[T](nnz_accel_max)
  B2_pos load B2_pos_d(0::(B1_dim + 1) par ip)
❺Foreach (C1_dim by 1 par bp) { i =>
    val A_vals = FIFO[T](16)
    val A2_crd = FIFO[T](16)
    val A2_pos = SRAM[T](nnz_accel_max)
    val jB_start = B2_pos(i)
    val jB_end = B2_pos((i + 1))
    val jB_len = jB_end - jB_start
    val B2_crd = FIFO[T](16)
    B2_crd load B2_crd_d(jB_start::jB_end par 1)❻
    val B_vals = FIFO[T](16)
    B_vals load B_vals_d(jB_start::jB_end par 1)
    Foreach (jB_len by 1 par 1) { jB =>
      val j = B2_crd.deq
      val B_hoisted = B_vals.deq❼

      val D_vals = SRAM[T]((nnz_accel_max / 4))
      D_vals load D_vals_d(j*D1_dim:: \
                      (j+1)*D1_dim par ip)
      val C_vals = SRAM[T]((nnz_accel_max / 4))
      C_vals load C_vals_d(i*C2_dim:: \
                      (i+1)*C2_dim par ip)

      val tjA_vals = Reg[T](0.to[T])
    ❽Reduce(tjA_vals)(D1_dim by 1 par ip) { k =>
        ((B_hoisted * C_vals(k)) * D_vals(k))
      } { _ + _ }
      A_vals.enq(tjA_vals)
      A2_crd.enq(j)
    }
    A2_pos(i + 1) = jB_end
    A_vals_d stream_store_vec(jB_start, \
      A_vals, jB_len)
    }}
```

Figure 6.4: Spatial implementation of SDDMM with CSR matrices. Lines highlighted in blue are memory management.

6.3 Stardust Compiler Contributions and Overview

We introduce Stardust, a compiler from tensor index notation to an RDA (Capstan) through Spatial language. Stardust users first control and schedule the data and computation placement on a high-level abstract RDA, allowing the compiler to infer lower-level architecture-specific details. The compiler automatically handles fine-grained data structure binding to different types of memories along with explicit decoupled memory movement between those memories. The compiler also manages transformations from abstract loops to scanner functions in the parallel-pattern output language. Our contributions are:

- a data representation language that can express accelerator tensor placement abstractly.
- an algorithm that binds data structures in abstract memory to different physical memories on

the RDA.

- a scheduling language that can express how portions of a (potentially transformed) sparse tensor algebra expression should be mapped to a sparse accelerator.
- a lowering rewrite system that maps sparse tensor algebra expressions to a parallel-pattern language.

We use Stardust to compile a previously used benchmark set [108] to the Capstan RDA [176]. Stardust produces code that performs on average 0.65× that of the only hand-optimized kernel from the benchmark (SpMV) written for Capstan by its authors [176]. We demonstrate the generality of Stardust by generating nine new sparse algorithms in addition to SpMV. These ten Capstan algorithms outperform CPUs by 138× on average (geo-mean) and GPUs by 41× on average. The speedups are of the same order of magnitude as in the original Capstan work, which stem from its massively parallel and pipelined design. These experiments show that Stardust makes it feasible to rapidly develop sparse RDA kernels.

We implement Stardust as a new compilation path inside the open-source TACO system [108] as shown in Figure 6.5, where blue indicates our contributions. Like TACO, Stardust takes as input tensor index notation, a format language [41], and a scheduling language [181]. Stardust extends the format language to describe whether tensor data is placed on the accelerator and the scheduling language to describe how (sub-)computation maps to compute units on the accelerator. Stardust generates Spatial code [111], which is then compiled using prior work [111, 251] to cycle-accurate simulations of the Capstan sparse RDA [176].

Tensor index notation lowers to *concrete index notation* (CIN) [106], a loop-based IR where compressed tensor data structures are abstracted away (shown in Figure 3.3). Scheduling language commands are applied as rewrites on CIN. We extend the original scheduling commands of TACO to target parallel patterns in Section 6.6. A scheduling command may additionally add metadata that is used during CIN lowering. Relation nodes store this metadata by tracking the relationships between CIN nodes, which are used to insert remapping code.

Stardust solves two key problems in compiling to sparse accelerators: mapping tensors to memories and mapping computation to the accelerator. Once mappings are decided, by the user or compiler, Stardust generates Spatial code.

Users only need to provide coarse-grained tensor placement information; Stardust automatically synthesizes the rest of the data placement during code generation. Users decide whether a tensor lives on or off the accelerator, with a new memory location construct in the format language (Section 6.4). During code generation, Stardust completes fine-grained data placement via a memory analysis algorithm. The memory analysis algorithm first determines the exact placement of tensor data for every level of the memory during compilation (Section 6.5). Stardust then generates the required data transfer code between memory types.

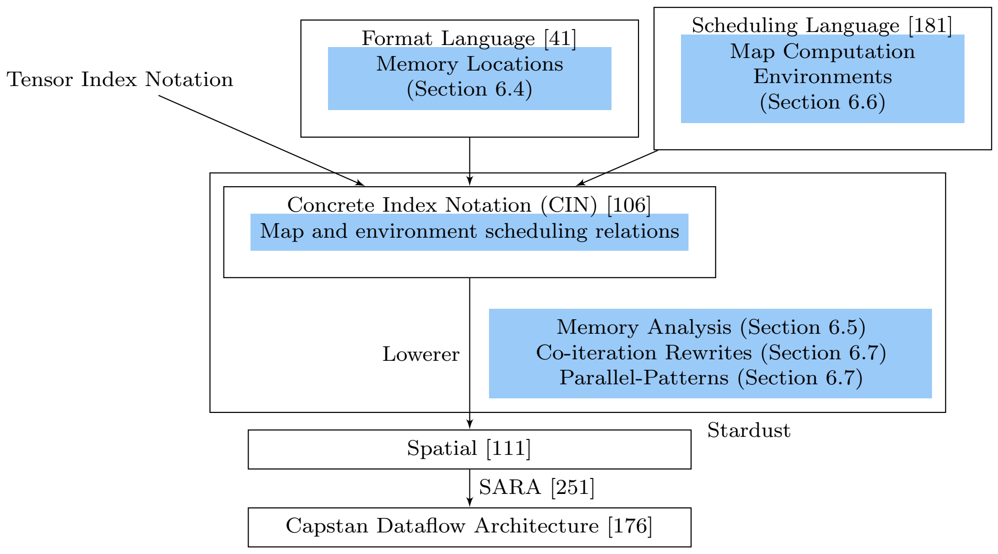

Tensor Index Notation
Format Language [41]
Memory Locations (Section 6.4)
Scheduling Language [181]
Map Computation Environments (Section 6.6)
Concrete Index Notation (CIN) [106]
Map and environment scheduling relations
Lowerer
Memory Analysis (Section 6.5)
Co-iteration Rewrites (Section 6.7)
Parallel-Patterns (Section 6.7)
Stardust
Spatial [111]
SARA [251]
Capstan Dataflow Architecture [176]

Figure 6.5: Stardust overview. Blue denotes new contributions.

Once tensors are placed on the accelerator using the format language, users must also map the computation that uses those tensors onto the accelerator (Section 6.6). To target specialized hardware, a user writes a schedule that reorganizes the computation until a sub-computation is exposed. The user then maps the exposed computation to a specialized hardware pattern. To simplify scheduling, Stardust also provides a single shorthand command that combines both the computation reorganization and mapping.

Finally, Stardust uses a term rewriting algorithm to recursively compile the CIN to parallel patterns. The algorithm recursively lowers to parallel patterns depending on the iteration properties of the tensors in the expression (Section 6.7).

6.4 Mapping Data to Memories

One key difficulty in generating Spatial code is determining how data should be stored in the sparse RDA. The abstract memory model in this work allows users to reason about RDA memory simply as a single level. Stardust raises the memory-model abstraction by providing a description of coarse-grained memory regions, which the user explicitly manages via the format language by denoting a tensor's memory scope as either *off-chip* or *on-chip*. Then, the compiler infers fine-grained memory details about the on-chip memories during compilation, as discussed in Section 6.5.

6.4.1 Abstract Memory Model in the Format Language

Stardust abstracts over multiple Spatial memories into two memory regions: either off (shared with the host) or on the accelerator. Users place tensors from an expression onto one of these memory regions. This memory model is essential because it affects how Stardust generates code and how

```
 1 // Format language: Define off-chip (global) tensor formats
 2 Format csr_off({dense, sparse.}, offChip);
 3 Format rm_off({dense, dense}, offChip);
 4 Format cm_off({dense, dense}, {1,0}, offChip);
 5
 6 // Declare input and output tensors
 7 Tensor<int> A({N,N}, csr_off);
 8 Tensor<int> B({N,N}, csr_off);
 9 Tensor<int> C({N}, rm_off);
10 Tensor<int> D({N}, cm_off);
11
12 // Define SDDMM computation (algorithm).
13 IndexVar i, j, k;
14 A(i, j) = B(i, j) * C(i, k) * D(k, j);
15
16 // Scheduling language: Define environment variables
17 IndexStmt stmt = A.getAssignment();
18 stmt = stmt.environment(innerPar, 16);
19 stmt = stmt.environment(outerPar, 2);
20
21 // Scheduling language: Precompute accumulation into a ws register,
22 // then accelerate using a Reduce pattern
23 Tensor<int> ws(onChip);
24 stmt = stmt.precompute(B(i,j)*C(i,k)*D(k,j),{},{}, ws);
25 stmt = stmt.accelerate(forall(k, ws+=B(i,j)*C(i,k)*D(k,j)), Spatial, Reduction, innerPar);
```

Figure 6.6: Stardust input (user) code for SDDMM.

users interact with that generated code. Therefore, the memory model of Stardust must not only differentiate between these two regions, but also give users explicit control over them.

The format language of Stardust lets a user explicitly place a tensor into a memory region of choice. The off-chip tensors are globally accessible to all backends involved in the computation (host and accelerators) whereas on-chip tensors are only locally accessible to one accelerator backend. An example of the format language for our SDDMM example is shown in Figure 6.6 lines 2–4. Lines 6–9 in Figure 6.6 then demonstrate how the format language is used to declare the input and output tensors of an index notation expression.

6.4.2 Representing Data Movement in CIN

We give users control of on- to off-chip transfers because an expression may have multiple transfer locations with different performance characteristics, as demonstrated by Section 4.8.4 and Chen et al. [34]. Since these decisions impact end performance, it is better to separate that concern from the Stardust compiler using schedules. Therefore, Stardust's new format language combines with the scheduling language such that users represent data movement in CIN. Stardust expresses transfers between the host and the accelerator within CIN as an assignment statement (A in Figure 3.3). An assignment between a tensor annotated with one memory region and another tensor in the other region necessitates a transfer of data between them. The assignment statement may have temporary tensors, which are tensor workspaces that store intermediate values.

A user inserts the memory-annotated temporary tensor into CIN via the `precompute` scheduling

$$\forall_i \forall_j (\forall_k (A_{ij} \mathrel{+}= B_{ij} C^{\text{on}}_k D^{\text{on}}_k) \ \textsf{where} \ \forall_k (C^{\text{on}}_k = C_{ik}) \ \textsf{where} \ \forall_k (D^{\text{on}}_k = D_{kj}))$$

```
stmt = stmt.precompute(C(i,k), {k}, {k}, C_on)
stmt = stmt.precompute(D(k,j), {k}, {k}, D_on)
```

(a) The rewritten CIN after partial on-chip loads of C_{rows} and D_{cols} in the j-loop body using two `precompute` commands.

$$\forall_i \forall_j \forall_k (A_{ij} \mathrel{+}= B_{ij} C^{\text{on}}_{ik} D^{\text{on}}_{kj}) \ \textsf{where} \ \forall_i \forall_k (C^{\text{on}}_{ik} = C_{ik}) \ \textsf{where} \ \forall_j \forall_k (D^{\text{on}}_{kj} = D_{kj})$$

```
stmt = stmt.precompute(C(i,k), {i,k}, {i,k}, C_on)
stmt = stmt.precompute(D(k,j), {k,j}, {k,j}, D_on)
```

(b) The rewritten CIN after initial load of C and D entirely before computation loops using two different `precompute` commands.

Figure 6.7: Two SDDMM CIN statements with corresponding schedules demonstrating distinct memory transfer patterns. Tensors $A, B, C^{\text{on}}, D^{\text{on}}$ live on-chip and C, D live off-chip.

command [106], whose C++ declaration is in Table 3.1. The `precompute` command transforms a CIN statement with a sub-expression e into a new CIN statement with a where sub-statement. A where is a producer-consumer statement whose producer and consumer sides both involve a temporary tensor $\mathcal{T}$. The producer side produces data from a sub-expression e and stores it into $\mathcal{T}$ via an assignment statement. The consumer side consumes data from $\mathcal{T}$ and uses that data to compute the result of the where statement. The transformed CIN, including its assignment statements embedded with memory movement information, is different depending on how the user applies the precompute schedules.

Consider the two SDDMM examples in Figure 6.7 with distinct `precompute` schedules. The examples demonstrate how modifications in the precompute command and tensor format results in different CIN statements. The two schedules differ in their on-chip temporary tensor memory sizes—Figure 6.7a uses two temporary vectors whereas Figure 6.7b uses two temporary matrices. The two schedules also differ in which indices load the tensor data—Figure 6.7a partially loads rows of C into C^{on}_k and columns of D into D^{on}_k at the j-loop body, whereas Figure 6.7b loads the entire C and D matrices into C^{on}_{ik} and D^{on}_{kj} respectively in the innermost loop. CIN embeds memory movement within its forall and access indices. Finally, our SDDMM example in Figure 6.6 has yet another schedule different from Figure 6.7, where off-chip data is loaded into an on-chip scalar temporary (line 21).

6.5 Physical Memory Mapping

As Stardust generates Spatial code, it transforms the abstract memory model into the physical memory model of Spatial (an abstraction that is closer to Capstan's physical memory design). The physical memory model is a finer-grained hierarchy representation, containing four memory types instead of two. As in a standard memory hierarchy, the memory types start with the largest capacity and farthest from the accelerator compute units and end with the smallest capacity closest to the accelerator compute units. The physical memory types are now fixed-length, which is not a requirement in the abstract memory model.

At this compilation step, sparse tensors are represented as compressed data structures made up of several arrays. These arrays are divided into chunks that are placed in different physical memories. The placement involves three decisions:

1. which memory type to place a chunk in (since different memory types have different capabilities),
2. where allocate the memory in the code, and
3. where to transfer data between chunks in the code.

Stardust solves the placement problem through a two-step memory analysis. The pass first performs a memory pinning analysis to decide which memory type to place each chunk in (Section 6.5.2), and then performs a memory lifetime analysis to generate allocation and transfer code (Section 6.5.3).

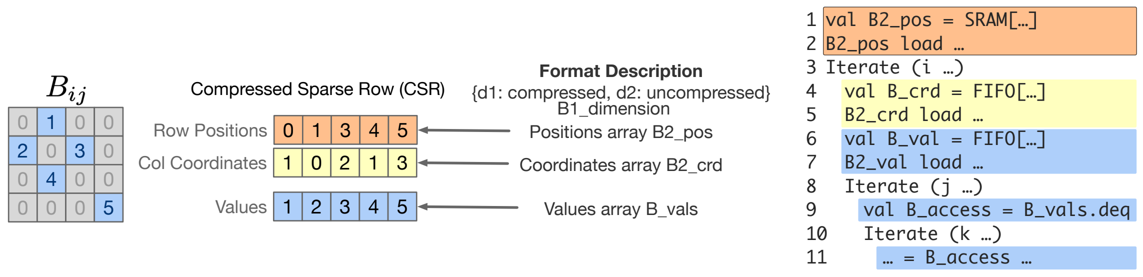
6.5.1 Tensor Data Structures

The compiler takes whole tensors in abstract memory and reasons about their constituent arrays. We refer to these as sub-arrays and they encode the physical data structure of a dense or compressed tensor. Stardust represents tensor data structures as per-level formats [41]. A tensor has multiple coordinate levels and a single value level [208, 108]. The value level always consists of a values array that stores actual tensor data. The specific coordinate level sub-arrays depend on the level format. If the format is a compressed (sparse) coordinate level, the level stores compressed coordinates in two arrays: positions and coordinates. Position arrays are addressed in an $addr, addr + 1$ fashion, while coordinate arrays are addressed indirectly based on the obtained positions. If the format is an uncompressed (dense) coordinate level, the level stores only its dense dimension as a scalar sub-array. A compressed tensor has one or more compressed levels.

The user provides format information to Stardust through the format language. Consider the tensor array representation of B in our running SDDMM example (illustrated in Figure 6.8). B's CSR data structure is shown in Figure 6.8b and format language description and sub-arrays are in Figure 6.8c.

6.5.2 Memory Pinning Analysis

Stardust maps tensor sub-arrays based on both sub-array and memory properties. To leverage locality, these memory types have different capacities, transfer speeds, access patterns, scopes, lifetimes, and programming constructs. In Spatial, the physical memory model is a memory hierarchy with (sparse/dense) DRAM $\rightarrow$ (sparse/dense) SRAM $\rightarrow$ FIFO $\rightarrow$ Register (from largest to smallest). The compiler binds sub-arrays to these memories and generates data transfers.



```
val B2_pos = SRAM[…]
B2_pos load …
Iterate (i …)
  val B_crd = FIFO[…]
  B2_crd load …
  val B_val = FIFO[…]
  B2_val load …
  Iterate (j …)
    val B_access = B_vals.deq
    Iterate (k …)
      … = B_access …
```

(a) Sparse matrix B_{ij} (b) B_{ij} stored in CSR (c) B_{ij}'s format description (d) Pseudocode for iterating over B_{ij}

Figure 6.8: Example sparse B matrix Figure 6.8a used in SDDMM with its corresponding data structure Figure 6.8b and format arrays Figure 6.8c. Figure 6.8d shows pseudocode generated by Stardust, where the colors correspond with the arrays in Figure 6.8b.

Stardust analyzes the memory needs of CIN to generate Spatial code. The algorithm recursively traverses the CIN. When the compiler sees a tensor access, it extracts that tensor's level format to identify the level's sub-arrays. Then, based on the access pattern of the tensor access and the capabilities of each sub-array, Stardust pins it to a physical memory type. Stardust starts with the sub-arrays pinned to an initial memory type based on the abstract memory region at the outermost access level. Then, as Stardust traverses the CIN, it propagates the sub-arrays outward to adjacent memory types in the hierarchy based on the following rules, which Stardust applies from the most strict to least strict:

Dense DRAMs. The system pins arrays of every off-chip tensor to dense DRAMs, which are initialized by the host.

Sparse DRAMs. These provide an interface for direct off-chip random accesses of sparse data. They are read-only DRAM with custom compression to optimize reads of closely-stored tiles. Stardust pins arrays to Sparse DRAMs when there is no identifiable working set to bring on-chip.

Dense SRAMs. The system only binds arrays with affine access patterns to dense SRAMs, including position arrays (addressed linearly) and values arrays of fully dense formats (which are generally traversed linearly).

Sparse SRAMs. These SRAMs include a reordering pipeline that dynamically schedules SRAM requests to avoid conflicting banks. The reordering is necessary as sparse access patterns are random, leading to many bank conflicts. Stardust pins any on-chip, small, fixed-size arrays that have an access pattern with reuse but random accesses to sparse SRAMs.

Bit Vectors. Bit vectors are on-chip integer streams that densely pack sparse coordinate information [176, 25]. Stardust automatically generates and manages bit vectors when two compressed tensor levels are being simultaneously traversed. Spatial requires the conversion from sparse coordinates to bit vectors for co-iteration, since the Capstan architecture does not support coordinate stream intersections. Capstan can only co-iterate through the scanning of bit vectors. We provide a more detailed explanation of how Stardust's co-iteration algorithm interacts with these bit vector memories in Section 6.7.

FIFO Buffers. Stardust may pin arrays accessed linearly with certain access patterns to FIFO buffers. Code that uses FIFOs cannot enqueue excess data that is not popped, and must pop data precisely when the storage lifetime ends. This restricts FIFOs to coordinate arrays of sparse tensors and value arrays that are accessed in order.

Registers. On-chip scalar variables are bound to registers.

The above memory pinning analysis does not consider array sizes, since it allocates the maximal possible size for one unit of memory. It assumes arrays fit based on the tiling to the accelerator as described by the scheduling language.

6.5.3 Memory Lifetime Analysis

Once the compiler has pinned a memory type to each array, it inserts code to allocate that memory and to transfer data to and from it. We can think of this data allocation and movement as larger tensors being partitioned into smaller chunks. The data from these smaller chunks are then copied and transferred to more local levels of memory, which is done automatically by Stardust. Stardust analyzes these two steps through one algorithm that determines the scope and lifetime of each sub-array.

Stardust ensures that sub-arrays are filled with elements before their use by adhering to three scoping rules:

1. The algorithm accesses value-arrays at the corresponding pattern body of the innermost tensor index.

2. Coordinate arrays are always accessed at the index pattern body corresponding to that coordinate's level.

3. Position arrays are always accessed one pattern higher than their corresponding coordinate array (with the highest array scope being the start of the kernel).

Our lowering algorithm must access value elements at the pattern body of the innermost tensor index, not lower in the pattern hierarchy. Stardust can not access array elements arbitrarily after array declarations (in subsequent scopes) because this is not possible for memories that do not support random access. Stardust addresses this scoping property by accessing the element and storing it into a temporary variable, which is used in place of the original value at an inner sub-scope. Using a FIFO for an in-order traversal of the levels, for example, requires that the FIFO values be accessed precisely at the level of its tensor access index and only used for one iteration of that loop. Figure 6.8d demonstrates in blue that if the value array of B_{ij} was bound to a FIFO with the iteration pattern of the computation as $\forall_i \forall_j \forall_k$, then `B_vals` array elements would have to be accessed in the j-loop (corresponding to B's last mode) instead of the k-loop (the innermost loop) as in imperative code. This hoisting behavior is also clear in Figure 6.4 on line 23, with the hoisted element used later in line 32.

Data Allocations The algorithm allocates tensor arrays within the pattern body just above the pattern with their first use by default. Emitting allocations immediately above where the variable is necessary allows for better compute efficiency and for ease of analysis, however, hoisting the memory allocations into outer patterns and inserting reset code between iterations is also possible. Lines 1, 4, and 6 in Figure 6.8d demonstrate this tensor memory allocation.

Data Transfers As Stardust allocates arrays, it will also emit the correct data transfer pattern between different memory types. The data transfer analysis actually determines when array elements are used in the code. Since data transfers must occur before array elements are needed, Stardust will place transfer code immediately after their associated allocations. Finally, Stardust generates data transfer code as different load and store keywords in Spatial. A transfer from a memory higher in the hierarchy to a lower one is a load; the inverse is a store. Depending on the exact memory types in the transfer, Stardust will emit a slightly different keyword (e.g., `store` from SRAM $\rightarrow$ DRAM vs. `store_stream_vec` for FIFO $\rightarrow$ DRAM as in Figure 6.4 line 38).

Putting the analysis all together, consider the SDDMM example in Figure 6.8 again. CIN describes the access B_{ij} as iteration over the index variables {i, j}, and the loops $\forall_i \forall_j$. The innermost level of B corresponds to the index variable j and iterates `B2_pos` and `B2_crd`. This means the generated Spatial code should access both `B_val` and `B2_crd` inside the j-pattern, so both arrays must be allocated right before in the i-pattern body. The code accesses position arrays one loop higher, meaning `B2_pos` is accessed in the i-loop and is allocated at the top (before any iteration patterns occur). The full algorithm can be found in Appendix D.

6.6 Mapping Computation to Hardware

Stardust approaches the mapping of computation similarly to the memory mapping described in previous sections. The abstract computation model in this work lets users reason about Spatial parallel patterns as special accelerator functions. Through the scheduling language, users control the mapping of sub-computations to certain parallel patterns, when there is ambiguity in which sub-computation should be generated for acceleration. During lowering, Stardust compiles the entire computation, including these accelerated sub-computation regions, to parallel patterns as described in Section 6.7.

Stardust models optimized computation on an accelerator backend as a function of that backend. The computation model allows users to pass in arguments, or metadata, to these backend functions through environment variables. Additionally, Stardust does not assume that these backend functions are necessarily parallel patterns. This computation model is general enough to represent hardware modules (units), accelerator kernels or function, parallel patterns, or accelerator instructions as backend functions in Stardust.

Users leverage Stardust schedules to reshape CIN sub-statements to expose sub-computations that can be mapped to high-level accelerated primitives, which in this case are the Spatial parallel patterns. Given any CIN statement S that includes a sub-statement S', where S' has an equivalent instruction f for a given platform, the scheduling language can transform S such that the sub-statement S' is isolated. Then, the sub-statement S' can be replaced and computed using the specialized pattern or function f for a given backend instead of being lowered directly to code. Since using the scheduling language to do all of this CIN reshaping and mapping may become tedious, we also provide a wrapper command for productivity that will also accelerate the computation.

Concretely, consider the simple vector-vector multiplication statement $(\forall_i a_i = b_i c_i)$ where the vectors start out off-chip. Assuming there exists an optimized multiplier $f_{\text{mul}}(\text{out}, \text{in}_1, \text{in}_2)$ for a given backend, the goal is to map the statement to that function f_{mul}. However, the vectors involved in the multiplication start off-chip, so the schedule must move all vectors on-chip first before mapping f_{mul}. We apply the following scheduling transforms to move the vectors on-chip and call the backend function.

The `map` command can be used in conjunction with the `precompute` command to optimize the kernel. The transformation is demonstrated by the following equations. Given the scheduling command
$c_1 \overset{\mathsf{def}}{=} \mathsf{precompute}(b_i * c_i, \{i\}, \{i\}, a^{\text{on}})$,

$$(\forall_i a_i = b_i * c_i) \xrightarrow{c_1} \begin{pmatrix} \forall_i a_i = a_i^{\text{on}} \\ \text{where } \forall_i a_i^{\text{on}} = b_i c_i \end{pmatrix}$$

transforms the vector multiplication to store into an on-chip result a^{on}. a^{on} is subsequently stored back off-chip into a.

Then, each off-chip input tensor needs a precompute on-chip so that the vector-vector multiplication is computed using only on-chip inputs. Given the command $c_2 \overset{\mathsf{def}}{=} \forall t \in \{b, c\}\ \mathsf{precompute}(t_i, \{i\}, \{i\}, t^{\text{on}})$, the transformation is

$$\begin{pmatrix} \forall_i a_i = a_i^{\text{on}} \\ \text{where } \forall_i a_i^{\text{on}} = b_i c_i \end{pmatrix} \xrightarrow{c_2} \begin{pmatrix} \forall_i a_i = a_i^{\text{on}} \\ \text{where } \forall_i a_i^{\text{on}} = b_i^{\text{on}} c_i^{\text{on}} \\ \text{where } \forall_i b_i^{\text{on}} = b_i \\ \text{where } \forall_i c_i^{\text{on}} = c_i \end{pmatrix},$$

where t^{on} denotes an on-chip tensor format and t denotes an off-chip format for all $t \in \text{tensors}(e)$.

Lastly, the vector-vector multiplication sub-expression maps to the vectorized multiplier f_{mul} using only on-chip tensors as operands $f_{\text{mul}}(a^{\text{on}}, b^{\text{on}}, c^{\text{on}})$. Given $c_3 \overset{\mathsf{def}}{=} \mathsf{map}(\forall_i a_i^{\text{on}} = b_i^{\text{on}} * c_i^{\text{on}}, \text{backend}, f_{\text{mul}})$, the transformation is

$$\begin{pmatrix} \forall_i a_i = a_i^{\text{on}} \\ \text{where } \forall_i a_i^{\text{on}} = b_i^{\text{on}} c_i^{\text{on}} \\ \text{where } \forall_i b_i^{\text{on}} = b_i \\ \text{where } \forall_i c_i^{\text{on}} = c_i \end{pmatrix} \xrightarrow{c_3} \begin{pmatrix} \forall_i a_i = a_i^{\text{on}} \\ \text{where } f_{\text{mul}}(a^{\text{on}}, b^{\text{on}}, c^{\text{on}}) \\ \text{s.t. } \mathsf{map}(\text{backend}, f_{\text{mul}}) \\ \text{where } \forall_i b_i^{\text{on}} = b_i \\ \text{where } \forall_i c_i^{\text{on}} = c_i \end{pmatrix}.$$

We also introduce a new `accelerate` scheduling command that composes all of these steps. `accelerate` is a compound command consisting of one or more `precompute` commands and a `map` command, and is necessary to map any sub-statement to a new backend function. Intuitively, the `accelerate` command first precomputes all off-chip tensors on-chip for a sub-statement that is being accelerated and then maps the on-chip tensors to the backend function f for substituted computation. Given that $S \overset{\text{def}}{=} \forall_{i*} a = e$, we define the `accelerate` transformation below.

$$S \xrightarrow{\mathsf{accelerate}(S', \text{backend}, f, c)} S_{\text{new}} \tag{6.1}$$

is equivalent to

$$\xrightarrow{c'_1 \; c'_2 \; c'_3} S_{new}, \tag{6.2}$$

where the c'_1, c'_2, and c'_3 commands are defined as variadic versions of the c_1, c_2, and c_3 commands respectively. Specifically,

$$c'_1 \overset{\text{def}}{=} \mathsf{precompute}(e, i*, i*, a^{\text{on}})$$
$$c'_2 \overset{\text{def}}{=} \text{For all } t \in \text{tensors}(e) \; \mathsf{precompute}(t_i, i*, i*, t^{\text{on}})$$
$$c'_3 \overset{\text{def}}{=} \mathsf{map}(S'[t^{\text{on}}/t \text{ for all } t \in \text{tensors}(S')], \text{backend}, f, c).$$

Finally, Stardust must pass accelerator and function metadata to the global scope of the generated code. Therefore, we introduce an `environment` command to set these metadata variables to values. Allowing `environments` in the scheduling language enables users to search the design space of kernels parameterized by these metadata values. We leverage this command in Section 6.8 to sweep our evaluated kernels for performance and improved resource utilization.

Our SDDMM example in Figure 6.6 shows the acceleration of reductions into registers in lines 21–23 and the configuration of parallelization factors in lines 17–18. Some scheduling commands to target accelerators from Stardust can be found in Table 3.1 and Table 6.1.

6.7 Stardust Compilation

Sparse accelerators speed up sparse tensor computations by contracting together and iterating through tensor elements efficiently, and a good compiler must support these algorithms natively. Stardust compiles these efficient sparse iterations to Spatial through a novel co-iteration rewrite system. The co-iteration is interleaved with the memory lowering in Section 6.7 to generate the final parallel-pattern code for Spatial as follows.

Table 6.1: New scheduling commands necessary in Stardust for targeting accelerators.

Scheduling Commands	**Description**
`map`$(S, \text{backend}, f, c)$	: Maps a CIN statement S to a backend-specific computation strategy (specialized block, function, pattern, or instruction) f with some optional constant factor, c.
$S \xrightarrow{\texttt{map}(S,\text{backend},f,c)}$	$f(\text{tensors}(S), V^{\dagger}, c)$ s.t. `map`$(\text{backend}, f)$
† Where V is the set of variables $\{i*, r*, \text{var}*\}$ defined by the scope of the CIN sub-tree right before the statement S.	
`accelerate` $(S, \text{backend}, f, c)$	: A compound scheduling command that accelerates a sub-statement S by precomputing all tensors of S into on-chip tensors for a new expression S' and then maps f onto S'.
`environment`(var, c)	: Sets a global hardware configuration variable to some value, c.
$S \xrightarrow{\texttt{environment}(\text{var},c)}$	S s.t. $\text{var} = c$

Environment variables set by the schedule are emitted first to be globally scoped. Next, Stardust recurses over CIN and replaces `map`-scheduled statements with their backend functions (see Figure 6.4 line 31 for an example of the generated `Reduce` function). Stardust then automatically lowers the remaining $\forall$ nodes to the correct parallel patterns depending on the rewrite system. For each $\forall$ node, the rewrite rules are applied to each tensor access that has that $\forall$ index.

Stardust uses the rewrite system shown in Table 6.2, for matching fused (sparse) iteration constructs to parallel patterns. The lowering mechanism recurses over the CIN and applies the rewrite rules for every CIN $\forall$ node. The iteration for each forall with index i, involves a single level of all tensors that have i in their tensor access. The rewrite system decomposes the iteration's fused tensor contraction set. The contraction set is rewritten into smaller tensor iterator contraction subsets based on the iterator formats for that level and the contraction type (intersection or union). The rewrite rules that decompose the contraction set stem from the type of iterations parallel patterns that Spatial supports (see Figure 6.9). The rewrite system uses set algebra to isolate binary iteration patterns of: dense iteration, single compressed tensor iteration, or compressed-compressed co-iteration. Then, it lowers to the correct parallel pattern.

Formally, the rewrite system has a set of iterator contractions I for a given $\forall$ node is $I = \mathcal{T}_1 \circ \mathcal{T}_2 \circ \cdots \circ \mathcal{T}_n$ s.t. where the contraction $\circ \in \{\cup, \cap\}$ and $n \geq 1$,. The format of an iterator

