Parallel Associative Reductions in Halide

Patricia Suriana
Google, USA
psuriana@google.com

Andrew Adams
Google, USA
abadams@google.com

Shoaib Kamil
Adobe, USA
kamil@adobe.com

Abstract

Halide is a domain-specific language for fast image processing that separates pipelines into the *algorithm*, which defines *what* values are computed, and the *schedule*, which defines *how* they are computed. Changes to the schedule are guaranteed to not change the results. While Halide supports parallelizing and vectorizing naturally data-parallel operations, it does not support the same scheduling for reductions. Instead, the programmer must create data parallelism by manually factoring reductions into multiple stages. This manipulation of the *algorithm* can introduce bugs, impairs readability and portability, and makes it impossible for automatic scheduling methods to parallelize reductions.

We describe a new Halide scheduling primitive `rfactor` which moves this factoring transformation into the *schedule*, as well as a novel synthesis-based technique that takes serial Halide reductions and synthesizes an equivalent binary associative reduction operator and its identity. This enables us to automatically replace the original pipeline stage with a pair of stages which first compute partial results over slices of the reduction domain, and then combine them. Our technique permits parallelization and vectorization of Halide algorithms which previously required manipulating both the algorithm and schedule.

1. Introduction

Halide [17] is a domain-specific language designed for fast image processing and computational photography. Halide decouples the *algorithm*, which defines *what* values are computed, from the *schedule*, which defines *how* values are computed. Halide guarantees consistency – an algorithm produces the same results no matter the schedule. Programmers are thus free to explore the space of schedules without introducing correctness bugs, and can vary the schedule per architecture without producing different results on different platforms.

Data-parallel operations, such as resizing an image, can be easily parallelized or vectorized in Halide. However, Halide does not support the same scheduling manipulations on reductions. To parallelize or vectorize a reduction, the programmer has to manually factor the reduction into multiple stages to expose new data parallelism. For example, instead of computing the histogram of an entire image, one might instead write an algorithm that computes the histogram of each row, and then adds those partial histograms. This need to rewrite the algorithm violates the core tenet of Halide: The algorithm should only specify *what* is computed. It is the role of the schedule to specify *how*. This manipulation of the *algorithm* to parallelize reductions is bug-prone and hampers readability and portability. It is a language wart.

```
1 Func out;
2 out() = 0;
3 RDom r(0, input.width());
4 out() = out() + input(r.x);
```

Listing 1. Halide sum reduction over a one-dimensional vector.

In this work, we present a new Halide scheduling primitive called `rfactor`, which moves this factoring of a reduction into the *schedule*, while maintaining Halide's consistency guarantees. `rfactor` takes a Halide serial reduction (expressed with an unstructured Halide *update* definition), and synthesizes the equivalent binary associative reduction operator and its identity. In some cases this synthesis problem is trivial. For example, given Halide code that sums a one-dimensional vector (Listing 1), it is straightforward to deduce that the binary operator involved is addition, and its identity is zero. In other cases this synthesis problem is more challenging. Listing 2 shows Halide code that finds the complex number with the greatest magnitude and its location within a two-dimensional array. It is not obvious what the equivalent associative binary operator is for this algorithm.

During the compilation process, `rfactor` splits the original serial reduction into a pair of stages: The *intermediate* stage computes partial results over slices of the domain of the reduction, and the *merge* stage combines those partial results. The intermediate stage is now data parallel over the slices, which means that it can now be vectorized or parallelized using Halide's existing scheduling primitives.

Combined with other Halide scheduling primitives, such as `split`, `rfactor` allows Halide to represent a broad class of schedules for parallel and vectorized reductions. For example, `rfactor` can express several divide-and-conquer strategies for parallelizing and vectorizing the summation of a one-dimensional array (see Figure 1).

```
Func out;
out() = {0, 0, 0, 0};
RDom r(0, input.width(), 0, input.height());
Expr real = input(r.x, r.y)[0];
Expr imag = input(r.x, r.y)[1];
Expr mag = real * real + imag * imag;
Expr best_mag = out()[0] * out()[0] +
                out()[1] * out()[1];
Expr c = mag > best_mag;
out() = {select(c, real, out()[0]),
         select(c, imag, out()[1]),
         select(c, r.x, out()[2]),
         select(c, r.y, out()[3])};
```

Listing 2. Halide reduction which finds the complex number with the greatest magnitude and its location in a two-dimensional array.

`rfactor` further separates the *algorithm* from its *schedule* by making it possible to factor reductions using the schedule alone. In addition to the readability and portability benefits, this means that tools that automatically generate schedules [15, 17] are now capable of parallelizing reductions, which was previously a task outside of their purview.

Our work makes the following contributions:

- We introduce a new Halide scheduling primitive `rfactor` which factors a Halide reduction into a pair of reductions: an *intermediate* stage that computes partial results over slices of the reduction domain, and a *merge* stage that combines those partial results.
- We describe a method for automatically discovering an equivalent associative binary reduction operator and its identity from a serial reduction expressed as an imperative Halide *update*.
- We implement a new stage in the Halide compiler that matches arbitrary reductions with a set of 17,905 fragments that are pre-generated using our synthesis method, and show that this enables the compiler to effectively transform a large set of reductions into their parallel equivalents.

The paper is structured as follows. Section 2 provides background on Halide and a discussion of related work. Section 3 presents the `rfactor` scheduling primitive and how it transforms Halide programs. Section 4 describes the associative binary reduction operator synthesis technique. Section 5 demonstrates that this technique does indeed produce the expected performance gains from vectorization and parallelization, and Section 6 describes limitations and summarizes this work.

