RESEARCH-ARTICLE

Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture

OLIVIA HSU, Stanford University, Stanford, CA, United States

ALEXANDER RUCKER, Stanford University, Stanford, CA, United States

TIAN ZHAO, Stanford University, Stanford, CA, United States

VARUN DESAI, Stanford University, Stanford, CA, United States

KUNLE OLUKOTUN, Stanford University, Stanford, CA, United States

FREDRIK BERG KJØLSTAD, Stanford University, Stanford, CA, United States

Published: 01 March 2025

Citation in BibTeX format

CGO '25: 23rd ACM/IEEE International Symposium on Code Generation and Optimization
March 1 - 5, 2025
NV, Las Vegas, USA

Conference Sponsors:
SIGPLAN
SIGMICRO

Stardust: Compiling Sparse Tensor Algebra to a Reconfigurable Dataflow Architecture

Olivia Hsu
Stanford University
Stanford, USA
owhsu@stanford.edu

Alexander Rucker
Stanford University
Stanford, USA
acrucker@stanford.edu

Tian Zhao
Stanford University
Stanford, USA
tianzhao@stanford.edu

Varun Desai
Stanford University
Stanford, USA
vdesai@stanford.edu

Kunle Olukotun
Stanford University
Stanford, USA
kunle@stanford.edu

Fredrik Kjolstad
Stanford University
Stanford, USA
kjolstad@stanford.edu

Abstract

We introduce Stardust, a compiler from sparse tensor algebra languages to a sparse reconfigurable dataflow architecture via a parallel-patterns programming model. Stardust lets performance engineers specify the placement of data into memories separately from the placement of computation onto compute units. Users first schedule data placement onto an abstract memory model, and then Stardust binds that data to complex, on-chip physical memories. With guidance from user schedules, Stardust binds computation using these on-chip data structures to the appropriate parallel patterns. Through cycle-accurate simulation, we show that Stardust generates nine more tensor algebra kernels than the original Capstan sparse RDA work. The generated kernels perform, on average, 138× better than generated CPU kernels and 41× better than generated GPU kernels.

CCS Concepts: • **Computer systems organization** → **Data flow architectures**; • **Software and its engineering** → *Domain specific languages*; **Compilers**.

Keywords: sparse tensor algebra, DSLs, compilers, dataflow, reconfigurable architectures, parallel patterns

ACM Reference Format:
Olivia Hsu, Alexander Rucker, Tian Zhao, Varun Desai, Kunle Olukotun, and Fredrik Kjolstad. 2025. 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), March 01–05, 2025, Las Vegas, NV, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3696443.3708918

CGO '25, March 01–05, 2025, Las Vegas, NV, USA
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1275-3/25/03
https://doi.org/10.1145/3696443.3708918

1 Introduction

Reconfigurable dataflow architectures (RDAs) with sparse operation support [7, 12, 26] is a promising approach for accelerating sparse tensor algebra. To enable widespread use, these RDAs should be accessible to performance engineers who do not understand the specifics of the underlying architecture. However, state-of-the-art RDAs are difficult to program; typically, only users who have low-level knowledge of the architecture (e.g., the designers themselves) can effectively program them. Currently, programmers use hardware configuration files [12] or languages [7, 20, 26] with low-level architectural information embedded in the programming model to program RDAs.

To make RDAs easier to program, we must raise the programming abstraction above that of a specific RDA design. A higher-level programming abstraction enables performance engineers, who are not RDA experts, to write sparse tensor algebra kernels that leverage these RDAs. If performance engineers can readily develop sparse libraries with RDA acceleration, then end-users will get better application performance. Such programming abstractions can be realized through advances in compilation techniques.

Tensor index notation (or Einsum notation) is a natural representation of sparse tensor algebra computation. It is a mathematical notation and computing language that uses tensor indices to express tensor operations. Programming sparse RDAs using this notation means users only have to describe the mathematical expression, along with sparse tensor data structures. Performance engineers can then map the computation to an RDA using conventional performance engineering knowledge by deciding which computations should be executed on the accelerator, data movement, and tiling for locality. In order to enable this programming methodology, we propose a compilation stack that compiles tensor index notation along with a schedule and tensor data structure descriptions to an RDA.