```
// Pattern Format: <Header> {<Indices> => <Body>}
// Uncompressed iteration and reduction
Foreach(len by inc par p) {i_dense => ...}
Reduce(reg)(len by inc par p) {i_dense => ...}
MemReduce(mem par mp)(len by inc par p) {i_dense => ...}
// Compressed single iteration (Reduce not shown)
Foreach(len by inc par p) {pos => ...}
Foreach(Scan(par=p, len=l, bitvector_A.deq))
    {A, i_crd => ...}
// Compressed-compressed coiteration (Reduce not shown)
Foreach(Scan(par=p, len=l, bitvector_A.deq,
       bitvector_b.deq)) {A, B, out, i_crd => ...}
```

Figure 6.9: Spatial parallel patterns for compressed and uncompressed iterations of an index. The parallel-pattern header and body with indices is shown.

contraction set is defined as format(I) = format$(\mathcal{T}_1) \circ$ format$(\mathcal{T}_2) \circ \ldots \circ$ format$(\mathcal{T}_n)$. The format of a tensor format$(\mathcal{T}_n)$ is defined as $\mathcal{C}_n$ for a compressed level, $\mathcal{B}_n$ for a bit vector level, and $\mathbb{U}$ the universe of coordinates for a dense level. Stardust applies the rewrite rules in Table 6.2 to I for every index. Lets look at the example of adding another matrix to SDDMM to demonstrate the rewrite system. The CIN for this computation is defined as $\forall_i \forall_j \forall_k B_{ij} C_{ik} * D_{kj} + E_{ij}$ where both E and B are CSR. The iterator contraction of level j is $I = E_2 \cup (B_2 \cap D_2)$. The format of I is $format(I) = \mathcal{C}_{E_2} \cup (\mathcal{C}_{B_2} \cap \mathbb{U})$. `lowerIter` is called on format of I, which will first call `lowerIter`$[\mathcal{C}_{B_2} \cap \mathbb{U}] \Rightarrow$ `lowerIter`$[\mathcal{C}_{B_2} \cap \mathbb{U}]$ and call lowerIter on the result of that

Special care is taken when Stardust generates the bit vector scanner parallel pattern (denoted by the `lowerIter`$[\mathcal{B}_1 \circ \mathcal{B}_2]$ rule). Two compressed bit vectors are either logically AND-ed for intersection or OR-ed for union by the sparse bit-vector `Scan` patterns. As the scanner processes the bit-vector data, it generates the following pattern indices: the position of A, position of B, the output position out, and the output coordinate i_crd. For each bit-vector iteration level, Stardust actually emits two scanner patterns: one to calculate the position sub-array entries by counting the number of nonzero results and the other to compute entries for the value sub-array. After the compiler emits the values scanner, it will traverse through the computation and use atomic accesses to sparse SRAMs for any value-array computation.

The compiler at this point generates the code within parallel-pattern bodies. The compiler lowers pattern bodies as: pattern indices that contain the iteration space of that pattern, memory allocations and data transfers as determined by Section 6.5, any other index calculations, and computation.

6.8 Evaluating End-to-End Dataflow Compilation

We demonstrate that Stardust compiled Spatial provides increased programmability, while still being comparable in performance to handwritten code. Stardust also enables the generation of many useful sparse kernels for Capstan, increasing the number of Capstan kernels by over 2× from the original work. For these newly generated kernels, we also show significant performance improvements when using Stardust to target an RDA over compiling to a CPU or GPU. Our evaluation increases the usability of Capstan (from the perspective of performance engineer programmability and algorithm expressibility), while providing the end-user performance improvements of an accelerator.

6.8.1 Methodology

We evaluate Stardust on a benchmark set that is completely new for Capstan. The benchmarks—listed in Table 6.3—are the same sparse tensor algebra expressions from Section 4.8 [108] with Stardust user schedules shown in Table 6.4. We profile CPU baselines on a 128-thread, four-socket Xeon E7-8890 v3 with a 32 KiB L1 data cache, 32 KiB L1 instruction cache, 256 KiB L2 cache, 46 080 KiB L3 cache, and 1024 GiB RAM. The machine runs Ubuntu 18.04.3 LTS and is clocked at 2494 MHz.

Table 6.2: General rewrite system that lowers tensor iteration contractions from forall nodes to parallel-patterns. Blue statements emit code and comp. stands for compressed.

	`lowerIter`[format(I)]	$\Rightarrow$ **emit** <backend block behavior>
Single-Iteration	`lowerIter`[$\mathbb{U}$]	$\Rightarrow$ **emit** `Foreach or Reduce(...=> i...)`
	`lowerIter`[$\mathcal{B}_1$]	$\Rightarrow$ **emit** scanner for result positions
		emit `Foreach(...=> pos...))`
	`lowerIter`[$\mathcal{C}_1$ **and** $\mathcal{T}_1$ is result]	$\Rightarrow$ **emit** $\mathcal{B}_1 =$ GENBITVECTOR($\mathcal{T}_1$)
		`lowerIter`($\mathcal{B}_1$)
	`lowerIter`[$\mathcal{C}_1$]	$\Rightarrow$ **emit** `Foreach(...=> pos...))`
Universe	`lowerIter`[$\mathbb{U} \cup \_$]	$\Rightarrow$ `lowerIter`($\mathbb{U}$)
	`lowerIter`[$\_ \cup \mathbb{U}$]	$\Rightarrow$ `lowerIter`($\mathbb{U}$)
	`lowerIter`[$\mathbb{U} \cap \mathbb{U}$]	$\Rightarrow$ `lowerIter`($\mathbb{U}$)
Compressed	`lowerIter`[$\mathcal{C}_1 \cap \mathbb{U}$]	$\Rightarrow$ `lowerIter`($\mathcal{C}_1$)
	`lowerIter`[$\mathbb{U} \cap \mathcal{C}_2$]	$\Rightarrow$ `lowerIter`($\mathcal{C}_2$)
Co-Iteration	`lowerIter`[$\mathcal{C}_1 \circ \mathcal{C}_2$]	$\Rightarrow$ **emit** $\mathcal{B}_1 =$ GENBITVECTOR($\mathcal{T}_1$)
		emit $\mathcal{B}_2 =$ GENBITVECTOR($\mathcal{T}_2$)
		`lowerIter`($\mathcal{B}_1 \circ \mathcal{B}_2$)
	`lowerIter`[$\mathcal{B}_1 \circ \mathcal{B}_2$]	$\Rightarrow$ **emit** scanner for result positions
		$\circ = \cup \Rightarrow$ **emit** `Foreach(Scan(...or...)`
		$\circ = \cap \Rightarrow$ **emit** `Foreach(Scan(...and...))`
Base	`lowerIter`[$\_$]	$\Rightarrow$ format($\mathcal{T}_{1k}$) = `lowerIter`($\mathcal{T}_1 \circ \ldots \circ \mathcal{T}_k$), largest $k \leq n$ that produces a match
		`lowerIter`(format($\mathcal{T}_{1k} \circ \ldots \circ \mathcal{T}_n$))

Table 6.3: The expressions used to evaluate Stardust. Sparse tensors are bolded.

		Lines of Code	
Name	**Expression**	**Input**	**Spatial**
SpMV	$y_i = \sum_j \mathbf{A}_{ij} x_j$	10	44
Plus3	$A_{ij} = \mathbf{B}_{ij} + \mathbf{C}_{ij} + \mathbf{D}_{ij}$	8	91
SDDMM	$\mathbf{A}_{ij} = \sum_k \mathbf{B}_{ij} C_{ik} D_{jk}$	17	62
Mat^{T}Mul	$y_i = \sum_j \alpha \mathbf{A}^T_{ji} x_j + \beta z_i$	13	50
Residual	$y_i = b_i - \sum_j \mathbf{A}_{ij} x_j$	9	48
TTV	$\mathbf{A}_{ij} = \sum_k \mathbf{B}_{ijk} c_k$	13	73
TTM	$\mathbf{A}_{ijk} = \sum_l \mathbf{B}_{ijl} C_{kl}$	11	83
MTTKRP	$\mathbf{A}_{ij} = \sum_{kl} \mathbf{B}_{ikl} C_{jk} D_{jl}$	15	86
InnerProd	$\alpha = \sum_{ijk} \mathbf{B}_{ijk} \mathbf{C}_{ijk}$	11	115
Plus2	$\mathbf{A}_{ijk} = \mathbf{B}_{ijk} + \mathbf{C}_{ijk}$	6	163

Table 6.4: User-provided schedules for the kernels in Table 6.3. Scalar promotion (`sPromote`) inserts a scalar workspace as a macro-scheduling command (instead of lines 21-22 in Figure 6.6) and `communicate` determines at which iteration pattern the result is communicated back off-chip [232]. `Parallel` is short for parallelize [181] and `env` is short for environment.

Name	Schedule
SpMV	stmt.parallel(j, Reduction, 16).sPromote().env("bp", 2)
Plus3	stmt.precompute(C(i,j)*D(i,j), {i, j}, {i, j}, ws)
SDDMM	See Figure 6.4 .
Mat^{T}Mul	stmt.parallel(j, Reduction, 16).sPromote().env("bp", 2)
Residual	stmt.parallel(j, Reduction, 16).sPromote().env("bp", 2)
TTV	stmt.accelerate(l, Reduction, 16).sPromote().communicate(A(i,j), j)
TTM	stmt.accelerate(l, Reduction, 16).sPromote().communicate(A(i,j,k), j)
MTTKRP	stmt.parallel(l, Reduction).parallel(k, Reduction).parallel(j, Reduction).sPromote().communicate(A, j)
InnerProd	stmt.parallel(l, Reduction, 16).parallel(k, Reduction, 16).parallel(j, Reduction, 16).sPromote() .communicate(A(i,j), j).env("bp", 2)
Plus2	stmt (default schedule)

Table 6.5: Capstan resources required by our compiled kernels. The specific limiting resource(s) are shown in bold type.

		PCU		PMU		MC		Shuf	
	Par	#	%	#	%	#	%	#	%
SpMV	16	44	(22 %)	41	(21 %)	35	(44 %)	**16**	**(100 %)**
Plus3	8	55	(28 %)	100	(50 %)	**58**	**(73 %)**	8	(50 %)
SDDMM	12	**163**	**(82 %)**	90	(45 %)	**61**	**(76 %)**	0	(0 %)
Mat$T$Mul	16	47	(24 %)	66	(33 %)	36	(45 %)	**16**	**(100 %)**
Residual	16	43	(22 %)	65	(33 %)	36	(45 %)	**16**	**(100 %)**
TTV	16	93	(47 %)	91	(46 %)	**67**	**(84 %)**	**16**	**(100 %)**
TTM	12	**161**	**(81 %)**	89	(45 %)	**70**	**(88 %)**	0	(0 %)
MTTKRP	8	**140**	**(70 %)**	70	(35 %)	**58**	**(73 %)**	0	(0 %)
InnerProd	8	53	(27 %)	**155**	**(78 %)**	**80**	**(100 %)**	0	(0 %)
Plus2	1	10	(5 %)	23	(12 %)	14	(18 %)	2	(13 %)

We compile TACO using GCC 7.4.0 with OpenMP enabled for the CPU baseline and NVCC version 10.0.0 for the GPU baseline. The GPU baselines run on an AWS EC2 p3.2xlarge instance with an NVIDIA V100 SXM-2 GPU. The GPU contains 64 KiB registers and 12 KiB L0 instruction cache per block, 128 KiB L1 data cache and shared memory and 2 KiB L1 constant cache per streaming multiprocessor, and 6144 KiB L2 cache. The device RAM is 16 160 MiB. The GPU has 84 Volta SMs and is clocked at up to 1328 MHz. We exclude data transfer time between the host and the GPU. We benchmark a single iteration with a cold cache. We evaluate Capstan applications with the same cycle-accurate simulator as in [176, 250], using an ideal memory model or Ramulator [104] to model four channels of DDR4-2133 or HBM-2E (at 1800 GB/s).

All evaluations use the datasets shown in Table A.2 in the Appendix. For most 2D kernels, we use the same SuiteSparse matrices demonstrated in the original Capstan paper for a fair comparison between Stardust generated and handwritten Capstan kernels [176]. However, Capstan's original architectural design does not perform well for highly sparse (less than about 5%) tensors. Therefore, we also generate synthetic datasets for Plus3, InnerProd, and Plus2 as described in detail in Appendix A.2.

We use CSR formats for all sparse 2D matrices and compressed sparse column (CSC) for Mat^{T}Mul.

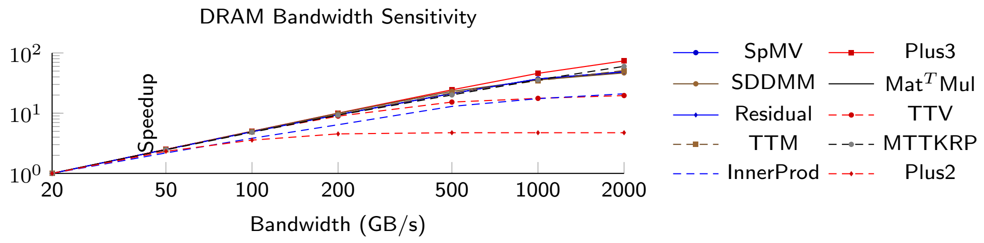


Figure 6.10: Impact of memory bandwidth on performance.

Table 6.6: Normalized runtimes (geomean of all datasets) to the compiled Capstan (HBM-2E) platform. We compile to Capstan, while CPU and GPU code is generated by TACO. Only SpMV, highlighted in gray, has handwritten kernels.

		Matrix Kernels					Tensor Kernels					
Platform (Memory)	**Compiled**	**SpMV**	**Plus3**	**SDDMM**	**Mat^{T}Mul**	**Residual**	**TTV**	**TTM**	**MTTKRP**	**InnerProd**	**Plus2**	**gmean**
Capstan (HBM2E)	No	0.65	—	—	—	—	—	—	—	—	—	0.65
Capstan (Ideal Net & Mem)	Yes	0.77	0.24	0.78	0.75	0.75	0.49	0.57	0.44	0.35	0.42	0.52
Capstan (HBM2E)	Yes	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
Capstan (DDR4)	Yes	12.13	10.07	8.33	12.31	12.06	4.92	9.80	7.76	3.28	1.72	7.09
Plasticine (HBM2E)	No	8.72	—	—	—	—	—	—	—	—	—	8.72
V100 GPU	Yes	3.15	41.89	18,259.50	3.59	3.54	232.85	284.47	6.77	2.76	381.38	41.31
128-Thread CPU	Yes	27.90	236.40	220.28	376.52	384.08	335.99	8.47	398.72	178.34	59.22	138.07

For 3D tensors, we use a CSR-like format for InnerProd and Plus2 and compressed sparse fiber (CSF) otherwise. We use the above formats for all platforms except the GPU baseline result tensors, which are fully dense since the TACO codebase [109] does not support sparse results for their GPU backend.

6.8.2 Resource Consumption

To understand which resources limit the performance of Stardust generated Spatial code, we provide some details about Capstan's design. Capstan is built as a grid of 200 vectorized compute units (PCUs) and 200 memory units (PMUs) with a surrounding ring of 80 memory controllers (MCs). Each PCU has six pipeline stages and 16 vector lanes that perform operations. Each PMU has 16 banks, supporting one read and write per bank per cycle. Capstan also has 16 shuffle networks (Shuf) that enable sparse accesses beyond the PMU, but they limit outer-level parallelism to 16.

To make good use of Capstan's hardware, a compiler must extract parallelism at both an inner-loop (vectorization) and outer-loop (cross-PCU) level. Outer-loop parallelism is harder to extract because it requires the compiler to explicitly manage distributed memories across physically unrolled partitions. Based on Capstan's distributed nature, it is unlikely that an application could use 100% of all on-chip resources. Limiting resources vary, but all applications except Plus2 make good use of resources via outer parallelization because they approach a limit in at least one resource dimension. By hitting physical resource limits, the compiler ensures that users can take full advantage of Capstan.

One key factor in RDA (and GPU) out-performance is a high-bandwidth memory system. Figure 6.10 shows that our applications (except Plus2, which is not outer-parallelized) are able to make good use of DRAM bandwidth as well. Spatial's decoupled access-execute memory model lets

Stardust factor out off-chip memory accesses into large, bulk transfers that expose significant memory parallelism.

6.8.3 Case Study: Sparse Matrix-Vector Multiplication

Sparse matrix-vector multiplication (SpMV) is the only application where a handwritten Spatial implementation exists. The first column of Table 6.6 provides a comparison of SpMV across all platforms. The Capstan and Plasticine rows from Table 6.6 are handwritten Spatial SpMV kernels from [176, 165], respectively. All Capstan and Plasticine rows in Table 6.6 that are not compiled (where the Compiled column is No) are handwritten Spatial SpMV kernels.

SpMV is simpler, making it easy to parallelize. Therefore, SpMV applications compiled by Stardust have a speedup relative baselines that is lower than other applications. However, the version of SpMV run in the original Capstan paper (Capstan, uncompiled) is more optimized than the compiled version (Capstan, compiled) because the code generated by Stardust uses the shuffle network (shown by SpMV×Shuf in Table 6.5) to coordinate parallel accesses to the input vector and the handwritten Capstan SpMV does not. Instead, the handwritten Capstan SpMV duplicates the input vector, which avoids shuffle-network contention and permits outer-parallelization beyond the shuffle network's limit of 16. We expect that these additional optimizations can be automated in the future, but for now they demonstrate that a dedicated hardware expert can get better performance compared to Stardust, albeit at the cost of significant development effort.

To demonstrate that Stardust both increases programmer productivity and decreases development effort in targeting Capstan, we compare the lines of code (LOC) of the handwritten Spatial SpMV kernel against the Stardust kernel for Capstan. The compiled SpMV kernel uses 10 input LOC total—a 76% decrease from the 52 lines of Spatial required for the handwritten version. Moreover, we believe that the input code to Stardust is simpler to write and to port to new architectures. The code required for Stardust includes: 3 LOC for the tensor formats, 2 LOC for the algorithm, 4 LOC for the scheduling transformations, and 1 LOC to compile and output our kernel. With the use of an auto-scheduler, which we leave as future work, the LOC could be cut down from 10 to 6 by removing the user-provided scheduling code. These numbers support the use of our compiler as a programmer productivity tool that enables the rapid development of new sparse tensor kernels for an RDA accelerator.

6.8.4 Tensor Algebra Expression Performance

Performance results for all platforms and applications are shown in Table 6.6, with a subset of results for only Stardust compiled Capstan, the TACO compiled GPU, and the TACO compiled CPU shown in Figure 6.11. Stardust compiled applications are, on average, 138× faster than CPU baselines, and 41× faster than GPU variants. These performance benefits further motivate using RDAs—and thus a compiler to target RDAs.

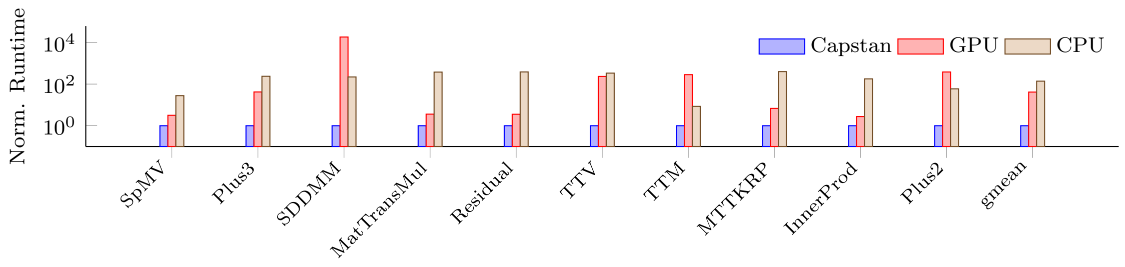


Figure 6.11: Performance of three select platforms normalized to Capstan.

Our TACO GPU baseline performance is significantly worse than both the literature [181] and compiled Capstan because TACO does not natively support sparse tensor outputs on the master branch of the system code base [109]. Most of the time is spent zero initializing the fully dense result tensor in device memory—which is often extremely large—on the host. Because Capstan is designed to outperform the GPU for sparse applications, it may seem counter-intuitive that the GPU speedup for MTTKRP is relatively low. However, these kernels have a dense dimension that the GPU can vectorize.

Currently, a comparison between compiled and handwritten implementations beyond SpMV is not possible since these kernels do not exist. The handwritten applications take considerable time to implement by an expert in Spatial, SARA, Capstan, and the domain of sparse applications. Our system is able to compile to 9 new applications, motivating the use of Stardust to generate new sparse tensor algebra kernels.

6.9 How Stardust Compares to Related Work

Stardust is, to the best of our knowledge, the first software stack to enable end-to-end compilation from tensor index notation to the architectural simulation of a reconfigurable sparse accelerator. There is, however, prior work on sparse tensor algebra systems targeting von Neumann architectures, domain-specific architectures that provide alternative targets for a compiler like ours, and different methodologies for programming these sparse DSAs.

Sparse Tensor Algebra Compilers for von Neumann architectures Several compilers have been proposed for sparse tensor algebra, but these compilers target CPUs (including the one presented in Chapter 3) [24, 116, 216, 108, 23, 242], GPUs [181, 242], and distributed machines of CPUs and GPUs [233] whereas our system compiles to domain-specific sparse dataflow hardware. Like many prior work compilers, we use an input API that follows a separation of concerns and start from an abstract loop-based IR. Stardust is unique, however, because it emits code with sparse iteration on bit vectors, memory management, and parallel patterns in the Spatial programming model.

Sparse Domain-Specific Hardware Many fixed-function accelerators have been proposed for sparse kernels [78, 184, 213, 249, 71, 201, 246, 252, 157], however, we will focus our discussion on reconfigurable sparse accelerators as they need for compilation. Our system targets Capstan [176] because it is a flexible RDA with an easy-to-understand programming model: it supports sparse iteration with composable parallel patterns. However, sparse iteration spaces are a general representation, and the ideas from Stardust could influence the software stack of any reconfigurable sparse accelerator. The SPU [45] and ExTensor [81] are two recent sparse DSAs with a different programming model than Capstan. Both are tiled architectures with explicit on-chip memory accesses, but they have different methods for combining sparse data. The SPU uses a stream-join programming abstraction in C code to combine sparse indices and a custom RDA fabric to perform the intersection operations. Similarly, ExTensor uses a programming model based on hierarchical tensor intersectors that are programmed through hardware configurations.

Programming Sparse Dataflow Architectures The Spatial compiler [111] and the idiomatic spatial accelerator compiler of Weng et al. [223] show how to compile high-level control-flow languages to sparse RDAs and CGRAs. The Custard compiler presented in Section 4.7, on the other hand, shows how to compile sparse tensor algebra to an abstract machine representing reconfigurable streaming dataflow accelerators. The Mosaic compiler described in the following chapter shows how to isolate a sparse tensor algebra sub-expression and to call out to a user-defined external function on that sub-expression. Our work is the first of these compilers to identify and compile a tensor index notation sub-expression all the way to an RDA.

Chapter 7

Scaling to Multiple Accelerators

"The whole is greater than the sum of its parts."

Aristotle

The ideas and systems in the preceding chapters lay the foundation for compiling high-level sparse tensor programs to individual accelerators. Each chapter progressively expanded the scope of programmability: Stardust established a compiler path that lowered imperative loops to a pre-existing dataflow architecture, SAM provided a unified sparse dataflow abstraction for multiple sparse dataflow architectures, and Onyx demonstrated a fabricated hardware backend for that abstraction. Together, they demonstrated how a careful choice of abstraction enables tractable end-to-end compilation to a single sparse accelerator.

However, a central tension arises from the fact that most sparse accelerators and high-performance computing platforms are programmed through *libraries*, not *compilers*. Libraries such as BLAS, cuSPARSE, oneAPI (MKL), and other vendor-specific kernel APIs serve as the defacto interfaces to specialized compute engines.[1] They provide exceptional performance, but also impose rigid boundaries between algorithmic expression and implementation, preventing automatic composition across heterogeneous systems. In contrast, the compiler-based abstractions developed in earlier chapters—such as the dataflow model of SAM and the imperative-to-dataflow compilation in Stardust—enable reasoning about sparse computations at a higher level of abstraction, but cannot directly interoperate with these library-driven systems.

Achieving high performance on sparse computation extends far beyond a single device or single library API. As the ecosystem of accelerators, hardware fabrics, and dataflow engines continues to expand, developers increasingly face a fragmented landscape of programming systems (see Chapter 2). Each accelerator exposes a distinct software interface, data layout, and execution model. This

[1]Evidence of this proliferation and performance is demonstrated by the fact that these libraries serve as the state-of-the-art baselines in many of the compiler evaluations throughout this dissertation.

fragmentation limits portability and reuse: performance is often trapped within hand-optimized or hand-tuned libraries, and programming systems remain isolated to the hardware they target.

Mosaic bridges these two worlds: a sparse programming system that can target both compilers and libraries, and in doing so, can target many accelerators simultaneously. It introduces an interoperable compiler that composes generated code with existing accelerator libraries through a unified abstraction layer (Figure 7.1). Rather than targeting a single hardware substrate, Mosaic allows a sparse program to be compiled into a mosaic of heterogeneous kernels with glue code. Some of the code is generated by the compiler, while others are invoked by the compiler as external functions. The external functions can be hand-written library kernels or generated by compilers themselves, like SAM and Stardust. Through careful abstraction design, Mosaic can still leverage compiler approaches to reason about external library calls at a higher level of abstraction. In doing so, Mosaic extends the compiler model of sparse programming to an ecosystem where multiple accelerators, each with its own library API, can coexist and cooperate. This unified framework enables compiler- and library-based programming systems to share a common intermediate representation, permitting automatic verification, mapping, and code generation across diverse backends.

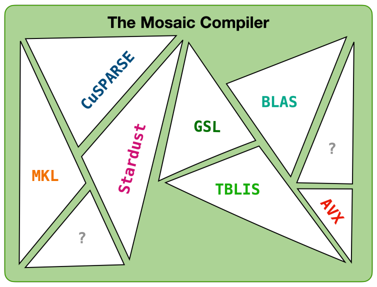


Figure 7.1: The Mosaic compiler composes a Mosaic of external functions with compiler glue code.

The design of Mosaic heavily influenced newer work that targets domains beyond sparse tensor algebra. The REPTILE compiler [209, 207] applies Mosaic's capability-based sub-expression matching to call external high-performance libraries for tiled recurrence relations and solvers. REPTILE demonstrates that Mosaic's external-library abstraction generalizes to other compilers, application domains, and a new class of hand-optimized library kernels.

Beyond its immediate contributions and impact, *Mosaic* further redefines how the research community should think about domain-specific programming systems. Rather than a tension between competing compiler and library approaches, domain-specific programming systems should interoperate between them. By unifying these trajectories, Mosaic demonstrates that a compiler need not replace existing ecosystems to achieve generality—it can integrate them. This perspective opens several new directions for both compiler and architecture research. For compiler researchers, Mosaic establishes a foundation for *meta-compilation*: automatically composing multiple code generators and library backends through verifiable capability descriptions. For architects, it suggests a new interface layer between hardware and software, where accelerators can expose their computational capabilities declaratively rather than through opaque APIs. In doing so, Mosaic reframes performance portability as a problem of semantic compatibility, enabling future systems to reason about how to combine specialized hardware and software rather than choose between them.

With Mosaic, the programming stack for sparse accelerators scales from *one accelerator* to *many*, and from a single execution model to a continuum of imperative, dataflow, and library-driven backends. The following sections describe the abstractions, algorithms, and interfaces that make such composition possible, and demonstrate how we achieve unification across heterogeneous accelerators through a common compiler infrastructure.

7.1 The Need for Interoperable Sparse Compilation

Sparse tensor algebra is an important computational language that describes multilinear expressions over dense and sparse tensors. It is used in many domains, including scientific computing, engineering, and machine learning. Performance is crucial in these domains, resulting in a proliferation of libraries for CPUs [126, 95, 90, 224, 69], GPUs [149, 46], vector processors [94, 38, 203], and domain-specific hardware [176, 45, 81, 99, 246, 169, 35, 202, 157, 201, 80]. These libraries have been optimized at great expense and effort. As a result, most tensor algebra expressions are written as a sequence of calls to libraries that compute different sub-expressions.

Fusing operations can lead to better performance, thus breaking library boundaries, as demonstrated in the previous chapters (Sections 3.7, 4.8 and 6.8) and prior sparse compilers [4, 108, 232]. Fused sparse tensor algebra operations may even have better asymptotic complexity. Moreover, generating bespoke code for an expression and specializing the loop order can avoid expensive tensor transposes and reshapes. Due to the cost of developing specialized fused operations, researchers have proposed compilers that can automatically generate fused code for both dense and sparse tensor algebra [108, 23, 242, 148, 253, 210]. Although these compilers generalize to any tensor algebra expression, they cannot match the performance of libraries for expressions that are sufficiently important to have been hand optimized. For example, dense matrix multiplication when implemented on specialized hardware can result in a speedup of two orders of magnitude. Therefore, the best performance for an expression may require fused code (e.g., as in sparse sampled dense-dense matrix multiplication), calls to libraries (e.g., as in dense matrix multiplication), or a mix of fused code and calls to libraries.

No current sparse tensor algebra compiler can mix generated specialized and fused code with calls to libraries or domain-specific hardware, leaving application developers to write low-level code by hand. The application developer must write code to traverse and compute on sparse data structures, fuse sparse expressions, and tile, transpose, and reshape sparse tensors to fit library APIs. Writing such code is a laborious and error-prone process. Moreover, since the performance benefits of loop ordering, tiling strategy, and external functions depend on both the data and the machine, it is necessary to explore many different optimization strategies in a large design space. Without compiler support to automate the low-level code generation and to assist with this design-space exploration, application developers leave performance on the table.

Even with a way to automatically generate a mix of fused code and library calls, writing an optimal program is still challenging. Due to the sheer number of libraries available, there is a combinatorial explosion in the number of choices available for code generation. Enabling programmers to specify schedules offers a rich space of optimizations and provides the possibility of incorporating domain expertise [170]. However, programmers must completely specify all transformations, including picking constants to tile with and loops to reorder or fuse. Some users may not have the time or expertise to write such precise schedules. On the other hand, a completely automatic scheduler that tunes every parameter can be too slow for quick design space exploration.

Prior work on sparse tensor algebra systems has only solved parts of the problem of mixing specialized/fused sparse tensor algebra code with calls to libraries and hardware. Several compilers have been developed that can generate specialized and fused imperative code for sparse tensor algebra expressions, including TACO [108], the MLIR SparseTensor Dialect [23], COMET [148], SparseTIR [242], and the Sparse Polyhedral Framework [204, 216]. Other systems reduce sparse linear or tensor algebra expressions to a fixed set of library calls, such as MATLAB [137], Julia [21], TTB [12, 13], and CTF [194, 193]. Moreover, the AMOS compiler [254] generates code that mixes bespoke code with calls to domain-specific hardware for dense tensor algebra. And, the Exo compiler [93] lets users manually compose different instructions to implement algorithms that can be expressed with affine loops, which includes dense tensor algebra. However, none can currently provide both fusion and function calls from external libraries for sparse tensor algebra compilation.

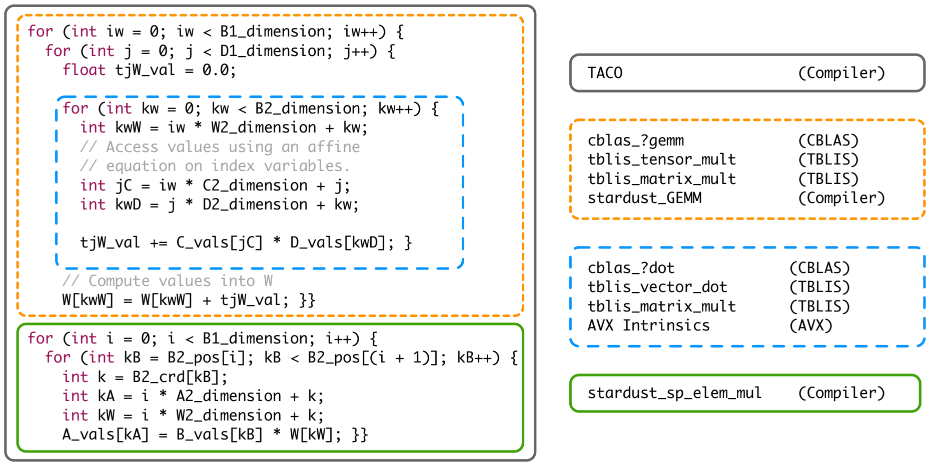
We propose Mosaic, a modular compiler for sparse tensor algebra that users can extend with library functions and specialized hardware to externally compute whole expressions or sub-expressions. Building on prior work on the TACO compiler [108], Mosaic can also generate specialized and fused imperative code for those expressions or sub-expressions where no suitable external function or hardware exists. With Mosaic, users can write partial schedules that map a sub-expression to an external function. Mosaic checks whether the mapping is valid against a user-provided specification, tiles and reshapes the expression to fit within the constraints of the function, and then completes the schedule. However, it is up to the user to ensure that the external function actually computes what the user-provided specification claims. To further increase programmer productivity, we also provide a completely automated search mechanism. This mechanism provides a list of valid schedules, but does not select the most performant schedule.

Mosaic is a productive and interactive system that helps developers identify ways to map an expression to external functions, produces code to transform data structures to match external function APIs, and generates any parts of the expression that cannot be mapped to external functions. Our contributions are:

- An *external function interface* that defines the algorithm to generate code targeting a given external function and specifies the tensor algebra expressions it can compute;
- *Composable scheduling commands* that map sub-expressions to external functions;

Table 7.1: The tensor algebra systems (and their features) used in Section 7.8 to evaluate Mosaic. Mosaic composes function calls from different libraries with generated code to compute any sparse tensor algebra expressions.

Tensor Algebra System	System Type (Language)	Features				
		Data Representation			Tensor	Platform
		Arbitrary Order	Dense	Sparse	Properties	
`Intel AVX` [94]	Intrinsic (C)	✘	✔	✘	✘	CPU
`BLAS` [126]	Library (C)	✘	✔	✘	✔	CPU
`Intel MKL` [95]	Library (C)	✘	✔	✔	✔	CPU
`GSL` [69]	Library (C)	✘	✔	✘	✔	CPU
`TBLIS` [138]	Library (C)	✔	✔	✘	✘	CPU
`TACO` [108]	Compiler (C/C++)	✔	✔	✔	✘	CPU
`cuSPARSE` [149]	Library (C)	✘	✔	✔	✘	GPU
`Stardust` (Chapter 6)	Compiler (Scala/Spatial)	✔	✔	✔	✘	Capstan [176]