2. Background & Related Work

Programmers define an *algorithm* in Halide using a graph of Halide *functions*, which each consist of a sequence of stages. These stages are the unit on which scheduling occurs. Each

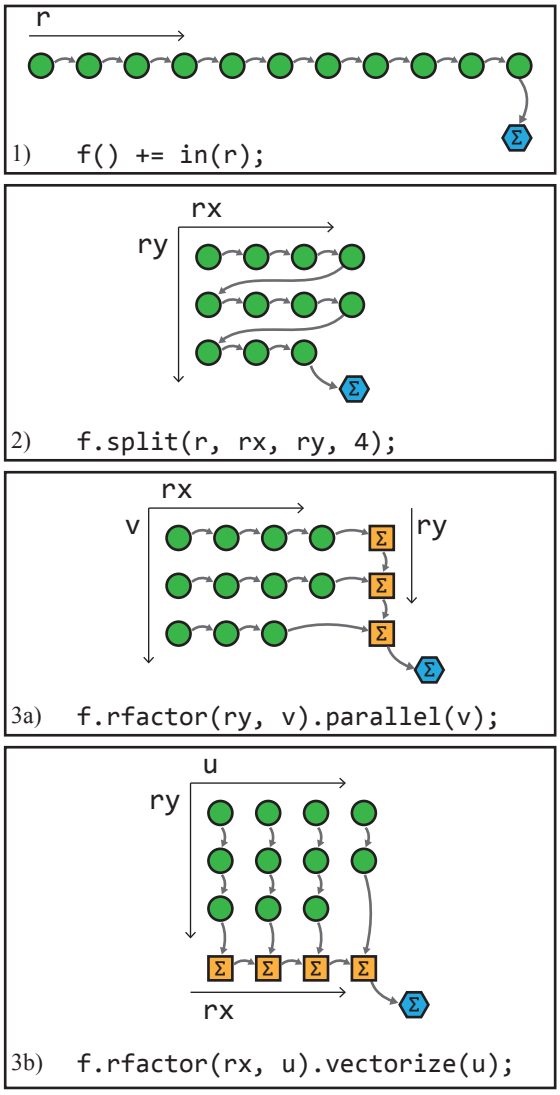


Figure 1. `split` followed by `rfactor` used to vectorize or parallelize a one-dimensional summation. 1) A serial summation of 11 elements over a one-dimensional domain `r`. 2) The `split` scheduling directive reshapes the domain into a two-dimensional reduction. The reduction is still serial. 3a) The `rfactor` directive creates and returns a new (orange square) *intermediate* stage which computes partial results along each row. The intermediate stage is now data parallel over its outer loop, which is indexed with the new pure variable `v`. The merge stage (blue hexagon) retains only the serial loop over the reduction variable `ry`. We have *reassociated* the summation. 3b) Alternatively, one can use `rfactor` to make the *inner* loop data parallel. This can be used to vectorize reductions. The intermediate stage computes sums of whole vectors, data parallel across the vector lanes, and then the merge stage sums the lanes of the vector result. The strategies in 3a and 3b can be combined to both vectorize *and* parallelize a summation. Note that 3b requires the reduction to be commutative as it changes the order of computation.

```
// First stage
for y:
 for x:
   blur(x, y) = 0
// Second "update" stage
for y:
 for x:
  for ry:
   for rx:
    blur(x, y) += k(rx, ry) * in(x-rx, y-ry);
```

Listing 3. Pseudocode for convolution. This algorithm reduces over `rx` and `ry` and is data-parallel over `x` and `y`. In the Halide source, `rx` and `ry` would be `RVars` in a two-dimensional `RDom`. `x` and `y` would be `Vars`.

stage represents a perfectly-nested loop in which a single value of the *function* is computed and stored in the innermost loop per iteration. Stages after the first are called *update* stages, and are allowed to recursively refer to the function. Some of the loops are data parallel and are constrained to be race-condition free by syntactic restrictions. These data-parallel loops iterate over variables called `Vars`. The bounds of these loops are inferred by Halide using interval arithmetic. Other loops may have user-specified bounds and a user-specified nesting order, and fewer syntactic restrictions on their use. These loop variables are known as `RVars` (for reduction variables), which together define a reduction domain or `RDom`. `RVars` are used to express reductions, scattering, scans, etc.

Each of these loop types, defined by `Vars` and `RVars`, can be manipulated in various ways using Halide's scheduling primitives: they can be tiled, unrolled, mutually interchanged, etc., provided that the nesting order of `RVars` is respected. Halide's scheduling language also specifies how the computation of producer functions are interleaved with their consumers, in order to optimize for locality. This is done by selecting a loop level in the loop nest of the consumer at which to insert the loop nest of the producer. It is expressed with the scheduling directive `compute_at`. `compute_root` specifies that the producer should be computer entirely outside of the loop nest of the consumer.

While `Vars` are safe to parallelize or vectorize by construction – `Vars` represents the naturally data-parallel axes of an *algorithm* – `RVars` can be parallelized or vectorized if and only if Halide can prove that no race condition exists (we call such `RVars` *pure*). This makes parallelizing or vectorizing stages that use only `RVars` difficult. For example, consider the two-dimensional convolution shown in Listing 3, which is parallelizable across `Vars` x and y. Computing the histogram of an image (see Listing 4) is harder to parallelize since its update stage only involves `RVars`.

Although much prior work has explored automatic generation of parallel associative reductions from a serial reduction, most work requires that an explicit associative binary reduction operator is given. One widely-deployed example is OpenMP [2] which has a first-class parallel reduction construct that takes one of some number of primitive reduction operators. Cilk [1] additionally supports user-specified reduction operators. This approach is not applicable to Halide. Reductions in Halide are implemented through implicit serial loops over `RVars`. The reduction operator is never explicitly stated. For Halide to support parallel reductions, it needs to be able to deduce an equivalent binary associative reduction operator and its identity from a serial reduction expressed as an imperative Halide *update*.

```
// First stage
for x:
 hist(x) = 0
// Update stage
for ry:
 for rx:
  hist(input(rx, ry)) += 1
```

Listing 4. Computing the histogram of an image is hard to parallelize in Halide, since its update stage would be expressed with serial `RVars`.

