# A Verified Approach to High-Performance Tensor-Program Optimization

by

Amanda Liu

B.S. Columbia University in the City of New York (2020)
S.M. Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2026



Authored by: Amanda Liu
Department of Electrical Engineering and Computer Science
May 15, 2026

Certified by: Adam Chlipala
Arthur J. Conner (1888) Professor of Computer Science
Thesis Supervisor

Certified by: Jonathan Ragan-Kelley
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by: Leslie A. Kolodziejski
Professor of Electrical Engineering and Computer Science
Chair, Department Committee on Graduate Students

# THESIS COMMITTEE

## Thesis Supervisors

**Adam Chlipala**
*Arthur J. Conner (1888) Professor of Computer Science*
*Department of Electrical Engineering and Computer Science*

**Jonathan Ragan-Kelley**
*Associate Professor of Electrical Engineering and Computer Science*
*Department of Electrical Engineering and Computer Science*

## Thesis Reader

**Saman Amarasinghe**
*Thomas and Gerd Perkins Professor of Electrical Engineering and Computer Science*
*Department of Electrical Engineering and Computer Science*

# A Verified Approach to High-Performance Tensor-Program Optimization

by

Amanda Liu

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 2026 in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

## ABSTRACT

Tensor programs underlie a broad range of performance-critical workloads in scientific computing, image processing, and machine learning. While these programs are often written as concise mathematical computations over multidimensional arrays, their efficient realization depends on sophisticated transformations governing iteration structure, memory organization, and data layout.

A central insight of this work is that high-performance tensor optimization depends on the ability to manipulate the relationship between compute and storage order. Efficient implementations require control not only over what computation is performed but also over when values are produced, how they are organized in memory, and how these choices interact to determine performance. These transformations are essential to performance, but in existing tensor systems their correctness is typically justified through testing and manual reasoning. This dissertation develops a formal framework in which these transformations are represented explicitly and reasoned about within a common semantic setting, enabling scheduling, layout, and representation changes to be expressed and verified in a uniform way.

This dissertation develops this idea through four technical components. First, it introduces ATL, a functional tensor language that represents tensor computations as algebraic expressions with explicit control over compute and storage structure. This representation supports scheduling as a process of verified, source-to-source program transformations. Second, it develops a mechanically verified lowering procedure that translates ATL programs into imperative array code and formally proves semantic correspondence between source programs and generated loop nests. Third, it formalizes the CUDA template (CuTe) layout abstraction and its associated layout algebra, providing a semantic account of tensor-layout combinators used in GPU programming and establishing correctness properties of their composition. Fourth, it extends this framework to sparse tensor computation by introducing a formal abstraction for sparse formats and deriving correctness-preserving transformations from dense tensor programs to compressed sparse implementations.

The resulting framework provides a single formal perspective from which we can treat scheduling, lowering, layout transformation, and sparse representation within the same reasoning discipline. Across these settings, the dissertation shows that the core transformations used in high-performance tensor systems admit a common algebraic treatment, enabling formal guarantees for optimized tensor programs while retaining the flexibility required for efficient execution.

Thesis supervisor: Adam Chlipala
Title: Arthur J. Conner (1888) Professor of Computer Science

Thesis supervisor: Jonathan Ragan-Kelley
Title: Associate Professor of Electrical Engineering and Computer Science

# Acknowledgments

I would like to thank my advisors, Professors Adam Chlipala and Jonathan Ragan-Kelley.

Adam has an extraordinarily sharp sense of discernment that applies to all levels of the stack: everything from keeping research discussions pointed at the right high-level questions all the way down to catching missing hyphens. His judgment, patience, and rigor have shaped nearly every part of this dissertation. I suspect that a nontrivial fraction of its technical clarity is his doing, and at least a third of its hyphens certainly are.

Jonathan's range as a researcher is so broad that if you threw two darts into his lab and hit two students, there is a decent chance they couldn't explain each other's work. Jonathan, however, has always been the connective tissue: the shared context, the source of synthesis, and the person somehow able to engage deeply with all of it at once. As his student, I have been continually grateful for the generosity of his attention. No problem is too small, no dive too deep, and no half-formed idea too rough to earn serious engagement in a meeting. He is also an excellent sport when being pranked.

More than being grateful to have had them each as an advisor, I am grateful to have had them both. Their strengths are distinct, complementary, and together far greater than the sum of their parts, though the parts themselves are already exceptional.

I also thank Professor Saman Amarasinghe for his presence and engagement throughout my PhD, and for his generosity with both his time and his thoughtful feedback on this dissertation. Saman also played a major role in encouraging my involvement in the service and community-building aspects of the programming languages field, including not only chairing but helping found the MIT Programming Languages Review. Beyond his technical influence, he has also consistently brought wit, warmth, and kindness to our interactions.

I would also like to thank Professor Gilbert Bernstein, my earliest and longest-standing collaborator on the work in this dissertation. The original conception of ATL was his, and I learned an enormous amount from working with him during the formative early years of my PhD.

I am grateful to Shoaib Kamil, who mentored me during my internship at Adobe, where we laid the groundwork for the verified-sparse-formats work in this dissertation. Shoaib has a remarkable ability to make difficult systems problems feel crisp and tractable, and his practical instincts shaped the direction of that work in lasting ways.

I would also like to thank Cris Cecka and Michael Garland at NVIDIA Research, with whom I worked during a summer internship that led to the verified-layout work in this dissertation. Cris worked closely with me throughout that project, and his deep expertise in both the mechanics of CuTe layouts and the structure of their algebra was invaluable to the formalization we developed.

I would like to thank Professor Jonathan Aldrich at Carnegie Mellon University, who was my advisor during a summer Research Experience for Undergraduates (REU) program. He was my first research advisor in computer science. Before working with Jonathan, I had never built a compiler, written a functional program, or even heard the term "lambda calculus" before. He introduced me not only to these ideas but to the kind of research community in which I wanted to develop my career. The programming-languages community at Carnegie Mellon set a standard for intellectual generosity and lab culture that has shaped what I value in research communities to this day.

I would also like to thank Professors Stephen Edwards and Ronghui Gu at Columbia University, my undergraduate research advisors, who received the full force of my enthusiasm, inexperience, and momentum coming back from Carnegie Mellon University. I was still extraordinarily green. From them I learned an appreciation for strong type systems, functional programming, and formal verification that became the foundation of my research taste and runs throughout this dissertation.

I am grateful to my labmates AJ, David, Karima, Kartik, Manya, Rahul, Tian, and Yuka for being the walls on which I bounced half-formed ideas, for patiently ELI5-ing technical details to me when I wandered into unfamiliar corners of the field, and for the endless shenanigans without which this dissertation might have been finished a year earlier (kidding), but with considerably less joy and laughter.

Finally, I would like to thank my family and friends for their constant love and support. In particular, I would like to thank my friend Hilary, whose contribution of Dramamine powered the writing of the final chapters of this dissertation on the Boston–New York Greyhound.

The work included in this dissertation includes work published in papers at POPL 2022 [50], at PLDI 2024 [49], and at PLDI 2026 [51], all of which are licensed by the Association for Computing Machinery (ACM). This thesis work was supported in part by the National Science Foundation (NSF) Graduate Research Fellowship under Grant No.1745302 and separately the NSF under Grants No.CCF-2313023 and No.CCF-2217064. It was also supported by the Defense Advanced Research Projects Agency (DARPA) under the PAPPA (agreement HR00112090017) and RTML (contract FA8650-20-2-7006) programs as well as the contract HR001125CE038 (MOCHA program). Parts of the work included in this dissertation was supported by NVIDIA and Adobe Research over the courses of summer internships.

# Contents

# List of Figures

# Chapter 1

# Introduction

Modern high-performance computing is increasingly driven by tensor programs: computations over multidimensional arrays that arise in domains ranging from scientific computing and image processing to machine learning and data analysis. These programs form the computational backbone of many performance-critical systems. While the mathematical description of a tensor computation is often straightforward, achieving efficient implementations requires careful optimization of low-level program concerns to exploit locality, parallelism, and hardware capabilities.

High-level, functional programs admit a large space of semantically equivalent implementations. A single tensor computation may be realized through many different loop organizations, data layouts, and execution strategies. These implementations may differ dramatically in performance despite computing the same mathematical result. These performance-critical execution details are usually left up to to the discretion compiler, which fixes an execution strategy as it lowers the source program to a lower-level target, optimizing along the way. However, the optimal execution strategy and set of optimizations to implement is often program-, data-, and hardware-dependent. For tensor programs, efficient scheduling must decide when each computation runs relative to the others, at what granularity, and when and where their resulting values are written and read. As a result, the scheduling space for tensor programs is extremely large and a black-box, optimizing compiler can only make static, monolithic choices when lowering programs and often is not able to generate the optimal code.

Modern high-performance, tensor compiler frameworks address this complexity by separating the specification of a computation called the *algorithm* from its optimizations and specific computation strategy called the *schedule*. This separation enables systematic exploration of optimization choices and improves programmability. Systems such as Halide [60], TVM [13], and MLIR [45] allow programmers to describe what computation should be performed independently from how it should be executed. These are user-schedulable languages as users have control over the schedules chosen for input algorithms.

However, this approach introduces a fundamental challenge: correctness. As optimization frameworks become more expressive, the transformations they support become more complex. Transformations such as fusion, tiling, storage reorganization, layout changes, and sparse format transformations fundamentally restructure programs. Ensuring that these transformations preserve program semantics is difficult, and current systems typically rely on testing or

informal reasoning rather than formal guarantees. This dissertation investigates how formal methods can be used to address this gap. In particular, by framing program optimization as a verified, proof-producing procedure using source-to-source rewrites on a high-level object language allows us to frame all of these optimizations as manipulations of relative compute and storage order that preserve functional equivalence. We explore how tensor program optimizations can be expressed as algebraic transformations over index mappings, enabling formal reasoning about correctness while still supporting high-performance implementations.

## 1.1 The Challenge of Verified Tensor Optimization

Verification of tensor optimizations presents several unique challenges.

First, tensor optimizations span multiple abstraction levels. High-level scheduling transformations restructure iteration spaces. Lowering transformations change storage organization and indexing behavior. Layout transformations determine how tensors are mapped to memory and parallel hardware resources. Sparse optimizations change how data itself is represented. Each of these transformations must preserve program semantics while operating on different representations.

Second, tensor optimizations are typically expressed as imperative program transformations. Traditional compiler optimizations manipulate loop nests and index expressions directly. While effective, this representation makes correctness reasoning difficult because the relationship between original and optimized programs becomes implicit.

Third, many tensor optimizations rely fundamentally on complex index manipulations. Loop transformations, layout transformations, and sparse format conversions all modify how indices map to data. Yet these transformations are rarely expressed explicitly as mathematical objects. Instead, they appear as low-level arithmetic manipulations embedded in imperative code.

These challenges suggest that verification of tensor optimizations requires a different perspective. Rather than reasoning about transformations purely at the level of programs, we can instead reason about the index mappings that these optimizations perform.

## 1.2 Index Transformations as a Unifying Perspective

This dissertation is based on the observation that many tensor optimizations can be understood as algebraic transformations of index spaces.

Scheduling transformations such as fusion and tiling can be viewed as transformations of iteration spaces. Lowering transformations can be viewed as transformations of storage mappings. Layout transformations can be viewed as mappings from logical tensor coordinates to physical memory locations. Sparse formats can be viewed as structured transformations between coordinate and storage representations.

Despite appearing in different contexts, these transformations share a common structure: they manipulate mappings between index spaces.

This observation suggests a unifying approach to verified tensor optimization. Rather than reasoning about tensor programs directly, we can reason about the index mappings that

underlie their execution. By representing these mappings explicitly and reasoning about their algebraic properties, we can develop verification techniques that apply across multiple layers of tensor compilation.

This dissertation explores the following central thesis:

> Tensor-program optimizations can be expressed as compositional algebraic transformations of index mappings, and this structure enables practical verification across multiple stages of the tensor-compilation stack.

## 1.3 Overview of the Approach

I developed this idea through ATL, a functional tensor language designed to express tensor programs and their optimizations as algebraic expressions. ATL represents tensor computations in a way that makes control over indexing structure explicit through principled manipulation of relative compute and storage order, enabling reasoning about program transformations as algebraic rewrites.

A central design goal of ATL is to expose index structure rather than hiding it inside imperative code. Tensor programs are expressed as compositions of generation, reduction, and reshape operators that explicitly describe how index spaces are constructed and transformed. This representation allows scheduling transformations to be expressed as source-to-source rewrites.

Building on this foundation, I develop a verified lowering algorithm that translates ATL programs into imperative implementations. This work introduces reindexing operators that describe how reshape transformations modify index mappings. By reasoning about these transformations formally, we establish correctness of the lowering process.

We further demonstrate that this algebraic perspective extends beyond scheduling and lowering. In particular, we show that the layout algebra used in CUDA tensor libraries can be understood as algebraic transformations of index mappings similar to those used in ATL. This formalization allows verification techniques developed for ATL to be applied to GPU layout reasoning.

Finally, we extend this perspective to sparse tensor computation. Sparse formats can be viewed as algebraic structures describing transformations between coordinate spaces and storage representations. By treating sparse formats as compositional objects, we develop a verified framework for sparse tensor optimization.

Across these contributions, a common pattern emerges: tensor optimizations across multiple abstraction levels can be understood as algebraic transformations of index mappings. This perspective provides a foundation for verified tensor compilation.

## 1.4 Key Technical Ideas

Several key ideas enable the approach developed in this dissertation.

First, we represent tensor programs in a functional style that exposes index structure explicitly. This representation allows scheduling transformations to be expressed as algebraic rewrites rather than imperative transformations.

Second, we introduce reshape operators that explicitly represent index transformations. These operators allow us to reason about how transformations change iteration and storage structure while preserving semantics.

Third, we develop reasoning principles for index transformations based on well-formedness and soundness properties. In particular, pad-type reasoning ensures that reshaping transformations preserve correctness even when intermediate storage structures change.

Fourth, we show that layout transformations and sparse format transformations can be understood as algebraic structures similar to reshape transformations. This approach allows techniques developed for functional tensor programs to be applied to GPU layouts and sparse representations.

Together, these ideas demonstrate that algebraic reasoning about index transformations provides a unified framework for verified tensor optimization.

## 1.5 Summary of Contributions

This dissertation develops a unified approach to verified tensor program optimization based on algebraic reasoning about index transformations. The main contributions are:

- **A functional representation of tensor programs that exposes index structure.** I introduce ATL, a functional tensor language that represents tensor programs as algebraic expressions over index spaces through language constructs referred to as reshape operators. This representation enables tensor optimizations to be expressed and verified as functional equivalence-preserving, source-to-source rewrites rather than imperative loop transformations.

- **A verified framework for scheduling transformations.** I develop a rewrite-based framework for expressing scheduling transformations such as fusion, tiling, and loop restructuring by applying these verified scheduling rewrites.

- **A verified lowering algorithm for tensor programs.** I develop a compiler that translates ATL programs into imperative implementations in low-level C code. This work introduces reindexing operators and a pad-type inference system that allow reshape-based transformations to be verified.

- **Formal verification of tensor layout algebra for GPU programs.** I formalize the layout abstraction used in CUDA tensor programming and prove correctness properties of a subset of layout algebra operations. This formalization demonstrates that similar soundness conditions used for reasoning about ATL programs can also be applied to layout transformations.

- **A verified abstraction for sparse tensor formats.** I extend ATL to sparse tensor computation and develop a compositional format algebra that enables verified reasoning about introducing sparse data representations into tensor programs.

- **Evidence that verification and performance are compatible.** Through empirical evaluation, I demonstrate that verified tensor optimizations can produce implementations with performance comparable to existing tensor-optimization systems.

## 1.6 Broader Perspective

This work suggests that formally reasoning about optimizations when compiling high-performance tensor programs can reduce to treating program transformations, layout transformations, and data representations as instances of a common algebraic idea. By reasoning about these transformations through their effects on index mappings and dual manipulation of compute and storage order to maintain functional equivalence, it becomes possible to develop highly extensible, verified optimization frameworks spanning multiple layers of the tensor software stack.

More broadly, this thesis demonstrates that formal verification can play a practical role in high-performance computing. Rather than limiting verification to small kernels or simplified models, this work shows that verification techniques can be applied to realistic tensor-optimization frameworks while preserving competitive performance. As tensor computation continues to grow in importance across scientific computing and machine learning, approaches that combine high performance with strong correctness guarantees may become increasingly important for building reliable computational infrastructure.

# Chapter 2

# Background

## 2.1 Tensor Programs and Optimization

In high-performance tensor-computing domains such as image-processing and machine-learning kernels, programs often take the form of complex mathematical pipelines. Efficiently executing programs in these domains often reduces to optimizing computations on arrays. Often a single natural algorithm over multidimensional arrays may have a bewildering variety of different code realizations. This optimization process requires careful management of crucial performance factors such as computational intensity and data locality. As a result, a common problem in functional array languages is that they conflate compute and storage order, eliminating a large family of optimizations from the set of programs these languages can describe, including but not limited to common, useful techniques like tiling. While this approach has been effective, conflating the optimization and lowering processes makes it very difficult to implement new directives, in addition to complicating any verification process one might try to apply to these systems. It is invaluable to be able to manipulate computation order directly through loop-nest arrangements and storage patterns indicated by storage-access indices. However, the low-level detail required in specifying these scheduling decisions is generally at odds with the high-level, functional representations of algorithms.

## 2.2 User-Schedulable Languages

Modern tensor compilation systems address this challenge through *user scheduling*. In these systems, programmers describe computations at a high level and separately specify scheduling directives that determine how the computation is executed.

Languages such as Halide for graphics [60] and TVM for machine learning [13] exemplify this approach. These systems allow programmers to explore optimization strategies without modifying the underlying algorithm.

Halide is a particularly relevant comparison point due to its widespread adoption. It is used in production systems such as Adobe Photoshop and the YouTube video pipeline, and it is widely regarded as a state-of-the-art system for high-performance array processing.

Matrix multiplication is a common linear-algebra operation that is ubiquitous in tensor-programming domains and primarily one of the building blocks of machine-learning and

image-processing subroutines. Matrix multiplication involves taking two matrices of dimension $m \times n$ and $n \times k$ and producing an output matrix of dimension $m \times k$. The output matrix is constructed by computing for each cell the dot product of the corresponding column in one matrix with the corresponding row in the second matrix. For such mathematical operations, as is the case with most tensor-programming computations, it is very natural to express this term as a pure, mathematical expression. Consider the following Halide program written below to express matrix-matrix multiplication. The Halide program declares the output buffer and several variables as well as one over which to perform reduction (or summation), but note that here that on the final line matrix multiplication is written as a pure, declarative program that does not yet specify how the computation is to be executed.

```
Func C("C");
Var i("i"), j("j");
C(i, j) = 0.f;
RDom k(0, A.dim(1).extent(), "k");
C(i, j) += A(i, k) * B(k, j);
```

As mentioned before, there are a vast number of execution strategies and imperative code realizations that are possible and equivalent to compute one given high-level algorithm. As a user-schedulable language, Halide allows programmers to specify the execution strategy via series of commands called scheduling directives that guide the lowering strategy of program algorithms. The following is one possible schedule for the matrix-multiplication algorithm and the associated C code that it would generate to execute this program.

```
C.compute_root();
```

This is the most standard execution strategy that can be scheduled in Halide and results in the following C code.

```
for (int j = 0; j < C.extent(1); j++) {
  for (int i = 0; i < C.extent(0); i++) {
    C(i, j) = 0.f;
    for (int k = 0; k < A.dim(1).extent(); k++) {
      C(i, j) += A(i, k) * B(k, j);
    }
  }
}
```

However, this example is only one such possible schedule for matrix multiplication. In fact, we can achieve a vastly more efficient schedule with a small change to the schedule shown below. This schedule effectively reorders the loops so that the program is no longer striding through the matrix as it processes the dot products so we are traversing the outer dimension of the matrix before the inner dimension. This kind of schedule improves locality and cache utilization of the computation and improves performance.

```
C.compute_root();
C.reorder(j, i);
```

This results in code producing a very similar loop-nest to the previous schedule. However, note that the top two for-loops have been swapped.

```
for (int i = 0; i < C.extent(0); i++) {
  for (int j = 0; j < C.extent(1); j++) {
    C(i, j) = 0.f;
    for (int k = 0; k < A.dim(1).extent(); k++) {
      C(i, j) += A(i, k) * B(k, j);
    }
  }
}
```

We can further improve the efficiency of the execution strategy for this matrix multiplication by using a new schedule that implements a common array-programming optimization technique known as tiling. By tiling a computation, we take a loop and split it into two nested loops so rather than computing the output in full by streaming and iterating over the full iterating space of that loop at once, we split it up into smaller chunks to be processed together at once. Not only can tiling improve performance by increasing locality of computation and cache utilization by making sure computation is being done over input chunks that fit in a cache line and avoiding reads from memory, it also can increase opportunities for parallelization. Below is the Halide schedule to produce a tiled matrix-multiplication program and the resulting generated C code.

```
Var io("io"), jo("jo"), ii("ii"), ji("ji");
C.compute_root();
C.split(i, io, ii, 8)
 .split(j, jo, ji, 8)
 .reorder(ii, ji, io, jo);
```

```
for (int jo = 0; jo < ceil_div(C.extent(1), 8); jo++) {
  for (int io = 0; io < ceil_div(C.extent(0), 8); io++) {
    for (int ji = 0; ji < 8; ji++) {
      for (int ii = 0; ii < 8; ii++) {
        int j = jo * 8 + ji;
        int i = io * 8 + ii;
        C(i, j) = 0.f;
        for (int k = 0; k < A.dim(1).extent(); k++) {
          C(i, j) += A(i, k) * B(k, j);
        }
      }
    }
  }
}
```

## 2.3 Source-to-Source Rewrite-Based Scheduling

However, it is worth pointing out that the intermediate results of Halide scheduling directives can often no longer be represented in the source language. The usage of scheduling directives as opaque decorators to direct the process of lowering conflates the optimization and lowering process, thereby making it difficult to iterate on. Not only this, but this approach makes the scheduling process difficult to reason about and prove correctness of in a formal way.

One popular alternative approach is to phrase optimizations as transformations within a single source or intermediate language; the most common kind of transformation is a *rewrite rule*, a quantified term equality. Phrasing optimizations as source-to-source transformations via quantified relations of equality is easiest when dealing with a pure, high-level object language rather than low-level, stateful code. In particular, tensor programs are particularly amenable to be phrased as such programs and therefore a prime candidate for formal reasoning about functional equivalence as the guiding principle of correctness when scheduling optimizations, since optimizing algorithm involves exploring the tradeoff space of functionally equivalent schedules.

For example, the ATL program below shows a simple matrix-matrix multiplication and the corresponding generated C code. It represents the product between two matrices $m_1$ and $m_2$:

$$\boxplus_{i=0}^{N} \boxplus_{j=0}^{M} \sum_{k=0}^{K} m_1[i;k] * m_2[k;j]$$

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    for (int k = 0; k < K; k++)
      out[i * M + j] +=
        m1[i * K + k] * m2[k * M + j];
```

The $\boxplus$ construct in this program represents tensor generation ("gen"), which constructs a tensor by describing each element as a function of its indices. This tensor generation produces a one-dimensional tensor of length $N$. Similarly, the tensor summation construct, sums the values of the inner body function evaluated as a function of its iterating index. Therefore this program produces a tensor, where each entry in the $i$-th row and $j$-th column is the dot product between the $i$-th row of $m_1$ and the $j$-th column of $m_2$.

Note that the semantics of generation and summation do not specify the order in which these elements are manifested or reduced into the output tensor. Instead, the ordering enforced by the loop-nests in the generated C code is an arbitrary decision made by the compiler in the lowering process. While this ordering is a fairly straightline interpretation of the source constructs (iterating in increasing order between the bounds specified in the order in which the binding variables were introduced), it is still an arbitrary decision made for the sake of implementing the compiler. Here in the lowering algorithm we allow generation and summation constructs to define the compute order of the low-level scheduled program in this specific way.

To optimize this program in the same way we optimized the Halide version, we would like to reorder its iteration structure to improve locality. In ATL, we do this by transforming the source program directly using an equivalence-preserving rewrite. The specific theorem we

will use here is shown below.

$$\boxplus_{i=0}^{N} \sum_{j=0}^{M} e = \sum_{j=0}^{M} \boxplus_{i=0}^{N} e$$

By using this rewrite we arrive at the following ATL matrix-multiplication program that is proven to be functionally equivalent to the original. If we follow the same lowering rule used before abiding by the convention of aligning the compute order generated in the form of for-loops with the generation and summation constructs, we arrive at the following C program as well. You will note that the compiled version of this derived matrix-multiplpication program is the same as the generated code from the loop-reordered Halide schedule. However, deriving and compiling this code separately allows us not only to inspect the optimized, high-level representation and iterate on that further making the optimization process less opaque, but it also allows us to more easily provide formal guarantees of correctness by separating out the concerns of soundness of the optimizations as proofs of functional equivalence from the soundness of lowering as a simpler, straightline compiler verification effort.

$$\boxplus_{i=0}^{N} \boxplus_{j=0}^{M} \sum_{k=0}^{K} m_1[i;k] * m_2[k;j]$$

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    for (int k = 0; k < K; k++)
      out[i * M + j] +=
        m1[i * K + k] * m2[k * M + j];
```

However, it is not immediately evident how to schedule an optimized tiled version of this algorithm through the source-to-source rewrite approach shown here. To do so, we would need to identify the target optimized code written in this high-level source that would need to be compiled down to achieve the tiled loop-nest program generated earlier using Halide scheduling directives. We can begin to try and do so by writing the following ATL program using tensor summations and generations to emulate the for-loop nest structure of the target program. To do so we take the original iterating structures in the ATL program and split them into two, with the new inner iterating structure processing the computation and striding by a factor of some tiling factor $C$. We then also reorder these loops such that looping structures that are iterating over the tiles going up to a bound of $C$ are moved inwards towards the bottom of the loop-nest. Finally, we replace the indices that were bound by the split loops with the equivalent index expression involving the indices bound by the newly split loops.

$$\boxplus_{i_o=0}^{N/C} \boxplus_{j_o=0}^{M/C} \boxplus_{i_i=0}^{C} \boxplus_{j_i=0}^{C} \sum_{k=0}^{K} m_1[i_o * C + i_i; k] * m_2[k; j_o * C + j_i]$$

While this ATL program appears to correspond to a tiled computation, lowering reveals a problem. The generated loop structure is correct, but the storage layout is not. The indexing expression used to store results does not match the expected tiled layout. This disparity demonstrates that transforming computation structure alone is insufficient. If we adhere to the same lowering convention we established before, this ATL program would generate the following imperative code.

```
for (int io = 0; io < N/C; io++)
  for (int jo = 0; jo < M/C; jo++)
    for (int ii = 0; ii < C; ii++)
      for (int ji = 0; ji < C; ji++)
        for (int k = 0; k < K; k++)
          out[io * M * C + jo * C * C + ii * C + ji] +=
            m1[(io * C + ii) * K + k] * m2[k * M + jo * C + ji];
```

Although the loop-nest order in the generated code properly executes the tiled compute order required by this optimization, the index expression generated from the lowering at the bottom of the loop-nest designating where computed values are to be stored into the output buffer `out` is not the same as the tiled program and is not storing in the order one would expect of matrix multiplication. In other words, the storage order of this computation is not what we wanted since through the process of lowering we have inherently tied the storage order to the compute order of the program. It is important to note that this is not an artficial issue due to the choice of lowering rule we made here for the purposes of this demonstration. The point is that any lowering algorithm that takes a similar high-level, declarative source has to make *some* kind of decision over how to resolve compute and storage order for all programs it lowers, thereby limiting its expressivity. And although we chose to anchor compute order to the source-level summation and generation constructs so we were able to use rewrites to transform the ATL program to manipulate the compute order to change the generated for-loop ordering, there is no source-level analogue to be manipulated to give us control over the storage index expression in the generated code.

## 2.4 Separating Compute and Storage Order

The failure of the tiled example highlights a key limitation: the representation allows manipulation of computation order but not storage order. Any lowering algorithm must choose a storage layout, and this choice constrains the set of expressible optimizations. Without explicit control over storage structure, it is not possible to express transformations that require coordinated changes to both computation and storage. To express and formally verify such transformations, we must be able to manipulate compute and storage order in concert.

Doing so requires source-level constructs that explicitly represent storage structure. By designing a high-level, functional tensor language to include such constructs, we can express both computation and storage transformations within a unified algebraic framework, enabling verified source-to-source optimization. We can separate compute and storage order while maintaining functional equivalence as our specification of correctness for scheduling optimizations.

# Chapter 3

# The ATL Language

ATL is a high-level functional tensor language designed to represent tensor programs as pure, algebraic expressions [5]. At its core, ATL represents tensor computations as algebraic compositions of tensor generation, reduction, tensor access, and scalar arithmetic operations. This yields a high-level representation in which the structure of a computation can be made explicit and scheduling transformations can be expressed as source-to-source rewrites. In this section, we introduce the core of the ATL language and the semantics of its tensor constructs, which correspond to how computation structure and compute order are represented in ATL.

## 3.1 A Simple Pipeline Example

In this section we begin the description and formalization of the core language constructs of ATL by analyzing a minimal, interesting program. Consider the simple two-stage pipeline written in ATL below.

$$\text{let } buf := \square_{j=0}^{n} f(j) \text{ in } \square_{i=0}^{n} [\ 0 \le i-1\ ] \cdot buf[i-1] \oplus buf[i]$$

This program encodes a pipeline in which the first stage is some computation represented by the function $f(x)$, and the second stage is computed as a sum of $f(x-1)$ and $f(x)$. In ATL, each function is represented as a tensor generated by the tensor-comprehension operator $\square$. This operator realizes a tensor with inputs computed from the expression in its body as a function of the index over a given range. Separate stages in a pipeline are represented as sequential let-bindings in which each stage is realized, and every reference to an upstream stage becomes an access into that buffer.

Given the loop bounds in this program, $buf[i-1]$ will be out-of-bounds for some iterations in the second stage. Specifically, in the first iteration of the generation when $i = 0$, there is a resulting attempt to access $buf$ at $-1$. As a fix, we logically guard the expression with the condition $0 \le i-1$, a predicate that ensures the access is valid. To do so, we use the indicator function from Ken Iverson's APL that returns 1 if its condition is true and 0 otherwise [26]. Consequently, for iterations where this access is valid, this guard acts as the identity function; and for the iteration where the access is invalid, the additive identity is returned instead.

$$
\begin{array}{rrcl}
\text{Variable} & x & \in & \mathbb{S} \\
\text{Index Expression} & I & ::= & x \mid i \mid I + I \mid I - I \mid I \times I \mid \lceil I \,/\, I \rceil \mid I \,/\, I \mid I \,\%\, I \\
\text{Predicate} & p & ::= & \mathsf{true} \mid \mathsf{false} \mid I = I \mid I < I \mid I \leq I \mid p \wedge p \\
\text{Expression} & e & ::= & x \mid [\ p\ ] \cdot e \mid \mathsf{let}\ x := e\ \mathsf{in}\ e \mid \boxplus_{x=I}^{I} e \mid \sum_{x=I}^{I} e \mid e \oplus e \mid e[I] \mid e * e \mid e/e
\end{array}
$$

Figure 3.1: Core ATL syntax

The sum of two values is symbolized by the general $\oplus$ operator. Notice that in this program, while the function types are well-defined, the dimensionality of the data that is being computed is unspecified and therefore polymorphic. In particular, it can be inferred upon inspection that $f$ has type $\mathbb{Z} \to X$, so the overall expression has type $X$, where $X$ is effectively a type-unification variable. In ATL, the possible instantiations of $X$ are limited to a class of scalar or tensor types that we elaborate on next. Therefore, for groups of operators that have type-specific implementations but maintain the same algebraic properties, we introduce single polymorphic operators, as is the case with addition and $\oplus$.

## 3.2 Syntax and Semantics

In this section we present the syntax of the core ATL language in Figure 3.1 presents, and its denotational semantics in Figure 3.2. For the scheduling stage of our framework, ATL is implemented as a shallow embedding in the Rocq proof assistant: each syntactic construct is represented as a Gallina definition in Rocq's dependently typed, pure functional language. As a result, the semantics of the core ATL constructs are presented directly in terms of their functional interpretation in Gallina, from which the denotational semantics are naturally inherited. This shallow embedding also allows us to leverage Rocq's existing rewriting infrastructure and interactive proof environment when applying scheduling transformations, requiring only a lightweight layer of domain-specific rewrite lemmas and automation on top of the underlying proof assistant.

The array index semantics explained in the final clause of Figure 3.2, treats out-of-bounds accesses as returning default values, relying on later static analysis to confirm that derived programs never actually make out-of-bounds accesses. This definition is important that default values are properly typed, so we compute them based on those arrays' first elements.

## 3.3 Types

In general, ATL programs are polymorphic in the dimensionality of the tensors they operate over. Unless a program uses scalar operations such as addition or multiplication, its computation is independent of the concrete tensor dimension. In other words, a program expressed in ATL is agnostic of the absolute dimensionality of the data it computes over unless it uses a type-specific operator. As a result, scheduling rewrites in ATL can be defined once and applied uniformly across all dimensional instantiations of a given algorithm. The

$$
\begin{aligned}
[\![\mathbb{R}]\!] &= \texttt{R} \\
[\![[\tau]]\!] &= \texttt{list}\ [\![\tau]\!] \\
[\![I_1 + I_2]\!] &= [\![I_1]\!]\ \texttt{+}\ [\![I_2]\!] \\
[\![|e|]\!] &= \texttt{length}\ [\![e]\!] \\
[\![[\ p\ ] \cdot e]\!] &= \texttt{(if}\ [\![p]\!]\ \texttt{then 1 else 0) *}\ [\![e]\!] \\
&\quad \mid [\![[\ p\ ] \cdot e[0]]\!] :: [\![[\ p\ ] \cdot e[1]]\!] :: \ldots :: [\![[\ p\ ] \cdot e[|e|-1]]\!] :: [] \\
[\![\mathsf{let}\ x\ := e_1\ \mathsf{in}\ e_2]\!] &= \texttt{let}\ [\![x]\!]\ \texttt{=}\ [\![e_1]\!]\ \texttt{in}\ [\![e_2]\!] \\
\left[\!\!\left[\boxplus_{x=I_1}^{I_2} e\right]\!\!\right] &= [\![e[I_1/x]]\!] :: [\![e[(I_1+1)/x]]\!] :: \ldots :: [\![e[(I_2-1)/x]]\!] :: [] \\
\left[\!\!\left[\sum_{x=I_1}^{I_2} e\right]\!\!\right] &= [\![e[I_1/x] \oplus e[(I_1+1)/x] \oplus \ldots \oplus e[(I_2-1)/x]]\!] \\
[\![e_1 \oplus e_2]\!] &= [\![e_1]\!] + [\![e_2]\!] \\
&\quad \mid [\![e_1[0] \oplus e_2[0]]\!] :: [\![e_1[1] \oplus e_2[1]]\!] :: \ldots :: \\
&\quad e_1[\texttt{max(}[\![|e_1|]\!]\texttt{,}[\![|e_2|]\!]\texttt{) - 1}] \oplus e_2[\texttt{max(}[\![|e_1|]\!]\texttt{,}[\![|e_2|]\!]\texttt{) - 1}] :: [] \\
[\![e[I]]\!] &= \begin{cases} [\![I]\!]\text{th element of } [\![e]\!], & \text{index is in-bounds} \\ [\![[\ \mathsf{false}\ ] \cdot e[0]]\!], & \text{index is out-of-bounds} \end{cases}
\end{aligned}
$$

Figure 3.2: Denotational semantics for the core of the ATL language

concrete dimensionality and dimension sizes of the tensors need only be fixed during code generation, when allocation sizes and index strides must be determined. This separation allows a single ATL algorithm to be scheduled and generate code for an entire family of kernels, each instantiated over tensor operands of different dimensionalities.

The universe of types of ATL expressions can be defined by the following grammar.

$$\tau \quad ::= \quad \mathbb{R} \mid [\tau]$$

This type is defined recursively. To put it simply, this means that any ATL expression is either an element of $\mathbb{R}$, a scalar; or recursively a list of tensor elements of this type.

As a result, the operators of ATL are inherently polymorphic. All instances of this polymorphism for binary addition ($\oplus$), sum reduction ($\sum$), tensor access ([]), and Iverson's bracket or guard ([ ]) can be found in the denotational semantics we present in Figure 3.2, where a given polymorphic operator will be defined as having two semantics separated by a pipe with the first being the semantics for scalars and the latter being for tensor types. In the process of lowering, this abstraction vanishes in both the types and the operator instances, when the absolute dimensionality and sizes of inputs must be given, which instantiates the dimensions in the rest of the pipeline.

## 3.4 Consistency and Shape

A detail worthy of note here is that in this construction, while it is possible to know symbolically the sizes and dimensions of expressions in ATL statically, there is no type-level

$$\frac{x : \mathbb{R}}{x \wr \cdot} \text{ Scalar Consistency}$$

$$\frac{x \wr s \qquad xs \wr (m, s)}{(x :: xs) \wr (m+1, s)} \text{ Tensor Consistency}$$

Figure 3.3: Consistency property

information concerning the size of each dimension. Moreover, there is no type-enforced property of uniformity within tensors. For example, the following is a completely valid program in ATL, syntactically speaking:

$$\underset{i=0}{\overset{n}{\square}} \; \underset{j=0}{\overset{i}{\square}} \; 1$$

This simple program computes a tensor that comprises tensors of various lengths–a complexity that we wish to disallow.

One idiomatic Rocq alternative would be to use dependently typed tensors and encode the tensor dimensionality and the size of each dimension into the type. However, dependently typed functional programming in Rocq can become onerous for end-user programmers. Although dependently-typed encodings can enforce strong static guarantees, they often make otherwise simple programs cumbersome to write and manipulate, since even routine constructions must be expressed through heavily indexed terms and dependent interfaces.

Instead of enforcing these invariants directly in the type system, we capture them as a separate formal well-formedness property describing the shape and internal consistency of ATL tensors. This property is discharged as a proof obligation only when required by the optimization or rewrite under consideration, and in practice can largely be mechanized through proof automation. This design keeps ATL programs lightweight to construct, avoids burdening users with dependent typing, and allows intermediate rewrite steps to temporarily leave the space of well-formed programs so long as consistency is reestablished. In contrast to a fully dependent encoding, these invariants need not be carried explicitly through every intermediate term and proof step, but are instead enforced only at the relevant points in the optimization process.

This property is formalized in Figure 3.3. The consistency property associates each tensor with a shape represented as a nested tuple, where each component corresponds to one dimension of the tensor. Scalars are treated as zero-rank tensors, so their consistency is defined trivially. Since scalars have no extent along any dimension, they are taken to be consistent with the unit shape. A tensor is consistent precisely when it is nonempty, each of its elements is consistent, and all of its elements share the same shape. Accordingly, a tensor is consistent with a shape represented by a pair consisting of the length of its outermost dimension and the common shape of its elements.

