Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs

Thomas Kœhler
University of Glasgow, Scotland, UK
thomas.koehler@thok.eu

Michel Steuwer
University of Edinburgh, Scotland, UK
michel.steuwer@ed.ac.uk

***Abstract*—Halide and many similar projects have demonstrated the great potential of domain specific optimizing compilers. They enable programs to be expressed at a convenient high-level, while generating high-performance code for parallel architectures. As domains of interest expand towards deep learning, probabilistic programming and beyond, it becomes increasingly clear that it is unsustainable to redesign domain specific compilers for each new domain. In addition, the rapid growth of hardware architectures to optimize for poses great challenges for designing these compilers.**

In this paper, we show how to extend a unifying domain-extensible compiler with domain-specific as well as hardware-specific optimizations. The compiler operates on generic patterns that have proven flexible enough to express a wide range of computations. Optimizations are not hard-coded into the compiler but are expressed as user-defined rewrite rules that are composed into strategies controlling the optimization process. Crucially, both computational patterns and optimization strategies are extensible without modifying the core compiler implementation.

We demonstrate that this domain-extensible compiler design is capable of expressing image processing pipelines and well-known image processing optimizations. Our results on four mobile ARM multi-core CPUs, often used for image processing tasks, show that the code generated for the Harris operator outperforms the image processing library OpenCV by up to 16× and achieves performance close to - or even up to 1.4× better than - the state-of-the-art image processing compiler Halide.

***Index Terms*—Code generation, Compilers, Performance, Image Processing, Rise, Elevate**

I. Introduction

Domain specific languages (DSLs) and their compilers promise convenient programming combined with high-performance. This is a proven success for domains ranging from signal and image processing to deep learning and more.

Building domain specific compilers that generate high-performance code has long been recognized as a highly complex task. To address this, domain-extensible compiler projects like Delite [1] and the more recent AnyDSL [2] aim to simplify the development of DSLs. Delite provides a fixed set of generic patterns that is used as the intermediate program representation. Traditional heuristic driven optimization passes are applied to lower the code. AnyDSL uses partial evaluation to blend the lines between the compiler implementation and library code, allowing compiler optimizations to be expressed more easily.

The recent growth of specialized hardware architectures introduces additional challenges for compiler designers. Relying on traditional heuristics - like Delite - or forcing a complete re-engineering of the optimized code - like AnyDSL - are not sufficient solutions for easily targeting new hardware devices. More flexibility is needed to explore different optimization strategies, but also to allow precise control of compiler optimizations in crucial cases.

The domain-specific compiler Halide [3] provides control over optimization decisions by separating a computation specification from its *schedule* that specifies how the computation should be optimized. This fine grained control allows steering the compiler to generate highly efficient code that would have been hard to reach with traditional compiler heuristics. Unfortunately, Halide only exposes a fixed set of built in optimizations via its scheduling API. This severely limits Halide's ability to adapt to new hardware architectures. For a domain-extensible compiler this would also prevent the addition of domain-specific optimizations.

To overcome the challenges of building a domain-extensible compiler, the LIFT project [4, 5] has proposed an extensible high-level intermediate representation (IR) made up of computational patterns. The LIFT compiler then automatically applies semantics-preserving rewrite rules to optimize the high-level program. The pattern-based IR has shown to provide portable performance across different GPU architectures [6] and to be extensible across domains with extensions for stencil computations [7] used in HPC, deep learning, and image processing. Unfortunately, the control of optimizations in LIFT is limited due to its automated search for best optimizations, which is not always desirable as it may result in poor performance or be too time consuming. The LIFT implementation is also missing optimizations that are important for image processing pipelines, leading to poor performance compared to Halide as shown in fig. 1.

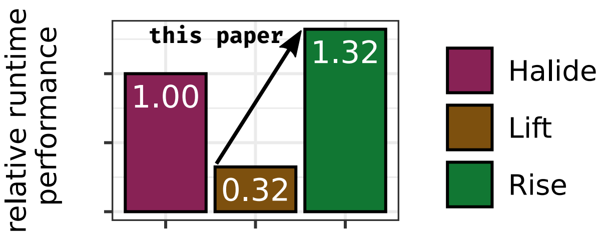


Fig. 1: The existing domain-extensible compiler LIFT performs poorly compared to Halide on image processing pipelines. Extended with additional optimizations, described as compositions of rewrite rules, our domain-extensible compiler for RISE outperforms Halide by 1.3× on Cortex A53.

To overcome these shortcomings we build upon recent work [8] that complements a LIFT-like high-level IR (called RISE) with a language for defining optimization strategies (called ELEVATE) outside of the compiler in an extensible and composable way. ELEVATE allows describing complex compiler optimizations such as tiling, as well as complete program schedules, as compositions of rewrite rules.

In this paper, we demonstrate the advantages of domain-extensible compilers through a case-study where we optimize a standard image processing pipeline: the Harris operator. Without needing to re-engineer a domain-specific optimizing compiler, we outperform Halide on mobile ARM CPUs (see fig. 1) by expressing optimizations outside of the compiler as compositions of rewrite rules.

To summarize, this paper makes the following contributions:

- We demonstrate that the generic patterns of RISE are capable of representing typical image processing pipelines such as the Harris operator (section III);
- We show how to encode well-known image processing optimizations as compositions of rewrite rules in ELEVATE, the language used to describe optimization strategies (section IV);
- We experimentally evaluate our approach, demonstrating that our domain-extensible compiler is capable of generating code that outperforms OpenCV by up to 16× and is competitive to - or up to 1.4× better than - the state-of-the-art image processing compiler Halide (section V).

II. A DOMAIN-EXTENSIBLE COMPILER DESIGN

To address the lack of flexibility of domain-specific compilers, and to reuse compiler infrastructure across DSLs, we advocate for a compiler design that does not rely on a fixed set of computational abstractions and optimizations. Instead, we aim to support a growing set of abstractions and optimizations; enabling to continuously target new hardware architectures and new application domains. This is why we follow ideas from LIFT [4] and more recently RISE and ELEVATE [8]. We combine a high-level pattern-based IR with optimizations that are expressed as compositions of rewrite rules. This approach has been successfully applied to linear algebra and extended to express individual stencil computations [7]. In this paper, we look at more complex image processing pipelines.

Fig. 2 illustrates the overall design. **High-level programs** focus on the description of what is computed. They are written in RISE, a LIFT-like functional IR, using an extensible and reusable set of *patterns*. **Optimization strategies** describe how to optimize the computation. They are written in the ELEVATE language, and orchestrate an extensible and reusable set of *semantics-preserving rewrite rules*. All optimization decisions are explicitly encoded by applying rewrite rules on the high-level RISE program, which leads to a low-level program describing how the computation is performed. From there, code is generated using a formal translation derived from [9] to translate low-level functional RISE programs to imperative code such as C or OpenCL.

As pictured on the right of fig. 2, this compiler design facilitates extension. Extending the compiler for a specific domain involves defining new macros, patterns, strategies and rewrite rules. Macros are used to build abstractions that expand to generic patterns, such as `stencil2d`. Algorithmic patterns are introduced, such as `slide` which creates a sliding window; enabling stencil computations. Optimization strategies are introduced, for example to `separateConvolutions` using an underlying rewrite rule encoding the separability of a convolution kernel (`separateConvKernel`). Extending the compiler for a specific target uses the same mechanisms. Low-level implementation patterns are introduced, such as `mapGlobal` that performs the high-level `map` pattern in parallel across all global threads. Each low-level pattern comes with a small code snippet extending the code generator to explain how low-level code (e.g. OpenCL code) is generated. Optimization strategies are introduced, for example to introduce `parallel`ism or the use of `circularBuffers` across pipeline `Stages`.

Although extensions are critical to achieve high performance, we strive to minimize extensions and to maximize generic infrastructure reuse to simplify compiler development. Most of the patterns and rewrite rules used this paper are generic or widely reusable.

