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

Thomas Kœhler Michel Steuwer

CGO 2021

Introduction

Domain-specific compilers such as Halide

- + convenient programming
- + high-performance

Halide algorithm: *what to compute*

```
blur_x(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
blur_y(x, y) = (blur_x(x, y-1) + blur_x(x, y) + blur_x(x, y+1))/3;
```

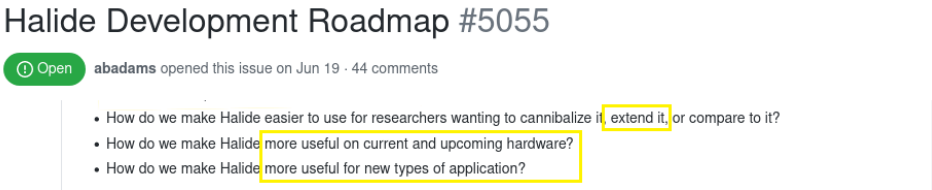
Halide schedule: *how to optimize*

```
blur_y.tile(x, y, xi, yi, 256, 32)
    .vectorize(xi, 8).parallel(y);
blur_x.compute_at(blur_y, x).vectorize(x, 8);
```

Introduction

Domain-specific compilers such as Halide

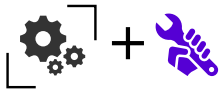
- fixed set of abstractions and optimizations
- lack of flexibility and extensibility



https://github.com/halide/Halide/issues/5055

Introduction

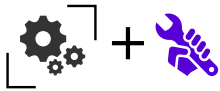
Compilers such as Lift aim to be domain-extensible



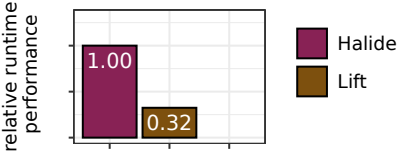
+ extensible set of abstractions and optimizations
- competitive with domain-specific compilers?

Introduction

Compilers such as Lift aim to be domain-extensible



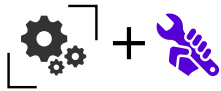
- \+ extensible set of abstractions and optimizations
- \- competitive with domain-specific compilers?



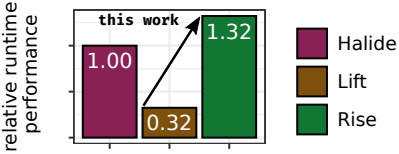
Lift performs poorly compared to Halide when compiling image processing pipelines, it is missing important optimizations.

Introduction

Compilers such as Lift aim to be domain-extensible



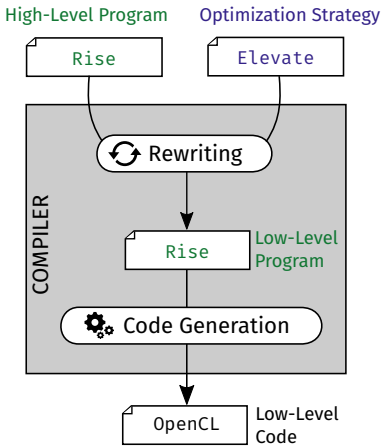
- \+ extensible set of abstractions and optimizations
- \- competitive with domain-specific compilers?



We extend a compiler for Rise (inspired from Lift) with well-known optimizations, outperforming Halide on our case study.

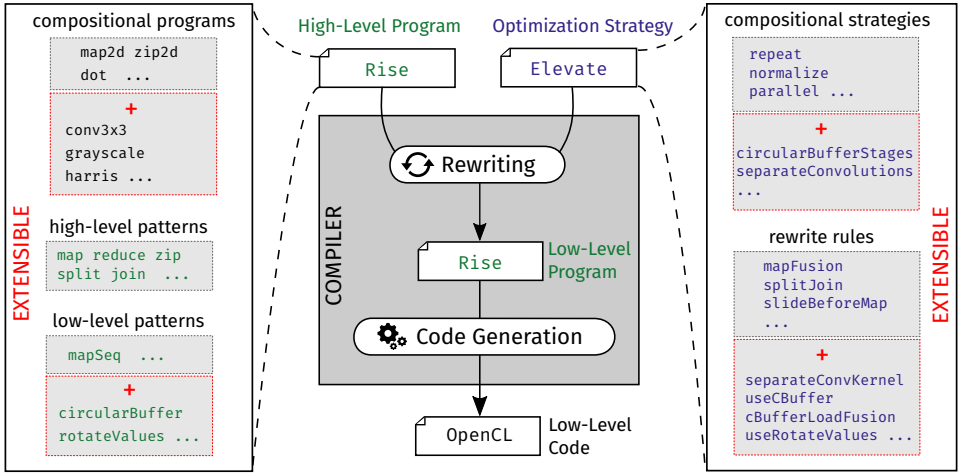
A Domain-Extensible Compiler Design

Overview



A Domain-Extensible Compiler Design

Overview



A Domain-Extensible Compiler Design

Dot product example

High-level Rise program

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

Elevate optimization strategy

```
strategy lowerDot = applyOnce(reduceMapFusion)

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

A Domain-Extensible Compiler Design

Dot product example

High-level Rise program

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

Elevate optimization strategy

