

Verified Scheduling Via High-Level Scheduling Rewrites

by
Amanda Liu

B.S., Columbia University (2020)

Submitted to the Department of Electrical Engineering and Computer
Science

in partial fulfillment of the requirements for the degree of
Master of Science

at the
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
May 2022

© Massachusetts Institute of Technology 2022. All rights reserved.

Author
Department of Electrical Engineering and Computer Science
May 13, 2022

Certified by.
Adam Chlipala
Associate Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Certified by.
Jonathan Ragan-Kelley
Assistant Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

Verified Scheduling Via High-Level Scheduling Rewrites

by
Amanda Liu

Submitted to the Department of Electrical Engineering and Computer Science
on May 13, 2022 in Partial Fulfillment of the
Requirements for the Degree of Master of Science in
Electrical Engineering and Computer

ABSTRACT

I propose a lightweight Coq framework for optimizing tensor kernels written in a pure, functional array language. Optimizations rely on user scheduling using series of verified, semantics-preserving rewrites. Unusually for compilation targeting imperative code with arrays and nested loops, all rewrites are source-to-source within a purely functional language. This language comprises a set of core constructs for expressing high-level computation detail and a set of what we call reshape operators, which can be derived from core constructs but trigger low-level decisions about storage patterns and ordering. We will demonstrate that not only is this system capable of deriving the optimizations of existing state-of-the-art languages like Halide and generating comparably performant code, it is also able to schedule a family of useful program transformations beyond what is reachable in Halide.

Thesis Supervisor: Adam Chlipala

Title: Associate Professor of Electrical Engineering and Computer Science

Thesis Supervisor: Jonathan Ragan-Kelley

Title: Assistant Professor of Electrical Engineering and Computer Science

Much of the content of this thesis has also been published as a paper in the POPL 2022 issue of the Proceedings of the ACM on Programming Languages [31] as well as written for a proposal submitted to the National Science Foundation's Core Programs co-authored with Gilbert Louis Bernstein, Prof. Adam Chlipala, and Prof. Jonathan Ragan-Kelley.

Contents

1	Introduction	6
1.1	A Coq Framework	6
1.2	Contributions	7
1.3	Overview and Motivating Example	7
2	Related Work	11
2.1	User-Schedulable Languages	11
2.2	Rewrite-Based Optimization Systems	11
2.3	Verified Program Derivation	11
2.4	Functional Languages	12
3	A Tensor Language	13
3.1	A Simple Pipeline Example	13
3.2	Specification	14
3.2.1	Types	14
3.2.2	Syntax and Semantics	14
4	Verified Rewrite Framework	16
4.1	Scheduling Rewrites	16
4.2	Binders and Contexts	17
4.3	Rewrite Tactics and Automation	18
5	Compilation	20
5.1	Normalization	20
5.2	Code Generation	22
6	Reshape Operators	25
6.1	Compute and Storage Order	25
6.2	Safe Garbage	28
6.3	Adjoint Introduction	29
7	Evaluation	31
7.1	Blur	31
7.2	Scatter-to-Gather Optimization	33
7.3	Im2col	35
8	Implementation Details	36
8.1	Consistency and Shape	36
8.2	Access Safety	37
9	Future Work	38
9.1	Streamlining the Optimization Process with Automation	38
9.2	Increasing the Source Language’s Expressivity	39
9.3	Verifying Current Lowering to C	40
9.4	Proof-Generating Compilation to a New Lower-Level IR	40

9.5	Verified Compilation From the New IR to Machine Code	41
9.6	Verification of Hardware Accelerators	41

Chapter 1

Introduction

In high-performance computing, a single natural algorithm over multidimensional arrays may have a bewildering variety of different code realizations, to optimize for performance on different machines. The alternatives generally trade off between optimizing for parallelism and data locality. There are clear software-engineering benefits from explaining optimized code in terms of transformations applied to unoptimized algorithms. For instance, the transformations may be checked by compilers, so that functionality bugs can only be missed in the algorithm, not specific optimizations on it. Programming languages like Halide for graphics [35] and TVM for machine learning [10] have emerged to facilitate directly programming in this style, with compilers driven by optimization directives. However, despite the intuitive appeal of this approach, most programming in the domain has effectively mixed algorithms and optimizations in single source files, which have no formal connections to alternative implementations of matching algorithms.

1.1 A Coq Framework

We propose a framework embedded in the Coq proof assistant, with a language of optimization commands that is simultaneously more formally assured and more flexible than in past work. One popular approach is phrasing optimizations as transformations within a single source or intermediate language; the most common kind of transformation is a *rewrite rule*, a quantified term equality. There have been several success stories, including in Elevate [17], Glenside [37], and others [38, 28]. A typical such project presents rewrite rules as axioms and then studies engines to apply them effectively. We instead give a formal semantics to a core language and formally verify rewrite rules as theorems. Steuwer et. al. [38] proved rewrite rules with respect to a fixed target language (OpenCL), but otherwise we are aware of no past work in this space that demonstrated a practical optimizer with mechanized proofs of rewrite rules, and our proofs are also noteworthy for applying to very general source-to-source rewrite rules.

Working in Coq has appealing advantages, similar to those that arise for embedded domain-specific languages in general. Where past projects have created custom orchestration frameworks for their languages of optimization commands, we inherit most functionality from Coq’s standard tactic engine. As the existing tactic language is Turing-complete, we have a powerful framework for coding derivation building blocks at many levels of abstraction and automation. With our tooling, any programmer

may add a new rewrite rule or automation procedure, with no danger of optimizer unsoundness, because all rules must be proved from first principles. Further, working within a proof assistant creates opportunities for connecting to other formal developments. We can imagine composing an algorithm soundness proof with one of our derivations of optimized code with correctness of a lower-level-language compiler or even a hardware accelerator – all of which are worthwhile future work.

In this proposal, our primary point of comparison is Halide [35], which uses a language of scheduling primitives to optimize graphics code or tensor computations in general. The rewrites provided in our framework allow us to emulate the majority of Halide’s scheduling directives. For instance, our rewrites are expressive enough to capture the functionality of Halide scheduling directives such as `compute_at`, `split`, `tile`, `reorder`, `fuse`, `unroll`, `compute_inline`, etc. While some of Halide’s scheduling features are out of scope,¹ we aim to demonstrate that we can nevertheless recover most of the appealing properties of Halide, including comparable performance on representative examples. In contrast to Halide, our scheduling process is entirely proved within Coq while still supporting a fairly pleasant programmer experience (modulo usual gripes about interactive proving in Coq). Furthermore, the chance to extend the system easily with new rewrite rules and procedures allows us to handle some important examples that Halide itself cannot, including an “im2col” procedure of the kind prevalent in machine learning.

1.2 Contributions

Summing up, we plan several contributions:

- We present the first HPC-guided-optimization system that produces machine-checked proofs from first principles, justifying transformations of specific programs.
- We also show for the first time how to accommodate this breadth of potential transformations solely as source-to-source rewrites within a purely functional language, thanks to the concept of *reshape operators*.
- We explain how to use standard Coq features to build a library of definitions and tactics supporting programming and transformation at a natural granularity for HPC programmers.
- We evaluate our prototype and show it produces C code competitive in performance with alternative tools that offer weaker formal guarantees.

The next section outlines the approach with a motivating example. Then we return to define our language (including formal semantics) bottom-up, before proceeding through three crucial elements of our pipeline: basic scheduling rewrites, lowering to imperative code, and reshape operators. After an interlude explaining Coq encoding details, we present preliminary results from an empirical evaluation showing that we achieve competitive performance w.r.t. Halide on a small set of examples, also managing to compile respectably fast versions of some algorithms beyond Halide’s applicability. We wrap up with further related-work discussion.

Our framework implementation and examples are available [open-source](#).

1.3 Overview and Motivating Example

The tensor-computation programs found in image-processing and machine-learning kernels often take the form of complex mathematical pipelines. The program expressed below in Halide [35] contains the definition of a simple pipeline comprising two stages where the first one reads from an input tensor f and the second stage, defining the output, computes each cell as the sum of two adjacent values computed in the first stage. It is followed by the scheduling directives that optimize this program into a tiled, two-stage schedule, which is generally accomplished by splitting a single loop into a nested outer and inner loop pair. This common scheduling strategy is useful for optimization in tensor-computation pipelines since it maintains a degree of producer/consumer locality proportional to the size of the inner

¹We do not yet address transformations requiring stateful looping structures, namely sliding-window optimization, which would be valuable to incorporate as future work. We also do not currently support user-level control of optimizations like vectorization and thread-level parallelism, instead delegating these decisions to the downstream compiler, but our framework would naturally support explicit parallelism as well.

generation, while also reducing the amount of redundant computation in pipelines with windows of computation that share values between iterations.

```

ImageParam image(type_of<float>(), 1);
Func f, buf, output;
Var x, x_outer, x_inner;
f = BoundaryConditions::constant_exterior(image,0);
buf(x) = f(x);
output(x) = buf(x-1) + buf(x);

output.split(x, x_outer, x_inner, 8, TailStrategy::GuardWithIf);
buf.compute_at(output, x_outer);

```

Halide is a good comparison point for our efforts, since it is widely adopted in industry to produce performance-competitive code. For instance, it is used to produce crucial routines within Adobe Photoshop, and the YouTube video-ingestion pipeline runs Halide-compiled code. Note the crucial split (indicated by a newline above) between the algorithm specification and scheduling (optimization) directives. We will not explain those directives in detail, since our own work accomplishes the same transformations by different means. However, it is worth pointing out that the intermediate results of Halide scheduling directives can often no longer be represented in the source language, a downside that we remedy in our framework.