A. Optimizing High-Level Programs via Rewriting

Our domain-extensible compiler takes a high-level RISE program and an ELEVATE optimization strategy as input (see top-left of fig. 2). This high-level program describes *what* to compute, rather than *how* to compute. For example, the dot product is represented as the high-level RISE program `dot`:

```
def dot(a, b) = zip(a, b) ▷ map(×) ▷ reduce(+, 0)
```

The `zip(a,b)` primitive combines two vectors `a` and `b` whose elements are multiplied pairwise using `map(×)` before they are summed using `reduce(+, 0)`. The triangle symbol (▷) indicates the sequencing of operations, `x ▷ y` reads: do `x`, then `y`.

This high-level program does not encode how it is executed, for example we could choose to parallelize the `map` primitive, store the intermediate result, and then perform a sequential `reduce`. Alternatively, we could choose to avoid storing an intermediate result by fusing the `map` and `reduce` patterns and perform a single sequential reduction. Obviously, many more options are possible, and such choices are encoded explicitly as *optimization strategies* in ELEVATE.

For the fused version avoiding the intermediate results, we write a rewrite rule. This rule states that mapping a function `f` over an array before reducing the array is equivalent to reducing the array while applying `f` on the go. Note that the reduction must be performed sequentially because the reduction operator is not commutative anymore.

```
rule reduceMapFusion = map(f) ▷ reduce(g, init)
    ↦ reduceSeq(fun (acc, x). g(acc, f(x)), init)
```

Using this rewrite rule we define an optimization strategy that explains how the rule is applied to the program:

```
strategy lowerDot = applyOnce(reduceMapFusion)
```

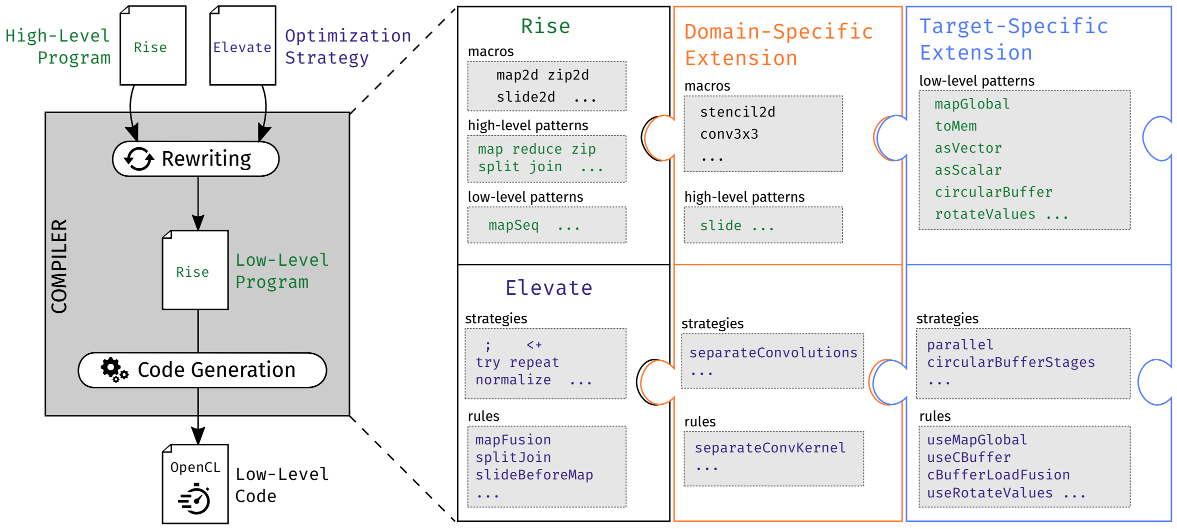


Fig. 2: A Domain-Extensible Compiler Design: Computations are expressed as compositions of extensible patterns in the RISE IR; Optimizations are expressed as compositions of extensible rewrite rules in the ELEVATE strategy language.

This strategy performs the rewrite by applying it to **map**(×) ▷ **reduce**(+,0) in the high-level dot product program, replacing it with the right hand side of the rewrite and producing the following low-level program:

```
  def dotSeq(a, b) = zip(a, b)
    ▷ reduceSeq(fun (acc, x). acc + fst(x) × snd(x), 0)
```

Generating the equivalent C or OpenCL code from this representation is conceptually straightforward, but has some technical challenges that have been explored in prior work [5, 9]. The C function `dotSeqC` is generated from `dotSeq` implementing the dot product with a sequential reduction loop as expected:

```
void dotSeqC(float* output, int n, float* a, float* b) {
    float acc;
    acc = 0.0f;
    for (int i = 0; i < n; i++)      {
        acc = acc + (a[i] * b[i]); }
    output[0] = acc;                                  }
```

B. *High-Level and Low-Level Programs Represented in* RISE

RISE is a functional IR with anonymous functions (written as **fun** `x. b`), familiar function application (written as `f(x)`), identifiers and literals. The language is embedded in Scala, which allows meta-programming and the definition of macros (Scala code that will generate RISE IR). RISE also defines a set of high-level patterns to describe computations as shown in fig. 3 together with their types. These high-level computational patterns include applying a function to each element of an array (**map**) or reducing all elements of an array to a single value given a binary reduction function (**reduce**). There are also patterns such as **split**, **join**, or **transpose** for reshaping multi-dimensional array data in various ways. We write $s \to t$ for a function type with input of type s and output of type t, $[n]\,t$ for an array type with n elements of type t, $(s \times t)$ for a pair type with component types s and t. In RISE, the type system ensures that function types cannot be stored in memory, only data types.

+ | ×	:	$t \to t \to t$
map	:	$(s \to t) \to [n]\,s \to [n]\,t$
reduce	:	$(t \to t \to t) \to t \to [n]\,t \to t$
split	:	$(n : \mathsf{nat}) \to [nm]\,t \to [m]\,[n]\,t$
join	:	$[n]\,[m]\,t \to [nm]\,t$
transpose	:	$[n]\,[m]\,t \to [m]\,[n]\,t$
slide	:	$(sz\ sp : \mathsf{nat}) \to [sp \times n + sz - sp]\,t \to [n]\,[sz]\,t$
zip	:	$[n]\,s \to [n]\,t \to [n]\,(s \times t)$
fst	:	$(s \times t) \to s$
snd	:	$(s \times t) \to t$

Fig. 3: RISE high-level patterns and their type

mapSeq	:	$(s \to t) \to [n]\,s \to [n]\,t$
reduceSeq	:	$(s \to t \to s) \to s \to [n]\,t \to s$
mapGlobal	:	$(s \to t) \to [n]\,s \to [n]\,t$
toMem	:	$(a : \mathsf{addr}) \to t \to t$
asVector	:	$(m : \mathsf{nat}) \to [nm]\,t \to [n]\,\langle m \rangle\,t$
asScalar	:	$[n]\,\langle m \rangle\,t \to [nm]\,t$
vectorFromScalar	:	$t \to \langle m \rangle\,t$

Fig. 4: RISE low-level patterns and their type (not exhaustive)

The RISE compiler rewrites the high-level program into a low-level program that describes *how* the result is computed, encoding implementation decisions explicitly.

RISE's low-level patterns (fig. 4) indicate specific implementation decisions. For example, **mapSeq** and **reduceSeq** respectively implement **map** and **reduce** with sequential loops.

Some low-level patterns are specific to the target programming model (such as OpenCL) or hardware architecture (such as SIMD vector support). For OpenCL, **mapGlobal** introduces parallelism by parallelizing across global threads. The **toMem** primitive is used to explicitly encode storing an expression in the given address space in memory. Other patterns enable SIMD vectorization (e.g. **asVector**). A vector type with m elements of type t is written as $\langle m \rangle\,t$.