LLVM [11] can automatically recognize a small set of associative reductions for the purposes of auto-vectorization. However, it has the usual problems associated with auto-vectorization – small changes to the program can cause auto-vectorization to fail for reasons mysterious to the programmer, and only simple loops reliably auto-vectorize. Halide's philosophy is one of explicit control. Nevertheless, Halide compiles to LLVM bitcode, and so for simple reductions we will benefit from this auto-vectorization even if the programmer does not employ `rfactor`.

Prior work by Morita et al. [14] introduced automatic generation of divide-and-conquer parallel programs using a framework based on the third homomorphism theorem and derivation of weak-right inverse. However, it requires that programmers specify the leftwards and rightwards forms of the sequential function. Teo et al. [22] proposed a method to synthesize parallel divide-and-conquer programs from a recurrence function (which is similar in form to a Halide serial reduction) through induction. They first derive two equivalent pre-parallel forms of the recurrence function by applying some generalization rules and deduce the intermediate and merge reduction functions through induction on those two pre-parallel forms. Although it can be applied to solve some complex recurrences, such as the product of a list of complex numbers, the technique is slow, and is unable to deal with reductions like `argmin`, which require non-trivial re-ordering of the chain of conditionals during the induction steps.

Recent work has applied program synthesis, which automatically discovers executable code based on user intent derived from examples or other constraints, to generate parallel programs. Smith et al. [20] used program synthesis to automatically generate MapReduce-style distributed

programs from input-output examples. SKETCH [21] and ROSETTE [23] are two solver-aided programming languages with support for program synthesis. MSL [24] is a synthesis-based language for distributed implementations that can derive many details of the distributed implementation from serial specifications. We tried SKETCH and ROSETTE and found them too slow to apply directly at compile time.

Superoptimization [6, 13] searches for the shortest or most optimized way to compute a branch-free sequence of instructions by exhaustively searching over a space of possible programs. These rewrites can then be turned into peephole optimizations in compilers. More recent work has used stochastic search [16, 19] and program synthesis [12] to find replacements for larger sequences of instructions. In this work, we use a combination of enumeration and synthesis; in addition, though our domain is more restricted, we search for larger replacements than most superoptimizers.

CHiLL [7] and URUK [5] allow users to apply a series of high-level transformations to Fortran and C code, freeing users from needing to hand-rewrite code to implement complicated optimizations. These code transformation systems use the polyhedral framework [4, 10] to represent loops and transformations, but do not support reductions. More recent work [18] adds new language constructs that allow users to express arbitrary reductions in the polyhedral model, enabling transformations to optimize such reductions.

3. The `rfactor` Transformation

Serial reductions in Halide (e.g. summation over an array, histogram, etc.) are implemented using `RVars` or `RDoms`. An `RVar` is an implicit serial loop, and an `RDom` is an ordered list of `RVars` specifying a serial loop nest. Since `RVars` are not trivially parallelizable or vectorizable, a programmer must manually *factor* a reduction into an *intermediate* function that performs reduction over distinct slices of the domain, and a *merge* function that combines those partial results.

To further complicate matters, it is hard to infer what binary reduction operator is equivalent to a Halide update definition, and even then, many binary operators are not obviously associative (e.g. $x + y + 7xy$ is in fact associative). We will defer these issues to Section 4, and for now assume that given a Halide update definition we can deduce the equivalent associative binary operator and its identity. Note that some transformations (e.g. factoring the inner loop of a reduction as in Figure 1-3b) require the binary operators to be commutative in addition to being associative as they may change the order of computation.

To remove the burden of factoring a reduction from the programmer, we introduce a new scheduling primitive called `rfactor`. This splits a reduction into a pair of reductions, which we will call the *intermediate* function and the *merge* function. `rfactor` takes a list of (`RVar`, `Var`) pairs. The *intermediate* reduces only over the `RVars` *not* mentioned in the list, and gains the associated `Vars` as new pure data-parallel dimensions. The *merge* then reduces over these additional dimensions using `RVars` that *are* mentioned the list. The intermediate is thus *higher*-dimensional than the original, but both the intermediate and merge do *lower*-dimensional reductions than the original. We will specify the transformation in more detail after looking at several examples.

As a first example, consider computing the histogram of an image in Halide:

```
// Algorithm
Func hist;
Var i;
hist(i) = 0;
RDom r(0, input.width(), 0, input.height());
hist(input(r.x, r.y)) += 1;

// Schedule
hist.compute_root();
```

The `RDom` defines an implicit loop nest over `r.x` and `r.y`. Halide will not permit either of these loops to be parallelized, as that would introduce a race condition on the += operation. Without the `rfactor` transformation, a user would need to rewrite the algorithm to manually factor the histogram:

```
// Algorithm
Func intm;
Var i, y;
intm(i, y) = 0;
RDom rx(0, input.width());
intm(input(rx, y), y) += 1;

Func hist;
hist(i) = 0;
RDom ry(0, input.height());
hist(i) += intm(i, ry);

// Schedule
intm.compute_root().update().parallel(y);
hist.compute_root().update().vectorize(i, 4);
```

Above, the programmer introduced an intermediate function `intm` that computes the histogram over each row of the input. This intermediate function is data-parallel over `y`, and so it can be parallelized. The original function `hist` now merely sums these partial histograms; since `hist` is data-parallel over histogram buckets, the programmer has vectorized it.