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.

$$\text{let } buf := \bigoplus_{i=0}^n f[i] \text{ in } \bigoplus_{i=0}^n [0 \leq i - 1] \cdot buf[i - 1] \oplus buf[i]$$

The big-operator \bigoplus 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 is a fairly simple program, 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:

```

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] = ..
  }
}

```

We implement ATL in Coq’s 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 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 Coq 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.

$$\text{trunc}_r n \begin{matrix} \lceil n/8 \rceil \\ \boxed{} \\ i_o=0 \end{matrix} \left(\text{let } v := \begin{matrix} 8 \\ \boxed{} \\ i_i=0 \end{matrix} [i_o * 8 + i_i < n] \cdot (([0 \leq i_o * 8 + i_i - 1] \cdot f[i_o * 8 + i_i - 1]) \oplus f[i_o * 8 + i_i]) \text{ in } \begin{matrix} 8 \\ \boxed{} \\ i_i=0 \end{matrix} [i_o * 8 + i_i < n] \cdot v[i_i] \right)$$

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; ii++) {
      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 `trunc`, 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 `⊞`, 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.

Chapter 2

Related Work

2.1 User-Schedulable Languages

Our work will contribute to the emerging area of user-schedulable languages. Languages and compilers that offer explicit programmer control over program transformations have a longer history in HPC [14, 16, 42, 21, 9], but most such systems provide few safety or correctness guarantees. Halide popularized the idea of a scheduling language with which a programmer could derive many different optimized implementations of a single reference program without changing its semantics [34, 35]. This approach has since been adapted to dense linear algebra and machine learning [10, 39, 40, 20], sparse tensor algebra [27], distributed-memory computing [2], graph processing [43], and physical simulation [24]. None of these systems provide formal assurance of program equivalence before and after scheduling, and many allow unsound transformations. They also offer *fixed* languages of scheduling transformations, which can only be extended at great complexity by modifying the language and compiler. In contrast, our system provides similar user-scheduling functionality but in a way that simultaneously supports extension with new user-defined transformations and formally guarantees equivalence of programs before and after transformation.

2.2 Rewrite-Based Optimization Systems

Key to our approach is defining scheduling transformations in terms of algebraic rewrite rules. Recent work develops rewrite-rule systems for optimizing array programs [38, 17], but they generally treat rewrites as axioms and provide no formal guarantees. Previous work such as the VOQC quantum-circuit optimizer also shows how a tactic engine and interactive theorem prover provide a natural framework for building a verified program-optimization framework [23]. However, they use Coq to validate prewritten optimization procedures, while we focus on step-by-step manual derivation of optimizations for specific programs. Additionally, their application domain is different enough from ours that it is not surprising they do not address the loop-and-computation-reordering challenge that our reshape operators solve.

2.3 Verified Program Derivation

Program derivation through proof in constructive logic has a long history. One recent framework (also based on Coq) is Fiat, within whose unified setting automated proof-producing procedures have been demonstrated for relational queries [12] and binary-format parsers [13]. Our work is complementary and could be integrated as another derivation-automation domain within Fiat, though we manage to state our specifications and derive programs deterministically, so the value of Fiat’s nondeterminism monad would be minimal. Fiat goes further than we do, in making the translation from optimized functional to low-level code proof-generating [33], and we would benefit from adopting similar techniques in future work.

2.4 Functional Languages

Finally, as a functional tensor language, our program representation (ATL) builds on ideas from array languages [26, 8, 7, 36] which have more recently been explored in a functional context [22, 6, 32]. It is specifically derived from recent work on a functional tensor language for automatic differentiation [3], which we have extended with features like reshape operators to express a richer space of implementation details relevant to performance optimization. Concurrent work on Glenside [37] attempts to capture some similar implementation details by augmenting a functional tensor language with an algebra of “access patterns.”

Chapter 3

A Tensor Language

The source language that our framework uses is a pure, functional language named ATL (A Tensor Language) that is designed to express tensor-computation-pipeline schedules as high-level, algebraic expressions [3]. In this section we will introduce and formalize the core constructs of ATL as well as their types and denotational semantics.

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.

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

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 \bigoplus . 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 $0 \leq i - 1$, 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 [19]. 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.

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 $\mathbb{Z} \rightarrow X$, 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 Specification

3.2.1 Types

In general, the computational pipelines described in ATL will be polymorphic like the simple pipeline example. In other words, a schedule expressed in ATL is agnostic of the absolute dimensionality of the data it computes over, unless it uses some type-specific operator like scalar multiplication to make dimensionality concrete. To capture the options, we define ATL types with a simple grammar.

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

To put it simply, this means that any ATL expression's type is either an element of \mathbb{R} , a scalar; or a tensor of elements of some type. As a result, the operators of ATL are inherently polymorphic. All instances of this polymorphism for binary addition (\oplus), sum reduction (Σ), 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, since the absolute dimensionality and sizes of inputs must be given, which instantiates the dimensions in the rest of the pipeline.

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

$$\prod_{i=0}^n \prod_{j=0}^i 1$$

This simple program computes a tensor that comprises tensors of various lengths—a complexity that we wish to disallow. We will use a convention function `shape` that takes an ATL expression with consistent internal dimension sizes and returns as a list the size of each dimension. It returns the empty list for scalar expressions. For example, reconsider the simple pipeline presented previously. If f were a function producing scalars, then the final output would be a tensor of n scalars. The same statement is expressed below using the *shape* function convention:

$$\text{shape} \left(\text{let } buf := \prod_{j=0}^n f(j) \text{ in } \prod_{i=0}^n [0 \leq i - 1] \cdot buf[i - 1] \oplus buf[i] \right) = [n]$$

We can use this same construct to discuss the simple pipeline in more general terms without having to examine the concrete type of f . Suppose it is known that $\forall x. \text{shape}(f(x)) = s$. It follows that `shape` applied to the simple pipeline yields the following:

$$\text{shape} \left(\text{let } buf := \prod_{j=0}^n f(j) \text{ in } \prod_{i=0}^n [0 \leq i - 1] \cdot buf[i - 1] \oplus buf[i] \right) = n :: s$$

3.2.2 Syntax and Semantics

The complete syntax of our embedding appears in Figure 3.1, with a denotational semantics excerpted in Figure 3.2. Note that we chose to use a shallow embedding, which means that each syntactic construct is just a library function written in Coq's dependently typed, pure functional language Gallina. We will not dwell here on more intuition for the different constructs, as we are about to see many examples via algebraic properties to be proved. One remark, though, is in order for the final clause of Figure 3.2, which explains array indexing. We treat 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. It is important that default values are properly typed, so we compute them based on those arrays' first elements.

Variable	$x \in \mathbb{S}$
Index Expression	$I ::= x \mid i \mid I + I \mid I - I \mid I \times I \mid [I / I] \mid I / I \mid I \% I$
Predicate	$p ::= \mathbf{true} \mid \mathbf{false} \mid I = I \mid I < I \mid I \leq I \mid p \wedge p$
Expression	$e ::= x \mid [p] \cdot e \mid \mathbf{let} \ x := e \ \mathbf{in} \ e \mid \bigsqcup_{x=I}^I e \mid \sum_{x=I}^I e \mid e \oplus e \mid e[I] \mid e * e \mid e / e$

Figure 3.1: Core ATL syntax

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

Figure 3.2: Denotational semantics for the core of the ATL language (selected rules)

Chapter 4

Verified 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 a series of algebraic rewrites verified within Coq. We detail the proof mechanisms used to prove the set of minimal equivalences that are used as composable scheduling rewrites to transform and optimize programs. 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. These mechanisms are what allows us to automate the reasoning for applying these rewrites on ATL programs at a high level.

4.1 Scheduling Rewrites

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.

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

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 $\prod_{i=0}^n f(i)$ 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:

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

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:

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

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:

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

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 \bigsqcup 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.