```
for (int iw = 0; iw < B1_dimension; iw++) {
  for (int j = 0; j < D1_dimension; j++) {
    float tjW_val = 0.0;

    for (int kw = 0; kw < B2_dimension; kw++) {
      int kwW = iw * W2_dimension + kw;
      // Access values using an affine
      // equation on index variables.
      int jC = iw * C2_dimension + j;
      int kwD = j * D2_dimension + kw;

      tjW_val += C_vals[jC] * D_vals[kwD]; }
    // Compute values into W
    W[kwW] = W[kwW] + tjW_val; }}
for (int i = 0; i < B1_dimension; i++) {
  for (int kB = B2_pos[i]; kB < B2_pos[(i + 1)]; kB++) {
    int k = B2_crd[kB];
    int kA = i * A2_dimension + k;
    int kW = i * W2_dimension + k;
    A_vals[kA] = B_vals[kB] * W[kW]; }}
```

```
TACO                      (Compiler)

cblas_?gemm               (CBLAS)
tblis_tensor_mult         (TBLIS)
tblis_matrix_mult         (TBLIS)
stardust_GEMM             (Compiler)

cblas_?dot                (CBLAS)
tblis_vector_dot          (TBLIS)
tblis_matrix_mult         (TBLIS)
AVX Intrinsics            (AVX)

stardust_sp_elem_mul      (Compiler)
```

Figure 7.2: Generated code for sampled dense-dense matrix multiplication (SDDMM). Color-coded boxes indicate external functions that could be used for corresponding sub-computations.

- An *automatic verification and mapping algorithm* that rewrites expressions to identify valid mappings and fills in partial schedules; and
- A *code generation algorithm* that generates external function calls wherever a mapping has been made and generates code natively for sub-computations that have not been mapped.

To evaluate our contributions, we show that Mosaic can outperform a homogeneous compiler, sometimes even adding two orders of magnitude speedup. We also demonstrate the generality of our approach by adding 38 external functions from eight existing tensor algebra systems (see Table 7.1) and compiling tensor algebra expressions that cannot be fully computed by just composing calls to different functions. Finally, we show that Mosaic's search system finds many bindings that require reshaping operators within an expression.

7.2 Motivating Example

Consider a user who wants to compute Alternating Least Squares [115] and needs a fast implementation of sampled dense-dense matrix multiplication (SDDMM). SDDMM is expressed in tensor index notation as $A_{ij} = \sum_k B_{ij} \cdot C_{ik} \cdot D_{kj}$, i.e., the dense multiplication of tensors C and D is sampled using the sparse matrix B. Figure 7.2 shows an implementation of SDDMM and a number of possible library functions that can be used to replace sub-computations. Using these functions, there are 19 possible ways to rewrite the existing code.

In order to rewrite the code to utilize external functions, users must refactor the surrounding program to interface correctly with each function. Computations that are mapped to a function need to be isolated from the rest of the program and the inputs and outputs to different computations need to be rewired. Users must also observe the functions' calling conventions and initialize system-specific objects. If inputs need to be tiled to fit in specialized memories, more finicky changes (indexing correctly into each operand) are required. Writing optimized code for a fixed set of functions is already challenging, doing the same for 19 possible function placements is unfeasible.

Users can describe possible function placements for SDDMM by choosing one of the three options in Mosaic:

```
// SDDMM in einsum notation.
stmt=A(i,j)=B(i,j)*C(i,k)*D(k,j)
// Precompute C*D in W and use
// iw,jw,kw as index vars in the code.
stmt.precompute(C*D,W,{i,j,k},
                      {iw,jw,kw})
// Split loop kw by 4 into ki and ko.
     .split(kw,ko,ki,4)
// Push ki to be the inner most loop.
     .reorder(iw, jw, ko, ki)
// Consider iw, jw to be constant.
     .fix(iw,jw)
// Bind the reduction of kw to AVXAdd().
     .bind(AVXAdd(), C*D)
```

Figure 7.3: Full schedule for targeting SDDMM to AVX vector add.

```
stmt = stmt.map(AVXAdd(), C*D)
```

Figure 7.4: Partial Schedule for targeting SDDMM to AVX vector add.

```
// Register AVX to Mosaic.
register(AVXAdd());
vec<stmts> schedules =
         stmt.getAllSchedules();
// Pick a schedule to apply.
A.compile(schedules[i])
```

Figure 7.5: Automatically search and find all schedules that use AVX add.

A full schedule: Users, such as performance engineers, may know exactly which functions to utilize and how to tile and reshape tensors in the sub-expression to meet the constraints of the function. Such users can specify these transformations precisely and then bind a sub-expression to an external function using the `bind` scheduling command (Figure 7.3). In this case, Mosaic ensures that every transformation and function mapping is correct.

A partial schedule: If a user wants to try a particular function for a fixed sub-expression, but does not know whether any code transformations are required to do so, they can use the `map` scheduling command (Figure 7.4). Mosaic will automatically discover a valid binding (if possible) by tiling and reshaping the tensors of that fixed sub-expression to match the constraints of the function.

An automatically generated schedule: Finally, if the user wants to explore the design space,

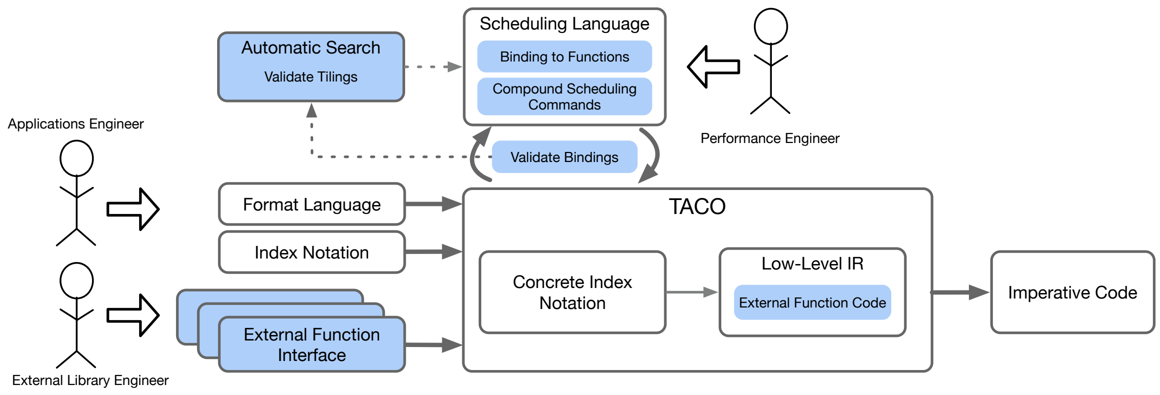


Figure 7.6: System overview of our Mosaic compiler with blue components signifying new contributions. Dotted arrows signify the automatic mapping loop, which an end-user may optionally enable. Mosaic enables independent development among experts: application engineers may write programs, performance engineers may pick function substitutions, and system developers may add external libraries.

Mosaic can automatically search the space of possible mappings (lines 2-3, Figure 7.5) and return a list of valid schedules. Then, the user can select one schedule out of all possible schedules (line 6, Figure 7.5).

7.3 Mosaic System Overview

Mosaic compiles tensor algebra expressions to a mix of natively generated code and external function calls. That is, while lowering a tensor algebra expression, it glues together library functions, filling in any blanks where no function is available with generated code. In this way, it gives users the ability to write performant code using highly optimized functions, while preserving generality. Figure 7.6 shows how different pieces of Mosaic interact, and we describe each piece below.

Mosaic is implemented as an extension to the TACO compiler [108], an open-source domain-specific compiler for sparse tensor algebra embedded in C++. As in TACO, users describe computations using tensor index notation (Einsum notation), a format language [41], and a scheduling language [181]. These languages combine to form concrete index notation (CIN), an abstract loop-based IR [106]. We add another domain-specific input language called the *external function interface* that adds new library functions to Mosaic. We extend the scheduling language with commands that specify function placement.

External functions are added as extensions through an external function interface (described in Section 7.4). In this work, an external function is any function that computes one or more tensor algebra expressions. Some libraries with functions that fall into this category are listed in Table 7.1. The external function interface provides two key pieces of information in order to correctly substitute a sub-computation with a call to an external function: the calling convention associated with the function and a description of the function's capability. A function's capability is characterized by the set of tensor algebra expressions it computes and the constraints it imposes over those expressions.

Mosaic characterizes a function's capability using a *capability description*. The external function interface of a function only needs to be written once for a single function, therefore, other users can include pre-written descriptions like a library.

Mosaic introduces new *scheduling commands* (Section 7.5) that integrate with TACO's scheduling framework, which consists of commands that form a composable set of rewrite rules that can transform concrete index notation expressions by reordering, splitting and fusing loops. With the additional commands provided by Mosaic, users can also bind computation to external functions. Or, users can map a computation to a function, and Mosaic will automatically discover any valid tiling or reshaping transformation that permits this mapping. Thus, Mosaic can complete under-specified schedules to expose valid mappings. Before binding to a function, Mosaic also validates mappings by translating constraints defined on tensor order and dimension to an SMT query.

Finally, we also provide a fully *automatic mapping* (Section 7.6) solution to end-users. Validity and speed are the two core pillars of the automatic mapping process. To ensure validity, we check whether the schedule matches the function's capabilities using an SMT solver. The capability language is critical for not only validity, but also the speed of the mapping process. By exploiting information embedded in the compute capability language to guide the search, Mosaic can apply guided rewrites to expose valid mappings. Using this solution, users can register the plugin and transparently schedule existing TACO programs to a combination of different functions in two simple lines of code. The automatic-mapping solution only returns a list of valid mappings; users can add an autoscheduler that ranks or prioritizes the returned mappings. We leave this autoscheduler as future work.

Extensibility and modularity are at the core of Mosaic. Users can not only extend Mosaic by adding external functions, but they can also change Mosaic's default code generation algorithm by adding a code generator. Users can add an autoscheduler that selects a mapping made by Mosaic's automatic mapping process. Each component of Mosaic—the user program, schedules, and plugins—is independent of the other. A user can write a program, system experts can write external function plugins, and performance engineers can write schedules.

7.4 External Function Interface

Users of Mosaic can add a new external function by supplying a description of its calling convention and compute capabilities. The calling convention tells Mosaic how to call the function. The compute capabilities description tells Mosaic what set of tensor algebra expressions the external function can compute, and is used to verify the correctness of user-requested bindings and to guide an automated search.

To add new external functions to Mosaic, users write an *external function interface*. A sample interface for vector addition using AVX intrinsics is given in Figure 7.7. Each external function interface consists of seven pieces of information:

A calling description that provides the name, return type, and arguments for the external function (line 6, Figure 7.7).

Setup code that is called before calling the function. The setup code may initialize specialized memories, pack data into function-specific data structures and user-defined formats, allocate additional memory, and configure meta-data values (line 9, Figure 7.7).

Teardown code that is called after the function. The teardown code can be used to check for error codes, unpack data from function-specific data structures and user-defined formats, and free allocated memory (line 12, Figure 7.7).

The function capabilities describe the expressions the function computes (line 14, Figure 7.7).

Include paths are paths to files where setup and teardown code is declared (line 17, Figure 7.7).

Library paths denote the path to the shared library where the external function and functions called during setup and teardown are defined (line 19, Figure 7.7).

Build flags denote any Makefile flags that should be used to compile the code (line 21, Figure 7.7).

An external function interface need only be written once for each external function. Because an external function interface is simply a C++ class, users may also embed code generators as plugins to Mosaic. Interfaces for code generators will not have a fixed definition or declaration at mapping time, but will emit a concrete definition or declaration when a successful mapping is identified.

7.4.1 Calling Convention

A calling description (Line 6, Figure 7.7) provides information that is used to generate code to call an external function. It consists of a *name*, *return type*, and a *list of arguments*. Arguments can be other calling descriptions, library objects, or tensor metadata. Using the recursive definition of arguments that includes other calling descriptions, users can perform any pre-processing or format change on function arguments. Additionally, libraries like TBLIS and GSL pack data into special structs that must be initialized before the function call. Through the addition of library objects as argument types, Mosaic can declare such objects during the code generation phase. These objects are then used by later parts of the code. Finally, tensor metadata includes commonly used parameters like an array of dimensions, the dimension of a particular rank, and the array of tensor values.

7.4.2 Compute Capabilities

When replacing sub-expressions with external functions, Mosaic ensures only valid substitutions are performed. That is, Mosaic determines whether a sub-computation lies in the space of a *function's*

```
class VecSumAVX : public FunctionAbstraction {
 public:
  GslTensorPlus() : x(Tensor(Float32, 1, {Dense})) ... {}
                             // datatype, #dimensions, format for dimension.
  // CallingDescriptions written in Mosaic's IR.
  CallingDescription getCallingDescription(){return x_avx = _mm256_add_ps(y_avx, z_avx);}
  // Initialize y_avx, z_avx.
  vector<CallingDescription> getSetup(){return {y_avx = _mm256_load_ps(y),
                                            z_avx = _mm256_load_ps(z)}}
  // Store result into x from x_avx.
  vector<CallingDescription> getTeardown(){return {_mm256_storeu_ps(y, &x_avx)};}
  FuncCapabilityStmt getFuncCapabilities(){stmt = x(i) = y(i) + z(i);
                                        return stmt.where(i==8);} // Constraint length of i.
  // Name and path of include file.
  vector<string, string> getIncludePaths(){return {{"immintrin.h", path}}}
  // Name and path of shared library object.  For AVX, we have none.
  vector<string> getLibraryPaths(){return {};}
  // Add "-mavx2" to the Makefile flags.
  vector<string> getCFlags(){return {"-mavx2"}}
 private:
    Tensor x, y, z;
    FuncObject x_avx, y_avx;}
```

Figure 7.7: Sample external function abstraction for `_mm256_add_ps` in Intel AVX intrinsics written in C++. Users inherit from Mosaic's abstract class `FuncAbstraction`. Each virtual function corresponds to a field of the external function abstraction.

capabilities. A function's capabilities are the set of expressions a function can calculate, as well as any constraints that the inputs must satisfy.

Specifying the semantics of external functions for sparse tensor computations presents a unique challenge. While some functions compute a single expression, others can calculate an infinite number of expressions. And, some functions are only suitable for tensors with a specified mathematical property or a restricted sparsity pattern. In addition, constraints of exotic hardware must also be expressible. To capture such a wide variety of interface semantics, three components work together:

A capability language: An tensor index notation language with added support to describe tensors with unspecified rank and to impose constraints on expressions.

A checker function: A user-defined function that takes as input an expression and returns true or false indicating a successful or failed match.

Tensor properties: A specification of the accepted tensor operands, including formats, mathematical properties, and sparsity patterns.

Capability Language

The capability language, shown in Figure 7.8, describes the expressions that a function can compute. The capability language expands upon tensor index notation (or einsum notation) and can describe classes of expressions by defining tensors with unspecified rank. For such expressions, constraints over dimension size (the size of each rank) and order (the number of ranks) are described using

Index Variable	i	*Concrete Index Variable List*	$i*$
Constant Integer	int	*Index into an Index-Variable List*	index

Capability Description	CD	::=	INS when $ILS \mid INS$
Index Notation Statement	INS	::=	$\text{forall}_i\ S \mid a = E \mid a \mathrel{+}= E$
Index Notation Expression	E	::=	$a \mid \text{int} \mid E + E \mid E * E \mid \ldots$
Index List Statement	ILS	::=	$jc \mid \forall(i, \mathcal{I}, ILS) \mid \exists(i, \mathcal{I}, ILS)$
Joint Condition	jc	::=	$c \wedge c \mid c \vee c$
Tensor Accesses	a	::=	$\mathcal{T}(\mathcal{I}) \mid \mathcal{T}(i*)$
Dynamic Index Variable List	$\mathcal{I}$	::=	$\mathcal{I}\,\mathcal{I} \mid i\,\mathcal{I} \mid \mathcal{I}\,i \mid \mathsf{range}(\ldots) \mid \mathsf{range}(\text{int}\ldots) \mid$ $\mathsf{range}(\ldots\text{int}) \mid \mathsf{range}(\text{int}\ldots\text{int})$
Condition	c	::=	$e \texttt{ != } e \mid e \texttt{ == } e \mid e \texttt{ <= } e \mid e \texttt{ >= } e \mid e \texttt{ < } e \mid e \texttt{ > } e$
Binary Op	b	::=	$e + e \mid e - e \mid e * e \mid e/e \mid e\%e$
Index List Expression	e	::=	$b \mid p \mid \text{int}$
Property	p	::=	$\mathsf{order}(\mathcal{I}) \mid \mathsf{dimension}(\mathcal{I}(\text{index})) \mid \mathsf{product}(\mathcal{I})$

Figure 7.8: Context-free grammar of our compute capability language, which augments index notation to include dynamic lists of index variables with constraints.

set-builder notation. Therefore, the capability language can describe both functions that compute a fixed expression, like the `cblas_saxpy` vector addition function from the `CBLAS` library, and functions that compute many expressions, like the `tblis_tensor_mult` any-order tensor contraction function from the `TBLIS` library.

The compute capability language defines the capabilities of functions that can compute a class of expressions by letting users index into tensors with an unspecified number of ranks. In order to achieve this, the capability language allows tensors to be indexed by a dynamically sized index variable list (see Dynamic Index Variable List in Figure 7.8) in addition to a fixed set of indices.

A dynamic index variable list may need to satisfy certain constraints in order to lie in the capability of a function. For example, Stardust—a tensor algebra compiler targeting specialized hardware—requires that the total number of values in a tensor does not exceed 65,536 (due to memory constraints) even though it does not restrict tensor rank. In this case, no matter what the tensor rank, the product of each index variable's dimension in the index list must not exceed 65,536. To specify such constraints over a dynamically sized index variable list, the capability language provides three language constructs:

1. `condition`: A `condition` node can describe constraints over index variables and properties of a dynamically sized index list like the order of an index list, the product of the dimension of index variables in an index list, etc.

2. $\forall$(`iterator`, `index list`, `condition`) : The $\forall$ node describes a condition that must be true for all elements of a dynamically sized index list.

3. $\exists$(`iterator`, `index list`, `condition`) : The $\exists$ node describes a condition that must be true for at least one element of a dynamically sized index list.

Therefore, the capability description $\texttt{product}(\mathcal{I}) < 65,536$, where $\mathcal{I}$ is the dynamically sized index list used to index into tensors in Stardust's description, can be used to specify the memory restriction of Stardust mappings.

In addition to iterating through index lists, the capability language also provides a concatenation operation for index lists that can be used to concatenate index lists with index variables or other index lists. By doing so, it is possible to specify different constraints on the indices being concatenated. For example, consider a tensor $\mathcal{T}$ with unfixed rank using IJ where $I = range(...)$ (i.e. a dynamically sized index list) and $J = [m, n]$ where m, n are concrete index variables. Due to the concatenation of I with J, the minimum order of $\mathcal{T}$ is set to 2. When matching $\mathcal{T}$ to a concrete tensor, A, that is used in the expression we want to schedule, the last two index variables that are being used to index into A are captured by m and n. Any constraints on m and n are then applied to the captured index variables.

Checker Function

As there are constraints that the capability language cannot express, Mosaic supplements the capability language with a checker function. The checker function is a C++ function that takes an expression written in tensor index notation and returns a boolean indicating whether the expression lies within the function's capability. Checker functions can be used to describe subtle constraints of highly specialized emerging hardware. For example, the checker function can be used to reject a function mapping for specialized hardware if the hardware is already in use.

Although the checker function is strictly more expressive than the compute capability language, it is opaque to other parts of Mosaic. If an external function writer only provides a checker function, we would need to call the checker function to check for a match on every transformation for every sub-expression for every function, limiting automatic mapping solutions from exploiting the structure of the function capabilities to guide the search. By combining both, we get full expressibility through the checker function, and fast search space exploration through the language. Working together, these two approaches enable us to do a coarse-grained match using the language and then an optional fine-grained match using the checker function.

Tensor Properties

Most external functions can only compute on input tensors of a specified format. In Mosaic, similar to TACO, users specify the formats of tensors used in the capability description using the Format Language [41] introduced in prior work.

In order to ensure correctness, Mosaic also gives users the ability to annotate a tensor with a *property*. Some functions may further restrict input tensors that have special mathematical properties or sparsity patterns. For example, `MKL` has a matrix multiply function optimized for Hermitian matrices. In this case, Mosaic must ensure that only a Hermitian matrix is mapped to this function. Users can tag tensors with properties to indicate such special characteristics. Properties have no semantic meaning associated with them. This tagging system can be used to indicate different types of sparsity, and mathematical properties.

To tag tensors with properties, we provide a `tag` argument to the tensor format. We show an example of a dense symmetric matrix: `Tensor<float>T("T",{dense, dense},Property::symmetric)`. Additionally, we give users a way to describe the interaction between different tags and operations to avoid tedious user tagging. Mosaic allows users to describe these rules using an index expression language variant that index expression with tag objects instead of tensor objects. Mosaic then propagates tags based on the user-provided tag rules.

7.5 Binding Expressions to External Functions

Many modern domain-specific programming systems use scheduling languages to guide program optimization [170, 36, 181, 243, 31]. A clean scheduling language separates the rewrite system and code generation from the decisions about what rewrites to apply. This design greatly increases the productivity of performance engineers and simplifies work on automatic optimizations. The scheduling commands typically include classical compiler loop optimizations (interchange, strip-mining, flattening, tiling, vectorization, and parallelization), but also commands to move computations into or out of loop nests such as Halide's compute-at command or TACO's precompute command.

Mosaic extends TACO's scheduling language [106, 181, 232] with new commands that can be used to bind a tensor algebra expression to an external function. These commands can be used to substitute either full expressions or sub-expressions with an external function. Mosaic will generate any code that is needed to transform the sparse data structures of tensors to match the function's API (see Section 7.7). In addition, Mosaic will verify, using an SMT solver, that the sub-expression conforms to the compute capabilities of the function.

7.5.1 The Bind Command

The bind command replaces a sub-expression in a tensor algebra expression with an external function after Mosaic has verified that the replacement is valid. As a result, Mosaic's code generator emits code that calls that function, instead of generating imperative code to compute the sub-expression. Mosaic also generates supporting code to transform the arguments to the function into the data structures expected by the function and code to transform the result back to the data structures specified by the result tensor.

The bind command `S'` = `S.bind(s, f)` is applied to a statement `S`, given in the concrete index notation, which it then rewrites. It takes as its inputs the sub-statement `s` to replace and the function `f` to replace it with. The bind command either returns a rewritten statement `S'` or an error message if the index expression implemented by `s` is not in the capability set of `f`.