We are able to prove a consistency lemma for each language construct, so as to enable a syntax-directed deduction strategy for automated consistency proving and shape deduction.

$$\frac{\forall x.\ 0 \leq x < n \to e[x/i] \ \wr \ s}{(\boxplus_{i=0}^{n}\ e) \ \wr \ (n, s)}\ \text{Gen Consistency}$$

$$\frac{\forall x.\ 0 \leq x < n \to e[x/i] \ \wr \ s}{(\sum_{i=0}^{n}\ e) \ \wr \ (n, s)}\ \text{Sum Consistency}$$

$$\frac{e \ \wr \ s}{([\ p\ ] \cdot e) \ \wr \ s}\ \text{Guard Consistency}$$

$$\frac{e_1 \ \wr \ s \qquad e_2 \ \wr \ s}{(e_1 \oplus e_2) \ \wr \ s}\ \text{Bin Consistency}$$

$$\frac{e \ \wr \ (n, s)}{(e[i]) \ \wr \ s}\ \text{Get Consistency}$$

$$\frac{e_1 \ \wr \ s \qquad \forall x.\ x \ \wr \ s \to \Gamma \vdash e_2[x/i] \ \wr \ s'}{(\mathsf{let}\ i\ := e_1\ \mathsf{in}\ e_2) \ \wr \ s'}\ \text{Let Consistency}$$

Figure 3.4: Syntax-directed consistency inference

Therefore, we can infer the simple pipeline example program and the matrix multiplication to be consistent with the following shapes.

$$\frac{m_1 \ \wr \ (N, K, \cdot) \qquad m_2 \ \wr \ (K, M, \cdot)}{\bigboxplus_{i=0}^{N} \bigboxplus_{j=0}^{M} \sum_{k=0}^{K} m_1[i;k] * m_2[k;j] \ \wr \ (N, M, \cdot)}$$

$$\frac{\forall j, 0 \leq j < n \rightarrow f(j) \ \wr \ s}{\mathsf{let}\ buf \ := \bigboxplus_{j=0}^{n} f(j)\ \mathsf{in}\ \bigboxplus_{i=0}^{n} [\ 0 \leq i-1\ ] \cdot buf[i-1] \oplus buf[i] \ \wr \ (n, s)}$$

Moreover, we can prohibit the expression of the jagged tensor ATL program expressed at the start of this chapter as it does not satisfy the consistency property and can not be infered as such.

# Chapter 4

# Verified Scheduling Via Source-to-Source Rewrites

ATL is a pure, high-level language, which allows scheduling transformations to be expressed directly as equivalence-preserving transformations on source programs. Instead of reasoning about low-level imperative code and establishing correctness through an operational semantics, we can work entirely at the level of functional terms and characterize correctness as ordinary functional equivalence. This approach makes it possible to formulate optimization in ATL as source-level program derivation: a schedule is constructed by applying a sequence of proved rewrites that transform one ATL expression into another with the same meaning. In this chapter, we develop this rewrite framework as an embedded scheduling system in Rocq. Scheduling is expressed as source-to-source transformation, where each optimization step is justified by a theorem and the result of optimization is itself another ATL program—one that encodes a different execution strategy while preserving the semantics of the original.

This approach serves two purposes. First, it gives scheduling a precise semantic foundation. Each optimization step is represented as an explicit transformation on the source program and justified by a proof that it preserves functional equivalence. Second, it makes optimized programs available for inspection, further transformation, and composition in a way that opaque scheduling directives do not. Optimization becomes a process of constructing equivalent programs, rather than annotating one program with instructions for a compiler.

The rewrite framework is implemented directly in Rocq, where ATL is embedded as a shallow domain-specific language. This design allows scheduling transformations to be expressed and verified in the same setting as the language semantics themselves. Rewrite rules are not introduced as trusted compiler axioms but instead proved from first principles as theorems over ATL's denotational semantics. Optimization procedures are then constructed by composing these proved transformations, yielding a scheduling system in which extensibility does not come at the expense of soundness.

A key limitation of a rewrite-based optimization system for functional array programs is that it can only transform execution strategies that are directly evident in the semantics of the source language. They may reorder iteration, fuse producers and consumers, or restructure reductions, but they cannot directly express coordinated changes to storage layout. As the previous chapter showed, many important tensor optimizations require changing both computation order and storage order together. In ATL, this capability is provided by *reshape*

*operators*, which make transformations of index structure explicit in the source language. These operators allow rewrites to express not only how a computation is performed but also how its intermediate and output tensors are organized in memory.

Reshape operators are the central mechanism that distinguish ATL's rewrite system from prior source-to-source approaches. They allow optimizations such as tiling, reordering, fusion, and layout-sensitive transformations to be expressed uniformly as rewrites over a pure functional language. These operators make it possible to recover much of the expressive power associated with scheduling systems such as Halide, while retaining a proof-oriented formulation in which each optimization step is explicitly represented and mechanically justified.

The remainder of this chapter develops this scheduling framework in detail. We begin by introducing the structure of rewrite-based scheduling in ATL and the tactic framework used to construct optimization derivations in Rocq. We then develop the core rewrite principles for scheduling transformations, culminating in reshape operators, which provide explicit control over storage structure in the source language. Finally, we show how these rewrites recover a broad class of practical tensor optimizations and form the basis of the verified scheduling system used throughout the rest of this dissertation.

## 4.1 Overview and Motivating Example

The tensor-computation programs to be optimized in our framework are written in a high-level, functional language with pure, algebraic constructs, called ATL. This notation allows intricate tensor-computation pipelines to be expressed in high-level terms closely resembling mathematics equations. Below is the same two-stage pipeline expressed in ATL presented earlier in Section 3.1.

$$\mathsf{let}\ buf\ := \boxplus_{i=0}^{n} f[i]\ \mathsf{in}\ \boxplus_{i=0}^{n} [\ 0 \le i-1\ ] \cdot buf[i-1] \oplus buf[i]$$

The big-operator $\boxplus$ syntax is for an array generation (comprehension), giving an iteration variable, its bounds, and an expression to evaluate with it for every value of the iteration variable. We will sometimes describe such expressions colloquially as "loops" (and we will see shortly how that connection is made precise through compilation). The colored brackets denote a guard expression, effectively evaluating to zero or one based on the truth of the Boolean expression therein. We use a guard to avoid depending on an out-of-bounds array access, in this case for the first iteration of the second loop. We write $\oplus$ for a type-overloaded notion of addition.

Although this program is fairly simple, most useful and interesting computational pipelines follow a similar form, only at scale: various stages of computation over various input and intermediary bound values using the loop-mimicking constructs of generation and summation, with data-dependent access patterns throughout.

When we presume the input tensor $f$ to be one-dimensional, our compiler lowers the computational expression into C code of the following form:

We implement ATL in Rocq specification language Gallina and therefore inherit its semantics. As a result, scheduling transformations on programs in our framework are phrased as quantified equivalences between small ATL expressions, each one formally verified as a

```
void pipeline(float *f, int n, float *output) {
    alloc buf;
    for (int i = 0; i < n; i++) {
        buf[i] = f[i];
    for (int i = 0; i < n; i++) {
        if (0 <= i-1)
            ..
        output[i] = ..
    }
}
```

theorem. These theorems establish a set of correct and composable scheduling rewrites that can be used to modify smaller expressions in large programs.

The process for using the scheduling theorems and optimizing ATL programs within our framework is conceived and implemented as a constructive proof of equivalence. The constructed value is the optimized schedule derived from the input program, and the machine-checkable proof of equivalence is constructed incrementally and automatically with each rewrite that is applied to transform the program.

We keep this process of proof-constructing scheduling at a high level by providing in our framework a powerful set of tactics to abstract away and automate any of the required low-level proof detail. This way, the proof script for scheduling a program is kept at the level of abstraction of a sequence of distinct rewrites resembling a paper proof of the same equivalence. Moreover, since Rocq is also an interactive theorem prover, the intermediate states of the program in the process of scheduling are visible as the proof progresses, at each step displaying the program that results immediately from the theorem rewritten before it. For example, the following is the proof script for scheduling the simple ATL pipeline above from a distinct two-stage program into a tiled two-stage program–the same optimization performed in the Halide example above.

```
reschedule.     (* generic marker to begin derivation *)
inline let_binding.
rw get_gen.
rw get_gen.
rw flatten_trunc_tile_id around (GEN [ _ < _ ] _) with 8.
inline tile.
rw<- gp_iverson.
rw ll_get.
rw get_gen_some.
rw lbind_helper for (fun x => |[ _ <? n ]| x).
rw ll_gen.
done.           (* generic marker to end derivation *)
```

The main tactic provided in our framework for rewriting programs is the `rw` tactic, which takes the name of a theorem to be used to rewrite the program as well as some optional

arguments to specify a site if ambiguities are present. The `inline` tactic is written to take as an argument some symbol and inline its definition in the program, performing some minor simplifications to clean up as well. This scheduling procedure performed a tiling optimization on the two-stage pipeline. The newly scheduled tile-pipeline program is shown below.

$$\mathsf{trunc}_r\ n \boxplus_{i_o=0}^{\lceil n/8 \rceil} \left( \mathsf{let}\ v := \boxplus_{i_i=0}^{8} [\ i_o * 8 + i_i < n\ ] \cdot (([\ 0 \leq i_o * 8 + i_i - 1\ ] \cdot f[i_o * 8 + i_i - 1]) \oplus f[i_o * 8 + i_i]) \ \mathsf{in} \right.$$
$$\left. \boxplus_{i_i=0}^{8} [\ i_o * 8 + i_i < n\ ] \cdot v[i_i] \right)$$

The C code generated from the tiled ATL program is:

```
void pipeline(float *f, int n, float *output) {
        for (int io = 0; io < (n + 8 -1) / 8; io++) {
                float *v = calloc(sizeof(float), 8);
                for (int ii = 0; ii < 8; i++) {
                        if (0 <= io *8 + ii < n)
                                ..
                        v[ii] = ..
                }
                for (int ii = 0; ii < 8; ii++) {
                        if (io * 8 + ii < n)
                                output[io * 8 + ii] = ..
                }
        }
}
```

Via the newly introduced loop and nested structure, we have achieved a tiled version of the original program. Furthermore, we were able to construct such an optimization through a series of formally verified rewrites on a high-level, algebraic representation of this pipeline.

It is not obvious that all important scheduling optimizations can be performed on terms as high-level as in ATL, but one of our main research contributions is demonstrating an effective interplay between *reshape operators* like $\mathsf{trunc}_r$ as introduced above and the process of compiling to C, such that functional programs signal all important design decisions for nested imperative loops. Interestingly, the reshape operators are defined in terms of more basic operators like $\boxplus$, not in terms of some explicitly imperative semantics as in past work with proved rewrite laws, making it relatively easy to prove the rewrites we need for effective optimization. Let us now see those core primitives spelled out, before turning to basic scheduling rewrites, compilation to C, and how reshape operators interact with both.

## 4.2 The Scheduling-Rewrite Framework

In this section, we detail the construction and utility of our rewrite framework so as to allow high-level user-scheduling of ATL programs through series of algebraic rewrites verified within

Rocq. In order to do so, we introduce some common and useful examples of the rewrites we have proven as theorems in our framework, as well as the lemmas we have proven to provide logical machinery necessary for automated, conditional rewriting under binders and logical contexts.

### 4.2.1 Scheduling Rewrite Theorems

Each scheduling rewrite is formulated as a theorem of functional equivalence between two ATL programs. Rather than being declared as an axiom, it is proven in accordance with the semantics of the language's embedding.

We begin once more by considering the simple two-stage pipeline program.

$$\mathsf{let}\ buf\ := \boxplus_{j=0}^{n} f(j)\ \mathsf{in}\ \boxplus_{i=0}^{n} [\ 0 \le i-1\ ] \cdot buf[i-1] \oplus buf[i]$$

This schedule for computing the two-stage pipeline will first realize $f$ over its full specified domain to be stored into $buf$ before the output stage is able to proceed and compute on $buf$. One possible scheduling transformation would be traditional loop fusion, which allows the two loops apparent in the program to be combined into one. In larger pipelines with stages requiring greater arithmetic intensity or a greater number of operations being fused, this optimization takes advantage of improved locality between when a value is computed and when it is used. In the case of our simple pipeline, this transformation can be achieved by inlining the expression $\boxplus_{j=0}^{n} f(j)$ into each occurrence of the binding $buf$ in the body of the let-binding. This transformation can be stated more generally as the following equivalence:

$$\overline{\mathsf{let}\ x\ := e_1\ \mathsf{in}\ e_2 = e_2[e_1/x]}$$

Although this equivalence is relatively simple given that it is exactly in-line with the denotational semantics for let-bindings, every transformation that our framework provides will be of this form: a quantified equivalence between two ATL expressions, possibly with further premises.

After this rewrite is applied on the pipeline program, we are left with the following program:

$$\boxplus_{i=0}^{n} [\ 0 \le i-1\ ] \cdot \left(\boxplus_{j=0}^{n} f(j)\right)[i-1] \oplus \left(\boxplus_{j=0}^{n} f(j)\right)[i]$$

Although the two stages in this pipeline have been combined into one, there is still a lot of redundant computation being performed, resulting in a lack of locality. The full inlined generation expression is computed only to have most array cells ignored, choosing just one index to read. This artifact is common with substitution-based reductions. In order to reduce this program further and achieve a fully fused form, a separate rewrite theorem is needed:

$$\frac{0 \le k < n}{\left(\boxplus_{i=0}^{n} e\right)[k] = e[k/i]}$$

This equivalence follows from the intuition that an in-bounds access to a comprehension yields the comprehension body evaluated at the right index. This identity, of course, only holds under the premise that $k$ is a nonnegative integer less than $n$.

In order to reduce the pipeline program further, this rewrite would need to be applied at two sites. However, this scheduling rewrite reducing an access into a generation cannot be applied in the same direct manner as the first let-inlining scheduling rewrite. The issue here is two-fold. First, the access index quantified as $k$ in the rewrite theorem statement will refer to $i-1$ and $i$ in the application sites of this rewrite. However, both of these index expressions contain $i$, which is a binding introduced by the $\boxplus$ operator and is not visible outside of the subexpression. Additionally, the rewrite will only succeed if we can prove its premise about indices staying in-bounds. Since the index expressions in question are not visible in the current context, there is no known information constraining the values they may take on.

Most scheduling rewrites provided in this framework are similar to this example in requiring proof of bounds at the rewrite site–possibly under binders. The semantics of the ATL language constructs such as generation and summation naturally impose these constraints on their subexpressions. In order for conditional scheduling rewrites to succeed in any ATL subexpression, additional logical machinery is needed to take advantage of the implicit additional information in ATL subexpressions.

### 4.2.2 Binders and Contexts

The generation, summation, and let-binding language constructs introduce name bindings for the iterated indices and the let-bound expression, respectively. Therefore, optimizing subexpressions in the bodies of such constructs will require rewriting under binders. A common approach used to apply rewrites under binders or otherwise reason about the expressions within the bodies of functions is to appeal to axioms like functional extensionality since otherwise functions are treated as opaque terms. The principle of functional extensionality states that two functions are equivalent if they agree on output values for every possible input.

$$\frac{\forall x.\ f\ x = g\ x}{f = g}\ \text{FunctionalExtensionality}$$

However, we need a stronger principle here that also allows proof of side conditions using assumptions introduced by binders. For example, in the body of tensor generation, the value of the bound index is limited by the extents of the loop. Therefore, equivalence of expressions in the body of a generation operation can be described in the following lemma:

$$\frac{\forall x.\ 0 \le x < n \to e_1[x/i] = e_2[x/i]}{\boxplus_{i=0}^{n} e_1 = \boxplus_{i=0}^{n} e_2}\ \text{GenExtensionality}$$

To allow for rewriting under binders introduced by tensor summation we prove the following similar context-producing lemma:

$$\frac{\forall x.\ 0 \le x < n \to e_1[x/i] = e_2[x/i]}{\sum_{i=0}^{n} e_1 = \sum_{i=0}^{n} e_2}\ \text{SumExtensionality}$$

This GenExtensionality theorem can be used to aid in applying the final rewrites needed to schedule the two-stage pipeline into a totally fused program. The equivalence we are trying to establish with the rewrites is stated below:

$$\boxplus_{i=0}^{n} [\ 0 \le i-1\ ] \cdot \left(\boxplus_{j=0}^{n} f(j)\right)[i-1] \oplus \left(\boxplus_{j=0}^{n} f(j)\right)[i] = \boxplus_{i=0}^{n} [\ 0 \le i-1\ ] \cdot f(i-1) \oplus f(i)$$

We want to apply the general tensor-comprehension lemma stated above, whose premise quantifies over in-bounds loop indices. Letting that fresh local variable also be called $i$, we must prove the following given $0 \le i < n$:

$$[\ 0 \le i-1\ ] \cdot \left(\boxplus_{j=0}^{n} f(j)\right)[i-1] \oplus \left(\boxplus_{j=0}^{n} f(j)\right)[i] = [\ 0 \le i-1\ ] \cdot f(i-1) \oplus f(i)$$

In this context, the access of $i$ into the generation is valid, and the rewrite is able to succeed and result in the following schedule:

$$\boxplus_{i=0}^{n} [\ 0 \le i-1\ ] \cdot \left(\boxplus_{j=0}^{n} f(j)\right)[i-1] \oplus f(i)$$

However, the access at $i-1$ is not provably valid in this context just yet. Although the knowledge that $0 \le i < n$ provides that $i-1 < n$, there is no guarantee that it is nonnegative. The guard surrounding the $i-1$ access provides additional logical context that may assist this rewrite. Since this guard acts as an indicator function of some predicate and acts as the identity function if the predicate is true and effectively zeroes out the guarded expression if false, any shape-preserving rewrite applied in the guarded expression may assume the guard's predicate. This equivalence is formulated as the context-producing lemma below:

$$\frac{\mathsf{shape}(e_1) = \mathsf{shape}(e_2) \qquad p \to e_1 = e_2}{[\ p\ ] \cdot e_1 = [\ p\ ] \cdot e_2}$$

We will take advantage of this lemma to prove our intended rewrite:

$$[\ 0 \le i-1\ ] \cdot \left(\boxplus_{j=0}^{n} f(j)\right)[i-1] = [\ 0 \le i-1\ ] \cdot f(i-1)$$

By applying the guard-specific context-producing lemma, we arrive at the following equivalence.

$$\left(\boxplus_{j=0}^{n} f(j)\right)[i-1] = f(i-1)$$

Now in addition to the information provided from the generation that $0 \le i < n$, the context includes the constraint that $0 \le i-1$. This is sufficient information to ensure the validity

of the access, and so the scheduling rewrite may be applied here. Finally, we arrive at the following fully fused schedule of the two-stage pipeline:

$$\boxplus_{i=0}^{n} [\ 0 \leq i-1\ ] \cdot f(i-1) \oplus f(i)$$

Through these context-producing lemmas, we are able to collect information to aid in the rewriting of a subexpression of the program in a syntax-directed manner. Our framework takes advantage of this pattern and is able to mechanize the descent through subexpressions of a program, while executing what appear to be single top-level rewrites. We equip other binding constructs with similar context lemmas, and our `rw` tactic applies the lemmas automatically to rewrite under binders.

### 4.2.3 Rewrite Tactics and Automation

In order to provide a user-scheduling experience that consists of high-level, algebraic rewrites to induce program transformations, our framework provides a set of tactics to abstract and automate the logical reasoning described in the previous sections. The central schedule-rewriting tactic in our framework is `rw`, which formalizes the patterns we used in the prior example. This tactic takes as an argument the name of the theorem representing the scheduling rewrite to be performed, plus a number of optional arguments for configuration and application-site specificity. For example, over the course of transforming the simple pipeline program from a two-stage schedule into a completely fused schedule, the conditions resulting from the use of scheduling rewrites include $0 \leq i < n$ and $0 \leq i - 1 < n$ in their respective contexts. Like most conditions generated within this framework, these are easily proven automatically and require no further user interaction. As a result, the scheduling process on the simple pipeline we have walked through is achieved in our framework as the following high-level, interactive proof script consisting of a series of rewrite theorems.

```
reschedule.      (* generic marker to begin derivation *)
rw unfold_let.
rw get_gen.
rw get_gen.
done.            (* generic marker to conclude derivation *)
```

The conditions that must be proven in context in order for a rewrite to be applied largely reduce to equalities and comparisons between the arithmetic expressions representing the shapes of dimensions within the structure of the program. Rewrites that produce source-code in the core ATL language result in indexing expressions and loop bounds that land in the world of affine arithmetic, which is decidable.

However, if we were to express programs that implement optimizations such as tiling, which would introduce dimensions with terms that include ceiling division and modulo. These arithmetic expressions and therefore the conditions for the soundness of certain rewrites are not exclusively affine. As a result, applying such rewrites and discharging their conditions automatically would be undecidable in the general case. However, even expressions generated from the use of reshape operators arise in a regular form expressing Euclidean factorization

of known terms. Therefore, we are still able to prove the conditions automatically for all interesting examples and common use cases demonstrated in this dissertation.

## 4.3 Compilation

In order to demonstrate the ability of our framework to express useful, performant schedules, we implemented a lowering algorithm from ATL into C to be able to produce runnable code (which will turn out to have competitive performance). In this section, we present the stages of the lowering process and the rules for code generation. The details of the proof of correctness of this lowering algorithm are explored in Chapter 5.

### 4.3.1 Access Safety

ATL programs make frequent use of the indicator function to guard branches of execution, particularly in the presence of boundary conditions where a tensor access may be conditionally valid. At the level of ATL semantics, these guards ensure that out-of-bounds accesses are benign: invalid accesses are masked by the guard and evaluate to the default zero value rather than producing undefined behavior. This convention is convenient for equational reasoning, since it allows boundary-sensitive programs to be written in a direct algebraic style without requiring explicit case analysis at every access.

The generated C code, however, does not enjoy this semantic leniency. In the target language, an out-of-bounds array access is undefined behavior, even if the accessed value is subsequently multiplied by zero or otherwise rendered semantically irrelevant. As a result, the operational interpretation used for reasoning about ATL programs is only sound with respect to generated code when all concrete memory accesses performed after lowering are in bounds.

To bridge this gap, the framework provides a tactic, `check_safe`, that statically discharges this safety obligation. Like the other proof-producing procedures in the scheduling framework, `check_safe` proceeds by structurally traversing the ATL term while accumulating the contextual information needed to reason about indices. As it descends through generators, reductions, guards, and let-bindings, it records the symbolic bounds associated with in-scope index variables and tensor shapes. When it encounters a tensor access, it attempts to prove that the corresponding index expression is both nonnegative and strictly within the bounds of the accessed tensor.

When this procedure succeeds, it establishes that every access performed by the lowered program is within bounds, ensuring that the generated C code cannot trigger undefined behavior through invalid memory access. This allows the permissive treatment of out-of-bounds indexing in ATL's semantics to serve as a convenient reasoning device at the source level, while still preserving the stronger safety requirements imposed by the imperative target. As elsewhere in the framework, this safety condition is enforced extrinsically through proof automation rather than intrinsically through dependent types, avoiding the overhead of carrying index bounds directly in the representation of ATL terms while still providing the guarantees required for sound compilation.

### 4.3.2 Normalization

In order to reduce the logical complexity required of code generation, we first normalize the form of the program to be compiled. This normalization, unlike code generation, need not be trusted since it consists entirely of verified rewrites.

#### Dimensionality Specialization

At the time of compilation, the input is no longer dimensionally polymorphic. In other words, $\tau$ has been instantiated with a fully concretized tensor type. Although the exact size of each dimension is still parametric and will be taken in as input into the compiled pipeline, the full dimensionality of the input and therefore the program is known. This allows a use of a polymorphic operator to be expanded to its type-specific equivalent. In particular, this allows the $\oplus$ binary operator to be replaced with standard addition for the scalar type and an addition function tensor_add that performs addition on tensors with the semantics described in Figure 3.2.

Take for example the unnormalized schedule for the fully fused simple pipeline program illustrated below.

$$\boxplus_{i=0}^{n} [\ 0 \leq i-1\ ] \cdot f(i-1) \oplus f(i)$$

At the time of compilation, the dimensionality of $f$ must be specified as an input to the pipeline. If the input $f$ is specialized to be a function of type $\mathbb{Z} \to \mathbb{R}$, then this stage of normalization will result in the following program:

$$\boxplus_{i=0}^{n} [\ 0 \leq i-1\ ] \cdot f(i-1) + f(i)$$

If the input $f$ were specialized to be a function of type returning a tensor of any dimension, then this stage of normalization will result in the following program:

$$\boxplus_{i=0}^{n} \mathsf{tensor\_add}\ ([\ 0 \leq i-1\ ] \cdot f(i-1))\ (f(i))$$

Since this is mere instantiation of an abstract construct, all specifications of the polymorphic operator and therefore all consequent properties proven still hold for the instantiation, so it maintains semantic equivalence.

#### Normalization Lemmas

A computational pipeline written in ATL can be interpreted as a series of computed values being realized into output and intermediary buffers, with these assignments occurring at the leaf nodes in the program's AST. As a result, each stage will correspond to a buffer and a set of loop nests shaping computation to be stored into the buffer at each iteration. In the target language C, this terminal assignment for expressions must be done at the scalar granularity. However, at this point in normalization, there still remain leaf-node expressions that are not scalar. Take for example the fully fused pipeline schedule. Consider the case where the input

function $f$ is taken to have a return type of $[\mathbb{R}]$, where the shape of its values is $[a]$. One step of type-specific operator specialization yields the program below:

$$\boxplus_{i=0}^{n} \mathsf{tensor\_add}\ ([\ 0 \leq i-1\ ] \cdot f(i-1))\ (f(i))$$

The final expression computed here to be stored in the output buffer is the sum of $f(i-1)$ and $f(i)$. However, this expression has a nonscalar type. In order to perform this assignment, a loop must be inserted for each dimension of this tensor and each element of it accessed and assigned. This transformation can be interpreted as yet another proved rewrite, similar to a scheduling rewrite:

$$\frac{\mathsf{shape}(v_1) = \mathsf{shape}(v_2) = n :: \_}{\mathsf{tensor\_add}\ v_1\ v_2 = \boxplus_{i=0}^{n}\ v_1[i] \oplus v_2[i]}$$

This lemma states that the sum of two tensors expressed as an application of the tensor_add function can be rephrased as a generation where each element is the sum of the elements of each tensor at that specific index. Not only is this equivalence directly in-line with the semantic definition of tensor addition, the transformation that it induces does not affect the ultimate schedule produced from code generation, unlike most scheduling rewrites. After one application of this normalization lemma on the example pipeline, we arrive at the following program:

$$\boxplus_{i=0}^{n} \boxplus_{j=0}^{a} [\ 0 \leq i-1\ ] \cdot (f(i-1))[j] \oplus (f(i))[j]$$

The notable differences here are that the leaf-node expression being computed and stored is no longer a sum of $f(i-1)$ and $f(i)$ but an access into $f(i-1)$ and $f(i)$, and there is a new loop nest explicitly introduced into the ATL source. There is once again a polymorphic $\oplus$ operator introduced by a rewrite lemma. However, the dimensionality of this program is still known, so this operator can once again be normalized into a specific instance, yielding the following program.

$$\boxplus_{i=0}^{n} \boxplus_{j=0}^{a} [\ 0 \leq i-1\ ] \cdot (f(i-1))[j] + (f(i))[j]$$

At this point there are no longer any polymorphic operators present in the program, and the inner expressions of all loop nests to be computed and stored are scalar, so we have arrived at the final normal form of this program.

Our remaining crucial loop-oriented normalization is stated more generally. It operates on leaf-node expressions that are not tensor additions but are of tensor types and need to be normalized.

$$\frac{\mathsf{shape}(v) = n :: \_}{v = \boxplus_{i=0}^{n}\ v[i]}$$

By lifting the logic and reasoning of normalization into the verified portion of our stack rather than reasoning about it during code generation, we formalize and prove these transformations as lemmas to be applied on the program as rewrites.

## 4.4 Lowering

We define lowering algorithm as a function, $\mathcal{L}$, that takes in an ATL program and produces a C program that materializes the equivalent tensor into a previously allocated output buffer in memory as a one-dimensional array. This lowering function takes an argument $o$ representing an identifier to name the output buffer and an argument $a$ that indicates whether or not the final computed value is meant to be stored ($=$) or reduced ($+=$) into the output buffer. The lowering function also include an argument $\Gamma_{\mathsf{sh}}$, representing the shape context which maps each tensor currently in scope to its shape, including both the program's input tensors and any intermediate tensors introduced by preceding let-bindings. This function is defined recursively on the ATL language constructs.

### 4.4.1 Compiling Compute Order

For example, let's reconsider the lowering algorithm applied to the matrix multiplication example ATL program from before.

$$\mathcal{L}\ \left(\boxplus_{i=0}^{N}\ \boxplus_{j=0}^{M}\ \sum_{k=0}^{K}\ m_1[i;k] * m_2[k;j]\right)\ o\ a\ \Gamma_{\mathsf{sh}}$$

When the lowering function encounters a tensor generation, it generates a `for` loop with equivalent bounds for the iteration index $i$. Naturally, the body of the `for` loop recursively contains the lowering of the body of the tensor generation.

```
for (int i = 0; i < N; i++) {
  L ( ⊞_{j=0}^{M} Σ_{k=0}^{K} m1[i;k] * m2[k;j] ) o a Γsh
}
```

In this case, lowering encounters another tensor generation and produces the equivalent `for` loop, this time with an index $j$ and the bound $M$.

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < M; j++) {
    L ( Σ_{k=0}^{K} m1[i;k] * m2[k;j] ) o a Γsh
  }
}
```

Next, the lowering encounters tensor summation which while similarly generate an equivalent `for` loop with an index $k$ and the upper bound $K$ similar to how the algorithm handles the lowering of generation constructs. However, the lowering of tensor summation modifies the storage mode argument $a$ to be reduction.

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < M; j++) {
    for (int k = 0; k < K; k++) {
```

```
      L (m1[i;k] * m2[k;j]) o (+=) Γsh
    }
  }
}
```

Once the lowering function is applied to a scalar ATL expression, it must flatten all higher-dimensional tensor accesses into a single integer access. To do so it uses the shape context $\Gamma_{\mathsf{sh}}$ to look up the shape of the tensors being accessed and uses the dimension sizes as strides and offsets to properly construct the flattened, one-dimensional array access. So for example, the matrix multiplication body expression $m_1[i;k] * m_2[k;j]$ will become $m_1[i * K + k] * m_2[k * M + j]$ after this access flattening process.

Now, notice that neither the ATL source language constructs nor their semantics actually enforce the computation order that is produced in this code-generation scheme. This is a natural albeit arbitrary choice made in the lowering algorithm to tie the computation order to the tensor generation and summation constructs in this. And in doing so, we allow programmers to control the compute order of the execution strategy of an ATL program by manipulating source-level constructs in the ATL program directly.

### 4.4.2 Compiling Storage Order

Once the ATL expression argument passed to the lowering function is a scalar expression, the lowering must generate the flattened storage expression indexing into the output buffer argument $o$. For example, in the matrix multiplication program, the final scalar expression of the program to be lowered is $m_1[i;k] * m_2[k;j]$, which can not be assigned directly to $o$, since that buffer is the destination for the full array described by the original ATL program, not a single scalar expression. In the lowering example below, we denote storage expression as $o(i,j)$.

```
for (int i = 0; i < N; i++) {
  for (int j = 0; j < M; j++) {
    for (int k = 0; k < K; k++) {
      o(i,j) += m1[i * K + k] * m2[k * M + j]
    }
  }
}
```

The logical access $o(i,j)$ corresponds to a physical access into the output buffer $o$ as a function of indices $i$ and $j$. Note that this expression is not a function of $k$, which was an index variable introduced by the tensor summation construct. This means that this variable corresponds to a tensor dimension over which reduction is being performed and therefore corresponds to a single value of the output and does not affect the index expression produced. The choice of function that $o(i,j)$ resolves to determines the storage order of this computation. As with compute order, this is not dictated by the source program itself but is instead a convention imposed during code generation. Unlike with compute order, there are no natural source-level constructs we can tie storage order to. Therefore, in order to generate a default storage ordering, the lowering will produce a flattened access expression that yields an access

into the equivalent flattened array the corresponds with a conventional row-major memory layout. For this particular example, $o(i, j)$ should resolve to the following flattened access.

```
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < M; j++) {
      for (int k = 0; k < K; k++) {
        o[i * M + j] += m1[i * K + k] * m2[k * M + j];
      }
    }
}
```

More generally, the lowering strategy constructs a storage index expression determined by the sequence in which index variables are introduced by generation and summation constructs. Each such variable contributes both an index expression and the extent of its associated dimension. For example, $i$ is associated with a dimension size of $N$ and $j$ with a dimension size of $M$. Taken together, this ordered sequence of index–extent pairs defines the logical layout of the tensor and can be systematically converted into the flattened storage index used in the generated code. We represent this intermediate structure for the destination storage index explicitly as a list of pairs where the first entry is an integer-valued index expression and the second is the size of the corresponding logical dimension. The corresponding index data structure produced by the example above would be:

$$[(\mathsf{i},\mathsf{n});(\mathsf{j},\mathsf{m})] \ : \ [Z_e * Z_e]$$

The index data structure represents the logical multidimensional access into a tensor. This convention allows us to construct a flatten_index function to generate the integer expression representing the equivalent flattened access index into the physical flattened array.

$$\mathsf{flatten\_index} \ : \ [Z_e * Z_e] \to Z_e$$
$$\mathsf{flatten\_index} \ [(\mathsf{i}, \mathsf{N}); (\mathsf{j}, \mathsf{M})] = \mathsf{i} * \mathsf{M} + \mathsf{j}$$

Therefore, it might be tempting to equip our lowering function $\mathcal{L}$ with an additional index structure argument $\mathcal{I}$ to represent the destination index. However, this will be insufficient once we introduce certain language constructs that require us to build this index structure in a more complex, deferred manner. Instead, rather than passing down an index data structure of type $[Z_e * Z_e]$ as an argument to lowering, we pass down a function of type $[Z_e * Z_e] \to [Z_e * Z_e]$, which describes how to build upon or modify an index. Logically, this function describes the ongoing *transformation* on the index space, with one possible transformation being the expansion of the space by the introduction of another dimension and index. We call this argument the *reindexer* and represent it as $\theta$ as an argument to the lowering function $\mathcal{L}$. Finally, we define the lowering algorithm for the core ATL language constructs as the function $\mathcal{L}$ shown in Figure 4.1.

## 4.5 Reshape Operators

The lowering algorithm described so far gives a straightforward interpretation of core ATL programs as imperative code. Generation and summation constructs determine the order in which values are computed, while the sequence of indices they introduce determines how

```
Fixpoint ℒ e o a θ Γ_sh := match e with
| ⊞_{i=lo}^{hi} e' ⇒ for (int i = lo; i < hi; i ++) { ℒ e' o a (λ idx. θ ((i − lo, hi − lo) :: idx)) Γ_sh }
| Σ_{i=lo}^{hi} e' ⇒ for (int i = lo; i < hi; i ++) { ℒ e' o (+=) θ Γ_sh }
| [ p ] · e' ⇒ if (p) { ℒ e' o a θ Γ_sh }
| let x := e1 in e2 ⇒ match ‖e1‖ with
                       | [] ⇒ (* Scalar *)
                       float x = 0; ℒ e1 x (=) (λx.x) c; ℒ e2 o a θ (Γ_sh[x] = [])
                       | sh ⇒ (* Array *)
                       float *x = calloc (fold_left mul sh 1, sizeof float);
                       ℒ e1 x (=) (λx.x) c; ℒ e2 o a θ (Γ_sh[x] = ‖e1‖); free(x);
                       end
| s ⇒ o[ flatten_index (θ []) ] a (|s|)_{Γ_sh}
end.
```

Figure 4.1: Lowering algorithm for core ATL constructs

| | |
|---|---|
| Concatenate | $e \circ e$ |
| Transpose | $e^T$ |
| Flatten | flatten $e$ |
| Split | split $I$ $e$ |
| Pad on the Right | $\text{pad}_\text{r}$ $I$ $e$ |
| Pad on the Left | $\text{pad}_\text{l}$ $I$ $e$ |
| Truncate from the Right | $\text{trunc}_\text{r}$ $I$ $e$ |
| Truncate from the Left | $\text{trunc}_\text{l}$ $I$ $e$ |

Figure 4.2: Reshape operators

those output values are to be stored in memory. This yields a simple and well-defined default lowering strategy, but it also exposes a central limitation: in the core language, compute order and storage order remain coupled. Reordering the structure of a program changes not only the order in which values are produced, but also the layout into which they are written. As the tiled examples in earlier chapters illustrated, many important tensor optimizations require these two decisions to vary independently. To express such transformations in the source language, ATL must be able to decouple compute from storage order. We need something like source-level, control over storage order through how these index expressions are generated.

Therefore, we introduce a family of operators into ATL called *reshape operators*. Reshape operators are a set of functions that perform certain standard tensor transformations such as transpose and concatenation. These operators are defined in terms of existing constructs in the core ATL embedding but additionally act as compiler directives to prompt special strategies when lowering to C. In this section, we present our set of reshape operators in Figure 4.2 and demonstrate the scheduling control they provide during code generation.

$$
\begin{aligned}
e_1 \circ e_2 &:= \boxplus_{i=0}^{|e_1|+|e_2|} \left[\, i < |e_1| \,\right] \cdot e_1[i] \oplus \left[\, |e_1| \leq i \,\right] \cdot e_2[i - |e_1|] \\
e^T &:= \boxplus_{x=0}^{|e[0]|} \boxplus_{y=0}^{|e|} e[y; x] \\
\mathsf{flatten}\ e &:= \boxplus_{i=0}^{|e|\times|e[0]|} \sum_{j=0}^{|e|} \sum_{k=0}^{|e[0]|} \left[\, i = j \times |e[0]| + k \,\right] \cdot e[j; k] \\
\mathsf{split}\ k\ e &:= \boxplus_{i=0}^{\lceil |e|/k \rceil} \boxplus_{j=0}^{k} \left[\, i \times k + j < |e| \,\right] \cdot e[i \times k + j]
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{pad_r}\ k\ e &:= \boxplus_{i=0}^{|e|+k} \left[\, i < |e| \,\right] \cdot e[i] \qquad & \mathsf{trunc_r}\ k\ e &:= \boxplus_{i=0}^{k} e[i] \\
\mathsf{pad_l}\ k\ e &:= \boxplus_{i=0}^{|e|+k} \left[\, k \leq i \,\right] \cdot e[i - k] \qquad & \mathsf{trunc_l}\ k\ e &:= \boxplus_{i=0}^{k} e[i + |e| - k]
\end{aligned}
$$

Figure 4.3: ATL definitions for reshape operators

### 4.5.1 Compute and Storage Order

Recall that these operators are fully expressible core ATL constructs discussed in prior sections. Reshape operators can be unfolded into their core ATL definitions, shown in Figure 4.3. By inlining these reshape operators, we introduce new compute orders by introducing the generation and summation constructs used in their definitions.