4.2 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. Coq users are accustomed to technicalities of rewriting under binders, appealing to axioms like functional extensionality. 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:

$$\frac{\forall x. 0 \leq x < n \rightarrow e_1[x/i] = e_2[x/i]}{\bigsqcup_{i=0}^n e_1 = \bigsqcup_{i=0}^n e_2}$$

This lemma 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:

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

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 $0 \leq i < n$:

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

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:

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

The access at $i - 1$ is not provably valid in this context just yet. Although the knowledge that $0 \leq i < n$ provides that $i - 1 < n$, there is no guarantee that it is nonnegative. However, 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:

$$\frac{\text{shape}(e_1) = \text{shape}(e_2) \quad p \rightarrow e_1 = e_2}{[p] \cdot e_1 = [p] \cdot e_2}$$

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

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

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

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

Now in addition to the information provided from the generation that $0 \leq i < n$, the context includes the constraint that $0 \leq i - 1$. 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:

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

We equip other binding constructs with similar context lemmas, and our `rw` tactic applies the lemmas automatically to rewrite under binders.

4.3 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.

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 end 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 do not involve tiling result in indexing expressions and loop bounds that land in the world of affine arithmetic, which is decidable. However, due to the use of tiling and flattening, which introduce dimensions with terms that respectively include ceiling division and multiplication, these arithmetic expressions are not exclusively affine. As a result, comparison between these expressions is undecidable in the general case. However, expressions arise in a regular form expressing Euclidean factorization of known terms. As a result, we are still able to prove the conditions automatically for all interesting examples.

Chapter 5

Compilation

In order to demonstrate the ability of our framework to express useful, performant schedules, we implemented a trusted lowering from ATL into C to be able to produce runnable code (which will turn out to have competitive performance). In this section we detail the trusted lowering process of ATL into C used for benchmarking purposes. This lowering process includes a normalization process that is driven by the same proof-rewriting framework as the optimization rewrites and includes a set of verified normalization rewrites. Normalized ATL programs are then compiled following a continuation-passing style to generate C code.

5.1 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, τ 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.

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

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 $\mathbb{Z} \rightarrow \mathbb{R}$, then this stage of normalization will result

in the following program:

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

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:

$$\prod_{i=0}^n \text{tensor_add} ([0 \leq i - 1] \cdot f(i - 1)) (f(i))$$

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 computer 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:

$$\prod_{i=0}^n \text{tensor_add} ([0 \leq i - 1] \cdot f(i - 1)) (f(i))$$

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:

$$\frac{\text{shape}(v_1) = \text{shape}(v_2) = n :: _}{\text{tensor_add } v_1 \ v_2 = \prod_{i=0}^n v_1[i] \oplus v_2[i]}$$

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:

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

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.

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

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.

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

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.

5.2 Code Generation

Once a program has been normalized, it is compiled into C code via a recursive, syntax-directed process in accordance with the rules shown in Figure 5.1. Since the lowering of predicates and integer-indexing expressions from ATL into C is almost a trivial syntactic mapping and is in-line with the denotational semantics given in Figure 3.2, we will continue using the same $\llbracket \cdot \rrbracket$ notation in this section to represent lowered integer and Boolean expressions. We also include the general usage of `alloc` to represent the allocation of properly sized memory initialized to 0.

In this compilation procedure, the assignment of the lowering for some expression e is symbolically constructed as $y [i] \ominus$. In this notation, y represents the name of the buffer where the result of this expression should be stored. The symbol \ominus represents either an assignment ($=$) or accumulation ($+=$) to be resolved. Finally, i is a lambda with arguments corresponding to indices and returns the final indexing expression for this assignment. The indices are resolved by application downstream in the recursive calls that process subexpressions of e .

Note that this construction of the left-hand indexing expression for assignment is delayed by using the lambda to accumulate indices. The reason for this slight overhead in complexity will become more apparent in the following section where we discuss reshape operators and their function as compilation directives.

To better illustrate the process, with a focus on the detail of index accumulation, we return to our running example of the fully fused pipeline. We begin by assuming that y is a buffer allocated with space for n scalars. Also, since the overall dimensionality of this expression is that of a 1-dimensional tensor, the lambda that we pass in for i takes in one index and returns it as a singleton.

$$\left\langle y [\lambda i.(i)] = , \bigoplus_{i=0}^n [0 \leq i - 1] \cdot f[i - 1] + f[i] \right\rangle$$

The first construct in the expression to be encountered and compiled is the tensor comprehension. This results in the generation of a for-loop whose body is to be generated by the body of the tensor generation. At this point, the lambda has been applied, and the index expression is instantiated. We know we will be storing into buffer y in the same order the for-loop is iterating.

```
for (int i = 0; i < n; i++) {
  < y [ (i) ] = , [ 0 ≤ i - 1 ] · f[i - 1] + f[i] >
}
```

The next construct to be lowered is a scalar addition. Since two expressions are needed to compute the sum, neither of which is the ultimate value to be stored into the buffer, temporary storage must be created to store them. Since the types of these expressions are known to be scalars, they are allocated as floats. Similarly, the starting lambda for the recursive lowering is simply unit.

```
for (int i = 0; i < n; i++) {
  float tmp1 = 0, tmp2 = 0;
  < tmp1 [ () ] = , [ 0 ≤ i - 1 ] · f[i - 1] >
  < tmp2 [ () ] = , f[i] >
}
```

$$\begin{aligned}
\text{let } \lambda_{\text{idx}}(0) &= () \\
\lambda_{\text{idx}}(1) &= \lambda_{i_0} \cdot (i_0) \\
\lambda_{\text{idx}}(2) &= \lambda_{i_0} \cdot \lambda_{i_1} \cdot (i_0, i_1) \\
&\dots
\end{aligned}$$

$$\begin{aligned}
\langle y[i] \ominus, c \rangle &= y[i] \ominus c; \\
\langle y[i] \ominus, x \rangle &= y[i] \ominus x; \\
\langle y[i] \ominus, x[I_1] \cdots [I_k] \rangle &= y[i] \ominus x[[I_1], \dots, [I_k]]; \\
\langle y[i] \ominus, e_1 + e_2 \rangle &= \begin{bmatrix} \text{alloc tmp1}; \\ \text{alloc tmp2}; \\ \langle \text{tmp1}[\lambda_{\text{idx}}(0)] =, e_1 \rangle; \\ \langle \text{tmp2}[\lambda_{\text{idx}}(0)] =, e_2 \rangle; \\ y[i] \ominus \text{tmp1} + \text{tmp2}; \end{bmatrix} \\
&\text{(similarly for } * \text{ and } / \text{)} \\
\langle y[i] \ominus, \text{let } x := e_1 \text{ in } e_2 \rangle &= \begin{bmatrix} \text{alloc } x(\text{shape}(x)); \\ \langle x[\lambda_{\text{idx}}(\text{ndim}(x))] =, e_1 \rangle; \\ \langle y[i] \ominus, e_2 \rangle; \end{bmatrix} \\
\left\langle y[i] \ominus, \bigoplus_{j=m}^n e \right\rangle &= \begin{bmatrix} \text{for } (\text{int } j = [m]; j < [n]; j++) \{ \\ \quad \langle y[i(j)] \ominus, e \rangle; \\ \} \end{bmatrix} \\
\left\langle y[i] \ominus, \sum_{j=m}^n e \right\rangle &= \begin{bmatrix} \text{for } (\text{int } j = [m]; j < [n]; j++) \{ \\ \quad \langle y[i] +=, e \rangle; \\ \} \end{bmatrix} \\
\langle y[i] \ominus, [p] \cdot e \rangle &= \begin{bmatrix} \text{if } ([p]) \{ \\ \quad \langle y[i] \ominus, e \rangle; \\ \} \end{bmatrix}
\end{aligned}$$

Figure 5.1: Compilation rules for lowering the core ATL constructs into C

```
y[i] = tmp1 + tmp2;
}
```

When we encounter a guard in the left-hand-side expression of the sum, a conditional is generated.

```
for (int i = 0; i < n; i++) {
  float tmp1 = 0, tmp2 = 0;
  if (0 ≤ i-1)
    ⟨ tmp1 [ () ] = , f[i - 1] ⟩
  ⟨ tmp2 [ () ] = , f[i] ⟩
  y[i] = tmp1 + tmp2;
}
```

Finally, we arrive at the translation of a tensor access. Due to our normalization process, it is known that this value should be a scalar. It can thus be translated directly and stored using the accumulated left-hand access expression.

```
for (int i = 0; i < n; i++) {
  float tmp1 = 0, tmp2 = 0;
  if (0 ≤ i-1)
    tmp1 = f[i-1];
  tmp2 = f[i];
  y[i] = tmp1 + tmp2;
}
```

And so, we arrive at the C program compiled from the ATL expression of the fully fused pipeline. However, even upon inspection of the schedules generated for this simple pipeline, potential for further optimization and rescheduling stands out. For example, the rescheduling that was performed on the original two-stage pipeline arrived at the same program in a totally fused state. Partial fusion involving splitting the original loop into two nested loops offers a middle ground in terms of locality and repeated computation. Additionally, we arrive at loops with conditional guards using conditions that only fail on the loops' very first iterations. It would be advantageous to separate such a loop into two, where the more computationally expensive one is free from the burden of checking the conditional. It is clear that although the core ATL language is capable of expressing a wide range of schedules distinguished by the storing and computation-staging of intermediary values, there still remains a class of optimizations that we cannot derive.