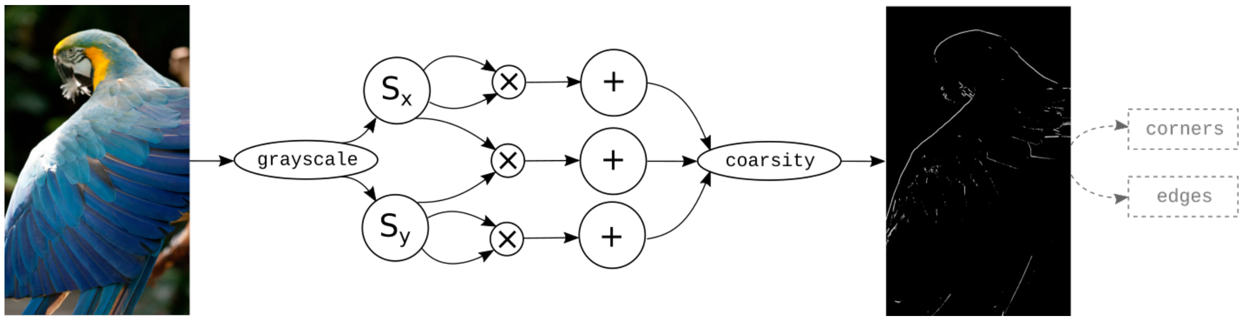


Fig. 5: Harris corner detection computation flow, illustrated with an example image from the Halide repository.

C. *Optimization Strategies Expressed in* ELEVATE

In addition to a high-level program describing the computations to be performed, our compiler also takes an optimization strategy as input as shown in the top left of fig. 2. Optimization strategies are written in the ELEVATE strategy language [8] and control the rewriting process by precisely determining how the given high-level program is optimized.

In ELEVATE, optimization strategies are encoded as functions transforming RISE programs. Such a strategy function might succeed and return the transformed program, or alternatively fail. Strategies are written as composition of other strategies and rewrite rules that directly encode the transformation of the program.

Strategy combinators help to write complex strategies by composition. The sequential (`;`) combinator composes two strategies by performing the second one on the transformed program from the first strategy. The `<+` combinator (called *left choice*) composes two strategies by performing the second if the first strategy fails. The `try` combinator tries to perform a given strategy, and does nothing if the strategy fails. `repeat` performs a strategy repeatedly until it fails.

Traversals control the location at which strategies perform their transformations. We have already seen `applyOnce` that traverses the program AST using a depth-first top-down traversal and performs the given strategy at the first possible location.

When writing rewrite rules and strategies, it helps to know that the program is in a particular syntactic form. To ensure that programs satisfy this form, we use the `normalize` traversal to perform a strategy to all possible program locations. After performing `normalize(s)`, we know that `s` can no longer be applied to any location in the transformed program.

D. *Summary*

We advocate a compiler design where both computational patterns and optimizations are easily extensible. Computations are expressed in the RISE IR using computational patterns. Optimizations are expressed in the ELEVATE strategy language as compositions of rewrite rules. Next we investigate how to represent image processing pipelines in the RISE language before later investigating how to encode important image processing optimizations. We focus on the Harris corner detection and ARM multi-core CPUs as a case-study.

III. REPRESENTING IMAGE PROCESSING PIPELINES IN RISE

The Harris corner (and edge) detector [10] is a well established image processing pipeline that we use as a case study in this paper. Many algorithmic variations exist, we use the one found in the Halide repository as our reference. This variant does not include padding for the stencil borders, and instead the output image is slightly smaller than the input image.

Fig. 5 shows the Harris operator. Given an image on the left, point-to-point operators (grayscale, multiplications $\times$, coarsity) and 3×3 convolutions (sobel operators S_x and S_y, sums $+$) are combined to detect corners and edges highlighted in the output on the right. As a composition of point wise and stencil operators, the Harris detector is more complex than its individual parts, and exposes more optimization opportunities. The optimizations that we study on the Harris detector are generalizable and applicable to other such compositions.

Representing the point-wise operators of the Harris corner detection in the generic high-level RISE intermediate language does not require any image-specific patterns. They are represented using a composition of the `map` pattern (`map2d`) and simple array transformations, as shown in listing 1. Note that we use `def` as syntactic sugar for functional `let` expressions which will be visible to optimization strategies.

Listing 1: High-level point-wise operators

```
def map2d(f: s → t): [n][m]s → [n][m]t = map(map(f))
def zip2d(a: [n][m]s, b: [n][m]t): [n][m](s × t) =
  zip(a, b) ▷ map(fun p. zip(fst(p), snd(p)))

def grayscale(RGB: [3][n][m]f32): [n][m]f32 =
  RGB ▷ transpose ▷ map(transpose)
  ▷ map2d(dot([0.299   0.587   0.114]))

def ×_2D(a, b: [n][m]f32): [n][m]f32 =
  zip2d(a, b) ▷ map2d(×)

def coarsity(
  S_xx, S_xy, S_yy: [n][m]f32, κ: f32
): [n][m]f32 =
  zip2d(S_xx, zip2d(S_xy, S_yy)) ▷ map2d(fun p.
    def (s_xx, (s_xy, s_yy)) = p
    def det = s_xx × s_yy - s_xy × s_xy
    def trace = s_xx + s_yy
    det - κ × trace × trace)
```

Listing 2: High-level stencil operators

```
def slide2d(n_sz: nat, n_sp: nat, m_sz: nat, m_sp: nat) =
  map(slide(n_sz, n_sp)) ▷ slide(m_sz, m_sp)
  ▷ map(transpose)

def stencil2d(f: [N][M]s → t):
    [n + N − 1][m + M − 1]s → [n][m]t =
  slide2d(N, 1, M, 1) ▷ map2d(f)

def conv3x3(weights: [3][3]f32):
    [n + 2][m + 2]f32 → [n][m]f32 =
  stencil2d(3, 3, fun w. dot(join(weights), join(w)))

                      ⎡−1  0  1⎤
def S_x = conv3x3(    ⎢−2  0  2⎥ × 1/12)
                      ⎣−1  0  1⎦

                      ⎡−1  −2  −1⎤
def S_y = conv3x3(    ⎢ 0   0   0⎥ × 1/12)
                      ⎣ 1   2   1⎦

def +_3×3 = stencil2d(3, 3, fun w. reduce(+, 0 join(w)))
```

Listing 3: High-level Harris detector

```
def harris(RGB: [3][n + 4][m + 4]f32): [n][m]f32 =
  def I = grayscale(RGB)
  def I_x = S_x(I)
  def I_y = S_y(I)
  def I_xx = ×_2D(I_x, I_x)
  def I_xy = ×_2D(I_x, I_y)
  def I_yy = ×_2D(I_y, I_y)
  def S_xx = +_3×3(I_xx)
  def S_xy = +_3×3(I_xy)
  def S_yy = +_3×3(I_yy)
  coarsity(S_xx, S_xy, S_yy, 0.04)
```

For the stencil operators we use the `slide` pattern, as introduced in LIFT in [7]. It creates a one-dimensional sliding window with a given size and step and is composed to create two-dimensional sliding windows with `slide2d`. Two-dimensional stencil operators are then expressed by first creating a sliding window with `slide2d` and then performing a local computation over the obtained neighborhood using `map2d` as shown in listing 2. Higher level abstractions are built on top, such as `conv3x3` creating a 3×3 convolution given weights.

Putting everything together, listing 3 shows the entire Harris corner detection expressed by composition of the previously defined building blocks. Note that the final RISE program only contains generic high-level patterns and basic language constructs - no image-specific internal representation is required. All abstractions such as `map2d` or `slide2d` are de-sugared into the patterns from fig. 3.

IV. OPTIMIZING IMAGE PROCESSING PIPELINES WITH ELEVATE

The LIFT project has been extended to express stencil computations and overlapped tiling [7], but it is missing crucial optimizations for image processing pipelines [11] that are supported by Halide. This leads to poor performance as seen in fig. 1. In this section, we use an optimized Halide schedule of the Harris operator as reference to demonstrate how ELEVATE is used to perform equivalent and additional optimizations by composing rewrites to transform RISE programs.

Listing 4: Optimized schedule for the Harris corner detection from the Halide GitHub repository

```
const int vec = natural_vector_size<float>();
output.split(y, y, yi, 32).parallel(y)
  .vectorize(x, vec);
gray.store_at(output, y).compute_at(output, yi)
  .vectorize(x, vec);
Ix.store_at(output, y).compute_at(output, yi)
  .vectorize(x, vec);
Iy.store_at(output, y).compute_at(output, yi)
  .vectorize(x, vec);
Ix.compute_with(Iy, x);
```

Listing 4 shows the Halide schedule describing the optimizations applied to the Harris operator. The schedule applies parallelism and vectorization as key optimizations as well as describing how the stages interact by storing memory in intermediate buffers. Halide makes some implicit optimization decisions appropriate for image processing pipelines, such as using circular buffers.