However, the true novel utility of reshape operators is in the way they are able to affect storage reordering by transforming the default index expression used in array assignments within the generated loop nests. This is accomplished by allowing each reshape operator to compose an index transformation onto the reindexer argument to lowering. The rules of the lowering algorithm specific to the reshape operators of the ATL language are shown in Figure 4.4.

Each reshape operator $R$ is associated with a corresponding reindexer $\theta_R$ that performs an index transformation on the index structure being constructed in lowering that analogous to the transformation accomplished by the tensor operation itself. In the following section, we break down the functional semantics of each reshape operator and ascribe their corresponding reindexer.

### 4.5.2 Reshape Operator Reindexers

Each manipulates an index in a way consistent with the functional semantics of the reshape operator it is associated with. For each reshape operator, we define the reindexer functions that the lowering algorithm uses in Figure 4.5.

#### Concatenate

The concatenation operation is defined to link together two separate tensor expressions into one. For one-dimensional expressions, it behaves exactly as list concatenation would and

```
Fixpoint ℒ e o a θ Γsh := match e with
  ...
  | flatten e' ⇒ ℒ e' o a (θ . θflatten) c
  | split k e' ⇒ ℒ e' o a (θ . (θsplit k)) c
  | e^T ⇒ ℒ e' o a (θ . θtranspose) c
  | e1 ∘ e2 ⇒ match (‖e1‖),(‖e2‖) with
  | n1::_,n2::_ ⇒ ℒ e1 o a (θ . (θpadr n2)) c; ℒ e2 o a (θ . (θpadl n1)) c
              | _,_ ⇒ ;  end
  | padl k e' ⇒ ℒ e' o a (θ . (θpadl k)) c
  | padr k e' ⇒ ℒ e' o a (θ . (θpadr k)) c
  | truncl k e' ⇒ ℒ e' o a (θ . (θtruncl k)) c
  | truncr k e' ⇒ ℒ e' o a (θ . (θtruncr k)) c
  end.
```

Figure 4.4: Lowering rules for reshape operators

```
Definition θflatten idx := match idx with
                        | (i1,dim1)::(i2::dim2)::idx' ⇒
                              (i1*dim2+i1,dim1*dim2)::idx'
                        | _ ⇒ idx  end.
Definition θtranspose idx := match idx with
                        | (i1,dim1)::(i2::dim2)::idx' ⇒
                              (i2,dim2)::(i1,dim1)::idx'
                        | _ ⇒ idx  end.
Definition θsplit k idx := match idx with
                        | (i,dim)::idx' ⇒
                              (i/k,dim//k)::(i % k,k)::idx'
                        | _ ⇒ idx  end.
Definition θtruncr k idx := match idx with
                        | (i,dim)::idx' ⇒  (i,dim − k)::idx'
                        | _ ⇒ idx  end.
Definition θtruncl k idx := match idx with
                        | (i,dim)::idx' ⇒  (i − k,dim − k)::idx'
                        | _ ⇒ idx  end.
Definition θpadr k idx := match idx with
                        | (i,dim)::idx' ⇒  (i,dim + k)::idx'
                        | _ ⇒ idx  end.
Definition θpadl k idx := match idx with
                        | (i,dim)::idx' ⇒  (i + k,dim + k)::idx'
                        | _ ⇒ idx  end.
```

Figure 4.5: Reshape-operator reindexers

naturally extends to higher dimensions by effectively gluing together the two expressions one after the other with regards to their outermost dimension. This operator signals code generation to store two tensors one after the other in a shared output buffer, which makes this operator particularly useful for implementing loop splitting and creating loop epilogues, since it results in more than one loop nest being able to write into the same buffer, as shown in Figure 4.4.

Using the $\circ$ operator, loop splitting can be introduced in a rewrite with the following theorem:

$$\frac{0 \leq k < n}{\boxplus_{i=0}^{n} e = \left(\boxplus_{i=0}^{k} e\right) \circ \left(\boxplus_{i=k}^{n} e\right)}$$

Using this theorem, we can further schedule the fused two-stage pipeline program by splitting the main generation at index 1 to isolate the guarded cases and achieve the following program:

$$\left(\boxplus_{i=0}^{1} \boxplus_{j=0}^{a} [\ 0 \leq i-1\ ] \cdot f(i-1)[j] + f(i)[j]\right) \circ \left(\boxplus_{i=1}^{n} \boxplus_{j=0}^{a} [\ 0 \leq i-1\ ] \cdot f(i-1)[j] + f(i)[j]\right)$$

Note that the guard against the nonnegativity of $i-1$ in the second loop is now trivially true within the context of the loop and can be removed. Our framework provides a tactic called `simpl_guard` that automatically descends through a program and reduces any provably true arithmetic guard condition into `true`, removing verifiably trivial guards using the following rewrite theorem:

$$\overline{[\ \mathsf{true}\ ] \cdot e = e}$$

After executing the `simpl_guard` tactic, the pipeline program arrives at the following schedule:

$$\left(\boxplus_{i=0}^{1} \boxplus_{j=0}^{a} ([\ 0 \leq i-1\ ] \cdot f(i-1))[j] + f(i))[j]\right) \circ \left(\boxplus_{i=1}^{n} \boxplus_{j=0}^{a} f(i-1)[j] + f(i)[j]\right)$$

### Transpose

When applied to a matrix, the transpose operator performs the equivalent function as its mathematical counterpart, in that it switches row and column indices and as a result produces an expression flipped along its diagonal with the outermost dimension swapped with the dimension immediately inside. Thanks to shape polymorphism, this operation naturally extends the mathematical definition of a matrix transpose and is well-defined in higher dimensions. In code generation, this operator is implemented by switching the indices associated with the dimensions being transposed inside the assignment-indexing expression (Figure 4.5).

#### Flatten

The flatten operator reduces the dimensionality of an $n$-dimensional tensor into an $(n-1)$-dimensional tensor while preserving the same contents. As shown in Figure 4.3, this operator effectively does so by sequentially concatenating each of its rows one after the other, modifying the storage-indexing expression by combining the two accesses associated with the indices being flattened into one (Figure 4.5). In the absence of any further reshaping, flattening does not introduce any fundamental change in the relationship between compute and storage order of a tensor. However, when combined with transposition, flattening allows for the expression of tiled computation orderings.

#### Split

A split operation is the natural left inverse of a flatten operation. This operator takes an $n$-dimensional tensor and some splitting factor $k$ and splits the tensor into an $n+1$-dimensional tensor containing subunits of length $k$. If the original tensor is unevenly split into subunits, the final tail is padded with 0 values (Figure 4.3). Compilation simply breaks the single iteration variable into higher- and lower-order components for purposes of indexing into the array being written to (Figure 4.5).

Our most common use of the split operation in this paper was to help introduce flatten operations, thanks to their natural adjunction with each other.

### 4.5.3 Safe Garbage

In order to maintain shape consistency within a program, buffers and computation windows are often extended and abbreviated to achieve specific dimensions. For instance, when processing an image in tiled order, the total image size is not always divisible by the tile size. We may want to overallocate intermediate memory (padding) or maintain regular loop sizes, without writing to unallocated or unimportant memory (truncation of the computation).

Often, the exact values with which these computations are extended from an output are not important or directly accessed, and so, instantiating them by writing into these regions of memory is a wasted effort. Since all core language constructs necessitate some form of writing into memory, we introduce pad and truncate operators as natural adjoints to construct shape-consistent tensors in the source but also stand in as no-op commands in the lowering. While these operators do not affect the relative compute and storage order of an expression, they do affect loop bounds and logic of the generated code. Therefore, not only is such an implementation more efficient, but it implies all garbage values in memory described by these operators may safely remain uninstantiated.

#### Pad

Often when an allocation or computation window is expanded due to shape constraints imposed in a larger pipeline, the extended memory is not actually used in downstream computation. We introduce the pad operators into ATL to add padding on the left and right sides of some tensor computation. These padded values are allocated and simply left uninitialized.

In code generation, padding is implemented by expanding the dimension that is being padded by the padding factor (Figure 4.3). When a tensor is padded on the right, no special change to indexing is required; when a tensor is padded on the left, then accesses are appropriately offset (Figure 4.5).

#### Truncate

Truncation allows programs to limit the range of computation for an inner expression in the lowering and is used as the left inverse of pad. We introduce the truncate operators to truncate expression from the left and from the right side. These operators take as arguments some expression $e$ to truncate and a length $k$ to truncate them to (Figure 4.3).

Similar to the pad operators, truncate operators are lowered to introduce offsets to the accessed index when truncation occurs on the left, and they have no effect on lowering when an array is truncated on its right (Figure 4.5).

This shift may seem inherently unsafe as it could result in out-of-bounds writes. However, the introduction of these reshape operators into programs with verified scheduling rewrites always maintains that these unsafe situations and undefined behavior are avoided, since complementary padding or guards must always be introduced at the same time.

### 4.5.4 Reshape Operator Adjoint Pairs

Programs can be written and scheduled with reshape operators in the program source, but to work with reshape operators safely, it is necessary to start with a program written in the ATL core with well-behaved accesses and no special compilation directives and be able to introduce these optimizations into the program. In our framework, the idiomatic way of doing so is to introduce a pair of reshape operators such that their composition yields the identity function. These identities are stated and proven as lemmas and may be used to rewrite a program in the same manner that scheduling rewrites are performed (Figure 4.6). Once the identity pair has been introduced into the program, one of the operators (often the inner operator) is unfolded to its definition in terms of basic ATL operators. Then it is simplified into the rest of the program, exposing opportunities for further scheduling. The other operator remains intact to serve as a compilation directive, inducing the desired decoupling between compute and storage order. We have found this pattern to be useful in many common situations.

To demonstrate how these operator duals are used to introduce reshapes into programs, consider once more the fully fused pipeline schedule.

$$\boxplus_{i=0}^{n} [\ 0 \leq i-1\ ] \cdot f(i-1) \oplus f(i)$$

In order to tile this program, we introduce the tile operator using the following theorem:

$$\frac{0 \leq k \qquad \mathsf{shape}(v) = n :: s}{v = \mathsf{trunc_r}\ n\ (\mathsf{flatten}\ (\mathsf{split}\ k\ v))}$$

$$\frac{\mathsf{shape}(v) = n :: \_}{v = \mathsf{trunc_r}\ n\ (\mathsf{pad_l}\ k\ v)} \qquad \frac{\mathsf{shape}(v) = n :: \_}{v = \mathsf{trunc_l}\ n\ (\mathsf{pad_r}\ k\ v)} \qquad \frac{}{v = (v^T)^T}$$

$$\frac{\mathsf{shape}(v) = n :: k :: \_}{v = \mathsf{split}\ k\ (\mathsf{flatten}\ v)} \qquad \frac{0 \leq k \qquad \mathsf{shape}(v) = n :: \_}{v = \mathsf{trunc_r}\ n\ (\mathsf{flatten}\ (\mathsf{split}\ k\ v))}$$

Figure 4.6: Reshape operator adjoint-pair identity theorems

We use this rewrite to wrap the entire program.

$$\mathsf{trunc_r}\ n\ \left(\mathsf{flatten}\ \left(\mathsf{split}\ k\ \boxplus_{i=0}^{n} [\ 0 \leq i-1\ ] \cdot f(i-1) \oplus f(i)\right)\right)$$

Next we unfold the split operator into its ATL definition.

$$\mathsf{trunc_r}\ n\ \left(\mathsf{flatten}\ \boxplus_{i_o=0}^{\lceil n/k \rceil} \boxplus_{i_i=0}^{k} [\ i_o \times k + i_i < n\ ] \cdot \left(\boxplus_{i=0}^{n} [\ 0 \leq i-1\ ] \cdot f(i-1) \oplus f(i)\right)[i_o \times k + i_i]\right)$$

Once more, we have direct access into a generation thanks to unfolding the split operator. We can reduce this expression using the same rewrite from Section 4.2.2 and arrive at the following program.

$$\mathsf{trunc_r}\ n\ \Big(\mathsf{flatten}\ \Big(\boxplus_{i_o=0}^{\lceil n/k \rceil} \boxplus_{i_i=0}^{k} [\ i_o \times k + i_i < n\ ] \cdot ([\ 0 \leq i_o \times k + i_i - 1\ ] \cdot f(i_o \times k + i_i - 1) \oplus f(i_o \times k + i_i))\ \Big)\Big)$$

In this schedule, where there was one tensor generation before, there are now an inner and outer tensor generation iterating over that domain. The flatten and truncation reduce the dimensionality of this expression and shear off any trailing padding respectively.

In order to further accelerate computation, we may choose to break off a loop prologue and epilogue, using the concatenation rewrite rule. After doing so, we can eliminate all guards from the main loop, which will expose the loop to further unrolling and vectorization optimizations, even if the prologue and epilogue continue to use scalar instructions.

$$\begin{aligned}&\mathsf{trunc_r}\ n\ \Big(\mathsf{flatten}\\ &\Big(\Big(\boxplus_{i_o=0}^{1} \boxplus_{i_i=0}^{k} [\ i_i < n\ ] \cdot [\ 1 \leq i_i\ ] \cdot f(i_i - 1) \oplus f(i_i)\Big) \circ\\ &\Big(\boxplus_{i_o=1}^{\lfloor n/k \rfloor} \boxplus_{i_i=0}^{k} f(i_o \times k + i_i - 1) \oplus f(i_o \times k + i_i)\Big) \circ\\ &\Big(\boxplus_{i_o=\lfloor n/k \rfloor}^{\lceil n/k \rceil} \boxplus_{i_i=0}^{k} [\ i_o \times k + i_i < n\ ] \cdot f(i_o \times k + i_i - 1) \oplus f(i_o \times k + i_i)\Big)\Big)\Big)\end{aligned}$$

## 4.6 Conclusion

In this chapter, we developed the scheduling framework for ATL and showed how tensor optimization can be expressed as program derivation within a pure functional language. Rather than treating scheduling as a separate compiler phase driven by opaque directives, ATL formulates optimization as a sequence of source-to-source rewrites, each justified by functional equivalence and proven directly in Rocq. This gives scheduling a precise semantic foundation: optimized programs are not merely intended to preserve the meaning of their sources, but are derived from them through explicit proofs of equivalence.

The central contribution of this framework is that it makes scheduling itself a first-class object of formal reasoning. The scheduling process itself is now a proof-producing procedure. By embedding ATL in Rocq, we are able to express optimization procedures in the same setting as the language semantics and to build scheduling derivations by composing proved rewrite rules. This yields a system that is both extensible and sound by construction: new rewrites and automation procedures can be added modularly, while correctness continues to reduce to proving equivalence in the underlying semantics.

A key step in making this approach practical is the introduction of reshape operators. In the core language, source-to-source rewrites can manipulate computation structure, but they cannot independently control how intermediate or output tensors are written to memory. Reshape operators address this limitation by making transformations of storage structure explicit in the source language in a functional, principled manner. This allows scheduling rewrites to express optimizations that require coordinated changes to both compute order and storage order such as tiling, fusion, and layout-sensitive reorganization, comprising a menu of ways to decouple compute order and storage.

Together, these components establish ATL as a verified scheduling language for tensor computation. Programs can be transformed through explicit algebraic rewrites, optimized by composing proved equivalences, and checked for safety before compilation. The result is a scheduling system that recovers much of the flexibility of modern tensor-optimization frameworks while maintaining a proof-oriented formulation throughout. Having established how ATL programs can be systematically optimized at the source level, we now turn to the next stage of the pipeline: how these optimized functional programs are lowered into imperative code while preserving their semantics.

# Chapter 5

# Verified Lowering of a Functional Tensor Language

In this chapter, we provide the first proof of correctness for a lowering algorithm of a functional tensor language that enables separate compute and storage reordering. This algorithm is implemented in a destination-passing style to generate flattened array programs in C to compile and benchmark ATL programs. While destination-passing style has been established as useful for compiling functional array code [65], the ATL lowering algorithm had to accommodate the compute and storage reordering effected in the source language. A similar approach was used by Lin and Dubach [48] who introduced views in their IR during their lowering process to express different storage-order choices. However, none of these compilers were formally verified. In this work, the proof of correctness of this lowering algorithm is mechanized using the Rocq proof assistant.

In the course of proving the lowering algorithm correct, we identified a semantic precondition on ATL programs that is necessary for lowering to preserve correctness. The significance of this precondition becomes clearest when examining programs that include reshape operators such as truncation. These operators are essential for expressing common layout-oriented transformations, including partitioning tensors to match vector widths, cache-line boundaries, or fixed tile sizes. Although such transformations are natural and well-behaved in the forms produced by scheduling rewrites, the core ATL language itself does not enforce the invariant that makes them safe to lower. As a result, the correctness proof depends on a property of ATL programs that is not captured by the core semantics alone. Therefore in addition to the formalization of lowering and its correctness proof, we develop a type system that captures this essential safety property and makes this invariant explicit to ensure that ATL programs satisfy the conditions required for sound compilation.

## 5.1 Formalizing the ATL Language

While the scheduling-rewrite optimization stage of the ATL framework was implemented using a shallow embedding of the language in Rocq, in this work we formalize ATL with a deep embedding for the ATL language, i.e. explicit syntax trees. Therefore, in this work we provide a formal, big-step operational semantics for the ATL language for programs in

a normalized form after the precompilation normalization process outlined in Section 4.3.2. These semantics are consistent with the previously presented denotational semantics provided for the shallow embedding of ATL in Figure 3.2. We also implemented a Rocq tactic that can reify shallowly embedded ATL programs into our deep embedding, generating proof of semantic equivalence.

For an ATL program $e$, in this chapter we will use the notation $\|e\|$ to denote the shape of the tensor computed from this program, represented as a list of symbolic integer expressions. The shape $\|e\|$ of an expression is equivalent to the shape $s$ for a program $e$ that satisfies the consistency relation presented in Section 3.4. For simplicity's sake, we will also use $|e|$ to represent the integer expression giving the size of the top-level dimension, or the length of the tensor. Here we refer to symbolic expressions in the sense that a tensor's dimensions may depend on program variables. We will use the notation $\|e\|_v$ to signify the shape of the ATL program $e$ where each of its symbolic dimensions has been evaluated to concrete integers under the index context $v$ that maps index variables to integers. Likewise, $|e|_v$ represents the evaluated integer of the first dimension. Finally, we define a function genpad that takes a list of integers $sh$ as an argument and produces a tensor of that shape filled entirely with zeros.

### 5.1.1 Tensor Access Semantics

In Figure 5.1 we present the relation for evaluating a multidimensional access into a tensor in ATL. Access into a single dimension list is represented as subscripting the access index. In this relation we only concern ourselves with accesses that are well-formed. This is both in the sense that each individual dimensional access is in bounds and in the sense that the dimensionality of the access index matches the dimensionality of the tensor. This is because the ATL framework implements a static check for these safe access properties as a Rocq tactic to be invoked before compilation, discussed in Section 4.3.1.

$$\textsc{LookupScalar}\ \frac{s \in \mathbb{R}}{s[\![\ ]\!] = s} \qquad \textsc{LookupTensor}\ \frac{t_i = t' \qquad t'[\![\ l\ ]\!] = r}{t[\![\ (i :: l)\ ]\!] = r}$$

Figure 5.1: Evaluation of higher-dimensional tensor access

### 5.1.2 Evaluation of Scalar Expressions

In Figure 5.2 we present the semantics of evaluating normalized scalar expressions in ATL. This is includes a number of standard binary arithmetic operations as well as variable resolution and tensor access.

### 5.1.3 ATL Semantics

In Figure 5.3 we present a big-step operational semantics for ATL, covering both the core language and the reshape operators introduced in the previous section. Earlier chapters relied on ATL's denotational semantics to reason about program equivalence and justify source-to-source rewrites. For the purposes of compiler correctness, however, we now require

$$\textsc{GetVar}\ \frac{\Gamma[x] = t \qquad t[\![\ [\ [I_0]_v; ...; [I_n]_v\ ]\ ]\!] = r}{\langle\ x[I_0; ...; I_n], v, \Gamma\rangle \Downarrow_S r}$$

$$\textsc{Var}\ \frac{\Gamma[x] = r}{\langle\ x, v, \Gamma\rangle \Downarrow_S r} \qquad \textsc{Literal}\ \frac{r \in \mathbb{R}}{\langle\ r, v, \Gamma\rangle \Downarrow_S r}$$

$$\textsc{Mul}\ \frac{\langle\ s_1, v, \Gamma\rangle \Downarrow_S r_1 \qquad \langle\ s_2, v, \Gamma\rangle \Downarrow_S r_2}{\langle\ s_1 \times s_2, v, \Gamma\rangle \Downarrow_S r_1 \times r_2} \qquad \textsc{Div}\ \frac{\langle\ s_1, v, \Gamma\rangle \Downarrow_S r_1 \qquad \langle\ s_2, v, \Gamma\rangle \Downarrow_S r_2 \qquad r_2 \neq 0}{\langle\ s_1/s_2, v, \Gamma\rangle \Downarrow_S r_1/r_2}$$

$$\textsc{Add}\ \frac{\langle\ s_1, v, \Gamma\rangle \Downarrow_S r_1 \qquad \langle\ s_2, v, \Gamma\rangle \Downarrow_S r_2}{\langle\ s_1 + s_2, v, \Gamma\rangle \Downarrow_S r_1 + r_2} \qquad \textsc{Sub}\ \frac{\langle\ s_1, v, \Gamma\rangle \Downarrow_S r_1 \qquad \langle\ s_2, v, \Gamma\rangle \Downarrow_S r_2}{\langle\ s_1 - s_2, v, \Gamma\rangle \Downarrow_S r_1 - r_2}$$

Figure 5.2: Evaluation semantics for scalar expressions

a semantics that makes evaluation explicit, so that the behavior of the lowering algorithm can be related directly to the execution of ATL programs. The operational semantics presented here serves that role. It is consistent with the denotational account given previously in Figures 3.2 and 4.3, but is structured to support reasoning about execution and its correspondence with generated imperative code.

## 5.2 Formalizing the Target Language

The lowering algorithm translates ATL programs into low-level imperative loop nests in a restricted fragment of C. To state and prove compiler correctness, we therefore need a formal account of the target language in which lowered programs execute. In this section, we define a big-step operational semantics for the subset of C generated by ATL lowering in Figure 5.4. These semantics are deliberately minimal: they include only the constructs required to model the imperative code emitted by the compiler, while omitting features irrelevant to ATL's execution model. Program state is represented by a stack and a heap. The stack stores scalar values as floating-point numbers[1], while the heap stores higher-dimensional tensors as flattened one-dimensional arrays. The heap is modeled as a partial map from identifiers to flat arrays, and arrays themselves as partial maps from integer indices to real values. The language includes explicit memory allocation and deallocation, loop nests, conditional statements, and assignment, but excludes features such as pointer arithmetic and aliasing, which are not used by code generated through ATL lowering.

## 5.3 Compiler Correctness

We can express the compiler-correctness theorem as something like the statement below, although additional side conditions will be required. The first premise of the theorem states

[1]The semantics models floating-point values as mathematical reals; extending the development to reason soundly about machine floating-point behavior would be a valuable direction for future work.

$$\textsc{GenBase}\ \frac{[\![hi]\!]_v \leq [\![lo]\!]_v}{\left\langle \boxplus_{x=lo}^{hi}\ e, v, \Gamma \right\rangle \Downarrow []} \qquad \textsc{SumBase}\ \frac{[\![hi]\!]_v \leq [\![lo]\!]_v \qquad \|e\|_v = sh}{\left\langle \sum_{x=lo}^{hi}\ e, v, \Gamma \right\rangle \Downarrow\ \mathsf{genpad}\ sh}$$

$$\textsc{GenStep}\ \frac{[\![lo]\!]_v < [\![hi]\!]_v \qquad \langle e, v[i \mapsto lo], \Gamma\rangle \Downarrow r \qquad i \notin \mathsf{dom}(v) \qquad \left\langle \boxplus_{x=lo+1}^{hi}\ e, v, \Gamma \right\rangle \Downarrow\ t}{\left\langle \boxplus_{x=lo}^{hi}\ e, v, \Gamma \right\rangle \Downarrow\ (r :: t)}$$

$$\textsc{SumStep}\ \frac{[\![lo]\!]_v < [\![hi]\!]_v \qquad i \notin \mathsf{dom}(v) \qquad \langle e, v[i \mapsto lo], \Gamma\rangle \Downarrow r_1 \qquad \left\langle \sum_{x=lo+1}^{hi}\ e, v, \Gamma \right\rangle \Downarrow r_2 \qquad r_1 \oplus r_2 = r}{\left\langle \sum_{x=lo}^{hi}\ e, v, \Gamma \right\rangle \Downarrow\ r}$$

$$\textsc{GuardFalse}\ \frac{[\![p]\!]_v = \mathsf{false} \qquad \|e\|_v = sh}{\langle [\ p\ ] \cdot e, v, \Gamma\rangle \Downarrow \mathsf{genpad}\ sh} \qquad \textsc{GuardTrue}\ \frac{[\![p]\!]_v = \mathsf{true} \qquad \langle e, v, \Gamma\rangle \Downarrow r}{\langle [\ p\ ] \cdot e, v, \Gamma\rangle \Downarrow r}$$

$$\textsc{Bind}\ \frac{\langle e_1, v, \Gamma\rangle \Downarrow r_1 \qquad \langle e_2, v, \Gamma[x \mapsto r_1]\rangle \Downarrow r_2 \qquad x \notin\ \mathsf{dom}(\Gamma) \qquad x \notin\ \mathsf{vars\_of}\ e_1 \cup\ \mathsf{vars\_of}\ e_2 \qquad \mathsf{vars\_of}\ e_1 \cap \mathsf{vars\_of}\ e_2 = \emptyset}{\langle \mathsf{let}\ x\ := e_1\ \mathsf{in}\ e_2, v, \Gamma\rangle \Downarrow r_2}$$

$$\textsc{Concat}\ \frac{\langle e_1, v, \Gamma\rangle \Downarrow\ t_1 \qquad \langle e_2, v, \Gamma\rangle \Downarrow\ t_2}{\langle e_1\ \circ\ e_2, v, \Gamma\rangle \Downarrow\ (t_1 +\!\!+\ t_2)} \qquad \textsc{Transpose}\ \frac{\langle e, v, \Gamma\rangle \Downarrow\ t \qquad \|e\|_v = n :: m :: sh}{\langle e^T, v, \Gamma\rangle \Downarrow\ (\mathrm{transpose}\ t\ m)}$$

$$\textsc{PadRight}\ \frac{\langle e, v, \Gamma\rangle \Downarrow\ l \qquad \|e\|_v = n :: sh}{\langle \mathsf{pad}_{\mathsf{l}}\ k\ e, v, \Gamma\rangle \Downarrow\ t\ +\!\!+\ (\mathrm{genpad}\ ([\![k]\!]_v :: sh))} \qquad \textsc{Split}\ \frac{\langle e, v, \Gamma\rangle \Downarrow\ t \qquad 0 < [\![k]\!]_v}{\langle \mathsf{split}\ k\ e, v, \Gamma\rangle \Downarrow\ (\mathrm{split}\ [\![k]\!]_v\ t)}$$

$$\textsc{PadLeft}\ \frac{\langle e, v, \Gamma\rangle \Downarrow\ t \qquad \|e\|_v = n :: sh}{\langle \mathsf{pad}_{\mathsf{l}}\ k\ e, v, \Gamma\rangle \Downarrow\ (\mathrm{genpad}\ ([\![k]\!]_v :: sh))\ +\!\!+\ t} \qquad \textsc{Flatten}\ \frac{\langle e, v, \Gamma\rangle \Downarrow\ t}{\langle \mathsf{flatten}\ e, v, \Gamma\rangle \Downarrow\ (\mathrm{flatten}\ t)}$$

$$\textsc{TruncLeft}\ \frac{\langle e, v, \Gamma\rangle \Downarrow\ t}{\langle \mathsf{trunc}_{\mathsf{l}}\ k\ e, v, \Gamma\rangle \Downarrow\ (\mathrm{skipn}\ [\![k]\!]_v\ t)}$$

$$\textsc{TruncRight}\ \frac{\langle e, v, \Gamma\rangle \Downarrow\ t}{\langle \mathsf{trunc}_{\mathsf{l}}\ k\ e, v, \Gamma\rangle \Downarrow\ (\mathrm{rev}\ (\mathrm{skipn}\ [\![k]\!]_v\ (\mathrm{rev}\ t)))} \qquad \textsc{Scalar}\ \frac{\langle s, v, \Gamma\rangle \Downarrow_S r}{\langle\ s, v, \Gamma\rangle \Downarrow\ r}$$

Figure 5.3: Operational semantics for ATL

that a source ATL program $e$ evaluates to a tensor $t$. The conclusion states that executing the lowered equivalent of this program will result in a state change described by the function

$$\textsc{ForStep}\ \frac{[\![lo]\!]_v < [\![hi]\!]_v \qquad i \notin \mathsf{dom}(v) \qquad \langle\ e, v[i \mapsto [lo]_v], st, h\rangle \Downarrow_C st', h' \qquad \langle\ \texttt{for (int i=lo+1;i<hi;i++)\{e\}}, v, st', h'\rangle \Downarrow_C st'', h''}{\langle\ \texttt{for (int i = lo; i < hi; i ++) \{ e \}}, v, st, h\rangle \Downarrow_C st'', h''}$$

$$\textsc{ForBase}\ \frac{[\![hi]\!]_v \leq [\![lo]\!]_v}{\langle\ \texttt{for (int i = lo; i < hi; i ++) \{ e \}}, v, st, h\rangle \Downarrow_C st, h}$$

$$\textsc{AssignS}\ \frac{idx = [] \qquad \langle s, v, \Gamma\rangle \Downarrow_S r}{\langle x\ \mathsf{idx} = s, v, st, h\rangle \Downarrow_C (st[x \mapsto r]), h} \qquad \textsc{ReduceS}\ \frac{idx = [] \qquad \langle s, v, \Gamma\rangle \Downarrow_S r \qquad st[x] = r'}{\langle x\ \mathsf{idx}\ \texttt{+=}\ s, v, st, h\rangle \Downarrow_C (st[x \mapsto r' + r]), h}$$

$$\textsc{ReduceV}\ \frac{idx \neq [] \qquad \langle s, v, \Gamma\rangle \Downarrow_S r \qquad [\![\mathsf{flatten\_index}\ idx]\!]_v = i \qquad h[x[i]] = r'}{\langle x\ \mathsf{idx}\ \texttt{+=}\ s, v, st, h\rangle \Downarrow_C s, (h[x[i \mapsto r' + r]])}$$

$$\textsc{AssignV}\ \frac{idx \neq [] \qquad \langle s, v, \Gamma\rangle \Downarrow_S r \qquad [\![\mathsf{flatten\_index}\ idx]\!]_v = i}{\langle x\ \mathsf{idx} = s, v, st, h\rangle \Downarrow_C st, h[x[i \mapsto r]]}$$

$$\textsc{IfTrue}\ \frac{\langle\ s, v, st, h\rangle \Downarrow_C st', h' \qquad [\![p]\!]_v = \mathsf{true}}{\langle\ \texttt{if}\ (p)\{s\}, v, st, h\rangle \Downarrow_C st', h'} \qquad \textsc{IfFalse}\ \frac{[\![p]\!]_v = \mathsf{false}}{\langle\ \texttt{if}\ (p)\{s\}, v, st, h\rangle \Downarrow_C st, h}$$

$$\textsc{AllocS}\ \frac{}{\langle\ \texttt{float x = 0}, v, st, h\rangle \Downarrow_C (st[x \mapsto 0]), h}$$

$$\textsc{AllocV}\ \frac{}{\langle\ \texttt{float *x = calloc(}z\texttt{,sizeof float)}\ , v, st, h\rangle \Downarrow_C st, h[x \mapsto\ \mathsf{alloc\_arr}\ [\![z]\!]_v]}$$

$$\textsc{Free}\ \frac{h[x \mapsto arr]}{\langle\ \texttt{free(}x\texttt{)}, v, st, h\rangle \Downarrow_C st, h - x} \qquad \textsc{DeallocStack}\ \frac{st[x \mapsto r]}{\langle\ \texttt{Dealloc}\ x, v, st, h\rangle \Downarrow_C st - x, h}$$

$$\textsc{Seq}\ \frac{\langle\ s_1, v, st, h\rangle \Downarrow_C st', h' \qquad \langle\ s_2, v, st', h'\rangle \Downarrow_C st'', h''}{\langle\ s_1; s_2, v, st, h\rangle \Downarrow_C st'', h''}$$

Figure 5.4: Operational semantics of C

tensor_to_array_delta. Variables $st$ and $h$ stand for the C stack and heap.

$$\langle e, v, \Gamma\rangle \Downarrow\ t \to \cdots \to \langle\ (\mathcal{L}\ e\ o\ a\ \theta\ c), v, st, h\rangle\ \Downarrow_C\ (st, h)\ \uplus\ \mathsf{tensor\_to\_array\_delta}\ \theta\ t\ v$$

The arguments to tensor_to_array_delta are a tensor $t$, a reindexer $\theta$, and an index context $v$. The function outputs an integer-domain partial map that contains a mapping for every element of tensor $t$. The key for each element in this map represents the physical, flattened index in the array that this element occupies.

This integer mapping is constructed for each element $r$ at some multidimensional index $I_Z$ by first building the corresponding index structure, by applying the zip function over $I_Z$ and the dimensional size list representing the shape of the tensor, $\|t\|$. Here, zip is a function that takes two lists and constructs a new list where each element is a tuple of the elements of the original lists at that position. We then apply the reindexer to this index structure. Finally, we flatten the resulting index structure and evaluate it under the index valuation $v$ to produce the concrete integer index for each mapping.

Although this intermediary mapping has the same type as an array in the heap, it is only meaningful once it is added onto an existing array in the heap. Hence, we refer to this mapping as an *array delta*. In the theorem statement, we symbolize array addition as $\uplus$. Adding two arrays constructs a new array containing the union of all integer-value mappings in both original arrays. Indices that are present in both original arrays are mapped to the sums of the original array mappings.

This definition decomposes very well to accommodate independent reasoning about the storage reordering effected by each reshape operator. Consider a reshape operator $R$ with its associated reindexer $\theta_R$ like those shown in Figure 4.5. We can produce an array delta representing the application of $R$ on some tensor $t$ by directly applying the tensor_to_array_delta transformation with some reindexer $\theta$. But we should be able to produce the same array delta by applying tensor_to_array_delta directly on the tensor $t$ with a reindexer that is the composition of $\theta$ and $\theta_R$, as when we apply tensor_to_array_delta composed with $\theta_R$ on the original tensor $t$.

This statement is very close to complete. The other conditions for correctness of lowering will include some standard properties regarding the well-formedness of $e$ and equivalence of states and environments, among other invariants. Most notably, we will have to include invariants regarding the behavioral properties and well-formedness of the reindexer $\theta$, which is an entirely unconstrained quantified value in the statement as written.

### 5.3.1 Context and State

This correctness statement would not be sound if either program were executing in arbitrary environments. To begin, we must establish that the starting state in which the lowering is executing and the context in which the ATL program is being evaluated are equivalent. The semantics of both ATL interpretation and execution of C code include the iteration index valuation $v$ as one of their arguments, so we can impose direct equivalence on the valuation $v$. Establishing equivalence between the ATL context $\Gamma$ and the stack and heap is less straightforward. The ATL context maintains a map of names to tensors computed from previous `let` bindings. The equivalent flattened arrays of these tensors should also be present in the stack and heap. Formally, we can define an equivalence between $\Gamma$ and the stack and heap $(st, h)$ in terms of tensor_to_array_delta.

$$\Gamma \sim\sim (st, h) \;:=\; \forall x\ t.\ \Gamma[x] = t \longrightarrow (st, h)[x] = \mathsf{tensor\_to\_array\_delta}\ (\lambda i.i)\ t\ \emptyset$$

Note that we use a one-directional implication in formulating this invariant, because the presence of a mapping for an identifier in the stack or heap does not necessarily mean it has been bound in the context. Let us take a look at the code the lowering algorithm generates for `let` bindings.

$$\mathcal{L}\ (\text{let}\ x\ := e_1\ \text{in}\ e_2)\ o\ a\ \theta\ c$$

---

```
float *x = calloc (fold_left mul ‖e1‖ 1, sizeof float);
```
$\mathcal{L}\ e_1\ x\ (=)\ (\lambda i.i);\ \mathcal{L}\ e_2\ o\ a\ \theta\ c;$ `free(x);`

---

The mapping in the heap might have been the result of a memory allocation, while the lowering of the `let`-bound expression has yet to be written to the allocated addresses, as is the case after the allocation. From here on, $x$ is visible in the low-level state as being mapped to an array in the stack. However, there is no corresponding mapping in the ATL context yet. In a `let` binding, $x$ is only in-scope in the body of the binding. Therefore, it is only in the ATL context after line 3. If $\sim\sim$ were defined bidirectionally, this invariant would be broken by each `let` binding in between the allocation and the writing of the bound tensor. Therefore we simply state that if a tensor is bound in the ATL context, its equivalent flattened counterpart must be present in the low-level state's stack and heap.

### 5.3.2 Well-Formed Allocation

We also specify the presence and well-formedness of the allocated memory that a computation is writing into. The lowering algorithm is implemented in a destination-passing style, meaning that the argument $o$ in $\mathcal{L}\ e\ o\ a\ \theta\ c$ must be a pointer or reference to stack/heap space that has already been allocated to accommodate the computation of $e$. The stack or heap must contain an existing mapping to store the values of $e$, even if $a$ is an assignment rather than a plus-equals, since even an assignment to an index without an existing mapping is equivalent to attempting to write to unallocated memory.

Additionally, not only must $o$ include the mappings to accommodate the *size* of $e$, it must contain the mappings for the indices to which $\theta$ may send the indices of $e$. In other words, if $e$ computes a scalar and $\theta$ is the identity reindexer, then the stack must contain a mapping for $o$. If $e$ computes a nonscalar, n-dimensional tensor $t$ or $\theta$ constructs a nontrivial index space, the heap must contain a mapping for $o$ to an array with a mapping for any index reachable by applying $\theta$ on any index within the index space of $t$. We define this property as follows, using the notation $[\![sh_t]\!]$ to denote the set of indices in a tensor with some shape $sh_t$. Below the index list $[I_0; ...; I_n]$ denotes an arbitrary sequence of distinct integer index variables generated to represent a generic input to which the reindexer $\theta$ may be applied.

$$\begin{aligned}
&\text{well_formed_allocation}\ t\ \theta\ st\ h\ o\ :=\ \textbf{match}\ \theta\ (\text{zip}\ [I_0; ...; I_n]\ \|t\|)\ \textbf{with}\\
&\quad \mid\ [] \Rightarrow \exists k.\ st[o] = k\\
&\quad \mid\ _ \Rightarrow\ \ \exists a.\ h[o] = a\ \wedge \forall i.\ i \in [\![\|t\|]\!] \rightarrow \theta\ (\text{zip}\ i\ \|t\|) \in \text{dom}(a)\\
&\textbf{end}
\end{aligned}$$

### 5.3.3 Well-Formed Reindexer

We impose a variety of constraints to define the well-formedness of reindexers. The overall well-formedness of a reindexer $\theta$ depends on the iteration-index context $v$ it is being evaluated

in, as well as the tensor $t$ that it is acting on. To begin, we remind ourselves of the type of reindexers.

$$\theta \ : \ [Z_e * Z_e] \to \ [Z_e * Z_e]$$

Reindexers are functions from index to index, where indices are represented as lists of tuples of syntactic integer expressions, denoted by the type $Z_e$. For the purposes of being able to characterize some functional properties of the flattening index function the reindexer actually represents, we will define a way to evaluate a reindexer using $\Downarrow_v$.

$$\Downarrow_v \ \theta \ : [\mathbb{Z}] \to \mathbb{Z}$$

By realizing the reindexer, we change it from a function that maps symbolic indices to symbolic indices, into a function that takes in an actual integer index and returns the integer representing its counterpart, physically flattened address. We must require properties including simple structural ones that only depend on the reindexer and the symbolic index form. However, we must also include behavioral properties that describe the functional properties of the interpreted reindexer.