```
strategy lowerDot = applyOnce(reduceMapFusion)

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

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

Rewriting

A Domain-Extensible Compiler Design

Dot product example

High-level Rise program

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

Elevate optimization strategy

```
strategy lowerDot = applyOnce(reduceMapFusion)

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

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

Rewriting

A Domain-Extensible Compiler Design

Dot product example

High-level Rise program

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

Elevate optimization strategy

```
strategy lowerDot = applyOnce(reduceMapFusion)

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

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

Rewriting

A Domain-Extensible Compiler Design

Dot product example

High-level Rise program

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

Elevate optimization strategy

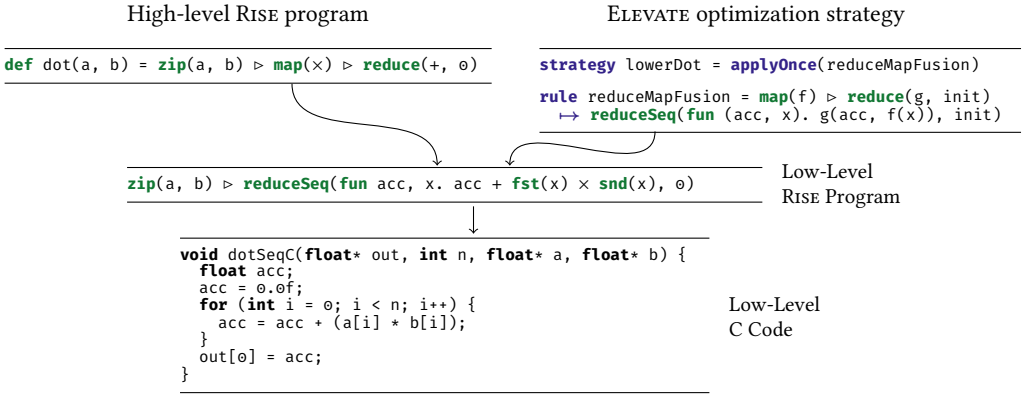
```
strategy lowerDot = applyOnce(reduceMapFusion)

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

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

Low-Level Rise Program

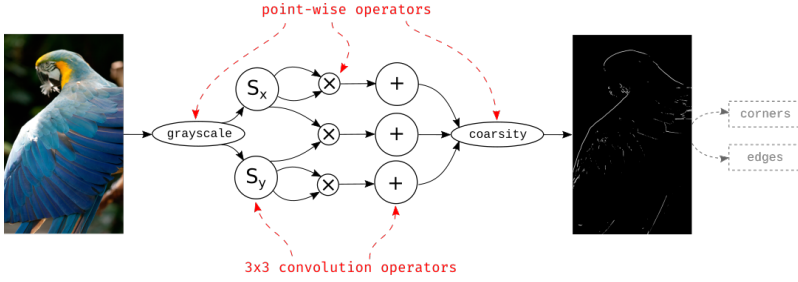
A Domain-Extensible Compiler Design

Dot product example



The Harris Operator

Our Case Study



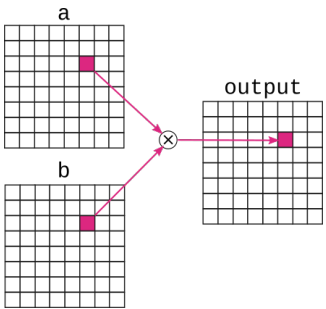
The Harris corner (and edge) detector is a well established image processing pipeline

The Harris Operator

In Rise

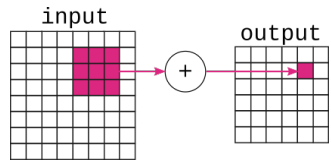
High-level point-wise operator

```
def ×2D(a, b: [n][m]f32): [n][m]f32 =
  zip2d(a, b) ▷ map2d(×)
```



High-level convolution operator