Figure 6 visualizes the computation with these optimizations applied. The upper part of fig. 6 shows the input image on the left, where the three color channel images are combined in the *grayscale* computation. Grayscale lines are stored in a temporary buffer to be processed by the sobel operators (S_x and S_y). The resulting buffers are then multiplied ($\times$), summed ($+$) and *coarsity* is applied to compute the final output. *Operator fusion* is applied to the computational flow (fig. 5) so that only two intermediate buffers are used. *Multi-threading* is exploited by parallelizing the y dimension and computing chunks of output lines in parallel (see the thread$_0$ and thread$_1$ annotations on the right). *Circular buffers* are used for the intermediate results. Each thread stores three lines in the buffer I. These lines are used to compute two lines and store them in buffers I_x and I_y. Similarly, three lines of both I_x and I_y are used to compute one line of the output.

The lower part of figure 6 shows two different ways to optimize the computations of individual image lines. The `cbuf` version is what Halide does: it uses *vectorization* to process lines one vector at a time. The `cbuf+rrot` version below is currently not possible with Halide: it also uses *vectorization* but further incorporates *convolution separation*, enabling *register rotation* as described in [11]. This is shown in the center and right of the bottom row: The two-dimensional reductions are decomposed in a vertical reduction followed by a horizontal reduction. Temporary vector registers are rotated to hold the last vertical reductions that are used for a horizontal reduction.

In the following subsections, we will first show how to replicate the optimizations described by the Halide schedule but as extensible ELEVATE strategies. This already goes beyond the capabilities of the existing LIFT compiler. Then, we will show how to go beyond the optimization that Halide performs by incorporating the additional optimizations convolution separation and register rotation.

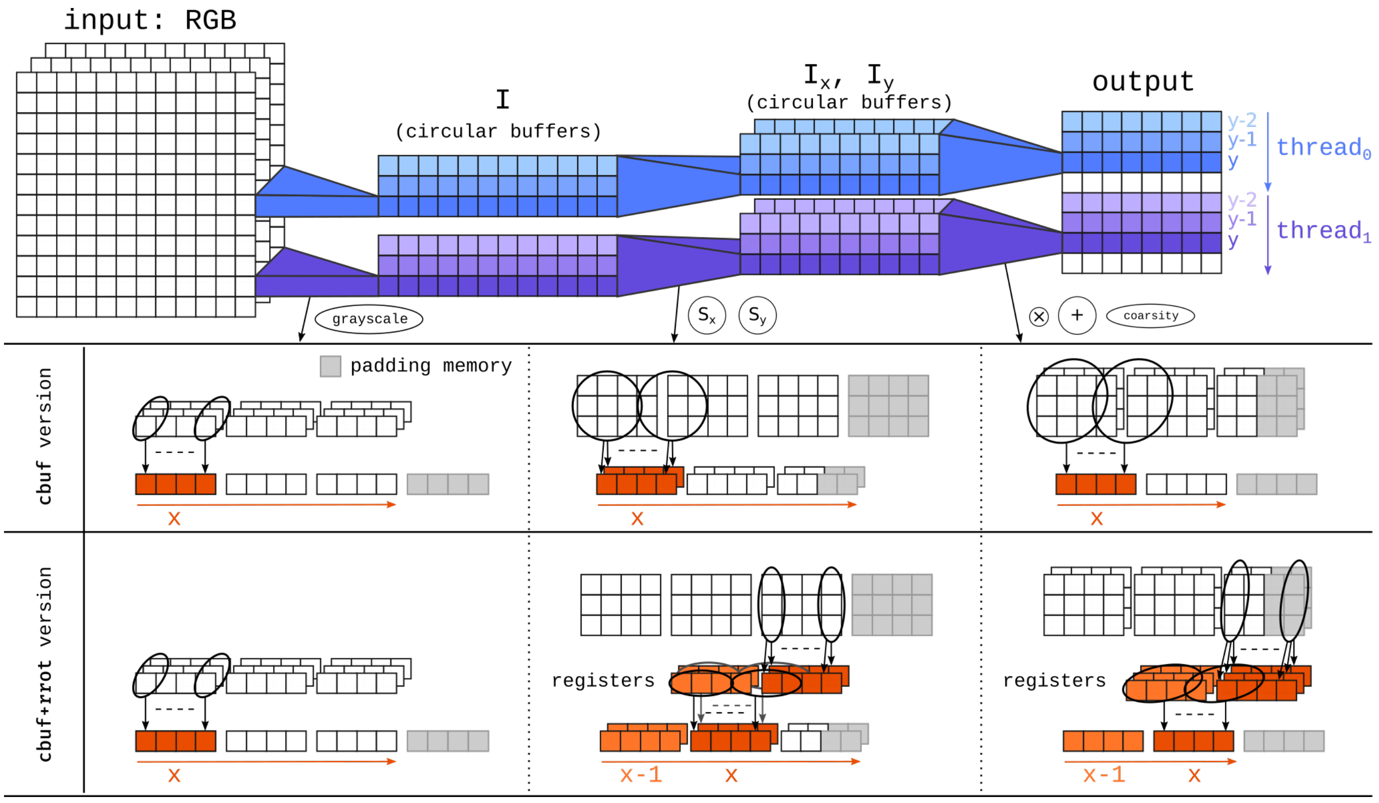


Fig. 6: Overview of the optimizations applied on the Harris corner detector.

A. *Reproducing the Halide Optimizations with* ELEVATE

Listing 5 shows the ELEVATE optimization strategy that reproduces the reference Halide schedule through a composition of smaller strategies – themselves composed of rewrite rules. In the following, we are discussing how to express the individual optimization strategies in ELEVATE one-by-one.

Operator Fusion: The reference Halide schedule specifies which temporary values should be stored in memory using `store_at` directives. Otherwise operators are fused by default, storing temporary results in registers instead of memory. This transformation is more complex than loop fusion, which is why LIFT fails to apply it using its simple `map`-fusion rules. In, ELEVATE we define the `fuseOperators` strategy transforming the Harris program (listing 3) into a pipeline over image lines:

```
map(grayLine)  ▷ slide(3,1) ▷
map(sobelLine) ▷ slide(3,1) ▷ map(coarsityLine)
```

Listing 5: ELEVATE optimization strategy using circular buffering for the Harris operator

```
strategy cbufVersion =
  fuseOperators;
  splitPipeline(32); parallel;
  vectorizeReductions(vec);
  harrisIxWithIy;
  circularBufferStages;
  sequentialLines;
  usePrivateMemory; unrollReductions
```

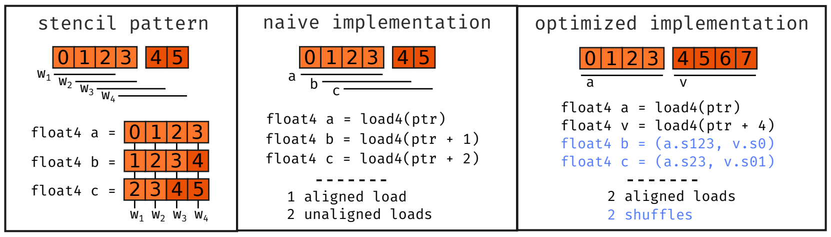
Where `grayLine` is a function computing a grayscale line, `sobelLine` a function computing a line of sobel convolutions, and `coarsityLine` a function computing a line of output (with multiplications, sums and coarsity fused) as shown in fig. 6.

Multi-threading: To take advantage of thread-level parallelism, the Halide schedule splits the output into chunks of 32 lines that are processed in parallel: `output.split(y, y, yi, 32).parallel(y)`. The ELEVATE strategy `splitPipeline(32);parallel` has the same effect, producing a program that slides over $p+4$ lines of input with step p to compute chunks of size p in parallel:

```
slide(p+4, p) ▷ mapGlobal(
  map(grayLine) ▷ slide(3,1) ▷
  map(sobelLine) ▷ slide(3,1) ▷
  map(coarsityLine)
) ▷ join
```

Parallelism is achieved by using the low-level `mapGlobal` primitive that applies the nested function in parallel across global threads. The strategy itself starts by splitting the last map in the pipeline with the `splitJoin` rewrite rule. Then, it propagates this split to the rest of the pipeline by normalizing it with various movement rules. Finally, all possible map fusions are applied in the pipeline. All the involved rules are in listing 6.