Using `rfactor`, the programmer can produce the same machine code as the manually-transformed version, using the simpler algorithm in the original `hist` implementation. While the schedule is more complex, recall that it is only the five lines of the algorithm that determine correctness. The programmer can freely transform the code to exploit parallelism without risking introducing a correctness bug:

```
// Algorithm
Func hist;
Var i;
hist(i) = 0;
RDom r(0, input.width(), 0, input.height());
hist(input(r.x, r.y)) += 1;

// Schedule
Var y;
hist.compute_root()
Func intm = hist.update().rfactor(r.y, y);
intm.compute_root().update().parallel(y);
hist.update().vectorize(i, 4);
```

Reduction domains need not be rectangular. Consider the function below, which computes the maximum over a circular domain using `RDom::where` to restrict the reduction domain to the points that lie within a circle of radius 10.

```
// Algorithm
Func max_val;
max_val() = 0;
RDom r(0, input.width(), 0, input.height());
r.where(r.x*r.x + r.y*r.y <= 100);
max_val() = max(max_val(), input(r.x, r.y));

// Schedule
max_val.compute_root();
```

In this case, manually factoring the reduction requires also manipulating the predicate associated with the `RDom`. The identity for `max` is the minimum value of the type in question, so the newly-factored algorithm becomes:

```
// Algorithm
Func intm;
Var y;
intm(y) = input.type().min();
RDom rx(0, input.width());
rx.where(rx*rx + y*y <= 100);
intm(y) = max(intm(y), input(rx, y));

Func max_val;
max_val() = 0;
RDom ry(0, input.height());
max_val() = max(max_val(), intm(ry));

// Schedule
intm.compute_root().update().parallel(y);
max_val.compute_root();
```

Using `rfactor` in the schedule, the programmer can produce the same machine code from the original algorithm:

```
// Algorithm
Func max_val;
max_val() = 0;
RDom r(0, input.width(), 0, input.height());
r.where(r.x*r.x + r.y*r.y <= 100);
max_val() = max(max_val(), input(r.x, r.y));

// Schedule
Var y;
max_val.compute_root();
Func intm = max_val.update().rfactor(r.y, y);
intm.compute_root().update().parallel(y);
```

In both of these examples we factored a two-dimensional reduction into two one-dimensional reductions. In general, one can simultaneously factor out any subset of the dimensions of an N-dimensional reduction. Recall that low-dimensional reductions can be reshaped into higher-dimensional ones using `split`, as in Figure 1.

With these concrete examples in mind, we now specify the general form of the transformation. Given a list of (`RVar`, `Var`) pairs, `rfactor` does the following:

1. The associative binary operator equivalent to the reduction is synthesized (see Section 4).
2. An *intermediate* function is created (called `intm` in the examples above). It is given a pure definition equal to the identity of the associative operator. The function has the same pure `Vars` as the original, plus the `Vars` specified in the `rfactor` call. The *intermediate* is thus a higher-dimensional function than the original.
3. The *intermediate* is given an *update* definition which is a copy of the update definition to which `rfactor` was applied. The reduction domain for this definition is a copy of the original reduction domain, minus the `RVars` being factored out. It is thus a lower-dimensional reduction than the original. In all expressions that comprise this update definition, each `RVar` being factored out is replaced with its associated `Var`, and those `Vars` also appear as pure variables on the left-hand-side of the update definition.
4. The original update definition being factored is deleted, and a new update definition is injected in its place which reduces over the *intermediate* using the associative operator. The domain of this reduction is the set of `RVars` which were factored out of the intermediate. The update operates element-wise along any pure dimensions of the merge stage (which may create yet more data parallelism, as in the histogram example). Other stages of the original function (e.g. its pure definition) are left unchanged.

Note that this transformation requires that the associative operator has an identity. This is not true for all associative operators, such as $2xy$, where $x, y \in \mathbb{Z}$.

4. Synthesizing Associative Binary Operators

In the previous section, we described how `rfactor` transforms a Halide reduction to expose new data parallelism. Doing so requires synthesizing the associative binary operator equivalent to the Halide *update* definition being factored. In this section we describe how we do this synthesis.

In some cases generating the equivalent associative operator from an update definition is trivial (e.g. summation). In other cases it is not, especially when the function reduces onto multiple values (see the examples in Figure 4). The *inverse* problem is easier: Given an associative binary operator, it is straightforward to generate the Halide update definition that implements reduction by it. Therefore, if there were a *small*

finite set of associative binary operators, we could simply attempt to match the update definition being factored against each of them in turn. Halide already includes facilities for doing regex-like matching of expressions against patterns with wildcards. Unfortunately, there are more meaningful associative binary operators than could reasonably be thought of ahead of time by a compiler author.

An alternative approach is to use program synthesis techniques [21, 23] to synthesize the corresponding associative reduction at compile-time when the call to `rfactor` is made. This is intractably slow, and can increase compile times of Halide pipelines from seconds to hours.

We use a hybrid of the two approaches, amplified with a strategy for decomposing each synthesis problem into several simpler problems. The overall strategy is shown in Figure 2. Offline, we generate a large finite table of one- and two-dimensional *elementary* associative binary operators and their identities. These are akin to primes – they are the associative operators which cannot be decomposed into a combination of simpler associative binary operators. The table also records whether each operator is commutative. At compile-time, we decompose the given Halide update definition into a composition of simpler, lower-dimensional definitions in the same way, then search the table for the elementary operator corresponding to each. If consistent matches are found, we reassemble the results into a composite associative operator equivalent to the original update definition. In Section 4.1 we describe how we generate the table, and in Section 4.2 we describe the decomposition procedure.

4.1 Generating Elementary Operators

We begin with an enumeration of all one- and two-dimensional tuples of expression trees in two tuple inputs x and y. The operators used to form the inner nodes of the trees are Halide's IR nodes (`*`, `+`, `-`, `min`, `max`, `select`, `<`, etc). We can reject some classes of uninteresting expressions by excluding them from the enumeration altogether.

1. We only generate trees which use both x and y.
2. During matching we canonicalize both the pattern and the input expression using the Halide simplifier and solver; thus, we can make the enumeration more tractable by only generating trees that are already in canonical form. The canonicalization strategy moves all instances of a variable in an expression tree as far to the left and as far up the tree as possible. We canonicalize expressions to the following form: wherever possible, x_i is to the left of x_{i+1} and any y_j is to the left of y_{j+1}. Constants are always on the right.
3. We generate trees using a single generic constant k, rather than generating trees containing all possible constants as leaves.
4. We do not generate trees that would be trivially simplified. For example, we do not generate subexpressions like $max(x_0, x_0)$.

