Verified Tensor-Program Optimization Via High-Level Scheduling Rewrites

AMANDA LIU, Massachusetts Institute of Technology, USA
GILBERT LOUIS BERNSTEIN, University of California, Berkeley, USA
ADAM CHLIPALA, Massachusetts Institute of Technology, USA
JONATHAN RAGAN-KELLEY, Massachusetts Institute of Technology, USA

We present 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. Our 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 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.

CCS Concepts: • **Software and its engineering → Domain specific languages**; **Software verification**; *Compilers.*

Additional Key Words and Phrases: formal verification, proof assistants, array programming, optimization

ACM Reference Format:
Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization Via High-Level Scheduling Rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (January 2022), 28 pages. https://doi.org/10.1145/3498717

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 [Ragan-Kelley et al. 2013] and TVM for machine learning [Chen et al. 2018] have emerged to directly facilitate 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

Authors' addresses: Amanda Liu, CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA, lamanda@mit.edu; Gilbert Louis Bernstein, University of California, Berkeley, Berkeley, CA, USA, gilbo@berkeley.edu; Adam Chlipala, CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA, adamc@csail.mit.edu; Jonathan Ragan-Kelley, CSAIL, Massachusetts Institute of Technology, Cambridge, MA, USA, jrk@csail.mit.edu.

2475-1421/2022/1-ART55
https://doi.org/10.1145/3498717

single source files, which have no formal connections to alternative implementations of matching algorithms.

In this paper, we present 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 [Fu et al. 2021], Glenside [Smith et al. 2021], and others [Kommrusch et al. 2021; Steuwer et al. 2015]. 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. [2015] 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 paper, our primary point of comparison is Halide [Ragan-Kelley et al. 2013], 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.

Summing up, we claim 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*.

[1] We 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.

- 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.

2 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 [Ragan-Kelley et al. 2013] 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 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.

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

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

Although this 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.

$$\text{trunc}_r\ n \boxplus_{i_o=0}^{\lceil n/8 \rceil} \Big(\text{let } v := \boxplus_{i_i=0}^{8} [\![\ i_o * 8 + i_i < n\ ]\!] \cdot (([\![\ 0 \le i_o * 8 + i_i - 1\ ]\!] \cdot f[i_o * 8 + i_i - 1]) \oplus f[i_o * 8 + i_i]) \text{ in } \boxplus_{i_i=0}^{8} [\![\ i_o * 8 + i_i < n\ ]\!] \cdot v[i_i] \Big)$$

The C code generated from the tiled ATL program is:

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

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

It is not obvious that all important scheduling optimizations can be performed on terms as high-level as in ATL, but one of our main research contributions is demonstrating an effective interplay between *reshape operators* like trunc_r as introduced above and the process of compiling to C, such that functional programs signal all important design decisions for nested imperative

loops. Interestingly, the reshape operators are defined in terms of more basic operators like ⊞, not in terms of some explicitly imperative semantics as in past work with proved rewrite laws, making it relatively easy to prove the rewrites we need for effective optimization. Let us now see those core primitives spelled out, before turning to basic scheduling rewrites, compilation to C, and how reshape operators interact with both.

3 CORE 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 [Bernstein et al. 2020].

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.

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

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

Given the loop bounds in this program, $buf[i-1]$ will be out-of-bounds for some iterations in the second stage. Specifically, in the first iteration of the generation when $i = 0$, there is a resulting attempt to access buf at -1. As a fix, we logically guard the expression with the condition $0 \le i-1$, a predicate that ensures the access is valid. To do so, we use the indicator function from Ken Iverson's APL that returns 1 if its condition is true and 0 otherwise [Graham et al. 2011]. 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 $\mathbb{Z} \to X$, so the overall expression has type X, where X is effectively a type-unification variable. In ATL, the possible instantiations of X are limited to a class of scalar or tensor types that we elaborate on next. Therefore, for groups of operators that have type-specific implementations but maintain the same algebraic properties, we introduce single polymorphic operators, as is the case with addition and $\oplus$.

3.1 Specification

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.