Vectorization: Vectorization uses SIMD parallelism (Single Instruction, Multiple Data) through special instructions such as the NEON instructions on ARM processors. In the Halide schedule, this optimization is enabled by multiple `.vectorize(x, vec)` directives.



```
float4 a = 0 1 2 3
float4 b = 1 2 3 4
float4 c = 2 3 4 5
```

```
float4 a = load4(ptr)
float4 b = load4(ptr + 1)
float4 c = load4(ptr + 2)
-------
1 aligned load
2 unaligned loads
```

```
float4 a = load4(ptr)
float4 v = load4(ptr + 4)
float4 b = (a.s123, v.s0)
float4 c = (a.s23, v.s01)
-------
2 aligned loads
2 shuffles
```

Fig. 7: Example of memory loads for a vectorized 1D stencil of size 3. The code is written in pseudo OpenCL syntax.

Listing 6: Rules involved in the multi-threading optimization

```
rule splitJoin(p: nat) =
  map(f) ↦ split(p) ▷ map(map(f)) ▷ join

rule slideAfterSplit =
  slide(n, m) ▷ split(p)
  ↦ slide(p+n-m, p) ▷ map(slide(n, m))

rule slideBeforeMap =
  map(f) ▷ slide(n, m) ↦ slide(n, m) ▷ map(map(f))

rule slideBeforeSlide =
  slide(n, 1) ▷ slide(m, k)
  ↦ slide(m+n-1, k) ▷ map(slide(n, 1))

rule mapFusion = map(f) ▷ map(h) ↦ map(f ▷ h)

rule useMapGlobal = map(f) ↦ mapGlobal(f)
```

Listing 7: Strategy and rules involved in vectorization

```
strategy vectorize(v: nat) =
  startVectorization(v);
  normalize(
    vectorizeBeforeMap <+
    vectorizeBeforeMapReduce);

rule startVectorization(v: nat) =
  a: [n × v] s ↦ a ▷ asVector(v) ▷ asScalar

rule vectorizeBeforeMap =
  map(f) ▷ asVector(v) ↦ asVector(v) ▷ map(mapVec(f))

rule vectorizeBeforeMapReduce =
  map(reduce(f, init)) ▷ asVector(v)
  ↦ a ▷ transpose ▷ map(asVector(v)) ▷ transpose
    ▷ map(reduce(mapVec(f), vectorFromScalar(init)))
```

The ELEVATE strategy `vectorizeReductions(vec)` has a similar effect, vectorizing all reductions of a program. To illustrate how the strategy works, we consider a sub-program found in the Harris operator:

```
map(reduce(+, 0)) ▷ map(f)
```

It is vectorized by interpreting the input as a two dimensional array of vectors using **asVector** and computing on vectorized data before going back to scalars using **asScalar**:

```
transpose ▷ map(asVector(v)) ▷ transpose
  ▷ map(reduce(mapVec(+), vectorFromScalar(0)))
  ▷ map(mapVec(f)) ▷ asScalar
```

Where the **mapVec** pattern vectorizes a scalar function - this is currently supported for functions that use basic operations such as addition and multiplication.

This program transformation can be defined with an ELEVATE strategy composing simpler rewrite rules as shown in listing 7. In practice, arrays are often not multiples of the vector width. There are different ways to handle this, we simply round inputs, outputs and temporaries up to a multiple of the vector width - an option that Halide also provides.

When vectorizing stencils the computations are performed on the w_i components of three vector values as shown in the left of fig. 7. The inputs of vectorized stencils are not aligned in memory and can be loaded in different ways. The naive implementation performs three loads, two of which are not aligned at a vector boundary. The optimized implementation, used by RISE, only performs two vector loads followed by vector shuffle instructions.

Circular Buffering: Circular buffers leverage both the spatial locality of stencils and the temporal locality of sequential execution: only the last m intermediate results need to be stored in memory, and modulo indexing is used: $T[i]$ can be stored in $M[i \mod m]$. With transparently managed caches, this reduces memory usage and delays cache overflow. In the Halide schedule, `.store_at(output, y).compute_at(output, yi)` implicitly triggers the use of circular buffers for the introduced temporary. When combined with the previous multi-threading optimization, a separate set of circular buffers is used inside each parallel chunk – where execution is still sequential – as shown in fig. 6. The ELEVATE strategy `circularBufferStages` has the same effect, producing a program with the shape:

```
slide(p+4, p) ▷ mapGlobal(
  circularBuffer(global, 3, grayLine) ▷
  circularBuffer(global, 3, sobelLine) ▷
  mapSeq(coarsityLine)
) ▷ join
```

The **circularBuffer** pattern is a new low-level pattern that we added to RISE. Given an input array, the **circularBuffer** pattern returns an array of sliding windows similar to the **slide** pattern, but the last m values have been loaded into the circular buffer. It also takes as arguments an OpenCL address space, the size m of the buffer, and a function used to load values into the buffer as arguments. The **mapSeq** pattern is used to read sequentially from the circular buffer. The `circularBufferStages` strategy itself works by rewriting **slide** into the **circularBuffer** pattern, fusing **circularBuffer** and **map**, and introducing the **mapSeq** pattern using the rewrites rules of listing 8.

Additional Optimizations: A couple of additional optimizations are encoded as ELEVATE strategies. The `harrisIxWithIy` strategy emulates the `Ix.compute_with(Iy, x)`

Listing 8: Rewrite rules involved in circular buffering

```
rule useCBuffer(a: addr) =
  slide(m, 1) ↦ circularBuffer(a, m, fun x. x)

rule cBufferLoadFusion =
  circularBuffer(a, m, load, map(f, in))
  ↦ circularBuffer(a, m, fun x. load (f x), in)

rule useMapSeq = map(f) ↦ mapSeq(f)
```

schedule directive from Halide, fusing the loops computing these two intermediate results. The `sequentialLines` strategy makes individual line computations sequential, `usePrivateMemory` stores various temporaries in private memory, and `unrollReductions` unrolls reduction loops. These transformations are not mentioned in the Halide schedule, the two first ones happen implicitly, while reductions have already been unrolled in the algorithm definition. All these optimizations are already well supported as rewrites by LIFT but have not been encoded as ELEVATE strategies.

B. *Expressing Optimizations beyond Halide with* ELEVATE

Listing 9 shows an ELEVATE strategy re-using the optimizations from section IV-A and additionally incorporating convolution separation and register rotation (highlighted in the listing). These two optimizations are orthogonal from multi-threading and circular buffering as they operate on a different dimension. Separating the convolution is necessary to enable register rotation, and is not expressible in Halide without manually changing the algorithm. Register rotation is not implemented in Halide, although it is seen as a worthwhile optimization (https://github.com/halide/Halide/issues/2905). Implementing register rotation in Halide would require extending the compiler as well as exposing the optimization via the scheduling API, resulting in significant work. We discuss here how this optimization is expressed elegantly outside of the compiler as an ELEVATE strategy.