Chapter 6

Reshape Operators

In this section we introduce a set of operators to the core ATL language called *reshape operators*. These operators are defined and ascribed semantics that correspond to intuitive tensor operations such as concatenate and transpose. These operators act as special directives in lowering that allow us to decouple compute order from storage order in accordance with the semantics of the reshape operator itself. This allow us greater flexibility and expressivity in terms of the generated imperative code from the level of the functional ATL language.

By virtue of how the core of ATL is constructed and lowering is defined, the shape of a program is inherently tied to its computation order. All loop-producing language constructs shown in the lowering in Figure 5.1 generate loops and modify the left-hand assignment expression in the same, well-defined relative ordering between iteration order and storage order. Moreover, each buffer may only be written by one loop nest. As a result, for the core language constructs of ATL, there is a default interpretation of storage order with respect to compute order understood in our lowering process. However, a large class of useful program transformations and optimizations require transfiguring program shape by modifying loop structure and storage order. In order to achieve the degree of low-level control and expressivity required for efficiency, it becomes necessary to be able to express these constructs within the source language. However, within the algebraic style we have adopted for scheduling reasoning, much of the low-level optimizations and structural changes that the system needs to be able to induce appear semantics-preserving – one could say that they change performance but not functional behavior. Our framework takes advantage of this fact and provides a family of operations called *reshape operators* that are defined and proven in terms of existing constructs in the 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 6.1 and demonstrate the scheduling control they provide during code generation.

6.1 Compute and Storage Order

In this section, we deal with introducing control over the relative compute and storage order of computations, via operators that act as directives in lowering. Again, these operators are fully expressible (Figure 6.2) in terms of those we worked with in prior sections, but their payoff is in signaling clever ways of manipulating the order (Figure 6.3) in which array assignments happen within loop nests.

Concatenate	$e \circ e$
Transpose	e^T
Flatten	$\text{flatten } e$
Split	$\text{split } I e$
Pad on the Right	$\text{pad}_r I e$
Pad on the Left	$\text{pad}_l I e$
Truncate from the Right	$\text{trunc}_r I e$
Truncate from the Left	$\text{trunc}_l I e$

Figure 6.1: Reshape operators

$$\begin{aligned}
e_1 \circ e_2 &:= \bigoplus_{i=0}^{|e_1|+|e_2|} [i < |e_1|] \cdot e_1[i] \oplus [|e_1| \leq i] \cdot e_2[i - |e_1|] \\
e^T &:= \bigoplus_{x=0}^{|e[0]|} \bigoplus_{y=0}^{|e|} e[y; x] \\
\text{flatten } e &:= \bigoplus_{i=0}^{|e| \times |e[0]|} \sum_{j=0}^{|e|} \sum_{k=0}^{|e[0]|} [i = j \times |e[0]| + k] \cdot e[j; k] \\
\text{split } k e &:= \bigoplus_{i=0}^{\lceil |e|/k \rceil} \bigoplus_{j=0}^k [i \times k + j < |e|] \cdot e[i \times k + j] \\
\text{pad}_r k e &:= \bigoplus_{i=0}^{|e|+k} [i < |e|] \cdot e[i] & \text{trunc}_r k e &:= \bigoplus_{i=0}^k e[i] \\
\text{pad}_l k e &:= \bigoplus_{i=0}^{|e|+k} [k \leq i] \cdot e[i - k] & \text{trunc}_l k e &:= \bigoplus_{i=0}^k e[i + |e| - k]
\end{aligned}$$

Figure 6.2: Definitions of reshape operators

$$\begin{aligned}
\langle y[i] \ominus, e_1 \circ e_2 \rangle &= \left[\langle y[i] \ominus, e_1 \rangle; \right. \\
&\quad \left. \langle y[\lambda j. i(j + |e_1[0]|)] \ominus, e_2 \rangle \right]; \\
\langle y[i] \ominus, e^T \rangle &= \langle y[\lambda j. \lambda k. i(k)(j)] \ominus, e \rangle \\
\langle y[i] \ominus, \text{flatten } e \rangle &= \langle y[\lambda j. \lambda k. i(j \cdot |e[0]| + k)] \ominus, e \rangle \\
\langle y[i] \ominus, \text{split } k e \rangle &= \langle y[\lambda j. i(\lfloor j/k \rfloor)(j \% k)] \ominus, e \rangle \\
\langle y[i] \ominus, \text{pad}_r k e \rangle &:= \langle y[i] \ominus, e \rangle \\
\langle y[i] \ominus, \text{trunc}_r k e \rangle &:= \langle y[i] \ominus, e \rangle \\
\langle y[i] \ominus, \text{pad}_l k e \rangle &:= \langle y[\lambda j. i(j + k)] \ominus, e \rangle \\
\langle y[i] \ominus, \text{trunc}_l k e \rangle &:= \langle y[\lambda j. i(j - k)] \ominus, e \rangle
\end{aligned}$$

Figure 6.3: Compilation rules for reshape operators

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 naturally extends to higher dimensions by effectively gluing together the two expressions one after the other with regards to their outermost dimension. Its in-language definition (along with several others we will get to shortly) is shown in Figure 6.2. 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 6.3.

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

$$\frac{0 \leq k < n}{\begin{array}{c} n \\ \boxed{\boxed{}} \\ i=0 \end{array} e = \left(\begin{array}{c} k \\ \boxed{\boxed{}} \\ i=0 \end{array} e \right) \circ \left(\begin{array}{c} n \\ \boxed{\boxed{}} \\ i=k \end{array} e \right)}$$

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:

$$\left(\begin{array}{c} 1 \\ \boxed{\boxed{}} \\ i=0 \end{array} \begin{array}{c} a \\ \boxed{\boxed{}} \\ j=0 \end{array} [0 \leq i - 1] \cdot f(i - 1)[j] + f(i)[j] \right) \circ \left(\begin{array}{c} n \\ \boxed{\boxed{}} \\ i=1 \end{array} \begin{array}{c} a \\ \boxed{\boxed{}} \\ j=0 \end{array} [0 \leq i - 1] \cdot f(i - 1)[j] + f(i)[j] \right)$$

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:

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

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

$$\left(\begin{array}{c} 1 \\ \boxed{\boxed{}} \\ i=0 \end{array} \begin{array}{c} a \\ \boxed{\boxed{}} \\ j=0 \end{array} ([0 \leq i - 1] \cdot f(i - 1)[j] + f(i)[j]) \right) \circ \left(\begin{array}{c} n \\ \boxed{\boxed{}} \\ i=1 \end{array} \begin{array}{c} a \\ \boxed{\boxed{}} \\ j=0 \end{array} f(i - 1)[j] + f(i)[j] \right)$$

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 (Figure 6.2). In code generation, this operator is implemented by switching the indices associated with the dimensions being transposed inside the assignment-indexing expression (Figure 6.3).

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 6.2, 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 6.3). 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.

For instance, the following pattern computes a matrix of size 256×1024 in tiles of size 4×8 . Such a pattern is important for improving data locality in image-processing pipelines and matrix

multiplication (where tiling is more commonly known as blocking) as well as for exposing fixed-length inner loops for unrolling and vectorization.

$$\text{flatten} \begin{matrix} 64 \\ \boxed{\boxed{}} \\ i_{hi}=0 \end{matrix} \left(\text{flatten} \begin{matrix} 128 \\ \boxed{\boxed{}} \\ j_{hi}=0 \end{matrix} \left(\begin{matrix} 4 & 8 & \dots \\ \boxed{\boxed{}} & \boxed{\boxed{}} & \dots \\ i_{lo}=0 & j_{lo}=0 & \dots \end{matrix} \right)^T \right)^T$$

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 6.2). Compilation simply breaks the single iteration variable into higher- and lower-order components for purposes of indexing into the array being written to (Figure 6.3).

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

6.2 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 6.2). 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 6.3).

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 6.2).

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 6.3).

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.

6.3 Adjoint Introduction

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 6.4). 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.