```
// Generate Z3 query from Capability Language's AST
void GenerateZ3Query(ASTNode node){
  switch(node->type){
    case ∀(i,𝓘,ILS):
      for(i ∈ Concrete(𝓘)):
       conditions[i] = GenerateZ3Query(ILS(i))
      emit z3.And(conditions.join(","))
      break
   case ∃(i,𝓘,ILS):
      for(i ∈ Concrete(𝓘)):
       conditions[i] = GenerateZ3Query(ILS(i))
      emit z3.Or(conditions.join(","))
      break
    case (Joint_Condition):
      //node.op is ∧ or ∨.
      emit GenerateZ3Query(node.RHS) node.op (GenerateZ3Query(node.LHS))
      break
    case (Condition):
      //node.op is ==,!=,≤,≥,>,<.
      emit GenerateZ3Query(node.RHS) node.op GenerateZ3Query(node.LHS)
      break
    case (Binary_Op):
      //node.op is +,−,%,∗,/.
      emit GenerateZ3Query(node.op_1) node.op GenerateZ3Query(node.op_2)
      break
    case (....)
```

Figure 7.9: Z3 code generation for a subset of the compute capability AST nodes (defined in Figure 7.8). Mosaic builds an SMT query that validates a given binding.

Binding Validation

To ensure the correctness of binding `s` to `f`, Mosaic validates the binding against two things that are associated with `f`: the compute capability description, written in the compute capability language, and the checker function, written in C++. If `f` has a checker function and it returns false when `s` is supplied as an input, then Mosaic cannot bind `s` to `f`. If the checker function returns true, Mosaic validates the binding against the capability description.

There are four steps to check whether `s` lies in the space of expressions defined by the capability description of `f`. First, Mosaic ensures that the datatypes of the operands of `f` matches that of the tensors in `s`. Second, Mosaic checks whether the operators used in `s` are the same as the operators used in the capability description. If these two checks pass, then Mosaic can associate each tensor in `s` with a tensor in the capability description. Third, for each associated tensor, Mosaic matches the tensor's index variables used in the user-provided expression (concrete index variables) to the tensor's index variable used in the capability language (abstract index variables). Fourth, Mosaic checks whether all constraints described over the abstract index variables are satisfied by the assigned concrete index variables. To do so, Mosaic explicitly enumerates the corresponding constraints over each assigned concrete index variable and generates code targeting the Z3 theorem prover [144]. Figure 7.9 shows how Mosaic generates the theorem prover inputs for this step of the validation. After step two, there may be several options for matching concrete index variables to abstract index variables due to the concatenation operator in the compute capability language. If the selected matching passes the rest of the validation steps, we bind `s` to `f`. Otherwise, we test other possible matchings.

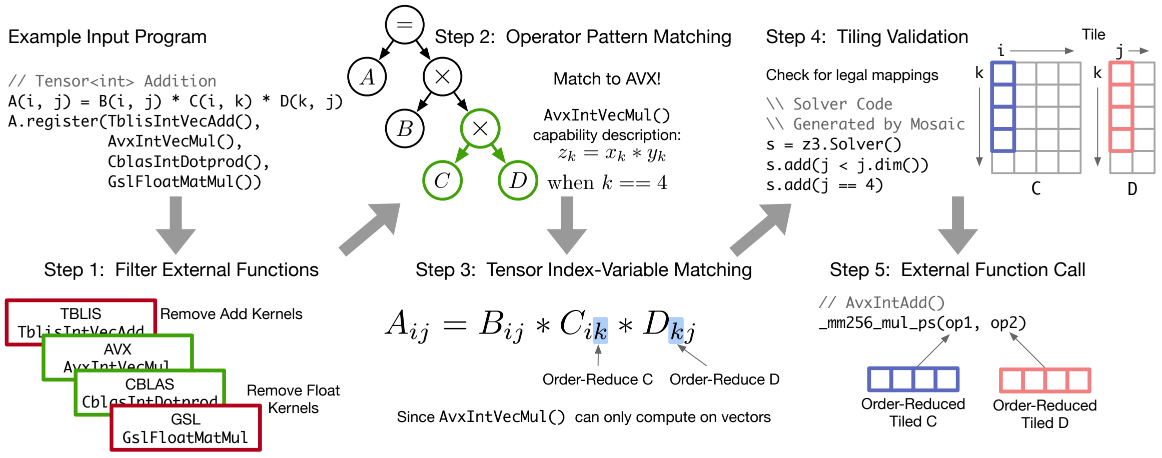
7.5.2 The Map Command

To increase productivity, Mosaic provides a compound command, `map`, that provides partial scheduling automation. When provided with a sub-expression, `map` is tasked with binding the whole sub-expression to `f`. Unlike `bind`, where users must explicitly specify how the computation gets associated with the interface and are tasked with tiling, reshaping, and precomputing the appropriate expressions, `map` tiles and reshapes the sub-expression to fit within the constraints of a specified function. Users do not need to search, read, or understand the minute details of the documentation for the external function they want to use. Essentially, using the map command, the user isolates a sub-expression that exactly matches f and leaves the rest of the transformations that are necessary for correctness, such as tiling and reshaping, to Mosaic.

The `map` command is applied to a concrete index notation statement and takes as input a sub-statement `s` and an interface `f` to produce `S'`, the result of applying `S.map(s, f)`. It uses several other scheduling commands to rewrite the CIN statement and target the sub-statement to an external interface. Some of these commands include existing commands (`split`, `fuse`, `precompute`, `reorder` and `parallelize`) from the TACO scheduling API [181]. However, we also add the `promote` and `fix` commands to Mosaic. These commands modify the sub-expressions, either by changing the shape of tensors or the scope of the sub-expression, to fit the hardened capabilities of an external function. The `map` command calls the automatic search at Step 3 of the automatic searching algorithm (Section 7.6.2) to transform the expression and validate the mapping. In Section 7.6, we describe how our automatic search machinery factorizes sub-expressions from a larger expression to bind to different functions.

Promotion

The `promote` command adds an additional dimension of size 1 to a given tensor at the provided index, creating an equivalent tensor with higher order. This command enables `map` to use functions that operate on higher-order tensors to target computations that have lower-order inputs. For example, `map` can bind matrix-multiply functions to matrix-vector computations with `promote`. `Promote` is called on a CIN statement as `S.promote(`$\mathcal{T}$`, int)` where $\mathcal{T}$ is a tensor and `int` specifies the position that the extra dimension is inserted at. For a tensor of order n, int can range from [0, n] inclusive since a dimension can be inserted between any n index variables.



```
// Tensor<int> Addition
A(i, j) = B(i, j) * C(i, k) * D(k, j)
A.register(TblisIntVecAdd(),
           AvxIntVecMul(),
           CblasIntDotprod(),
           GslFloatMatMul())
```

```
\\ Solver Code
\\ Generated by Mosaic
s = z3.Solver()
s.add(j < j.dim())
s.add(j == 4)
```

```
// AvxIntAdd()
_mm256_mul_ps(op1, op2)
```

Figure 7.10: Steps in the automatic searching process of Mosaic.

Fix

The **`fix`** command enables **`map`** to target functions operating on lower-order tensors to computations that use high-order inputs. For example, the contraction at index j of a matrix-matrix multiply computation $A_{ik} = \sum_j B_{ij} * C_{jk}$ can be computed using a dot product function if indices for each i and k combination are fixed. The **`fix`** command specifies the index variables to ignore while mapping the computation. These indices are effectively held constant for the mapped computation.

7.6 Automated Search for Bindings

While the scheduling commands in Section 7.5 provide users explicit control over function placement, some users may want to replace such fine-grained control with more automation. For example, a user may not have the time or expertise to make scheduling decisions, or have legacy code that they want to speedup without much effort. To meet the needs of such users, Mosaic provides an automated search mechanism that finds all valid bindings. Given a set of registered external functions, the automated search mechanism returns all the schedules that use those functions.

The search machinery consists of five steps shown in Figure 7.10. First, it filters all external functions whose parameter and return datatypes are different from the user-specified computation. Second, it applies mathematical rewrites, looking for equivalent operator patterns (see Section 7.6.1). Third, it matches index variables in the compute capability description to index variables in the user-defined statement (see Section 7.6.2), effectively resolving conditions on tensor order. Fourth, it performs a tiling validation step that explores and validates potential index variable tilings (see Section 7.6.3) and ensures that constraints on dimension size are satisfied. Fifth, a check against the checker function described in Section 7.4 is performed. If the check returns true, the search is complete and the external function is called during code generation. If not, the search algorithm considers all permutations of rewrites up to a provided depth.

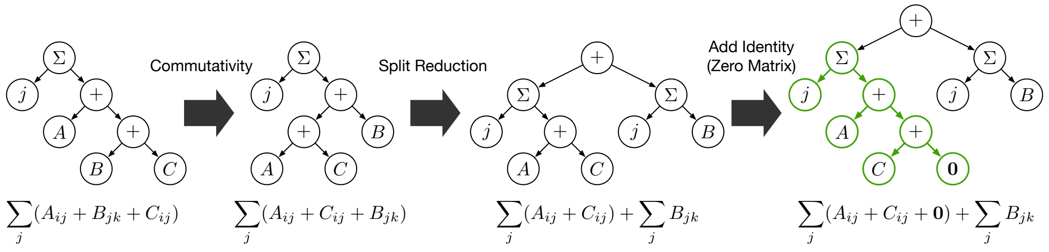

Commutativity
Split Reduction
Add Identity (Zero Matrix)
$\sum_j (A_{ij} + B_{jk} + C_{ij})$
$\sum_j (A_{ij} + C_{ij} + B_{jk})$
$\sum_j (A_{ij} + C_{ij}) + \sum_j B_{jk}$
$\sum_j (A_{ij} + C_{ij} + \mathbf{0}) + \sum_j B_{jk}$

Figure 7.11: Series of mathematical rewrites to map to a function that has the capability: $\sum_j A_{ij} + B_{ij} + C_{ij}$.

Optionally, users can tell Mosaic to pick a schedule out of all possible schedules at random. Therefore, with just two additional lines of code, users can target their legacy code to other functions through Mosaic's automatic search mechanism. Programmers may also choose to add an autoscheduler that ranks all possible scheduling options, which we leave as future work.

7.6.1 Operator Pattern Matching

The goal of this stage is to simply match the operator(s) in the user-provided sub-expression with the operator(s) in the external functions. The search algorithm ignores all information about tensor order and dimension leaving them to be resolved at a later stage. For example, if the user-defined expression performs an addition of two tensors, any function that computes additions of two tensors of any order (where a scalar is a tensor of order 0) will be identified as a potential match. Note that since tensor algebra also includes reductions over index variables, Mosaic considers reduction as another operator to match for correctness. For example, in the expression $a_i = b_i \cdot c_i$, substituting $b_i \cdot c_i$ with a dot product $\alpha = \sum_i x_i \cdot y_i$ will give an incorrect result even though the element-wise multiplication $(\cdot)$ operator pattern matches.

To expose matches, the algorithm applies basic mathematical rewrites (see Figure 7.11 for an example) such as distributivity, associativity, commutativity, splitting reductions (when all operands of the reductions are being added) and adding an identity operand. These rewrites are guided by the compute capability description. For example, if a function targets the addition of three operands, and the user-provided expression consists of two operands, adding an identity tensor is more fruitful than applying a commutativity or associativity transformation. Since Mosaic is rewriting expressions at a high-level mathematical IR (concrete index notation), it is easy to validate that the rewrites preserve the semantic meaning of the original expression.

7.6.2 Tensor Index-Variable Matching

At this stage, the operator pattern of the sub-statement is the same as that of the compute capability of the function, and Mosaic can associate each operand in the compute capability description with a corresponding tensor in the user-defined expression. For each pair of associated tensors, the algorithm checks whether it is possible to match the index variables used to index into the two tensors. To do so, Mosaic must check that (1) any reduction or free variable is matched with another reduction or free variable respectively, and (2) once an index variable has been mapped, it is mapped to the same index across all other tensor accesses.

However, there are many possible ways of selecting matching index variables. These choices emerge because of tensor reshape transformations and the concatenation operation between dynamically sized index lists. Because of the `fix` command, Mosaic can restrict the combination of indices under consideration. And, because of the `promote` command, Mosaic can increase the order of the tensor, giving the algorithm more indices to work with. As a heuristic, Mosaic never `promotes` the order of the tensor to be greater than the minimum order required by the function. Moreover, the concatenation operation between dynamically sized index lists can introduce several choices for how to split indices into concatenated dynamically sized index lists.

We use a brute-force search to enumerate index-variable choices. In practice, exhaustively searching this space is fast, adding a negligible slow-down in compilation time. This low overhead is due to two reasons. First, the tensors we need to schedule rarely have order greater than 4. In fact, only 5% of distinct expressions inputted into the TACO website [107] contain tensors of order greater than 4. Second, we have not seen an interface that needs more than one dynamically sized index list. With only one list, the mapping for dynamically sized index lists is fixed, and there is no search space to explore. A list of legal mappings is sent to the next stage.

7.6.3 Tiling Validation

After the search mechanism has assigned each index variable to another variable or added it to a dynamic index list, any constraints that have been specified over the indices using the compute capability language (described in Figure 7.8) must be satisfied. To check that the constraints are satisfied, Mosaic generates a query targeting the Z3 theorem prover using the algorithm described in Figure 7.9 and validation from Section 7.5.1. However, this query is too restrictive as it neglects potential tilings of index variables. To include such tilings, we loosen the constraint that the index variable size must be equal to the size of the dimension it indexes into, and now permit sizes of index variables that are less than the sizes of their respective dimensions. With this weaker constraint, the algorithm checks for the satisfiability of the new model. If the model is unsatisfiable, Mosaic will give up. But, if the model is satisfiable, Mosaic will query the model to return valid values for every index variable's dimensions, adding an additional constraint requiring that the product of future tilings exceeds the product of previously returned tilings and stopping once an unsatisfiable model is reached. The largest tiling (the tiling that gives the largest product of the dimensions of the index variables) is selected as the final dimensions of the tiled tensor.

```
// Full Plus3T Expression
A(i, j, k) = B(i, j, k) + C(i, j, k) + D(i, j, k)

// Schedule sub-computation to TblisPlus3 which calculates
// X_I = Y_I + X_I where I = range(...)
A.getSchedule().bind(B(i, j, k)+C(i, j, k), TblisPlus3())
```

```
// Code for Plus3T where B(i,j,k) + C(i,j,k)
// is mapped to tblis_add_tensor.
void compute(taco_tensor_t * A, B, C, D){
 // Emit variables to access tensor metadata
 int A1_dim = A->dimensions[0];
 float * A_vals = (float *) B->vals;
 ...
 // Declare and init workspace tensor W.
 float*W=malloc(sizeof(float)*num_val);
 for (int i = 0; ...)
  for (int j = 0; ...)
   for (int k = 0; ...)
    int index = ((i*C2_dim)+j)*C3_dim+k
    // Copy the second operand into W.
    W[index] = C_vals[index];

 // Emit object declarations.
 tblis_tensor t1; tblis_tensor t2;
 // Emit setup functions.
 tblis_init_tensor_helper_s(&t1, B_dim, 3, B);
 tblis_init_tensor_helper_s(&t2, W_dims, 3, W);

 // Call function that computes
 // and stores result into t2
 tblis_tensor_add(NULL,NULL,&t1,"ijk",&t2,"ijk");

 // No teardown, compute result into W array

 // Compute A_ijk=W_ijk+D_ijk with TACO code
 for (int i = 0; ...)
   for (int j = 0; ...)
     for (int k = 0; ...)
       int index = ((i*C2_dim)+j)*C3_dim+k
       // Copy the second operand into W.
       A_vals[index] = W[index]+D_vals[index];