Convolution Separation: The two-dimensional sobel and sum convolutions in the Harris detector have an important property: they are separable into two one-dimensional convolutions following the observation that the convolution kernel matrix is separable into a column and row vector:

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

Listing 9: ELEVATE strategy using circular buffering and register rotation. Changes compared to listing 5 are highlighted.

```
strategy cbuf+rrotVersion =
  fuseOperators;
  splitPipeline(32); parallel;
  separateConvolutions;
  vectorizeReductions(vec);
  harrisIxWithIy;
  circularBufferStages;
  rotateValuesAndConsumeLines;
  usePrivateMemory; unrollReductions
```

Listing 10: Strategy and rules to apply and push convolution separation through a line computation

```
rule separateConvKernel(weights2d, wV, wH) =
  dot(join(weights2d), join(nbh))
  ↦ nbh ▷ transpose ▷ map(dot(wV)) ▷ dot(wH)

strategy pushSeparation(separate) =
  applyOnce(separate); reducedFissionedForm;
  applyOnce(mapSlideAfterTranspose);
  reducedFusedForm; reducedFissionedForm;
  normalize(slideAfterMapMapF)

rule mapSlideAfterTranspose =
  map(slide(n, m)) ▷ transpose
  ↦ transpose ▷ slide(n, m) ▷ map(transpose)

rule slideAfterMapMapF =
  slide(n, m) ▷ map(map(f)) ↦ map(f) ▷ slide(n, m)
```

This decomposition is used to reduce both memory accesses and arithmetic complexity, but is not possible for arbitrary convolutions as it depends on the weights involved. With ELEVATE, such a domain- or even program-specific optimization is easily definable outside of the compiler.

To separate a convolution such as the sobel convolution (S) we start by takeing a one-dimensional stencil neighborhood (`nbhV`) and using `slide` and `transpose` to create the 2D neighborhood (`nbh2d`) before performing a dot product between the 2D weights and the neighborhood:

```
nbhV ▷ map(slide(3,1)) ▷ transpose ▷ map(fun nbh2d.
  dot(join(weights2d), join(nbh2d)) )
```

The 2D weights are separated into vertical weights (`weightsV`) that are used to perform a 1D convolution and horizontal weights (`weightsH`) that are used in a second 1D convolution:

```
nbhV ▷ transpose ▷ map(dot(weightsV))
    ▷ slide(3,1) ▷ map(dot(weightsH))
```

The strategy `pushSeparation(separateConvKernel(weights2d, weightsV, weightsH))` shown in listing 10 orchestrates this transformation by composing simpler rewrite rules. The domain-specific rewrite rule `separateConvKernel` encodes the decomposition of the convolution kernel as a dot product decomposition. It needs to be given the separated weights explicitly. A similar rule could also encode a horizontal-vertical decomposition. To separate the entire convolution, we have to "push" the dot product decomposition across the surrounding dimensions. A sequence of generic rules is used to implement the `pushSeparation` strategy that achieves this.

`separateConvolutions` in listing 9 uses these components to separate the sobel and sum convolutions of the Harris operator.

Register Rotation: Like circular buffering, register rotation leverages spatial locality of stencils and temporal locality of sequential execution. Instead of using circular indexing, register rotation puts temporary results into registers and rotates them between computation iterations. In the bottom of fig. 6, convolution separation is combined with register rotation and vectorization: vectors of computed vertical reductions are rotated while computing vectors of horizontal reductions.

Listing 11: Strategy and rules involved in register rotation

```
strategy rotateValuesAndConsume =
  applyOnce(useRotateValues(private));
  applyOnce(useMapSeq)

rule useRotateValues(a: addr) =
  slide(m, 1) ↦ rotateValues(a, m, fun x. x)
```

Starting from a separated convolution resulting from the `separateConvolutions` optimization discussed above:

```
transpose ▷ map(dot(wV)) ▷ slide(3,1) ▷ map(dot(wH))
```

we introduce a set of rotating registers to store the sliding window, replacing the second **`slide`** pattern. As the registers rotate one register at a time we must use **`mapSeq`**:

```
transpose ▷ map(dot(wV)) ▷ rotateValues(private, 3)
▷ mapSeq(dot(wH))
```

To express register rotation, RISE has been extended with the **`rotateValues`** low-level pattern:

`rotateValues`: $(a : \mathsf{addr}) \to (m : \mathsf{nat}) \to [n + m - 1]\,t \to [n]\,[m]\,t$

Given an input array, the **`rotateValues`** pattern returns an array of sliding windows: the last m values that have been stored in registers. Values are rotated while the array is read sequentially. The underlying register allocation is performed by the OpenCL compiler, thus this pattern alone does not guarantee the use of registers but we have observed the expected behaviour and good performance with this design.

The ELEVATE strategy `rotateValuesAndConsume` shown in listing 11 orchestrates this program transformation by composing simpler rewrite rules. A similar strategy is used in listing 9 to apply this optimization to the Harris operator.

C. Summary

In this section we have discussed how well-known image processing pipeline optimizations that were not implemented in the LIFT compiler are expressed in a composable and extensible way as ELEVATE optimization strategies. As well as expressing optimizations already supported by Halide, we expressed additional optimizations. In the next section, we will investigate whether we truly achieve performance competitive with the Halide reference, and what are the benefit of the additional optimizations we expressed.

V. Experimental Evaluation

In this section we report a systematic runtime performance comparison between the RISE compiler, the Halide compiler, the LIFT compiler, and the OpenCV image processing library.

A. Experimental Setup

Experiments are conducted on two computers with ARM big.LITTLE configuration as these mobile CPUs are often used in image processing applications. We use an Odroid XU4 board with a 4-core Cortex A7 and a 4-core Cortex A15 as well as an Odroid N2 board with a 2-core Cortex A53 and a 4-core Cortex A73.

During benchmarks, we set the frequencies to 1.5Ghz for the XU4, and 1.8Ghz for the N2. Our compiler implementation generates OpenCL kernels that are executed using the POCL [12] open source implementation of OpenCL that is built on top of LLVM. We used POCL 1.3 with LLVM 8 on the XU4 and POCL 1.5 with LLVM 10 on the N2.

The OpenCL kernels generated with RISE are compared against OpenCV, the OpenCL kernels generated with the LIFT implementation from [7], and the binaries generated by Halide (commit c2b6da2 https://tinyurl.com/rr7awsr). We use OpenCV 4.3 with NEON vector support enabled. For RISE we use the optimizations discussed in section IV implemented as ELEVATE strategies to optimize the Harris corner detection. For Halide we use the reference optimized schedule from listing 4. Neither the RISE-generated OpenCL code nor the Halide schedule is specialized for each individual processor, but the final assembly will be respectively specialized by the OpenCL implementation and the Halide compiler.

We report the median runtime of 30 executions, which we found gives reasonably stable results. To measure the runtime of the OpenCL kernels, we use the OpenCL profiling API. For Halide we use C++'s `std::chrono` clocks as it is done with other benchmarks in the Halide repository.

Two input images are used, one with a resolution of 1536×2560 pixels, and one of 4256×2832 pixels. The first one is taken from the Halide repository, and was shown in fig. 5.

While benchmarking, we also verify that the outputs of the different Harris operator implementations are consistent by computing the Mean-Squared Error and PSNR (Peak Signal-to-Noise Ratio) with the reference output from Halide. The recorded PSNR is always above 170 decibels. This high value indicates a very strong similarity.

B. Performance Results

Figure 8 shows the measured runtime performances. All three compilers, LIFT, RISE and Halide, outperform the OpenCV baseline on all processors, although OpenCV describes itself as a highly optimized library. This highlights the performance benefits brought by whole-program optimizing compilers exploiting the semantics of high-level abstractions. While Halide's abstractions are specifically designed for image processing pipelines, the abstractions for LIFT and RISE are the high-level patterns that they exploit via rewriting.

RISE outperforms LIFT clearly, because prior LIFT work focuses on individual stencil computations and lacks crucial optimizations for image processing pipelines: notably operator fusion and circular buffering.

Without convolution separation and register rotation, RISE is on par with the Halide reference on all processors (except for the small input image on the A73). We observe that while the coarse-grain optimizations are the same, small differences in the generated code remain and that the resulting performance depends on fine-grain code generation details down to assembly which are out of the scope of this paper.