```
def +3×3: [n+2][m+2]f32 → [n][m]f32 =
  slide2d(3, 1, 3, 1) ▷ map2d(fun w. reduce(+, 0 ▷ join(w)))
```

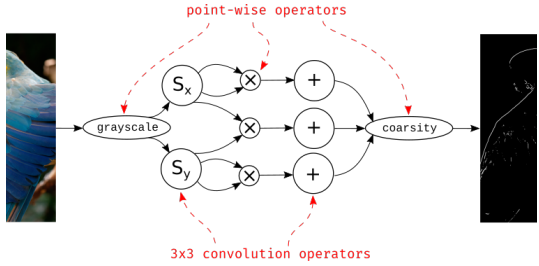


The Harris Operator

In Rise

High-level Harris operator

```
def harris(RGB: [3][n+4][m+4]f32): [n][m]f32 =
  def I = grayscale(RGB)
  def Ix = Sx(I)
  def Iy = Sy(I)
  def Ixx = ×2D(Ix, Ix)
  def Ixy = ×2D(Ix, Iy)
  def Iyy = ×2D(Iy, Iy)
  def Sxx = +3×3(Ixx)
  def Sxy = +3×3(Ixy)
  def Syy = +3×3(Iyy)
  coarsity(Sxx, Sxy, Syy, 0.04)
```



Optimizations

Halide Reference

CPU schedule for Harris

from the Halide GitHub repository

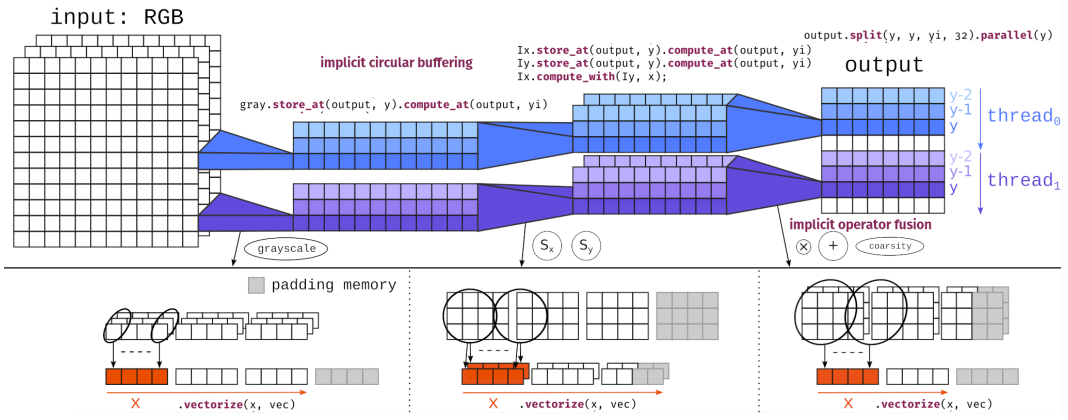
```
const int vec = natural_vector_size<float>();
output.split(y, y, yi, 32).parallel(y)
  .vectorize(x, vec);
gray.store_at(output, y).compute_at(output, yi)
  .vectorize(x, vec);
Ix.store_at(output, y).compute_at(output, yi)
  .vectorize(x, vec);
Iy.store_at(output, y).compute_at(output, yi)
  .vectorize(x,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

Simplified internal representation of lowered code

```
let t1226 = ((output.extent.1 + 31)/32)
parallel (output.s0.y.y, 0, t1226) {
 allocate gray[float32 * (output.extent.0 + 4) * 8]
 allocate Iy[float32 * t1247 * 4]
 allocate Ix[float32 * t1247 * 4]
 for (output.s0.y.yi, 0, 32) {
  for (gray.s0.y, gray.s0.y.min_2, gray.s0.y.loop_extent) {
   for (gray.s0.x.x, 0, t1265) {
    gray[ramp(((gray.s0.x.x*4) + t1268), 1, 4)] = [...] }}
  for (Iy.s0.fused.y, Iy.s0.y.min_2, t1269) {
   for (Iy.s0.x.fused.x, 0, t1251) {
    Iy[ramp(((Iy.s0.x.fused.x*4) + t1275), 1, 4)] = [...]
    Ix[ramp(((Iy.s0.x.fused.x*4) + t1275), 1, 4)] = [...] }}
  for (output.s0.x.x, 0, t1250) {
   output[ramp(((output.s0.x.x*4) + t1281), 1, 4)] = [...] }}
 free gray
 free Iy
 free Ix }
```