Prior work has made important strides toward this goal, but an end-to-end compilation stack does not yet exist. Two major approaches partially address compiling to sparse RDAs.

The Custard compiler in SAM [14] provides a compilation framework from tensor index notation to a streaming dataflow intermediate representation (IR). However, SAM is only an abstraction and does not automatically lower to any concrete sparse RDA. The Spatial language is a domain-specific language (DSL) based on parallel patterns (such as map and reduce) with compilers [20, 41] that target the Capstan [26]. Spatial, however, still requires programmers to write complex code, which includes detailed architectural knowledge of Capstan and its intricate memory model. Thus, neither approach is complete in compiling from raising the programming abstraction of sparse RDAs. We build on the existing system infrastructure of Capstan and provide the missing step from high-level tensor index notation to Spatial code running on the RDA and its host.

There are several challenges when compiling to RDAs: 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 [4, 22, 24–26]. 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 [19].

Therefore, 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 [17] to the Capstan RDA [26]. 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 [26]. 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.

2 Background

Our work builds on two lines of prior work: sparse tensor algebra compilation techniques for CPUs [6, 17, 27] and the extended Spatial DSL [20] that targets the Capstan RDA [26].

2.1 Sparse Tensor Algebra Compilation

The TACO compiler separates the algorithm (tensor index notation) from the tensor compression formats and computation transformations through the use of format [6] and scheduling [27] languages, respectively. It compiles sparse tensor algebra to imperative code by decomposing sparse iteration spaces into hierarchical set expressions of per-dimension data structures. Sparse algorithms are expressed in CIN (see Figure 1), which encodes iteration, computation, transformations, and temporary tensors [16]. Finally, TACO lowers CIN to generate efficient fused code that traverses irregular data structures by skipping unnecessary computation.

Scheduling. The sparse scheduling language proposed by Senanayake et al. [27] 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. See Table 1 for a reference to one TACO scheduling command.

Format Language. The format language proposed by Chou et al. [6] 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 8 for an example matrix) can be represented by an uncompressed (dense) dimension followed by a compressed

Index Variable i *Index Variable List* $i*$ *Constants* c *Tensors* $\mathcal{T}$

Accesses	a	::=	$\mathcal{T}_{i*}$
Expressions	e	::=	$a \mid c \mid e + e \mid e * e \mid \ldots$
Assignment	A	::=	$a = e \mid a \mathrel{+}= e$
Statements	S	::=	$\forall_{i*}\ S \mid A \mid$
			$S\ ;\ S \mid S$ where $S \mid S$ s.t. $r*$
Scheduling Relation	r	::=	$\text{split}(i, i_o, i_i, c) \mid \text{fuse}(i_o, i_i, i_f) \mid \ldots$

Figure 1. Concrete index notation (CIN) syntax.

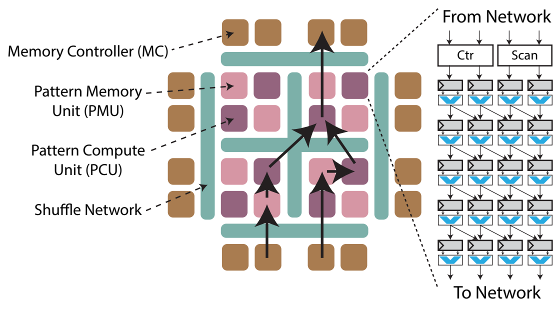


Figure 2. A high-level overview of the Capstan architecture, showing the opportunities for high-level parallelism across PCUs and vectorized parallelism within a PCU.

(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.

2.2 Capstan and Spatial

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 [26], shown in Figure 2, derives from Plasticine [25] 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 [20][1]. Compilers that handle low-level optimizations and insert memory-consistency logic [20, 41] 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 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 9 and described later in Section 7.

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).