With convolution separation and register rotation, RISE always performs much better than without (almost 30% faster

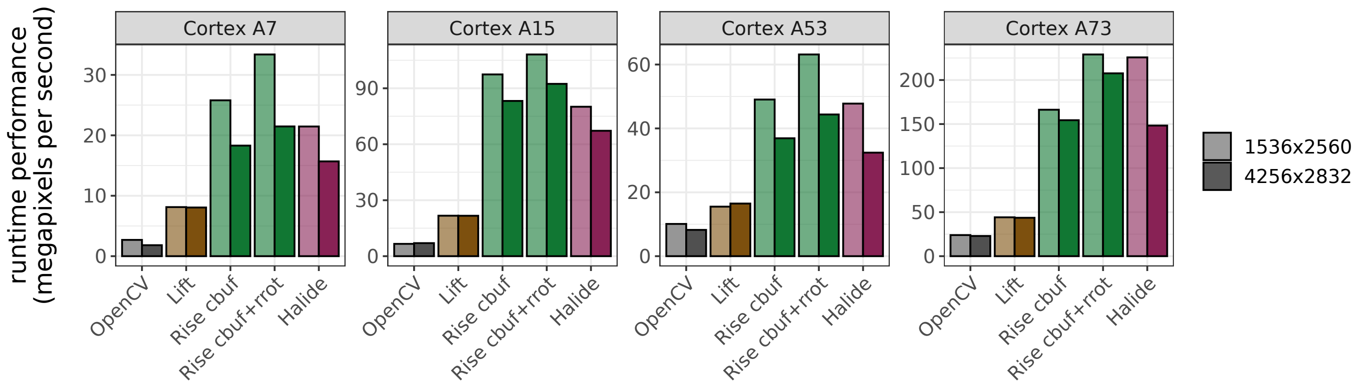


Fig. 8: Runtime performance of the Harris operator for the different processors, implementations and image resolutions.

on average) and is also faster than the Halide reference in almost all cases by more than 30%. This shows that register rotation is an optimization worth considering even though it has not been implemented in Halide.

These results demonstrate that a domain-extensible compiler design where compiler optimizations are implemented as extensible strategies composed of rewrite rules is capable of achieving the same performance as a highly optimized compiler - such as Halide - specifically built for image processing. Furthermore, higher performance can be reached by extending such a domain-extensible compiler with optimizations that are not built into existing domain-specific compilers.

VI. Related Work

Image Processing Optimizations: Image processing optimizations have been studied for decades, and are essential to increase run-time performance and energy efficiency. All the optimizations that we use in this paper are well-known, and have been studied on the Harris operator before in [11]. Circular buffering is commonly used between pipeline stages, and is automatically applied by many image processing specific compilers such as Halide [3] or Darkroom [13]. Convolution separation has long been used to reduce the computational complexity of convolutions [14, 15, 16, 17], but is often manually applied. More generally, stencil optimizations such as overlapped tiling [18, 19, 20, 21] are well-studied and are used beyond the image processing domain.

Numerous libraries of optimized functions (e.g. OpenCV, NPP, Intel IPP) implement some of these optimizations. However, they are limited in functionality and many critical optimizations cannot be applied across library calls.

Domain-Specific Compilers: Domain-specific compilers generate high performance code from high level abstractions by exploiting domain and hardware knowledge. Polyhedral compilation techniques [22, 23] are often leveraged, such as in PolyMage [24] for image processing, but also in compilers for linear algebra or machine learning [25, 26]. Although automatically yielding great performance, such compilers rely on fixed heuristics and performance models, offering little control over the optimizations. By separating algorithm from schedule, Halide [3] offers more control over the optimizations. This empowers users to optimize their programs with far more precision than compiler flags or configuration files. Since the success of Halide, such a separation is found in many compilers [27, 28, 29]. However, it remains that domain-specific compiler designs mostly account for a restricted set of abstractions and optimizations. As a result, they lack flexibility and extensibility.

Domain Extensible Compilers: Delite [1] has been an early framework providing parallel patterns, generic optimizations and code generation to simplify the development of domain specific languages. This allows a fixed set of parallel patterns and generic optimizations to be re-used across domains. However, Delite relies on optimization passes that offer little control over the optimizations applied. AnyDSL [2] is a more recent approach leveraging partial evaluation to allow writing more powerful and flexible optimizing libraries. However, designing and extending such libraries remains an open problem.

This paper follows ideas from the Lift project [4], where computations are encoded with data-parallel patterns, and optimizations are encoded in a system of rewrite rules. This paper directly builds on the recent work on Rise and Elevate [8] that allow compiler optimizations to be expressed as user-defined strategies in the new strategy language Elevate. We make use of these user-defined optimization strategies to provide precise control over the applied optimizations. This paper is the first to study how to apply these ideas on a complete image processing pipeline composed of multiple operators, along with the specific optimizations involved to reach high-performance.

VII. Conclusion and Future Work

In this paper, we showed that a domain-extensible compiler is capable of expressing and optimizing image processing pipelines. Using the Harris corner detection as a case study, our runtime results on four mobile ARM multi-core CPUs show that our compiler significantly outperforms OpenCV library code, clearly outperforms the similarly designed Lift compiler and is competitive to the state-of-the-art image processing compiler Halide. On top of expressing an optimized schedule from Halide as extensible user-defined optimization strategies applying operator fusion and circular buffering, we are able to reach higher performance by incorporating convolution separation and register rotation optimizations. These two optimizations cannot be expressed in a Halide schedule, but are easily expressed as RISE patterns, rewrite rules and ELEVATE strategies.

We used optimization strategies to precisely control the applied optimizations: automated heuristics and explorations are not always desirable or even feasible as they lack user control, may result in poor performance, and may be too time consuming. However, such fine-grain optimization strategies are not easy to write, can be over-detailed and program-specific. In the future, we envision using strategies to offer tradeoffs between precise control (as in this paper) and full automation (as in LIFT). Users would then provide strategies that constrain and guide the automated exploration process. Finally, we believe that the presented patterns and rewrite rules are re-usable beyond image processing and across hardware targets, just as the underlying optimizations are.

ACKNOWLEDGMENT

We would like to thank the LIFT and RISE teams for their work; the Halide developers for maintaining an open-source project; and Phil TRINDER for his valuable feedback.

APPENDIX

A. Abstract

This artifact contains the source code used to produce the performance results presented in the paper. The host computer drives benchmarks on multiple target processors over ssh.

B. Artifact Check-List (Meta-Information)

- **Program:** The RISE and ELEVATE Scala implementations; The Halide compiler; The LIFT-generated OpenCL kernels; Benchmark and plotting programs
- **Compilation:** With provided scripts
- **Data set:** Provided images
- **Run-time environment:** X86 Linux host; Linux targets with OpenCL
- **Hardware:** Any OpenCL-enabled CPU
- **Output:** OpenCL kernels; runtime CSVs; figure PDFs
- **How much disk space required (approximately)?:** 2GB on the host; 20MB on the targets (dependencies excluded)
- **How much time is needed to prepare workflow (approximately)?:** 1 hour to 1 day
- **How much time is needed to complete experiments (approximately)?:** 1 hour
- **Publicly available?:** Yes

C. Description

1) How Delivered: The artifact is publicly available on GitHub: https://github.com/rise-lang/2021-CGO-artifact

2) Hardware Dependencies: To reproduce the results reported in fig. 1 and fig. 8, you will need access to ARM Cortex A7, A15, A53 and A73 processors (we used Odroid XU4 and Odroid N2 boards). Other OpenCL-enabled processors can be used, but expect different performance behavior.

3) Software Dependencies: We recommend using an X86 Linux for the host, and Linux targets. The software dependencies are listed in the README and we provide an Ubuntu Focal Fossa (20.04 LTS) Dockerfile for convenience.

D. Installation

Clone the repository on the host (potentially from the docker image). Detailed instructions are given in the README.

E. Experiment Workflow

The provided scripts should be used to generate code, run benchmarks and plot figures, as described in the README.

F. Evaluation and Expected Result

The main goals for artifact evaluation is to use the provided Rise compiler to regenerate the OpenCL kernels used in the experimental evaluation and to reproduce the performance results seen in fig. 1 and fig. 8. For the same processors (or similar enough), we expect the results to show similar performance trends as observed in section V-B.

G. Experiment Customization

The benchmarks can be run on different CPUs by writing a small configuration file as long as the benchmark dependencies are available.