### Preservation of Variables

When a reindexer generates an expression for a flat index, intuitively that expression may mix variables present in the reindexer with variables present in the symbolic indexes that were given as input to the reindexer. In fact, the resulting expression should contain *exactly* those free variables, not just a strict subset of them, intuitively because a reindexer should only introduce and shuffle indices–it should not drop any.

$$\forall l.\ \mathsf{vars\_of}\ (\theta\ l) = (\mathsf{vars\_of}\ (\theta\ [])) \ \cup \ (\mathsf{vars\_of}\ l) \qquad \text{(Variable Preservation)}$$

### Well-Scoped Variables

Another property we will use to characterize a well-formed reindexer with respect to some valuation $v$ is the proper scoping of its own variables. Each variable present in the reindexer at a given point has been produced in the lowering process that binds that variable to an integer value, such as an iteration index from a tensor generation or summation, so it must be in scope.

$$\mathsf{vars\_of}(\theta\ []) \subseteq \ \mathsf{dom}\ v \qquad \text{(Well-Scoped Variables)}$$

### Variable Substitution

In addition to preserving the variables of the reindexer's arguments, it should be possible for us to reason about the evaluation of those variables under the reindexer independently of what the reindexer does. In other words, if we are substituting a variable on the application of some opaque reindexer on an index, we should be able to distribute the variable substitution onto the index itself. In the case where the substituted variable is not present within the variables of the opaque reindexer itself, the substitution can be fully moved under the reindexer application.

$$\forall l, i, x.\ i \notin \ \mathsf{dom}\ v \to \ (\theta\ l)[x/i] = \theta\ (l[x/i]) \qquad \text{(Variable Substitution)}$$

**extensionality**

Another characteristic of a well-formed reindexer is extensionality. In other words, if two indices are equivalent, then the results after applying the reindexer are equivalent. However, since indices are represented as lists of syntactic integer expressions, we relax our notion of equivalence. We need not require they be syntactically equivalent. Instead we define a notion of equivalence of integer expressions that states that, for any valuation, the expressions evaluate to the same integer.

$$x \sim_{Z_e} y := \forall v.\ [\![x]\!]_v = [\![y]\!]_v$$

From here, we can very naturally extend this equivalence from integer expressions to indices themselves.

$$\forall \mathsf{idx}_1, \mathsf{idx}_2.\ \mathsf{idx}_1 \ \sim_{[Z_e * Z_e]}\ \mathsf{idx}_2 \to \theta\ \mathsf{idx}_1 \ \sim_{[Z_e * Z_e]}\ \theta\ \mathsf{idx}_2 \qquad \text{(Extensionality)}$$

**Injectivity**

Another well-formedness property we define for reindexers is injectivity. Specifically, the reindexer must be injective over the domain of possible indices over the shape of the tensor $t$ to be computed. In other words, if we evaluate the reindexer and apply it on two literal integer indices in the index space of $t$, if the resulting integers are equal then the two integer indices must be equal.

$$\forall \mathsf{idx}_1, \mathsf{idx}_2.\ \mathsf{idx}_1 \in [\![||t||_v]\!] \to \mathsf{idx}_2 \in [\![||t||_v]\!] \to \Downarrow_v \theta\ \mathsf{idx}_1 = \Downarrow_v \theta\ \mathsf{idx}_2 \to \mathsf{idx}_1 = \mathsf{idx}_2 \qquad \text{(Injectivity)}$$

**Non-destructive Assignment**

The final well-formedness property we define for reindexers is non-destructivity in the case of assignment storage operators. The reindexer must not be able to produce an index and overwrite a value that was previously written. As a result, all indices to which the reindexer could send the indices of the tensor $t$ must not have been written previously, so those indices retain their original value, which was 0 at the time of allocation.

$$h[o] = \mathsf{arr} \to a = (\mathsf{=}) \to \forall \mathsf{idx}.\ \mathsf{idx} \in [\![||t||_v]\!] \to \mathsf{arr}[\Downarrow_v \theta\ \mathsf{idx}] = 0 \qquad \text{(Non-destructivity)}$$

Finally we can define the well_formed_reindexer property as the conjunction of the properties described above.

### 5.3.4 Compiler Correctness

To return to our correctness theorem, we include the preconditions defined above. This includes the well-formedness of the reindexer, the well-formedness of the allocation in the stack or heap, and the equivalence of execution state between the functional evaluation context and the low-level stack and heap. We have the following statement.

$\langle e, v, \Gamma \rangle \Downarrow\ t \rightarrow$
$\Gamma \sim\sim (st, h) \rightarrow$
well_formed_allocation $t\ \theta\ st\ h\ o \rightarrow$
well_formed_reindexer $\theta\ v\ t\ h\ o\ a \rightarrow$
$\langle\ (\mathcal{L}\ e\ o\ a\ \theta\ c), v, st, h \rangle\ \Downarrow_C\ (st, h)\ \uplus$ tensor_to_array_delta $\theta\ t\ v$

## 5.4 A Motivating Counterexample

Let us revisit the example used to demonstrate lowering in Section 4.5.4. The optimized program was tiled under the assumption that the tensor dimensions were evenly divisible by the tiling factor. However, if we were to produce an optimized program without this assumption, it would look like the one below.

$$\mathsf{trunc_r}\ (k_1 - n\ \%\ k_1)\ \left(\mathsf{flatten}\ \left(\left(\boxplus_{i_o=0}^{n//k_1}\ \mathsf{trunc_r}\ (k_2 - m\ \%\ k_2)\ \left(\mathsf{flatten}\ \left(\boxplus_{j_o=0}^{m//k_2}\ \boxplus_{i_i=0}^{k_1}\ \boxplus_{j_i=0}^{k_2}\ [\ \ldots\ ] \cdot \ldots\right)^T\right)\right)^T\right)\right)$$

This program structure is similar to the optimized program shown before, but the outer loop-nest bounds are calculated using ceiling division indicated by the // operator rather than floor division. Thus, the computation accounts for the elements at the end of the tensor that are not evenly divisible by the tile size. Without further adjustment, the output tensor would be larger than the original. To produce the expected output, truncation operators are introduced around the flattened tiled dimensions, to truncate the overcompute caused by the rounding. The body also contains a guard to limit the actual loop computation to the domain of the original tensor size.

While this program generates the proper tiled imperative code, the usage of truncation operators in general is unsafe, because the truncation reindexers reduce the dimension size, thereby reducing the possible index storage space while leaving the iteration space untouched. In fact, the two truncation reshape operators ($\mathsf{trunc_l}$ and $\mathsf{trunc_r}$) do not satisfy the well-formedness conditions as stated and can be used to write programs that produce unsound code.

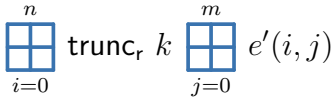
Consider the following ATL program and its lowered C program. Here, the leftmost $k$ elements are removed from a tensor generation of length $n$. The lowering algorithm would produce the following C code.

$$\mathsf{trunc_l}\ k\ \boxplus_{i=0}^{n}\ e(i)$$

```
for (int i = 0; i < n; i++) {
  o[ i - k ] = e(i);
}
```

While the index offset of $k$ in the storage expression shifts the $k$-th evaluation of $e$ into the first element of the buffer $o$, this program fails because this access is unguarded. The first $k$ iterations of the loop make out-of-bounds accesses.

The lowering of the right-truncation operator can also introduce unsoundness. The right-truncation reindexer subtracts $k$ from only the dimension in the index-dimension tuple, restricting the index space. Consider the following usage of right-truncation and its corresponding lowered C program.




```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < m; j++) {
    o[ i * (m - k) + j ] = e′(i, j);
  }
}
```

The expected output of this program is a two-dimensional tensor of size $n * (m - k)$ with elements $e'(i, j)$ evaluated for $i$ up to $n$ and $j$ up to $m - k$. The lowering algorithm would have only allocated enough memory in the buffer $o$ to store $n * (m - k)$ elements, so the last few iterations of this loop nest would also result in out-of-bounds memory accesses.

We can conclude that this lowering algorithm is generally unsound for arbitrary ATL programs. However, it was never intended for programmers to use reshape operators arbitrarily in their programs. The conceit of the ATL scheduling framework was to be able to start with a program written in core ATL constructs and to introduce reshape operators using verified scheduling rewrites. The tiled program example above was indeed derived in that way. The unsound examples are unreachable using this derivation approach. Reshape operators should only be introduced in adjoint pairs. Therefore, truncation should only be in a program if it had been introduced with a complementary pad inside it, or some unfolded or downstream rewritten/optimized equivalent. (Pad is truncation's dual that adds extra zero values to array ends.)

The intuition behind why having a pad immediately inside a truncate would be a safe reindexing transformation is relatively straightforward: the composition of a truncation reindexer and its complementary pad reindexer $(\theta_{\mathsf{truncr}}\ k) \circ (\theta_{\mathsf{padr}}\ k)$ would yield the identity reindexer. The reasoning for why an unfolded and subsequently rescheduled pad inside a complementary truncation is safe is less obvious. It is safe because any program derived from the unfolding and rescheduling of a pad would have some form of a guard that evaluates to false at the indices of the padded cells. Likewise, pad reshape operators introduce padding since the increased dimension size increases the amount of memory allocated for this tensor, but they do not expand the computational space. These guarded and padded values take on the value of zero because our buffers are zeroed upon allocation. Hence, zero values introduced by a pad or a guard are present exclusively due to the absence of a storage operation being performed at that index. As a result, they are not to be included in the index space to be considered when evaluating reindexer properties or when transforming tensors to flattened heap representations.

## 5.5 Padding

It is not the case that *any* zero value in a program is safe for truncating. A zero value may still have been computed explicitly, so that it still results in a storage access. Therefore, we must be able to make a formal distinction between zero values that were computed and zero values that were introduced by padding at the time of allocation. By making this distinction, we would be able to identify the tensor values that are safe to truncate, since a guarded or pad operator-induced zero value at an index symbolizes that there is no storage being performed at that address. Introducing explicit padding values allows our formalization

to encode the safety property of a program only ever truncating padding. In doing so, we would be able to represent formally and prove the implicit safety properties we had in mind originally when designing the lowering algorithm. Therefore we introduce a new pad type system for statically tracking a conservative estimate of the padding pattern within a tensor computation. Finally, we are able to prove correctness of the lowering algorithm on properly pad-typed programs.

### 5.5.1 Pad Values and Semantics

In order to represent padded zero values, we modify the tensor type so the scalar value can either be a real number or a pad value represented here as unit, ().

$$T := \ list\ T \mid \mathbb{R} \mid ()$$

A pad value should have the same algebraic behavior as a computed zero value. However, we do not need to concern ourselves with tracking pad values produced at the level of scalar expression computations—scalar ATL expressions cannot produce pad values. We focus on the padding produced by language constructs such as the pad operator and the guard. We modify ATL semantics so that pad operators and the false guard generate tensors of padding values rather than simply zeros.

### 5.5.2 Pad-Set Type System

To make this invariant explicit, we introduce the *pad-set type system*, a static analysis that tracks which elements of a tensor may represent padding rather than meaningful data. Under this analysis, each ATL expression is assigned a *pad-set*, written $\pi$, consisting of the multidimensional indices in its result that contain padding values.

Judgments in the pad-set system have the following form.

$$\Gamma_\Pi, v \vdash e : \pi$$

This judgement should be read as stating that the ATL expression e evaluates to a tensor whose padded positions are precisely those described by $\pi$. Here, $\Gamma_\Pi$ represents the pad-set context, mapping tensor identifiers in scope to the pad-sets associated with their values, and $v$ is the valuation of integer program variables used to interpret index expressions.

### 5.5.3 Pad-Set Type Inference

We construct the following set of typing-judgment rules to infer the pad type of an ATL program in a largely syntax-directed manner. The inference rules for the core constructs of the ATL language and its reshape operators are presented in Figure 5.5 and Figure 5.6 respectively. We formally enforce the colloquial constraint of only truncating padding in rules TruncrPad and TruncrPad. The truncation operator only type-checks if the coordinates of elements to be truncated on one of its tensor argument exists in that tensor's pad-set.

$$\textsc{PadSetScatter}\ \frac{\forall\ j, [\![\mathsf{lo}]\!]_v \leq j < [\![\mathsf{hi}]\!]_v \to \Gamma_\pi, v[i \mapsto j] \vdash e : \Pi \qquad \Pi' = [i :: I \mid I \in \Pi \wedge 0 \leq i < [\![n]\!]_v]}{\Gamma_\pi, v \vdash \bigtimes_{i=lo}^{N \leq hi} \{e\}\ : \Pi'}$$

$$\textsc{PadSetGen}\ \frac{[\![\mathsf{lo}]\!]_v < [\![\mathsf{hi}]\!]_v \qquad \Gamma_\pi, v[i \mapsto [\![\mathsf{lo}]\!]_v] \vdash e : \Pi \qquad \Gamma_\pi, v \vdash \boxplus_{i=lo+1}^{hi} e : \Pi' \qquad \Pi'' = [i :: I \mid I \in \Pi \wedge i = 0 \vee (i-1) :: I \in \Pi']}{\Gamma_\pi, v \vdash \boxplus_{i=lo}^{hi} e : \Pi''}$$

$$\textsc{PadSetGenEmpty}\ \frac{[\![\mathsf{hi}]\!]_v \leq [\![\mathsf{lo}]\!]_v}{\Gamma_\pi, v \vdash \boxplus_{i=lo}^{hi} e : \emptyset}$$

$$\textsc{PadSetGuardFalse}\ \frac{\mathsf{shape}\ e = \mathsf{sh} \qquad [\![p]\!]_v = false}{\Gamma_\pi, v \vdash [\ p\ ] \cdot e : ||[\![\mathsf{sh}]\!]_v||}$$

$$\textsc{PadSetGuardTrue}\ \frac{\Gamma_\pi, v \vdash e : \Pi \qquad [\![p]\!]_v = true}{\Gamma_\pi, v \vdash [\ p\ ] \cdot e : \Pi}$$

$$\textsc{PadSetSum}\ \frac{[\![\mathsf{lo}]\!]_v < [\![\mathsf{hi}]\!]_v \qquad \forall j, [\![\mathsf{lo}]\!]_v \leq j < [\![\mathsf{hi}]\!]_v \to \Gamma_\pi, v[i \mapsto j] \vdash e : \Pi}{\Gamma_\pi, v \vdash \sum_{i=lo}^{hi} e : \Pi}$$

$$\textsc{PadSetSumEmpty}\ \frac{[\![\mathsf{hi}]\!]_v \leq [\![\mathsf{lo}]\!]_v \qquad \mathsf{shape}\ e = \mathsf{sh}}{\Gamma_\pi, v \vdash \sum_{i=lo}^{hi} e : ||[\![\mathsf{sh}]\!]_v||}$$

$$\textsc{PadSetBind}\ \frac{\Gamma_\pi, v \vdash e_1 : \Pi_1 \qquad \Gamma_\pi[x \mapsto \Pi_1], v \vdash e_2 : \Pi_2}{\Gamma_\pi, v \vdash \mathsf{let}\ x\ := e_1\ \mathsf{in}\ e_2 : \Pi_2}$$

Figure 5.5: Pad-set type inference rules for core ATL constructs

$$\textsc{PadSetConcat}\ \frac{\mathsf{shape}\ e_1 = \mathsf{n} :: \mathsf{sh}\ \ \mathsf{shape}\ e_2 = \mathsf{m} :: \mathsf{sh} \qquad \Gamma_\pi, v \vdash e_1 : \Pi_1 \qquad \Gamma_\pi, v \vdash e_2 : \Pi_2}{\Gamma_\pi, v \vdash e_1\ \circ\ e_2 : \Pi_1 \cup [(i + [\![\mathsf{n}]\!]_v) :: I \mid i :: I \in \Pi_2]}$$

$$\textsc{PadSetTranspose}\ \frac{\Gamma_\pi, v \vdash e : \Pi}{\Gamma_\pi, v \vdash e^T : [i :: j :: I \mid j :: i :: I \in \Pi]}$$

$$\textsc{PadSetFlatten}\ \frac{\Gamma_\pi, v \vdash e : \Pi \qquad \mathsf{shape}\ e = \mathsf{n} :: \mathsf{m} :: \mathsf{sh}}{\Gamma_\pi, v \vdash \mathsf{flatten}\ e : [(i * [\![\mathsf{m}]\!]_v + j) :: I \mid i :: j :: I \in \Pi]}$$

$$\textsc{PadSetSplit}\ \frac{\Gamma_\pi, v \vdash e : \Pi \qquad \mathsf{shape}\ e = \mathsf{n} :: \mathsf{sh}}{\Gamma_\pi, v \vdash \mathsf{split}\ k\ e : [(i/[\![\mathsf{k}]\!]_v) :: (i\ \mathsf{mod}\ [\![\mathsf{k}]\!]_v) :: I \mid i :: I \in \Pi]}$$

$$\textsc{PadSetPadl}\ \frac{\Gamma_\pi, v \vdash e : \Pi \qquad \mathsf{shape}\ e = \mathsf{n} :: \mathsf{sh}}{\Gamma_\pi, v \vdash \mathsf{pad_l}\ k\ e : [i :: I \mid i - [\![k]\!]_v :: I \in \Pi] \cup ||[\![\mathsf{k}]\!]_v :: [\![\mathsf{sh}]\!]_v||}$$

$$\textsc{PadSetPadr}\ \frac{\Gamma_\pi, v \vdash e : \Pi \qquad \mathsf{shape}\ e = \mathsf{n} :: \mathsf{sh}}{\Gamma_\pi, v \vdash \mathsf{pad_r}\ k\ e : \Pi \cup [i :: I \mid [\![\mathsf{n}]\!]_v \leq i < [\![\mathsf{n}]\!]_v + [\![\mathsf{k}]\!]_v \vee I \in ||[\![\mathsf{sh}]\!]_v||]}$$

$$\textsc{PadSetTruncl}\ \frac{\Gamma_\pi, v \vdash e : \Pi \qquad \mathsf{shape}\ e = \mathsf{n} :: \mathsf{sh} \qquad \forall i\ I,\ i < [\![k]\!]_v \to I \in ||[\![\mathsf{sh}]\!]_v|| \to i :: I \in \Pi}{\Gamma_\pi, v \vdash \mathsf{trunc_l}\ k\ e : [i - [\![\mathsf{k}]\!]_v :: I \mid [\![\mathsf{k}]\!]_v \leq i \wedge i :: I \in \Pi]}$$

$$\textsc{PadSetTruncr}\ \frac{\Gamma_\pi, v \vdash e : \Pi \qquad \mathsf{shape}\ e = \mathsf{n} :: \mathsf{sh} \qquad \forall i\ I,\ [\![\mathsf{n}]\!]_v - [\![k]\!]_v \leq i \to I \in ||[\![\mathsf{sh}]\!]_v|| \to i :: I \in \Pi}{\Gamma_\pi, v \vdash \mathsf{trunc_r}\ k\ e : [i :: I \mid i < [\![\mathsf{n}]\!]_v - [\![k]\!]_v \wedge i :: I \in \Pi]}$$

$$\begin{array}{lrl}
\mathsf{let} & \Pi' & := [(j - i + ([\![\mathsf{n}]\!]_v - 1) :: i - \mathsf{max}\ 0\ ([\![\mathsf{n}]\!]_v - (\mathsf{min}\ [\![\mathsf{n}]\!]_v\ [\![\mathsf{m}]\!]_v) - (\mathsf{max}\ 0\ (j - i + ([\![\mathsf{n}]\!]_v - 1) - (\mathsf{min}\ [\![\mathsf{n}]\!]_v\ [\![\mathsf{m}]\!]_v - 1)))) :: I \\
& & \quad \mid i :: j :: I \in \Pi]\ \mathsf{in} \\
\mathsf{let} & \Pi_\triangle & := [i :: j :: I \mid I \in ||[\![\mathsf{sh}]\!]_v|| \wedge 0 \leq i < [\![\mathsf{n}]\!]_v + [\![\mathsf{m}]\!]_v - 1 \wedge 0 \leq j < \mathsf{min}\ [\![\mathsf{n}]\!]_v\ [\![\mathsf{m}]\!]_v \\
& & \quad \wedge \neg((0 \leq \mathsf{max}\ 0\ ([\![\mathsf{n}]\!]_v - (\mathsf{min}\ [\![\mathsf{n}]\!]_v\ [\![\mathsf{m}]\!]_v) - (\mathsf{max}\ 0\ (i - (\mathsf{min}\ [\![\mathsf{n}]\!]_v\ [\![\mathsf{m}]\!]_v - 1)))) + j - ([\![\mathsf{n}]\!]_v - 1) + i) < [\![\mathsf{m}]\!]_v))]\ \mathsf{in}
\end{array}$$

$$\textsc{PadSetShear}\ \frac{\mathsf{shape}\ e = \mathsf{n} :: \mathsf{m} :: \mathsf{sh} \qquad \Gamma_\pi, v \vdash e : \Pi}{\Gamma_\pi, v \vdash \mathsf{shear}\ e : \Pi' \cup \Pi_\triangle}$$

$$\begin{array}{lrl}
\mathsf{let} & \Pi' & := [(\mathsf{max}\ 0\ ([\![\mathsf{n}]\!]_v - (\mathsf{min}\ [\![\mathsf{n}]\!]_v\ [\![\mathsf{m}]\!]_v) - (\mathsf{max}\ 0\ (i - (\mathsf{min}\ [\![\mathsf{n}]\!]_v\ [\![\mathsf{m}]\!]_v - 1)))) + j \\
& & \quad :: \mathsf{max}\ 0\ ([\![\mathsf{n}]\!]_v - (\mathsf{min}\ [\![\mathsf{n}]\!]_v\ [\![\mathsf{m}]\!]_v) - (\mathsf{max}\ 0\ (i - (\mathsf{min}\ [\![\mathsf{n}]\!]_v\ [\![\mathsf{m}]\!]_v - 1)))) + j - ([\![\mathsf{n}]\!]_v - 1) + i :: I) \\
& & \quad \mid (i :: j :: I) \in \Pi]\ \mathsf{in} \\
\forall i\ j\ I, & & \\
\mathsf{let} & D & := \mathsf{max}\ 0\ ([\![\mathsf{n}]\!]_v - (\mathsf{min}\ [\![\mathsf{n}]\!]_v\ [\![\mathsf{m}]\!]_v) - (\mathsf{max}\ 0\ (i - (\mathsf{min}\ [\![\mathsf{n}]\!]_v\ [\![\mathsf{m}]\!]_v - 1)))) + j - ([\![\mathsf{n}]\!]_v - 1) + i)\ \mathsf{in}
\end{array}$$

$$\textsc{PadSetUnshear}\ \frac{\begin{array}{c} I \in ||[\![\mathsf{sh}]\!]_v|| \to \\ \neg 0 \leq D < [\![\mathsf{m}]\!]_v \to \\ i :: j :: I \in \Pi \end{array} \qquad \mathsf{shape}\ e = \mathsf{n} + \mathsf{m} - 1 :: \mathsf{min}\ \mathsf{m}\ \mathsf{n} :: \mathsf{sh} \qquad \Gamma_\pi, v \vdash e : \Pi}{\Gamma_\pi, v \vdash \mathsf{unshear}\ e : \Pi'}$$

Figure 5.6: Pad-set type inference rules for reshape operators

### 5.5.4 Pad-Set Type Soundness

We prove the following soundness theorem for the pad-set type system in Rocq. At a high level, the theorem establishes that the static approximation computed by pad-set typing is semantically meaningful: every index identified by the analysis as padded is guaranteed to evaluate to a padding value. In other words, the pad-set inferred for an ATL expression soundly approximates the positions in its result that contains padding values.

**Theorem 1** (Pad Type Soundness)**.**

$$\begin{aligned}
&\forall\ v\ \Gamma_\Pi\ e\ \pi, \Gamma_\Pi, v \vdash e : \pi \rightarrow \\
&\forall\ \Gamma\ r, \langle e, v, \Gamma\rangle \Downarrow\ r \rightarrow \\
&\forall\ i,\ i \in \pi \rightarrow \\
&r[\![\ i\ ]\!] = ()
\end{aligned}$$

### 5.5.5 Strengthening the Compiler Correctness Theorem

By distinguishing pad values from standard computed tensor values, we strengthen various definitions of conditions and semantics given previously for the overall compiler-correctness theorem.

#### Tensor to Array Delta

We first revisit the function tensor_to_array_delta. The function of an array delta is to represent a map of tensor values to the flattened integer indices where the computation is meant to be stored. However, a pad value is meant to represent a zero value that was allocated but not actually computed and written. Therefore, we modify our original definition of tensor_to_array_delta to account for the distinction between scalar values and pad values by only mapping scalar values.

#### Injectivity

We similarly redefine our domain required for injectivity of well-formed reindexers. Previously, a well-formed reindexer over a tensor $t$ had to be injective over the entire index space of $t$. With the static knowledge of some distribution of padding values, we can constrain the domain of indices we consider reindexers to be transforming. We redefine the domain for injectivity to be the indices of a tensor $t$ that contain non-padding values.

$$\forall \mathsf{idx}_1, \mathsf{idx}_2.\ \mathsf{idx}_1 \in [\![\|t\|_v]\!] \rightarrow \mathsf{idx}_2 \in [\![\|t\|_v]\!] \rightarrow\ t[\mathsf{idx}_1] \neq () \rightarrow\ t[\mathsf{idx}_2] \neq () \rightarrow \Downarrow_v \theta\ \mathsf{idx}_1 = \Downarrow_v \theta\ \mathsf{idx}_2 \rightarrow \mathsf{idx}_1 = \mathsf{idx}_2 \quad \text{(Injectivity)}$$

### Non-destructive Assignment

We similarly constrain the domain required for non-destructive assignments produced by the reindexer, since we reduced the domain of injectivity.

$$h[o] = \mathsf{arr} \to a = (\mathsf{=}) \to \forall \mathsf{idx}.\ \mathsf{idx} \in [\![\|t\|_v]\!] \to t[\mathsf{idx}] \neq () \to \mathsf{arr}[\Downarrow_v\ \theta\ \mathsf{idx}] = 0$$
(Non-destructivity)

### Context and State Equivalence

We also modify the equivalence property between the functional evaluation context and the stack and heap. The previous definition of equivalence equated any tensor mapped in the functional context to the array delta produced by the application of tensor_to_array_delta. However, this condition no longer applies for our new definition of tensor_to_array_delta since it omits pad values. Although pad values are not present in the array delta, they should have still been present in the heap upon its original allocation. Therefore, the restatement of equivalence shown below adds the array delta produced from $t$ and its original allocation.

$$\Gamma \sim\sim (st, h)\ :=\ \forall x.\ \Gamma[x] = t \to$$
$$(st, h)[x] = \mathsf{tensor\_to\_array\_delta}\ (\lambda i.i)\ t\ \emptyset \uplus \mathsf{tensor\_to\_array\_delta}\ (\lambda i.i)\ (\mathsf{genpad}\ \|t\|)\ \emptyset$$
(Environment Equivalence)

### Correctness Theorem

In addition to the redefinitions stated above, we must also restate our overall correctness theorem to include the precondition that an ATL program must be well-typed within the pad type system.

**Theorem 2** (Strengthened Compiler Correctness)**.**

$$\begin{array}{l}
\langle e, v, \Gamma\rangle \Downarrow\ t \to \\
\Gamma \sim\sim (st, h) \to \\
\mathsf{well_formed_allocation}\ t\ \theta\ st\ h\ o \to \\
\mathsf{well_formed_reindexer}\ \theta\ v\ t\ h\ o\ a \to \\
\Gamma_\Pi, v \vdash e : \pi \to \\
\langle\ (\mathcal{L}\ e\ o\ a\ \theta\ c), v, st, h\rangle\ \Downarrow_C\ (st, h)\ \uplus\ \mathsf{tensor_to_array_delta}\ \theta\ t\ v
\end{array}$$

From here we use this strengthened correctness theorem equipped with all the proper invariants to prove the following top-level correctness theorem, which corresponds to the top-level call to lowering in an empty environment except for the initial allocation in which the tensor computation is to be stored.

**Theorem 3** (Compiler Correctness)**.**

$$\begin{array}{l}
\langle e, \emptyset, \emptyset\rangle \Downarrow\ t \to \\
\vdash e : \pi \to \\
\langle\ (\mathcal{L}\ e\ o\ (=)\ \theta\ \emptyset), v, (\emptyset, \emptyset)[o \mapsto \mathsf{alloc}\ \|e\|_\emptyset]\rangle\ \Downarrow_C\ (\emptyset, \emptyset)[o \mapsto \mathsf{alloc}\ \|e\|_\emptyset]\ \uplus\ \mathsf{tensor_to_array_delta}\ (\lambda i.i)\ t\ \emptyset
\end{array}$$

Note that most of the intricate invariants we defined earlier do not appear in this final theorem statement, so bugs in their statements cannot lead us to accept unsound compilers. For a given program, its concrete typing derivation can be constructed easily enough, and the specific action of tensor_to_array_delta can be computed, so that we derive correct execution of the compiled program without needing to trust either the type system or the auxiliary functions used to describe action on the heap.

## 5.6 Conclusion

In this chapter, we have given a formal account of lowering for ATL and established its correctness in Rocq. To our knowledge, this is the first rigorous proof—mechanized or otherwise—of correctness for a lowering algorithm for a functional tensor language that supports high-performance array code generation through source-level operators that explicitly decouple compute order from storage order. This result shows that the additional scheduling control introduced by ATL's reshape operators can be compiled soundly, without sacrificing the semantic guarantees established at the source level.

A central challenge in this proof is that reshape operators effect storage reorderings by inducing nontrivial transformations of index structure representing the final storage index access expression in the generated C loop-nests. Over the course of lowering, these transformations appear as remappings of storage indices, and correctness depends on showing that these remappings preserve the intended relationship between logical tensor coordinates and their physical realization in memory. To make this precise, the lowering algorithm composes reindexer functions to construct the index structures representing how reshape operations transform storage structure throughout lowering. The correctness proof identifies a key behavioral invariant on these reindexers: they must remain well-formed as they are composed through compilation. This property ensures that the storage-index expressions generated in the imperative target remain consistent with the semantics of the source program.

The proof also exposes the fact that not every well-formed ATL term is safe to lower. Certain reshape operators introduce tensor regions whose values are semantically irrelevant but operationally present, and compilation is only correct when these operators are used in a disciplined way. Specifically, ATL programs are safe if they are expressed using core ATL terms and then scheduled using equivalence-preserving rewrites to introduce reshape operators. To capture this condition explicitly, we developed the pad-set type system, which tracks the propagation of padding through ATL programs and formalizes the invariant required for reshape-based programs to lower soundly. This type system complements the lowering proof by making explicit a semantic property that is latent in the source language but essential to compilation correctness.

Taken together, these results establish that ATL's source-level scheduling model admits a sound compilation strategy. Programs may be transformed algebraically using verified rewrites, lowered through explicit manipulation of compute and storage order, and compiled into imperative code while preserving their functional meaning. This provides the formal foundation needed for the later chapters, where the same perspective on index structure is extended beyond lowering to richer layout and storage abstractions.

# Chapter 6

# Formal Verification of the CUDA Tensor Layout Abstraction and Algebra

Modern GPU architectures are increasingly optimized for tensor-centric computation, driven by the computational demands of machine-learning and scientific-computing workloads. Recent NVIDIA architectures introduce specialized hardware support for tensor operations, including Tensor Cores and specialized instructions for structured movement of tensor data throughout the GPU memory hierarchy [3,72,75]. Efficiently exploiting these architectural features requires careful control over how multidimensional data is mapped to memory and to parallel execution resources.

To address this need, NVIDIA developed CuTe (CUDA Template) layouts, a C++ template library designed to provide abstractions for representing tensor layouts and the transformations required to map tensor computations efficiently onto GPU hardware [36]. At its core, CuTe introduces two central ideas: a hierarchical representation of tensor layouts and an algebra of operations over these layouts. These abstractions enable programmers to express complex data layouts and transformations while separating layout concerns from algorithmic logic.

CuTe represents tensor layouts using hierarchical structures built from nested tuples describing tensor shapes and memory strides [8]. This representation generalizes traditional row-major and column-major layouts by allowing layouts to be constructed compositionally from smaller layouts to describe more complex orderings. This hierarchical structure allows layouts to represent the complex data organizations required by modern GPU instructions, including tiling strategies, memory partitioning, and thread-level data mappings.

In addition to this representation, CuTe defines a collection of algebraic operations over layouts. These operations include composition, concatenation, division, and inversion, among others, and allow programmers to construct new layouts from existing ones. This layout algebra provides a powerful mechanism for expressing common program transformations such as tiling, loop restructuring, and data partitioning in a declarative manner.

A key design goal of CuTe is to make layout transformations compositional and reusable. By treating layouts as first-class mathematical objects rather than implicit properties of arrays, CuTe enables developers to reason about data movement and thread organization independently of the computational kernel itself. This abstraction allows programmers to write tensor kernels in standard, canonical loop forms agnostic over the exact layouts of its

input and output tensors and instead be parameterized over these orderings as CuTe layouts. This separation of concerns is particularly valuable in high-performance GPU programming, where correctness of layout transformations is critical but difficult to reason about due to the interaction between indexing arithmetic, memory-hierarchy constraints, and parallel execution structure.

The CuTe layout abstraction and its algebra function similarly to ATL reindexers and reshape operators in that both are algebraic structures and combinators that construct compositional transformations over index mappings. In ATL, reshape operators and their reindexers act as combinators to manipulate relative compute and storage order in a principled, sound way by compositionally transforming index expressions. In CuTe, layouts represent functions that map logical tensor coordinates to physical memory addresses through compositions of shape and stride transformations. They can be understood as a "view" into a tensor and allow programs to parameterize over the storage orders of their tensor operands. Layout operations such as composition, tiling, and division therefore correspond to algebraic transformations of these index mappings. Similarly, in ATL, reindexing operators induced by reshape transformations act as algebraic transformations on iteration indices, changing how loop coordinates map to tensor accesses while preserving program semantics. From this perspective, reshape operators in ATL and layout operators in CuTe can be seen as dual views of the same underlying concept: ATL reshapes the iteration space of a computation while inducing the corresponding storage-order change to maintain functional equivalence, while CuTe allows programmers to fix the compute order of a kernel and reshapes the coordinate system through which data is addressed. Both systems therefore manipulate tensor programs by transforming index mappings through compositional algebraic operators. This correspondence suggests that tensor layout algebra and tensor program reindexing share a common mathematical foundation as compositional transformations over index spaces, differing primarily in whether the transformations are expressed over computation or data representation. In fact, many of the well-formedness conditions imposed on reindexer transformations induced by ATL reshape operators to formally ensure soundness of their lowering into imperative, loop-nest programs can be applied directly to the soundness of layout operators used in canonical loop-based programs.

This observation motivates the need for formal reasoning about tensor layouts and their algebraic properties. In particular, establishing correctness properties such as layout equivalence, invertibility of layout transformations, and preservation of indexing semantics is important for ensuring that layout transformations preserve program meaning. The algebraic properties of layouts and the transformations they represent very naturally admit mathematical reasoning, and there have been previous informal efforts to characterize their semantics [7]. Despite these advantages, the correctness of layout transformations in CuTe is largely established through testing and manual reasoning. Because these layouts determine how threads access memory and how tensor data is partitioned across hardware resources, errors in layout reasoning can lead to incorrect computation, race conditions, or performance degradation. The complexity of hierarchical layouts and their associated algebra makes informal reasoning about correctness difficult, particularly as layouts are composed and transformed through multiple operations. Moreover, the soundness of applying certain layout-algebra operators and the well-formedness of the resulting layout structures is contingent on certain properties of the layouts' operands.

In this work, I developed Verified CUDA Template layouts, or VeriCuTe layouts, and focused specifically on the formal verification of the CuTe layout representation and its associated layout algebra. Much of the formalization was developed from the contributions and documentation published by Cecka [8] on programming with CuTe layouts and their algebraic operators. The contributions described in this chapter include:

- A generalized, formal model of the CuTe layouts and their semantics
- A formalization of key layout-algebra operations used in CUDA tensor programs
- Proofs of correctness properties of certain common layout operations
- Proofs of correctness for the usage of layout permutations in simple, canonical CUDA programs and the required preconditions for their soundness

## 6.1 CUTLASS and CuTe Layouts

CUTLASS is NVIDIA's high-performance C++ template library for implementing dense linear algebra and tensor kernels on GPUs. It provides a collection of reusable components designed to map tensor computations efficiently onto the GPU execution hierarchy, including thread blocks, warps, and specialized units such as Tensor Cores. Rather than presenting GPU kernels as monolithic handwritten CUDA code, CUTLASS decomposes them into a hierarchy of abstractions for tiling, memory movement, and instruction-level computation. This structure allows kernels to be specialized to particular architectures and problem sizes while retaining a uniform programming model.

At a high level, a CUTLASS kernel is obtained by decomposing a logical tensor operation into a hierarchy of tiles aligned with the GPU's execution and memory hierarchy. For example, matrix multiplication is expressed by partitioning the iteration space into thread-block tiles, then warp tiles, and finally instruction tiles corresponding to the underlying hardware primitive. Each level of this hierarchy determines how data is partitioned, moved, and accessed.

In traditional C kernels, these decisions are encoded through a combination of template parameters, iterator abstractions, and explicit index arithmetic. To illustrate, consider again the standard dense matrix multiplication written in C:

```
for (int i = 0; i < N; i++)
  for (int j = 0; j < M; j++)
    for (int k = 0; k < K; k++)
      out[i * M + j] += m1[i * K + k] * m2[k * M + j]
```

In CUTLASS, this same computation is expressed in terms of tensor layouts that define how logical coordinates map to memory. Using CuTe layout abstractions, this computation can be written as:

```
for (int i = 0; i < size<0>(Lout); i++)
  for (int j = 0; j < size<1>(Lout); j++)
    for (int k = 0; k < size<1>(Lm1); k++)
      Lout(i,j) += Lm1(i,k) * Lm2(k,j);
```

Here, raw index arithmetic has been replaced by applications of layout objects. The expressions `Lm1(i,k)`, `Lm2(k,j)`, and `Lout(i,j)` interpret coordinate tuples according to their associated layouts, rather than directly computing flattened indices. Similarly, loop bounds are derived from the shapes of these layouts rather than standalone size variables. This formulation separates the computational structure from the details of memory organization. A CUTLASS library routine is typically parameterized by tensor layouts, allowing a single implementation to be specialized over the storage orderings of its input operands as well as the desired layout of the output.