3 Motivating Example

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 [9], 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): $\forall_i \forall_j \forall_k \left(A_{ij} \mathrel{+}= B_{ij} C_{ik} D_{kj}\right)$.

This notation is called concrete index notation (CIN) [16], and we provide its syntax in Figure 1. Many compiler decisions in prior work presuppose an imperative target language. Figure 3 shows the C-like code generated from the CIN statement by one suchcompiler [19]. The following code locations in Figure 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 $\forall$ 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 4 manages data movement. Specifically, Stardust must address the following issues when compiling to Spatial code as shown in Figure 4: ❺ the $\forall$ nodes are converted to *different* parallel patterns, which may include sparse patterns

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

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 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.

4 Overview

We implement Stardust as a new compilation path inside the open-source TACO system [17] as shown in Figure 5, where blue indicates our contributions. Like TACO, Stardust takes as input tensor index notation, a format language [6], and a scheduling language [27]. 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 [20], which is then compiled using prior work [20, 41] to a cycle-accurate simulations of the Capstan sparse RDA [26].

Tensor index notation lowers to *concrete index notation* (CIN) [16], a loop-based IR where compressed tensor data structures are abstracted away (shown in Figure 1). Scheduling language commands are applied as rewrites on CIN.

```
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 3. C implementation of SDDMM with CSR matrices, generated by the TACO compiler.

Table 1 has one example of the original scheduling commands in TACO, which we extend to target parallel patterns in Section 7. 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 5). 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). Stardust then generates the required data transfer code between memory types.

```
1  // Spatial header code ...
2  // Initialize all DRAM arrays as <name>_d
3  val A2_pos_d = DRAM[T](nnz_max)
4  ...
5  Accel {
6    val B2_pos = SRAM[T](nnz_accel_max)
7    B2_pos load B2_pos_d(0::(B1_dim + 1) par ip)
8  ❺Foreach (C1_dim by 1 par bp) { i =>
9      val A_vals = FIFO[T](16)
10     val A2_crd = FIFO[T](16)
11     val A2_pos = SRAM[T](nnz_accel_max)
12     val jB_start = B2_pos(i)
13     val jB_end = B2_pos((i + 1))
14     val jB_len = jB_end - jB_start
15     val B2_crd = FIFO[T](16)
16     B2_crd load B2_crd_d(jB_start::jB_end par 1)❻
17     val B_vals = FIFO[T](16)
18     B_vals load B_vals_d(jB_start::jB_end par 1)
19     Foreach (jB_len by 1 par 1) { jB =>
20       val j = B2_crd.deq
21       val B_hoisted = B_vals.deq❼
22
23       val D_vals = SRAM[T]((nnz_accel_max / 4))
24       D_vals load D_vals_d(j*D1_dim::(j+1)*D1_dim par ip)
25       val C_vals = SRAM[T]((nnz_accel_max / 4))
26       C_vals load C_vals_d(i*C2_dim::(i+1)*C2_dim par ip)
27
28       val tjA_vals = Reg[T](0.to[T])
29     ❽Reduce(tjA_vals)(D1_dim by 1 par ip) { k =>
30         ((B_hoisted * C_vals(k)) * D_vals(k))
31       } { _ + _ }
32       A_vals.enq(tjA_vals)
33       A2_crd.enq(j)
34     }
35     A2_pos(i + 1) = jB_end
36     A_vals_d stream_store_vec(jB_start, A_vals, jB_len)
37   }}
```

Figure 4. Spatial implementation of SDDMM with CSR matrices. Lines highlighted in blue are memory management.

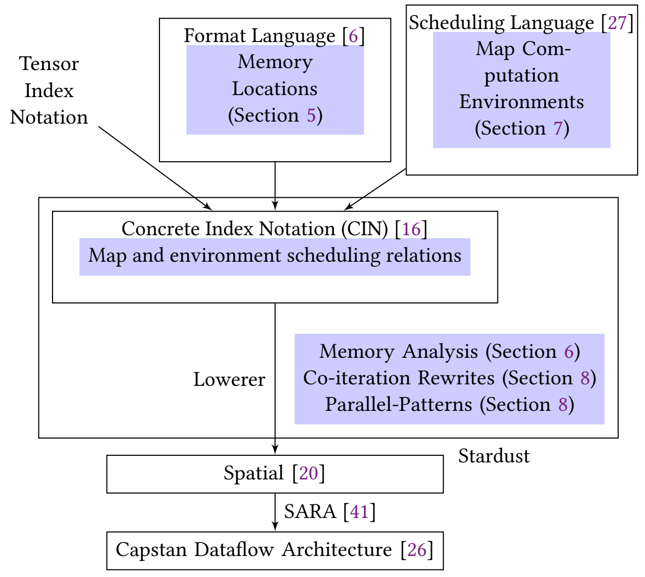

Tensor Index Notation
Format Language [6]
Memory Locations (Section 5)
Scheduling Language [27]
Map Computation Environments (Section 7)
Concrete Index Notation (CIN) [16]
Map and environment scheduling relations
Lowerer
Memory Analysis (Section 6)
Co-iteration Rewrites (Section 8)
Parallel-Patterns (Section 8)
Stardust
Spatial [20]
SARA [41]
Capstan Dataflow Architecture [26]

Figure 5. Stardust overview. Blue denotes new contributions.

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

Figure 6. Stardust input (user) code for SDDMM.

Once tensors are placed on the accelerator using the format language, users must also map the computation that uses those tensors onto to the accelerator (Section 7). 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 8).

5 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.

5.1 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 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 lines 2–4. Lines 6–9 in Figure 6 then demonstrate how the format language is used to declare the input and output tensors of an index notation expression.

5.2 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 [5, 14]. 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 1). 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 stores intermediate values.

A user inserts the memory-annotated temporary tensor into CIN via the `precompute` scheduling command [16], whose C++ declaration is in Table 1. The `precompute` command transforms a CIN statement with a sub-expression e into a new CIN statement with a where sub-statement. A

Table 1. The precompute command from the scheduling language of the TACO work [16, 27]. $e[x'/x]$ denotes the expression e with each occurrence of x replaced by x'.

Scheduling Commands	**Description**
`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 $\forall_{i_w*} \mathcal{T}(i_w*) = e[i_w*/i*]$

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