H. Methodology

Submission, reviewing and badging methodology:

- http://cTuning.org/ae/submission-20190109.html
- http://cTuning.org/ae/reviewing-20190109.html
- https://www.acm.org/publications/policies/artifact-review-badging

REFERENCES

[1] H. Chafi, A. K. Sujeeth, K. J. Brown, H. Lee, A. R. Atreya, and K. Olukotun, "A domain-specific approach to heterogeneous parallelism," in *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2011, San Antonio, TX, USA, February 12-16, 2011.* ACM, 2011, pp. 35–46. [Online]. Available: https://doi.org/10.1145/1941553.1941561

[2] R. Leißa, K. Boesche, S. Hack, A. Pérard-Gayot, R. Membarth, P. Slusallek, A. Müller, and B. Schmidt, "Anydsl: a partial evaluation framework for programming high-performance libraries," *Proc. ACM Program. Lang.*, vol. 2, no. OOPSLA, pp. 119:1–119:30, 2018. [Online]. Available: https://doi.org/10.1145/3276489

[3] J. Ragan-Kelley, A. Adams, S. Paris, M. Levoy, S. P. Amarasinghe, and F. Durand, "Decoupling algorithms from schedules for easy optimization of image processing pipelines," *ACM Trans. Graph.*, vol. 31, no. 4, pp. 32:1–32:12, 2012. [Online]. Available: https://doi.org/10.1145/2185520.2185528

[4] M. Steuwer, C. Fensch, S. Lindley, and C. Dubach, "Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance opencl code," in *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015.* ACM, 2015, pp. 205–217. [Online]. Available: https://doi.org/10.1145/2784731.2784754

[5] M. Steuwer, T. Remmelg, and C. Dubach, "Lift: a functional data-parallel IR for high-performance GPU code generation," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017.* ACM, 2017, pp. 74–85. [Online]. Available: http://dl.acm.org/citation.cfm?id=3049841

[6] T. Remmelg, T. Lutz, M. Steuwer, and C. Dubach, "Performance portable GPU code generation for matrix multiplication," in *Proceedings of the 9th Annual Workshop on General Purpose Processing using Graphics Processing Unit, GPGPU@PPoPP 2016, Barcelona, Spain, March 12 - 16, 2016*. ACM, 2016, pp. 22–31. [Online]. Available: https://doi.org/10.1145/2884045.2884046

[7] B. Hagedorn, L. Stoltzfus, M. Steuwer, S. Gorlatch, and C. Dubach, "High performance stencil code generation with lift," in *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018, Vösendorf / Vienna, Austria, February 24-28, 2018*. ACM, 2018, pp. 100–112. [Online]. Available: https://doi.org/10.1145/3168824

[8] B. Hagedorn, J. Lenfers, T. Koehler, X. Qin, S. Gorlatch, and M. Steuwer, "Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies," *Proc. ACM Program. Lang.*, vol. 4, no. ICFP, pp. 92:1–92:29, 2020. [Online]. Available: https://doi.org/10.1145/3408974

[9] R. Atkey, M. Steuwer, S. Lindley, and C. Dubach, "Strategy preserving compilation for parallel functional code," *CoRR*, vol. abs/1710.08332, 2017. [Online]. Available: http://arxiv.org/abs/1710.08332

[10] C. G. Harris and M. Stephens, "A combined corner and edge detector," in *Proceedings of the Alvey Vision Conference, AVC 1988, Manchester, UK, September, 1988*, 1988, pp. 1–6. [Online]. Available: https://doi.org/10.5244/C.2.23

[11] L. Lacassagne, D. Etiemble, A. H. Zahraee, A. Dominguez, and P. Vezolle, "High level transforms for SIMD and low-level computer vision algorithms," in *Proceedings of the 2014 Workshop on Programming models for SIMD/Vector processing, WPMVP 2014, Orlando, Florida, USA, February 16, 2014*, 2014, pp. 49–56. [Online]. Available: https://doi.org/10.1145/2568058.2568067

[12] P. Jääskeläinen, C. S. de La Lama, E. Schnetter, K. Raiskila, J. Takala, and H. Berg, "pocl: A performance-portable opencl implementation," *Int. J. Parallel Program.*, vol. 43, no. 5, pp. 752–785, 2015. [Online]. Available: https://doi.org/10.1007/s10766-014-0320-y

[13] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: compiling high-level image processing code into hardware pipelines," *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144:1–144:11, 2014. [Online]. Available: https://doi.org/10.1145/2601097.2601174

[14] P. M. Narendra, "A separable median filter for image noise smoothing," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 3, no. 1, pp. 20–29, 1981. [Online]. Available: https://doi.org/10.1109/TPAMI.1981.4767047

[15] J. Geusebroek, A. W. M. Smeulders, and J. van de Weijer, "Fast anisotropic gauss filtering," *IEEE Trans. Image Process.*, vol. 12, no. 8, pp. 938–943, 2003. [Online]. Available: https://doi.org/10.1109/TIP.2003.812429

[16] V. Areekul, U. Watchareeruetai, K. Suppasriwasuseth, and S. Tantaratana, "Separable gabor filter realization for fast fingerprint enhancement," in *Proceedings of the 2005 International Conference on Image Processing, ICIP 2005, Genoa, Italy, September 11-14, 2005*. IEEE, 2005, pp. 253–256. [Online]. Available: https://doi.org/10.1109/ICIP.2005.1530376

[17] T. Q. Pham and L. J. van Vliet, "Separable bilateral filtering for fast video preprocessing," in *Proceedings of the 2005 IEEE International Conference on Multimedia and Expo, ICME 2005, July 6-9, 2005, Amsterdam, The Netherlands*. IEEE, 2005, pp. 454–457. [Online]. Available: https://doi.org/10.1109/ICME.2005.1521458

[18] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: ACM, 2007, p. 235–244. [Online]. Available: https://doi.org/10.1145/1250734.1250761

[19] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. A. Patterson, J. Shalf, and K. A. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the ACM/IEEE Conference on High Performance Computing, SC 2008, November 15-21, 2008, Austin, Texas, USA*. IEEE/ACM, 2008, p. 4. [Online]. Available: https://doi.org/10.1109/SC.2008.5222004

[20] J. Holewinski, L.-N. Pouchet, and P. Sadayappan, "High-performance code generation for stencil computations on gpu architectures," in *Proceedings of the 26th ACM International Conference on Supercomputing*, ser. ICS '12. New York, NY, USA: ACM, 2012, p. 311–320. [Online]. Available: https://doi.org/10.1145/2304576.2304619

[21] X. Zhou, J.-P. Giacalone, M. J. Garzarán, R. H. Kuhn, Y. Ni, and D. Padua, "Hierarchical overlapped tiling," in *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, ser. CGO '12. New York, NY, USA: ACM, 2012, p. 207–218. [Online]. Available: https://doi.org/10.1145/2259016.2259044

[22] C. Bastoul, "Code generation in the polyhedral model is easier than you think," in *13th International Conference on Parallel Architectures and Compilation Techniques (PACT 2004), 29 September - 3 October 2004, Antibes Juan-les-Pins, France*, 2004, pp. 7–16. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/PACT.2004.10018

[23] A. Cohen, M. Sigler, S. Girbal, O. Temam, D. Parello, and N. Vasilache, "Facilitating the search for compositions of program transformations," in *Proceedings of the 19th Annual International Conference on Supercomputing*, ser. ICS '05. New York, NY, USA: ACM, 2005, p. 151–160. [Online]. Available: https://doi.org/10.1145/1088149.1088169

[24] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 429–443, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694364

[25] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, "Diesel: Dsl for linear algebra and neural net computations on gpus," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018. New York, NY, USA: ACM, 2018, p. 42–51. [Online]. Available: https://doi.org/10.1145/3211346.3211354

[26] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions," *arXiv preprint arXiv:1802.04730*, 2018.

[27] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze *et al.*, "{TVM}: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 578–594.

[28] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, A. Akkas, Y. Zhang, P. Suriana, S. Kamil, and S. P. Amarasinghe, "Tiramisu: A polyhedral compiler for expressing fast and portable code," *CoRR*, vol. abs/1804.10694, 2018. [Online]. Available: http://arxiv.org/abs/1804.10694

[29] B. Hagedorn, A. S. Elliott, H. Barthels, R. Bodik, and V. Grover, "Fireiron: A scheduling language for high-performance linear algebra on gpus," *arXiv preprint arXiv:2003.06324*, 2020.