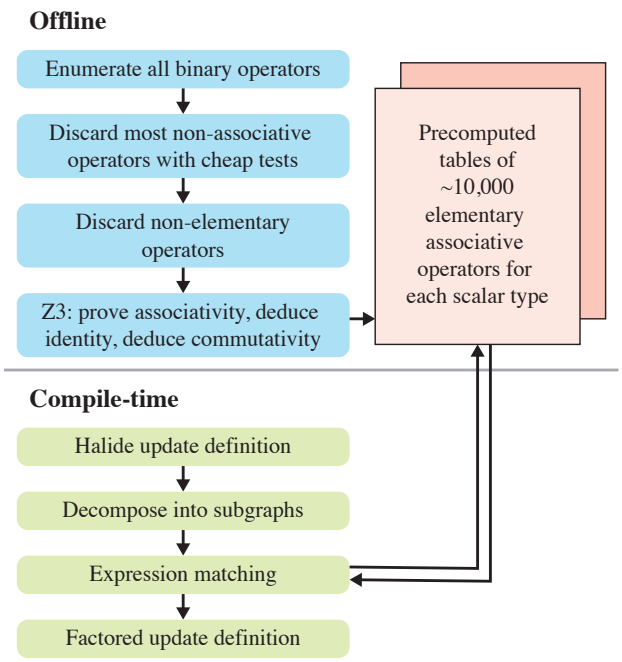


Figure 2. To factor a reduction written as a Halide update definition, we must first synthesize the equivalent associative binary operator. We generate a large table of elementary associative operators offline by enumerating all non-trivial expression trees and filtering out the ones that are not associative operators. At compile-time, we then decompose the given update definition into simpler definitions and match each against the table. Combining the results gives us the equivalent associative binary operator, which we can use to generate the factored form of the reduction.

After generating a large set of candidate expressions in this manner, we then subject the expressions to a battery of tests so that only elementary associative operators remain. The tests are arranged in increasing order of expense, so that we can cheaply reject most expressions.

1. For an operator f, we construct the expressions $f(f(x, y), z)$ and $f(x, f(y, z))$, and substitute in 100 randomly-selected values for x, y, z, k. If the two expressions don't evaluate to the same thing, the expression is not associative and can be rejected.
2. We then reject operators which can be decomposed using the decomposition procedure in Subsection 4.2.
3. Finally, to *prove* that the expression is associative, we use the Z3 [3] SMT solver to verify that $\forall x, y, z, k \;\; f(f(x, y), z) = f(x, f(y, z))$, where k is the constant contained within the function f. If this proof succeeds, we then ask Z3 to solve $\forall x, k \; f(id, x) = x$ to synthesize the identity id and to decide its commutativity. This commutativity flag is used to determine the validity of calling `rfactor` on the inner loop dimensions of an associative reduction.

Add	Mul	Min	Max
1: x+y	1: x*y	1: min(x,y)	1: max(x,y)
....	....	0: min(max(x,k),y)	0: max(min(x,k),y)
....	....	0: min(max(y,k),x)	0: max(min(y,k),x)
....	....	0: min(max(k-x,y),x)	0: max(min(k-x,y),x)
....	....	....	....

Figure 3. The first 10 elementary associative operators for 32-bit signed integers organized into subtables based on the root IR node, with simpler expressions located at the top. The boolean flag on the left indicates the commutativity of the expression.

In the table of expressions that this produces, x represents the partial result being reduced onto, and y is a wildcard which could match any expression. When matching, x must also appear on the left-hand-side of the update definition. y may depend on the reduction domain coordinates, but may not depend on the partial results. The constant k, if it exists, may match anything which neither depends on the reduction domain coordinates nor the partial result. For example, the update definition which separately computes the sum-of-squares of the even and odd values of some input `in` could be written as:

```
f(in(r)&1) = f(in(r)&1) + in(r)*in(r)
```

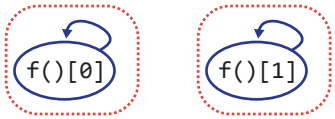
This would match against the pattern $x = x + y$, where x matches `f(in(r)&1)`, and y matches `in(r)*in(r)`.

We terminate the enumeration at trees with 8 leaves for the purposes of the experiments described in this work. Since some operations are associative in certain scalar types and not others, we generate different tables for each of Halide's scalar types. This takes 1.5 days and generates around 9,000 elementary operators to match against per type. Many of the operators are simple operators written in a complex way: as we wish to catch all ways in which a programmer might write a reduction, we make no attempt to exclude these from the table. For faster retrieval, the table of each Halide's scalar type is split into subtables based on the root IR node (e.g. `Add`, `Mul`, etc.) of the expression. Within each subtable, the operators are ordered based on the total number of leaves (simple expressions are located at the top of the table since they are more likely to be encountered in practice). The operators we use are `Add`, `Sub`, `Mul`, `Min`, `Max`, comparisons, and `Select` (Halide's if-then-else construct). We then linearly scan the sub-table searching for a matching expression. The first 10 elementary associative operators for signed 32-bit integers are shown in Listing 3.

As we only consider one- and two-dimensional expressions, there are meaningful primitive operators that we never discover. For example, we never generate quaternion multiplication, since it is an elementary associative operator in 4 tuple elements, where every expression tree has 8 leaf nodes. We must add any important higher-dimensional primitive operators to our table manually.