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

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

$$\frac{0 \leq k \quad \text{shape}(v) = n :: s}{v = \text{trunc}_r n (\text{flatten} (\text{split } k v))}$$

We use this rewrite to wrap the entire program.

$$\text{trunc}_r n \left(\text{flatten} \left(\text{split } k \bigoplus_{i=0}^n [0 \leq i - 1] \cdot f(i - 1) \oplus f(i) \right) \right)$$

Next we unfold the split operator into its ATL definition.

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

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 and arrive at the following program.

$$\text{trunc}_r n \left(\text{flatten} \left(\bigoplus_{i_o=0}^{\lceil n/k \rceil} \bigoplus_{i_i=0}^k [i_o \times k + i_i < n] \cdot ([0 \leq i_o \times k + i_i - 1] \cdot f(i_o \times k + i_i - 1) \oplus f(i_o \times k + i_i)) \right) \right)$$

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.

$$\text{trunc}_r n \left(\text{flatten} \left(\left(\left(\bigoplus_{i_o=0}^1 \bigoplus_{i_i=0}^k [i_i < n] \cdot [1 \leq i_i] \cdot f(i_i - 1) \oplus f(i_i) \right) \circ \left(\bigoplus_{i_o=1}^{\lceil n/k \rceil} \bigoplus_{i_i=0}^k f(i_o \times k + i_i - 1) \oplus f(i_o \times k + i_i) \right) \circ \left(\bigoplus_{i_o=\lceil n/k \rceil}^{\lceil n/k \rceil} \bigoplus_{i_i=0}^k [i_o \times k + i_i < n] \cdot f(i_o \times k + i_i - 1) \oplus f(i_o \times k + i_i) \right) \right) \right) \right)$$

$$\begin{array}{c}
\frac{\text{shape}(v) = n :: _}{v = \text{trunc}_r n (\text{pad}_l k v)} \quad \frac{\text{shape}(v) = n :: _}{v = \text{trunc}_l n (\text{pad}_r k v)} \quad \frac{}{v = (v^T)^T} \\
\frac{\text{shape}(v) = n :: k :: _}{v = \text{split } k (\text{flatten } v)} \quad \frac{0 \leq k \quad \text{shape}(v) = n :: _}{v = \text{trunc}_r n (\text{flatten } (\text{split } k v))}
\end{array}$$

Figure 6.4: Dual-introduction rules

Chapter 7

Evaluation

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 [35]. 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 7.1) and two which are significantly beyond the scope of Halide’s scheduling language (Sections 7.2 & 7.3), 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.

7.1 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 3×3 kernel that can be expressed as the composition of two functions: `blur_x` and `blur_y`.

$$\text{blur}_x \ v \ x \ y := v[y; x - 1] \oplus v[y; x] \oplus v[y; x + 1]$$

$$\text{blur}_y \ v \ x \ y := \text{blur}_x \ v \ x \ (y - 1) \oplus \text{blur}_x \ v \ x \ y \oplus \text{blur}_x \ v \ x \ (y + 1)$$

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 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

$$\begin{aligned}
\text{let } \text{buf} & := \begin{array}{c} 1 \\ \boxed{} \\ y=0 \end{array} \dots \circ \\
& \begin{array}{c} n+1 \\ \boxed{} \\ y=1 \end{array} \left(\begin{array}{c} 1 \\ \boxed{} \\ x=0 \end{array} \dots \circ \begin{array}{c} m-1 \\ \boxed{} \\ x=1 \end{array} v[y-1; x-1] \oplus v[y-1; x] \oplus v[y-1; x+1] \circ \begin{array}{c} m \\ \boxed{} \\ x=m-1 \end{array} \dots \right) \circ \\
& \begin{array}{c} n+2 \\ \boxed{} \\ y=n+1 \end{array} \dots \\
\text{in } & \begin{array}{c} n \\ \boxed{} \\ y=0 \end{array} \begin{array}{c} m \\ \boxed{} \\ x=0 \end{array} \text{buf}[y; x] \oplus \text{buf}[y+1; x] \oplus \text{buf}[y+2; x]
\end{aligned}$$

Figure 7.1: Breadth-first schedule expressed in ATL

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 the resulting loops 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 7.1. 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 [35]. 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 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 7.2. 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 7.3, 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 64×64 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 `03` 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.

$$\begin{matrix} 1 \\ \boxed{} \\ \boxed{} \\ y=0 \end{matrix} \dots \circ \left(\begin{matrix} n-1 & 1 \\ \boxed{} & \boxed{} \\ y=1 & x=0 \end{matrix} \right)^T \circ (body)^T \circ \left(\begin{matrix} n-1 & m \\ \boxed{} & \boxed{} \\ y=1 & x=m-1 \end{matrix} \right)^T \circ \begin{matrix} n \\ \boxed{} \\ \boxed{} \\ y=n-1 \end{matrix} \dots$$

body is defined as:

```

truncr (n - 2)
flatten
 $\begin{pmatrix} (n-2)/k_n \\ \boxed{\phantom{0}} \\ y_o=0 \end{pmatrix} \circ \begin{pmatrix} (n-2)/k_n \\ \boxed{\phantom{0}} \\ y_o=0 \end{pmatrix} \dots$ 
truncr (m - 2)
flatten
 $\begin{pmatrix} (m-2)/k_m \\ \boxed{\phantom{0}} \\ x_o=0 \end{pmatrix} \circ \begin{pmatrix} (m-2)/k_m \\ \boxed{\phantom{0}} \\ x_o=0 \end{pmatrix} \dots$ 
let buf :=
 $\begin{matrix} n_k+2 & m_k \\ \boxed{\phantom{0}} & \boxed{\phantom{0}} \\ y_i=0 & x_i=0 \end{matrix} \quad l[y_o \times k_n + y_i; x_o \times k_m + x_i] \oplus$ 
 $\quad l[y_o \times k_n + y_i; x_o \times k_m + x_i + 1] \oplus$ 
 $\quad l[y_o \times k_n + y_i; x_o \times k_m + x_i + 2]$ 
in  $\left( \begin{matrix} k_n & k_m \\ \boxed{\phantom{0}} & \boxed{\phantom{0}} \\ y_i=0 & x_i=0 \end{matrix} \text{ buf}[y_i; x_i] \oplus \text{buf}[y_i + 1; x_i] \oplus \text{buf}[y_i + 2; x_i] \right)^T \right)^T$ 

```

Figure 7.2: 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 7.3: Performance of different schedules for the blur algorithm, our system vs. Halide

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

7.2 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 [30, 3]. 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 [30].

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

Figure 7.4: Rewrite sequence for scheduling a scatter-to-gather optimization on a simple program

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:

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

Notably, the form of the scatter involves an outermost summation that cannot be trivially 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 7.4](#). Each line corresponds to one rewrite written in our rescheduling framework, resulting in the final program shown below:

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

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.

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

Figure 7.5: Rewrite sequence for scheduling an im2col optimization on a simple convolution

7.3 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 7.5, 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. (A full 2D version would add in a vertical image dimension $\sum_{q=0}^H$ and vertical filter offset $\sum_{s=0}^R$.) 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.

Chapter 8

Implementation Details

In this section, we elaborate on the Coq-related technical details of parts of our framework. This includes the formulation of the notion of shape and consistency as separate from the type system of ATL as well as tactic machinery provided for overall safety-checking in a program before lowering.

8.1 Consistency and Shape

Let us describe the implementation of the formal property of consistency used in our framework that provides the functionality of the `shape` function abstraction used throughout this paper. One idiomatic Coq alternative would be to use dependently typed tensors, so extra checking of shape values would be unnecessary. However, dependently typed functional programming in Coq today is far from smooth, so we went with a halfway approach: we use dependent types just to assign a shape type to each dimensionality, in terms of nested tuples (rather than normal lists, as was the case with the `shape` function used in the statements of our previous rewrite theorems), one component per dimension. Then we apply not a shape *function* but a (partial) *relation* to connect expressions with shapes. Consistency of scalars is defined to be trivially true. Consistency of tensors is defined so that a tensor is consistent if and only if it is nonempty and every element it contains has the same shape and is also internally consistent. 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.

$$\frac{x : \mathbb{R}}{x \wr \cdot} \text{Scalar Consistency}$$
$$\frac{x \wr s \quad xs \wr (m, s)}{(x :: xs) \wr (m + 1, s)} \text{Tensor Consistency}$$