 // Free the workspace tensor.
 ... }
```

Figure 7.12: Generated code for a scheduled Plus3T computation.

7.7 Mosaic Code Generation

In this section, we describe the code generation for the `map` command. The code generation produces code that calls external functions for any sub-expressions that have been mapped, orchestrates data movement between the function and surrounding code, and generates code to natively compute sub-computations that have not been mapped.

The `map` command given in Section 7.5 replaces the sub-expression that is being computed by the function with a workspace tensor. First, this workspace tensor is declared (line 9, Figure 7.12). The result of the function call will be stored in the workspace tensor. Functions may store the result tensor back into an input operand. For example, `tblis_tensor_add` in the `TBLIS` library stores the result of the tensor addition into the second operand. So, before the code generation can start emitting code for the function call, we may need to copy the operand into the result workspace tensor (lines 11-16, Figure 7.12). During the next stages of code generation, the second operand is replaced by the workspace tensor.

Next, the compiler emits setup-function calls. Arguments are recursively lowered, and special user-defined objects used as arguments are declared before calls to setup functions. Similarly, the compiler emits calls to perform the computation and the teardown (lines 20-29, Figure 7.12). As an optimization, when the whole expression can be targeted to a single external function, the code generation does not use a temporary workspace to store the result of the function call. The original result tensor is used as the result tensor for the function call.

If no external function interfaces have been registered to Mosaic, then we default to TACO's code generation algorithm. TACO expresses the whole range of tensor algebra expressions and produces fused code by default. For sparse inputs, fused code can run asymptotically faster, while mapping computations to separate functions results in unfused code. A default compiler producing fast fused code gives users an additional option to choose the extent of fused versus unfused code.

7.8 Evaluating Interoperable Tensor Compilation

We evaluate the performance of Mosaic's generated code and its ability to search and find successful mappings. We contrast the performance of Mosaic's default code generator (TACO), to expressions that are lowered to a mix of calls to external functions and TACO code using Mosaic. Our results show that there are regimes where fused code is the most performant and other regimes where a mix of generated code and external function calls is the most performant. Thus, we provide evidence for the utility of Mosaic's ability to mix generated code with calls to libraries.

Mosaic also enables a quick and systematic search over the design space created by the combination of sub-expressions and external functions. We demonstrate the quality of our search by evaluating along two axes: the number of discovered bindings and the speed of the automatic mapper. We also demonstrate that by dividing the specification of a function's capability into a checker function and

Table 7.2: The full expressions used to evaluate our compiler where sparse tensors are bolded.

Name	Expression	Name	Expression	Name	Expression
`GEMV`	$a_i = \sum_j B_{ij} c_j$	`GEMM`	$A_{ij} = \sum_j B_{ij} C_{jk}$	`SDDMM`	$A_{ij} = \sum_k \mathbf{B}_{ij} C_{ik} D_{jk}$
`SpMV`	$a_i = \sum_j \mathbf{B}_{ij} c_j$	`SpMM`	$A_{ij} = \sum_j \mathbf{B}_{ij} C_{jk}$	`TTTP4`	$A_{ijkl} = \sum_m B_{ijkl} * C_{im} D_{jm} E_{km} F_{lm}$
`TTV`	$A_{ij} = \sum_k B_{ijk} c_k$	`Block-Sparse SpMM`	$A_{ijkl} = \mathbf{B}_{ijmn} * C_{mnkl}$	`SpMMAdd`	$A_{ij} = \mathbf{B}_{ij} + \mathbf{C}_{ij}$

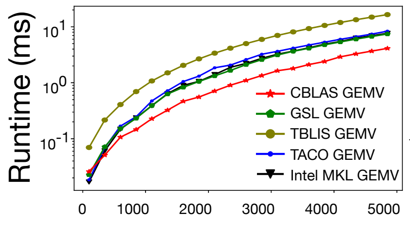
a compute capability language, we get both expressibility and search speed.

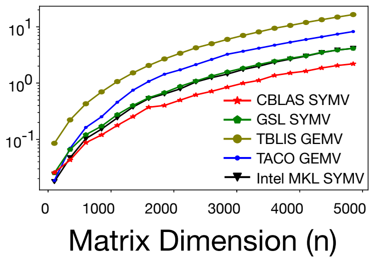
7.8.1 Methodology

We evaluate Mosaic on real-world tensor expressions from prior work [108, 187] (see Table 7.2). The studies in Section 7.8.2, Section 7.8.3, and Section 7.8.4 use dense or uniformly random sparse synthetic data. Synthetic data lets us control and vary computation regimes, and demonstrate their underlying performance tradeoffs. We sweep the tensor dimension and sparsity (where applicable) of the synthetic data. All tensors are square with tensor dimension n referring to the size of all tensor ranks, and varying sparsity refers to changing the percentage of nonzero values. We also include benchmarks in Section 7.8.3 run on real-world sparse matrices from the SuiteSparse Matrix Collection [49] listed in Table 7.3.

For our evaluation, we register 38 external functions from eight tensor algebra systems to Mosaic. Table 7.1 lists each tensor algebra system's features, backend platform, and programming language. AVX [94] is a set of single instruction multiple data (SIMD) extensions to the x86 instruction set made available through Intel Intrinsics. CBLAS [126], GSL [69], and Intel MKL [95] are all fixed-function, linear algebra libraries. We use the OpenBLAS implementation for the CBLAS library. GSL calls BLAS headers that are implemented by the Automatically Tuned Linear Algebra Software (ATLAS) project [224] under the hood. TBLIS [138] is a library for dense tensor operations based on the BLIS framework [215]. As opposed to translating tensor operations into matrix operations and using BLAS to compute the decomposed pieces, TBLIS decomposes the operation into simpler, optimized functions. The cuSPARSE library [149] is designed to run on NVIDIA GPUs and provides subroutines that can be used to compute linear algebra expressions on sparse matrices. As described in Chapter 6, Stardust compiles sparse tensor algebra to the Capstan reconfigurable dataflow accelerator [176]. Stardust is unique in our evaluation since it is a compiler, is not natively embedded in the C ecosystem, and compiles to a hardware accelerator backend. We use the same methodology as in the original Stardust and Capstan papers [176] as described in Section 6.8.1, with details of the evaluation methodology provided below.

Stardust is written in C++ and generates Spatial code, a Scala-embedded DSL for accelerator design [111]. All Spatial applications are compiled [251] and run using the same cycle-accurate Capstan simulator modeling 4 channels of HBM-2E (at 1800 GB/s) [104] and a non-ideal network [250]. All experiments are run on an AWS EC2 g4dn.xlarge instance with an NVIDIA Tesla T4 GPU and a four-socket Xeon(R) Platinum 8259CL CPU. The CPU has a 64 KiB L1 data and instruction cache, 2000 KiB L2 cache, 3580 KiB L3 cache, and 16082 MB RAM. The machine runs Ubuntu 22.04.1





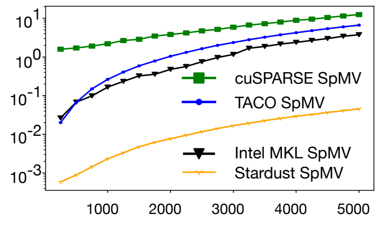


Figure 7.13: Log performance for varying matrix dimensions when using Mosaic to bind functions to dense GEMV.

Figure 7.14: Log performance across matrix dimension for SYMV (a dense, symmetric GEMV).

Figure 7.15: Log performance across various matrix dimension for SpMV where $\%\text{nnz}_B = 20\%$.

LTS and is clocked at 2500 MHz. All code is compiled using GCC 11.3.0 with `-O3 -ffast-math` optimizations on. For all benchmarks, we report the median runtime of 10 iterations with 10 warmup iterations. To generate pure TACO code, we simply do not register any plugins to Mosaic, kicking off its default code generator.

7.8.2 Comparing Systems

In this section, we quantitatively show that hand-written functions and generated code from different systems have performance benefits in different situations. Thus, we motivate a system like Mosaic that can automatically map expressions to a mix of external functions and generated code. We compare the performance of several systems on matrix-vector multiplication using three types of matrices: a dense matrix (GEMV), a symmetric matrix (SYMV), and a sparse matrix (SpMV). Across the three types of matrix data, no one system (including TACO) is sufficient to outperform the rest. Therefore, it is beneficial for users to explore several options through Mosaic to find the most performant implementation for their use-case.

For dense GEMV, calling a CBLAS and GSL function provides an average of 1.9× and 1.1× speedup respectively (Figure 7.13) over the code generated by TACO. This difference is to be expected since TACO is not optimized for dense code. TACO computes GEMV using a naive implementation whereas BLAS is a 40-year-old, hand-optimized library and GSL uses an ATLAS implementation of BLAS, which tunes for machine-specific quirks.

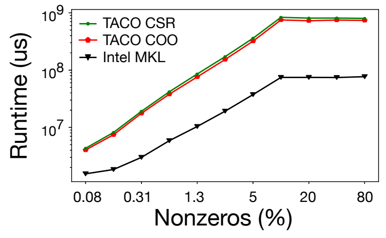


Figure 7.16: Log performance comparing tensor formats coordinate list (COO) and compressed sparse row (CSR) for SpMMAdd, where $n = 10,000$.

For the symmetric GEMV computation, we notice an average of 2.9× and 1.8× speedup when we use the BLAS and MKL library implementations respectively over TACO-generated code (Figure 7.14). The three libraries—MKL, GSL and CBLAS—have

special functions for computing matrix-vector products with sym-
metric matrices (SYMV). SYMV implementations allow these systems to deliver better performance over TACO by exploiting the mathematical structure of the data.

For the SpMV computation (Figure 7.15), we cannot compare against GSL and CBLAS as these libraries do not have functions that perform sparse linear algebra operations. In Figure 7.15, we see an order of magnitude difference between the Stardust and TACO runtimes because Stardust compiles to the Capstan accelerator, a dataflow architecture built for sparse computations. As the GPU is built for inference, we do not see an advantage of targeting cuSPARSE.

Finally, we also show the benefits of paying the penalty of runtime format conversion to target an external library. Figure 7.16 shows that the TACO implementation for SpMMAdd using the COO format is slightly more performant than the one using the CSR format. However, when the COO matrix is converted to CSR at runtime to meet the format constraints of the MKL library, we see a maximum speedup of 10.1× over the TACO COO implementation.

Figures 7.13 to 7.15 show performance variation due to the underlying mathematical properties of the data, the availability of specialized hardware, and the difference in data structure formats. This section shows that systems optimize for different subsets of these factors, and Sections 7.8.3 and 7.8.4 will show that this specialization can be leveraged by generating performant code through Mosaic.

7.8.3 Leveraging System Specialization: SDDMM

For expressions involving sparse tensors, even for a fixed expression and fixed format, a fixed mapping does not always yield maximum performance benefits. As the sparsity of the tensor changes, the optimal function to map to a sub-expression also changes. Because of this behavior, we show that there exist computations that benefit from both fused and factorized code optimizations and they can be scheduled through Mosaic.

We compare the performance of sampled dense-dense matrix multiplication (SDDMM, see Table 7.2) across various tensor dimensions and sparsities over a set of external functions mapped using Mosaic. We chose SDDMM since it is a core building block, and often the bottleneck, for many applications including graph learning, matrix completion, and alternating least squares.

For a low percentage of nonzeros $\%\text{nnz}_B < 0.24\%$, Figure 7.17 shows that fused code generated by TACO generally outperforms code that calls external functions. When mapping SDDMM to a dense-dense matrix multiplication function, Mosaic needs to insert a temporary to store the result of the function call. This insertion produces unfused code: CD is computed first and then multiplied by the compressed tensor B. The resulting asymptotic complexity is proportional to the total number of dense elements ($O(n^3)$ here). TACO, on the other hand, produces fused code, multiplying elements of C and D only when there is a corresponding nonzero in the sparse input B. The runtime is then $O(\text{nnz}_B * n)$ where nnz_B denotes the number of nonzeroes in B, which is very small when density is low (and sparsity is high). Any performance gained from a fast dense-dense matrix-multiply is lost to computing redundant values.

For a high percentage of nonzeros $\%\text{nnz}_B > 0.24\%$, we can see the benefit of a fast dense matrix multiplication function. Even though the unfused code requires more storage, is still calculating redundant values, and is asymptotically worse, BLAS and GSL are poised to take advantage of machine specific information and recuperate the cost of doing extra work. Moreover, the run time for dense-function mappings remain constant as shown in Figure 7.17 since the amount of work for dense systems does not change with the sparsity of B. However, as the number of nonzeros increase, TACO's runtime steadily rises, resulting in an order of magnitude slowdown.[2]

When we fix the $\%\text{nnz}_B$ to be 40% and sweep dimension (Figure 7.20), we see an average of 31.5×, 21.7× and 18.8× speedup over TACO when the multiplication of CD is mapped to dense matrix multiplication functions provided by BLAS, GSL and TBLIS respectively. The fused code generated by TACO suffers because the percentage of nonzeroes is high, and GCC cannot use usual optimization strategies to analyze deep loop nests that iterate over sparse data structures.

Finally, we test Mosaic's performance on real-world data from the SuiteSparse matrix collection [49] on the matrices listed in Table 7.3. We order all matrices from the SuiteSparse matrix collection with respect to the number of non-zeroes they contain, limiting the maximum number of non-zeroes at 50,000 due to machine memory constraints. From this ordered list, we select, at random, 4 matrices each from the 50 matrices with the least, median, and largest number of nonzeros to instantiate the sparse matrix in the SDDMM expression. Figure 7.23 shows that as the dimension of the sparse matrices increases, the benefits of using a faster dense-dense matrix multiply eclipse the cost of doing redundant work. However, as density decreases, the cost of doing redundant work undoes the benefit of a fast multiply. While MKL and BLAS are 0.28× and 0.31× as fast as the TACO implementation for the `lp_sc50b` matrix, the performance of MKL and BLAS improves to be 1.33× and 1.69× faster than the TACO baseline for the `cavity02` matrix. The maximum speedup is observed in the case of the `IG5-12` matrix, with MKL and BLAS being 3.57× and 6.48× faster than the TACO implementation. However, MKL and BLAS are only 0.29× and 0.57× as fast as TACO for the `mimo28x28_system` matrix, which has a density of only 0.03%

7.8.4 Performance Comparisons on Real-World Expressions

We demonstrate Mosaic's ability to bind expressions to commonly used functions on a wide range of real-world expressions using the `map` scheduling command (Section 7.5). That is, after we indicated a function substitution through the `map` command, Mosaic was able to fix indices and tile computation to fit within the constraints of the function. Through Mosaic, users can rapidly compare the performance of possible factorizations side-by-side.

[2]There is no benefit in mapping dot-product functions to the dense matrix multiplication. Mosaic needs to create temporaries for vector data collection, causing slowdown, and dot-product bindings `fix` the schedule to an ikj-loop ordering, resulting in an inner-product algorithm that performs asymptotically worse as shown in Section 4.8.3 [246, 75, 244].

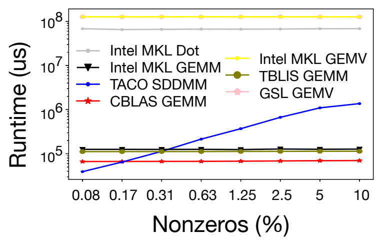


Figure 7.17: Log performance of mapped SDDMM using Mosaic as B's number of nonzeros vary, where B's dimension is $n = 2000$.

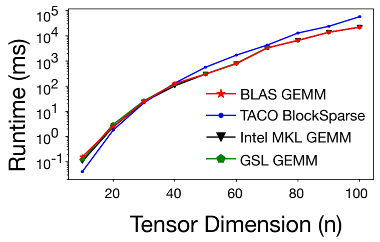


Figure 7.18: Log performance of mapped block-sparse matrix multiply using Mosaic across varying dimension, where $\%\text{nnz}_B = 5\%$.

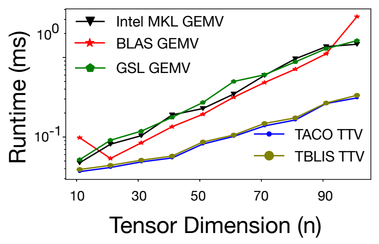


Figure 7.19: Log performance of dense tensor-times-vector (TTV) across varying tensor dimensions.

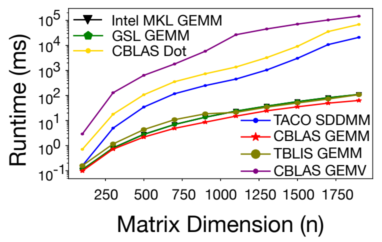


Figure 7.20: Log performance of external functions mapped to SDDMM using Mosaic as the dimension of B is varied when $\%\text{nnz}_B = 40\%$

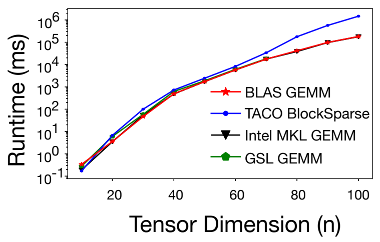


Figure 7.21: Log performance of mapped block-sparse matrix multiply using Mosaic across varying dimension, where $\%\text{nnz}_B = 20\%$

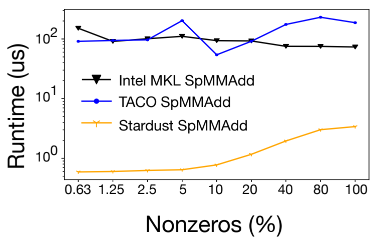


Figure 7.22: Log performance of sparse matrix-matrix addition (SpMMAdd) across varying number of nonzeros, where $n = 200$.

Table 7.3: Matrices from the SuiteSparse matrix collection [49].

Name	Domain	Dimensions	Nonzeros	Density (%)
Trec5	Combinatorial Problem	3×7	12	57.14
Ragusa18	Directed Weighted Graph	23×23	64	12.09
lpi_bgprtr	Linear Programming Problem	20×40	70	8.75
lp_sc50b	Linear Programming Problem	50×78	148	3.79
cavity02	Subsequent Computational Fluid Dynamics Problem	317×317	5,923	5.89
cavity03	Subsequent Computational Fluid Dynamics Problem	317×317	7,311	7.28
lp_nug08	Linear Programming Problem	$912 \times 1,632$	7,296	0.49
m3plates	Acoustics Problem	$11,107 \times 11,107$	6,639	0.01
IG5-12	Combinatorial Problem	$2,296 \times 2,875$	46,260	0.70
g7jac020sc	Economic Problem	$5,850 \times 5,850$	42,568	0.12
gemat1	Power Network Problem	$4,929 \times 10,595$	46,591	0.09
mimo28x28_system	Eigenvalue/Model Reduction Problem	$13,251 \times 13,251$	48,737	0.03

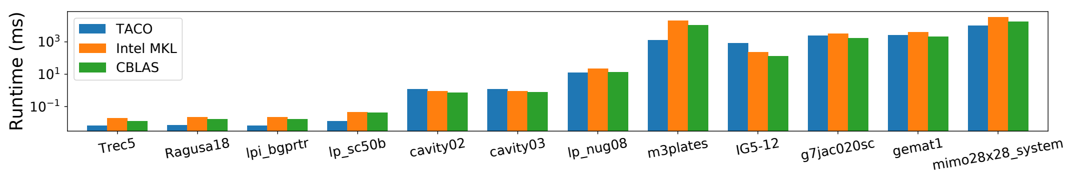


Figure 7.23: Log performance of external functions mapped to SDDMM using Mosaic for SuiteSparse matrices.

We evaluate performance on block-sparse matrix-matrix multiplication (Block-Sparse SpMV), tensor-times-vector multiplication (TTV), and sparse matrix-matrix addition (SpMMAdd). Block-sparse matrix-matrix multiplication demonstrates higher-order sparse computation that can be mapped to dense functions (see Figures 7.18 and 7.21). TTV demonstrates Mosaic's ability to handle higher-order expressions (see Figure 7.19). Finally, SpMMAdd demonstrates an expression with additions and two sparse operands (see Figure 7.22).

Block-sparse matrices contain dense blocks of values that are spread uniformly throughout the matrix. These blocks effectively act as dense matrices inside a larger sparse matrix. So, while computing the product of a block-sparse matrix with a dense matrix, we can use the blocks of values as inputs to a dense-dense matrix multiply function. Figure 7.18 and Figure 7.21 show the results of this mapping with 5% and 20% nonzeroes respectively. For 5% of nonzeroes and $n < 40$, TACO outperforms other mappings. But, as the dimension increases, BLAS delivers a 2.4× speedup. To call a dense-dense matrix multiply function, Mosaic packs the blocked values into an input array that corresponds to the same format the external function is expecting data in. When dimension $n < 40$, an optimized matrix multiply cannot recover the cost of this packing and unpacking. However, as n increases, this cost becomes negligible and we reap the benefits of a hand-optimized matrix multiply. For the 20% sparsity case, we see a similar trend. Since the number of nonzeroes is already higher at lower dimensions, we do not see much difference between the TACO and BLAS implementations. However, we see a 7.4× speedup over TACO as the dimension increases.

Next, we evaluate Mosaic's performance on the TTV expression when it is mapped to GEMV and TTV functions. From Figure 7.19, we notice that neither the GEMV nor the TTV functions can compete with TACO. The optimizations discovered by GCC on the TACO code trump optimizations performed by TBLIS. We found that when running the TTV benchmark on GCC 7.5.0, a lower version than what is shown, TBLIS outperforms TACO. While underlying systems and libraries evolve, handwritten code does not. Therefore, having a system like Mosaic that can incorporate new systems and automatically generate code that targets a new selection of functions helps users rapidly adapt old code to new developments (like new GCC versions).

Finally, we look at the performance of Mosaic when computing SpMMAdd (see Figure 7.22). Similar to results found in Chapter 6, Stardust boosts performance by up to 173× over TACO.

7.8.5 External Function Abstraction Study

We perform a lines of code (LOC) study on the external function abstractions in Mosaic. The LOC numbers from Table 7.4 demonstrate that the development of external function abstractions for Mosaic is relatively straightforward since each expression requires 20 lines of code on average. Furthermore, each of these external functions only needs to be written once and is usable by all Mosaic users. Table 7.4 also shows a subset of the total number of external functions (38) we were able to plug in to Mosaic in a limited amount of time.

Table 7.4: A subset of the external functions plugged into Mosaic for experiments described in Section 7.8. Functions that compute two or more expressions are underlined after their first instance (denoting duplicates).

Name	Expression	Lines of Code (LOC)						
		BLAS	GSL	TBLIS	AVX	Stardust	cuSPARSE	MKL
`VecAdd`	$A_i = B_i + C_i$			16	16			
`Saxpy`	$A_i = B_i + \gamma D_i$	10	15					
`Dot`	$\alpha = B_i * C_i$	12	16	17	18			
`GEMV`	$A_i = B_{ij} * C_j$	11	20	19				15
`SGEMV`	$A_i = \alpha * B_{ij} * C_j + \beta * D_i$	$\underline{11}$	$\underline{20}$			30		
`SpMV`	$A_i = \alpha * B_{ij} * C_j + \beta * D_i$			30			43	21
`SpMMAdd`	$A_{ik} = B_{ij} + C_{ij}$					30		18
`GEMM`	$A_{ik} = B_{ij} * C_{jk}$	12	19	$\underline{19}$		30		17
`SGEMM`	$A_{ik} = \alpha * B_{ij} * C_{jk} + \beta * C_{ik}$	$\underline{12}$	$\underline{19}$					$\underline{17}$
`TTM`	$A_{ijk} = B_{ijl} * C_{kl}$			$\underline{18}$		30		
`Plus3`	$A_{ijk} = B_{ijk} + C_{ijk}$			18		30		

Table 7.5: End-to-end times and number of distinct matching found by Mosaic (excluding redundant mappings generated by consecutively adding/multiplying by an identity operand).

Name	Expression	# Registered Functions = 3		# Registered Functions = 9	
		End-to-End Runtime (s)	# Mappings	End-to-End Runtime (s)	# Mappings
`VecAdd`	$A_i = B_i + C_i$	0.027195	3	0.439966	7
`Dot`	$\alpha = B_i * C_i$	0.034908	3	0.102808	7
`GEMV`	$A_i = B_{ij} * C_j$	0.037356	3	0.107068	7
`SGEMM`	$A_{ik} = B_{ij} * C_{jk} + C_{ik}$	0.124247	5	0.338617	13
`Plus3`	$A_{ijk} = B_{ijk} + C_{ijk}$	0.037763	3	1.19884	7
`SDDMM`	$A_{ik} = \mathbf{B}_{ik} * C_{ij} * D_{jk}$	0.09578	2	0.351564	13

7.8.6 Evaluation of the Search

We present the end-to-end runtime and number of mappings found for the automatic mapping algorithm in Table 7.5. For this experiment, we ran our search twice with 3 and 9 external functions registered to Mosaic: (CBLAS); and `Gemm` (CBLAS, MKL, GSL), `Dot` (CBLAS, MKL, GSL) and `VecAdd` (CBLAS, MKL, AVX). Additionally, the depth of mathematical rewrites was set to 3. We notice that the runtime of the search algorithm depends on the complexity of the expression i.e. the scope of mathematical rewrites and scales linearly with the number of registered functions. We also see a significant jump in the `VecAdd` and `Plus3` expressions as the number of registered functions increases. In both expressions, the mapper found valid AVX mappings and since the AVX interface requires tiling and has a compute capability language constraint, extra time is spent generating and executing the resultant Z3 query.

System	Capability Language	Checker Linear Search	Checker Random Search
Stardust	0.421 sec	21.36 sec	1.72 sec
AVX	0.0201 sec	0.00163 sec	14.027 sec

Figure 7.24: Valid tiling times for the AVX and Stardust functions when using the compute capability language or an opaque checker function (using both a linear and random search of the checker function).

Benefits of the Compute Capability Language

To check whether the compute capability language gives us any benefits over the checker function, we timed how quickly Mosaic can discover mappings when both descriptions or only the checker function are given. We ran this experiment for the AVX and Stardust functions since they have the most interesting language constraints. AVX requires vectors to be exactly of size 4 (with floating-point data), and Stardust only allows tensors that contain less than 65,536 values. A tiling transformation is necessary to target expressions containing large tensors that do not fit these constraints. When finding a concrete tile size using only the checker function, we use two strategies. First, we use a linear search from 1 to 65,536 and call the checker function on each tile size to validate correctness. After finding a valid tiling, we maximize tile size by continuing the linear search until we find a tile size that the checker function rejects. We also use a randomized search between 1 and 65,536. When the checker function returns a match on a randomly chosen tile size, a linear search starts to find the maximum tile size.

The times presented using both the linear and random search are noted in Figure 7.24. Picking a single strategy to step through transformations is hard because of the range of functions available. While the linear increment was beneficial for small tilings used in AVX, a random increment was much more productive in finding larger tilings for Stardust. Using information about Stardust, we were able to bound our search, otherwise, picking a maximum tile size may be a challenging decision in itself. By encoding such information in the compute capability language, we are able to apply precise rewrites without having to search for concrete parameters (like tile sizes) in the dark.

7.9 Work Foundational to the Design of Mosaic

Mosaic is the first programming system for sparse tensor algebra that can generate code that mixes fully generated code with calls to external functions. It uses a scheduling language to reshape and bind expressions to functions, verifies that the bindings are correct, and also provides an automated search system that, given an expression and one or more functions, will return to the user schedules that take advantage of those functions. In this section, we discuss other programming systems for sparse tensor algebra, other scheduling languages, fully-automated scheduling systems, libraries, and domain-specific hardware that could be used by our system.

Programming Systems for Sparse Tensor Algebra

The traditional way to compute sparse linear and tensor algebra expressions is to write a sequence of calls to libraries like Intel MKL [95] or cuSPARSE [149]. Several systems have been designed to automate this mapping, such as MATLAB's sparse support [65], the MATLAB Tensor Toolbox [112], Julia [21], and CTF [193, 194]. These systems have excellent performance for expressions for which they have a suitable function to call, but their performance can suffer for other expressions, as these have to be factorized to use a fixed set of available functions. These systems are also hard-coded to utilize specific functions, and a user cannot add new functions to this set.

More recently, several compilers have been developed that compile sparse tensor algebra expressions to imperative code. These include the TACO compiler [108], the MLIR SparseTensor Dialect [23], and the Sparse Polyhedral Framework [253]. These systems can compile sparse and dense tensor algebra expressions all the way down to imperative code, but cannot mix the generated code with calls to external functions. Thus, their performance suffers where a library can compute a specific expression—such as dense GEMM—faster than their generated code. Some compilers, like SparseTIR [242], can leverage hand-optimized code for CPUs or domain-specific architecture like the NVIDIA Tensor Cores [38]. However, unlike Mosaic, SparseTIR cannot generate efficient fused code with coiteration and cannot be extended to utilize other external functions.

Scheduling Languages, Binding, and Automatic Scheduling

Starting with the Halide compiler [170], many domain-specific programming systems have adopted scheduling languages, including TVM [36], TACO [106, 181], Lift/Elevate [76]. The CHiLL [31] and POET [243] compilers also enable the separate scheduling of C loops. The scheduling language of the Exo compiler [93] includes commands to substitute code for specialized user-defined instruction. Out of these systems, the Exo compiler is the closest to our approach, however, its affine loop-nest IR is not suitable for sparse computations. Exo is also tailored for working with low-level instructions. Mosaic, in contrast, can compile sparse tensor algebra and can replace whole sub-computations with calls to external functions.

In addition to manual approaches to scheduling, researchers have explored automatic scheduling systems [36, 146, 171, 254, 8, 3]. Out of these systems, the AMOS [254] compiler is closest to our approach. The Amos compiler is an automatic compilation framework for spatial hardware targeting dense tensor algebra. Although their compute abstraction is similar to Concrete Index Notation, they target domain-specific accelerators. Mosaic on the other hand, lives within the C ecosystem and can target both CPUs and specialized hardware, albeit the external function must handle hierarchical memory accesses itself.

Libraries and Domain-Specific Hardware

There are a large number of libraries that bundle hand-written for sparse and dense linear and tensor algebra computations. Examples for dense tensor algebra include the BLAS library [126], Intel MKL [95], the GNU Scientific Library [69], and NVIDIA CUTLASS [154]. More recently, researchers have developed hand-optimized libraries for dense tensor algebra, most prominently the TBLIS library [138]. Sparse linear and tensor algebra libraries are also common, including Intel MKL, and NVIDIA cuSPARSE [149]. Moreover, in the last eight years, many domain-specific architectures have been developed that accelerate specific expressions and, in some cases, a class of expressions. These domain-specific architectures include both fixed-function accelerators [246, 157, 169, 201, 80] and more general reconfigurable accelerators, like Onyx presented in Chapter 5 and others [45, 81, 176, 202]. The Mosaic compiler is designed to take advantage of these libraries and domain-specific architectures by generating code that composes them and that fills sub-expressions that cannot be mapped to available external functions or hardware.

7.10 Concluding Thoughts on Mosaic

Mosaic is a system that can compose externally defined library functions to implement an arbitrary sparse tensor algebra expression, unlocking good performance where hand-optimized implementations or specialized hardware exist. It fills in the gaps that are not provided by the libraries, guaranteeing generality in both expressions and data structures, as well as fusion. As opposed to writing code that is hard-wired to utilize a small set of fixed libraries, users can choose Mosaic, allowing them to access these libraries under the umbrella of a single host compiler. Because of the completion of partial schedules, users can use external functions with ease without having to sift through documentation for hours. Because Mosaic also validates bindings and automatically generates code, users can have more trust in optimizations and their supporting code. We hope this empowers users to utilize upcoming, state-of-the-art libraries with a few lines of code and minimal refactoring. We also hope that performance engineers contribute to Mosaic's growing library of external functions so that every Mosaic user may benefit.

Chapter 8

Conclusions

“Life can only be understood backwards; but it must be lived forwards.”

Søren Kierkegaard

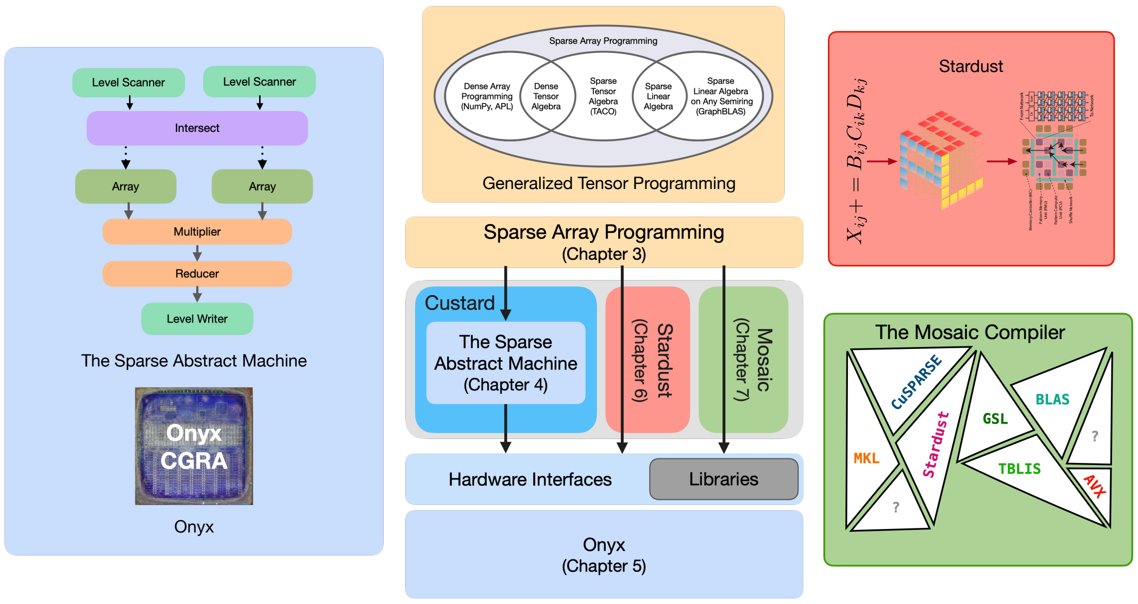


Figure 8.1: Summary of the ideas and systems presented within this thesis.

My dissertation, at its core, is about the search for the right abstraction for sparse hardware. One that aligns with the structure of the workload and the architecture beneath it, making programming feel less like wrestling with complexity and more like uncovering clarity. Whether that abstraction is dataflow-based (Chapters 4 and 5), loop-based (Chapters 3 and 6), or kernel-based (Chapter 7), it enables elegant programming of complex sparse workloads on complex dataflow accelerators

(Figure 8.1). I demonstrate how these abstractions enable simpler end-to-end compiler solutions, better computational performance, a more general notion of sparse dataflow to create portable code across accelerators, a first-of-its-kind fabricated hardware architecture for sparse tensor computation, heterogeneous coordination of accelerators, and hardware that is more accessible to performance engineers and end users. This thesis provides a first step towards prolific domain-specific accelerators by way of more mature programming systems in the domain of sparse operations. The ideas, techniques, and systems presented have led to many unsolved problems and open questions, which I describe below.

Sparse Benchmarking

Due to the complexity of sparse workloads, it is often impossible to determine a fair comparison between systems. Because, again, performance depends on: 1) the sparse data and 2) decisions made by the system.

First, researchers often benchmark on differing sparse data. Although strides have been made in sparse dataset collection [221, 190, 49], they often do not include the context from which the data came. For instance, data in these datasets are not tied to their original expression or to the other sparse operands they should be computed with. The datasets are also so large that authors choose to test their systems on a subset of the data. These limitations lead researchers to make seemingly ad-hoc benchmarking decisions with unfounded justifications.

Second, it is often difficult to find baseline comparison points from prior systems that are fair. Certain systems may not support all combinations of a given sparse kernel. For example, many systems only support expressions with one sparse operand [97, 240, 188], others may only support specific data structure formats [98, 173, 160], and even more may only support a constrained number of schedules [119, 106]. Even if a system claims support in a publication, their support may be limited in scope when using their programming system implementation. To make matters worse, once baseline systems are chosen, other users do not know how to optimize the input parameters of the system to produce the most optimal runtime or fair configurations of code. Given these constraints, we need a more rigorous and systematic way of building better system baselines.

Standardizing sparse kernel benchmarking to compare systems in a fair way that is more rooted in reality is an open problem.[1] While the work in this dissertation also encounters these practical benchmarking constraints, its primary focus is on developing end-to-end programming systems for sparse accelerators. Therefore, addressing these broader benchmarking challenges is left as an important direction for future work.

[1]Benchmarking data and fair baseline comparisons was also historically an issue in the database community given similar complexity challenges. I believe the sparse tensor community should learn from practices within the database community.

Generalizing Sparse Tiling

One of the biggest challenges to using dataflow accelerators is their fixed memory hierarchy. Many of the programming systems and contributions in the thesis tackle this challenge by compiling to accelerator memories (Chapters 5 to 7), but they still make many fundamental assumptions. Chapter 6 assumes that all tensors fit fully on the accelerator, Chapter 5 assumes that tensors are tiled using an oracle ahead of time to naively match the accelerator memory constraints, and the systems presented in Chapters 5 and 7 can generate sparse tiles with only a few tiling schemes (algorithms), which can lead to poor performance or resource utilization. In this dissertation, I focused on generating accelerator code given a data structure and tiling strategy that is provided by the user, leaving the dynamic generation of accelerator tiles and automatically choosing that tiled data structure as future work.

Meanwhile, other researchers are proposing sophisticated sparse tiling schemes for better memory utilization or performance [155, 97, 84, 186, 127]. However, these tiling schemes are handwritten algorithms, often hard-coded to one sparse accelerator implementation. This approach leaves the coordination and tiling code directly tied to each hardware backend, which does not allow for designers to easily try tiling schemes with different data on different machines. There has been work on software-only preprocessing techniques to improve accelerator utilization [186, 236]. However, it requires tremendous effort to integrate new hardware using those techniques.

Therefore, it is an open question how programming systems can generate general sparse tiling schemes suitable for the constraints and performance of multiple heterogeneous accelerators. To answer this question, we must create programming systems that can more generally create sparse tiles dynamically based on the data, where each tile is tailored to run on an individual accelerator implementation. From there, rather than assuming that all tiling is done ahead of time on the host device, perhaps the tiling can be partially fused/pipelined with the expression computation? Answering such questions is necessary to fully utilize machines consisting of heterogeneous sparse accelerators. I am convinced that this research will further inspire solutions for general heterogeneous accelerator programming for other domains, which I discuss later in **Scheduling Multiple Heterogeneous Accelerators**.

Sparse Performance Modeling

The key to designing promising accelerators before hardening them during fabrication is the ability to explore the space of possible designs with lighter-weight performance modeling. Since sparse operations ultimately depend on the exact tensor data, it is often unknown how an optimization decision might affect overall performance and how to estimate the performance of these operations. Chapter 4 does a decent job of providing an infinite resource model for sparse tensor algebra dataflow operations through its cycle-approximate simulator. Other performance models and simulators also attempt to address this challenge [231, 150], but are limited to a single accelerator on one kernel. Therefore, there is an open problem of how to create accurate and lightweight analytical models of sparse operations for distributed heterogeneous machines. This research needs to predict performance for unknown data by treating inputs as a statistical distribution of test data and possible future data. The ideas also need to better integrate sparse tensor algebra kernels with dense tensor computation, which include leveraging prior sparse compiler theory on format conversions for storing data in any compressed data structure [41, 40, 244]. Research in this direction also includes better modeling of more general hierarchical, distributed heterogeneous hardware systems and the mapping of computation to these systems using sparse communication patterns. Most important, any sparse performance model at this scale needs to solve the problem of modeling sparse operations beyond single sparse kernels, which has been the focus of this dissertation. Specifically, larger sparse applications require a modeled estimation of every sparse intermediate result without knowing the exact input data or sparse operations, which is an unsolved research direction.[2]

Beyond Sparse Kernels

The abstraction ideas presented in this dissertation generalize beyond sparse tensor algebra kernels to other large-scale sparse applications. I believe the first step is to generalize the unified dataflow abstraction presented in Chapter 4 to general sparse tensor programs by encoding set complements and other join primitives beyond intersection and union into the abstract machine model. From there, adding dataflow primitives and memory handling for tensor reshape operations would enable machine learning applications with sparse tensors. Building these abstractions and programming systems for larger applications, beyond individual tensor kernels, is necessary for widespread programming of (sparse) AI accelerators.[3] Finally, the challenge of creating end-to-end programming systems from high-level languages for accelerators of other domains has yet to be addressed.

Portable Code Across Accelerators

The sparse abstract machine is one of the first attempts, to my knowledge, of unifying representations across multiple accelerators. This representation and its compilation approach, similar to an ISA, allows portable abstract machine code across different sparse accelerator implementations. I showed in this thesis how to generate that low-level bitstream code for the Onyx accelerator and how it would be mapped to the other accelerators in theory. However, the ultimate goal is to complete the low-level compiler paths from the sparse abstract machine to all possible sparse dataflow accelerator implementations. I leave these compilers, which would allow us to generate portable abstract machine

[2]Akin to database cardinality estimation, where estimation accuracy decreases heavily with more join operations.

[3]Another open question arises on the prevalence of unstructured sparsity for scaling ML model architectures. I raise this open question here, but do not go in depth as it is less relevant to programming systems and accelerators.

code to other sparse accelerators automatically, as future work. The introduction of other abstract machines, and their compilers, have the potential to advance this concept of portable code to other accelerators beyond sparsity.

Beyond abstractions and portability for sparse dataflow, I am convinced that abstract machines should be created 1) for other domains of computation and 2) for evolutions of architectures across generations. The explosion of accelerator designs motivates the need for abstract machines so that we do not have to create individual programming stacks for each accelerator. There exist many accelerators for graph processing and learning [89, 77, 188, 189, 67, 151, 45], robotics [152, 182, 34, 206], and bio-informatics [42, 165, 72, 19]—just to name a few domains—that do not yet have programming systems.

Abstract machines should serve as a representation where code can span generations of accelerator designs that share common architectural capabilities. I envision abstract machines as a unifying layer for programming systems across evolving architectures, each of which must continue to support modern workloads. An abstract machine model could contain the shared functionality across N generations of accelerator architectures, where each generation has M variants, before lowering code to run on each $N \times M$ hardware implementation. In this scenario, a shared abstract machine model has the potential to greatly simplify this programming system. I look forward to seeing the future of abstract machine designs and the programming systems that they will enable.

Scheduling Multiple Heterogeneous Accelerators

My dissertation uses modern programming paradigms to bring together the generality of compilers and performance of fixed-function library kernels for tensor algebra. The work in this dissertation makes strides in how to integrate external functions from other programming systems into one base system and how to map those external functions onto more general expressions. From these techniques, we can map more general programs to a variety of heterogeneous hardware backends.

Given the rise of heterogeneous hardware designs [136, 185, 103], ideas on how to program them are becoming increasingly important. In this thesis, we provide an automatic search mechanism to enumerate all possible heterogeneous external function mappings for an expression of interest. However, that mechanism does not provide a way to order programs based on some end-user metric, like performance. We leave as future work how to build this automatic scheduling mechanism to pick the best heterogeneous accelerator configuration for a given expression. Furthermore, future work should be able to plug-in differing metrics and heuristics, beyond performance, like utilization, area, or energy, to automatically guide the ordering of heterogeneous accelerator mappings.

The programming systems in this dissertation are compilers that center on user-schedulable decisions and information made available statically at compile time. However, scheduling across multiple heterogeneous accelerators efficiently also raises the question of how to coordinate multiple heterogeneous accelerators at runtime. This problem is also referred to as program segmentation across heterogeneous backends. Many open questions and problems exist on how to send program segments (or sub-expressions) to different accelerators given performance, area, and metric information both at compile time and runtime. This research may include hardware scheduling, work and data stealing, heuristics and learned models, data structure conversion and communication costs, and architectural enhancements to inform runtime metrics.

Appendix A

System and Evaluation Artifacts

A.1 SAM Artifact

Abstract

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

A.1.1 Artifact Check-List (Meta-Information)

- **Compilation:** C++ compiler (either `gcc` or `clang`). The `gcc` 9.4.0 compiler is included with the Docker image. A fork of the TACO compiler (found here) is included as a submodule in our artifact (fork `weiya711/taco`, commit hash `cf8f007`).
- **Data set:** Suitesparse Matrix Market matrices (a script to download the dataset is included and the full dataset can be found at https://sparse.tamu.edu/), Frostt Tensor Dataset tensors (a script to download the dataset is included and the full dataset can be found at http://frostt.io/), and synthetically generated matrices/higher-order tensors (included).
- **Run-time environment:** Docker, git, Python 3, and bash need to be installed on the local machine. Docker is available for many operating systems. Proficiency in bash and git is recommended.
- **Hardware:** Any conventional x86 CPU with at least 32 GB of RAM should work.
- **Metrics:** Cycles (modeled as iteration counts in our simulator and source code), expression counts, and primitive counts

- **Output:** Terminal outputs, files, tables, and graphs (PDF figures, PNG figures, and DOT file format [58] graphs). Expected results are included in the submitted paper.
- **Experiments:** All steps are detailed in the `README.md` in https://github.com/weiya711/sam-artifact. The steps include pulling a Docker image and running/attaching a container, running scripts within the docker, running one Python 3 script locally outside of the Docker to copy results, and verifying result images/files. The experiments should have less than 5% variation since the simulator is deterministic. The 5% variation is caused by different data patterns in synthetic data generation (even with sparsity held constant due to random statistics). However, these variations do not affect the paper's conclusions.
- **How much disk space required (approximately)?:** Approximately 20GB of space should be sufficient.
- **How much time is needed to prepare the workflow (approximately)?:** About 10-15 minutes.
- **How much time is needed to complete experiments (approximately)?:** To complete all experiments it takes approximately 65 hours. We also include scripts to complete a subset of the experiments, which include Table 4.2, Table 4.3, Figure 4.12, Figure 4.13, Figure 4.14, Figure 4.15, and only 8 points in Figure 4.16, that takes about 10 hours to run on a standard machine.
- **Publicly available?:** Yes, on Github at the *sam* repository (https://github.com/weiya711/sam) for active development of source code and at the *sam-artifact* repository (https://github.com/weiya711/sam-artifact) for the artifact evaluation of this paper. The specific commits for this artifact are tagged with `asplos23-ae` in both repositories.
- **Code licenses (if publicly available)?:** MIT License
- **Workflow framework used?:** Docker
- **Archived (provide DOI)?:** Yes, the DOI is https://doi.org/10.5281/zenodo.7591742 [86].

A.1.2 Description

How to Access

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

Hardware dependencies

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

Software Dependencies

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

Data sets

Table A.1: Matrices from the SuiteSparse matrix collection [49] used to analyze the overhead of our stream representation in matrix identity (Section 4.8.4). We randomly selected each set of 5 matrices (delineated in the table above) from the smallest, median, and largest 50 SuiteSparse matrices—based on dense dimension size—that would fit in memory.

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

The evaluation requires matrices from the Suitesparse Matrix Market dataset (script to download the dataset is included, full dataset can be found at https://sparse.tamu.edu/), the Frostt Tensor Dataset (script to download the dataset is included, full dataset can be found http://frostt.io/), and synthetically generated matrices/higher-order tensors (included in the artifact evaluation). The synthetic data generation pattern for Section 4.8.3 is shown in Figure A.1.

A.1.3 Installation

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