```
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.

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

```
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 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.

where is a producer-consumer statement whose sides 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 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 7a uses two temporary vectors whereas Figure 7b uses two temporary matrices. The two schedules also differ in which indices load the tensor data—Figure 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 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 has yet another schedule different from Figure 7, where off-chip data is loaded into an on-chip scalar temporary (line 21).

6 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 6.2), and then performs a memory lifetime analysis to generate allocation and transfer code (Section 6.3).

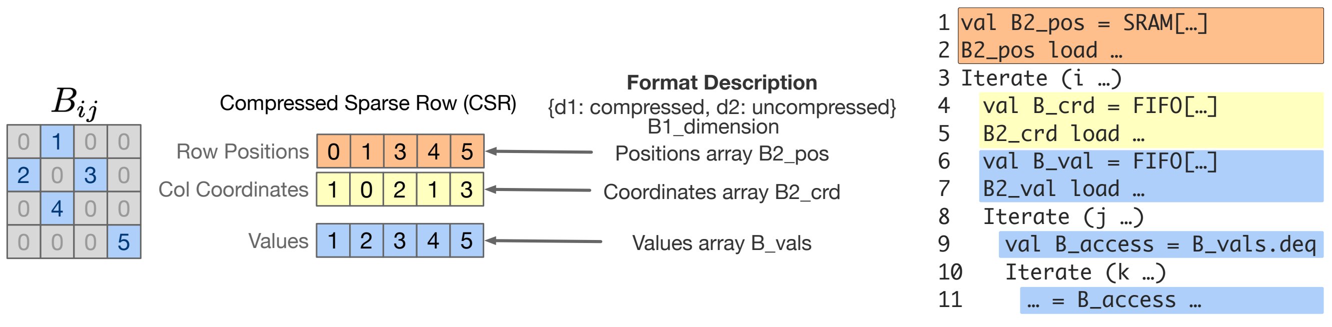
6.1 Tensor Data Structures

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 [6]. A tensor has multiple coordinate levels and a single value level [17, 30]. 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 8). B's CSR data structure is shown in Figure 8b and format language description and sub-arrays are in Figure 8c.

6.2 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,