Layouts themselves can be constructed explicitly. The following definitions produce the same row-major indexing behavior as the original C program:

```
auto m1_layout = make_layout(make_shape(N,K),make_stride(K,1));
auto m2_layout = make_layout(make_shape(K,M),make_stride(M,1));
auto out_layout = make_layout(make_shape(N,M),make_stride(M,1));

auto Lm1 = make_tensor(make_smem_ptr(m1), Lm1_layout);
auto Lm2 = make_tensor(make_smem_ptr(m2), Lm2_layout);
auto Lout = make_tensor(make_smem_ptr(out), out_layout);
```

The make_layout function constructs a layout from two components: a shape and a stride. The shape specifies the ordered extents of the tensor, while the stride determines how coordinates advance through memory. In this example, the chosen shapes match the dimensions implied by the loop bounds of the original program, and the strides implement the same row-major traversal.

The make_tensor function pairs this layout with a pointer to the underlying data buffer, defining how the buffer is interpreted as a logical tensor during computation. A layout maps a multi-dimensional coordinate within its shape to a linear offset by multiplying each coordinate by its corresponding stride and summing the results.

For example, the layout associated with the m1 tensor results in the following coordinate-index translation. Note that this expression is the same as the index-access expression into the raw array m1 in the C program.

$$\mathsf{Lm1}(i,j) = (i,j) \circ (K,1) = i * K + j$$

A key advantage of this representation is that the computation itself need not change to adapt to new inputs with different storage orders. Now to adapt this kernel to compute and output matrices in column-major order, we do not have to rewrite the core procedure itself. Instead, we can construct tensors with the following CuTe layouts to be used as program parameters instead. These layouts have the same shape as the row-major order one but different strides that result in a column-major stride.

```
auto m1_layout = make_layout(make_shape(N,K),make_stride(1,N));
auto m2_layout = make_layout(make_shape(K,M),make_stride(1,K));
auto out_layout = make_layout(make_shape(N,M),make_stride(1,N));
```

Under this layout, the same coordinate access now evaluates to the following expression without having to modify the computation itself.

$$\mathsf{Lm1}(i,j) = (i,j) \circ (1,N) = j * N + i$$

The examples above illustrate the key structural property of CuTe that motivates our formalization: tensor layouts are explicit mappings from logical coordinates to physical storage, and computation interacts with them only through their application. This perspective closely parallels the role of reindexing functions in ATL, where transformations of storage are expressed as mappings between index spaces. In both settings, changes to data layout can be understood as composing such mappings rather than rewriting the computation itself.

This observation suggests viewing layouts not merely as implementation artifacts, but as elements of a compositional algebra of index transformations. In the remainder of this chapter, we make this connection precise by formalizing the layout abstraction and their interpreted semantics as functions over index spaces and layout operators as algebraic transformations on these functions. This formulation allows us to reason about layout composition in the same style as reindexer composition in ATL and to establish correctness properties for programs parameterized by layouts. Moreover, by formalizing CuTe layouts and proving correctness of their usage in CUTLASS programs, this framework also supports reasoning about the safety of layout-driven optimizations, ensuring that such transformations preserve well-formed memory accesses and do not introduce invalid indexing behavior.

## 6.2 Formalizing CuTe Layouts

We now present a formal account of the CuTe layout abstraction. A layout is represented as a pair of hierarchical tuples. Unlike standard tuples in functional programming, which are polymorphic and allow for a different type in each tuple entry, this definition of hierarchical tuples supports only one terminal type at its nodes. These structures are analogous to binary trees. Its inductive definition is shown below.

```
Inductive Tuple {X} := Tup (l r : Tuple X) | Singleton (x : X)
```

Hierarchical tuples are considered to be *congruent* if they have the same tree structure. Relatedly, hierarchical tuples are considered to be *compatible* if the tree structure of one tuple is subsumed within the tree structure of the other starting from the root.

In a layout, the first element of the pair is an integer hierarchical tuple that represents the *shape* of a tensor. The second element in this pair represents the *stride*, which encodes how this tensor is to be traversed. The layout's stride is polymorphic and can be parameterized and interpreted meaningfully as any type that has certain algebraic properties that we will discuss later in Section 6.6. In a well-formed layout, the shape and stride tuples must be congruent. The following is the definition of the Layout type parameterized on a type X.

```
Layout {X} := Tuple Z * Tuple X
```

For now, we consider an integer stride of type Z, as a layout configuration with this type instantiation is the most common form of layout used. When paired together, the shape and stride implement a mapping from any higher-dimensional coordinate within the domain specified by the shape to a single, integer index via the stride.

## 6.3 Interpreting Layouts

Similar to the integer-expression-tuple list structure used within the ATL compiler to represent unevaluated index expressions, layouts can also be seen as an uninterpreted structure meant to represent an index mapping function. In order to use layouts as functions to perform coordinate-tuple index mapping into integer offsets, we must define how to interpret a layout functionally.

### 6.3.1 Converting Between Hierarchical Tuples and Lists

To begin, we define the rank helper function below. This function returns the number of dimensions in a hierarchical tuple and corresponds to the sum of the heights of each branch in the tuple tree structure.

```
Fixpoint rank s :=
  match s with
  | Singleton _ ⇒ 1
  | Tup s1 s2 ⇒ rank s1 + rank s2
  end.
```

Next, we define the helper function ind_crd_list. This function takes a shape represented as a list of integers and an integer index and lifts this index into the higher-dimensional coordinate representation in this shape. This higher-dimensional coordinate is also represented as a list of integers.

```
Fixpoint idx_crd_list s i :=
  match s with
  | [] ⇒ []
  | [n] ⇒ i
  | n::ns ⇒ i % n :: idx_crd_list ns (i / n)
  end.
```

The following two helper functions tuple_to_list and list_to_tuple convert between list and hierarchical-tuple representations. The tuple_to_list function returns the flattened list representation of the entries in the hierarchical tuple in a left-to-right traversal order. The list_to_tuple function takes in a list to be reconstructed into a tuple as well as a reference tuple with the target hierarchical tree structure to be reconstructed. It is defined using the rank helper function. If the length of the list matches the rank of the tuple, it will return a congruent hierarchical tuple with the entries of the input list.

```
Fixpoint tuple_to_list s :=
  match s with
  | Singleton x ⇒ [x]
  | Tup s1 s2 ⇒ tuple_to_list s1 ++ tuple_to_list s2
  end.

Fixpoint list_to_tuple s l :=
  match s with
  | Singleton _ ⇒ match l with
                  | x::_ ⇒ Singleton x
                  | _ ⇒ Singleton -1
                  end
  | Tup s1 s2 ⇒ Tup (list_to_tuple s1 (firstn (rank s1) l))
                    (list_to_tuple s2 (skipn (rank s1) l))
  end.
```

### 6.3.2 Converting Between Indices and Coordinates

Next, we can define the helper function idx_crd that takes an integer index and a shape represented as a hierarchical tuple and lifts it into a congruent hierarchical tuple representing the higher-dimensional coordinate equivalent of the index within the tensor shape given.

```
Definition idx_crd s i :=
  let crd_list := idx_crd_list 2 i in
  list_to_tuple 2 crd_list.
```

The following helper function inner_prod computes the inner product between two congruent hierarchical integer tuples. The products of the corresponding leaf nodes are summed. In the case that the tuples are not congruent, a value of zero is returned. However, in practice, this function is a helper function, and we only expect it to be invoked on congruent tuples.

```
Fixpoint inner_prod s c :=
  match s,c with
  | Tup s1 s2,Tup c1 c2 ⇒ (inner_product s1 c1) + (inner_product s2 c2)
  | Singleton s', Singleton c' ⇒ s' * c'
  | _,_ ⇒ 0
  end.
```

### 6.3.3 Interpreting Layouts on Coordinates

The function lift_crd defined below takes two tuples representing a shape and a coordinate. This coordinate must be compatible with the shape. It returns a new coordinate tuple congruent to the shape by recursing to the leaves of the coordinate and lifting those integer arguments into coordinates matching the remaining shape substructural elements.

```
Fixpoint lift_crd s c :=
  match s,c with
  | Tup s1 s2, Tup c1 c2 ⇒ Tup (lift_crd s1 c1) (lift_crd s2 c2)
  | Tup s1 s2, Singleton c' ⇒ idx_crd s1 c'
  | _,_ ⇒ c
  end.
```

Finally, these helper functions are used to define the interpret function that takes a layout and applies it to a coordinate tuple. This coordinate need not be congruent to the shape and stride in the layout, but simply compatible. This coordinate is lifted into a congruent coordinate tuple in the domain of the shape of the layout by using lift_crd. The inner product of this coordinate tuple and the layout stride is then returned as the ultimate integer index output.

```
Definition interpret L crd := let (s,d) := L in
                              let crd' := lift_crd s crd in
                              inner_prod crd' d.
```

## 6.4 Layout-Operator Algebra

In this section, we develop a formal account of CuTe layout operators and their semantics. Building on the view of layouts as mappings from logical index spaces to physical storage, we focus on a core subset of operators that are central to practical layout construction, including concatenation, coalescing, and composition.

Our goal is to make precise how these operators act on layouts and to establish that their implementations correspond to their intended functional behavior. To this end, we define a function to realize each operator and prove soundness theorems relating these definitions to their semantic interpretations. Moreover, this formalization enables precise reasoning about the structural and behavioral conditions required of layout operands, ensuring that these operators produce new layouts that preserve certain well-formedness and correctness properties such as injectivity. Together, these results characterize a compositional fragment of the CuTe layout algebra and provide a foundation for reasoning about layout transformations in subsequent sections.

### 6.4.1 Concatenate

The concatenate operator constructs a larger layout by joining together a sequence of sublayouts by constructing larger tuples of each operands' shape and stride. Rather than defining a new indexing scheme from scratch, concatenation combines layouts by allowing each component layout to contribute independently to the resulting coordinate mapping. This operator provides a modular mechanism for assembling complex layouts from simpler constituent pieces while preserving the structure of the underlying sublayouts. The definition for this operator is shown below.

```
L1 ++ L2 := let (s1,d1) := L1 in
            let (s2,d2) := L2 in
            (Tup s1 s2, Tup d1 d2).
```

Conceptually, concatenation treats a layout as being built from multiple coordinate components, each interpreted according to its associated sublayout. The resulting physical offset is then determined collectively from these component mappings. This compositional structure makes the operator useful for expressing hierarchical layouts and for reasoning about layout structure in terms of smaller, independently understood parts.

To establish that concatenation behaves consistently with its intended interpretation, we prove the following soundness theorem relating the operational definition of the operator to its functional semantics. The theorem shows that the concatenated layout computes offsets by combining the application of its constituent sublayouts. As a result, reasoning about the behavior of a concatenated layout reduces directly to reasoning about the behaviors of its components. This property is fundamental to the compositional structure of the layout algebra and allows larger layout constructions to inherit correctness from their constituent parts.

$$\mathsf{interpret}\ (L_1{+}{+}L_2) = \lambda(c_1, c_2).\ \mathsf{interpret}\ L_1\ c_1 + \mathsf{interpret}\ L_2\ c_2$$

### 6.4.2 Coalesce

Since CuTe layouts represent coordinate-index matching functions, it makes sense that there can be many layout structures that represent the same logical function. The coalesce function is a unary layout operator that performs a simplification on a given layout without changing its interpreted functional semantics operating on integer inputs. This operation is particularly useful for normalizing layouts and exposing opportunities for further optimization.

This operator simplifies a layout by merging adjacent modes when they collectively describe a contiguous region of memory. Intuitively, coalescing eliminates redundant structure in a layout while preserving its induced mapping from logical coordinates to physical offsets.

We begin by observing that certain modes contribute no meaningful structure to the layout. In particular, any mode of size one always produces the coordinate value zero, regardless of its stride. As a result, such modes do not affect the computed offset and may be safely ignored. This simplification corresponds to eliminating degenerate dimensions that carry no indexing information.

More generally, consider two adjacent dimensions of a layout, written as $(s_0 : d_0)$ and $(s_1 : d_1)$, where $s_i$ denotes the size of the mode and $d_i$ its stride. The coalescing operation attempts to replace this pair with a single equivalent dimensions. There are several cases to consider. If either mode has size one, it may be removed without affecting the layout, leaving only the other mode. Otherwise, if the stride of the second mode satisfies $d_1 = s_0 \times d_0$, then the two modes describe contiguous traversal in memory and may be combined into a single mode $(s_0 \times s_1 : d_0)$. In all other cases, the modes cannot be merged and must be retained as distinct dimensions.

The full coalescing procedure applies this binary operation iteratively over the sequence of modes in a layout. The functional definition of this operator is given below.

```
coalesce L := let (S,D) := L in
              match S,D with
              | Tup S1 S2, Tup D1 D2 =>
                let l1 := coalesce (S1,D1) in
                let l2 := coalesce (S2,D2) in
                match l1,l2 with
                | _,(Singleton 1, Singleton _) => l1
                | (Singleton 1, Singleton),_ => l2
                | (Singleton s1, Singleton d1),(Singleton s2, Singleton d2) =>
                  if (d2 == s1 ⊛ d1)
                  then (Singleton (s1 * s2),Singleton d1)
                  else l1 ++ l2
                | _,_ => l1 ++ l2
                end
              | _,_ => L
              end.
```

Importantly, this transformation preserves the semantics of the layout: the resulting layout induces the same mapping from logical coordinates to physical indices as the original. In this sense, coalesce acts as a canonicalization step that simplifies layout structure without altering its behavior. We prove the following soundness theorem describing this property.

$$\mathsf{interpret}\ (\mathsf{coalesce}\ L) = \mathsf{interpret}\ L$$

### 6.4.3 Compose

Functional composition is the central operation of the CuTe layout algebra and forms the basis of many higher-level layout transformations. Intuitively, composition constructs a new layout by applying one layout within the coordinate space defined by another. Through this mechanism, complex storage organizations can be assembled compositionally from simpler layouts while preserving a uniform interpretation in terms of coordinate mappings.

Since a layout may itself be viewed as a composition of sublayouts, composition distributes across concatenated substructures in a natural way. In particular, when the inner layout is injective, composing a layout with a concatenation of sublayouts is equivalent to composing it independently with each constituent component. In this sense, composition induces a strided projection of the first layout operand. This observation allows reasoning about composition to proceed recursively over layout structure and reduces many cases to the composition of layouts with integral shapes and strides.

The composition operator is defined below as the infix operator $\circ$. First, the structure of the outer layout is transformed to represent traversal at the granularity induced by the stride of the inner layout. Operationally, this process corresponds to progressively factoring dimensions of the outer layout according to the stride value, producing a layout that visits only the appropriate subset of coordinates. Second, the resulting layout is restricted so that its domain matches the shape of the inner layout. Together, these operations construct a

layout whose coordinate mapping corresponds precisely to applying the outer layout to the offsets generated by the inner layout.

```
Fixpoint div_layout_rec k L :=
  let '(S,D) := L in
  match S,D with
  | Singleton x, Singleton n =>
      if ((k mod x == 0) || (x mod k == 0))
      then Some (k / gcd k x,Singleton (x / gcd k x),Singleton (gcd k x * n))
      else None
  | Tup x y, Tup a b =>
    match div_layout_rec k (x,a) with
    | None => None
    | Some (k',x',a') =>
      match div_layout_rec k' (y,b) with
      | None => None
      | Some (k'',y',b') => Some (k'',Tup x' y',Tup a' b')
      end
    end
  | _,_ => None
  end.

Definition div_layout k L :=
  match div_layout_rec k L with
  | Some (_,s,d) => Some (s,d)
  | None => None
  end.

Fixpoint mod_layout_rec k L :=
  let '(S,D) := L in
  match S,D with
  | Singleton x, Singleton n =>
    if (x <=? k)
    then if (k mod x == 0)
      then Some (k / x,Singleton x,Singleton n)
      else None
    else Some (0,Singleton k,Singleton n)
  | Tup x y, Tup a b =>
    match mod_layout_rec k (x,a) with
    | None => None
    | Some (k',x',a') =>
      if (k' == 0) then Some (0,x',a')
      else match mod_layout_rec k' (y,b) with
           | None => None
           | Some (k'',y',b') => Some (k'',Tup x' y',Tup a' b')
```

```
            end
    end
  | _,_ => None
  end.

Definition mod_layout k L :=
  match mod_layout_rec k L with
  | Some (_,s,d) => Some (s,d)
  | None => None
  end.

Fixpoint L1 ∘ L2 :=
  let '(as_,ad_) := L1 in
  let '(bs_,bd_) := L2 in
  match as_,ad_,bs_,bd_ with
  | Singleton s1, Singleton d1,
    Singleton s2, Singleton d2 =>
      Some (Singleton s2,Singleton (d1 * d2))
  | _,_,Tup s1 s2,Tup d1 d2 =>
    match (as_,ad_) ∘ (s1,d1),(as_,ad_) ∘ (s2,d2) with
    | Some l1, Some l2 => Some (l1 ++ l2)
    | _,_ => None
    end
  | _,_, Singleton s2, Singleton d2 =>
    match div_layout d2 (as_,ad_) with
    | None => None
    | Some divL => let '(as',ad') := coalesce divL in
      match mod_layout s2 (as',ad') with
      | None => None
      | Some modL => Some (coalesce modL)
      end
    end
  | _,_,_,_ => None
  end.
```

---

Note that this operator is defined as a partial function, since not all layouts are composable. The transformation relies on divisibility conditions relating layout shapes and strides to ensure that coordinate traversal remains well-defined. CuTe enforces these conditions statically whenever possible. Assuming these conditions are met and the partial function returns a layout, we can prove the following theorem for decomposing the interpretation of composed layouts.

$$\mathsf{interpret}\ (L_1 \circ L_2) = (\mathsf{interpret}\ L_1)\ .\ (\mathsf{interpret}\ L_2)$$

## 6.5 Loop Transformations Using CuTe Layouts

In this section, we examine how CuTe layouts are commonly used to optimize and transform loop-nest-based tensor kernels in CUTLASS. Rather than treating layouts merely as descriptions of memory organization, CUTLASS uses them as active program parameters that shape iteration structure, index computation, and data movement throughout a kernel. Many optimizations in CuTe are therefore expressed not by rewriting the computation itself but by transforming the layouts through which tensor operands are accessed.

### 6.5.1 Loop-Nest Program Semantics

To reason formally about the correctness of these transformations, we introduce a simplified semantic model for loop-nest tensor programs. Our goal is not to formalize the entirety of the CUDA C execution environment targeted by CUTLASS and CuTe but rather to isolate the aspects of execution relevant to layout-based optimization. In particular, we focus on how layouts determine the mapping between logical tensor coordinates and physical memory accesses within imperative loop nests. Accordingly, we adopt a restricted model of program execution that captures tensor iteration, indexing behavior, and memory updates while abstracting away lower-level details of the CUDA execution model such as thread scheduling, synchronization, and hardware-specific behavior. Although simplified, this model is sufficient for expressing the correctness conditions relevant to our development and for establishing soundness properties of optimizations formulated in terms of layout transformations.

We model programs as functions taking and returning flattened arrays representing the output buffer that is allocated. An array is represented simply as a list of real numbers. Since not all such accesses by index expressions in the program are within the bounds of the array, we model this behavior using the option monad. We define the function for array update below.

```
Fixpoint update_rec arr i x :=
  match i, arr with
  | 0, _::arr' => ret (x::arr')
  | S i', a::arr' => bind (update_rec arr' i' x) (fun x => ret (cons a x))
  | _, _ => None
  end.

Definition update i x arr :=
  match i with
  | Z.neg _ => None
  | _ => update_rec arr (Z.to_nat i) x
  end.
```

From here on, we will adopt the imperative notation arr[i] = x; for an invocation of update i x arr.

We model for-loops as functional folds over the range of input indices with the array being written to being passed along as the accumulator argument. We also include in our model of loop programs a predicate, corresponding to an if-statement guarding the body of a loop nest.

This model of program definition is defined below as the function loop_nest. Our program model is parameterized on the loop bounds represented as lo and hi, the predicate expression pred which is defined as a function of an index, and the function representing the loop's body f which is a function of both the current index and the current state of the output array buffer.

```
Definition loop_nest lo hi pred f := fold_left (fun acc i => if pred i
                                                  then (f i acc)
                                                  else acc) (map Z.of_nat (seq lo (hi-lo))).
```

From here on, we will adopt the imperative notation for (int i = lo; i < hi; ++i) { if pred then f } for an invocation of loop_nest lo hi pred f. With these definitions and notations, we can model the semantics of the following program functionally. This structure will primarily be the form of program we are considering when formally examining layout-based optimizations.

```
for (int i = lo; i < hi; ++i) {
  if (pred i) {
    out[ f i ] += e i;
  }
}
```

### 6.5.2 Programming with Layouts

Using the program model we established, we can model the case where the predicate, output storage index, and body function are functions of the iterating index as mediated through some layouts. Note that these three functions need not use the same layout.

```
for (int i = lo; i < hi; ++i) {
  if (pred (interpret H i)) {
    out[ interpret F i ] += e (interpret G i);
  }
}
```

From here, a common mode of usage is to optimize this program by introducing a permuting layout $P$ and composing it with the existing layouts used in the program. This optimization not only introduces the permuting layout but also allows us to modify the loop bounds, since we can take advantage of the transformed predicate to maintain program equivalence. This approach allows us to reuse the same kernel structure and implement the same procedure but computing and outputting an optimized tensor layout.

```
for (int i = lo'; i < hi'; ++i) {
  if (pred (interpret (H ∘ P) i)) {
    out[ interpret (F ∘ P) i ] += e (interpret (G ∘ P) i);
  }
}
```

From the functional-soundness property we proved on the compose operator, we can decompose the composition and rewrite the program into the following form.

```
for (int i = lo'; i < hi'; ++i) {
  if (pred (interpret H (interpret P i))) {
    out[ interpret F (interpret P i) ] += e (interpret G (interpret P i));
  }
}
```

Now our goal to prove correctness of the program optimization induced by introducing the permuting layout and the preservation of its functional semantics reduces to proving the following equivalence.

```
for (int i = lo; i < hi; ++i) {
  if (pred (interpret H i)) {
    out[ interpret F i ] +=
      e (interpret G i);
  }
}
```

=

```
for (int i = lo'; i < hi'; ++i) {
  if (pred (interpret H (interpret P i))) {
    out[ interpret F (interpret P i) ] +=
      e (interpret G (interpret P i));
  }
}
```

The conditions for this equivalence to hold concern the functional properties of the interpreted layouts. Since we are reasoning about semantic preservation under composition of interpreted layouts viewed as index-transformation functions, many of the soundness conditions identified in our formalization of the ATL lowering algorithm for ensuring well-formed reindexers carry over directly to this setting. In particular, the required properties of the permutation layout correspond to constraints on its range, together with relaxed forms of surjectivity and injectivity.

### Range Conditions

Another condition restricts the range of the permuting layout $P$ over the new loop-nest iterating domain to be in the domain of the original loop-nest bounds. This ensures that introducing the new permuting layout and loop bounds does not result in the program computing over and writing any new values.

$$\forall i.\ \mathsf{lo}' \leq i < \mathsf{hi}' \wedge \mathsf{pred}\ \ (\mathsf{interpret}\ \mathsf{P}\ \ \mathsf{i})\ \ \rightarrow \mathsf{lo} \leq (\mathsf{interpret}\ i) < \mathsf{hi} \wedge \mathsf{pred}\ i). \qquad \text{(RangeRestriction)}$$

### Surjectivity Conditions

One condition pertains to the surjectivity of the permuting layout $P$. It must be surjective from the index range between the loop bounds into the range of indices between the old loop bounds for which the predicate pred holds true. This property means that by introducing the permuting layout we are not reducing the domain of index values the program computation is being applied to.

$$\forall i.\ \mathsf{lo}' \leq i < \mathsf{hi}' -> \exists\ j.\ j \in \mathsf{filter}\ \mathsf{pred}\ [\mathsf{lo}, \mathsf{hi}) \wedge \mathsf{interpret}\ \mathsf{P}\ j = i. \qquad \text{(Surjectivity)}$$

**Injectivity Conditions**

Another condition on the soundness of the permutation requires that the permuting layout must be injective over the new iterating domain.

$$\begin{aligned}\forall i,\ j.\ &\mathsf{lo}' \le i < \mathsf{hi}' \wedge \mathsf{pred}\ \ (\mathsf{interpret}\ \mathsf{P}\ \ \mathsf{i}) \ \rightarrow \\ &\mathsf{lo}' \le j < \mathsf{hi}' \wedge \mathsf{pred}\ \ (\mathsf{interpret}\ \mathsf{P}\ \ \mathsf{j}) \ \rightarrow \\ &\mathsf{interpret}\ \mathsf{P}\ i = \mathsf{interpret}\ \mathsf{P}\ j \rightarrow \\ &i = j. \end{aligned} \qquad \text{(Injectivity)}$$

## 6.6 Polymorphic Strides and Semimodule Structure

While the definitions above present layouts in terms of integer-valued strides, this choice is not fundamental. The construction of a layout depends only on the ability to combine coordinates with strides through addition and scalar multiplication. This observation allows layouts to be defined over a more-general class of stride domains, provided they support the necessary algebraic structure.

In particular, we can generalize layouts by allowing strides to range over a type equipped with the structure of a semimodule. Under this interpretation, a layout maps a coordinate vector to a value in the stride domain by forming a linear combination of stride components weighted by the coordinates. This generalization makes explicit that layouts are not inherently tied to integer indexing, but instead capture a broader class of structured mappings between index spaces.

### 6.6.1 Stride Function Interface

The interface of functions that an integer semi-module of some type $X$ must provide is shown below. An integer semi-module must provide a binary operator $\oplus$ that is a generalization of the scalar-addition operator. Similarly, the $\circledast$ operator is a generalization of scalar multiplication, taking in an integer and an element of type $X$. A stride must also provide operators generalizing less-than-or-equal, division and modulo, and congruence as well.

$$\begin{gathered}\oplus : X \rightarrow X \rightarrow X \\ \circledast : Z \rightarrow X \rightarrow X \\ \leq : X \rightarrow X \rightarrow \mathsf{bool} \\ \mathsf{divmod} : Z \rightarrow X \rightarrow X \rightarrow X * X \\ \cong : X \rightarrow X \rightarrow \mathsf{bool}\end{gathered}$$

Figure 6.1: Integer semi-module and stride function interface

All previous function and operator definitions can be generalized to this parametric stride type by replacing scalar addition and multiplication with $\oplus$ and $\circledast$ respectively, as well as generalizing the computation of a greatest common divisor using the interface-provided definition of divmod.

### 6.6.2 Integer Semi-Module Congruence Properties

The provided definitions for congruence as well as for each integer semi-module operator must satisfy the following algebraic properties.

$$\textsc{CongruentReflexive}\ \frac{}{a \cong a}$$

$$\textsc{CongruentTransitive}\ \frac{a \cong b \qquad b \cong c}{a \cong c}$$

$$\textsc{CongruentSymmetric}\ \frac{a \cong b}{b \cong a}$$

$$\textsc{MulCongruent}\ \frac{a \cong b}{x \circledast a \cong b}$$

$$\textsc{BinCongruent}\ \frac{a \cong c \qquad b \cong c}{a \oplus b \cong c}$$

Figure 6.2: Properties of congruence involving integer semi-module operators

### 6.6.3 Integer Semi-Module Properties

Finally, the stride type and its operators must follow the properties defining the algebraic structure of integer semi-modules.

$$\textsc{DivModSound}\ \frac{\mathsf{divmod}\ x\ a = (q, r) \qquad 0 < x}{r \oplus (x \circledast q) = a}$$

$$\textsc{BinComm}\ \frac{}{a \oplus b = b \oplus a}$$

$$\textsc{BinAssoc}\ \frac{}{a \oplus b \oplus c = a \oplus (b \oplus c)}$$

$$\textsc{BinMulId}\ \frac{a \cong b}{(0 \circledast a) \oplus b = b}$$

$$\textsc{MulId}\ \frac{}{1 \circledast a = a}$$

$$\textsc{MulAssoc}\ \frac{}{x \circledast y \circledast a = (x \times y) \circledast a}$$

$$\textsc{MulBinDistr}\ \frac{}{x \circledast (a \oplus b) = x \circledast a \oplus x \circledast b}$$

$$\textsc{AddMulDistr}\ \frac{}{(x + y) \circledast a = x \circledast a \oplus y \circledast a}$$

Figure 6.3: Integer semi-module properties for stride

### 6.6.4 Non-Integer Stride-Type Instantiations

A consequence of the polymorphic treatment of layout stride types is that layouts may be instantiated over coordinate-valued strides rather than only integer offsets. That is, a stride can have type Tuple X where X satisfies the properties outlined in the previous sections, where X itself is the type Tuple Z. While the preceding sections focused primarily on the standard integer interpretation corresponding to physical memory indexing, the same layout structure can also be interpreted as producing coordinates. This alternative instantiation makes it possible to construct *predicate tensors*, which are commonly used in CUTLASS kernels to guard accesses to partially filled tiles.

The basic idea behind predication in CuTe is to apply the same layout transformations used to partition a tensor operand to an identity layout over the tensor's coordinate space. Rather than producing physical offsets into memory, the transformed identity layout produces the logical coordinates associated with each position in the tiled structure. These coordinates can then be compared against the bounds of the original tensor to determine whether a given access is in-bounds. The resulting Boolean tensor is used to predicate memory accesses and avoid invalid reads or writes in edge tiles.

Importantly, this functionality does not require any additional layout machinery beyond the polymorphic formulation already developed. Once layouts are generalized over arbitrary stride domains satisfying the required algebraic structure, coordinate-valued layouts arise naturally as another valid instantiation. Although predication is not the primary focus of the present formalization, this application illustrates the flexibility of the abstraction and

demonstrates that the same verified layout operators can support richer indexing structures beyond ordinary integer-address computation.

## 6.7 Conclusion

This chapter developed a formal account of CuTe layouts as compositional, algebraic objects that map structured index spaces to structured outputs and established a foundation for reasoning about their correctness. Starting from the concrete role of layouts in CUTLASS kernels, we abstracted tensor layouts into hierarchical tuples equipped with shape and stride structure and defined their semantics as index-space mappings via a functional interpretation function. This allowed us to move from an implementation-centric view of layouts to a denotational model in which layout transformations can be reasoned about mathematically.

On top of this semantic foundation, we introduced a core fragment of the CuTe layout algebra, including concatenation, coalescing, and composition, and proved key soundness properties showing that these operators preserve or appropriately transform the interpretation of layouts. In particular, we showed that layout composition corresponds to functional composition of index mappings, while coalescing preserves semantic equivalence despite structural simplification. These results establish that common layout transformations used in practice in CUTLASS kernels admit compositional reasoning principles and are not merely syntactic rewrites.

We then extended this formalism to a simplified loop-nest model of tensor programs, in which layouts parameterize indexing, predication, and memory-access patterns. Within this model, we characterized a class of layout-driven program transformations, in particular those induced by introducing permuting layouts, and identified the structural conditions—surjectivity, domain preservation, and injectivity—required for semantic preservation. This work connects layout transformation directly to correctness of imperative tensor code, rather than treating layouts as an external optimization artifact.

Finally, we generalized the stride component of layouts from integer offsets to arbitrary semimodule structures. This polymorphic view separates the algebraic role of strides from their physical interpretation as memory addresses, revealing layouts as abstract coordinate transformers that can be instantiated in multiple semantic domains. In particular, we showed that coordinate-valued instantiations arise naturally and enable the construction of predicate tensors, demonstrating that the same verified algebra supports both memory addressing and structured control-flow mechanisms such as predication.

Overall, this work establishes CuTe layouts as a formally well-founded algebra of index transformations, unifying data-layout manipulation and program reindexing under a shared mathematical structure. Beyond providing correctness guarantees for a practical subset of CUTLASS-style optimizations, this perspective suggests a broader principle: that high-performance tensor programming can be understood as computation over compositional index-space morphisms. This work opens the door to future work on extending verification to richer fragments of the CuTe algebra, integrating these results with compiler transformations, and further exploring the connection between layout algebras and reindexing systems such as ATL as dual presentations of the same underlying semantics.

# Chapter 7

# Verified Sparse-Tensor Optimizations via an Abstract Format Algebra

Up to this point, the tensor programs considered in this dissertation have followed the predominant model of tensor computation using dense representations of tensors. In a dense tensor, every element is stored explicitly in contiguous memory and computation proceeds uniformly over the full index space. In many applications, however, tensor operands are *sparse*, with most entries equal to zero, which makes dense storage and dense evaluation unnecessarily expensive. Exploiting sparsity allows computation to skip unnecessary operations and data movement by compressing zeros and maintaining index metadata that maps the compressed data back to its logical dense form. Different compression methods produce different metadata structures, giving rise to a variety of sparse formats.

The choice of sparse format has major performance implications because the resulting metadata determines how values are accessed, iterated over, and laid out in memory. Some formats, such as HASH, optimize for fast random access using hash-indexed coordinates. Others, like compressed sparse row (CSR) and its higher-order variants, optimize for efficient iteration. Therefore, writing fast sparse tensor programs is complicated by format-specific logic [14].

This chapter extends the scheduling framework developed thus far to sparse tensor computation. We present a verified approach to introducing sparse tensor formats into tensor programs by treating sparsity as another instance of storage transformation. In the dense setting, ATL expresses optimization by separating compute order from storage order and reasoning about their interaction algebraically. The central idea of this chapter is that sparse formats can be understood through the same lens. Rather than treating sparsity as a separate compilation problem, we model sparse representations as structured reorganizations of tensor storage and extend ATL's scheduling framework to optimize programs over these representations within the same verified setting. We extend the ATL framework to support sparse format optimizations via the following contributions:

- A formal, level-based format abstraction capturing how a tensor is compressed on a per-dimension basis, including necessary properties for its soundness as a representation of the original dense tensor
- Multidimensional compress and decompress functions built from the level-based abstrac-

tion

- A format-agnostic soundness theorem of compress and decompress to be used as an ATL scheduling adjoint-pair rewrite
- Mechanized proofs of a core set of level-format instances that can be used in combination with ATL reshape operators to express many common multidimensional tensor-compression formats
- New core ATL language constructs for scatter and coiteration that allow iteration over compressed structures

Ultimately, we achieve an end-to-end verified, source-to-source optimization-derivation process. Starting from a simple, dense ATL program, we are able to derive an optimized ATL program operating over sparse structures while providing formal guarantees of the preservation of functional equivalence. We extend the lowering algorithm presented and verified in chapters 4 and 5 with new lowering rules to accommodate the new scatter and coiteration constructs and two reshape operators, shear and unshear. The lowering of coiteration is currently trusted. We use this extended lowering implementation to evaluate our framework. We evaluate our sparse extension to ATL by optimizing a set of example programs computing over sparse operands stored in various formats. After compiling the resulting ATL programs to C, we find that ATL-generated sparse kernels achieve comparable performance to the Tensor Algebra Compiler (TACO) framework [39]. Currently, our approach supports deriving programs operating over sparse inputs but only produces dense outputs. Supporting writing sparse output structures in a verified, format-parametric way would require extending our existing format algebra with verified layout-construction operations, which we leave to future work.

Our formal level-format abstraction is similar to that proposed by Chou, Kjolstad, and Amarasinghe [14]. However, their abstraction was designed primarily to guide code generation across different storage formats, whereas ours is designed as a formal algebraic interface for reasoning about compression and decompression schemes. In TACO's design, level formats expose different sets of operations depending on their capabilities, and the presence or absence of particular operations informs the generated code. While effective for code generation, this design does not provide a uniform interface suitable for formal reasoning. In contrast, our abstraction is mechanized in Rocq and presented as a unified interface consisting of functions together with semantic specifications and theorems that each format instance must satisfy. This structure enables mechanized proofs of soundness for compression and decompression routines across format instances.

## 7.1 Computing Over Sparse Structures

Treating sparse formats as storage transformations requires extending the scheduling framework beyond representation alone to account for how computation proceeds over compressed data. In the dense setting, ATL programs are written over logical tensors and evaluated by traversing their full index spaces. Compressed sparse representations disrupt this correspondence. Logical tensor coordinates are no longer stored explicitly or contiguously, and

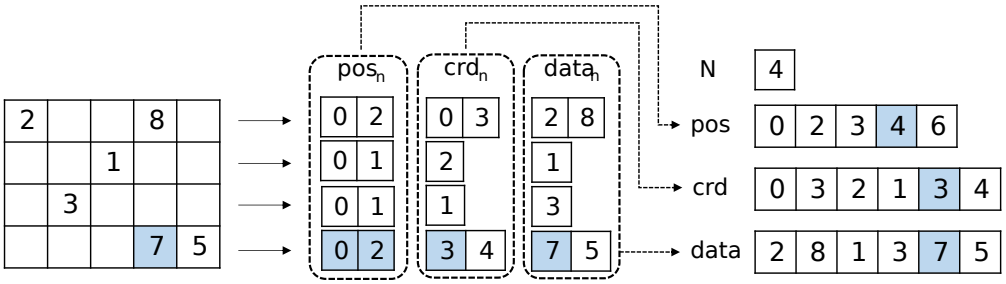



Figure 7.1: Derivation of the CSR representation of a matrix

iteration over physical storage no longer coincides with iteration over the logical tensor domain. Efficient execution over sparse operands therefore requires not only transforming how tensor data is represented, but also reorganizing how computation is expressed. Both the storage transformation induced by a sparse format and the corresponding iteration and access structure required to compute over it are specific to each format.

### 7.1.1 A Concrete Sparse Format

While there are numerous formats used across real-world sparse applications, we begin by examining one widespread, common format: compressed sparse row, or CSR. In CSR, each row of a tensor is compressed, and its zero values are removed. In Figure 7.1, we show an example $4 \times 5$ sparse matrix, alongside the construction of its compressed representation in CSR format. An element of the dense matrix and its corresponding sparse components are highlighted. Each row of the matrix is compressed using the function compress_row defined below that returns the remaining nonzero values along with their corresponding column coordinates. The overall CSR compression structure is produced by the function dense_to_CSR shown below. The final positional array is the prefix sum of the row-positional arrays' final entries representing the numbers of nonzeros per row. Therefore in the final row-position pointer array, each $i$-th entry marks the position in the underlying coordinate and data arrays that belong to the $i$-th row of the dense tensor. The final coordinate array is constructed by concatenating the rows' coordinate arrays. Likewise the final data array contains the concatenated data-value arrays of the rows, storing the tensor's nonzeros in one flat, contiguous buffer.

```
compress_row v = let enum := zip (range 0 |v|) v in
                 let nonzeros := filter (fun (c,x) => x != 0) enum in
                 ([0;|nonzeros| ],map fst nonzeros, map snd nonzeros)
dense_to_CSR m = let N := |m| in
                 let pcv := map compress_row m in
                 let poss := fold (fun l a => l ++ [a + last l])
                                  (map fst pcv) [0] in
                 let crds := concat (map (fst . snd) pcv) in
                 let vals := concat (map (snd . snd) pcv) in
                 (N, poss, crds, vals)
```

We draw a distinction between an index representing the *position* of an element, its physical position in the compressed data layout; and one representing the *coordinate* of an element, its logical position in a tensor's dense representation.