```
$ git clone https://github.com/weiya711/sam-artifact
$ cd sam-artifact
$ git submodule update --init --recursive
$ docker build -t sam-artifact .
```

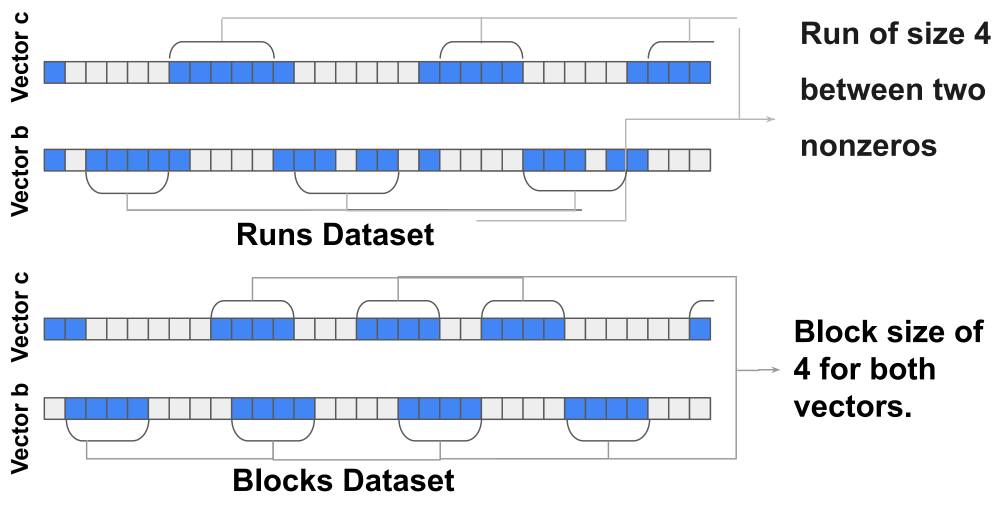

Vector c
Vector b
Run of size 4 between two nonzeros
Runs Dataset
Vector c
Vector b
Block size of 4 for both vectors.
Blocks Dataset

Figure A.1: Sample *runs* and *blocks* vector data patterns used for our synthetic data generation in Section 4.8.3.

The docker container can be started with the following command:

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

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

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

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

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

A.1.4 Experimental Workflow

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

A.1.5 Evaluation and Expected Results

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

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

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

The expected results for the above commands are:

- **Table 1**: The standard output from the fire test, which is also saved at `/sam-artifact/sam/tab1.log` in the Docker container, should match the right hand side of Table 4.2. The left-hand side (Sparse Tensor Algebra Features) contains manually derived summaries of the expressions.
- **Table 2**: The file `/sam-artifact/taco-website/tab2.log` in the Docker container should match Table 4.3.
- **Figure 11**: The file `fig11.pdf` on the local machine should match Figure 4.12.
- **Figure 12**: The file `fig12.pdf` on the local machine should match Figure 4.13.
- **Figure 13**: The files `fig13a.pdf`, `fig13b.pdf`, and `fig13c.pdf` on the local machine should match Figure 4.14a, Figure 4.14b, and Figure 4.14c respectively.
- **Figure 14**: The file `fig14.pdf` on the local machine should match Figure 4.15.

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

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

Table A.2: The datasets used to evaluate Stardust.

App		Name	Dimensions	Density
SpMV, SDDMM, Mat...Mul, Residual		bcsstk30 [49]	28924×28924	2.48×10^{-3}
		ckt11752_dc_1 [49]	49702×49702	1.35×10^{-4}
		Trefethen_20000 [49]	20000×20000	1.39×10^{-3}
Plus3		random	800×800	1.00×10^{-2}
		random	800×800	10.00×10^{-2}
		random	800×800	50.00×10^{-2}
TTV, TTM, MTTKRP		facebook [221]	$1591 \times 63891 \times 63890$	1.14×10^{-7}
	InnerProd, Plus2	random	$200 \times 200 \times 200$	1.00×10^{-2}
		random	$200 \times 200 \times 200$	10.00×10^{-2}
		random	$200 \times 200 \times 200$	50.00×10^{-2}

```
$ python sam/scripts/artifact_docker_copy.py \
    --output_dir <OUTPUT_DIRECTORY> \
    --docker_id <DOCKER_ID>
```

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

- **Figure 15**: The file `fig15.pdf` on the local machine should contain a subset of scatter points that match Figure 4.16.
- **Figure 16:** This figure is manually derived.

A.1.6 Experiment Customization

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

A.2 Evaluation Datasets for Stardust

Below is a description of the datasets used to evaluate Stardust in Section 6.8. As mentioned, we use the same SuiteSparse matrices demonstrated in the original Capstan paper for most 2D kernels to maintain a fair comparison between Stardust-generated and handwritten Capstan kernels [176]. SuiteSparse only contains matrices and cannot be used for higher-order kernels. Therefore, we use the real-world Facebook [221] tensor for most 3D kernels as in prior work [108].

The highly sparse nature ($> 99\%$) of these datasets makes them unideal for Capstan's original bitvector design as bitvectors still have to iterate over a dense iteration space divided by a constant factor (the bitvector size) [176]. Therefore, we use uniformly randomly generated data with higher densities, as in the original Capstan work, for Plus3, InnerProd, and Plus2. These expressions are prone to slowdowns on high-sparsity data since they perform higher-order computation and/or have multiple sparse operands. For Plus3, we generate the other operands by rotating the input matrix's columns right by one and two as in Section 4.8. For Plus2 and InnerProd, we generate the additional operand by rotating the even coordinates of the last tensor dimension by one.

A.3 Mosaic Artifact

The Mosaic compiler code is open-source and can be found at the mosaic repository on GitHub. Instructions for reproducibility and reusability are available on an archived version on Zenodo [17] and at the mosaic-artifact repository on GitHub. Benchmarking results depend on access to specialized hardware and vary based on external library versions and machine configuration.

Appendix B

Generalized Sparse Tensor Programming Supplemental Material

B.1 Generalized Tensor Index Notation Grammar

The full syntax of generalized tensor index notation can be found in Figure B.1.

$$
\begin{aligned}
&\langle \textit{tensor\_stmt} \rangle ::= \langle \textit{access} \rangle \text{ `}\texttt{=}\text{' } \langle \textit{expr} \rangle \\
&\langle \textit{access} \rangle ::= \langle \textit{tensor} \rangle_{\{\langle \textit{index} \rangle\}} \\
&\langle \textit{index} \rangle ::= \langle \textit{index\_var} \rangle \; [\; \langle \textit{index\_slice} \rangle \;] \\
&\langle \textit{index\_slice} \rangle ::= \text{`}\texttt{(}\text{' } \langle \textit{lo} \rangle \text{ `}\texttt{:}\text{' } \langle \textit{hi} \rangle \; [\; \text{`}\texttt{:}\text{' } \langle \textit{st} \rangle \;] \text{ `}\texttt{)}\text{'} \\
&\langle \textit{expr} \rangle ::= \langle \textit{literal} \rangle \mid \langle \textit{access} \rangle \mid \langle \textit{call\_expr} \rangle \mid \langle \textit{reduce\_expr} \rangle \\
&\qquad\quad \mid \; \langle \textit{binary\_expr} \rangle \mid \langle \textit{unary\_expr} \rangle \mid \text{`}\texttt{(}\text{' } \langle \textit{expr} \rangle \text{ `}\texttt{)}\text{'} \\
&\langle \textit{call\_expr} \rangle ::= \langle \textit{func} \rangle \text{ `}\texttt{(}\text{' } \langle \textit{expr} \rangle \; \{\text{`}\texttt{,}\text{' } \langle \textit{expr} \rangle\} \text{ `}\texttt{)}\text{'} \\
&\langle \textit{reduce\_expr} \rangle ::= \underset{\langle \textit{index\_var} \rangle}{\langle \textit{func} \rangle} \; \langle \textit{expr} \rangle \\
&\langle \textit{binary\_expr} \rangle ::= \langle \textit{expr} \rangle \; \langle \textit{op} \rangle \; \langle \textit{expr} \rangle \\
&\langle \textit{unary\_expr} \rangle ::= \langle \textit{op} \rangle \; \langle \textit{expr} \rangle
\end{aligned}
$$

Figure B.1: The syntax of generalized tensor index notation. Expressions within braces may be optional or repeated any number of times. $\langle \textit{func} \rangle$ and $\langle \textit{op} \rangle$ both represent arbitrary (user-defined or predefined) functions and are implemented in the same way; they differ only in how they are invoked.

B.2 PyData/Sparse API

An example of performing the `xor` operation on two sparse tensors using PyData/Sparse is found below.

```
import numpy
import sparse

# Create some tensors.
dim = 1000
A = sparse.random((dim, dim, dim))
B = sparse.random((dim, dim, dim))
# Perform the XOR computation.
C = numpy.logical_xor(A, B)
```

An example performing the GCD operation can be found below:

```
import numpy
import sparse

def gcd(x, y):
  return ... # Compute the GCD between x and y.
# Register the gcd function as a ufunc.
gcd = numpy.frompyfunc(gcd, 2, 1)

# Create some tensors.
dim = 1000
A = sparse.random((dim, dim, dim))
B = sparse.random((dim, dim, dim))
# Perform the XOR computation.
C = gcd(A, B)
```

While this code is simpler than the code to use our generalized sparse tensor compiler, users do not have control over many factors, such as the formats of the tensors, and are restricted to the predefined set of NumPy functions.

B.3 Medical Imaging Edge Detection

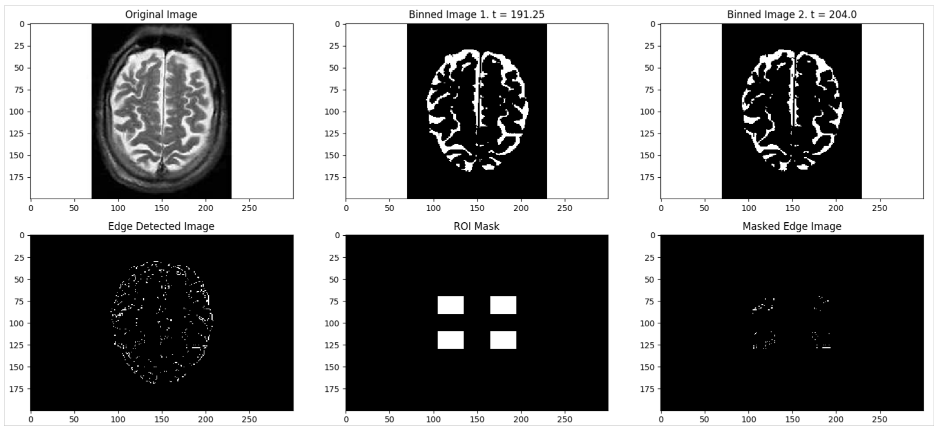


Figure B.2: Example MRI image, thresholding, ROI mask, and output

Appendix C

Example Onyx Mapping

We provide an example mapping of a matrix multiplication SAM dataflow graph onto an abstracted 5×5 CGRA in Figure C.1. The light-gray arrow demonstrates the dataflow on the CGRA. Red, blue, and green arrows represent coordinate, reference, and value stream routes on the CGRA interconnect, respectively. Yellow tiles are MEMs and blue tiles are PEs. The diagram also highlights an example SplitFIFO teal to demonstrate how unused registers compose to form two-element FIFOs.

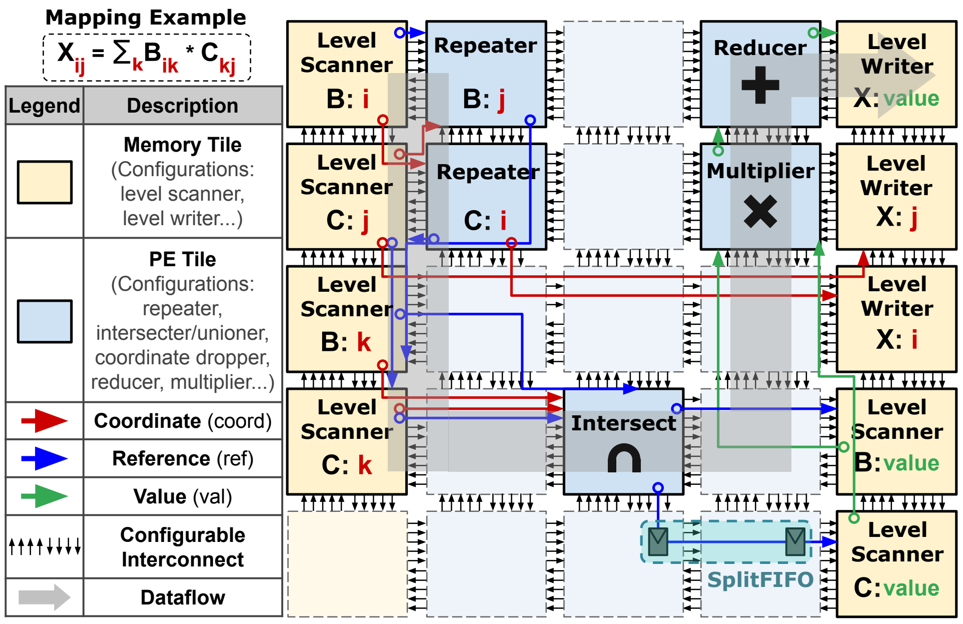


Figure C.1: An example of a matrix multiplication SAM graph placed and routed (mapped) on a small CGRA fabric.

Appendix D

Stardust's Full Memory Analysis Algorithm

We present the memory analysis algorithm as described in Section 6.5 below in Algorithm 1. The method LowerWithMemInsert is called recursively on the concrete index notation (CIN) abstract syntax tree (AST) during Stardust code generation. Algorithm 1 only shows the LowerWith-MemInsert function definition for a $\forall$ node since all other nodes behave similarly to the Lower function as in prior work [108, 106, 110, 41, 181] with some automatic inference extensions on memory types during the creation of temporary tensors and variables based on the GetMemoryType function. The algorithm shown also omits memory allocation and transfers (stores) of the result tensor since the analysis is similar as input tensor loading but occurs after the innermost CIN forall body code has been generated. We provide more details associated with the memory analysis for compressed (sparse) level format arrays since they are more complex to reason about, but a simpler analysis occurs with uncompressed (dense) level formats that only needs to emit code for a scalar array containing the dense dimension.

Algorithm 1 Memory Insertion Algorithm