Figure 8.1: Consistency

$$\begin{array}{c}
\frac{\forall x. 0 \leq x < n \rightarrow e[x/i] \lambda s}{\left(\bigoplus_{i=0}^n e\right) \lambda (n, s)} \text{ Gen Consistency} \\
\\
\frac{\forall x. 0 \leq x < n \rightarrow e[x/i] \lambda s}{\left(\sum_{i=0}^n e\right) \lambda (n, s)} \text{ Sum Consistency} \\
\\
\frac{e \lambda s}{([p] \cdot e) \lambda s} \text{ Guard Consistency} \\
\\
\frac{e_1 \lambda s \quad e_2 \lambda s}{(e_1 \oplus e_2) \lambda s} \text{ Bin Consistency} \\
\\
\frac{e \lambda (n, s)}{(e[i]) \lambda s} \text{ Get Consistency} \\
\\
\frac{e_1 \lambda s \quad \forall x. x \lambda s \rightarrow \Gamma \vdash e_2[x/i] \lambda s'}{(\text{let } i := e_1 \text{ in } e_2) \lambda s'} \text{ Let Consistency}
\end{array}$$

Figure 8.2: Consistency

8.2 Access Safety

Our framework makes heavy use of the indicator function to guard branches of execution – thus protecting us from undefined behavior from out-of-bounds accesses on tensors. We provide a tactic called `check_safe` to check that all accesses within a given program are valid. In order to do so, this tactic follows a familiar pattern in our framework where it descends through program structure, collecting information-rich contexts. Once this tactic happens upon an access, it tries to prove that the index of that access is nonnegative and less than the length of the tensor. If the procedure succeeds, then we know that the C compilation of a term will not make out-of-bounds accesses, and thus it does not matter that C marks out-of-bounds accesses as undefined behavior, rather than always returning zero as in our semantics. Again, we chose this method vs. using dependent types to assure accesses, given the logistical baggage of dependent types in Coq today.

Chapter 9

Future Work

We have developed a verified framework for optimizing tensor programs through a series of algebraic rewrites, and our evaluation shows that this is able to match Halide’s performance for some important benchmarks. In this section we discuss prospective threads of future and ongoing work to provide a formally verified stack that supports writing tensor computations in high-level functional notation, choosing optimizations on them via proof scripts in Coq, feeding them through verified compilers, and running them on verified hardware, including data-parallel accelerators – leading to final Coq theorems relating the original functional programs to Verilog designs and their proper initial memory contents.

9.1 Streamlining the Optimization Process with Automation

User scheduling improves on the default experience of using a compiler. Programmers are used to being at the mercy of automatic optimizers, where they can influence the process at most by choosing among coarse-grained command-line flags. Instead, with user scheduling, explicit annotations in programs control which optimizations apply. However, with languages like Halide, it remains very difficult to add support for new kinds of scheduling directives, and even experts tend to introduce bugs that endanger soundness. Our preliminary work presents optimization directives as Coq *tactics*, or proof-producing procedures. As a result, no one needs to trust a new optimization directive, since its generated proofs can be checked from first principles. Coq also provides a Turing-complete tactic language that eases construction of new directives. The advantage is similar to the classic one for verification, comparing proof assistants to SMT solvers, where the latter at first appear to provide superior automation but can also more commonly lead users to “brick walls” of unsolved goals with no recourse to more manual proof.

However, the granularity of our optimization directives is much finer than in Halide. As a result, more programmer expertise is required to use them, and even experts will take longer to write scripts than they would need in Halide. The natural solution is the same as for verification in Coq: develop higher-level tactics on top of the ones we already wrote. The design space here is somewhat unexplored, with past user-scheduled languages providing only quite limited vocabularies, with no concern for easy extension with new directives.

An obvious first step is to draw on the suite of operators in Halide, TVM, and other such languages for inspiration. Similar abstractions should be feasible to provide with modest to moderate levels of

additional automation. One complexity is the need for higher-level tactics to infer sufficient but tight bounds for tensor comprehensions. To make schedules concise, Halide, for example, lets programmers elide bounds from most scheduling directives, leaving them instead to be inferred automatically from context. ATL is currently designed to require affine index expressions and loop bounds, which makes inferring tight bounds generally decidable. Beyond this, we believe there is opportunity to raise the abstraction further to blur the boundary between schedule-directed and automatic optimization. Rather than either manually specifying tiling choices or relying on a fully automatic “autoscheduler,” a programmer could employ an “auto-tiling” tactic which used a cost model or autotuning to search just over potential tiling choices, while leaving them free to control which tensors would be tiled and to perform arbitrary additional scheduling rewrites before and after. An advantage of working in Coq is that arbitrary outside logic can be employed to infer bounds or make higher-level scheduling decisions with no risk of affecting correctness, since the ultimate program and proof are still formally checked.

9.2 Increasing the Source Language’s Expressivity

To allow our tools to be used to write and optimize a broader range of tensor kernels, we plan to explore extensions to the source language ATL. Experience developing particular case-study programs will inform which extensions we tackle, but here we sketch one that we already have in mind, support for *sliding-window computations*.

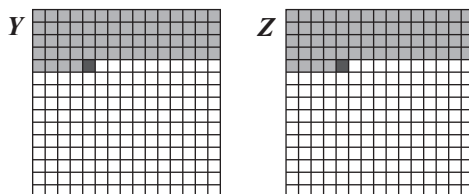
In our preliminary work, all explicit loop-style behavior comes either from tensor comprehensions \boxplus or summations \sum . Each considers all values of an index variable within a contiguous range of integers, and quite a variety of access patterns to existing tensors can be phrased in this way. However, it is difficult to encode general loop accumulators, where one cell of a generated tensor is defined partly in terms of values filled into other cells of the output *by the same loop*. Another way to describe the limitation is that ATL has very good support for “map”-style computations but weak support for “fold”-style computations, only providing summation as a special case.

Consider a 2D two-stage pipeline that averages 2×2 windows of an image. It first averages along one axis and then along the other.

$$\begin{aligned} \text{let } Y &:= \boxplus_{i=0}^n \boxplus_{j=0}^n [i > 0] \cdot X[i-1, j] + X[i, j] \text{ in} \\ \text{let } Z &:= \boxplus_{i=0}^n \boxplus_{j=0}^n [j > 0] \cdot Y[i, j-1] + Y[i, j] \text{ in} \\ &Z \end{aligned}$$

This organization of the computation is work-efficient (no redundant recomputation) but has poor locality (the entire Y intermediary buffer is pushed out to memory before any of its values are reused to compute Z). If we inline the computation of Y into the computation of Z , the trade-off reverses: we immediately use the Y intermediary values but have to recompute $Y[i, j]$ for $Z[i, j+1]$ rather than reusing the already computed value.

If we could interleave the computations of Y and Z , then we could achieve better locality without sacrificing work-efficiency. In Halide, this kind of optimization is known as a “sliding window” optimization. If we paused such a computation after computing $Z[i, j]$, we would see that the computed entries of the Y and Z buffers look like they do in the following image:



Given a simultaneous let-binding of both Y and Z , we would like to specify a different way to compute both buffers together.

We expect to do nontrivial design-space exploration in figuring out the best way to encode such computations. Our best idea of the moment is a kind of general recursive tensor comprehension. To

use the example of a prefix sum, we might encode it as:

$$X = \bigoplus_{i=0}^n (a[i] + [i > 0] \cdot X[i-1])$$

We add a left superscript on the comprehension, naming the final result. The body expression then carefully references that result, ideally respecting some total order on “recursive calls” to ensure termination. Such a feature is more difficult than it may seem to encode in Coq, where the functional programming language Gallina enforces termination of every definition. General recursion is famously involved to fit in such a language. To that end, we extend the hypothetical recursive comprehension to also take in a well-founded relation as parameter:

$$X[\lambda a, b. a+1=b] = \bigoplus_{i=0}^n (a[i] + [i > 0] \cdot X[i-1])$$

The relation $\lambda a, b. a + 1 = b$ expresses that recursive calls are only allowed on indices one lower. If the result tensor X is referenced in the body at any other index, we provide safe default behavior of returning a type-appropriate zero value. There is a natural alternative, of using dependent types to ensure all accesses respect the relation, but our experience suggests internalizing that kind of reasoning to type checking leads to excessive complexity in Coq. The trick we suggest has much in common with the way we already handle out-of-bounds tensor accesses, where, to support generation of efficient C code, we require proof that no such accesses actually occur. If this encoding of sliding windows pans out, then we will require the same kind of proof that recursion relations are always respected. Then we can generate code to loop over output positions in relation order.

9.3 Verifying Current Lowering to C

Our evaluation of our framework currently uses a trusted lowering process from pure, functional ATL into low-level, imperative C code. We are conducting ongoing work to verify this compilation process so as to reduce the trusted portion of our stack.

To do so we begin by formalizing the normal form of ATL that programs are normalized to before lowering. This is because the normalizing rewrites are the last of the verified transformations in our framework. We then formalize the specific subset of C that our lowering generates and define its semantics as a relation transforming starting state, represented as a stack and heap, to a final state that is what results after the imperative program executes. Our statement of correctness then relates the functional ATL semantics to the imperative one by stating the computed ATL value must reside in the specified memory location in the final state produced by the execution of the lowered program.