```
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 8. Example sparse B matrix Figure 8a used in SDDMM with its corresponding data structure Figure 8b and format arrays Figure 8c. Figure 8d shows pseudocode generated by Stardust, where the colors correspond with the arrays in Figure 8b.

scopes, lifetimes, and programming constructs. In Spatial, the physical memory model is a memory hierarchy with (sparse/dense) DRAM → (sparse/dense) SRAM → FIFO → Register (from largest to smallest). The compiler binds sub-arrays to these memories and generates data transfers.

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 [3, 26]. 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.

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.

6.3 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 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 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 8d demonstrates 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 → DRAM vs. `store_stream_vec` for FIFO → DRAM as in Figure 4 line 38).

Putting the analysis all together, consider the SDDMM example in Figure 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 B.

7 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 one generate, for acceleration. During lowering, Stardust compiles the entire computation, including these accelerated sub-computation regions, to parallel patterns as described in Section 8.

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

$c_1 \overset{\text{def}}{=} \text{precompute}(b_i * c_i, \{i\}, \{i\}, a^{\text{on}})$,

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

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 $c_2 \overset{\text{def}}{=} \forall t \in \{b, c\}$ 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 $c_3 \overset{\text{def}}{=} \text{map}(\forall_i a_i^{\text{on}} = b_i^{\text{on}} * c_i^{\text{on}}, \text{backend}, f_{\text{mul}})$, the transformation is

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

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. $S \xrightarrow{\text{accelerate}(S',\text{backend},f,c)} S_{\text{new}}$ is equivalent to $S \xrightarrow{c_1' \; c_2' \; c_3'} S_{new}$, 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,

$$c_1' \overset{\text{def}}{=} \text{precompute}(e, i*, i*, a^{\text{on}})$$

$$c_2' \overset{\text{def}}{=} \text{For all } t \in \text{tensors}(e) \text{ precompute}(t_i, i*, i*, t^{\text{on}})$$

$$c_3' \overset{\text{def}}{=} \text{map}(S'[t^{\text{on}}/t \text{ for all } t \in \text{tensors}(S')], \text{backend}, f, c).$$

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 leverage this command in Section 9 to sweep our evaluated kernels for performance and improved resource utilization.

Our SDDMM example in Figure 6 shows the acceleration of reductions into registers in lines 21–23 and the configuration of parallelization factors in lines 17–18. Some of scheduling commands to target accelerators from Stardust can be found in Table 1 and Table 2.

8 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 8 to generate the final parallel-pattern code for Spatial as follows.

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 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 3, 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