```
// An iteration graph (IterationGraph) denotes which index variable (IndexVar) paths (Path) are taken for the CIN expression [108]
procedure FORALL::LOWERWITHMEMINSERT(CinNode N, Tensors T, IterationGraph G)    ▷ Forall Node
    Var indexvar = N.getIndexVar()
    // Iterators (Iterator) determine how to iterate through the forall loop based on a combination of the tensor level formats at that index variable
    Iterator iterator = N.getIterator()
    if iterator is a dense (dimension) or single sparse (position) iteration then
        for all Tensor tensor in T do
            Path path = G.getPaths(tensor)
            // The distance of an index variable from a tensor's path denotes how many levels away from an access index variable it is
            if indexvar in path && distance(indexvar, path) == 1 && tensor.getFormat(indexvar) == Sparse then
                MemType posMem = GETMEMORYTYPE(ArrayType::pos, tensor, indexvar)    ▷ Memory type based on case conditions from Section 6.5.2
                EMIT(initialize tensor_pos to posMem)
                EMIT(tensor_pos load from tensor.getPrevMemType())
                tensor.setPrevMemType(ArrayType::pos, posMem)
            else if indexvar in path && distance(indexvar, path) == 0 && tensor.getFormat(indexvar) == Sparse then
                MemType crdMem = GETMEMORYTYPE(ArrayType::crd, tensor, indexvar)
                EMIT(initialization of tensor_crd to crdMem)
                EMIT(tensor_crd load from tensor.getPrevMemType(ArrayType::crd, crdMem))
                tensor.setPrevMemType(ArrayType::crd, crdMem)
                // We are at the innermost access index variable of a tensor if the index variable is at the last position in that tensor's path
                if path.at(-1) == indexvar then
                    MemType valMem = GETMEMORYTYPE(ArrayType::val, tensor, indexvar)
                    EMIT(initialize tensor_val to valMem)
                    EMIT(tensor_val load from tensor.getPrevMemType(ArrType::val))
                    tensor.setPrevMemType(ArrayType::val, valMem)
        EMIT(Parallel pattern to iterate node based on IndexVar indexvar and Iterator iter))    ▷ Emit parallel pattern
        CinNode forallBody = N.getChild()
        for all Tensor tensor in T do
            Path path = G.getPaths(tensor)
            // Hoist out tensors as tensor values must be read at the same level of their innermost access indexvar
            if indexvar in path && distance(indexvar, path) == 0 && path.at(-1) == indexvar then
                EMIT(tensor_hoisted = read of tensor_vals)
                forallBody = forallBody[tensor_hoisted/tensor_val]
    else if iter is sparse coiteration then
        // Generate FIFO read from appropriate memory location
        for all Tensor tensor in T do
            GENERATEFIFOSFROMMEM(tensor)
        // Generate bitvectors from FIFOs
        GENERATEITERATORBITVECTORS(N, indexvar, iter)    ▷ Function that follows rewrite rules from Section 6.7
        EMIT(Scan across bitvectors using iteration algebra and rewrite system in Section 6.7 )    ▷ Emit parallel pattern
    // Proceed normally by emitting loop-body code
    LOWERWITHMEMINSERT(forallBody)
```

Bibliography

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

[2] Hameer Abbasi. "Sparse: a more modern sparse array library". In: *Proceedings of the 17th Python in Science Conference*. 2018, pp. 27–30.

[3] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, and Jonathan Ragan-Kelley. "Learning to Optimize Halide with Tree Search and Random Programs". In: *ACM Trans. Graph.* 38.4 (July 2019). DOI: 10.1145/3306346.3322967. URL: https://doi.org/10.1145/3306346.3322967.

[4] Peter Ahrens, Fredrik Kjolstad, and Saman Amarasinghe. "Autoscheduling for Sparse Tensor Algebra with an Asymptotic Cost Model". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 269–285. DOI: 10.1145/3519939.3523442. URL: https://doi.org/10.1145/3519939.3523442.

[5] Willow Ahrens, Teodoro Fields Collin, Radha Patel, Kyle Deeds, Changwan Hong, and Saman Amarasinghe. "Finch: Sparse and Structured Tensor Programming with Control Flow". In: *Proc. ACM Program. Lang.* 9.OOPSLA1 (Apr. 2025). DOI: 10.1145/3720473. URL: https://doi.org/10.1145/3720473.

[6] Willow Ahrens, Daniel Donenfeld, Fredrik Kjolstad, and Saman Amarasinghe. "Looplets: A Language for Structured Coiteration". In: *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization*. CGO '23. Montréal, QC, Canada: Association for Computing Machinery, 2023, pp. 41–54. DOI: 10.1145/3579990.3580020. URL: https://doi.org/10.1145/3579990.3580020.

[7] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. “Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 2016, pp. 1–13. DOI: `10.1109/ISCA.2016.11`.

[8] Luke Anderson, Andrew Adams, Karima Ma, Tzu-Mao Li, Tian Jin, and Jonathan Ragan-Kelley. “Efficient Automatic Scheduling of Imaging and Vision Pipelines for the GPU”. In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: `10.1145/3485486`. URL: `https://doi.org/10.1145/3485486`.

[9] *ARM Cortex-M3*. `https://developer.arm.com/Processors/Cortex-M3`.

[10] Bahar Asgari, Ramyad Hadidi, Tushar Krishna, Hyesoon Kim, and Sudhakar Yalamanchili. “ALRESCHA: A Lightweight Reconfigurable Sparse-Computation Accelerator”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 2020, pp. 249–260. DOI: `10.1109/HPCA47549.2020.00029`.

[11] John W. Backus, R. J. Beeber, Sheldon Best, Richard Goldberg, Lois M. Haibt, Harlan L. Herrick, Robert A. Nelson, David Sayre, Peter B. Sheridan, Harold Stern, Irving Ziller, Robert A. Hughes, and Roy Nutt. “The FORTRAN automatic coding system”. In: *Western Joint Computer Conference*. Los Angeles, California, Feb. 1957, pp. 188–198. DOI: `10.1145/1455567.1455599`.

[12] Brett W Bader and Tamara G Kolda. “Efficient MATLAB computations with sparse and factored tensors”. In: *SIAM Journal on Scientific Computing* 30.1 (2008), pp. 205–231.

[13] Brett W. Bader and Tamara G. Kolda. “Algorithm 862: MATLAB Tensor Classes for Fast Algorithm Prototyping”. In: *ACM Trans. Math. Softw.* 32.4 (Dec. 2006), pp. 635–653. DOI: `10.1145/1186785.1186794`. URL: `https://doi.org/10.1145/1186785.1186794`.

[14] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. “Creating an Agile Hardware Design Flow”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, pp. 1–6. DOI: `10.1109/DAC18072.2020.9218553`.

[15] Ubaid Bakhtiar, Amirmahdi Namjoo, and Bahar Asgari. “Chasoň: Supporting Cross HBM Channel Data Migration to Enable Efficient Sparse Algebraic Acceleration”. In: *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. MICRO ’25. Association for Computing Machinery, 2025, pp. 778–794. DOI: `10.1145/3725843.3756086`. URL: `https://doi.org/10.1145/3725843.3756086`.

[16] Satish Balay, William D. Gropp, Lois Curfman McInnes, and Barry F. Smith. "Efficient Management of Parallelism in Object-Oriented Numerical Software Libraries". In: *Modern Software Tools for Scientific Computing.* Ed. by Erlend Arge, Are Magnus Bruaset, and Hans Petter Langtangen. Boston, MA: Birkhäuser Boston, 1997, pp. 163–202. DOI: `10.1007/978-1-4612-1986-6_8`. URL: `https://doi.org/10.1007/978-1-4612-1986-6_8`.

[17] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. *Artifact for Mosaic: An Interoperable Compiler for Tensor Algebra.* Version 0.1.1. Mar. 2023. DOI: `10.5281/zenodo.7814275`. URL: `https://doi.org/10.5281/zenodo.7814275`.

[18] Manya Bansal, Olivia Hsu, Kunle Olukotun, and Fredrik Kjolstad. "Mosaic: An Interoperable Compiler for Tensor Algebra". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: `10.1145/3591236`. URL: `https://doi.org/10.1145/3591236`.

[19] Sairam Behera, Severine Catreux, Massimiliano Rossi, Sean Truong, Zhuoyi Huang, Michael Ruehle, Arun Visvanath, Gavin Parnaby, Cooper Roddey, Vitor Onuchic, Andrea Finocchio, Daniel L. Cameron, Adam English, Shyamal Mehtalia, James Han, Rami Mehio, and Fritz J. Sedlazeck. "Comprehensive genome analysis and variant detection at scale using DRAGEN". In: *Nature Biotechnology* 43.7 (July 2025), pp. 1177–1191. DOI: `10.1038/s41587-024-02382-1`. URL: `https://doi.org/10.1038/s41587-024-02382-1`.

[20] Nathan Bell and Michael Garland. "Implementing sparse matrix-vector multiplication on throughput-oriented processors". In: *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis.* SC '09. Portland, Oregon: Association for Computing Machinery, 2009. DOI: `10.1145/1654059.1654078`. URL: `https://doi.org/10.1145/1654059.1654078`.

[21] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. DOI: `10.1137/141000671`. eprint: `https://doi.org/10.1137/141000671`. URL: `https://doi.org/10.1137/141000671`.

[22] Vivek Bharadwaj, Aydın Buluç, and James Demmel. *Distributed-Memory Sparse Kernels for Machine Learning.* 2022. DOI: `10.48550/ARXIV.2203.07673`. URL: `https://arxiv.org/abs/2203.07673`.

[23] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. "Compiler Support for Sparse Tensor Computations in MLIR". In: *ACM Trans. Archit. Code Optim.* 19.4 (Sept. 2022). DOI: `10.1145/3544559`. URL: `https://doi.org/10.1145/3544559`.

[24] Aart J. C. Bik and Harry A. G. Wijshoff. "Compilation Techniques for Sparse Matrix Computations". In: *International Conference on Supercomputing.* ACM. July 1993, pp. 416–424. DOI: `10.1145/165939.166023`.

[25] Preston Briggs and Linda Torczon. “An efficient representation for sparse sets”. In: *ACM Letters on Programming Languages and Systems (LOPLAS)* 2.1-4 (1993), pp. 59–69.

[26] Aydın Buluç and John R. Gilbert. “On the representation and multiplication of hypersparse matrices”. In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008, pp. 1–11. DOI: `10.1109/IPDPS.2008.4536313`.

[27] A. Canning, G. Galli, F. Mauri, A. De Vita, and R. Car. “O(N) tight-binding molecular dynamics on massively parallel computers: an orbital decomposition approach”. In: *Computer Physics Communications* 94.2 (Apr. 1996), pp. 89–102. DOI: `10.1016/0010-4655(96)00009-4`.

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

[29] Navoneel Chakrabarty. *Brain MRI Images for Brain Tumor Detection*. 2019. URL: `https://www.kaggle.com/navoneel/brain-mri-images-for-brain-tumor-detection` (visited on 03/09/2021).

[30] Bradford L. Chamberlain, Sung-Eun Choi, E. Christopher Lewis, Calvin Lin, Lawrence Snyder, and W. Derrick Weathersby. “ZPL: A Machine Independent Programming Language for Parallel Computers”. In: *IEEE Trans. Softw. Eng.* 26.3 (Mar. 2000), pp. 197–211. DOI: `10.1109/32.842947`. URL: `https://doi.org/10.1109/32.842947`.

[31] Chun Chen, Jacqueline Chame, and Mary W. Hall. *CHiLL : A Framework for Composing High-Level Loop Transformations*. Tech. rep. 2007.

[32] Po-Han Chen, Bo-Wun Cheng, Michael Oduoza, Zhouhua Xie, Kalhan Koul, Sai Gautham Ravipati, Yuchen Mei, Rupert Lu, Alex Carsello, Mark Horowitz, and Priyanka Raina. “Opal: A 16nm Coarse-Grained Reconfigurable Array for Full Sparse ML Applications.” In: *CICC*. IEEE, 2025, pp. 1–3. URL: `http://dblp.uni-trier.de/db/conf/cicc/cicc2025.html#ChenCOXKRMLCHR25`.

[33] Hongzheng Chen, Bin Fan, Alexander Collins, Bastian Hagedorn, Evghenii Gaburov, Masahiro Masuda, Matthew Brookhart, Chris Sullivan, Jason Knight, Zhiru Zhang, and Vinod Grover. *Tawa: Automatic Warp Specialization for Modern GPUs with Asynchronous References*. 2025. arXiv: `2510.14719 [cs.LG]`. URL: `https://arxiv.org/abs/2510.14719`.

[34] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. "Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks". In: *IEEE Journal of Solid-State Circuits* 52.1 (2017), pp. 127–138. DOI: 10.1109/JSSC.2016.2616357.

[35] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. "Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices". In: *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9.2 (2019), pp. 292–308. DOI: 10.1109/JETCAS.2019.2910232.

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

[37] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. "CGRA-ME: A unified framework for CGRA modelling and exploration". In: *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP).* 2017, pp. 184–189. DOI: 10.1109/ASAP.2017.7995277.

[38] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. "NVIDIA A100 Tensor Core GPU: Performance and Innovation". In: *IEEE Micro* 41.2 (2021), pp. 29–35. DOI: 10.1109/MM.2021.3061394.

[39] Stephen Chou and Saman Amarasinghe. "Compilation of dynamic sparse tensor algebra". In: *Proc. ACM Program. Lang.* 6.OOPSLA2 (Oct. 2022). DOI: 10.1145/3563338. URL: https://doi.org/10.1145/3563338.

[40] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. "Automatic Generation of Efficient Sparse Tensor Format Conversion Routines". In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* PLDI 2020. London, UK: Association for Computing Machinery, 2020, pp. 823–838. DOI: 10.1145/3385412.3385963. URL: https://doi.org/10.1145/3385412.3385963.

[41] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. "Format Abstraction for Sparse Tensor Algebra Compilers". In: *Proc. ACM Program. Lang.* 2.OOPSLA (Oct. 2018), 123:1–123:30.

[42] Nafsika Chrysanthou, Grigorios Chrysos, Euripides Sotiriades, and Ioannis Papaefstathiou. "Parallel accelerators for GlimmerHMM bioinformatics algorithm". In: *2011 Design, Automation & Test in Europe.* 2011, pp. 1–6. DOI: 10.1109/DATE.2011.5763024.

[43] *CUDA C Programming Guide, Version 13.0.* https://docs.nvidia.com/cuda/cuda-c-programming-guide/. NVIDIA Corporation. 2025.

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

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

[46] Steven Dalton, Nathan Bell, Luke Olson, and Michael Garland. *Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations*. Version 0.5.0. 2014. URL: http://cusplibrary.github.io/.

[47] Pratyush Das, Amirhossein Basareh, Adhitha Dias, Artem Pelenitsyn, Kirshanthan Sundararajah, and Milind Kulkarni. *SABLE: Staging Blocked Evaluation of Sparse Matrix Computations*. 2025. arXiv: 2407.00829 [cs.DC]. URL: https://arxiv.org/abs/2407.00829.

[48] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. "Hardware Acceleration of Sparse and Irregular Tensor Computations of ML Models: A Survey and Insights". In: *Proceedings of the IEEE* 109.10 (2021), pp. 1706–1752. DOI: 10.1109/JPROC.2021.3098483.

[49] Timothy A Davis and Yifan Hu. "The University of Florida sparse matrix collection". In: *ACM Transactions on Mathematical Software (TOMS)* 38.1 (2011), pp. 1–25.

[50] Timothy A. Davis. "Algorithm 1000: SuiteSparse:GraphBLAS: Graph Algorithms in the Language of Sparse Linear Algebra". In: *ACM Trans. Math. Softw.* 45.4 (Dec. 2019). DOI: 10.1145/3322125. URL: https://doi.org/10.1145/3322125.

[51] Timothy A. Davis. *Direct Methods for Sparse Linear Systems (Fundamentals of Algorithms 2)*. USA: Society for Industrial and Applied Mathematics, 2006.

[52] Kyle Deeds, Willow Ahrens, Magdalena Balazinska, and Dan Suciu. "Galley: Modern Query Optimization for Sparse Tensor Programs". In: *Proc. ACM Manag. Data* 3.3 (June 2025). DOI: 10.1145/3725301. URL: https://doi.org/10.1145/3725301.

[53] Adhitha Dias, Logan Anderson, Kirshanthan Sundararajah, Artem Pelenitsyn, and Milind Kulkarni. "SparseAuto: An Auto-scheduler for Sparse Tensor Computations using Recursive Loop Nest Restructuring". In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689730. URL: https://doi.org/10.1145/3689730.

[54] Adhitha Dias, Kirshanthan Sundararajah, Charitha Saumya, and Milind Kulkarni. "SparseLNR: accelerating sparse tensor computations using loop nest restructuring". In: *Proceedings of the 36th ACM International Conference on Supercomputing*. ICS '22. Virtual Event: Association for Computing Machinery, 2022. DOI: 10.1145/3524059.3532386. URL: https://doi.org/10.1145/3524059.3532386.

[55] Daniel Donenfeld, Stephen Chou, and Saman Amarasinghe. “Unified compilation for lossless compression and sparse computing”. In: *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*. CGO ’22. Virtual Event, Republic of Korea: IEEE Press, 2022, pp. 205–216. DOI: `10.1109/CGO53902.2022.9741282`. URL: `https://doi.org/10.1109/CGO53902.2022.9741282`.

[56] James Dong and Fredrik Kjolstad. *A Compiler for Operations on Relations with Bag Semantics*. 2025. arXiv: `2502.06988 [cs.PL]`. URL: `https://arxiv.org/abs/2502.06988`.

[57] Albert Einstein. “The foundation of the general theory of relativity.” In: *Annalen Phys.* 49.7 (1916). Ed. by Jong-Ping Hsu and D. Fine, pp. 769–822. DOI: `10.1002/andp.19163540702`.

[58] John Ellson, Emden Gansner, Lefteris Koutsofios, Stephen C. North, and Gordon Woodhull. “Graphviz— Open Source Graph Drawing Tools”. In: *Graph Drawing*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Berlin, Heidelberg: Springer Berlin Heidelberg, 2002, pp. 483–484.

[59] Kathleen Feng, Taeyoung Kong, Kalhan Koul, Jackson Melchert, Alex Carsello, Qiaoyi Liu, Gedeon Nyengele, Maxwell Strange, Keyi Zhang, Ankita Nayak, Jeff Setter, James Thomas, Kavya Sreedhar, Po-Han Chen, Nikhil Bhagdikar, Zach A. Myers, Brandon D’Agostino, Pranil Joshi, Stephen Richardson, Christopher Torng, Mark Horowitz, and Priyanka Raina. “Amber: A 16-nm System-on-Chip With a Coarse-Grained Reconfigurable Array for Flexible Acceleration of Dense Linear Algebra”. In: *IEEE Journal of Solid-State Circuits* (2023), pp. 1–13. DOI: `10.1109/JSSC.2023.3313116`.

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

[61] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. “Sparse GPU Kernels for Deep Learning”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press, 2020. Chap. 17, pp. 1–14.

[62] Robert A. van de Geijn and Jerrell Watts. *SUMMA: Scalable Universal Matrix Multiplication Algorithm*. Tech. rep. USA, 1995.

[63] Hasan Nazim Genc, Hansung Kim, Prashanth Ganesh, and Yakun Sophia Shao. “ Stellar: An Automated Design Framework for Dense and Sparse Spatial Accelerators ”. In: *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. Los Alamitos, CA, USA: IEEE Computer Society, Nov. 2024, pp. 409–422. DOI: `10.1109/MICRO61859.2024.00038`. URL: `https://doi.ieeecomputersociety.org/10.1109/MICRO61859.2024.00038`.

[64] Gerasimos Gerogiannis, Serif Yesil, Damitha Lenadora, Dingyuan Cao, Charith Mendis, and Josep Torrellas. “SPADE: A Flexible and Scalable Accelerator for SpMM and SDDMM”. In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA

'23. Orlando, FL, USA: Association for Computing Machinery, 2023. DOI: `10.1145/3579371.3589054`. URL: `https://doi.org/10.1145/3579371.3589054`.

[65] John R Gilbert, Cleve Moler, and Robert Schreiber. "Sparse matrices in MATLAB: Design and implementation". In: *SIAM journal on matrix analysis and applications* 13.1 (1992), pp. 333–356.

[66] Graham Gobieski, Souradip Ghosh, Marijn Heule, Todd Mowry, Tony Nowatzki, Nathan Beckmann, and Brandon Lucia. "RipTide: A Programmable, Energy-Minimal Dataflow Compiler and Architecture". In: *Proceedings of the 55th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '22. Chicago, Illinois, USA: IEEE Press, 2023, pp. 546–564. DOI: `10.1109/MICRO56248.2022.00046`. URL: `https://doi.org/10.1109/MICRO56248.2022.00046`.

[67] Courtney Golden, Axel Feldmann, Joel Emer, and Daniel Sanchez. "Quartz: A Reconfigurable, Distributed-Memory Accelerator for Sparse Applications". In: *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. MICRO '25. Association for Computing Machinery, 2025, pp. 929–943. DOI: `10.1145/3725843.3756035`. URL: `https://doi.org/10.1145/3725843.3756035`.

[68] 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-52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 151–165. DOI: `10.1145/3352460.3358291`. URL: `https://doi.org/10.1145/3352460.3358291`.

[69] Brian Gough. *GNU scientific library reference manual*. Network Theory Ltd., 2009.

[70] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing". In: *IEEE Micro* 32.5 (2012), pp. 38–51. DOI: `10.1109/MM.2012.51`.

[71] Paul Grigoras, Pavel Burovskiy, Eddie Hung, and Wayne Luk. "Accelerating SpMV on FPGAs by compressing nonzero values". In: *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE. 2015, pp. 64–67.

[72] Yufeng Gu, Arun Subramaniyan, Tim Dunn, Alireza Khadem, Kuan-Yu Chen, Somnath Paul, Md Vasimuddin, Sanchit Misra, David Blaauw, Satish Narayanasamy, and Reetuparna Das. "GenDP: A Framework of Dynamic Programming Acceleration for Genome Sequencing Analysis". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA '23. Orlando, FL, USA: Association for Computing Machinery, 2023. DOI: `10.1145/3579371.3589060`. URL: `https://doi.org/10.1145/3579371.3589060`.

[73] Gaël Guennebaud, Benoit Jacob, et al. "Eigen". In: *URl: http://eigen. tuxfamily. org* 3 (2010).

[74] Ahan Gupta, Yueming Yuan, Devansh Jain, Yuhao Ge, David Aponte, Yanqi Zhou, and Charith Mendis. “SPLAT: A Framework for Optimised GPU Code-Generation for SParse reguLar ATtention”. In: *Proc. ACM Program. Lang.* 9.OOPSLA1 (Apr. 2025). DOI: `10.1145/3720503`. URL: `https://doi.org/10.1145/3720503`.

[75] Fred G. Gustavson. “Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition”. In: *ACM Trans. Math. Softw.* 4.3 (1978).

[76] Bastian Hagedorn, Johannes Lenfers, Thomas Koehler, Sergei Gorlatch, and Michel Steuwer. *A Language for Describing Optimization Strategies.* 2020. arXiv: `2002.02268 [cs.PL]`.

[77] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. “Graphicionado: A high-performance and energy-efficient accelerator for graph analytics”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO).* 2016, pp. 1–13. DOI: `10.1109/MICRO.2016.7783759`.

[78] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. “EIE: Efficient inference engine on compressed deep neural network”. In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 243–254.

[79] Charles R Harris, K Jarrod Millman, Stéfan J Van Der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Bret, Allan Haldane, Jaime Fernández del Rio, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. “Array programming with NumPy”. In: *Nature* 585.7825 (2020), pp. 357–362.

[80] Xin He, Subhankar Pal, Aporva Amarnath, Siying Feng, Dong-Hyeon Park, Austin Rovinski, Haojie Ye, Yuhan Chen, Ronald Dreslinski, and Trevor Mudge. “Sparse-TPU: Adapting Systolic Arrays for Sparse Matrices”. In: *Proceedings of the 34th ACM International Conference on Supercomputing.* New York, NY, USA: Association for Computing Machinery, 2020. Chap. 19, pp. 1–12. URL: `https://doi.org/10.1145/3392717.3392751`.

[81] 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 ’52. Columbus, OH, USA: Association for Computing Machinery, 2019, pp. 319–333. DOI: `10.1145/3352460.3358275`. URL: `https://doi.org/10.1145/3352460.3358275`.

[82] Erik Orm Hellsten, Artur Souza, Johannes Lenfers, Rubens Lacouture, Olivia Hsu, Adel Ejjeh, Fredrik Kjolstad, Michel Steuwer, Kunle Olukotun, and Luigi Nardi. “BaCO: A Fast and Portable Bayesian Compiler Optimization Framework”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages*

and Operating Systems, Volume 4. ASPLOS '23. Vancouver, BC, Canada: Association for Computing Machinery, 2024, pp. 19–42. DOI: 10.1145/3623278.3624770. URL: https://doi.org/10.1145/3623278.3624770.

[83] Rawn Henry, Olivia Hsu, Rohan Yadav, Stephen Chou, Kunle Olukotun, Saman Amarasinghe, and Fredrik Kjolstad. "Compilation of Sparse Array Programming Models". In: *Proc. ACM Program. Lang.* 5.OOPSLA (Oct. 2021). DOI: 10.1145/3485505. URL: https://doi.org/10.1145/3485505.

[84] Changwan Hong, Aravind Sukumaran-Rajam, Israt Nisa, Kunal Singh, and P. Sadayappan. "Adaptive sparse tiling for sparse matrix multiplication". In: *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*. PPoPP '19. Washington, District of Columbia: Association for Computing Machinery, 2019, pp. 300–314. DOI: 10.1145/3293883.3295712. URL: https://doi.org/10.1145/3293883.3295712.

[85] Olivia Hsu, Alexander Rucker, Tian Zhao, Varun Desai, Kunle Olukotun, and Fredrik Kjolstad. "Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture". In: *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*. CGO '25. Las Vegas, NV, USA: Association for Computing Machinery, 2025, pp. 628–643. DOI: 10.1145/3696443.3708918. URL: https://doi.org/10.1145/3696443.3708918.

[86] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark Horowitz, and Fredrik Kjolstad. *Artifact for The Sparse Abstract Machine*. Version 0.1.0. Jan. 2023. DOI: 10.5281/zenodo.7591742. URL: https://doi.org/10.5281/zenodo.7591742.

[87] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. "The Sparse Abstract Machine". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 710–726. DOI: 10.1145/3582016.3582051. URL: https://doi.org/10.1145/3582016.3582051.

[88] Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. "Taichi: a language for high-performance computation on spatially sparse data structures". In: *ACM Transactions on Graphics (TOG)* 38.6 (2019), pp. 1–16. DOI: 10.1145/3355089.3356506. URL: https://doi.org/10.1145/3355089.3356506.

[89] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. "GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs". In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 2021, pp. 1–9. DOI: 10.1109/ICCAD51958.2021.9643582.

[90] Jianyu Huang, Devin A. Matthews, and Robert A. van de Geijn. "Strassen's Algorithm for Tensor Contraction". In: *CoRR* abs/1704.03092 (2017). arXiv: `1704.03092`. URL: `http://arxiv.org/abs/1704.03092`.

[91] Su Huang, Rafail Baimouratov, Pengdong Xiao, Anand Ananthasubramaniam, and Wieslaw L Nowinski. "A Medical Imaging and Visualization Toolkit in Java". In: *Journal of Digital Imaging* 19.1 (2006), pp. 17–29. DOI: `10.1007/s10278-005-9247-6`.

[92] Wen-Cong Huang, I-Ting Lin, Wen-Ching Chen, Liang-Yi Lin, Nian-Shyang Chang, Chun-Pin Lin, Chi-Shi Chen, and Chia-Hsiang Yang. "A 28-nm 25.1 TOPS/W Sparsity-Aware CNN-GCN Deep Learning SoC for Mobile Augmented Reality". In: *2019 Symposium on VLSI Circuits*. 2022, pp. 42–43. DOI: `10.1109/VLSITechnologyandCir46769.2022.9830261`.

[93] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. "Exocompilation for Productive Programming of Hardware Accelerators". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 703–718. DOI: `10.1145/3519939.3523446`. URL: `https://doi.org/10.1145/3519939.3523446`.

[94] *Intel Advanced Vector Extensions Programming Reference*. Santa Clara, USA: Intel Corporation, 2011. URL: `https://www.intel.com/content/dam/develop/external/us/en/documents/36945`.

[95] *Intel Math Kernel Library. Reference Manual*. Santa Clara: Intel Corporation, 2009.

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

[97] Anirudh Jain, Pulkit Gupta, and Thomas M. Conte. "RASSM: Residue-based Acceleration of Single Sparse Matrix Computation via Adaptive Tiling". In: *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. ASPLOS '25. Rotterdam, Netherlands: Association for Computing Machinery, 2025, pp. 907–923. DOI: `10.1145/3669940.3707219`. URL: `https://doi.org/10.1145/3669940.3707219`.

[98] Eric Jones, Travis Oliphant, Pearu Peterson, et al. *SciPy: Open source scientific tools for Python*. [Online; accessed 12-05-2025]. 2001. URL: `http://www.scipy.org/`.

[99] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-datacenter performance analysis of a tensor processing unit". In: *Proceedings of the 44th annual international symposium on computer architecture*. 2017, pp. 1–12.

[100] J. Kepner, P. Aaltonen, D. Bader, A. Buluç, F. Franchetti, J. Gilbert, D. Hutchison, M. Kumar, A. Lumsdaine, H. Meyerhenke, S. McMillan, C. Yang, J. D. Owens, M. Zalewski, T. Mattson, and J. Moreira. "Mathematical foundations of the GraphBLAS". In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. 2016, pp. 1–9. DOI: `10.1109/HPEC.2016.7761646`.

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

[102] Jinman Kim, David D. Feng, and Tom W. Cai. "A Web Based Medical Image Data Processing and Management System". In: *Selected Papers from the Pan-Sydney Workshop on Visualisation - Volume 2*. VIP '00. Sydney, Australia: Australian Computer Society, Inc., 2000, pp. 89–91.

[103] Seah Kim, Jerry Zhao, Krste Asanović, Borivoje Nikolić, and Yakun Sophia Shao. "AuRORA: A Full-Stack Solution for Scalable and Virtualized Accelerator Integration". In: *IEEE Micro* 44.4 (2024), pp. 97–105. DOI: `10.1109/MM.2024.3409546`.

[104] Yoongu Kim, Weikun Yang, and Onur Mutlu. "Ramulator: A Fast and Extensible DRAM Simulator". In: *IEEE Comput. Archit. Lett.* 15.1 (Jan. 2016), pp. 45–49. DOI: `10.1109/LCA.2015.2414456`. URL: `https://doi.org/10.1109/LCA.2015.2414456`.

[105] David R. Kincaid, Roger G. Grimes, David M. Young, and William I. MacGregor. "ITPACK - Adaptive Iterative Algorithms Using Symetric Sparse Storage". In: vol. SPE Reservoir Simulation Symposium. SPE Reservoir Simulation Conference. Jan. 1979, SPE-7687–MS. DOI: `10.2118/7687-MS`. eprint: `https://onepetro.org/spersc/proceedings-pdf/79RS/79RS/SPE-7687-MS/2056307/spe-7687-ms.pdf`. URL: `https://doi.org/10.2118/7687-MS`.

[106] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. "Tensor Algebra Compilation with Workspaces". In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2019, pp. 180–192. DOI: `10.1109/CGO.2019.8661185`.

[107] Fredrik Kjølstad, Stephen Chou, and Saman Amarasinghe. *Taco: The tensor algebra compiler*. Nov. 2022. URL: `http://tensor-compiler.org/`.

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

[109] Fredrik Kjolstad, Ryan Senanayake, Stephen Chou, Rawn Henry, David Lugato, Shoaib Kamil, Mark Glines, Olivia Hsu, Patricio Noyola, Willow Ahrens, Rohan Yadav, Genghan Zhang, Nirvik Baruah, Advay Pal, Yishen Chen, Sam Kaplan, Penporn Koanantakool, Gurtej Kanwar, Yisu Remy Wang, Lorenzo Chelini, Shizhi Tang, Daniel Bougeois, David Hagen, and Syoyo Fujita. *The Tensor Compiler (TACO)*. `https://github.com/tensor-compiler/taco`. 2023.

[110] Fredrik Berg Kjølstad. “Sparse tensor algebra compilation”. PhD thesis. Massachusetts Institute of Technology, 2020.

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

[112] Tamara G. Kolda and Brett W. Bader. *MATLAB Tensor Toolbox, Version 00.* Aug. 2006. URL: https://www.osti.gov/biblio/1230898.

[113] Tamara G. Kolda and Jimeng Sun. “Scalable Tensor Decompositions for Multi-aspect Data Mining”. In: *2008 Eighth IEEE International Conference on Data Mining.* 2008, pp. 363–372. DOI: 10.1109/ICDM.2008.89.

[114] Taeyoung Kong, Kalhan Koul, Priyanka Raina, Mark Horowitz, and Christopher Torng. *Hardware Abstractions and Hardware Mechanisms to Support Multi-Task Execution on Coarse-Grained Reconfigurable Arrays.* 2023. arXiv: 2301.00861 [cs.AR]. URL: https://arxiv.org/abs/2301.00861.

[115] Yehuda Koren, Robert Bell, and Chris Volinsky. “Matrix Factorization Techniques for Recommender Systems”. In: *Computer* 42.8 (2009), pp. 30–37. DOI: 10.1109/MC.2009.263.

[116] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. “A relational approach to the compilation of sparse matrix programs”. In: *Euro-Par Parallel Processing.* Passau, Germany: Springer, 1997, pp. 318–327. DOI: 10.1007/BFb0002751.

[117] Kalhan Koul, Olivia Hsu, Yuchen Mei, Sai Gautham Ravipati, Maxwell Strange, Jackson Melchert, Alex Carsello, Taeyoung Kong, Po-Han Chen, Huifeng Ke, Keyi Zhang, Qiaoyi Liu, Gedeon Nyengele, Zhouhua Xie, Akhilesh Balasingam, Jayashree Adivarahan, Ritvik Sharma, Christopher Torng, Joel S. Emer, Fredrik Kjolstad, Mark Horowitz, and Priyanka Raina. “Onyx: A 12-nm Programmable Accelerator for Dense and Sparse Applications”. In: *IEEE Journal of Solid-State Circuits* (2025), pp. 1–13. DOI: 10.1109/JSSC.2025.3604724.

[118] Kalhan Koul, Jackson Melchert, Kavya Sreedhar, Leonard Truong, Gedeon Nyengele, Keyi Zhang, Qiaoyi Liu, Jeff Setter, Po-Han Chen, Yuchen Mei, Maxwell Strange, Ross Daly, Caleb Donovick, Alex Carsello, Taeyoung Kong, Kathleen Feng, Dillon Huff, Ankita Nayak, Rajsekhar Setaluri, James Thomas, Nikhil Bhagdikar, David Durst, Zachary Myers, Nestan Tsiskaridze, Stephen Richardson, Rick Bahr, Kayvon Fatahalian, Pat Hanrahan, Clark Barrett, Mark Horowitz, Christopher Torng, Fredrik Kjolstad, and Priyanka Raina. “AHA: An Agile Approach to the Design of Coarse-Grained Reconfigurable Accelerators and Compilers”.

In: *ACM Trans. Embed. Comput. Syst.* 22.2 (Jan. 2023). DOI: 10.1145/3534933. URL: https://doi.org/10.1145/3534933.

[119] Kalhan Koul, Maxwell Strange, Jackson Melchert, Alex Carsello, Yuchen Mei, Olivia Hsu, Taeyoung Kong, Po-Han Chen, Huifeng Ke, Keyi Zhang, Qiaoyi Liu, Gedeon Nyengele, Akhilesh Balasingam, Jayashree Adivarahan, Ritvik Sharma, Zhouhua Xie, Christopher Torng, Joel Emer, Fredrik Kjolstad, Mark Horowitz, and Priyanka Raina. "Onyx: A 12nm 756 GOPS/W Coarse-Grained Reconfigurable Array for Accelerating Dense and Sparse Applications". In: *2024 IEEE Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. 2024, pp. 1–2. DOI: 10.1109/VLSITechnologyandCir46783.2024.10631383.

[120] Kalhan Koul, Zhouhua Xie, Maxwell Strange, Sai Gautham Ravipati, Bo Wun Cheng, Olivia Hsu, Po-Han Chen, Mark Horowitz, Fredrik Kjolstad, and Priyanka Raina. "Designing Programmable Accelerators for Sparse Tensor Algebra". In: *IEEE Micro* 45.3 (2025), pp. 58–65. DOI: 10.1109/MM.2025.3556611.

[121] Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. "Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs". In: *Proc. ACM Program. Lang.* 7.PLDI (June 2023). DOI: 10.1145/3591268. URL: https://doi.org/10.1145/3591268.