An ATL program is lowered in a destination-passing style, by passing down and building upon a continuation that represents the accumulation of the index expression for storage of the computed values. Given that this is effectively a construct representing where each value of the corresponding higher-dimensional ATL computed tensor will reside in memory from the execution of its lowering, it’s no surprise that being able to reason about its effects is the key to proving our top-level statement of correctness.

Although we eventually intend to retire this specific lowering to a C backend in favor of a new low-level IR with greater hardware-specifying expressivity, we expect a fair amount of proof structure and strategy to be not only portable but essential in the verification of compilation to a new target as well. This is because regardless of the specific target we chose, be it C or some other low-level imperative language, we expect to have to do similar kinds of algebraic reasoning about the accumulation of the storage continuation.

9.4 Proof-Generating Compilation to a New Lower-Level IR

Although our current work implements a lowering of ATL into C code, this naive translation is effectively specialized to a choice of available hardware operations, for instance providing no way to signal a call to a tensor processing unit (if available) for fast matrix multiply. Therefore, we propose to

build a proof-generating translation to a new intermediate language that exposes available hardware accelerators explicitly. Our starting point is the Exo language already designed and implemented [25].

Exo has been embedded in Python as a practical choice, but it is probably not the right move for supporting a fully proof-generating pipeline. As we embedded ATL in Coq, we propose to do the same with Exo. We will explore the tradeoff space between a shallowly embedded version (make programming with special memories look just like normal Coq functional programming, probably in an appropriate monad) and a deeply embedded version (define an explicit type of abstract syntax trees).

One important question here is the right kind of formal semantics to apply to ExoL-like programs. Our current guess is that the final meaning of a program should be in the style of interaction trees [41]. That is, we would work with explicit syntax trees of monadic programs calling stateful primitives from a menu that may vary from program to program. In this case, the menu is based on which special memories and instructions are configured. The operations are then (1) creation of special memories, (2) standard memory operations performed on them, and (3) invocation of special instructions.

Another design dimension here is the separation of semantics and compilation details for special memories. Exo programs mix interface and implementation, giving literal C code along with each callable procedure. Instead, standard memory operations could be given without accompanying code, and special instructions could be presented just as Coq functional programs to express semantics. With these semantic versions alone, we can develop a rewriting-based refinement framework for programmer-guided optimization of programs, working at this lower abstraction level than we explored with ATL. The technique must be adapted to the presence of explicit side effects, unless we come up with a trick similar to reshape operators, for presentation of these side effects as non-monadic pure code.

9.5 Verified Compilation From the New IR to Machine Code

The next transition must be to a low-level programming language that exposes certain hardware primitives directly. This new low-level IR will likely be very similar to the subset of C we compile to, so much of the verification process will likely be similar as well.

We will be starting from an Exo-inspired IR, outputting the Bedrock2 software language at the center of previous work conducted in end-to-end hardware-software verification [15]. Bedrock2 has been designed from the start to be flexible in exposing I/O primitives, with the chance to reuse core compiler proofs as the I/O model changes. The proposed work would stress-test that mechanism as a connection to accelerators. The top-level specification for an application should involve only simple I/O: input tensors arrive in memory somehow, then at termination output tensors should be found in memory. However, the I/O internal to the system’s operation gets more interesting. We want to enable separate proof of application, software compiler, processor, accelerators, and hardware compiler – it should be possible to replace any with a more optimized version, without needing to change the proofs of the others. That means we need to prove software programs against specifications with intricate dances of moving tensors back and forth between DRAM and scratchpads. Luckily, again, those are internal implementation details that do not infect the final theorem. However, we must still deal with the proof-engineering challenges they raise, on the path to that end theorem. Since this whole proposed project is more performance-oriented than in our past work [15], we expect to occasionally notice the need for new optimizations that are orthogonal to tensor programs.

9.6 Verification of Hardware Accelerators

Exo programs explain hardware connections by generating C source code. A compilation flow proved from first principles could use a different style, to simplify assignment of formal semantics to all pieces. We could adopt a variety of approaches to mechanize the semantics of hardware accelerators, but the gold standard to detect semantics mistakes is to prove the semantics *from both sides*, as popularized in the NSF Expedition on the Science of Deep Specification (DeepSpec) [1], in which Chlipala was a PI. That is, when formalizing an interface between software and hardware, it pays to prove both software and hardware against that interface. DeepSpec studied such questions, but without any hardware accelerators.

To adapt that activity to high-performance computing with tensors, the first research direction we propose is formal verification of hardware accelerators in Coq. One outgrowth of the DeepSpec

work is the Kôika hardware description language and verification framework [5]. Kôika is inspired by the hardware language Bluespec, which provides higher-level modularity features than found in more popular languages like Verilog, allowing more techniques from software verification to be ported over, including proof of libraries against clean interfaces. At the same time, there is a Coq-verified compiler to Verilog RTL, so the trust story is not endangered by the move to higher-level code. The canonical examples implemented in Kôika have been general-purpose processors, and work is proceeding on developing tools to prove their functional correctness, adapting ideas from our predecessor project Kami [11], where modular proof was given of an infinite family of RISC-V multicore systems with cache-coherent shared memory.

Though Bluespec itself has been used to implement hardware accelerators, the prior work with Kami and Kôika has not proved accelerators. We propose to undertake several such proofs in this project. The exact set of accelerators will be determined by our experiences coding and performance-tuning example kernels. Our best guess at the moment is that we should tackle a *vector ISA extension* and a simple *tensor processing unit (TPU)*. These are interesting for their different styles of hardware interface.

The vector extension introduces new machine-code instructions, for applying operators in parallel across arrays. We would almost certainly implement a subset of the standardized vector extension for the RISC-V architecture family, as we already have a pipelined processor in Kôika for the base instruction set, whose correctness Coq proof is underway. The new work would involve modifying that processor to support vector instructions, along with extending our ISA formal specification and adapting the processor correctness proof.

The TPU would support variants of dense matrix multiplication and perhaps a few other operations. Its implementation in Kôika would likely look interestingly different from a processor's. We expect some of the main complications would be in the use of special registers and memory regions (e.g., scratchpads) for communicating large tensors back and forth between processor and accelerator. Indeed, with this accelerator we do not get out of modifying the processor and its specification and correctness proof. There are new research challenges in specifying and verifying the two pieces modularly, even before we bring in proof of software programs. Co-PI Ragan-Kelley has already helped develop a relatively simple but efficient TPU-like architecture family that functions as a RISC-V co-processor [18]. We will likely begin by porting parts of that design from Chisel to Kôika and developing a formal specification of its functionality.

Another nontrivial aspect is adding hardware support for the floating-point numbers assumed by most tensor kernels. We hope to build upon the Coq library Flocq [4] used by the verified C compiler CompCert [29]. Perhaps we can implement proof-generating translation of functional programs from that library to Kôika expressions, providing a convenient path to verified implementations. However, we are allocating time for significant implementation and proof work to obtain a RISC-V-compatible implementation.

With these software-hardware interfaces formalized (and validated with proof of the hardware implementations), we can connect to the Exo-style IR of the prior subsection. The reference implementation of each special instruction can be proved equivalent to machine-code instructions that are also given, by referring to what the hardware semantics says about those lower-level instructions. That exercise validates the rewrites applied to the IR, by grounding them in hardware implementation.