Table 2. 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.
$S \xrightarrow{\text{map}(S,\text{backend},f,c)}$	$f(\text{tensors}(S), V^{\dagger}, c)$ s.t. map(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.
$S \xrightarrow{\text{environment}(\text{var},c)}$	S s.t. var = c

```
// 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 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 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 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 contraction set is defined as $\text{format}(I) = \text{format}(\mathcal{T}_1) \circ \text{format}(\mathcal{T}_2) \circ \ldots \circ \text{format}(\mathcal{T}_n)$. The format of a tensor $\text{format}(\mathcal{T}_n)$ is defined as 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 3 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 computaiton 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 $format(I) = C_{E_2} \cup (C_{B_2} \cap \mathbb{U})$. `lowerIter` is called on format of I, which will first call `lowerIter`$[C_{B_2} \cap \mathbb{U}] \Rightarrow$ `lowerIter`$[C_{B_2} \cap \mathbb{U}]$ 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, any other index calculations, and computation.

Table 3. 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`$[C_1$ **and** $\mathcal{T}_1$ is result$]$	$\Rightarrow$ **emit** $\mathcal{B}_1 = \textsc{genBitvector}(\mathcal{T}_1)$ `lowerIter`$(\mathcal{B}_1)$
	`lowerIter`$[C_1]$	$\Rightarrow$ **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`$(\mathbb{U})$
Comp.	`lowerIter`$[C_1 \cap \mathbb{U}]$	$\Rightarrow$ `lowerIter`(C_1)
	`lowerIter`$[\mathbb{U} \cap C_2]$	$\Rightarrow$ `lowerIter`(C_2)
Co-Iteration	`lowerIter`$[C_1 \circ C_2]$	$\Rightarrow$ **emit** $\mathcal{B}_1 = \textsc{genBitvector}(\mathcal{T}_1)$ **emit** $\mathcal{B}_2 = \textsc{genBitvector}(\mathcal{T}_2)$ `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 4. 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

9 Evaluation

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.

9.1 Methodology

We evaluate Stardust on a benchmark set that is completely new for Capstan. The benchmarks are sparse tensor algebra expressions listed in Table 4 from the literature [14, 17] with Stardust user schedules shown in Table 5. We profile CPU baselines on a 128-thread, four-socket Xeon E7-8890 v3 with

Table 5. User-provided schedules for the kernels in Table 4. Scalar promotion (`sPromote`) inserts a scalar workspace as a macro-scheduling command (instead of lines 21-22 in Figure 6) and `communicate` determines at which iteration pattern the result is communicated back off-chip [35]. `Parallel` is short for parallelize [27] 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 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. 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 %)

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. 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 [26, 39], using an ideal memory model or Ramulator [15] to model four channels of DDR4-2133 or HBM-2E (at 1800 GB/s).

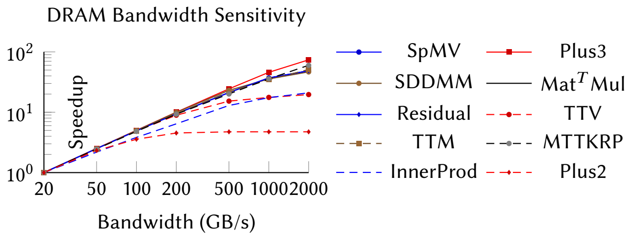


Figure 10. Impact of memory bandwidth on performance.

All evaluations use the datasets shown in Table 8 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 [26]. 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.

We use CSR formats for all sparse 2D matrices and compressed sparse column (CSC) for Mat^{T}Mul. 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 [18] does not support sparse results for their GPU backend.

9.2 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 10 shows that our applications (except Plus2, which is not outer-parallelized)

Table 7. 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

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.