### 7.1.2 Compression-Decompression as an Adjoint Pair

In many ways, compressed formats can be viewed as a particularly exotic form of storage reordering, and decompression is the natural dual of compression: the optimized pattern for iterating over the compressed data. In this case, a specific compression routine will provide the specification for this storage reordering and how to reinterpret it faithfully, given its metadata, into the original dense tensor representation.

Rather than implementing a new reshape-operator pair for each sparse format and necessitating individual adjoint-theorem proofs, we take a parametric approach. In this work, we propose the definition of a new reshape-operator pair, compress and decompress. We define a general, levelized (per-dimension), modular abstraction for describing sparse formats, similar to the work of Chou, Kjolstad, and Amarasinghe [14]. We use this level-format abstraction to construct multidimensional compression formats compositionally and implement a format-agnostic way to generate code to compute over sparse structures. In this case, compress will be parameterized by a list of level-format instances, $F$, compositionally describing a multidimensional tensor-compression format.

We not only define a level-format abstraction, but we also formalize it with a set of properties that must be proven for instances of these level formats. These properties ensure soundness of these structures and well-behaved decompression schemes. Therefore, not only are we able to derive computations over sparse structures without case analysis over each potential format, we are also able to prove soundness for this adjoint theorem once and for all in a format-agnostic way.

Let us revisit the matrix-multiplication program. The ATL program for matrix-vector multiplication is shown below. Let's consider the case where $m$ is a sparse matrix.

$$\boxplus_{i=0}^{N} \sum_{j=0}^{M} m[i;j] * v[j]$$

To derive an optimized sparse program, we begin by rewriting with the compress-decompress adjoint theorem around the tensor operand we want to compute over in a sparse format. Applying this adjoint-theorem rewrite, we arrive at the following program.

$$\boxplus_{i=0}^{N} \sum_{j=0}^{M} (\mathsf{decompress}\ \ (\mathsf{compress}\ F\ m))\,[i;j] * v[j]$$

Note that in general we never intend to call compress at runtime. Instead, we abstract out the call to compress into a binding as follows.

$$\mathsf{let}\ [m_1;...;m_n]\ := \mathsf{compress}\ F\ m\ \mathsf{in}\ \boxplus_{i=0}^{N} \sum_{j=0}^{M} (\mathsf{decompress}\ \ ([m_1;...;m_n]))\,[i;j] * v[j]$$

Rather than evaluating this let-binding as part of our program, we simply require that these let-bound variables are passed in as function input to the let-binding body, which is the newly derived sparse kernel. In compilation, the kernel will be interpreted as the body of this let-binding; and the bound metadata index structures are interpreted as free variables taken in as input to the program, similar to the vector argument. Furthermore, when $F$ is instantiated with specific level formats, we know the types of each of these metadata objects. For example, if we instantiate $F$ with the level formats to express CSR, the metadata index structures $m_1; ...; m_n$ returned from compress will be the integer and value arrays pos, crd, and data like the ones shown in Figure 7.1. Hence we can unfold decompress to introduce a new compute order that iterates over the compressed metadata-index structures instead of the full space of dense coordinates and arrive at the following program.

$$\boxplus_{i=0}^{N} \sum_{p=\mathsf{pos}[i]}^{\mathsf{pos}[i+1]} \mathsf{data}[p] * v[\mathsf{crd}[p]]$$

The outer tensor generation remains the same since each row in CSR is still represented in its compression. However, the bounds of the inner tensor summation are different. Rather than iterating over the full size of the dense dimension, it considers only the nonzero values stored in the compressed format by iterating over the row-position array. As a result, there is an access into the underlying data array by position rather than an access into $m$ by coordinate. To access the vector $v$, the positional access index $p$ must be converted into the equivalent logical, dense coordinate. This translation is performed by an access into the coordinate-index array stored as part of CSR.

## 7.2 Extending ATL

In this section we go over the extensions we made to the ATL framework to support new operators and iteration patterns that will appear in ATL programs after the introduction of sparse formats. Extensions include a set of new core constructs to the ATL language in order to accommodate the operations we will need to express computation over sparse structures and a new pair of reshape operators that extends the set of expressible formats.

### 7.2.1 New Core ATL Constructs

We begin by introducing new core ATL language constructs used to express sparse iterations and computation over sparse metadata-index structures. These constructs are instrumental in being able to express decompress in a format-agnostic way, as well as generally enabling us to describe efficient traversal of sparse spaces in ways that the existing ATL constructs could not.

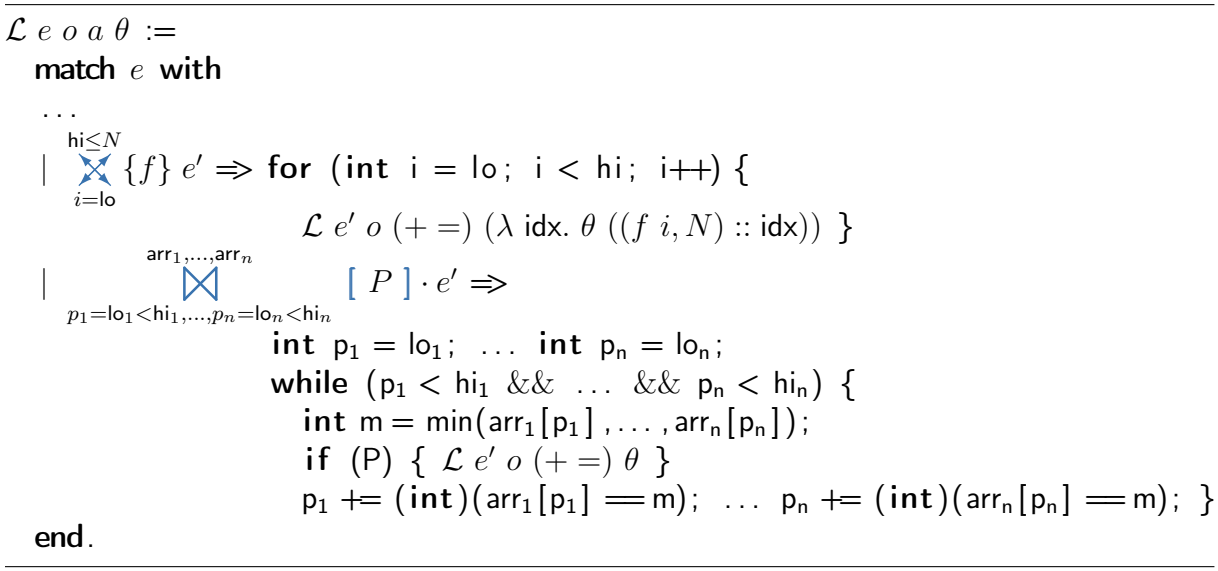



Figure 7.2: Lowering of scatter and coiteration constructs

### Integer and Boolean Arrays

In the syntax specification, we extend the grammar of index and Boolean expressions to include access into integer arrays as follows:

$$\text{Index Expression}\quad I \ ::=\ \ldots \mid x[I] \qquad\qquad \text{Predicate}\quad p \ ::=\ \ldots \mid x[I]$$

These expressions will be used in programs computing over certain sparse index-metadata structures. Recall from the CSR-formatted matrix-vector-multiplication program that such expressions are needed to capture accesses into the position and coordinate arrays. Similarly, Boolean array access can appear in the predicate expressions of Iverson brackets in programs. Note here that the array access that appears in index expressions accesses into integer arrays whereas the array access that appears in predicate expressions accesses into Boolean arrays. With this extension, we can guard the values of expressions based on separate index-metadata structures.

### Scatter

We implement a new scatter construct—an iterating construct similar to tensor generation and tensor summation, denoted by two crossed arrows ($\bowtie$). The scatter construct allows us to iterate over a the sparse coordinate space of one tensor while writing and relating it into the dense dimension of the output. The denotational semantics of the scatter construct are shown in Figure 7.3. The scatter begins with a zero-filled tensor of length $N$ with the proper shape and dimensionality as typed by the expression $e$, which is a function of $i$. Then the index $i$ for every value between lo inclusive and hi exclusive will update this tensor at index $f\ i$ with a value $e\ i$. In this sense it is similar to tensor generation, althrough rather than storing the value $e\ i$ at index $i$, it adds it at a "scattered" index $f\ i$. The rewrite rule used to introduce scatter into ATL programs, RWScat, is shown in Figure 7.4.

$$\left[\!\!\left[ \bigbowtie_{i=\mathsf{lo}}^{\mathsf{hi}\leq N} \{f\}\ e \right]\!\!\right] = \mathsf{fold}\ (\lambda\ l\ i.\ l[f\ i \leftarrow [\![e\ i]\!] + l[f\ i]])\ [\mathsf{lo}, \mathsf{lo}+1, ..., \mathsf{hi}-1]\ (\mathsf{repeat}\ 0\ N)$$

$$\left[\!\!\left[ \bigbowtie_{p_1=\mathsf{lo}_1<\mathsf{hi}_1,...,p_n=\mathsf{lo}_n<\mathsf{hi}_n}^{\mathsf{arr}_1,...,\mathsf{arr}_n} [\ P\ ] \cdot e \right]\!\!\right] = \mathsf{fold}\ (\lambda\ (r, \mathsf{ps})\ \_\ .\ (\mathsf{incr}\ \mathsf{ps}\ [\mathsf{arr}_1; ...; \mathsf{arr}_n], r \oplus [\ \mathsf{fold}\ (\&)\ ([\mathsf{ps}[0] < \mathsf{hi}_1; ...; \mathsf{ps}[n-1] < \mathsf{hi}_n])\ (P\ \mathsf{ps})\ ] \cdot [\![e\ \mathsf{ps}]\!]))\ (\mathsf{repeat}\ 0\ (\Sigma_{i=1}^{n} \mathsf{hi}_i - \mathsf{lo}_i))(0, [\mathsf{lo}_1; ...; \mathsf{lo}_n])$$

Figure 7.3: Scatter and coiteration denotational semantics

$$\text{RWCoit}\ \frac{\mathsf{ordered}_c^n\ c_1 \quad \mathsf{unique}_c^n\ c_1 \quad \mathsf{ordered}_k^m\ c_2 \quad \mathsf{unique}_k^m\ c_2}{\sum_{i=0}^{n} \sum_{j=k}^{m} [\ c_1[i] = c_2[j]\ ] \cdot d_1[i] * d_2[j]} = \bigbowtie_{i=0<n, j=k<m}^{c_1, c_2} [\ c_1[i] = c_2[j]\ ] \cdot d_1[i] * d_2[j]$$

$$\text{RWScat}\ \frac{\forall i.\ 0 \leq i < m \rightarrow 0 \leq \mathsf{rdx}\ i < n}{\boxplus_{i=0}^{n} \sum_{j=0}^{m} [\ \mathsf{rdx}\ j = i\ ] \cdot e\ j = \bigbowtie_{p=0}^{m \leq n} \{\mathsf{rdx}\ i\}\ e\ p}$$

Figure 7.4: Rewrite rules for introducing scatter and coiteration

We extended the ATL lowering algorithm with a new production rule for the scatter operator as shown in Figure 7.2. This rule generates a for-loop with bounds matching the iterating bounds of lo and hi. The body of this for-loop is the lowering of the body of the scatter, but it builds upon the index by adding a tuple of the scattered version of its iterating index $(f\ i)$ and the full size of the scattered dimension $N$.

### Coiteration

The coiteration operator denoted by $\bowtie$ enables joint iteration over multiple integer arrays in ATL, enabling efficient traversal of multiple integer arrays and accumulation of results based on values at corresponding index positions that satisfy a given predicate. The efficiency of coiteration comes from the knowledge that the integer arrays are *locally ordered*. Instead of scanning each array to find matching and predicate-satisfying entries during each iteration, the construct performs an n-finger merge [9]. As a result, all array indices are advanced in a coordinated manner so that each element is visited at most once.

The denotational semantics for coiteration is shown in Figure 7.3. The construct takes as input $n$ integer arrays $\mathsf{arr}_1, ..., \mathsf{arr}_n$, along with $n$ lower and upper bounds, specifying the positional index ranges for each array. It also takes a body $e$ and predicate $P$, both functions

over the array positions. At each step of the iteration, if the predicate $P$ holds for the current positions and all positions are within bounds, the application of $e$ is accumulated. Then, among the current positions, those pointing to the smallest array value are incremented using the incr function (referenced in Figure 7.3). Because at least one position is advanced in each iteration, the number of evaluations of the body expression is bounded by the combined positional spans of the input arrays, or $(\Sigma_{i=1}^{n} \mathsf{hi}_i - \mathsf{lo}_i)$. A common rewrite rule RWCoit used to introduce coiteration into ATL programs is shown in Figure 7.4. Note that it requires that the input integer arrays being coiterated must contain elements that are ordered and unique within the coiterating bounds.

The corresponding lowering rule shown in Figure 7.2 produces a while loop. Each integer array is associated with a pointer initialized to its corresponding lower bound outside this loop, which continues as long as every pointer remains below its corresponding upper bound. Inside the loop, the current minimum value among the pointed-to elements of the arrays is determined. The construct then checks whether the current positions satisfy the predicate. If so, the body expression is evaluated, and its result is accumulated. Finally, each pointer that currently points to the minimum value is incremented before the next iteration begins.

### 7.2.2 New Reshape Operators

We introduce a new pair of reshape operators, shear and unshear, to expand the set of multidimensional sparse formats expressible in the ATL framework. Later, we will show how these operators are used to derive computations involving a sparse tensor format known as DIA (Section 7.6).

The shear operator constructs a new tensor where each row stores a diagonal of the original tensor. Since many diagonals of a tensor are not the same length, the shear operator pads these diagonals to the length of the longest diagonal (the main diagonal of the tensor) with zero values. unshear reconstructs a tensor with $n$ rows and $m$ columns by storing an input tensor's rows as diagonals, truncating the lengths of these rows as necessary to produce a $n \times m$-size output. The definitions for these operators are shown below.

$$\begin{aligned}\mathsf{shear}\ t := \ & \mathsf{let}\ D\ := \mathsf{min}\ |t|\ |t[0]|\ \mathsf{in}\ \boxplus_{d=0}^{|t|+|t[0]|-1} \boxplus_{r=0}^{D} \\ & [\ 0 \le \mathsf{max}\ 0\ (n - D - (\mathsf{max}\ 0\ (d - D + 1))) + r - n + 1 + d < |t[0]|\ ] \cdot \\ & t[\mathsf{max}\ 0\ (n - D - (\mathsf{max}\ 0\ (d - (D-1)))) + r; \\ & \ \ \mathsf{max}\ 0\ (n - D - (\mathsf{max}\ 0\ (d - D + 1))) + r - n + 1 + d]\end{aligned}$$

$$\mathsf{unshear}\ n\ m\ t := \boxplus_{i=0}^{n} \boxplus_{j=0}^{m} t[j-i+n-1; i-\mathsf{max}\ 0\ (n-(\mathsf{min}\ n\ m)-(\mathsf{max}\ 0\ (j-i+n-1-(\mathsf{min}\ n\ m-1))))]$$

We prove the following adjoint identity theorem for these reshape operators.

$$\textsc{UnshearShearId}\ \frac{}{t = \mathsf{unshear}\ |t|\ |t[0]|\ (\mathsf{shear}\ t)}$$

We extend the ATL lowering algorithm to accommodate shear and unshear by following the reshape-operator lowering form shown earlier with the following corresponding reindexers.

```
θshear idx := match idx with
              | (v,d)::(v',d')::idx' ⇒
                  (v' − v + d − 1, d + d' − 1)::
                  (v − (max 0 ((d − min d d') −
                  (max 0 (v' − v + d − min d d')))),min d d')::idx'
              | _ ⇒ idx end
θunshear N M idx := match idx with
              | (d,n)::(r,m)::idx' ⇒
                  (max 0 (N − (min N M) − (max 0 (d − min N M + 1))) + r,N)::
                  (max 0 (N − (min N M) − (max 0 (d − min N M + 1))) + r
                       − N +1 + d,M)::idx'
              | _ ⇒ idx end
```

## 7.3 Level-Format Abstraction

The core of our formalization of sparse formats is the level-based abstraction. The abstraction encapsulates how to apply an encoding routine to one dimension of a tensor and produce a corresponding metadata-index structure for an abstract level format $F$. It also includes functions detailing how to access and interpret the underlying data within a format's layout. Our level-format abstraction includes not just the level-format functions but also properties of their semantic specifications. These ensure that the abstraction functions encode a compression format that can be reinterpreted into the original dense tensor. For each level-format instance, these functions are implemented and their properties proven. The algebraic signature for this abstraction presented in Figure 7.5 is realized in the Rocq proof assistant.

### 7.3.1 Level-Encoding Functions and Index Structures

As part of the abstraction, a level format $F$ must provide certain definitions that define the format encoding. These definitions are listed in Figure 5(a).

They include the level-encoding function $\mathsf{encode}_F$ that takes a tensor of rank at least 1 and returns a tuple of the remaining elements after compression and a corresponding metadata-index structure of type $M_F$. This type $M_F$ is also specified per level format.

The abstraction also requires an operation $\mathsf{append}_F$, which combines two metadata index objects into one. This function is folded over the metadata-index structures produced when compression is mapped over elements in one dimension. The resulting metadata-index structure is then used to describe how to access and decode the dimension as a whole.

The abstraction also includes a specific metadata-index structure $\phi_F$, which is meant roughly to represent an "empty" index structure.

The final definition regarding the encoding-metadata-index structures that must be provided is a well-formedness predicate defined on elements of type $M_F$.

A format must also provide functions to access a metadata-index structure and meaningfully iterate over it. The indices used for access and iteration are integers, though they could represent (logical) coordinates or (physical) positions. To make this difference apparent in

$$\textsc{MonoidStruct}\ \frac{}{(\mathsf{well\_formed}_F\ M_F, \phi_F, \mathsf{append}_F)\ \text{forms a monoid}}$$

$$\mathsf{Let}\ (m_F, t')\ :=\ \mathsf{encode}_F\ t\ \ \mathsf{in}$$

$$\textsc{EncodeWellFormed}\ \frac{}{\mathsf{well\_formed}_F\ m_F}$$

$$\begin{gathered}\mathsf{encode}_F\ : \mathsf{list}\ X \to M_F * \mathsf{list}\ X\\ \mathsf{append}_F : M_F \to M_F \to M_F\\ \phi_F : M_F\\ \mathsf{well\_formed}_F : M_F \to \mathsf{Prop}\\ \mathsf{crd\_pos}_F : M_F \to \mathbb{Z}_p \to \mathbb{Z}_c \to\ \mathsf{option}\ \mathbb{Z}_p\\ \mathsf{pos\_crd}_F : M_F \to \mathbb{Z}_p \to \mathbb{Z}_p \to\ \mathsf{option}\ \mathbb{Z}_c\\ \mathsf{pos\_pos}_F : M_F \to \mathbb{Z}_p \to \mathbb{Z}_p \to\ \mathsf{option}\ \mathbb{Z}_p\\ \mathsf{pos\_bounds}_F : M_F \to \mathbb{Z}_p \to\ \mathbb{Z}_p * \mathbb{Z}_p\\ \mathsf{valid}_F : M_F \to \mathbb{Z}_p \to\ \mathsf{bool}\end{gathered}$$

(a) Format-abstraction definitions

$$\textsc{CrdPosGet}\ \frac{\mathsf{crd\_pos}_F\ m_F\ 0\ c = p}{t[c] = t'[p]}$$

$$\textsc{CrdPosNoneGet}\ \frac{\mathsf{crd\_pos}_F\ m_F\ 0\ c = \mathsf{None}}{t[c] = 0}$$

$$\textsc{InvalidPosGet}\ \frac{\neg\ \mathsf{valid}_F\ m_F\ 0\ p}{t'[p] = 0}$$

(b) Properties of encoding $M_F$

$$\textsc{PosCrdInj}\ \frac{\mathsf{valid}_F\ m_F\ p'\ p_1 \qquad \mathsf{valid}_F\ m_F\ p'\ p_2 \qquad \mathsf{pos\_crd}_F\ m_F\ p'\ p_1 = \mathsf{pos\_crd}_F\ m_F\ p'\ p_2}{p_1 = p_2}$$

$$\textsc{CrdPosInv}\ \frac{\mathsf{crd\_pos}_F\ m_F\ p'\ c = p \qquad \mathsf{valid}_F\ m_F\ p'\ p}{\mathsf{pos\_crd}_F\ m_F\ p'\ p = c} \qquad \textsc{PosCrdInv}\ \frac{\mathsf{pos\_crd}_F\ m_F\ p'\ p = c \qquad \mathsf{valid}_F\ m_F\ p'\ p}{\mathsf{crd\_pos}_F\ m_F\ p'\ c = p}$$

$$\textsc{CrdPosNone}\ \frac{\mathsf{crd\_pos}_F\ m_F\ p'\ c = \mathsf{None} \qquad i \in \mathsf{pos\_bounds}_F\ m_F\ p' \qquad 0 \le c < \mathsf{dim}\ m_F}{\mathsf{pos\_crd}_F\ m_F\ p'\ i \neq\ c}$$

(e) Properties of position-coordinate translation functions

$$\textsc{CrdBound}\ \frac{\mathsf{valid}_F\ m_F\ p'\ p \qquad \mathsf{pos\_crd}_F\ m_F\ p'\ p = c}{0 \le c < \mathsf{dim}\ m_F} \qquad \textsc{PosBound}\ \frac{\mathsf{crd\_pos}_F\ m_F\ p'\ c = p}{p \in (\mathsf{pos\_bounds_F}\ m_F\ \mathsf{p'})}$$

(h) Properties of coordinate and position bounds

$$\textsc{CrdPosApp}\ \frac{\begin{array}{rcl}\mathsf{Let}\ \phi s & := & \mathsf{map}\ (\mathsf{fst} \circ \mathsf{encode}_F)\ t \quad \mathsf{in}\\ \mathsf{Let}\ \delta & := & \mathsf{map}\ (\mathsf{snd} \circ \mathsf{encode_F})\ t \quad \mathsf{in}\\ \mathsf{Let}\ \Phi & := & \mathsf{fold}\ \mathsf{append}_F\ \phi s\ \phi_F \quad \mathsf{in}\end{array}}{\mathsf{crd\_pos}_F\ \Phi\ p\ c = \mathsf{crd\_pos}_F\ (\phi s[p])\ 0\ c}$$

$$\textsc{BoundsGet}\ \frac{}{\begin{array}{c}\mathsf{snd}\ (\mathsf{pos\_bounds}\ (\mathsf{encode_F}\ \mathsf{t[p]}))\ 0 - \mathsf{fst}\ (\mathsf{pos\_bounds}\ (\mathsf{encode_F}\ \mathsf{t[p]}))\ 0 =\\ \mathsf{snd}\ (\mathsf{pos\_bounds_F}\ \Phi\ p) - \mathsf{fst}\ (\mathsf{pos\_bounds_F}\ \Phi\ p)\end{array}}$$

$$\textsc{ValidGet}\ \frac{}{\mathsf{valid}_F\ \Phi\ (\mathsf{pos\_pos}_F\ \Phi\ p'\ p) = \mathsf{valid}_F\ (\mathsf{fst}\ (\mathsf{encode_F}\ \mathsf{t[p']}))\ 0\ p}$$

$$\textsc{AppGet}\ \frac{}{(\mathsf{concat}\ \delta)[\mathsf{pos\_pos}_F\ \Phi\ c\ (\mathsf{crd\_pos}_F\ \Phi\ c\ p)] = t[c; p]}$$

(i) Properties of $\mathsf{append}_F$

Figure 7.5: Level-format abstraction for a format $F$, with the tensor type $X$

the type signatures, we will indicate the type of a positional index as $\mathbb{Z}_p$ and a coordinate as $\mathbb{Z}_c$.

The format function crd_pos is a partial function that converts coordinates to positions. In addition to the integer coordinate, this function takes in a metadata-index structure and a position argument. This position represents an offset in the metadata structure in case the dimension being decoded is an inner dimension (recall that the metadata structure may be a combination structure of several encoded elements within this dimension). We will call this kind of positional argument the parent position. Similarly, a partial function pos_crd must be provided to be able to convert from a position to its corresponding coordinate.

The pos_pos function takes a positional index in one dimension of a metadata-index structure and returns the positional index in the next dimension. This transition corresponds to the start of the structure describing the elements of the tensor element, representing an access from a higher dimension into a lower one but through sparse positions rather than coordinates.

Relatedly, the function pos_bounds must return the span of positions corresponding to a tensor element in the underlying dimension, as a function of a position and a metadata-index structure. The span of these positions is returned as a tuple of integers denoting the lower and upper bound of the relevant positions.

Finally, depending on the level format, not every position within the positional bounds returned by $\mathsf{pos\_bounds}_F$ necessarily corresponds to a nonzero tensor element. Therefore, a level format also provides a valid function that takes a metadata-index structure and position and returns whether or not that position is valid, or actually contains an element to be decoded further.

### 7.3.2 Level-Encoding and Index-Structure Properties

Given the abstraction index-structure and function definitions, the level-format abstraction also requires these definitions to uphold the properties stated in Figure 5(b).

The first property MonoidStruct requires that the set of well-formed elements of $M_F$ must be closed under $\mathsf{append}_F$, an associative operation over well-formed elements of $M_F$ with $\phi_F$ as an identity element. In other words, well-formed $M_F$ must form an algebraic monoid under $\mathsf{append}_F$.

The rest of this set of properties listed in Figure 5(b) establishes how we expect the access and iteration functions to interact with a metadata-index structure produced by $\mathsf{encode}_F$.

First, EncodeWellFormed states that the encoding procedure must produce metadata index structures that are well-formed under the definition of this abstraction.

Next, the property CrdPosGet states if a coordinate $c$ is encoded to have a position $p$ by $m_F$, then accessing into the original tensor $t$ at index $c$ would be equivalent to accessing into the compressed data output $t'$ at index $p$.

Similarly, CrdPosNoneGet states if coordinate $c$ does not have a position represented in $m_F$, then accessing into the dense tensor $t$ at index $c$ must be a tensor of zeros. In other words, the element in $t$ at index $c$ was compressed out.

Likewise, InvalidPosGet states if $p$ is not a valid position within $m_F$, then accessing into the compressed data $t'$ at index $p$ will return a tensor of zeros.

### 7.3.3 Properties of Coordinate-Position Translation

The coordinate-to-position and position-to-coordinate translation functions of a level format must also fulfill key properties listed in Figure 5(c).

The first property CrdPosInv states that if a coordinate $c$ is translated to a position $p$, and $p$ is a valid position within a metadata-index structure $m_F$, then converting $p$ back to a coordinate through $m_F$ should yield $c$.

Likewise, PosCrdInv states that if we begin with a valid position $p$, and converting $p$ to a coordinate yields $c$, then the position translation of $c$ should return $p$. Additionally, PosCrdInj also enforces injectivity on these translation functions. If there are two valid positions $p_1$ and $p_2$ under an index structure $m_F$, and they translate to the same coordinate, then $p_1$ must equal $p_2$.

Finally, CrdPosNone states if a coordinate $c$ does not translate to any position in $m_F$, then no position $p$ between the positional bounds established by $m_F$ translates to coordinate $c$.

### 7.3.4 Properties of Coordinate and Position Bounds

The properties in Figure 5(f) constrain the range of values that the indices translated through the metadata-index structure can take on. These bounds ensure that accessing the dense and sparse structures through these translation functions will always be in-bounds. These properties pertain to any well-formed metadata-index structure $m_F$ and also any parent position $p'$, as this index structure may result from the appending of several index structures.

The first property CrdBound states that if $p$ is a valid position and translates to a coordinate $c$ through some metadata structure $m_F$, then $c$ must be a nonnegative integer less than the size of the tensor dimension encoded by $m_F$.

The next property PosBound enforces a similar bound on positions that result from translation. If a coordinate $c$ translates to a position $p$, then $p$ must be between the positional bounds established by pos_bounds for this index structure.

### 7.3.5 Properties of $\mathsf{append}_F$

The set of properties in Figure 5(i) pertain to the append function that combines metadata-index structures for a given format. In these properties we consider metadata-index structure $\Phi$ that is the aggregate structure resulting from the fold of $\mathsf{append}_F$ across a list of metadata-index structures, $\phi s$. Specifically, $\phi s$ is obtained from mapping $\mathsf{encode}_F$ over the elements of a tensor $t$. Note the implication that $t$ is at least two-dimensional. This aggregated metadata-index structure, $\Phi$, represents every element in each "row" of $t$. These properties defining $\mathsf{append}_F$ allow us to decompose a function call at some position $p$ indexing through $\Phi$ into a function call to the $p$-th metadata-index structure in $\phi s$ instead. In essence, we may "access" within a sparse tensor dimension and focus in on subdimensions.

The first property CrdPosApp states that a coordinate $c$ translated through $\Phi$ with a parent position of $p$ is equivalent to the position translation of coordinate $c$ from the $p$-th index structure of $\phi s$ with a new parent-position offset of 0.

Next, the BoundsGet property states that the positional range encoded by $\Phi$ at a parent position $p$ must be the same as the positional range obtained by directly encoding the $p$-th element of tensor $t$.

Similarly, ValidGet states if a position $p$ is valid within $\Phi$ with a parent position of $p'$, then position $p$ must be valid in the metadata-index structure produced by directly encoding the element in $t$.

Finally, AppGet states that $\mathsf{append}_F$ maintains the positional properties that align with the underlying encoded data described by the metadata structures. The access into a concatenated collection of sparse data is analogous to an access into a vector produced from flattening a matrix. In other words, accessing into this concatenated list of the encoded data at the position offset from some parent position $c$ through $\Phi$ and position $p$ corresponding to a position within an element of that dimension is equivalent to the two-dimensional access of $c$ and then $p$ into the original dense tensor $t$.

## 7.4 Level-Format Instances

In this work, we provide proofs and implementations for a small set of formats as instances of the level-format abstraction. We chose these formats to demonstrate expressivity and compositionality.

### 7.4.1 Scalar

The scalar format, denoted as $\mathsf{s}$ when passed as an argument to $\mathsf{compress}$, is a trivial format, only applicable as a degenerate compression format onto scalars, or zero-rank tensors. The following shows the scalar instance definition of the level-format abstraction.

$$\mathsf{M_s} := \mathsf{list}\ \mathbb{R} \quad \mathsf{encode_s} r := ([r], r) \quad \mathsf{append_s}\ m_1\ m_2 := m_1 +\!\!+ m_2$$

The scalar format must be the final format in the format-list argument to $\mathsf{compress}$ as it represents the data payload beneath all the metadata-index structures produced from composing compression. Therefore, the scalar format type is a list of scalar values that represents the data. The $\mathsf{encode}$ function for the scalar format is implemented to encapsulate the scalar value as a singleton list. We expect the degenerate case of compressing a zero-rank tensor using the scalar format to result simply in an array containing the scalar value. Finally, the $\mathsf{append}$ function is defined as concatenation.

### 7.4.2 Dense

The dense format is denoted as $\mathsf{d}$ when passed as an argument to $\mathsf{compress}$. This format represents a dimension that will remain uncompressed and retain all of its original elements. The definitions of this instance are shown below.

$$\mathsf{M_d} := \mathbb{Z} \quad \mathsf{encode_d}\ t := (|t|, t) \quad \mathsf{append_d}\ m_1\ m_2 := m_1$$

The only information in the metadata-index structure that needs to be carried by the dense format is the original dimension size. Therefore a metadata index value is an integer.

Additionally, since the dense format retains all tensor elements, the data returned by encode will just be the original tensor. The index data returned is the dimension size or simply the length of the input tensor. Therefore, the dense format's encode just returns the input tensor and its size. Given two dimension sizes as metadata structures, the append function will return the first one. This convention is reasonable: since we expect only to be appending together metadata-index structures of tensor elements of the same dimensionality within one consistent tensor, we can assume that the dimension sizes stored in these index structures will match.

### 7.4.3 Bitmask

The bitmask format will be denoted as b. This format, like the dense format, retains all elements of a tensor. However, the bitmask format additionally stores a bitvector that encodes whether the tensor element at each corresponding position is a nonzero.

$$\begin{aligned}
&\mathsf{M_b} := \mathsf{list}\ \mathbb{B} * \mathbb{Z} \\
&\mathsf{encode_s}\ t := (\mathsf{map}\ (\neq\ 0)\ t, |t|, t) \\
&\mathsf{append_s}\ m_1\ m_2 := (\mathsf{fst}\ m_1 +\!\!+ \mathsf{fst}\ m_2, \mathsf{snd}\ m_1)
\end{aligned}$$

For this level format, the metadata-index structure type will store this bitvector as a list of Booleans and the original dimension size as an integer. The bitmask format's encode function returns the input tensor with no values removed along with its bitvector. To construct this bitvector, we map over the input tensor an indicator returning a Boolean denoting whether that element is nonzero. The append function for combining bitvectors will concatenate them. Additionally, like the dense level format, combining metadata-index structures will only be well-defined for tensors of the same shape, so we expect the input index structures to agree on dimension size.

### 7.4.4 Hash

The hash format compresses one dimension of a tensor and stores a hash array of the coordinate-index entries of the nonzero elements. This hash array has a fixed width $n$ and must be able to accommodate all remaining tensor elements in order to be sound. The hash format is parametrized by this width $n$ as well as a hash function $f$ that maps integers to integers. The parameterized hash format can be denoted as $\mathsf{h}\ n\ f$ and defined as follows. The metadata-index structure for the hash level format must contain the hash integer array, the hash function, the width of the hash arrays, and the original size of the dimension.

$$\begin{aligned}
&\mathsf{M_h} := \mathsf{list}\ \mathbb{Z} * \mathbb{Z} * \mathbb{N} * (\mathbb{Z} \to \mathbb{Z}) \\
&\mathsf{encode_h} := (\mathsf{hash_level}\ t, \mathsf{filter}\ t) \\
&\mathsf{append_h}\ m_1\ m_2 := m_1 +\!\!+ m_2
\end{aligned}$$

The following function is a helper used to implement the construction of the hash array, by taking a coordinate c and inserting that coordinate and a corresponding tensor element at either its hashed position or searching for the closest available position without an existing entry.

```
insert n f c x h := match n with
  | 0 ⇒ h (* no room in hashmap *)
  | S n' ⇒ if (fst (h[(f c + |h| − n) % |h| ]) == −1)
    then h[(f c + |h| − n) % |h| ← (c,x)] else insert n' f c x h end
```

To begin, the hash array is entirely empty with index entries of $-1$ mapped to zero-valued tensors. This definition is provided for the empty metadata-index structure, $\phi_h$, for the hash format as well.

$$\phi_{\mathsf{h}} := (\mathsf{repeat}\ (-1, 0)\ n, N, n, f)$$

The encoding function starts with an empty hash array. Then folding over a given tensor, it calls insert to insert each nonzero element along with its dense coordinate into the hash array. We define this function hash_level as follows.

```
hash_level t := snd (fold (fun (c,a) x ⇒ (c+1
                    , if x == 0 then a else insert |a| f c x a)) t
                      (0,(repeat (−1, 0) n))
```

Similarly to previous formats, the append function is well-formed only if its arguments are of the same format. For the hash format, this means that the hash widths as well as the hash functions are the same. Therefore, we concatenate together the two arrays and adopt the same hash function. Note that we need not construct a new hash function for this larger map, because all the level-abstraction access functions take a parent-position argument. Thus all hashed indices will translate properly to the offset within the proper, original bucket.

### 7.4.5 Compressed

The compressed format removes zero elements in one dimension of a tensor and stores two integer arrays as its metadata-index structure. This level-format instance is defined as follows.

$$\begin{aligned}
&\mathsf{M_c} := \mathsf{list}\ \mathbb{Z} * \mathsf{list}\ (\mathsf{list}\ \mathbb{Z}) * \mathbb{Z} * (\mathbb{Z} \to \mathsf{list}\ \mathbb{Z}) \\
&\mathsf{encode}_c\ t := ([0; |\mathsf{filter}\ t|], \mathsf{comp}\ f\ t, |t|, f, \mathsf{filter}\ t) \\
&\mathsf{append}_{\mathsf{c}}\ m_1\ m_2 := \mathsf{app}\ (\mathsf{fst}\ m_1)\ (\mathsf{fst}\ m_2)\ (\mathsf{snd}\ m_1)\ (\mathsf{snd}\ m_2)
\end{aligned}$$

The first integer array has entries marking the position where each corresponding segment in the underlying data array begins. This array we call the position array, as it stores starting positions for these segments. So the initial call to $\mathsf{encode_c}$ will always return the position array $[0; n]$, where $n$ is the number of tensor elements remaining.

The second integer array stores the coordinates corresponding to the tensor elements remaining after compression. In this work we implement an augmented compressed format. Rather than storing the coordinates directly, we parameterize the format with an injective function $f$ of type $\mathbb{Z} \to \mathsf{list}\ \mathbb{Z}$. For each nonzero tensor element, we apply this function to convert its coordinate into a new representation and store these in an array in the metadata-index structure.

Overall the compressed format metadata-index structure will store the position-index array, the converted-coordinate array, the coordinate-conversion function, and the dimension size. Technically, this format also requires a proof term showing that the coordinate-conversion

function is injective over the domain established by the dimension size. However, this proof term can be automated, and we omit it from the type definition for simplicity's sake.

To define the encode function for this format, we define the following helper function to construct the proper coordinate structures for the compressed format. This function returns a list of the converted coordinates of the elements that remain after compression.

```
comp f t i := map (fun x => f (snd x))
                  (filter (fun x => fst x != 0) (zip t (range 0 |t|)))
```

As mentioned before, the append function of the level-format abstraction is only meant to be applied between metadata-index structures of the same level format. Therefore for the compressed format we expect the inputs to have the same coordinate-conversion function and dimension size. To preserve the coordinate-index information, we can concatenate together converted coordinate arrays. To construct the aggregated position array, the latter metadata-index structure being appended must be incremented by the number of positions encoded in the first metadata-index structure to reflect the coordinate arrays being concatenated. This number of positions is the length of the metadata structure's converted coordinate array, or the *last* element of its position array. Therefore we can implement the $\mathsf{append}_c$ level-format function as follows.

```
app p1 p2 c1 c2 := (p1 ++ (map (fun x => x + last p1) (remove_last p2)),c1++c2)
```

## 7.5 Multidimensional Compression and Decompression

Using the level-format-abstraction interface functions, we can describe multidimensional compression and decompression schemes in a format-agnostic way. Since each level format denotes an encoding scheme on a given dimension of a tensor, we can define compression and decompression recursively over a tensor's dimensions. Compression takes in a rank-$n$ tensor and a list of $n$ level formats, returning a stack of $n$ metadata-index structures, while decompression reverses that flow.