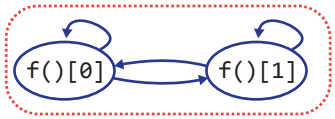
Sum of complex numbers

```
f() = {f()[0] + real(r), f()[1] + imag(r)}
```



Product of complex numbers

```
f() = {f()[0]*real(r) - f()[1]*imag(r),
       f()[1]*real(r) + f()[0]*imag(r)}
```



Two-dimensional argmin

```
f() = {min(f()[0], in(r.x, r.y)),
       select(f()[0] < in(r.x, r.y), f()[1], r.x),
       select(f()[0] < in(r.x, r.y), f()[2], r.y)}
```

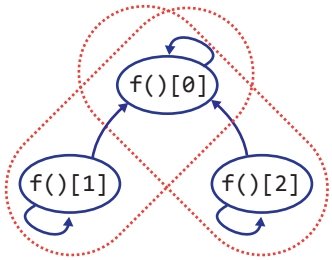


Figure 4. Dependency graphs of various multi-valued Halide *update* definitions. Their subgraph decompositions are shown in red dotted circle. To find the associative operator equivalent to a complex Halide update definition, we first decompose it into subgraphs, then search for each subgraph in our precomputed table of elementary operators, then recompose the results into a single multi-valued associative operator.

4.2 Subgraph Decomposition

To describe how we decompose reductions into elementary operators, we must first discuss how one might *compose* elementary reductions. The simplest way to compose two reductions into a higher-dimensional reduction is to compute them at the same time, independently. This means that reductions that are a concatenation of smaller independent associative reductions are also associative. For example, the tuple computation:

```
f() = {f()[0] + in(r), f()[1] * in(r)}
```

is associative, because it is a composition of the following two associative reductions:

```
f0() = f0() + in(r)
f1() = f1() * in(r)
```

Secondly, if we have an associative operator in which two tuple elements compute the same value, we can deduplicate it, reducing the dimensionality by one. The result is still

associative. Therefore, we can prove a reduction is associative by duplicating one of the elements to break a dependency and then applying the rule above. Consider the case of two-dimensional `argmin`:

```
f() = {
  min(f()[0], in(r.x, r.y)),
  select(f()[0] < in(r.x, r.y), f()[1], r.x),
  select(f()[0] < in(r.x, r.y), f()[2], r.y)}
```

The three tuple elements are the minimum value, and its x and y coordinate. All three tuple elements depend on the minimum value `f()[0]`, so we cannot decompose this into independent reductions immediately. Let us duplicate the first tuple element to break the dependency:

```
f() = {
  min(f()[0], in(r.x, r.y)),
  select(f()[0] < in(r.x, r.y), f()[1], r.x),
  min(f()[2], in(r.x, r.y)),
  select(f()[2] < in(r.x, r.y), f()[3], r.y)}
```

There are now no dependencies between the first two elements and the last two. This is a simple concatenation of two single-variable `argmin` operations. Single-variable `argmin` uses two tuple elements, and is present in our table of primitive operators, so we recognize two-dimensional `argmin` as associative via its decomposition into two one-dimensional `argmin` reductions.

In general we consider the directed graph of dependencies between tuple elements. There is a vertex per tuple element, and an edge from vertex i to vertex j whenever the definition of tuple element i refers to tuple element j. If we repeatedly duplicate vertices (tuple elements) to break dependencies, in the limit the graph has one connected component per original tuple element, and that component is the subgraph containing the vertices reachable from that tuple element. If each such subgraph is an associative reduction, then the original reduction is associative. See Figure 4 for several examples.

After finding the associative operator equivalent to each subgraph separately, we need to combine the results into a single multi-valued associative operator equivalent to the entire update definition. If all the subgraphs are associative and have identities, we need to ensure that for each tuple element appearing in multiple subgraphs, the binary associative operators deduced via each subgraph are all the same in that tuple element. If they are consistent, we have succeeded in finding an equivalent multi-valued associative operator for the update definition.

In some cases, this procedure over-partitions the graph. We are searching for an associative operator in x and y. Only the x is apparent from the Halide update definition – it is the term that also appears on the left-hand-side. If we decompose based on cross-dependencies within x alone we will miss dependencies between tuple elements that exist only in y. Consider 2x2 matrix multiplication written as a four-dimensional reduction:

```
f() = {f()[0] * in(r)[0]) + f()[1] * in(r)[2]),
       f()[0] * in(r)[1]) + f()[1] * in(r)[3]),
       f()[2] * in(r)[0]) + f()[3] * in(r)[2]),
       f()[2] * in(r)[1]) + f()[3] * in(r)[3])}
```

Decomposition based on x alone (i.e. `f()[i]`) results in 2 subgraphs: one containing the 1st and 2nd tuple elements and one containing the 3rd and 4th tuple elements. Including y (i.e. `in(r)[i]`) tells us that the two subgraphs are indeed connected. Therefore, if we fail to find a match for the initial subgraphs (the ones decomposed based on x only), we need to consider other possible grouping of those initial subgraphs. The total number of possible grouping is the Bell number of the initial number of subgraphs. However, we only need to consider grouping expressions which share a common subexpression, and we do not need to consider groups of size greater than the maximum tuple size in our precomputed table. If we do not find a matching associative operator under all possible groupings of the initial subgraphs, we terminate and return an error.

4.3 Algorithm Summary

To summarize, synthesizing an equivalent binary associative operator from a Halide reduction involves the following steps:

1. Starting from the right-hand-side of the update definition, replace all references to the partial results being reduced onto with the symbol x_i where $0 \leq i < n$ and n is the number of components in the reduction (i.e. the tuple size).
2. Canonicalize these expressions. Wherever possible, x_i is to the left of x_{i+1} and constants to the right.
3. Construct the graph G that represents the dependency relationships between these terms.
4. Decompose G into an initial set of connected subgraphs S_0.
5. Pick a grouping S from all possible groupings of S_0 (see Section 4.2). For the first iteration, we pick $S = S_0$ as the grouping.
6. For each subgraph in S, search the appropriate subtable (based on the data type and root IR node) for a matching associative operator via wildcard matching. If any of the subgraphs are not found, return to step 5.
7. Combine the results into a single multi-valued associative operator equivalent to the entire reduction. If the results are consistent (see Section 4.2), we have found an equivalent associative operator; otherwise, return to step 5. If we do not find a matching associative operator after exhausting all possible groupings, we terminate and return an error.