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

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 ($\sum$), tensor access ([]), and Iverson's bracket or guard ([ ]) can be found in the denotational semantics we present in Figure 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:

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

This simple program computes a tensor that comprises tensors of various lengths–a complexity that we wish to disallow. The details of the logical mechanisms we use to implement this constraint on ATL programs are elaborated on more formally in Section 7.1, but for now we introduce a simpler convention. We will use a 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:

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

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:

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

Syntax and Semantics. The complete syntax of our embedding appears in Figure 1, with a denotational semantics excerpted in Figure 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 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.

4 THE SCHEDULING-REWRITE FRAMEWORK

In this section, we detail the construction and utility of our rewrite framework so as to allow high-level, user scheduling of ATL programs through a series of algebraic rewrites verified within Coq. In order to do so, we introduce some common and useful examples of the rewrites we have

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

Fig. 1. Core ATL syntax

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

Fig. 2. Denotational semantics for the core of the ATL language (selected rules)

proven as theorems in our framework, as well as the lemmas we have proven to provide logical machinery necessary for automated, conditional rewriting under binders and logical contexts.

4.1 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.

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

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

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

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:

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

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

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

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

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

Most scheduling rewrites provided in this framework are similar to this example in requiring proof of bounds at the rewrite site–possibly under binders.

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:

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

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:

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

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

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

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

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

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

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

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

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

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

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

Now in addition to the information provided from the generation that $0 \le i < n$, the context includes the constraint that $0 \le i-1$. This is sufficient information to ensure the validity of the access, and so the scheduling rewrite may be applied here. Finally, we arrive at the following fully fused schedule of the two-stage pipeline:

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

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, even expressions generated from the use of reshape operators 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 demonstrated in this paper.

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 present the stages of the lowering process and the rules for code generation.

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 2.

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

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

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

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

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

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

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:

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

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:

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

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

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

The notable differences here are that the leaf-node expression being computed and stored is no longer a sum of $f(i-1)$ and $f(i)$ but an access into $f(i-1)$ and $f(i)$, and there is a new loop nest

explicitly introduced into the ATL source. There is once again a polymorphic $\oplus$ operator introduced by a rewrite lemma. However, the dimensionality of this program is still known, so this operator can once again be normalized into a specific instance, yielding the following program.

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

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 type and need to be normalized.

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

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

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 3. 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 2, we will continue using the same $[\![\ ]\!]$ 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.

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

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.

$$
\begin{array}{lll}
\text{let} & \lambda_{\text{idx}}(0) & = () \\
& \lambda_{\text{idx}}(1) & = \lambda i_0.(i_0) \\
& \lambda_{\text{idx}}(2) & = \lambda i_0.\lambda i_1.(i_0, i_1) \\
& & \cdots
\end{array}
$$