### 7.5.1 Multidimensional Compression

The function compress is parameterized by a list of level formats that have been implemented as instances of the formal abstract interface defined previously and applied on a tensor $t$. The function implementation is shown below.

```
compress F t := match F with
              | [f1;...;fn ] => let (m1,t') := encode_f1 t in
                                let metas' := map (compress [f2;...;fn]) t' in
                                let metas := fold append_list metas'
                                                 [φ_f2;...;φ_fn] in (m1::metas)
              | [s] => [ t ] end
```

The length of this format list must be the same as the rank of tensor $t$, because each format applies to one dimension of the tensor, corresponding to its order in the list. For this

format list to be well-defined, we also expect its last level format to be the scalar format, or s, with $t$ being a scalar value, reflected in the base case as shown above. Moreover, this function will return a list of metadata-index structures of the same length as the format list.

The top-level dimension will be compressed according to the encoding routine specified by the first level format in the format list. Then each surviving element in the resulting compressed data will be compressed recursively with the remaining formats. Subsequently, each recursively compressed element returns its own stack of metadata objects. Each of these stacks is elementwise appended to construct one final stack of metadata-index structures, the job of the append_list function. Instead of taking two metadata-index structures and returning a new metadata-index structure, it takes two *lists* of metadata-index structures and returns a list. It does so by calling the appropriate $\mathsf{append}_F$ elementwise between the two lists. Note that this function only makes sense if the ordering of formats of metadata-index structures matches between its two list arguments.

By designing the level-format abstraction to include the compress and append functions without interdimensional dependencies, we can build a format-agnostic, compositional way to define compression over a tensor. More notably, we can build and customize multidimensional tensor-compression formats by composing different level formats without having to build by-hand the reasoning needed to interact with and interpret the compression structures between dimensions.

### 7.5.2 Multidimensional Decompression

We are also able to use the level-format abstraction to write a format-agnostic definition of multidimensional decompression. Recall that the output of the multidimensional compress function is a list of metadata-index structures of the same length as the rank of the multidimensional tensor it encodes. Likewise, the multidimensional decompress function will take a list of metadata-index structures to decompress and return a tensor of corresponding dimensionality. It will also take in a parent positional index to keep track of its position within the metadata-index structure of the dimension being decompressed. It is defined as follows.

```
decompress M p' :=
  match M with
  | [m_1; ...; m_n] ⇒ let (lo, hi) := pos_bounds m_1 in
      ⨉_{p=lo}^{hi≤dim m_1} {pos_crd m_1 p'p} [ valid m ]· decompress [m_2; ...; m_n] (pos_pos m_1 p' p)
  | [ r ] ⇒ r[p'] end
```

Similar to the definition of multidimensional compression, the list of metadata-index structures is only well-formed if the last object is the scalar format's index-structure type. In other words, the final object $\mathsf{m}_n$ for a rank-$n$ tensor passed to decompress must be a list of scalar values. When this payload array of values is reached in the base case of multidimensional decompression, we simply perform the access of the positional argument that has been translated throughout each dimensional call to decompress.

### 7.5.3 Soundness of Compression and Decompression

Using these two function definitions, we are able to define the top-level soundness theorem on compression and decompression. This theorem states that decompressing the metadata-index structures obtained from composing any properly formed list of level formats to compress a tensor will return the original tensor.

$$\textsc{DecompressCompress}\ \frac{\mathsf{formats\_have\_space}\ F \qquad \mathsf{proper\_formats}\ F\ t}{t = \mathsf{decompress}\ (\mathsf{compress}\ \mathsf{F}\ t)\ 0}$$

The property proper_formats states the length of $F$ matches the dimensionality of the tensor $t$ and that its terminal format is scalar. The formats_have_space property states that any fixed-size level formats (such as hash) in $F$ have enough space for the nonzero entries in that dimension of $t$.

Rewriting using this adjoint-pair theorem, we can turn a standard ATL program computing over dense tensors into one computing over the sparse metadata-index structures. First, the call to compress can be simplified into the list of metadata-index structures. Just like how in the dense ATL case the programs are parameterized over the input tensors and take them in as arguments, the new sparse program will accept the metadata-index structures in this list as arguments. Then, decompress can be unfolded. We are able to unfold this implementation of decompress into a program purely using core ATL language constructs, including the scattering construct we introduced. In doing so, we introduce a loop to iterate over the sparse positions of the compressed tensor rather than the dense coordinates. Finally, we have arrived at an ATL program computing and iterating over the compressed sparse structures of some format that is formally proven to be semantically equivalent to the original ATL program computing over dense operands. Moreover, this program can now be further optimized using the existing ATL scheduling-rewrite framework.

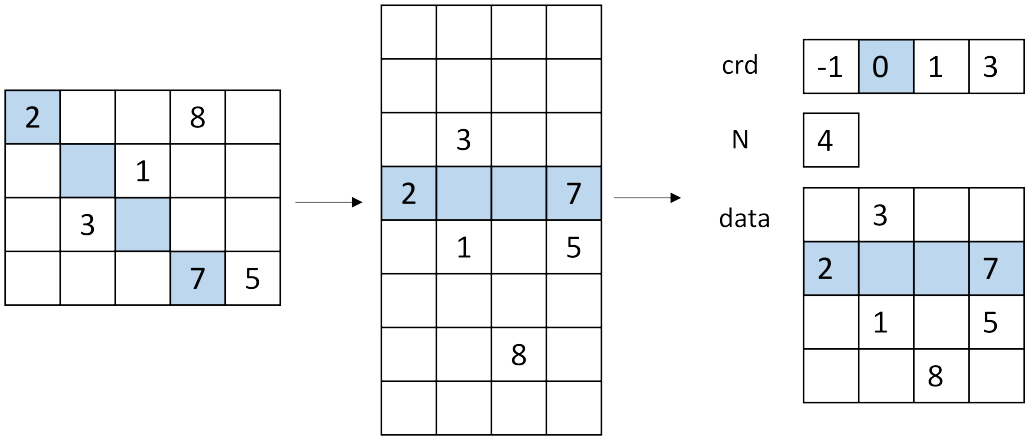
## 7.6 Integrating Reshape Operators

This set of level-format instances we have provided in this framework is powerful and composes to create numerous multidimensional sparse formats. However, there are some common, idiomatic formats that cannot be expressed through their composition alone. For example, DIA format [78] is a specialized sparse format in which a matrix is stored by its diagonals with empty diagonals being compressed out. No combination of the level-format instances implemented above can produce this layout. In fact, no combination of level-format instances that abide by the algebraic dimensional-isolation property enforced by levelization can produce this layout. The levelization inherently enforces independence between dimensions of a tensor so that information about an element's coordinate or position in one level does not affect its position in another. This separation of concerns ensures modularity and compositional reasoning but limits expressiveness. Indeed, the TACO formats system [14] has to introduce level-format instances including Offset, Range, Singleton and a system of format decorators that complicate the format system and weakens levelization in these cases.

Rather than implementing more level formats or breaking the levelization enforced by the current abstraction, we integrate ATL's reshape operators into the sparse framework to

| Format | compress Call |
|---|---|
| CSF/DCSR | compress [c; ...; c; s] $t$ |
| CSR | compress [d; c; s] $t$ |
| HASH | compress [h; s] $t$ |
| Bitmask | compress [b; s] $t$ |
| CSC | compress [d; c; s] $(t^T)$ |
| BCSR | compress [d; c; d; d; s] $\left(\boxplus_{i=0}^{k_1} \boxplus_{j=0}^{k_2} ((\text{split } k_1 \ ((\text{split } k_2 \ t)[j])^T)[i])^T\right)$ |
| CSB | compress [d; d; c; c; s] $\left(\boxplus_{i=0}^{k_1} \boxplus_{j=0}^{k_2} ((\text{split } k_1 \ ((\text{split } k_2 \ t)[j])^T)[i])^T\right)$ |
| DIA | compress [c; d; s] (shear $t$) |
| COO | compress [c; s] (flatten $t$) |
| mode-generic | compress [c; d; s] $\left(\text{flatten } \left(\boxplus_{i=0}^{n} t[i]^T\right)\right)$ |

Figure 7.6: Compositional level-format derivation of multidimensional formats




achieve the same expressive power through verified rewrites. This combination allows us to represent a broader family of sparse formats, including DIA, without breaking the levelization and modularity properties of the abstraction. Specifically, the addition of shear and unshear reshape operators enables expressing DIA format naturally within ATL. We conceptualize the conversion of a dense matrix into DIA format via the process shown above. We first shear the tensor so that its diagonals become rows, then compress out empty rows while storing the indices so that the main diagonal corresponding to index 0. Each row (formerly a diagonal) remains dense.

To achieve this result formally, we apply the reshape-adjoint theorem for shear and unshear followed by the decompress–compress adjoint theorem via the series of rewrites shown below. The level formats we pass to compress will be compressed, dense, and then scalar.

$$t \to \text{unshear } (\text{shear } t) \to \text{unshear } (\text{decompress } (\text{compress } [c; d; s] \ (\text{shear } t)) \ 0)$$

This reshape rearrangement in combination with the level formats passed to the compress

function will produce the sparse structures of $t$ stored in DIA format. The structures can be bound in the program like in the unreshaped example cases, and likewise, decompress can be unfolded and rescheduled as a normal ATL program. By combining a minimal set of proven level-format instances with the reshape operators of the ATL framework, we can express numerous multidimensional sparse formats. We derive some of the most common formats including CSF (compressed sparse fiber) or DCSR (doubly compressed sparse row), CSR (compressed sparse row), HASH, bitmask, CSC (compressed sparse column), BCSR (block compressed sparse row), CSB (compressed sparse block), DIA, COO (coordinate list), and mode-generic. These formats and the corresponding calls to compress used to express them are shown in Figure 7.6.

As currently formulated, the level-format abstraction is limited to structural compression schemes whose encoding depends only on index structure and not on the tensor values themselves. As a result, it can not directly capture value-dependent compression formats such as run-length encoding (RLE) or specialized schemes for structured tensors such as symmetric matrices. The present formulation can be understood as a restricted instance of value-dependent compression in which the distinguished value is fixed implicitly to zero. Extending the abstraction to admit compression policies parameterized by program-defined values is a natural direction for future work and would broaden its expressiveness and allow the same framework to capture a wider class of value-sensitive sparse representations.

## 7.7 Conclusion

We present the first mechanized, formally verified algebraic framework for compositionally constructing sparse tensor formats. We implemented it as an extension of the ATL scheduling framework, which had previously only supported dense tensor operands [49,50]. Our extension includes a formalized abstraction for levelized encoding schemes on tensors that are used to build multidimensional sparse tensor formats. Additionally, this framework includes format-agnostic soundness theorems that allow us to generate sparse kernel code, derived to be semantically equivalent to an original dense tensor program. We show that a more-minimal set of core level formats than previous works [14], combined with the existing concept of reshape operators from ATL, is sufficient to recover numerous common tensor compression formats. Overall, we demonstrate that compiling the derived sparse ATL programs using a minimally extended version of the existing ATL lowering algorithm yields performance competitive with TACO, a state-of-the-art sparse tensor compiler.

# Chapter 8

# Results and Evaluation

This chapter evaluates ATL as a unified framework for optimizing tensor programs through verified program transformations. Rather than evaluating scheduling, lowering, and sparse optimization independently, we organize the evaluation around scheduling dense optimizations, scheduling sparse optimizations, and lowering optimized programs, and how these components together enable expressive transformations while maintaining correctness guarantees.

Our evaluation demonstrates three central properties of the ATL scheduling framework:

- Expressivity capturing complex scheduling transformations capable of achieving performance comparable to those available in state-of-the-art tensor DSLs.
- Verified lowering and pad-type reasoning enabling these transformations in realistic optimized programs.
- Expressivity and compositionality of the sparse format abstraction to derive sparse tensor kernels with common sparse formats attaining performance comparable to specialized sparse tensor compilers.

We begin by evaluating dense scheduling transformations and their performance characteristics. We then evaluate the correctness infrastructure supporting these transformations through pad-type inference and lowering. Finally, we evaluate ATL's sparse tensor extension against the widely-used sparse tensor compiler TACO.

## 8.1 Evaluation of Dense Tensor-Program Scheduling Optimizations

Producing efficient implementations for the computational pipelines of interest in image-processing and tensor kernels generally involves navigating the trade-off space of schedules along the axes of locality, repeated computation, and parallelism [60]. In this section we demonstrate the capability of our framework to do so by showing its expressiveness and flexibility to perform complex scheduling transformations on three programs spread across this trade-off space. This includes one example whose scheduling transformations correspond to the core features of Halide (Section 8.1.1) and two which are significantly beyond the

scope of Halide's scheduling language (Sections 8.1.2 & 8.1.3), demonstrating our system's generality and extensibility. Additionally, to demonstrate that our system is capable of generating comparably optimized code to state-of-the-art systems, for the programs to which it applies, we provide performance benchmarks against their equivalents written in Halide.

### 8.1.1 Blur

We begin by investigating the performance of a number of possible optimized schedules produced by our framework and Halide. We evaluate the expressivity of our framework and the efficacy of its optimizations on another pipeline program very similar in form to the running example we have been rescheduling. We evaluate an unnormalized blur function with a $3 \times 3$ kernel that can be expressed as the composition of two functions: `blur_x` and `blur_y`.

$$\mathsf{blurx}\ v\ x\ y := v[y; x-1] \oplus v[y; x] \oplus v[y; x+1]$$

$$\mathsf{blury}\ v\ x\ y := \mathsf{blurx}\ v\ x\ (y-1) \oplus \mathsf{blurx}\ v\ x\ y \oplus \mathsf{blurx}\ v\ x\ (y+1)$$

Although this algorithm is slightly more complex, the rewrites we will perform to reschedule it will be similar in nature to those used to reschedule the smaller pipeline. The scheduling proof script for this transformation is quite verbose due to common repeated patterns of rewrites, especially when passing through intermediary states that require similar structural transformations. However, these finer details can easily be factored out into higher-level tactics to carry out common operations more concisely.

One thing to note is that if this algorithm were to be applied over the full extent of an input–say a blur over an entire image–there would be out-of-bounds accesses along the border since each stage operates over a nontrivial window. In order to remain consistent, both Halide and our framework use the strategy of conditionally guarding these boundary accesses and returning 0 for what would be out-of-bounds accesses and partitioning each resulting loop into a constant-state region and more logically complex prologue and epilogue loops to deal separately with the boundary conditions.

#### Two-Stage Blur

The first schedule we examine is a two-stage schedule, whose corresponding representation in ATL is shown in Figure 8.1. This schedule first computes the full extent of the blurx function as defined above over the input image in an intermediary buffer. It then computes the output in the second stage blury from the buffer produced from the first stage. This type of schedule is a common strategy in handwritten pipelines and often results from composing separate routines, since each function that comprises the pipeline is computed in full in a breadth-first manner [60]. While this approach has the benefit of ample opportunity for parallelization, it is lacking in properties of computational locality since every value in the first stage is computed and stored before any are used in the computation of the second stage.

#### Tiled Blur

While total fusion and breadth-first scheduling represent two scheduling extremes across considerations of redundant computation and locality, a tiled strategy is able to take advantage

$$
\begin{aligned}
\textsf{let}\ buf \ :=\ & \boxplus_{y=0}^{1} .. \circ \\
& \boxplus_{y=1}^{n+1} \left( \boxplus_{x=0}^{1} .. \circ \boxplus_{x=1}^{m-1} v[y-1;x-1] \oplus v[y-1;x] \oplus v[y-1;x+1] \circ \boxplus_{x=m-1}^{m} .. \right) \circ \\
& \boxplus_{y=n+1}^{n+2} .. \\
\textsf{in}\ & \boxplus_{y=0}^{n} \boxplus_{x=0}^{m} buf[y;x] \oplus buf[y+1;x] \oplus buf[y+2;x]
\end{aligned}
$$

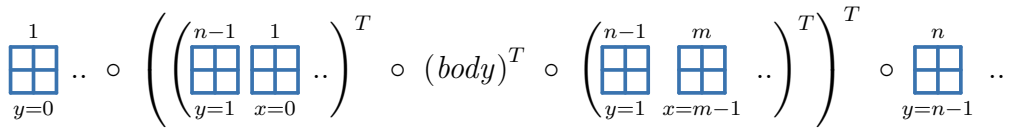
Figure 8.1: Breadth-first schedule expressed in ATL

of both properties to a certain degree. In this strategy, within an iteration, a section of the output called a tile is processed at once, and the corresponding first-stage values are computed and stored for the tile. The corresponding code for this schedule utilizing a $k_n \times k_m$ tile size is shown in Figure 8.2. While there is still redundant computation between tiles along their borders, within a tile all values computed from the first stage are stored and persistent across all iterations of computation, producing the output in the second stage. Likewise, the finer granularity the tiles provide relative to the fully breadth-first two-stage approach leverages greater locality since values computed in the first stage are used within the span of one tile. Additionally, there is still abundant opportunity for parallelism both within and between tiles.

### Performance Benchmarks

In Figure 8.3, we compare the performance of code generated by our framework and code produced by Halide for the blur algorithm with each of the schedules described above. For each benchmark, we juxtapose the performance of a specific ATL schedule with the corresponding Halide schedule. Tile sizes were set at $64 \times 64$ for both ATL and Halide. The outer loops of the benchmark programs were parallelized, and vectorization was left to the downstream C compiler for ATL programs and autovectorization for Halide programs. C code generated from ATL was compiled by clang 12.0 with openmp, fast−math, and O3 flags enabled. A snapshot of Halide was taken in early June 2021 and built against LLVM 12.0. Halide tiling was set to use the GuardWithIf strategy for loop tails/epilogues. The benchmark was performed using 2000×2000 input and output buffers on an iMac Pro with a 3.2 GHz 8-Core Skylake Xeon processor.

It can be seen that for both schedules, our system is able to express a schedule achieving comparable performance to a roughly equivalent schedule programmed in Halide. Results do not match precisely for a number of potential reasons: First, Halide automates the splitting off of loop epilogues and prologues, whereas ATL places the way in which to do perform this optimization under the control of the programmer. Second, rather than compiling to C code that goes through the clang front-end and standard optimization configurations, Halide directly targets LLVM intermediate code and uses a custom configuration of downstream optimization passes. More importantly, this comparison to Halide (an existing high-performance, user-



*body* is defined as:

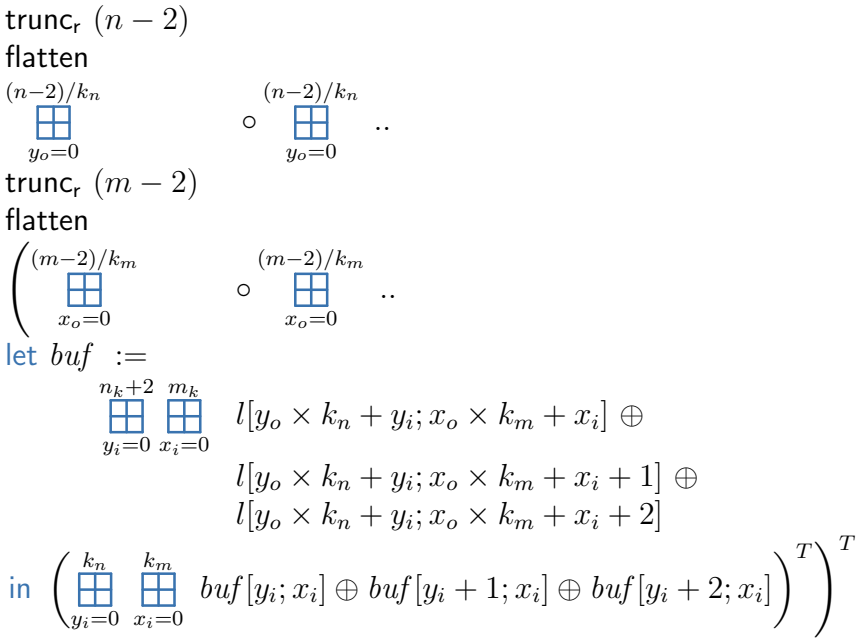


Figure 8.2: Tiled schedule expressed in ATL

| | Halide | ATL |
|---|---|---|
| two-stage blur | 3.75 ms | 3.71 ms |
| tiled blur | 1.13 ms | 1.24 ms |

Figure 8.3: Performance of schedules for the blur algorithm, our system vs. Halide

schedulable DSL) demonstrates that our language prototype is expressive enough to be used for generating competitive high-performance code.

### 8.1.2 Scatter-to-Gather Optimization

One pattern often found in computational pipelines for array processing, including image processing and deep learning, involves outputs that must read and compute multiple input values; this process is called a *gather*. The natural dual to this idiom, called a *scatter*, is an operation where each input writes to multiple elements in the output. Gathers are often more efficient than scatters, especially in the presence of parallelism, where scattering requires atomic operations to prevent data races.

When computing the gradients in reverse automatic differentiation, computations written purely in terms of gather produce scatters in the differentiated result [5,47]. As a result, scatter-to-gather loop optimizations are particularly useful when optimizing and simplifying derivative code. However, this kind of program transformation lies outside the expressive range of existing user-schedulable languages such as Halide, requiring ad-hoc workarounds in order to support automatic differentiation [47].

In this section we demonstrate using a simple example where our rewrite framework is capable of performing this transformation through a series of simple rewrites. Consider the simple scattering program shown below:

$$\sum_{i=0}^{W} \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \sum_{c=0}^{C} \boxplus_{p=0}^{W} x[n;c;i] \times ([\![\ i-p < R \wedge 0 \leq i-p\ ]\!] \cdot w[k;c;i-p])$$

Notably, the form of the scatter involves an outermost summation that cannot be naively parallelized and would likely hurt performance. However, we are able to reschedule this program into a more parallelizable gather program by applying a series of high-level rewrites, verified within our framework, with the structural ease and abstraction of a paper proof. More specifically, we apply the sequence of step-by-step transformations shown in Figure 8.4. Each line corresponds to one rewrite written in our rescheduling framework, resulting in the final program shown below:

$$\boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} [\![\ p+r < W\ ]\!] \cdot x[n;c;p+r] \cdot w[k;c;r]$$

In this equivalent form, the summations have been moved inside the loop nests, and the offset when accessing $w$ has been replaced by an offset when indexing $x$. The outermost generation loop is now amenable to thread-level data parallelism. This example demonstrates that our rewrite framework is capable of expressing scatter-to-gather optimizations.

$$
\begin{aligned}
& \sum_{i=0}^{W} \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \sum_{c=0}^{C} \boxplus_{p=0}^{W} x[n;c;i] \cdot ([\ i-p < R \wedge 0 \le i-p\ ] \cdot w[k;c;i-p]) \\
= & \sum_{i=0}^{W} \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \sum_{c=0}^{C} \boxplus_{p=0}^{W} [\ i-p < R \wedge 0 \le i-p\ ] \cdot (x[n;c;i] \cdot w[k;c;i-p]) \\
= & \sum_{i=0}^{W} \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \sum_{c=0}^{C} \boxplus_{p=0}^{W} \sum_{r=0}^{R} [\ r = i-p\ ] \cdot x[n;c;i] \cdot w[k;c;r] \\
= & \sum_{i=0}^{W} \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} [\ r = i-p\ ] \cdot x[n;c;i] \cdot w[k;c;r] \\
= & \boxplus_{n=0}^{B} \sum_{i=0}^{W} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} [\ r = i-p\ ] \cdot x[n;c;i] \cdot w[k;c;r] \\
= & \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \sum_{i=0}^{W} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} [\ r = i-p\ ] \cdot x[n;c;i] \cdot w[k;c;r] \\
= & \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{i=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} [\ r = i-p\ ] \cdot x[n;c;i] \cdot w[k;c;r] \\
= & \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{i=0}^{W} \sum_{r=0}^{R} [\ r = i-p\ ] \cdot x[n;c;i] \cdot w[k;c;r] \\
= & \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} \sum_{i=0}^{W} [\ r = i-p\ ] \cdot x[n;c;i] \cdot w[k;c;r] \\
= & \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} \sum_{i=0}^{W} [\ i = p+r\ ] \cdot x[n;c;i] \cdot w[k;c;r] \\
= & \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} [\ 0 \le p+r \wedge p+r < W\ ] \cdot x[n;c;p+r] \cdot w[k;c;r] \\
= & \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} [\ p+r < W\ ] \cdot (x[n;c;p+r] \cdot w[k;c;r])
\end{aligned}
$$

Figure 8.4: Scheduling-rewrite sequence for a scatter-to-gather optimization

### 8.1.3 Im2col

Early convolutional neural networks exploited high-throughput GPUs by transforming batched convolution operations into matrix multiplies. This transformation worked by first marshalling and duplicating the input tensor of images and then using existing BLAS subroutines to perform matrix-matrix multiplies at close to peak machine utilization. Later, GPUs were extended with tensor operations (smaller granularity matrix-matrix multiplies) and/or replaced with tensor processing units (also implementing matrix-matrix multiply) to speed up machine-learning pipelines. Throughout these changes, the data-marshalling-and-duplication transformation (known as im2col, after the Matlab function) has been essential to exploiting specialized hardware and hand-optimized subroutines.

Usually, the im2col transformation is explained in a series of diagrams that attempt to show how an image is first packed into a 1D vector, then duplicated and offset to account for translation within the image domain. Such explanations are further complicated by the presence of additional channel and batch dimensions within standard specifications of neural-network convolution operations. In Figure 8.5, we show how this transformation can be achieved and explained purely algebraically, via rewriting in our system. For simplicity we show a 1D version of convolution. The essence of the transformation comes down to binding the read-with-offset subexpression $x[n; c; p + r]$ into an intermediary variable named $a$ and then simply hoisting this intermediary outside of the nested loops. The resulting computation of the intermediary $a$ then expresses what is commonly known as the im2col operation, while the remaining body expresses a standard matrix-matrix multiply.

As with our scatter-to-gather example, this kind of program transformation falls outside the expressive limits of Halide and similar languages such as TVM. By incorporating it into an expressive scheduling framework, we also make it possible to apply this transformation at different intermediate levels of tiling and memory hierarchy, depending on the granularity of the accelerated matrix-matrix subroutine being targeted.

## 8.2 Evaluation of Pad-Type Inference and Lowering Algorithm

We evaluate our implementation of the lowering algorithm and tactic machinery for type-checking on various programs. We include the programs that were used in the evaluation of the scheduling-rewrite framework in Section 8.1 to evaluate the scheduling expressivity of the ATL optimization framework, with scheduled programs for a number of algorithms including the two-dimensional box blur and various convolutional programs that demonstrate gather-to-scatter and im2col transformations. We previously used these programs to evaluate the performance and scheduling expressivity of the rewrite optimization framework and lowering as a whole. In contrast, we use these programs to evaluate the implementation of the lowering algorithm and the effectiveness of our pad-type system and associated tactic machinery on various programs. We focus on optimizations that emphasize the use of reshape operators in nontrivial ways. We also introduce a new collection of additional programs that represent other common program structures and optimizations.

In Figure 8.6 we list each program we used to evaluate our pad type system and compiler

$$
\begin{aligned}
& \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} w[k;c;r] \times x[n;c;p+r] \\
= \; & \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} \mathsf{let}\ a := x[n;c;p+r]\ \mathsf{in}\ w[k;c;r] \times a \\
= \; & \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \mathsf{let}\ a := \boxplus_{r=0}^{R} x[n;c;p+r]\ \mathsf{in}\ \sum_{r=0}^{R} w[k;c;r] \times a[r] \\
= \; & \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \mathsf{let}\ a := \boxplus_{c=0}^{C} \boxplus_{r=0}^{R} x[n;c;p+r]\ \mathsf{in}\ \sum_{c=0}^{C} \sum_{r=0}^{R} w[k;c;r] \times a[c;r] \\
= \; & \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \mathsf{let}\ a := \boxplus_{p=0}^{W} \boxplus_{c=0}^{C} \boxplus_{r=0}^{R} x[n;c;p+r]\ \mathsf{in}\ \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} w[k;c;r] \times a[p;c;r] \\
= \; & \boxplus_{n=0}^{B} \mathsf{let}\ a := \boxplus_{p=0}^{W} \boxplus_{c=0}^{C} \boxplus_{r=0}^{R} x[n;c;p+r]\ \mathsf{in}\ \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} w[k;c;r] \times a[p;c;r] \\
= \; & \mathsf{let}\ a := \boxplus_{n=0}^{B} \boxplus_{p=0}^{W} \boxplus_{c=0}^{C} \boxplus_{r=0}^{R} x[n;c;p+r]\ \mathsf{in}\ \boxplus_{n=0}^{B} \boxplus_{k=0}^{K} \boxplus_{p=0}^{W} \sum_{c=0}^{C} \sum_{r=0}^{R} w[k;c;r] \times a[n;p;c;r]
\end{aligned}
$$

Figure 8.5: Scheduling-rewrite sequence for an im2col optimization on a simple convolution

implementation along with occurrence counts for different language operators. We do so to try and capture the diverse reshape complexity as well as the pad-pattern complexity that our type system is able to account for to make a case for its applicability on desirable real-world optimizations.

To explore the extent to which certain reshape-operator patterns and idiomatic optimizations affect padding, compute order, and storage order, we will discuss in-depth the two new evaluation programs for computing matrix multiplication and tensor addition, as well as the most sophisticated prior example, the tiled box-blur program.

| Program | # of Gen (⊞) | # of Concat | # of Truncate | # of Transpose | # of Flatten | # of Split |
|---|---|---|---|---|---|---|
| Gather | 3 | - | - | - | - | - |
| Scatter | 3 | - | - | - | - | - |
| Im2col Conv | 3 | - | - | - | - | - |
| Im2col Mat | 7 | - | - | - | - | - |
| Matmul | 2 | - | - | - | - | - |
| Tiled Matmul | 4 | - | 2 | 1 | 2 | - |
| Tiled+Tails Matmul | 10 | 2 | 2 | 1 | 2 | - |
| Two-Stage Blur | 4 | - | - | - | - | - |
| Fused Blur | 2 | - | - | - | - | - |
| Fused+Tails Blur | 8 | 4 | - | - | - | - |
| Tiled+Tails+Staged Blur | 21 | 6 | 3 | 7 | 3 | - |
| Tensor Add | 4 | - | - | - | - | - |
| Split Tensor Add | 1 | - | - | - | - | 3 |

Figure 8.6: Reshape-operator complexity of evaluated programs

### 8.2.1 Matrix Multiplication

Matrix multiplication is a fundamental operation in linear algebra. It involves multiplying corresponding elements of each row of one matrix with the elements of the corresponding column from the second matrix and summing up these products.

#### Standard Matrix Multiplication

The following program is a standard matrix multiplication written in ATL.

$$\boxplus_{i=0}^{M} \boxplus_{j=0}^{N} \sum_{k=0}^{K} m_1[i;k] * m_2[k;j]$$

Provided that the tensor-generation bounds match input matrices' dimensions, this program trivially typechecks and passes our safety preconditions for compilation as it introduces no padding and performs no truncation operations.

#### Tiled Matrix Multiplication

For our optimized evaluation program, we focus on the technique of tiling, a common optimization that increases data locality within a program. It also involves a sophisticated usage of reshape operators. As we have discussed, it introduces padding when the tiling factor does not evenly divide the tiled dimension. The optimized program below is the tiled equivalent of the matrix-multiplication program from above that we achieved through applying rewrites from ATL's verified scheduling framework.

$$\mathsf{trunc_r}\ (C - M\%C) \quad \Big( \mathsf{flatten} \Big( \boxplus_{i_o=0}^{M//C} \Big( \mathsf{trunc_r}\ (C - N\%C) \quad \Big( \mathsf{flatten} \Big( \boxplus_{j_o=0}^{N//C} \boxplus_{i_i=0}^{C} \boxplus_{j_i=0}^{C} [\ j_o * C + i_i < N \wedge i_o * C + i_i < M\ ] \cdot \sum_{k=0}^{K} m_1[i_o * C + i_i; k] * m_2[k; j_o * C + j_i] \Big) \Big) \Big)^T \Big) \Big)$$

The program structure includes two right-truncations–one per tiled dimension to accommodate the case where the dimensions $M$ and $N$ are not divisible by the tiling factor $C$. There is a flatten operator for each tiled dimension that collapses the storage of the new outer and inner loop dimensions. The transpose operator moves the outer loops next to each other and the inner loops next to each other.

This program properly type-checks in our pad type system. This result is notable since it is not obvious upon inspection that the truncations are safe, as they do not have matching explicit $\mathsf{pad_r}$ operators. It would not even suffice to limit the value of one explicit loop index. Instead, the safety of the truncation is determined by the guard conditions in the loop body that involve reasoning about the two inner and outer loop-index pairs.

**Tiled Matrix Multiplication with Loop Separation**

We can take our optimization a step further to extend the evaluation of our type-checking and safety mechanisms. One might notice that the innermost guard can only be false in one iteration of the index $i_o$ and one iteration of the index $j_o$. Another common optimization technique in this case is to split these loops into two segments where the guard is always true in one of the segments and can be removed. When the segment in which the guard is always true dominates the latter in size, as in this case, the optimization is very useful: it reduces the number of times the guard has to be executed and produces a main loop structure more amenable to vectorization. We can schedule this optimization to produce the program below.

$$\mathsf{trunc_r}\ (C - M\%K)\ \Bigg(\mathsf{flatten}\ \Bigg(\boxplus_{i_o=0}^{M/C}\ \Bigg(\mathsf{trunc_r}\ (C - N\%C)\ \Bigg(\mathsf{flatten}\ \Bigg(\boxplus_{j_o=0}^{N/C}\ \boxplus_{i_i=0}^{C}\ \boxplus_{j_i=0}^{C}\ [\ j_o * C + i_i < N \wedge i_o * C + i_i < M\ ]\cdot \sum_{k=0}^{K} m_1[i_o * C + i_i; k] * m_2[k; j_o * C + j_i]\Bigg) \circ \boxplus_{j_o=N/C}^{N//C} \ldots\Bigg)\Bigg)^T\Bigg) \circ \boxplus_{i_o=M/C}^{M//C} \ldots\Bigg)$$

In addition to the reshape-operator complexity involved in the previous tiled matrix-multiplication example, this program uses the concatenation reshape operator to join the main loop with its tail regions. Being able to use concatenation in a nontrivially reshaped program like this one with padding is one of the primary cases that motivated us to adopt a tree-like pad type, decoupling the inner pad term on the left from the right. Our system is indeed able to type and compile this program.

### 8.2.2 Tensor Addition

In another example, we compute the sum of two tensors. We use an elementwise sum between two tensors with a different loop iterating for each tensor dimension. This kind of loop with a pointwise inner computation is a common target for optimization through parallelization.

$$\boxplus_{i=0}^{M}\ \boxplus_{j=0}^{N}\ \boxplus_{k=0}^{K}\ \boxplus_{l=0}^{C}\ t_1[i; j; k; l] * t_2[i; j; k; l]$$

The depth of this loop nest will increase linearly with the dimensionality of the input tensors. The control-flow complexity is excessive since physically we will be iterating straight through the flattened tensors in the heap. As a result, there will be unnecessary overhead introduced by each loop—especially high if we want to parallelize them all. Instead, we can

use scheduling rewrites to produce the following program.

$$\mathsf{split}\ K \left( \mathsf{split}\ N \left( \mathsf{split}\ C \left( \boxplus_{i=0}^{MNKC} t_1[i/(NKC); i/(NC)\%K; i/C\%N; i\%C] * t_2[i/(NKC); i/(NC)\%K; i/C\%N; i\%C] \right) \right) \right)$$

The loop nest has been flattened into one loop. The split reshape operator allows us to tile the storage, thereby producing the desired higher-dimensional tensor sum. Although we demonstrate this optimization on four-dimensional tensors, it can be applied to tensors of any dimension. Both the initial and optimized program here are properly safety- and type-checked by our system and are properly compiled.

### 8.2.3 Blur

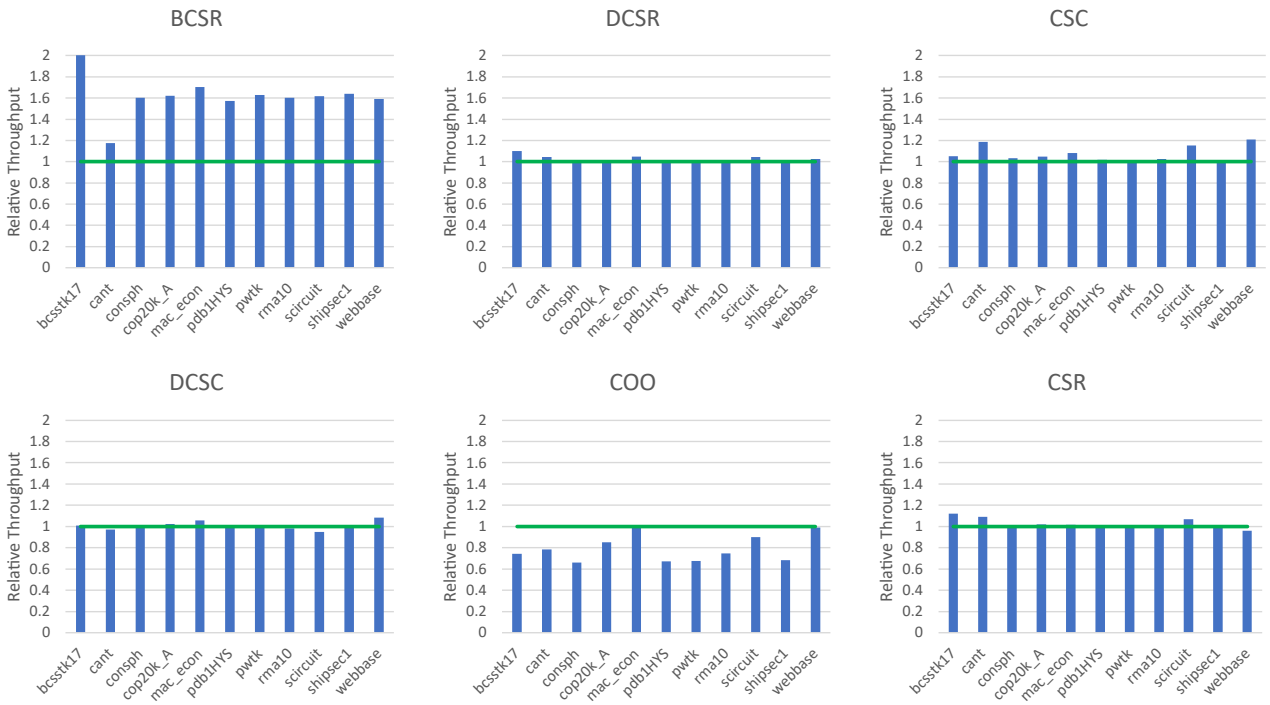
We examine the same blur algorithm used in the evaluation in Section 8.1. Like the tiled matrix-multiplication example, tiling introduces guards. However, the tiling in this program occurs across computational stages, which yields additional guards since a stencil involves a neighborhood of computation. These guards are individually split into separate loop segments to produce one main, steady-state loop without any guards. Not only does this example cover a common programming pattern and optimization, it uses reshape operators extensively to achieve this structure. This reordering interacts nontrivially with the loop-splitting optimizations that separate out loop tails, thereby sharding padding within the tensor. We found that our pad type system is able to accept each of these scheduled forms of the blur algorithm and that compilation succeeds.