As we show in the next section, this algorithm is fast to execute at compile-time, and successfully finds equivalent binary associative operators for many Halide reductions.

Benchmark	Data Type	Serial (ms)	`rfactor` (ms)	Speed-up
Maximum	int32	5.54	1.22	4.5
2D histogram	int32	8.80	1.71	5.1
4D `argmin`	int8	28.52	1.07	26.6
Complex multiplication	int32	28.53	2.47	11.5
Dot product	float	25.9	2.66†	9.7
Kitchen sink	int32	30.13	1.91	15.7

Table 1. Benchmark results: serial reductions vs. parallel reductions using `rfactor`. † To give the numbers some context, Intel's MKL [9] takes 2.8ms on the dot product task.

5. Evaluation

In this section we discuss the speed-ups one can expect by using `rfactor` to vectorize and parallelize serial reductions. These numbers are unsurprising – they are equivalent to the speed-ups one can attain by manually factoring Halide reductions. We benchmark the feature using a suite of reductions of varying complexity[1]. Some operations, like large matrix multiplication or convolution, reduce along some axes and are data parallel along others. `rfactor` provides little benefit in these cases, as they are already straight-forward to vectorize and parallelize, so we do not include such cases. Our benchmarks are:

- Maximum: The maximum integer in a list.
- Dot product: The dot product of two vectors.
- 4D `argmin`: The coordinates of the minimum value in a four-dimensional volume.
- 2D histogram: A histogram of values present in an 8-bit image.
- Complex multiplication: The product of a list of complex numbers.
- Kitchen sink: An 8-tuple-element reduction that simultaneously computes the sum, product, minimum, maximum, `argmin`, `argmax`, sum-of-squares, and a count of the number of even values in a list of integers. This tests exists to demonstrate we can decompose multi-valued reductions into primitive ones.

All benchmarks run on an inputs of size 2^{24}, which is a typical number of pixels for an input image to a Halide program. The input image data types are specified in Table 1. We run the benchmarks on 8 cores of a Xeon E5-2690. In each case we use the same Halide *algorithm* code, and compare the performance attainable with `rfactor` to the performance attainable without it. For each benchmark, we took the minimum across 10 trials where each trial is the average of 10 iterations. We measure the execution time of the pipeline, not including the compilation time. Table 1 shows the results. Without `rfactor`, each algorithm would require almost twice as much algorithm code to reach the same performance (see Table 2). Most importantly, `rfactor` provides a much less error-prone way of factoring a reduction as more of the logic is in the schedule instead of the algorithm. We also measured the increase in compile times due to the call to `rfactor`, and found it to be consistently under three milliseconds. The time taken to search the table for a matching operator is shown in Table 3. The search is fast because the table is split into subtables by the root IR node type (the largest subtable has $\sim$ 2800 entries), and the most common operations are simple, and so they are close to the top of the tables. For a non-associative operation (which ultimately returns a compile error) `rfactor` must search to the end of the table. This takes around 1.2 ms.

Benchmarks generally fall into two categories. Either we benefit from from vectorization *and* multi-core parallelism, or we benefit from multi-core parallelism alone. Histogram and maximum, fall into the second category. The histogram benchmark cannot be cleanly vectorized, because the bulk of the work involves scattering to data-dependent locations. The maximum benchmark does vectorize cleanly, but underneath Halide LLVM[2] auto-vectorizes the reference code without `rfactor`, so we only see a speed-up from multi-core parallelism. Dot product, 4D `argmin`, complex multiplication, and the kitchen sink test all benefit from parallelism and vectorization. Complex multiplication, dot product, and maximum all hit the memory bandwidth limit, limiting the possible benefit from parallelism.

Note that we did not add any patterns to the tables manually; the generated fragments were sufficient for the benchmarks and the applications in the Halide open source repository. The generated fragments were also sufficient for the reductions present in the HDR+ pipeline [8], which is the largest Halide pipeline of which we are aware. We could, however, manually add operators to the table if necessary in the future (for example for quaternion multiplication).

6. Conclusion

In this paper, we have presented a new Halide scheduling primitive called `rfactor`, which makes it possible to factor reductions into multiple stages using the schedule alone, with all the concomitant correctness guarantees. This permits parallelization and vectorization of Halide algorithms which previously required manipulating both the algorithm and schedule.

Although our framework is able to handle a broad range of associative reductions, there are several limitations:

- Our precomputed table can only recognize reductions decomposable into elementary reductions of a fixed max-

[1] https://github.com/halide/rfactor_benchmarks

[2] Trunk LLVM as of Sept 9, 2016

Benchmark	Serial (lines)	`rfactor` (lines)	Reduction (%)
Maximum	9	5	44.4
2D histogram	6	4	33.3
4D `argmin`	24	13	45.8
Complex multiplication	12	7	41.7
Dot product	9	5	44.4
Kitchen sink	45	17	62.2

Table 2. Using `rfactor` reduces the lines of code in the benchmarks by 45% on average. Only the lines of code required to define the reduction functions and `rfactor` calls are included in the calculation.

Benchmark	Search time (ms)	Total compilation time (ms)
Maximum	0.08	127.5
2D histogram	0.09	220.9
4D `argmin`	0.57	196.2
Complex multiplication	0.21	150.4
Dot product	0.12	131.2
Kitchen sink	0.55	187.1

Table 3. The time taken to search the table to find a matching operator is relatively small with respect to the total compilation time.

imum size. The number of expression trees grows very quickly as a function of the number of leaves and the number of tuple components, so this approach is never going to handle operations like multiplying a long sequence of 4x4 matrices, each expressed as a 16-tuple. We would need more aggressive decomposition tools, or a more directed runtime search over the space of associative expressions (as opposed to our exhaustive offline enumeration of it).