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

Fig. 3. Compilation rules for lowering the core ATL constructs into C

```
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] ⟩
   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.

6 RESHAPE OPERATORS

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 3 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

Concatenate	$e \circ e$
Transpose	e^T
Flatten	flatten e
Split	split I e
Pad on the Right	pad_r I e
Pad on the Left	pad_l I e
Truncate from the Right	trunc_r I e
Truncate from the Left	trunc_l I e

Fig. 4. Reshape operators

$$e_1 \circ e_2 := \boxplus_{i=0}^{|e_1|+|e_2|} \left[\!\left[\ i < |e_1|\ \right]\!\right] \cdot e_1[i] \oplus \left[\!\left[\ |e_1| \le i\ \right]\!\right] \cdot e_2[i - |e_1|]$$

$$e^T := \boxplus_{x=0}^{|e[0]|} \boxplus_{y=0}^{|e|} e[y;x]$$

$$\text{flatten } e := \boxplus_{i=0}^{|e|\times|e[0]|} \sum_{j=0}^{|e|} \sum_{k=0}^{|e[0]|} \left[\!\left[\ i = j \times |e[0]| + k\ \right]\!\right] \cdot e[j;k]$$

$$\text{split } k\ e := \boxplus_{i=0}^{\lceil |e|/k \rceil} \boxplus_{j=0}^{k} \left[\!\left[\ i \times k + j < |e|\ \right]\!\right] \cdot e[i \times k + j]$$

$$\text{pad}_\text{r}\ k\ e := \boxplus_{i=0}^{|e|+k} \left[\!\left[\ i < |e|\ \right]\!\right] \cdot e[i] \qquad \text{trunc}_\text{r}\ k\ e := \boxplus_{i=0}^{k} e[i]$$

$$\text{pad}_\text{l}\ k\ e := \boxplus_{i=0}^{|e|+k} \left[\!\left[\ k \le i\ \right]\!\right] \cdot e[i-k] \qquad \text{trunc}_\text{l}\ k\ e := \boxplus_{i=0}^{k} e[i + |e| - k]$$

Fig. 5. Definitions of reshape operators

special strategies when lowering to C. In this section, we present our set of reshape operators in Figure 4 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 5) in terms of those we worked with in prior sections, but their payoff is in signaling clever ways of manipulating the order (Figure 6) in which array assignments happen within loop nests.

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 5. 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.

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

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

Fig. 6. Compilation rules for reshape operators

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

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

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

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:

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

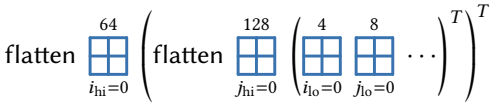
After executing the `simpl_guard` tactic, the pipeline program arrives at the following schedule:

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

Transpose. When applied to a matrix, the transpose operator performs the equivalent function as its mathematical counterpart, in that it switches row and column indices and as a result produces an expression flipped along its diagonal with the outermost dimension swapped with the dimension immediately inside. Thanks to shape polymorphism, this operation naturally extends the mathematical definition of a matrix transpose and is well-defined in higher dimensions (Figure 5). In code generation, this operator is implemented by switching the indices associated with the dimensions being transposed inside the assignment-indexing expression (Figure 6).

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 5, 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). 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.



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 5). 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).

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

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 5). 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).

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 5).

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).

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 7). Once the identity pair has been introduced into the program, one of the operators (often the inner operator) is unfolded to its definition in terms of basic ATL operators. Then it is simplified into the rest of the program, exposing opportunities for further scheduling. The other operator remains intact to serve as a compilation directive, inducing the desired decoupling between compute and storage order. We have found this pattern to be useful in many common situations.

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

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

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

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

We use this rewrite to wrap the entire program.

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

Next we unfold the split operator into its ATL definition.

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

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.

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

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

In order to further accelerate computation, we may choose to break off a loop prologue and epilogue, using the concatenation rewrite rule. After doing so, we can eliminate all guards from the