Bibliography

- [1] APPEL, A. W., BERINGER, L., CHLIPALA, A., PIERCE, B. C., SHAO, Z., WEIRICH, S., AND ZDANCEWIC, S. Position paper: the science of deep specification. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 375 (2017).
- [2] BAUER, M., TREICHLER, S., SLAUGHTER, E., AND AIKEN, A. Legion: expressing locality and independence with logical regions. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12* (Piscataway, NJ, USA, Nov 2012), IEEE, p. 66.
- [3] BERNSTEIN, G., MARA, M., LI, T.-M., MACLAURIN, D., AND RAGAN-KELLEY, J. Differentiating a tensor language, 2020.
- [4] BOLDO, S., AND MELQUIOND, G. Flocq: A unified library for proving floating-point algorithms in Coq. In *2011 IEEE 20th Symposium on Computer Arithmetic* (2011), pp. 243–252.
- [5] BOURGEAT, T., PIT-CLAUDEL, C., CHLIPALA, A., AND ARVIND. The essence of Bluespec: A core language for rule-based hardware design. In *PLDI'20: Proceedings of the ACM SIGPLAN 2020 Conference on Programming Language Design and Implementation* (June 2020).
- [6] CHAKRAVARTY, M. M. T., KELLER, G., LEE, S., MCDONELL, T. L., AND GROVER, V. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming* (New York, NY, USA, 2011), M. Carro and J. H. Reppy, Eds., Association for Computing Machinery, pp. 3–14.
- [7] CHAMBERLAIN, B., CALLAHAN, D., AND ZIMA, H. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312.
- [8] CHAMBERLAIN, B. L. *The design and implementation of a region-based parallel programming language*. PhD thesis, The University of Washington, 2001.
- [9] CHEN, C., CHAME, J., AND HALL, M. CHILL: A framework for composing high-level loop transformations. Tech. rep., University of Southern California, 2008.
- [10] CHEN, T., MOREAU, T., JIANG, Z., ZHENG, L., YAN, E., COWAN, M., SHEN, H., WANG, L., HU, Y., CEZE, L., GUESTRIN, C., AND KRISHNAMURTHY, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Berkeley, CA, USA, 2018), OSDI'18, USENIX Association, pp. 579–594.
- [11] CHOI, J., VIJAYARAGHAVAN, M., SHERMAN, B., CHLIPALA, A., AND ARVIND. Kami: A platform for high-level parametric hardware specification and its modular verification. In *ICFP'17: Proceedings of the 22nd ACM SIGPLAN International Conference on Functional Programming* (Sept. 2017).
- [12] DELAWARE, B., PIT-CLAUDEL, C., GROSS, J., AND CHLIPALA, A. Fiat: Deductive synthesis of abstract data types in a proof assistant. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015* (2015), pp. 689–700.

- [13] DELAWARE, B., SURIYAKARN, S., PIT-CLAUDEL, C., YE, Q., AND CHLIPALA, A. Narcissus: Correct-by-construction derivation of decoders and encoders from binary formats. In *Proc. ICFP* (Aug. 2019).
- [14] DONADIO, S., BRODMAN, J. C., ROEDER, T., YOTOV, K., BARTHOU, D., COHEN, A., GARZARÁN, M. J., PADUA, D. A., AND PINGALI, K. A language for the compact representation of multiple program versions. In *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005* (Berlin, Heidelberg, 2005), Springer Berlin Heidelberg, pp. 136–151.
- [15] ERBSEN, A., GRUETTER, S., CHOI, J., WOOD, C., AND CHLIPALA, A. Integration verification across software and hardware for a simple embedded system. In *PLDI'21* (2021).
- [16] FATAHALIAN, K., HORN, D. R., KNIGHT, T. J., LEEM, L., HOUSTON, M., PARK, J. Y., EREZ, M., REN, M., AIKEN, A., DALLY, W. J., AND HANRAHAN, P. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (New York, NY, USA, 2006), SC '06, Association for Computing Machinery, p. 83–es.
- [17] FU, R., QIN, X., DARDHA, O., AND STEUWER, M. Row-polymorphic types for strategic rewriting, 2021.
- [18] GENC, H., KIM, S., AMID, A., HAJ-ALI, A., IYER, V., PRAKASH, P., ZHAO, J., GRUBB, D., LIEW, H., MAO, H., OU, A., SCHMIDT, C., STEFFL, S., WRIGHT, J., STOICA, I., RAGAN-KELLEY, J., ASANOVIC, K., NIKOLIC, B., AND SHAO, Y. S. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)* (2021).
- [19] GRAHAM, R. L., KNUTH, D. E., AND PATASHNIK, O. *Concrete Mathematics*. Addison Wesley, 2011, p. 36–37.
- [20] HAGEDORN, B., ELLIOTT, A. S., BARTHEL, H., BODIK, R., AND GROVER, V. Fireiron: A scheduling language for high-performance linear algebra on GPUs, 2020.
- [21] HARTONO, A., NORRIS, B., AND SADAYAPPAN, P. 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* (Piscataway, NJ, USA, 2009), IEEE, pp. 1–11.
- [22] HENRIKSEN, T., SERUP, N. G. W., ELSMAN, M., HENGLEIN, F., AND OANCEA, C. E. 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* (New York, NY, USA, 2017), PLDI 2017, ACM, pp. 556–571.
- [23] HIETALA, K., RAND, R., HUNG, S.-H., WU, X., AND HICKS, M. A verified optimizer for quantum circuits. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan 2021), 1–29.
- [24] HU, Y., LI, T., ANDERSON, L., RAGAN-KELLEY, J., AND DURAND, F. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.* 38, 6 (2019), 201:1–201:16.
- [25] IKARASHI, Y., BERNSTEIN, G. L., REINKING, A., GENC, H., AND RAGAN-KELLEY, J. Exo-compilation for productive programming of hardware accelerators. *Proceedings of the 43rd ACM SIGPLAN Conference on Programming Language Design and Implementation* (2022).
- [26] IVERSON, K. E. *A Programming Language*. John Wiley & Sons, Inc., New York, NY, USA, 1962.
- [27] KJOLSTAD, F., KAMIL, S., CHOU, S., LUGATO, D., AND AMARASINGHE, S. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (oct 2017), 1–29.
- [28] KOMMRUSCH, S., BAROLLET, T., AND POUCHET, L.-N. Proving equivalence between complex expressions using graph-to-sequence neural models, 2021.
- [29] LEROY, X. A Formally Verified Compiler Back-end. *Journal of Automated Reasoning* 43, 4 (Dec. 2009), 363–446.

- [30] LI, T.-M., GHARBI, M., ADAMS, A., DURAND, F., AND RAGAN-KELLEY, J. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)* 37, 4 (2018), 139:1–139:13.
- [31] LIU, A., BERNSTEIN, G., CHLIPALA, A., AND RAGAN-KELLEY, J. Verified tensor-program optimization via high-level scheduling rewrites. In *POPL'22: Proceedings of the 49th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Jan. 2022).
- [32] PASZKE, A., JOHNSON, D. D., DUVENAUD, D., VYTINIOTIS, D., RADUL, A., JOHNSON, M. J., RAGAN-KELLEY, J., AND MACLAURIN, D. 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)* (August 2021), ACM SIGPLAN, ACM.
- [33] PIT-CLAUDEL, C., WANG, P., DELAWARE, B., GROSS, J., AND CHLIPALA, A. 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* (June 2020).
- [34] RAGAN-KELLEY, J., ADAMS, A., PARIS, S., LEVOY, M., AMARASINGHE, S. P., AND DURAND, F. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4 (2012), 32:1–32:12.
- [35] RAGAN-KELLEY, J., BARNES, C., ADAMS, A., PARIS, S., DURAND, F., AND AMARASINGHE, S. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proc. PLDI* (Seattle, 2013), ACM.
- [36] SLEPAK, J., SHIVERS, O., AND MANOLIOS, P. An array-oriented language with static rank polymorphism. In *Programming Languages and Systems* (Berlin, Heidelberg, 2014), Z. Shao, Ed., Springer Berlin Heidelberg, pp. 27–46.
- [37] SMITH, G. H., LIU, A., LYUBOMIRSKY, S., DAVIDSON, S., MCMAHAN, J., TAYLOR, M., CEZE, L., AND TATLOCK, Z. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (New York, NY, USA, 2021), MAPS 2021, Association for Computing Machinery, p. 21–31.
- [38] STEUWER, M., FENSCH, C., LINDLEY, S., AND DUBACH, C. 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* (09 2015), vol. 50, Association for Computing Machinery.
- [39] VASILACHE, N., ZINENKO, O., THEODORIDIS, T., GOYAL, P., DEVITO, Z., MOSES, W. S., VERDOOLAEGE, S., ADAMS, A., AND COHEN, A. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions, 2018.
- [40] VENKAT, A., RUSIRA, T., BARIK, R., HALL, M., AND TRUONG, L. Swirl: High-performance many-core CPU code generation for deep neural networks. *The International Journal of High Performance Computing Applications* 33, 6 (2019), 1275–1289.
- [41] XIA, L., ZAKOWSKI, Y., HE, P., HUR, C., MALECHA, G., PIERCE, B. C., AND ZDANCEWIC, S. Interaction trees: representing recursive and impure programs in Coq. *Proc. ACM Program. Lang.* 4, POPL (2020), 51:1–51:32. Distinguished paper award.
- [42] YI, Q., SEYMOUR, K., YOU, H., VUDUC, R. W., AND QUINLAN, D. J. POET: parameterized optimizations for empirical tuning. In *21st International Parallel and Distributed Processing Symposium (IPDPS 2007)* (Piscataway, NJ, USA, 2007), IEEE, pp. 1–8.
- [43] ZHANG, Y., YANG, M., BAGHDADI, R., KAMIL, S., SHUN, J., AND AMARASINGHE, S. P. Graphit: a high-performance graph DSL. *PACMPL* 2, OOPSLA (2018), 121:1–121:30.