- We can only recognize associative operations that Z3 can prove are associative. This is fortunately a large set. One failing example in this category is summing a sequence of 128-bit integers, where each 128-bit integer is represented as a pair of 64-bit integers, and addition is implemented as elementwise addition plus some logic to handle the carry bit. We plan to explore additional encodings into Z3 to increase the operations it is able to prove associative.
- We only recognize reductions where the first stage in the factorization – the *intermediate* – has the same form as the original update definition. The simplest failing example in this category is repeated subtraction of a list of values from some initial value. It can be manually factored into an *intermediate* that sums slices of the list, and then a *merge* stage that does repeated subtraction, but `rfactor` cannot do this transformation.

Despite these caveats, our technique can handle all of the associative reductions we have seen in the wild, which are mostly summations of one or more tuple components, minima, maxima, and `argmin`-like operations which find some value associated with an extremum.

`rfactor` lifts the burden of factoring a reduction from the programmer. This improves readability, as the *algorithm*, which is entirely responsible for *what* values are computed, is shorter. It also improves portability, as a reduction can be factored in different ways on different platforms, without risking producing different results on each platform. However, we suspect the most important benefit of `rfactor` is that by moving this factoring into the schedule alone, `rfactor` will make it possible for automatic schedule generation tools to parallelize and vectorize reductions.

REFERENCES

[1] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, 1995. ISBN 0-89791-700-6. doi: 10.1145/209936.209958. URL `http://doi.acm.org/10.1145/209936.209958`.

[2] L. Dagum and R. Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, Jan. 1998. ISSN 1070-9924. doi: 10.1109/99.660313. URL `http://dx.doi.org/10.1109/99.660313`.

[3] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, 2008. ISBN 3-540-78799-2, 978-3-540-78799-0. URL `http://dl.acm.org/citation.cfm?id=1792734.1792766`.

[4] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[5] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *International Journal of Parallel Programming*, 34(3):261–317, 2006.

[6] T. Granlund and R. Kenner. Eliminating branches using a superoptimizer and the gnu c compiler. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 341–352, 1992. ISBN 0-89791-475-9. doi: 10.1145/143095.143146. URL `http://doi.acm.org/10.1145/143095.143146`.

[7] M. Hall, J. Chame, C. Chen, J. Shin, G. Rudy, and M. M. Khan. *Loop Transformation Recipes for Code Generation and*

Auto-Tuning, pages 50–64. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010. ISBN 978-3-642-13374-9.

[8] S. Hasinoff, D. Sharlet, R. Geiss, A. Adams, J. T. Barron, F. Kainz, J. Chen, and M. Levoy. Burst photography for high dynamic range and low-light imaging on mobile cameras. *ACM Transactions on Graphics (SIGGRAPH Asia 2016)*, 2016. URL `http://www.hdrplusdata.org/hdrplus.pdf`.

[9] Intel. Mkl. http://software.intel.com/mkl, 2016.

[10] F. Irigoin and R. Triolet. Supernode partitioning. In *Symposium on Principles of Programming Languages (POPL'88)*, pages 319–328, San Diego, CA, January 1988. URL `http://ssh.cri.ensmp.fr/classement/doc/A-179.pdf`.

[11] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, 2004. ISBN 0-7695-2102-9. URL `http://dl.acm.org/citation.cfm?id=977395.977673`.

[12] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 22–32, 2015. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737965. URL `http://doi.acm.org/10.1145/2737924.2737965`.

[13] H. Massalin. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, ASPLOS II, pages 122–126, 1987. ISBN 0-8186-0805-6. doi: 10.1145/36206.36194. URL `http://dx.doi.org/10.1145/36206.36194`.

[14] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 146–155, 2007. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250752. URL `http://doi.acm.org/10.1145/1250734.1250752`.

[15] R. T. Mullapudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016. ISSN 0730-0301. doi: 10.1145/2897824.2925952. URL `http://doi.acm.org/10.1145/2897824.2925952`.

[16] P. M. Phothilimthana, A. Thakur, R. Bodik, and D. Dhurjati. Scaling up superoptimization. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 297–310, 2016. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872387. URL `http://doi.acm.org/10.1145/2872362.2872387`.

[17] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 519–530, 2013. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176. URL `http://doi.acm.org/10.1145/2491956.2462176`.

[18] C. Reddy, M. Kruse, and A. Cohen. Reduction drawing: Language constructs and polyhedral compilation for reductions on gpu. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 87–97, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4121-9. doi: 10.1145/2967938.2967950. URL `http://doi.acm.org/10.1145/2967938.2967950`.

[19] E. Schkufza, R. Sharma, and A. Aiken. Stochastic superoptimization. *SIGARCH Comput. Archit. News*, 41(1):305–316, Mar. 2013. ISSN 0163-5964. doi: 10.1145/2490301.2451150. URL `http://doi.acm.org/10.1145/2490301.2451150`.

[20] C. Smith and A. Albarghouthi. Mapreduce program synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 326–340, 2016. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908102. URL `http://doi.acm.org/10.1145/2908080.2908102`.

[21] A. Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, 2008. AAI3353225.

[22] Y. M. Teo, W.-N. Chin, and S. H. Tan. Deriving efficient parallel programs for complex recurrences. In *Proceedings of the Second International Symposium on Parallel Symbolic Computation*, PASCO '97, pages 101–110, 1997. ISBN 0-89791-951-3. doi: 10.1145/266670.266697. URL `http://doi.acm.org/10.1145/266670.266697`.

[23] E. Torlak and R. Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 135–152, 2013. ISBN 978-1-4503-2472-4. doi: 10.1145/2509578.2509586. URL `http://doi.acm.org/10.1145/2509578.2509586`.

[24] Z. Xu, S. Kamil, and A. Solar-Lezama. Msl: A synthesis enabled language for distributed implementations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '14, pages 311–322, 2014. ISBN 978-1-4799-5500-8. doi: 10.1109/SC.2014.31. URL `http://dx.doi.org/10.1109/SC.2014.31`.