## 8.3 Evaluation of Sparse-Format Tensor-Program Optimizations

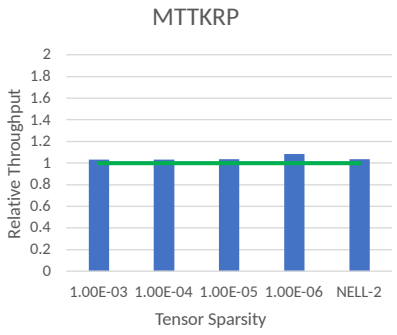
We evaluate the ATL-derived sparse kernels against the state-of-the-art sparse tensor compiler, TACO [39]. We show that ATL's sparse extension can express a wide range of sparse formats and generate kernels with efficient sparse iteration strategies, achieving desirable performance. Our goal is to obtain performance and format-expressivity comparable to TACO to show we have developed a verified, extensible algebra capable of producing similar code.

Our experiments target sparse matrix-vector multiplication (SpMV), sparse matrix-sparse vector multiplication (SpMSpV), sparse tensor-times-vector (TTV), sparse tensor-times-sparse vector (SpTSpV), and sparse matricized-tensor times Khatri-Rao product (MTTKRP). We focus on kernels generating dense outputs since writing into sparse structures through our format algebra remains future work.

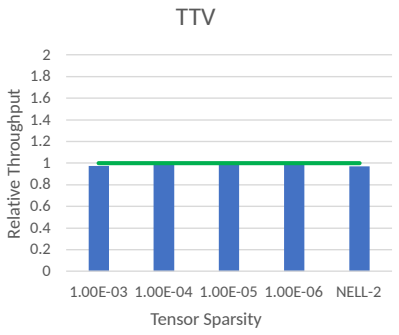
Figure 8.7 reports the throughput of ATL-generated programs relative to TACO on the same formats and input tensors. The green line represents the performance of TACO, or a relative throughput (speedup) of 1. We ran all experiments as single-threaded programs on a 2.8GHz Intel Core i5-8400 machine with 9 MB of L3 cache and 24 GB of main memory. All programs were compiled using `clang 17` with `-O3` and `-ffast-math` flags enabled.



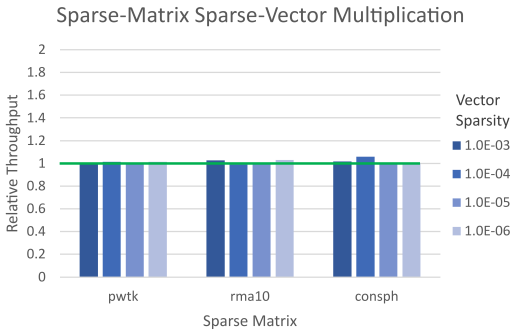

(a) Sparse matrix-vector multiplication



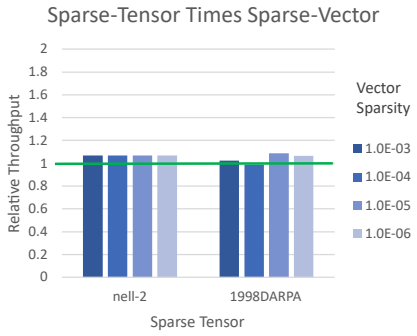

(b) Matricized tensor times Khatri-Rao product




(c) Tensor times vector




(d) Sparse matrix-sparse vector multiplication




(e) Sparse tensor-sparse vector multiplication

Figure 8.7: Relative speedup of ATL-generated kernels against TACO-generated kernels of the same format

### 8.3.1 Sparse Matrix Computations

We evaluate sparse matrix–vector multiplication (SpMV) to showcase the breadth of our framework's sparse-matrix-format support, as it is both ubiquitous in sparse linear algebra and used in TACO's own evaluation. Using the sparse format abstraction and the ATL rewrite framework, we derived ATL programs with input matrices stored in block-compressed sparse row (BCSR), double-compressed sparse row (DCSR), compressed-sparse column (CSC), double-compressed sparse column (DCSC), coordinate-list (COO), and compressed sparse row (CSR) formats. We chose this set of formats as they are the ones readily available in the stable release build of TACO [14]. Our experiments used real-world sparse matrix inputs from the SuiteSparse Matrix Collection [17]. These matrices ranged in size from $10^8$ to $10^{12}$ total elements (including zeros) and nonzero densities from $3 \times 10^{-6}$ to $4 \times 10^{-3}$.

In Figure 7(a) we show the evaluation results for each format in a separate graph. Overall, we found that ATL-generated code is comparable to TACO's performance across formats, except for COO and BCSR. We analyze both cases below.

#### COO Results

Figure 7(a) shows that for various input matrices ATL-generated code for COO has worse throughput than TACO with a geometric-mean relative throughput across all workloads of 0.78. The reason is likely an additional optimization performed by the TACO compiler. TACO recognizes that the coordinate arrays in COO are ordered and non-unique. Therefore, when TACO is reducing into coordinates from this array, it batches a set of reductions to one output index by searching for the end of the segment containing the same index. The effectiveness of this optimization varies depending on the sparsity pattern of the matrix. The overhead of the loop searching for the end of a segment is outweighed only if there were a significant number of nonzero elements with that given coordinate. This difference explains why the magnitude of the peformance gap between TACO and ATL varies. Currently, this optimization is not expressible in ATL, but we acknowledge it as valuable future work.

#### BCSR Results

The ATL-generated code for BCSR is able to outperform TACO with a geometric mean of relative throughput across all inputs of 1.6. In TACO, generating a BCSR kernel requires manually constructing a 4-D tensor format, so the compiler generates inner loops whose extents correspond to input tensor dimensions. It lacks knowledge that these inner dimensions are the tiling factors. In contrast, a programmer using the ATL framework begins from a 2-D tensor and introduces tiling through rewrites, making the tile size explicit at compile time. Consequently, ATL emits loops with constant extents, which downstream compilers like `clang` can better optimize.

### 8.3.2 Sparse Tensor Computations

We chose TTV and MTTKRP as the focus of our higher-order tensor evaluation because they are widely used building blocks in sparse tensor computations. TTV is a core operation in many tensor-decomposition algorithms, while MTTKRP is the computational bottleneck

in canonical polyadic (CP) decomposition, a common tensor-factorization algorithm. Figures 7(b) and 7(c) report results for TTV and MTTKRP kernels, shown in separate graphs. All input tensors were third-order tensors stored in CSF format. We ran these kernels on a range of synthetic tensor sparsities from $10^{-3}$ to $10^{-6}$ with sizes of around $10^9$ total elements and the real-world sparse tensor NELL-2 from the FROSTT tensor collection [69], which has a size of about $3 \times 10^{12}$ total elements and a nonzero density of approximately $2 \times 10^{-5}$. Across both kernels and all inputs, ATL-generated kernels consistently matched the performance of TACO equivalents.

### 8.3.3 Computations with Multiple Sparse Operands

We also derived a sparse-matrix sparse-vector multiplication kernel (Figure 7(d)) and a sparse-tensor times sparse-vector kernel (Figure 7(e)), to demonstrate the ability of our framework to support multiple sparse operands. For these benchmarks, we chose a subset of the real-world sparse matrices and sparse tensors from the single-sparse-operand tests that had the largest dimension sizes and most nonzeros. The matrices were stored in CSR and the tensors in DCSR and multiplied against synthesized sparse vectors stored in compressed format at a range of sparsities ranging from $10^{-3}$ to $10^{-6}$. Across both kernels and all inputs, ATL-generated kernels consistently matched the performance of TACO equivalents.

# Chapter 9

# Related Work

This chapter situates our work within the broader landscape of tensor-programming systems, scheduling frameworks, compiler verification, and sparse tensor computation. While prior work has made substantial advances in each of these areas, they have largely been explored independently. In contrast, our approach unifies scheduling, lowering, and sparsity within a single framework based on algebraic program transformations, allowing these concerns to interact in a principled and composable way.

## 9.1 Scheduling Languages and Tensor-Programming Systems

Languages and compilers that provide explicit programmer control over program transformations have a long history in high-performance computing [12,20,22,29,77]. These systems expose transformation primitives such as loop tiling, fusion, and parallelization, enabling programmers to optimize performance-critical code. However, most such systems provide limited guarantees of correctness, placing the burden on the programmer to ensure that transformations preserve semantics.

Halide introduced a particularly influential design by separating the specification of a computation from its execution strategy [59,60]. In Halide, the programmer writes a functional description of the computation and then specifies a schedule that determines how that computation is executed. This separation has proven highly effective, enabling both high performance and a clear programming model. As a result, Halide has inspired a wide range of subsequent systems across multiple domains, including dense linear algebra and machine learning [13,28,73,74], sparse tensor algebra [39], distributed-memory computing [4], graph processing [79], and physical simulation [32].

Despite their success, these systems share several limitations. First, they typically provide no formal guarantees that optimized programs are equivalent to their specifications, and in some cases they allow transformations that are unsound. Second, their scheduling languages are fixed: extending the space of transformations generally requires modifying the compiler itself. Finally, many of these systems are designed primarily for dense, regular computations, making it difficult to express transformations involving irregular access patterns or structural changes to the computation.

Our approach retains the key insight of separating specification from optimization but differs in how transformations are expressed. Rather than relying on a fixed scheduling language, we represent transformations as algebraic rewrite rules. This allows users to define new transformations within the system itself, while also enabling formal reasoning about their correctness.

Recent work has explored the use of rewrite rules for optimizing array programs [23,70]. These systems provide flexible transformation mechanisms but typically treat rewrites as axioms, without providing formal guarantees. In contrast, our framework integrates rewriting into a formally verified setting, ensuring that all transformations preserve program semantics.

## 9.2 Program Derivation and Verified Optimization

The idea of deriving programs through formal reasoning has a long history in programming languages and formal methods. Systems based on constructive logic allow programs to be developed alongside proofs of correctness, ensuring that the resulting implementations satisfy their specifications.

The Fiat framework [18] is a prominent example of this approach, supporting automated derivation of programs in domains such as relational queries and binary-format parsers [19]. Fiat also supports proof-producing compilation pipelines [58], enabling verified translation from high-level specifications to efficient low-level code.

Our work is complementary to Fiat but differs in several important respects. Fiat emphasizes nondeterministic specifications and automated refinement, whereas our system focuses on deterministic, step-by-step derivations of optimized tensor programs. This design makes our approach more directly applicable to performance-oriented transformations, where fine-grained control over intermediate steps is often necessary.

Other work has explored verified optimization in specialized domains. For example, the VOQC quantum-circuit optimizer [31] uses a tactic-based approach within a theorem prover to validate optimization procedures. While effective in its domain, VOQC focuses on validating predefined optimizations rather than supporting interactive derivation of new transformations.

In contrast, our system allows users to construct and verify transformations directly, at the level of individual programs. This enables a more flexible and exploratory style of optimization while still providing formal guarantees.

## 9.3 Functional Tensor Languages and Intermediate Representations

Our tensor language builds on a long tradition of array programming languages, including APL [34], ZPL [11], Chapel [10], and Remora [67]. These languages emphasize high-level, declarative descriptions of array computations, often with implicit parallelism and strong compositional structure.

More recent systems, such as Futhark [30], Accelerate [9], and Dex [56], adopt functional programming paradigms to express tensor computations while enabling efficient compilation

to parallel hardware. These languages provide powerful abstractions but typically separate high-level program structure from lower-level implementation decisions.

Our language is specifically derived from prior work on a functional tensor language for automatic differentiation [5], which we extend with additional constructs, such as reshape operators, to express a broader range of transformations. These extensions are particularly important for capturing performance-relevant program rewrites, such as changes to iteration order and data layout.

Concurrent work such as Glenside [68] similarly explores richer abstractions for tensor computations, introducing an algebra of access patterns to describe implementation details. While these approaches share the goal of increasing expressiveness, our work differs in integrating these abstractions with a verified transformation framework.

Compiler infrastructures such as MLIR provide extensible intermediate representations that support multi-level optimization. These systems improve modularity and enable sophisticated compilation pipelines, but typically maintain a separation between high-level transformations and low-level lowering decisions. In contrast, our approach represents all transformations within a single framework, allowing them to interact more directly.

## 9.4 Verified Compilation and Program Extraction

The verification of compilers has been extensively studied, with CompCert [46] representing a landmark achievement. CompCert provides a fully verified compilation pipeline from C to assembly, ensuring that generated code preserves the semantics of the source program.

However, CompCert operates at a relatively low level of abstraction, making it difficult to express and reason about high-level transformations such as those required for tensor optimization. As a result, it is not well-suited to the kinds of program transformations considered in this work.

Other projects focus on compiling functional languages. CakeML [42] provides a verified compiler with a verified runtime system, while Fiat Cryptography [21] generates efficient C code from functional specifications. While these systems demonstrate the feasibility of verified compilation, they are not designed for tensor computations and often rely on techniques such as partial evaluation that are less applicable in this domain.

Several frameworks use proof assistants to derive imperative programs from functional ones. These include work by Myreen [53], later extended in systems such as CakeML [1] and CertiCoq [55], as well as refinement-based approaches in Isabelle/HOL [43,44] and Rocq-based systems such as Fiat [18,58] and Rupicola [57].

While these systems provide powerful tools for verified program construction, they have not been widely applied to tensor kernels. Our work addresses this gap by providing a verified compilation framework specifically designed for tensor computations.

## 9.5 Loop Transformations and Verified Compilation for Arrays

Optimizing loop nests and array programs has been a central concern in compiler research. The polyhedral model provides a formal framework for reasoning about loop transformations, enabling optimizations such as tiling, fusion, and parallelization.

Recent work [16] has explored verified compilation based on the polyhedral model, providing correctness guarantees for transformations over affine loop nests. While powerful, these approaches are typically limited to programs with regular structure.

Other work focuses on verifying transformations within existing systems. For example, Clément and Cohen [15] apply translation validation to Halide programs, checking correctness after optimization. This approach provides stronger guarantees than unverified compilation but does not establish full compiler correctness.

Additional work explores alternative representations for tensor computations. Kovach et al. [41] introduce indexed streams as an intermediate representation for tensor contractions, supporting sparse computations but focusing on a restricted class of programs.

In contrast, our approach supports a broader range of transformations, including those involving reshaping and reordering, while providing full correctness guarantees.

## 9.6 Sparse Tensor Computation

Sparse tensor computation has been extensively studied in both numerical computing and machine learning. Early work introduced storage formats such as CSR [71], with later extensions including ELL, ELLPACK [37], HYB [27], and blocked CSR [35,76]. Many libraries, including cuSparse [54], MKL [33], and SciPy [63], provide support for these formats.

Compiler support for sparse tensor-algebra has also seen significant advances. Early work [6,40] derived sparse implementations from dense specifications, while the TACO compiler [39] introduced a flexible approach based on tensor algebra expressions and format abstractions.

Subsequent work has extended these ideas in various directions, including more flexible format abstractions [14], alternative intermediate representations [38,64], and support for transformations such as sparse reshaping [61] and coiteration [2]. MLIR-based approaches further extend format customization [52].

More recent work explores declarative approaches to sparse computation, combining format specifications with program transformations [24,62,66]. These systems improve flexibility but generally do not provide formal guarantees of correctness.

Our work differs in integrating sparse and dense computations within a single framework. Our levelized format abstraction allows formats to be constructed from reusable components with formally specified correctness properties, enabling correct-by-construction sparse transformations.

## 9.7 Formal Methods for Sparse Computation

Formal reasoning about sparse computation has received relatively less attention. Gladshtein et al. [25] develop a separation logic for reasoning about loops over compressed data structures, while earlier work [6] uses attribute grammars to analyze sparse transformations.

These approaches provide valuable insights but do not offer a unified framework for reasoning about both sparse and dense computations. Our system addresses this gap by integrating sparse reasoning into a general framework for tensor-program transformations, enabling formal guarantees across a wider class of programs.

## 9.8 Summary

Prior work has made substantial progress in tensor programming, scheduling, compiler verification, and sparse computation. However, these areas have largely been developed independently, limiting the ability to compose transformations across abstraction boundaries.

Our work unifies these strands by providing a framework in which scheduling, lowering, and sparsity are expressed as composable, formally verified transformations over a shared representation. This integration enables both greater expressiveness and stronger correctness guarantees than existing approaches.

# Chapter 10

# Conclusion

High-performance tensor programs lie at the core of modern scientific-computing, image-processing, and machine-learning workloads. Although such programs are often expressed as simple mathematical computations over multidimensional arrays, their efficient execution depends on complex, low-level choices about iteration order, memory layout, and data representation. Existing tensor optimization systems expose many of these choices through scheduling and layout abstractions, but the correctness of the resulting transformations is typically established only through testing or informal reasoning. In my work I develop a formal framework for verified tensor optimization based on a unifying perspective: many performance-critical tensor transformations can be understood as compositional algebraic transformations of index mappings.

This work develops that perspective through four connected contributions. First, it introduces ATL, a functional tensor language in which tensor computations are represented as algebraic expressions with explicit control over compute and storage structure. ATL exposes scheduling decisions directly in the source language, allowing optimizations to be expressed as source-to-source rewrites and verified as proofs of functional equivalence in the Rocq proof assistant.

Second, it presents a verified lowering algorithm for ATL that compiles optimized functional tensor programs into imperative loop-nest code while preserving their semantics. This development introduces reindexing functions to model storage transformations during compilation and establishes the first mechanized correctness proof for lowering a functional tensor language with explicit compute–storage decoupling into efficient low-level array code.

Third, the dissertation shows that the same algebraic view of index transformation extends beyond ATL to the tensor layout abstractions used in modern GPU programming. It formalizes the CUDA CuTe layout abstraction and its associated layout algebra, showing that layout operators can be understood as compositional transformations over index spaces analogous to ATL reshape operators. This correspondence provides a formal account of layout soundness for a widely-used, industrial tensor abstraction and demonstrates that the underlying algebraic principles are not specific to ATL.

Finally, the dissertation extends this framework to sparse tensor computation by treating sparse formats as structured transformations of tensor storage order. It develops a verified abstraction for sparse tensor formats and shows how computations over dense tensors can be systematically transformed into functionally equivalent kernels over compressed

representations. This transformation is both dimensionally compositional and format-agnostic. To accomplish this, the key abstraction is a formal notion of a level format, which captures the encoding strategy for a single tensor dimension as a compositional unit of sparse representation. By building sparse formats as compositions of these per-dimension encodings, the framework supports verified reasoning about a broad family of sparse layouts within a uniform algebraic setting.

Together, these results establish a unified approach to verified tensor optimization spanning source-level scheduling, verified compilation, GPU layout abstractions, and sparse tensor representations. Across these domains, the dissertation demonstrates that tensor optimization can be treated as algebraic manipulation of index structure, enabling practical verification of high-performance tensor transformations without sacrificing expressivity or performance.

# References


[1] Oskar Abrahamsson, Son Ho, Hrutvik Kanabar, Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Yong Kiam Tan. "Proof-Producing Synthesis of CakeML from Monadic HOL Functions". *J. Autom. Reason.*, **64**(7), Oct. 2020, pp. 1287–1306. ISSN: 0168-7433. DOI: 10.1007/s10817-020-09559-8. URL: https://doi.org/10.1007/s10817-020-09559-8.

[2] 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. ISBN: 9798400701016. DOI: 10.1145/3579990.3580020. URL: https://doi.org/10.1145/3579990.3580020.

[3] *Ampere Architecture Whitepaper*. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf. 2021.

[4] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: expressing locality and independence with logical regions". In: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12*. Salt Lake City, UT, USA: IEEE, Nov. 2012, p. 66. DOI: 10.1109/SC.2012.71. URL: https://doi.org/10.1109/SC.2012.71.

[5] Gilbert Bernstein, Michael Mara, Tzu-Mao Li, Dougal Maclaurin, and Jonathan Ragan-Kelley. *Differentiating a Tensor Language*. 2020. arXiv: 2008.11256 [cs.PL].

[6] A.J.C. Bik and H.A.G. Wijshoff. "Automatic data structure selection and transformation for sparse matrix computations". *IEEE Transactions on Parallel and Distributed Systems*, **7**(2), 1996, pp. 109–126. DOI: 10.1109/71.485501.

[7] Jack Carlisle, Jay Shah, Reuben Stern, and Paul VanKoughnett. *Categorical Foundations for CuTe Layouts*. 2026. arXiv: 2601.05972 [cs.PL]. URL: https://arxiv.org/abs/2601.05972.

[8] Cris Cecka. *CuTe Layout Representation and Algebra*. 2026. arXiv: 2603.02298 [cs.MS]. URL: https://arxiv.org/abs/2603.02298.

[9] Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. "Accelerating Haskell array codes with multicore GPUs". In: *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming*. Ed. by Manuel Carro and John H. Reppy. New York, NY, USA: Association for Computing Machinery, 2011, pp. 3–14. DOI: 10.1145/1926354.1926358. URL: https://doi.org/10.1145/1926354.1926358.

[10] B.L. Chamberlain, D. Callahan, and H.P. Zima. "Parallel Programmability and the Chapel Language". *The International Journal of High Performance Computing Applications*, **21**(3), 2007, pp. 291–312. DOI: 10.1177/1094342007078442.

[11] Bradford L. Chamberlain. "The design and implementation of a region-based parallel programming language". PhD thesis. The University of Washington, 2001.

[12] Chun Chen, Jacqueline Chame, and Mary Hall. *CHiLL: A framework for composing high-level loop transformations*. Tech. rep. University of Southern California, 2008.

[13] Tianqi Chen et al. "TVM: An Automated End-to-end Optimizing Compiler for Deep Learning". In: *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*. OSDI'18. Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594. ISBN: 978-1-931971-47-8. URL: http://dl.acm.org/citation.cfm?id=3291168.3291211.

[14] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. "Format abstraction for sparse tensor algebra compilers". *Proc. ACM Program. Lang.*, **2**(OOPSLA), Oct. 2018. DOI: 10.1145/3276493. URL: https://doi.org/10.1145/3276493.

[15] Basile Clément and Albert Cohen. "End-to-End Translation Validation for the Halide Language". *Proc. ACM Program. Lang.*, **6**(OOPSLA1), Apr. 2022. DOI: 10.1145/3527328. URL: https://doi.org/10.1145/3527328.

[16] Nathanaël Courant and Xavier Leroy. "Verified Code Generation for the Polyhedral Model". *Proc. ACM Program. Lang.*, **5**(POPL), Jan. 2021. DOI: 10.1145/3434321. URL: https://doi.org/10.1145/3434321.

[17] Timothy A. Davis and Yifan Hu. "The University of Florida sparse matrix collection". *ACM Trans. Math. Softw.*, **38**(1), Dec. 2011. ISSN: 0098-3500. DOI: 10.1145/2049662.2049663. URL: https://doi.org/10.1145/2049662.2049663.

[18] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. "Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant". In: *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '15. Mumbai, India: Association for Computing Machinery, 2015, pp. 689–700. ISBN: 9781450333009. DOI: 10.1145/2676726.2677006. URL: https://doi.org/10.1145/2676726.2677006.

[19] Benjamin Delaware, Sorawit Suriyakarn, Clément Pit-Claudel, Qianchuan Ye, and Adam Chlipala. "Narcissus: Correct-By-Construction Derivation of Decoders and Encoders from Binary Formats". In: *Proc. ICFP*. Berlin, Germany, Aug. 2019. DOI: 10.1145/3341686. URL: http://adam.chlipala.net/papers/NarcissusICFP19/.

[20] Sébastien Donadio, James C. Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David A. Padua, and Keshav Pingali. "A Language for the Compact Representation of Multiple Program Versions". In: *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 136–151. DOI: 10.1007/978-3-540-69330-7_10. URL: https://doi.org/10.1007/978-3-540-69330-7_10.

[21] Andres Erbsen, Jade Philipoom, Jason Gross, Robert Sloan, and Adam Chlipala. "Simple High-Level Code for Cryptographic Arithmetic - With Proofs, Without Compromises". *2019 IEEE Symposium on Security and Privacy (SP)*, **54**(1), May 2019, pp. 1202–1219. DOI: 10.1109/sp.2019.00005.

[22] Kayvon Fatahalian et al. "Sequoia: Programming the Memory Hierarchy". In: *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*. SC '06. Tampa, Florida: Association for Computing Machinery, 2006, 83–es. ISBN: 0769527000. DOI: 10.1145/1188455.1188543. URL: https://doi.org/10.1145/1188455.1188543.

[23] Rongxiao Fu, Xueying Qin, Ornela Dardha, and Michel Steuwer. *Row-Polymorphic Types for Strategic Rewriting*. 2021. arXiv: 2103.13390 [cs.PL].

[24] Mahdi Ghorbani, Mathieu Huot, Shideh Hashemian, and Amir Shaikhha. "Compiling Structured Tensor Algebra". *Proc. ACM Program. Lang.*, **7**(OOPSLA2), Oct. 2023. DOI: 10.1145/3622804. URL: https://doi.org/10.1145/3622804.

[25] Vladimir Gladshtein, Qiyuan Zhao, Willow Ahrens, Saman Amarasinghe, and Ilya Sergey. "Mechanised Hypersafety Proofs about Structured Data". *Proc. ACM Program. Lang.*, **8**(PLDI), June 2024. DOI: 10.1145/3656403. URL: https://doi.org/10.1145/3656403.

[26] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. "Concrete Mathematics". In: Addison Wesley, 2011, pp. 36–37.

[27] Dahai Guo, William Gropp, and Luke N Olson. "A hybrid format for better performance of sparse matrix-vector multiplication on a GPU". *The International Journal of High Performance Computing Applications*, **30**(1), 2016, pp. 103–120. DOI: 10.1177/1094342015593156.

[28] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. *Fireiron: A Scheduling Language for High-Performance Linear Algebra on GPUs*. 2020. arXiv: 2003.06324 [cs.PL].

[29] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. "Annotation-based empirical performance tuning using Orio". In: *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009*. Rome, Italy: IEEE, 2009, pp. 1–11. DOI: 10.1109/IPDPS.2009.5161004. URL: https://doi.org/10.1109/IPDPS.2009.5161004.

[30] Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. "Futhark: Purely Functional GPU-programming with Nested Parallelism and In-place Array Updates". In: *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2017. Barcelona, Spain: ACM, 2017, pp. 556–571. ISBN: 978-1-4503-4988-8. DOI: 10.1145/3062341.3062354. URL: http://doi.acm.org/10.1145/3062341.3062354.

[31] Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. "A verified optimizer for quantum circuits". *Proceedings of the ACM on Programming Languages*, **5**(POPL), Jan. 2021, pp. 1–29. ISSN: 2475-1421. DOI: 10.1145/3434318. URL: http://dx.doi.org/10.1145/3434318.

[32] 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". *ACM Trans. Graph.*, **38**(6), 2019, 201:1–201:16. DOI: 10.1145/3355089.3356506. URL: https://doi.org/10.1145/3355089.3356506.

[33] Intel. *Intel Math Kernel Library*. 2025. URL: https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html.

[34] Kenneth E. Iverson. *A programming language*. New York, NY, USA: John Wiley & Sons, Inc., 1962. ISBN: 978-0-471-43014-8. DOI: 10.5555/1098666.

[35] Ankit Jain. "pOSKI: An extensible autotuning framework to perform optimized SpMVs on multicore architectures". MA thesis. University of California, Berkeley, 2009. URL: https://bebop.cs.berkeley.edu/pubs/jain2008-masters.pdf.

[36] Andrew Kerr, Duane Merrill, Julien Demouth, and John Tran. *CUTLASS: Fast Linear Algebra in CUDA C++*. https://developer.nvidia.com/blog/cutlass-linear-algebra-cuda. 2017.

[37] David R. Kincaid, Thomas C. Oppe, and David M. Young. "ITPACKV 2D user's guide". In: 1989. URL: https://api.semanticscholar.org/CorpusID:56590066.

[38] Fredrik Kjolstad, Willow Ahrens, Shoaib Kamil, and Saman Amarasinghe. "Tensor algebra compilation with workspaces". In: *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE. 2019, pp. 180–192. DOI: 10.1109/CGO.2019.8661185.

[39] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. "The tensor algebra compiler". *Proceedings of the ACM on Programming Languages*, **1**(OOPSLA), Oct. 2017, pp. 1–29. DOI: 10.1145/3133901. URL: https://doi.org/10.1145/3133901.

[40] Vladimir Kotlyar, Keshav Pingali, and Paul Stodghill. "A Relational Approach to the Compilation of Sparse Matrix Programs". In: *Euro-Par '97: Proceedings of the Third International Euro-Par Conference on Parallel Processing*. London, UK: Springer-Verlag, 1997, pp. 318–327. ISBN: 3-540-63440-1. URL: http://iss.ices.utexas.edu/Publications/Papers/EUROPAR1997.pdf.

[41] Scott Kovach, Praneeth Kolichala, Tiancheng Gu, and Fredrik Kjolstad. "Indexed Streams: A Formal Intermediate Representation for Fused Contraction Programs". *Proc. ACM Program. Lang.*, **7**(PLDI), June 2023. DOI: 10.1145/3591268. URL: https://doi.org/10.1145/3591268.

[42] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. "CakeML: A Verified Implementation of ML". In: *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '14. 10.1145/2535838.2535841. San Diego, California, USA: Association for Computing Machinery, 2014, pp. 179–191. ISBN: 9781450325448. URL: https://ts.data61.csiro.au/publications/nicta_full_text/7494.pdf.

[43] Peter Lammich. "Automatic Data Refinement". In: *Interactive Theorem Proving - 4th International Conference, ITP 2013, Rennes, France, July 22-26, 2013. Proceedings*. Ed. by Sandrine Blazy, Christine Paulin-Mohring, and David Pichardie. Vol. 7998. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2013, pp. 84–99. DOI: 10.1007/978-3-642-39634-2_9. URL: https://doi.org/10.1007/978-3-642-39634-2%5C_9.

[44] Peter Lammich. "Refinement to Imperative HOL". *J. Autom. Reason.*, **62**(4), Apr. 2019, pp. 481–503. ISSN: 0168-7433. DOI: 10.1007/s10817-017-9437-1. URL: https://doi.org/10.1007/s10817-017-9437-1.

[45] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation". In: *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2021, pp. 2–14. DOI: 10.1109/CGO51591.2021.9370308.

[46] Xavier Leroy. "A Formally Verified Compiler Back-End". en. *Journal of Automated Reasoning*, **43**(4), Dec. 2009, pp. 363–446. ISSN: 0168-7433, 1573-0670. DOI: 10.1007/s10817-009-9155-4.

[47] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. "Differentiable programming for image processing and deep learning in Halide". *ACM Trans. Graph. (Proc. SIGGRAPH)*, **37**(4), 2018, 139:1–139:13. DOI: 10.1145/3197517.3201383.

[48] Zhitao Lin and Christophe Dubach. "From Functional to Imperative: Combining Destination-Passing Style and Views". In: *Proceedings of the 8th ACM SIGPLAN International Workshop on Libraries, Languages and Compilers for Array Programming*. ARRAY 2022. San Diego, CA, USA: Association for Computing Machinery, 2022, pp. 25–36. ISBN: 9781450392693. DOI: 10.1145/3520306.3534502. URL: https://doi.org/10.1145/3520306.3534502.

[49] Amanda Liu, Gilbert Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. "A Verified Compiler for a Functional Tensor Language". *Proc. ACM Program. Lang.*, **8**(PLDI), June 2024. DOI: 10.1145/3656390. URL: https://doi.org/10.1145/3656390.

[50] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. "Verified tensor-program optimization via high-level scheduling rewrites". *Proc. ACM Program. Lang.*, **6**(POPL), Jan. 2022. DOI: 10.1145/3498717. URL: https://doi.org/10.1145/3498717.

[51] Amanda Liu, Shoaib Kamil, Adam Chlipala, and Jonathan Ragan-Kelley. "A Mechanizd Algebra of Verified Data Structures for Optimizing Sparse Tensor Programs". *Proc. ACM Program. Lang.*, **10**(PLDI), June 2026. DOI: 10.1145/3808261. URL: https://doi.org/10.1145/3808261.

[52] Jie Liu, Zhongyuan Zhao, Zijian Ding, Benjamin Brock, Hongbo Rong, and Zhiru Zhang. "UniSparse: An Intermediate Language for General Sparse Format Customization". *Proc. ACM Program. Lang.*, **8**(OOPSLA1), Apr. 2024. DOI: 10.1145/3649816. URL: https://doi.org/10.1145/3649816.

[53] Magnus O. Myreen and Scott Owens. "Proof-producing synthesis of ML from higher-order logic". In: *International Conference on Functional Programming (ICFP)*. Ed. by Peter Thiemann and Robby Bruce Findler. New York, NY, USA: ACM, 2012, pp. 115–126.

[54] NVIDIA. *cuSPARSE*. 2021. URL: https://docs.nvidia.com/cuda.

[55] Zoe Paraskevopoulou, John M. Li, and Andrew W. Appel. "Compositional Optimizations for CertiCoq". *Proc. ACM Program. Lang.*, **5**(ICFP), Aug. 2021. DOI: 10.1145/3473591. URL: https://doi.org/10.1145/3473591.

[56] Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. "Getting to the Point. Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming". In: *The 25th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM SIGPLAN. ACM, Aug. 2021. DOI: 10.1145/3473593.

[57] Clément Pit-Claudel, Jade Philipoom, Dustin Jamner, Andres Erbsen, and Adam Chlipala. "Relational Compilation for Performance-Critical Applications: Extensible Proof-Producing Translation of Functional Models into Low-Level Code". 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. 918–933. ISBN: 9781450392655. DOI: 10.1145/3519939.3523706. URL: https://doi.org/10.1145/3519939.3523706.

[58] Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. "Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs". In: *IJCAR'20: Proceedings of the 9th International Joint Conference on Automated Reasoning*. Paris, France: Springer, June 2020, pp. 119–137. DOI: 10.1007/978-3-030-51054-1_7.

[59] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. "Decoupling algorithms from schedules for easy optimization of image processing pipelines". *ACM Trans. Graph.*, **31**(4), 2012, 32:1–32:12. DOI: 10.1145/2185520.2185528. URL: https://doi.org/10.1145/2185520.2185528.

[60] 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. ISBN: 9781450320146. DOI: 10.1145/2491956.2462176. URL: https://doi.org/10.1145/2491956.2462176.

[61] Alexander J Root, Bobby Yan, Peiming Liu, Christophe Gyurgyik, Aart J.C. Bik, and Fredrik Kjolstad. "Compilation of Shape Operators on Sparse Arrays". *Proc. ACM Program. Lang.*, **8**(OOPSLA2), Oct. 2024. DOI: 10.1145/3689752. URL: https://doi.org/10.1145/3689752.

[62] Maximilian Schleich, Amir Shaikhha, and Dan Suciu. "Optimizing Tensor Programs on Flexible Storage". *Proc. ACM Manag. Data*, **1**(1), May 2023. DOI: 10.1145/3588717. URL: https://doi.org/10.1145/3588717.

[63] SciPy. *SciPy*. 2025. URL: https://www.scipy.org/.

[64] 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". *Proceedings of the ACM on Programming Languages*, **4**(OOPSLA), 2020, pp. 1–30. DOI: 10.1145/3428226.

[65] Amir Shaikhha, Andrew Fitzgibbon, Simon Peyton Jones, and Dimitrios Vytiniotis. "Destination-Passing Style for Efficient Memory Management". In: *Proceedings of the 6th ACM SIGPLAN International Workshop on Functional High-Performance Computing*. FHPC 2017. Oxford, UK: Association for Computing Machinery, 2017, pp. 12–23. ISBN: 9781450351812. DOI: 10.1145/3122948.3122949. URL: https://doi.org/10.1145/3122948.3122949.

[66] Amir Shaikhha, Mathieu Huot, and Shideh Hashemian. "A Tensor Algebra Compiler for Sparse Differentiation". *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024, pp. 1–12. URL: https://api.semanticscholar.org/CorpusID:268045813.

[67] Justin Slepak, Olin Shivers, and Panagiotis Manolios. "An Array-Oriented Language with Static Rank Polymorphism". In: *Programming Languages and Systems*. Ed. by Zhong Shao. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 27–46. ISBN: 978-3-642-54833-8. DOI: 10.1007/978-3-642-54833-8_3.

[68] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. "Pure Tensor Program Rewriting via Access Patterns (Representation Pearl)". In: *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*. MAPS 2021. Virtual, Canada: Association for Computing Machinery, 2021, pp. 21–31. ISBN: 9781450384674. DOI: 10.1145/3460945.3464953. URL: https://doi.org/10.1145/3460945.3464953.

[69] 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/.

[70] Michel Steuwer, Chris Fensch, Sam Lindley, and Christophe Dubach. "Generating Performance Portable Code using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code". In: *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*. Vol. 50. Association for Computing Machinery, Sept. 2015. DOI: 10.1145/2784731.2784754.

[71] W.F. Tinney and J.W. Walker. "Direct solutions of sparse network equations by optimally ordered triangular factorization". *Proceedings of the IEEE*, **55**(11), 1967, pp. 1801–1809. DOI: 10.1109/PROC.1967.6011.

[72] *Turing Architecture Whitepaper*. https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing-architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf. 2018.

[73] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. *Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions*. 2018. arXiv: 1802.04730 [cs.PL].

[74] Anand Venkat, Tharindu Rusira, Raj Barik, Mary Hall, and Leonard Truong. "SWIRL: High-performance many-core CPU code generation for deep neural networks". *The International Journal of High Performance Computing Applications*, **33**(6), 2019, pp. 1275–1289. DOI: 10.1177/1094342019866247. eprint: https://doi.org/10.1177/1094342019866247. URL: https://doi.org/10.1177/1094342019866247.

[75] *Volta Architecture Whitepaper*. https://images.nvidia.com/content/volta-architecture/pdf/volta-architecture-whitepaper.pdf. 2017.

[76] Richard Vuduc, James W Demmel, and Katherine A Yelick. "OSKI: A library of automatically tuned sparse matrix kernels". *Journal of Physics: Conference Series*, **16**(1), Jan. 2005, p. 521. DOI: 10.1088/1742-6596/16/1/071. URL: https://dx.doi.org/10.1088/1742-6596/16/1/071.

[77] Qing Yi, Keith Seymour, Haihang You, Richard W. Vuduc, and Daniel J. Quinlan. "POET: Parameterized Optimizations for Empirical Tuning". In: *21st International Parallel and Distributed Processing Symposium (IPDPS 2007)*. Rome, Italy: IEEE, 2007, pp. 1–8. DOI: 10.1109/IPDPS.2007.370637. URL: https://doi.org/10.1109/IPDPS.2007.370637.

[78] Liang Yuan, Yunquan Zhang, Xiangzheng Sun, and Ting Wang. "Optimizing Sparse Matrix Vector Multiplication Using Diagonal Storage Matrix Format". In: *2010 IEEE 12th International Conference on High Performance Computing and Communications (HPCC)*. 2010, pp. 585–590. DOI: 10.1109/HPCC.2010.67.

[79] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. "GraphIt: a high-performance graph DSL". *PACMPL*, **2**(OOPSLA), 2018, 121:1–121:30. DOI: 10.1145/3276491. URL: https://doi.org/10.1145/3276491.