$$\frac{\text{shape}(v) = n :: \_}{v = \text{trunc}_\text{r}\ n\ (\text{pad}_\text{l}\ k\ v)} \qquad \frac{\text{shape}(v) = n :: \_}{v = \text{trunc}_\text{l}\ n\ (\text{pad}_\text{r}\ k\ v)} \qquad \frac{}{v = (v^T)^T}$$

$$\frac{\text{shape}(v) = n :: k :: \_}{v = \text{split}\ k\ (\text{flatten}\ v)} \qquad \frac{0 \le k \qquad \text{shape}(v) = n :: \_}{v = \text{trunc}_\text{r}\ n\ (\text{flatten}\ (\text{split}\ k\ v))}$$

Fig. 7. Dual-introduction rules

main loop, which will expose the loop to further unrolling and vectorization optimizations, even if the prologue and epilogue continue to use scalar instructions.

$$\begin{aligned}
&\text{trunc}_\text{r}\ n\ \Bigg(\text{flatten} \\
&\Bigg(\Bigg(\boxplus_{i_o=0}^{1}\ \boxplus_{i_i=0}^{k}\ [\![\ i_i < n\ ]\!] \cdot [\![\ 1 \le i_i\ ]\!] \cdot f(i_i - 1) \oplus f(i_i)\Bigg) \circ \\
&\Bigg(\boxplus_{i_o=1}^{\lfloor n/k \rfloor}\ \boxplus_{i_i=0}^{k}\ f(i_o \times k + i_i - 1) \oplus f(i_o \times k + i_i)\Bigg) \circ \\
&\Bigg(\boxplus_{i_o=\lfloor n/k \rfloor}^{\lceil n/k \rceil}\ \boxplus_{i_i=0}^{k}\ [\![\ i_o \times k + i_i < n\ ]\!] \cdot f(i_o \times k + i_i - 1) \oplus f(i_o \times k + i_i)\Bigg)\Bigg)\Bigg)
\end{aligned}$$

7 IMPLEMENTATION DETAIL

In this section, we elaborate on the Coq-related technical details of parts of our framework.

7.1 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.

7.2 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.

8 EXPERIMENTAL 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 [Ragan-Kelley et al. 2013]. In this section we demonstrate the capability of our framework to do so by showing its expressiveness and flexibility to perform complex scheduling transformations on three programs spread across this trade-off space. This includes one example whose scheduling transformations correspond to the core features of Halide (Section 8.1) and two which are significantly beyond the scope of Halide's scheduling language (Sections 8.2 & 8.3), demonstrating our system's generality and extensibility. Additionally, to demonstrate that our system is capable of generating comparably optimized code to state-of-the-art systems, for the programs to which it applies, we provide performance benchmarks against their equivalents written in Halide.

8.1 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`.

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

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

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

One thing to note is that if this algorithm were to be applied over the the full extent of an input–say a blur over an entire image–there would be out-of-bounds accesses along the border since each stage operates over a nontrivial window. In order to remain consistent, both Halide and our framework use the strategy of conditionally guarding these boundary accesses and returning 0 for what would be out-of-bounds accesses and partitioning 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 8. 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 [Ragan-Kelley et al. 2013]. 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.

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

Fig. 8. Breadth-first schedule expressed in ATL

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 9. 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 10, 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 `O3` flags enabled. A snapshot of Halide was taken in early June 2021 and built against LLVM 12.0. Halide tiling was set to use the `GuardWithIf` strategy for loop tails/epilogues. The benchmark was performed using 2000×2000 input and output buffers on an iMac Pro with a 3.2 GHz 8-Core Skylake Xeon processor.

It can be seen that for both schedules, our system is able to express a schedule achieving comparable performance to a roughly equivalent schedule programmed in Halide. Results do not match precisely for a number of potential reasons: First, Halide automates the splitting off of loop epilogues and prologues, whereas ATL places the way in which to do 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.

8.2 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

$$\boxplus_{y=0}^{1} .. \circ \left(\left(\boxplus_{y=1}^{n-1} \boxplus_{x=0}^{1} .. \right)^T \circ (body)^T \circ \left(\boxplus_{y=1}^{n-1} \boxplus_{x=m-1}^{m} .. \right)^T \right)^T \circ \boxplus_{y=n-1}^{n} ..$$

body is defined as:

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

Fig. 9. Tiled schedule expressed in ATL

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

Fig. 10. Performance of different schedules for the blur algorithm, our system vs. Halide

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 [Bernstein et al. 2020; Li et al. 2018]. 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 [Li et al. 2018].

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

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

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

Fig. 11. Rewrite sequence for scheduling a scatter-to-gather optimization on a simple program

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 11. Each line corresponds to one rewrite written in our rescheduling framework, resulting in the final program shown below:

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

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

8.3 Im2col

Early convolutional neural networks exploited high-throughput GPUs by transforming batched convolution operations into matrix multiplies. This transformation worked by first marshalling

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

Fig. 12. Rewrite sequence for scheduling an im2col optimization on a simple convolution

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 12, 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 $\boxplus_{q=0}^{H}$ 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.

9 RELATED WORK

Languages and compilers that offer explicit programmer control over program transformations have a longer history in HPC [Chen et al. 2008; Donadio et al. 2005; Fatahalian et al. 2006; Hartono et al. 2009; Yi et al. 2007], 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 [Ragan-Kelley et al. 2012; Ragan-Kelley et al. 2013]. This approach has since been adapted to dense linear algebra and machine learning [Chen et al. 2018; Hagedorn et al. 2020; Vasilache et al. 2018; Venkat et al. 2019], sparse tensor algebra [Kjolstad et al. 2017], distributed-memory computing [Bauer et al. 2012], graph processing [Zhang et al. 2018], and physical simulation [Hu et al. 2019]. 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. Key to our approach is defining scheduling transformations in terms of algebraic rewrite rules. Recent work develops rewrite-rule systems for optimizing array programs [Fu et al. 2021; Steuwer et al. 2015], 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 [Hietala et al. 2021]. 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.

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 [Delaware et al. 2015] and binary-format parsers [Delaware et al. 2019]. 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 [Pit-Claudel et al. 2020], and we would benefit from adopting similar techniques in future work.

Finally, as a functional tensor language, our program representation (ATL) builds on ideas from array languages [Chamberlain et al. 2007; Chamberlain 2001; Iverson 1962; Slepak et al. 2014] which have more recently been explored in a functional context [Chakravarty et al. 2011; Henriksen et al. 2017; Paszke et al. 2021]. It is specifically derived from recent work on a functional tensor language for automatic differentiation [Bernstein et al. 2020], 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 [Smith et al. 2021] attempts to capture some similar implementation details by augmenting a functional tensor language with an algebra of "access patterns."

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 1745302, as well as DARPA under the PAPPA (agreement HR00112090017) and RTML (contract FA8650-20-2-7006) programs.

REFERENCES

Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: expressing locality and independence with logical regions. In *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12* (Salt Lake City, UT, USA). IEEE, Piscataway, NJ, USA, 66. https://doi.org/10.1109/SC.2012.71

Gilbert Bernstein, Michael Mara, Tzu-Mao Li, Dougal Maclaurin, and Jonathan Ragan-Kelley. 2020. Differentiating a Tensor Language. arXiv:2008.11256 [cs.PL]

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *Proceedings of the POPL 2011 Workshop on Declarative Aspects of Multicore Programming*, Manuel Carro and John H. Reppy (Eds.). Association for Computing Machinery, New York, NY, USA, 3–14. https://doi.org/10.1145/1926354.1926358

B.L. Chamberlain, D. Callahan, and H.P. Zima. 2007. Parallel Programmability and the Chapel Language. *The International Journal of High Performance Computing Applications* 21, 3 (2007), 291–312. https://doi.org/10.1177/1094342007078442

Bradford L. Chamberlain. 2001. *The design and implementation of a region-based parallel programming language.* Ph.D. Dissertation. The University of Washington.

Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. University of Southern California.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-end Optimizing Compiler for Deep Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI'18)*. USENIX Association, Berkeley, CA, USA, 579–594. http://dl.acm.org/citation.cfm?id=3291168.3291211

Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. 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*. 689–700. https://doi.org/10.1145/2676726.2677006

Benjamin Delaware, Sorawit Suriyakarn, ClÃĩment Pit-Claudel, Qianchuan Ye, and Adam Chlipala. 2019. Narcissus: Correct-By-Construction Derivation of Decoders and Encoders from Binary Formats. In *Proc. ICFP* (Berlin, Germany). https://doi.org/10.1145/3341686

Sébastien Donadio, James C. Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David A. Padua, and Keshav Pingali. 2005. A Language for the Compact Representation of Multiple Program Versions. In *Languages and Compilers for Parallel Computing, 18th International Workshop, LCPC 2005*. Springer Berlin Heidelberg, Berlin, Heidelberg, 136–151. https://doi.org/10.1007/978-3-540-69330-7_10

Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhoon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. 2006. Sequoia: Programming the Memory Hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing* (Tampa, Florida) *(SC '06)*. Association for Computing Machinery, New York, NY, USA, 83âĂŞes. https://doi.org/10.1145/1188455.1188543

Rongxiao Fu, Xueying Qin, Ornela Dardha, and Michel Steuwer. 2021. Row-Polymorphic Types for Strategic Rewriting. arXiv:2103.13390 [cs.PL]

Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. 2011. *Concrete Mathematics.* Addison Wesley, 36âĂŞ37.

Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Scheduling Language for High-Performance Linear Algebra on GPUs. arXiv:2003.06324 [cs.PL]

Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. 2009. Annotation-based empirical performance tuning using Orio. In *23rd IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2009, Rome, Italy, May 23-29, 2009* (Rome, Italy). IEEE, Piscataway, NJ, USA, 1–11. https://doi.org/10.1109/IPDPS.2009.5161004

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. 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* (Barcelona, Spain) *(PLDI 2017)*. ACM, New York, NY, USA, 556–571. https://doi.org/10.1145/3062341.3062354

Kesha Hietala, Robert Rand, Shih-Han Hung, Xiaodi Wu, and Michael Hicks. 2021. A verified optimizer for Quantum circuits. *Proceedings of the ACM on Programming Languages* 5, POPL (Jan 2021), 1âĂŞ29. https://doi.org/10.1145/3434318

Yuanming Hu, Tzu-Mao Li, Luke Anderson, Jonathan Ragan-Kelley, and Frédo Durand. 2019. Taichi: a language for high-performance computation on spatially sparse data structures. *ACM Trans. Graph.* 38, 6 (2019), 201:1–201:16. https://doi.org/10.1145/3355089.3356506

Kenneth E. Iverson. 1962. *A Programming Language.* John Wiley & Sons, Inc., New York, NY, USA.

Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The tensor algebra compiler. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (oct 2017), 1–29. https://doi.org/10.1145/3133901

Steve Kommrusch, Théo Barollet, and Louis-Noël Pouchet. 2021. Proving Equivalence Between Complex Expressions Using Graph-to-Sequence Neural Models. arXiv:2106.02452 [cs.PL]

Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. 2018. Differentiable programming for image processing and deep learning in Halide. *ACM Trans. Graph. (Proc. SIGGRAPH)* 37, 4 (2018), 139:1–139:13. https://doi.org/10.1145/3197517.3201383

Adam Paszke, Daniel D. Johnson, David Duvenaud, Dimitrios Vytiniotis, Alexey Radul, Matthew J. Johnson, Jonathan Ragan-Kelley, and Dougal Maclaurin. 2021. Getting to the Point. Index Sets and Parallelism-Preserving Autodiff for Pointful Array Programming. In *The 25th ACM SIGPLAN International Conference on Functional Programming (ICFP)*. ACM SIGPLAN, ACM. https://doi.org/10.1145/3473593

Clément Pit-Claudel, Peng Wang, Benjamin Delaware, Jason Gross, and Adam Chlipala. 2020. Extensible Extraction of Efficient Imperative Programs with Foreign Functions, Manually Managed Memory, and Proofs. In *IJCAR'20: Proceedings of the 9th International Joint Conference on Automated Reasoning* (Paris, France). https://doi.org/10.1007/978-3-030-51054-1_7

Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2012. Decoupling algorithms from schedules for easy optimization of image processing pipelines. *ACM Trans. Graph.* 31, 4 (2012), 32:1–32:12. https://doi.org/10.1145/2185520.2185528

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines. In *Proc. PLDI*. ACM, Seattle. https://doi.org/10.1145/2491956.2462176

Justin Slepak, Olin Shivers, and Panagiotis Manolios. 2014. An Array-Oriented Language with Static Rank Polymorphism. In *Programming Languages and Systems*, Zhong Shao (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 27–46. https://doi.org/10.1007/978-3-642-54833-8_3

Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure Tensor Program Rewriting via Access Patterns (Representation Pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS 2021)*. Association for Computing Machinery, New York, NY, USA, 21âĂŞ31. https://doi.org/10.1145/3460945.3464953

Michel Steuwer, Chris Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating Performance Portable Code using Rewrite Rules: From High-Level Functional Expressions to High-Performance OpenCL Code. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, Vol. 50. Association for Computing Machinery. https://doi.org/10.1145/2784731.2784754

Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions. arXiv:1802.04730 [cs.PL]

Anand Venkat, Tharindu Rusira, Raj Barik, Mary Hall, and Leonard Truong. 2019. 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. https://doi.org/10.1177/1094342019866247 arXiv:https://doi.org/10.1177/1094342019866247

Qing Yi, Keith Seymour, Haihang You, Richard W. Vuduc, and Daniel J. Quinlan. 2007. POET: Parameterized Optimizations for Empirical Tuning. In *21st International Parallel and Distributed Processing Symposium (IPDPS 2007)* (Rome, Italy). IEEE, Piscataway, NJ, USA, 1–8. https://doi.org/10.1109/IPDPS.2007.370637

Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. 2018. GraphIt: a high-performance graph DSL. *PACMPL* 2, OOPSLA (2018), 121:1–121:30. https://doi.org/10.1145/3276491