Optimizations

Overview



Optimizations

In Rise and Elevate

ELEVATE optimization strategy

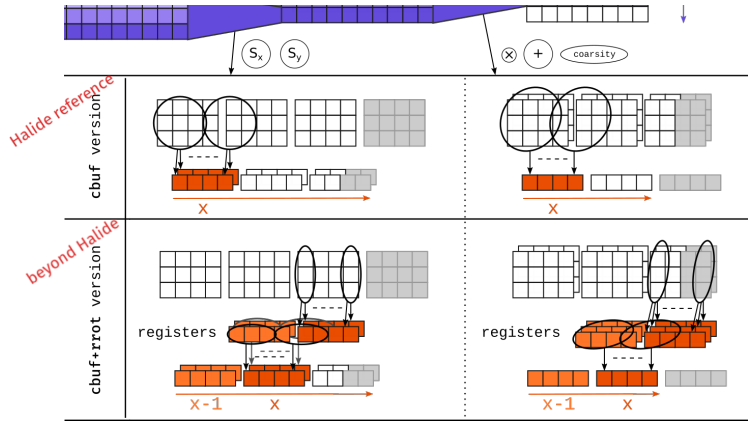
```
strategy cbufVersion =
  fuseOperators;
  splitPipeline(32); parallel;
  vectorizeReductions(vec);
  harrisIxWithIy;
  circularBufferStages;
  sequentialLines;
  usePrivateMemory; unrollReductions
```

Harris after applying `cbufVersion`

```
slide(32+4, 32) ▷ mapGlobal(
  circularBuffer(global, 3, grayLine) ▷
  circularBuffer(global, 3, sobelLine) ▷
  mapSeq(coarsityLine)
) ▷ join
```

Optimizations beyond Halide

Overview



Optimizations beyond Halide

In Rise and Elevate

ELEVATE optimization strategy

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

Typical 2D Convolution

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

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

Optimizations beyond Halide

In Rise and Elevate

ELEVATE optimization strategy

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

Typical 2D Convolution

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

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

```
nbhV ▷ map(slide(3,1)) ▷ transpose ▷ map(fun nbh2d.
    nbh2d ▷ transpose ▷ map(dot(wV)) ▷ dot(wH))
```

Optimizations beyond Halide

In Rise and Elevate

ELEVATE optimization strategy

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

Typical 2D Convolution

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

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

```
nbhV ▷ map(slide(3,1)) ▷ transpose ▷ map(fun nbh2d.
    nbh2d ▷ transpose ▷ map(dot(wV)) ▷ dot(wH))
```

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

Optimizations beyond Halide

In Rise and Elevate

Elevate optimization strategy

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

Typical 2D Convolution

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

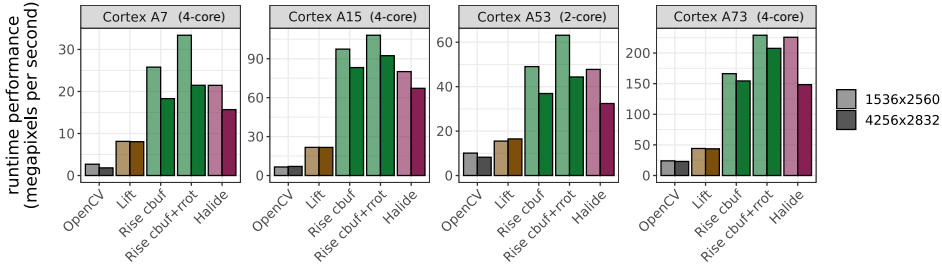
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

```
nbhV ▷ map(slide(3,1)) ▷ transpose ▷ map(fun nbh2d.
    nbh2d ▷ transpose ▷ map(dot(wV)) ▷ dot(wH))
```

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

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

Experimental Evaluation



- All compilers outperform the OpenCV library: Rise by up to 16×
- Rise outperforms Lift by up to 4.5×
- Rise `cbuf` is roughly on par with Halide
- Rise `cbuf+rrot` is faster than Halide by up to 40%

Conclusion

Harris Operator case study on ARM CPUs

- We reproduced an optimized Halide schedule by defining compositional Elevate optimization strategies; by extending and re-using Rise patterns.
- The achieved performance is on par with the highly optimized Halide compiler, which is specifically built for image processing pipelines.
- We reached higher performance through additional optimizations that cannot be expressed in a Halide schedule, showing the benefit of compiler extensibility.

Conclusion

Harris Operator case study on ARM CPUs

- We reproduced an optimized Halide schedule by defining compositional Elevate optimization strategies; by extending and re-using Rise patterns.
- The achieved performance is on par with the highly optimized Halide compiler, which is specifically built for image processing pipelines.
- We reached higher performance through additional optimizations that cannot be expressed in a Halide schedule, showing the benefit of compiler extensibility.

Thanks!

Paper: thok.eu/publications/2021/cgo.pdf
Artifact: github.com/rise-lang/2021-CGO-artifact

---

thomas.koehler@thok.eu
thok.eu

rise-lang.org
elevate-lang.org