9.3 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 7 provides a comparison of SpMV across all platforms. The Capstan and Plasticine rows from Table 7 are handwritten Spatial SpMV kernels from [25, 26], respectively. All Capstan and Plasticine rows in Table 7 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) 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.

9.4 Tensor Algebra Expression Performance

Performance results for all platforms and applications are shown in Table 7. 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.

Our TACO GPU baseline performance is significantly worse than both the literature [27] and compiled Capstan because TACO does not natively support sparse tensor outputs on the master branch of the system code base [18]. 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.

10 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 [1, 2, 13, 17, 21, 32, 37], GPUs [27, 37], and distributed machines of CPUs and GPUs [36] 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 [10, 11, 23, 28, 29, 31, 38, 40, 42], however, we will focus our discussion on reconfigurable sparse accelerators as they need for compilation. Our system targets Capstan [26] 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 [7] and ExTensor [12] 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 [20] and the idiomatic spatial accelerator compiler of Weng et al. [34] show how to compile high-level control-flow languages to sparse RDAs and CGRAs. The Custard compiler [14], on the other hand, shows how to compile sparse tensor algebra to an abstract machine representing reconfigurable streaming dataflow accelerators. The Mosaic compiler 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.

11 Conclusion

We described the first compiler that enables the end-to-end compilation of sparse tensor algebra from tensor index notation to a sparse reconfigurable dataflow accelerator. We expect Stardust to be the first of many compilers from high-level sparse languages to target these sparse accelerators. We expect its design—of giving users abstract control of memory and computation and having the compiler complete the remaining accelerator mapping information—will influence future compiler designs for domain-specific accelerators.

Table 8. The datasets used to evaluate Stardust.

App		Name	Dimensions	Density
SpMV SDDMM Mat…Mul Residual		bcsstk30 [8]	28924×28924	2.48×10^{-3}
		ckt11752_dc_1 [8]	49702×49702	1.35×10^{-4}
		Trefethen_20000 [8]	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 [33]	$1591 \times 63891 \times 63890$	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}

12 Acknowledgments

We would like to thank Manya Bansal, James Dong, Parthiv Krishna, Rubens Lacouture, Jackson Melchert, Aviral Pandey, Alexander J. Root, Marco Siracusa, Amalee Wilson, Genghan Zhang, and the anonymous reviewers for their feedback on this paper. We would especially like to thank Rohan Yadav and Scott Kovach for help on technical ideas and formalisms in the paper. This work was supported in part by the NSF under grant numbers 1937301, 2028602, CCF-1563078, 1563113, and CCF-2216964. This research was also supported in part by the Stanford Data Analytics for What's Next (DAWN) Affiliate Program and the Semiconductor Research Corporation (SRC) PRISM center. Olivia Hsu was supported in part by an NSF GRFP. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the aforementioned funding agencies.

A Evaluation Datasets

Below is a description of the datasets used to evaluate Stardust in Section 9. 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 [26]. SuiteSparse only contains matrices and cannot be used for higher-order kernels. Therefore, we use the real-world Facebook [33] tensor for most 3D kernels as in prior work [17].

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) [26]. 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 [14]. For Plus2 and InnerProd,

Algorithm 1 Memory Insertion Algorithm