[122] Rubens Lacouture, Nathan Zhang, Ritvik Sharma, Marco Siracusa, Fredrik Kjolstad, Kunle Olukotun, and Olivia Hsu. *FuseFlow: A Fusion-Centric Compilation Framework for Sparse Deep Learning on Streaming Dataflow.* 2025. arXiv: 2511.04768 [cs.LG]. URL: https://arxiv.org/abs/2511.04768.

[123] Avery Laird, Bangtian Liu, Nikolaj Bjørner, and Maryam Mehri Dehnavi. "SpEQ: Translation of Sparse Codes using Equivalences". In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: 10.1145/3656445. URL: https://doi.org/10.1145/3656445.

[124] Leslie Lamport. "The Parallel Execution of DO Loops". In: *Communications of the ACM* 17.2 (1974), pp. 83–93. URL: http://research.microsoft.com/en-us/um/people/lamport/pubs/do-loops.pdf.

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

[126] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. "Basic linear algebra subprograms for Fortran usage". In: *ACM Transactions on Mathematical Software (TOMS)* 5.3 (1979), pp. 308–323.

[127] Xintong Li, Zhiyao Li, and Mingyu Gao. "HYTE: Flexible Tiling for Sparse Accelerators via Hybrid Static-Dynamic Approaches". In: *Proceedings of the 52nd Annual International Symposium on Computer Architecture.* ISCA '25. Association for Computing Machinery, 2025,

pp. 1613–1626. DOI: 10.1145/3695053.3731044. URL: https://doi.org/10.1145/3695053.3731044.

[128] Zhiyao Li, Jiaxiang Li, Taijie Chen, Dimin Niu, Hongzhong Zheng, Yuan Xie, and Mingyu Gao. "Spada: Accelerating Sparse Matrix Multiplication with Adaptive Dataflow". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 747–761. DOI: 10.1145/3575693.3575706. URL: https://doi.org/10.1145/3575693.3575706.

[129] Calvin Lin and Lawrence Snyder. "ZPL: An array sublanguage". In: *International Workshop on Languages and Compilers for Parallel Computing*. Springer. 1993, pp. 96–114.

[130] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. "Sparta: high-performance, element-wise sparse tensor contraction on heterogeneous memory". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 318–333. DOI: 10.1145/3437801.3441581. URL: https://doi.org/10.1145/3437801.3441581.

[131] Jie Liu, Zhongyuan Zhao, Zijian Ding, Benjamin Brock, Hongbo Rong, and Zhiru Zhang. "UniSparse: An Intermediate Language for General Sparse Format Customization". In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024). DOI: 10.1145/3649816. URL: https://doi.org/10.1145/3649816.

[132] Peiming Liu, Alexander J Root, Anlun Xu, Yinying Li, Fredrik Kjolstad, and Aart J.C. Bik. "Compiler Support for Sparse Tensor Convolutions". In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: 10.1145/3689721. URL: https://doi.org/10.1145/3689721.

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

[134] Liqiang Lu, Yicheng Jin, Hangrui Bi, Zizhang Luo, Peng Li, Tao Wang, and Yun Liang. "Sanger: A Co-Design Framework for Enabling Sparse Attention using Reconfigurable Architecture". In: *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '21. Virtual Event, Greece: Association for Computing Machinery, 2021, pp. 977–991. DOI: 10.1145/3466752.3480125. URL: https://doi.org/10.1145/3466752.3480125.

[135] José Wesley de Souza Magalhães, Jackson Woodruff, Elizabeth Polgreen, and Michael F. P. O'Boyle. "C2TACO: Lifting Tensor Code to TACO". In: *Proceedings of the 22nd ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences*. GPCE 2023. Cascais, Portugal: Association for Computing Machinery, 2023, pp. 42–56. DOI: 10.1145/3624007.3624053. URL: https://doi.org/10.1145/3624007.3624053.

[136] Paolo Mantovani, Davide Giri, Giuseppe Di Guglielmo, Luca Piccolboni, Joseph Zuckerman, Emilio G. Cota, Michele Petracca, Christian Pilato, and Luca P. Carloni. "Agile SoC development with open ESP". In: *Proceedings of the 39th International Conference on Computer-Aided Design*. ICCAD '20. Virtual Event, USA: Association for Computing Machinery, 2020. DOI: `10.1145/3400302.3415753`. URL: `https://doi.org/10.1145/3400302.3415753`.

[137] MATLAB. *version 7.10.0 (R2010a)*. Natick, Massachusetts: The MathWorks Inc., 2010.

[138] Devin Matthews. "High-Performance Tensor Contraction without BLAS". In: *CoRR* abs/1607.00291 (2016). arXiv: `1607.00291`. URL: `http://arxiv.org/abs/1607.00291`.

[139] Tim Mattson, David Bader, Jon Berry, Aydin Buluc, Jack Dongarra, Christos Faloutsos, John Feo, John Gilbert, Joseph Gonzalez, Bruce Hendrickson, et al. "Standards for graph algorithm primitives". In: *2013 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2013, pp. 1–2.

[140] John Michael McNamee. "Algorithm 408: a sparse matrix package (part I) [F4]". In: *Commun. ACM* 14.4 (Apr. 1971), pp. 265–273. DOI: `10.1145/362575.362584`. URL: `https://doi.org/10.1145/362575.362584`.

[141] Jackson Melchert, Yuchen Mei, Kalhan Koul, Qiaoyi Liu, Mark Horowitz, and Priyanka Raina. "Cascade: An Application Pipelining Toolkit for Coarse-Grained Reconfigurable Arrays". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2024), pp. 1–1. DOI: `10.1109/TCAD.2024.3390542`.

[142] Jackson Melchert, Keyi Zhang, Yuchen Mei, Mark Horowitz, Christopher Torng, and Priyanka Raina. "Canal: A Flexible Interconnect Generator for Coarse-Grained Reconfigurable Arrays". In: *IEEE Computer Architecture Letters* 22.1 (2023), pp. 45–48. DOI: `10.1109/LCA.2023.3268126`.

[143] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nico, and K. Crowley. "Principles of runtime support for parallel processors". In: *Proceedings of the 2nd International Conference on Supercomputing*. ICS '88. St. Malo, France: Association for Computing Machinery, 1988, pp. 140–152. DOI: `10.1145/55364.55378`. URL: `https://doi.org/10.1145/55364.55378`.

[144] Leonardo de Moura and Nikolaj Bjørner. "Z3: An Efficient SMT Solver". In: *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. TACAS'08/ETAPS'08. Budapest, Hungary: Springer-Verlag, 2008, pp. 337–340.

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

[146] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. "Automatically Scheduling Halide Image Processing Pipelines". In: *ACM Trans. Graph.* 35.4 (July 2016). DOI: 10.1145/2897824.2925952. URL: https://doi.org/10.1145/2897824.2925952.

[147] Francisco Muñoz-Martínez, Raveesh Garg, Michael Pellauer, José L. Abellán, Manuel E. Acacio, and Tushar Krishna. "Flexagon: A Multi-dataflow Sparse-Sparse Matrix Multiplication Accelerator for Efficient DNN Processing". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 252–265. DOI: 10.1145/3582016.3582069. URL: https://doi.org/10.1145/3582016.3582069.

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

[149] M. Naumov, L. S. Chien, P. Vandermersch, and U. Kapasi. "Cusparse library". In: *GPU Technology Conference (GTC)* (2010).

[150] Nandeeka Nayak, Toluwanimi O. Odemuyiwa, Shubham Ugare, Christopher Fletcher, Michael Pellauer, and Joel Emer. "TeAAL: A Declarative Framework for Modeling Sparse Tensor Accelerators". In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture*. MICRO '23. Toronto, ON, Canada: Association for Computing Machinery, 2023, pp. 1255–1270. DOI: 10.1145/3613424.3623791. URL: https://doi.org/10.1145/3613424.3623791.

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

[152] Dima Nikiforov, Shengjun Chris Dong, Chengyi Lux Zhang, Seah Kim, Borivoje Nikolic, and Yakun Sophia Shao. "RoSÉ: A Hardware-Software Co-Simulation Infrastructure Enabling Pre-Silicon Full-Stack Robotics SoC Evaluation". In: *Proceedings of the 50th Annual International Symposium on Computer Architecture*. ISCA '23. Orlando, FL, USA: Association for Computing Machinery, 2023. DOI: 10.1145/3579371.3589099. URL: https://doi.org/10.1145/3579371.3589099.

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

[154] NVIDIA. *CUTLASS*. NVIDIA. 2022. URL: `https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda/`.

[155] Toluwanimi O. Odemuyiwa, Hadi Asghari-Moghaddam, Michael Pellauer, Kartik Hegde, Po-An Tsai, Neal C. Crago, Aamer Jaleel, John D. Owens, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. "Accelerating Sparse Data Orchestration via Dynamic Reflexive Tiling". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 18–32. DOI: `10.1145/3582016.3582064`. URL: `https://doi.org/10.1145/3582016.3582064`.

[156] Toluwanimi O. Odemuyiwa, Joel S. Emer, and John D. Owens. *The EDGE Language: Extended General Einsums for Graph Algorithms*. 2024. arXiv: `2404.11591 [cs.DS]`. URL: `https://arxiv.org/abs/2404.11591`.

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

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

[159] 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: *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 2017, pp. 27–40. DOI: `10.1145/3079856.3080254`.

[160] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit

Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. "PyTorch: An Imperative Style, High-Performance Deep Learning Library". In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[161] Michael Pellauer, Jason Clemons, Vignesh Balaji, Neal Crago, Aamer Jaleel, Donghyuk Lee, Mike O'Connor, Angshuman Parashar, Sean Treichler, Po-An Tsai, Stephen W. Keckler, and Joel S. Emer. "Symphony: Orchestrating Sparse and Dense Tensors with Hierarchical Heterogeneous Processing". In: *ACM Trans. Comput. Syst.* 41.1–4 (Dec. 2023). DOI: 10.1145/3630007. URL: https://doi.org/10.1145/3630007.

[162] 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*. ASPLOS '19. Providence, RI, USA: Association for Computing Machinery, 2019, pp. 137–151. DOI: 10.1145/3297858.3304025. URL: https://doi.org/10.1145/3297858.3304025.

[163] Raghu Prabhakar, Sumti Jairath, and Jinuk Luke Shin. "SambaNova SN10 RDU: A 7nm Dataflow Architecture to Accelerate Software 2.0". In: *2022 IEEE International Solid-State Circuits Conference (ISSCC)*. Vol. 65. 2022, pp. 350–352. DOI: 10.1109/ISSCC42614.2022.9731612.

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

[165] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. "Plasticine: A Reconfigurable Architecture For Parallel Paterns". In: *SIGARCH Comput. Archit. News* 45.2 (June 2017), pp. 389–402. DOI: 10.1145/3140659.3080256. URL: https://doi.org/10.1145/3140659.3080256.

[166] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. "Programming Heterogeneous Systems from an Image Processing DSL". In: *ACM Trans. Archit. Code Optim.* 14.3 (Aug. 2017). DOI: 10.1145/3107953. URL: https://doi.org/10.1145/3107953.

[167] William Pugh and Tatiana Shpeisman. “SIPR: A New Framework for Generating Efficient Code for Sparse Matrix Computations”. In: *Languages and Compilers for Parallel Computing*. Ed. by Siddhartha Chatterjee, Jan F. Prins, Larry Carter, Jeanne Ferrante, Zhiyuan Li, David Sehr, and Pen-Chung Yew. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 213–229.

[168] Eric Qin, Geonhwa Jeong, William Won, Sheng-Chun Kao, Hyoukjun Kwon, Sudarshan Srinivasan, Dipankar Das, Gordon E. Moon, Sivasankaran Rajamanickam, and Tushar Krishna. “Extending Sparse Tensor Accelerators to Support Multiple Compression Formats”. In: *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2021, pp. 1014–1024. DOI: `10.1109/IPDPS49936.2021.00110`. URL: `https://doi.ieeecomputersociety.org/10.1109/IPDPS49936.2021.00110`.

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

[170] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. “Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines”. In: *ACM Trans. Graph.* 31.4 (July 2012). DOI: `10.1145/2185520.2185528`. URL: `https://doi.org/10.1145/2185520.2185528`.

[171] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. “Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines”. In: *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’13. Seattle, Washington, USA: Association for Computing Machinery, 2013, pp. 519–530. DOI: `10.1145/2491956.2462176`. URL: `https://doi.org/10.1145/2491956.2462176`.

[172] Saurabh Raje, Hunter McCoy, Atanas Rountev, Prashant Pandey, and P. Sadayappan. “FaSTCC: Fast Sparse Tensor Contractions on CPUs”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’25. Association for Computing Machinery, 2025, pp. 617–630. DOI: `10.1145/3712285.3759841`. URL: `https://doi.org/10.1145/3712285.3759841`.

[173] Saurabh Raje, Yufan Xu, Atanas Rountev, Edward F. Valeev, and P. Sadayappan. “CoNST: Code Generator for Sparse Tensor Networks”. In: *ACM Trans. Archit. Code Optim.* 21.4 (Nov. 2024). DOI: `10.1145/3689342`. URL: `https://doi.org/10.1145/3689342`.

[174] M.M.G. Ricci and T. Levi-Civita. “Méthodes de calcul différentiel absolu et leurs applications”. In: *Mathematische Annalen* 54 (1901), pp. 125–201. URL: `http://eudml.org/doc/157997`.

[175] Alexander J Root, Bobby Yan, Peiming Liu, Christophe Gyurgyik, Aart J.C. Bik, and Fredrik Kjolstad. "Compilation of Shape Operators on Sparse Arrays". In: *Proc. ACM Program. Lang.* 8.OOPSLA2 (Oct. 2024). DOI: `10.1145/3689752`. URL: `https://doi.org/10.1145/3689752`.

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

[177] Alexander C. Rucker, Shiv Sundram, Coleman Smith, Matthew Vilim, Raghu Prabhakar, Fredrik Kjolstad, and Kunle Olukotun. "Revet: A Language and Compiler for Dataflow Threads". In: *2024 IEEE International Symposium on High-Performance Computer Architecture (HPCA).* Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2024, pp. 1–14. DOI: `10.1109/HPCA57654.2024.00016`. URL: `https://doi.ieeecomputersociety.org/10.1109/HPCA57654.2024.00016`.

[178] Yousef Saad. *Sparsekit: A basic tool kit for sparse matrix computations.* Technical Report. University of Minnesota, 1994.

[179] Nobou Sato and W. F. Tinney. "Techniques for Exploiting the Sparsity or the Network Admittance Matrix". In: *IEEE Transactions on Power Apparatus and Systems* 82.69 (1963), pp. 944–950. DOI: `10.1109/TPAS.1963.291477`.

[180] SciPy. *SciPy Roadmap V1.6.2.* [Online; accessed 04/12/2021]. 2021. URL: `https://docs.scipy.org/doc/scipy-1.6.2/reference/roadmap.html`.

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

[182] Deval Shah and Tor M. Aamodt. "Collision Prediction for Robotics Accelerators". In: *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA).* 2024, pp. 566–581. DOI: `10.1109/ISCA59077.2024.00048`.

[183] Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. "FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-precision". In: *The Thirty-eighth Annual Conference on Neural Information Processing Systems.* 2024. URL: `https://openreview.net/forum?id=tVConYid20`.

[184] Yi Shan, Tianji Wu, Yu Wang, Bo Wang, Zilong Wang, Ningyi Xu, and Huazhong Yang. "FPGA and GPU implementation of large scale SpMV". In: *2010 IEEE 8th Symposium on Application Specific Processors (SASP).* IEEE. 2010, pp. 64–70.

[185] Sophia Yakun Shao. "Design and Modeling of Specialized Architectures". PhD thesis. Harvard University, 2016.

[186] Ritvik Sharma, Zi Yu Xue, Nathan Zhang, Rubens Lacouture, Fredrik Kjolstad, Sara Achour, and Mark Horowitz. "A Probabilistic Perspective on Tiling Sparse Tensor Algebra". In: *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture.* MICRO '25. New York, NY, USA: Association for Computing Machinery, 2025, pp. 795–808. DOI: `10.1145/3725843.3756095`. URL: `https://doi.org/10.1145/3725843.3756095`.

[187] Navjot Singh, Zecheng Zhang, Xiaoxiao Wu, Naijing Zhang, Siyuan Zhang, and Edgar Solomonik. "Distributed-memory tensor completion for generalized loss functions in python using new sparse tensor kernels". In: *J. Parallel Distributed Comput.* 169 (2022), pp. 269–285. DOI: `10.1016/j.jpdc.2022.07.005`. URL: `https://doi.org/10.1016/j.jpdc.2022.07.005`.

[188] Marco Siracusa, Olivia Hsu, Victor Soria-Pardos, Joshua Randall, Arnaud Grasset, Eric Biscondi, Doug Joseph, Randy Allen, Fredrik Kjolstad, Miquel Moretó Planas, and Adrià Armejach. "Ember: A Compiler for Efficient Embedding Operations on Decoupled Access-Execute Architectures". In: *Proceedings of the 2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO).* Sydney, Australia, 2026.

[189] Marco Siracusa, Víctor Soria-Pardos, Francesco Sgherzi, Joshua Randall, Douglas J. Joseph, Miquel Moretó Planas, and Adrià Armejach. "A Tensor Marshaling Unit for Sparse Tensor Algebra on General-Purpose Processors". In: *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture.* MICRO '23. Toronto, ON, Canada: Association for Computing Machinery, 2023, pp. 1332–1346. DOI: `10.1145/3613424.3614284`. URL: `https://doi.org/10.1145/3613424.3614284`.

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

[191] Shaden Smith, Niranjay Ravindran, Nicholas D. Sidiropoulos, and George Karypis. "SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication". In: *2015 IEEE International Parallel and Distributed Processing Symposium.* 2015, pp. 61–70. DOI: `10.1109/IPDPS.2015.27`.

[192] Gina Sohn, Genghan Zhang, Konstantin Hossfeld, Jungwoo Kim, Nathan Sobotka, Nathan Zhang, Olivia Hsu, and Kunle Olukotun. *Streaming Tensor Program: A streaming abstraction for dynamic parallelism.* 2025. arXiv: `2511.07776 [cs.PL]`. URL: `https://arxiv.org/abs/2511.07776`.

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

[194] Edgar Solomonik, Devin Matthews, Jeff R. Hammond, John F. Stanton, and James Demmel. “A massively parallel tensor contraction framework for coupled-cluster computations”. In: *Journal of Parallel and Distributed Computing* 74.12 (2014). Domain-Specific Languages and High-Level Frameworks for High-Performance Computing, pp. 3176–3190. DOI: https://doi.org/10.1016/j.jpdc.2014.06.002. URL: https://www.sciencedirect.com/science/article/pii/S074373151400104X.

[195] K. Somkantha, N. Theera-Umpon, and S. Auephanwiriyakul. “Boundary Detection in Medical Images Using Edge Following Algorithm Based on Intensity Gradient and Texture Gradient Features”. In: *IEEE Transactions on Biomedical Engineering* 58.3 (2011), pp. 567–573. DOI: 10.1109/TBME.2010.2091129.

[196] Linghao Song, Yuze Chi, Licheng Guo, and Jason Cong. “Serpens: a high bandwidth memory based accelerator for general-purpose sparse matrix-vector multiplication”. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. DAC ’22. San Francisco, California: Association for Computing Machinery, 2022, pp. 211–216. DOI: 10.1145/3489517.3530420. URL: https://doi.org/10.1145/3489517.3530420.

[197] Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong. “Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication”. In: *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. FPGA ’22. Virtual Event, USA: Association for Computing Machinery, 2022, pp. 65–77. DOI: 10.1145/3490422.3502357. URL: https://doi.org/10.1145/3490422.3502357.

[198] Seokchan Song, Donghyeon Han, Sangjin Kim, Sangyeob Kim, Gwangtae Park, and Hoi-Jun Yoo. “GPPU: A 330.4-μJ/task Neural Path Planning Processor with Hybrid GNN Acceleration for Autonomous 3D Navigation”. In: *2023 IEEE Symposium on VLSI Technology and Circuits*. 2023, pp. 1–2. DOI: 10.23919/VLSITechnologyandCir57934.2023.10185367.

[199] *Sparse Multidimensional Arrays Github*. 2025. URL: https://github.com/pydata/sparse/tree/main/sparse (visited on 11/02/2025).

[200] Benjamin Frederick Spector, Simran Arora, Aaryan Singhal, Arjun Parthasarathy, Daniel Y Fu, and Christopher Re. “ThunderKittens: Simple, Fast, and $\textit{Adorable}$ Kernels”. In: *The Thirteenth International Conference on Learning Representations*. 2025. URL: https://openreview.net/forum?id=0fJfVOSUra.

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

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

[203] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanaël Prémillieu, Alastair Reid, Alejandro Rico, and Paul Walker. "The ARM Scalable Vector Extension". In: *CoRR* abs/1803.06185 (2018). arXiv: `1803.06185`. URL: `http://arxiv.org/abs/1803.06185`.

[204] Michelle Mills Strout, Mary Hall, and Catherine Olschanowsky. "The Sparse Polyhedral Framework: Composing Compiler-Generated Inspector-Executor Code". In: *Proceedings of the IEEE* 106.11 (2018), pp. 1921–1934. DOI: `10.1109/JPROC.2018.2857721`.

[205] Chamika Sudusinghe, Gerasimos Gerogiannis, Damitha Lenadora, Charles Block, Josep Torrellas, and Charith Mendis. *COGNATE: Acceleration of Sparse Tensor Programs on Emerging Hardware using Transfer Learning.* 2025. arXiv: `2506.00424 [cs.LG]`. URL: `https://arxiv.org/abs/2506.00424`.

[206] Amr Suleiman, Zhengdong Zhang, Luca Carlone, Sertac Karaman, and Vivienne Sze. "Navion: A 2-mW Fully Integrated Real-Time Visual-Inertial Odometry Accelerator for Autonomous Navigation of Nano Drones". In: *IEEE Journal of Solid-State Circuits* 54.4 (2019), pp. 1106–1119. DOI: `10.1109/JSSC.2018.2886342`.

[207] Shiv Sundram, Muhammad Usman Tariq, and Fredrik Kjolstad. "Compiling Recurrences over Dense and Sparse Arrays". In: *Proc. ACM Program. Lang.* 8.OOPSLA1 (Apr. 2024). DOI: `10.1145/3649820`. URL: `https://doi.org/10.1145/3649820`.

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

[209] Muhammad Usman Tariq, Shiv Sundram, and Fredrik Kjolstad. "REPTILE: Performant Tiling of Recurrences". In: *Proc. ACM Program. Lang.* 9.OOPSLA2 (Oct. 2025). DOI: `10.1145/3763074`. URL: `https://doi.org/10.1145/3763074`.

[210] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. "A High Performance Sparse Tensor Algebra Compiler in MLIR". In: *2021 IEEE/ACM 7th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. 2021, pp. 27–38. DOI: `10.1109/LLVMHPC54804.2021.00009`.

[211] W.F. Tinney and J.W. Walker. "Direct solutions of sparse network equations by optimally ordered triangular factorization". In: *Proceedings of the IEEE* 55.11 (1967), pp. 1801–1809. DOI: 10.1109/PROC.1967.6011.

[212] Christopher Torng, Peitian Pan, Yanghui Ou, Cheng Tan, and Christopher Batten. "Ultra-Elastic CGRAs for Irregular Loop Specialization". In: *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 2021, pp. 412–425. DOI: 10.1109/HPCA51647.2021.00042.

[213] Yaman Umuroglu and Magnus Jahre. "An energy efficient column-major backend for FPGA SpMV accelerators". In: *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE. 2014, pp. 432–439.

[214] Miheer Vaidya, Shreya Singh, Devanshu Mantri, Michael Shannon Eydenberg, Brian Michael Kelley, Sivasankaran Rajamanickam, Atanas Rountev, and P. Sadayappan. "Distributed Sparse Tensor Computations in MLIR". In: *Proceedings of the SC '25 Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC Workshops '25. Association for Computing Machinery, 2025, pp. 1088–1095. DOI: 10.1145/3731599.3767484. URL: https://doi.org/10.1145/3731599.3767484.

[215] Field G. Van Zee and Robert A. van de Geijn. "BLIS: A Framework for Rapidly Instantiating BLAS Functionality". In: *ACM Trans. Math. Softw.* 41.3 (June 2015). DOI: 10.1145/2764454. URL: https://doi.org/10.1145/2764454.

[216] Anand Venkat, Mary Hall, and Michelle Strout. "Loop and data transformations for sparse matrix code". In: *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '15. Portland, OR, USA: Association for Computing Machinery, 2015, pp. 521–532. DOI: 10.1145/2737924.2738003. URL: https://doi.org/10.1145/2737924.2738003.

[217] Francisco-Javier Veredas, M. Scheppler, W. Moffat, and Bingfeng Mei. "Custom implementation of the coarse-grained reconfigurable ADRES architecture for multimedia purposes". In: *International Conference on Field Programmable Logic and Applications*. 2005, pp. 106–111. DOI: 10.1109/FPL.2005.1515707.

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

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

[220] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. "SciPy 1.0: fundamental algorithms for scientific computing in Python". In: *Nature methods* 17.3 (2020), pp. 261–272.

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

[222] Stefan van der Walt, S. Chris Colbert, and Gael Varoquaux. "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science and Engg.* 13.2 (Mar. 2011), pp. 22–30. DOI: `10.1109/MCSE.2011.37`. URL: `https://doi.org/10.1109/MCSE.2011.37`.

[223] Jian Weng, Sihao Liu, Dylan Kupsh, and Tony Nowatzki. "Unifying spatial accelerator compilation with idiomatic and modular transformations". In: *IEEE Micro* 42.5 (2022), pp. 59–69.

[224] R. Clint Whaley and Antoine Petitet. "Minimizing development and maintenance costs in supporting persistently optimized BLAS". In: *Software: Practice and Experience* 35.2 (2005), pp. 101–121. DOI: `https://doi.org/10.1002/spe.626`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.626`. URL: `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.626`.

[225] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. "Optimization of sparse matrix-vector multiplication on emerging multicore platforms". In: *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*. SC '07. Reno, Nevada: Association for Computing Machinery, 2007. DOI: `10.1145/1362622.1362674`. URL: `https://doi.org/10.1145/1362622.1362674`.

[226] Gert Wollny, Peter Kellman, María J. Ledesma-Carbayo, Matthew M. Skinner, Jean-Jaques Hublin, and Thomas Hierl. "MIA - A free and open source software for gray scale medical image analysis". In: *Source Code Biol Med* (2013). URL: `https://doi.org/10.1186/1751-0473-8-20`.

[227] Jaeyeon Won, Willow Ahrens, Joel S. Emer, and Saman Amarasinghe. *Insum: Sparse GPU Kernels Simplified and Optimized with Indirect Einsums*. 2025. arXiv: `2510.17505 [cs.PL]`. URL: `https://arxiv.org/abs/2510.17505`.

[228] Jaeyeon Won, Willow Ahrens, Joel S. Emer, and Saman Amarasinghe. *The Continuous Tensor Abstraction: Where Indices are Real*. 2024. arXiv: `2407.01742 [cs.PL]`. URL: `https://arxiv.org/abs/2407.01742`.

[229] Jaeyeon Won, Changwan Hong, Charith Mendis, Joel Emer, and Saman Amarasinghe. "Unified Convolution Framework: A compiler-based approach to support sparse convolutions". In: *Proceedings of Machine Learning and Systems*. Ed. by D. Song, M. Carbin, and T. Chen. Vol. 5. Curan, 2023, pp. 666–679. URL: `https://proceedings.mlsys.org/paper_files/paper/2023/file/ccf7262fb986e4367ccd3903960c57a0-Paper-mlsys2023.pdf`.

[230] Jaeyeon Won, Charith Mendis, Joel S. Emer, and Saman Amarasinghe. "WACO: Learning Workload-Aware Co-optimization of the Format and Schedule of a Sparse Tensor Program". In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 920–934. DOI: `10.1145/3575693.3575742`. URL: `https://doi.org/10.1145/3575693.3575742`.

[231] 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)*. 2021, pp. 232–234. DOI: `10.1109/ISPASS51385.2021.00043`.

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

[233] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. "SpDISTAL: compiling distributed sparse tensor computations". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '22. Dallas, Texas: IEEE Press, 2022.

[234] Rohan Yadav, Michael Garland, Alex Aiken, and Michael Bauer. "Task-Based Tensor Computations on Modern GPUs". In: *Proc. ACM Program. Lang.* 9.PLDI (June 2025). DOI: `10.1145/3729262`. URL: `https://doi.org/10.1145/3729262`.

[235] Rohan Yadav, Wonchan Lee, Melih Elibol, Manolis Papadakis, Taylor Lee-Patti, Michael Garland, Alex Aiken, Fredrik Kjolstad, and Michael Bauer. "Legate Sparse: Distributed Sparse Computing in Python". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '23. Denver, CO, USA: Association for Computing Machinery, 2023. DOI: `10.1145/3581784.3607033`. URL: `https://doi.org/10.1145/3581784.3607033`.

[236] Sanjali Yadav and Bahar Asgari. "Bootes: Boosting the Efficiency of Sparse Accelerators Using Spectral Clustering". In: *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. MICRO '25. Association for Computing Machinery, 2025, pp. 809–823. DOI: `10.1145/3725843.3756125`. URL: `https://doi.org/10.1145/3725843.3756125`.

[237] Sanjali Yadav, Amirmahdi Namjoo, and Bahar Asgari. “Misam: Machine Learning Assisted Dataflow Selection in Accelerators for Sparse Matrix Multiplication”. In: *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. MICRO ’25. Association for Computing Machinery, 2025, pp. 824–838. DOI: `10.1145/3725843.3756126`. URL: `https://doi.org/10.1145/3725843.3756126`.

[238] Bobby Yan, Alexander J. Root, Trevor Gale, David Broman, and Fredrik Kjolstad. *Scorch: A Library for Sparse Deep Learning*. 2024. arXiv: `2405.16883 [cs.LG]`. URL: `https://arxiv.org/abs/2405.16883`.

[239] Bobby Yan, Alexander J. Root, Trevor Gale, David Broman, and Fredrik Kjolstad. “Scorch: A Library for Sparse Deep Learning”. In: *Proceedings of the 2026 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. Sydney, Australia, 2026.

[240] Yifan Yang, Joel S. Emer, and Daniel Sanchez. “Trapezoid: A Versatile Accelerator for Dense and Sparse Matrix Multiplications”. In: *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 2024, pp. 931–945. DOI: `10.1109/ISCA59077.2024.00072`.

[241] Hanchen Ye and Deming Chen. “StreamTensor: Make Tensors Stream in Dataflow Accelerators for LLMs”. In: *Proceedings of the 58th IEEE/ACM International Symposium on Microarchitecture*. MICRO ’25. Association for Computing Machinery, 2025, pp. 201–216. DOI: `10.1145/3725843.3762817`. URL: `https://doi.org/10.1145/3725843.3762817`.

[242] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. “SparseTIR: Composable Abstractions for Sparse Compilation in Deep Learning”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS 2023. Vancouver, BC, Canada: Association for Computing Machinery, 2023, pp. 660–678. DOI: `10.1145/3582016.3582047`. URL: `https://doi.org/10.1145/3582016.3582047`.

[243] Qing Yi. “POET: A Scripting Language for Applying Parameterized Source-to-Source Program Transformations”. In: *Softw. Pract. Exper.* 42.6 (June 2012), pp. 675–706. DOI: `10.1002/spe.1089`. URL: `https://doi.org/10.1002/spe.1089`.

[244] Genghan Zhang, Olivia Hsu, and Fredrik Kjolstad. “Compilation of Modular and General Sparse Workspaces”. In: *Proc. ACM Program. Lang.* 8.PLDI (June 2024). DOI: `10.1145/3656426`. URL: `https://doi.org/10.1145/3656426`.

[245] Genghan Zhang, Weixin Liang, Olivia Hsu, and Kunle Olukotun. “Adaptive Self-improvement LLM Agentic System for ML Library Development”. In: *Forty-second International Conference on Machine Learning*. 2025. URL: `https://openreview.net/forum?id=gdsZ3uMPsY`.

[246] 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 '21. Virtual, USA: Association for Computing Machinery, 2021, pp. 687–701. DOI: 10.1145/3445814.3446702. URL: https://doi.org/10.1145/3445814.3446702.

[247] Qirui Zhang, Hyochan An, Zichen Fan, Zhehong Wang, Ziyun Li, Guanru Wang, Hun-Seok Kim, David Blaauw, and Dennis Sylvester. "A 22nm 3.5TOPS/W Flexible Micro-Robotic Vision SoC with 2MB eMRAM for Fully-on-Chip Intelligence". In: *2022 IEEE Symposium on VLSI Technology and Circuits*. 2022, pp. 72–73. DOI: 10.1109/VLSITechnologyandCir46769.2022.9830340.

[248] 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)*. 2016, pp. 1–12. DOI: 10.1109/MICRO.2016.7783723.

[249] Yan Zhang, Yasser H Shalabi, Rishabh Jain, Krishna K Nagar, and Jason D Bakos. "FPGA vs. GPU for sparse matrix vector multiply". In: *2009 International Conference on Field-Programmable Technology*. IEEE. 2009, pp. 255–262.

[250] Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. "Scalable Interconnects for Reconfigurable Spatial Architectures". In: *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 2019, pp. 615–628.

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

[252] Z. Zhang, H. Wang, S. Han, and W. J. Dally. "SpArch: Efficient Architecture for Sparse Matrix Multiplication". In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. Los Alamitos, CA, USA: IEEE Computer Society, Feb. 2020, pp. 261–274. DOI: 10.1109/HPCA47549.2020.00030. URL: https://doi.ieeecomputersociety.org/10.1109/HPCA47549.2020.00030.

[253] Tuowen Zhao, Tobi Popoola, Mary Hall, Catherine Olschanowsky, and Michelle Strout. "Polyhedral Specification and Code Generation of Sparse Tensor Contraction with Co-Iteration". In: *ACM Trans. Archit. Code Optim.* 20.1 (Dec. 2022). DOI: 10.1145/3566054. URL: https://doi.org/10.1145/3566054.

[254] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. "AMOS: Enabling Automatic Mapping for Tensor Computations On Spatial Accelerators with Hardware Abstraction". In: *Proceedings of the 49th Annual International Symposium on Computer Architecture*. ISCA '22. New York, New York: Association for Computing Machinery, 2022, pp. 874–887. DOI: `10.1145/3470496.3527440`. URL: `https://doi.org/10.1145/3470496.3527440`.

[255] Kai Zhong, Zhenhua Zhu, Guohao Dai, Hongyi Wang, Xinhao Yang, Haoyu Zhang, Jin Si, Qiuli Mao, Shulin Zeng, Ke Hong, Genghan Zhang, Huazhong Yang, and Yu Wang. "FEASTA: A Flexible and Efficient Accelerator for Sparse Tensor Algebra in Machine Learning". In: *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ASPLOS '24. La Jolla, CA, USA: Association for Computing Machinery, 2024, pp. 349–366. DOI: `10.1145/3620666.3651336`. URL: `https://doi.org/10.1145/3620666.3651336`.

[256] Tong Zhou, Ruiqin Tian, Rizwan A. Ashraf, Roberto Gioiosa, Gokcen Kestor, and Vivek Sarkar. "ReACT: Redundancy-Aware Code Generation for Tensor Expressions". In: *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*. PACT '22. Chicago, Illinois: Association for Computing Machinery, 2023, pp. 1–13. DOI: `10.1145/3559009.3569685`. URL: `https://doi.org/10.1145/3559009.3569685`.

[257] Xuda Zhou, Zidong Du, Qi Guo, Shaoli Liu, Chengsi Liu, Chao Wang, Xuehai Zhou, Ling Li, Tianshi Chen, and Yunji Chen. "Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach". In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 2018, pp. 15–28. DOI: `10.1109/MICRO.2018.00011`.