```
// An iteration graph (IterationGraph) denotes which index variable (IndexVar) paths (Path) are taken for the CIN expression [17]
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 6.2
                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 8
        Emit(Scan across bitvectors using iteration algebra and rewrite system in Section 8 )                 ▷ Emit parallel pattern
    // Proceed normally by emitting loop-body code
    LowerWithMemInsert(forallBody)
```

we generate the additional operand by rotating the even coordinates of the last tensor dimension by one.

B Full Memory Analysis Algorithm

We present the memory analysis algorithm as described in Section 6 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 LowerWithMemInsert function definition for a ∀ node since all other nodes behave similarly to the Lower function as in prior work [6, 16, 17, 19, 27] 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.

References

[1] Aart Bik, Penporn Koanantakool, Tatiana Shpeisman, Nicolas Vasilache, Bixia Zheng, and Fredrik Kjolstad. 2022. Compiler Support for Sparse Tensor Computations in MLIR. *ACM Trans. Archit. Code Optim.* 19, 4, Article 50 (sep 2022), 25 pages. https://doi.org/10.1145/3544559

[2] Aart J. C. Bik and Harry A. G. Wijshoff. 1993. Compilation Techniques for Sparse Matrix Computations. In *International Conference on Supercomputing*. ACM, 416–424. https://doi.org/10.1145/165939.166023

[3] Preston Briggs and Linda Torczon. 1993. An efficient representation for sparse sets. *ACM Letters on Programming Languages and Systems (LOPLAS)* 2, 1-4 (1993), 59–69.

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

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

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

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

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

[9] Trevor Gale, Matei Zaharia, Cliff Young, and Erich Elsen. 2020. *Sparse GPU Kernels for Deep Learning*. IEEE Press.

[10] Paul Grigoras, Pavel Burovskiy, Eddie Hung, and Wayne Luk. 2015. Accelerating SpMV on FPGAs by compressing nonzero values. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*. IEEE, 64–67.

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

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

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

[14] Olivia Hsu, Maxwell Strange, Ritvik Sharma, Jaeyeon Won, Kunle Olukotun, Joel S. Emer, Mark A. Horowitz, and Fredrik Kjølstad. 2023. The Sparse Abstract Machine. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 710–726. https://doi.org/10.1145/3582016.3582051

[15] Yoongu Kim, Weikun Yang, and Onur Mutlu. 2016. Ramulator: A Fast and Extensible DRAM Simulator. *IEEE Comput. Archit. Lett.* 15, 1 (Jan. 2016), 45–49. https://doi.org/10.1109/LCA.2015.2414456

[16] Fredrik Kjolstad, Peter Ahrens, Shoaib Kamil, and Saman Amarasinghe. 2019. Tensor Algebra Compilation with Workspaces. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 180–192. https://doi.org/10.1109/CGO.2019.8661185

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

[18] 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. 2023. The Tensor Compiler (TACO). https://github.com/tensor-compiler/taco.

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

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

[21] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. 1997. A relational approach to the compilation of sparse matrix programs. In *Euro-Par Parallel Processing*. Springer, Passau, Germany, 318–327. https://doi.org/10.1007/BFb0002751

[22] Qiaoyi Liu, Jeff Setter, Dillon Huff, Maxwell Strange, Kathleen Feng, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. 2023. Unified Buffer: Compiling Image Processing and Machine Learning Applications to Push-Memory Accelerators. *ACM Trans. Archit. Code Optim.* 20, 2, Article 26 (mar 2023), 26 pages. https://doi.org/10.1145/3572908

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

[24] Michael Pellauer, Yakun Sophia Shao, Jason Clemons, Neal Crago, Kartik Hegde, Rangharajan Venkatesan, Stephen W. Keckler, Christopher W. Fletcher, and Joel Emer. 2019. 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* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 137–151. https://doi.org/10.1145/3297858.3304025

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

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

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

[28] Yi Shan, Tianji Wu, Yu Wang, Bo Wang, Zilong Wang, Ningyi Xu, and Huazhong Yang. 2010. FPGA and GPU implementation of large scale SpMV. In *2010 IEEE 8th Symposium on Application Specific Processors (SASP)*. IEEE, 64–70.

[29] Nitish Srivastava, Hanchen Jin, Jie Liu, David Albonesi, and Zhiru Zhang. 2020. Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product. In *2020 53rd Annual IEEE/ACM*

International Symposium on Microarchitecture (MICRO). IEEE, 766–780.

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

[31] Yaman Umuroglu and Magnus Jahre. 2014. An energy efficient column-major backend for FPGA SpMV accelerators. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*. IEEE, 432–439.

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

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

[34] Jian Weng, Sihao Liu, Dylan Kupsh, and Tony Nowatzki. 2022. Unifying Spatial Accelerator Compilation With Idiomatic and Modular Transformations. *IEEE Micro* 42, 5 (2022), 59–69. https://doi.org/10.1109/MM.2022.3189976

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

[36] Rohan Yadav, Alex Aiken, and Fredrik Kjolstad. 2022. SpDISTAL: Compiling Distributed Sparse Tensor Computations. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis* (Dallas, Texas) *(SC '22)*. IEEE Press, Article 59, 15 pages.

[37] Zihao Ye, Ruihang Lai, Junru Shao, Tianqi Chen, and Luis Ceze. 2023. 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* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 660–678. https://doi.org/10.1145/3582016.3582047

[38] Guowei Zhang, Nithya Attaluri, Joel S Emer, and Daniel Sanchez. 2021. Gamma: leveraging Gustavson's algorithm to accelerate sparse matrix multiplication. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 687–701.

[39] Yaqi Zhang, Alexander Rucker, Matthew Vilim, Raghu Prabhakar, William Hwang, and Kunle Olukotun. 2019. Scalable Interconnects for Reconfigurable Spatial Architectures. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. 615–628.

[40] Yan Zhang, Yasser H Shalabi, Rishabh Jain, Krishna K Nagar, and Jason D Bakos. 2009. FPGA vs. GPU for sparse matrix vector multiply. In *2009 International Conference on Field-Programmable Technology*. IEEE, 255–262.

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

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

Received 2024-05-30; accepted 2024-07-22