

Optimizing Processing Pipelines with a Rewrite-Based Domain-Extensible Compiler

Thomas K  HLER

Michel STEUWER



University
of Glasgow



THE UNIVERSITY
of EDINBURGH

Languages, Systems, and Data Seminar — November 2021

Domain-Agnostic Compilers

Some compilers are domain-agnostic:

- + generic program abstractions and optimizations
- + compile programs from any domain (turing complete)
- no automation of domain-specific optimizations
- manual optimization takes months and risks introducing bugs



Domain-Specific Compilers

Some compilers are domain-specific:

- + 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);
```

Domain-Specific Compilers

Some compilers are domain-specific:

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



Halide Development Roadmap #5055



abadams opened this issue on Jun 19 · 44 comments

- How do we make Halide easier to use for researchers wanting to cannibalize it, extend it, or compare to it?
- How do we make Halide more useful on current and upcoming hardware?
- How do we make Halide more useful for new types of application?

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

Domain-Extensible Compilers

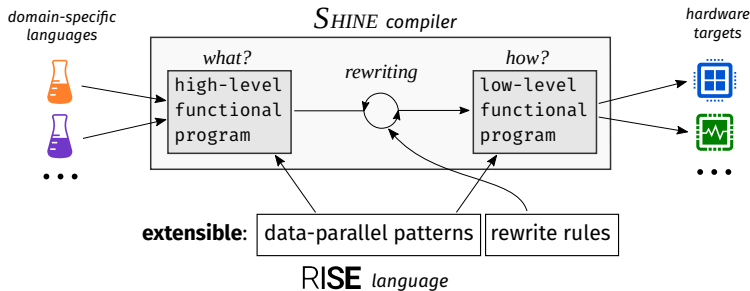
Compilers should be domain-extensible:

- + extensible set of abstractions and optimizations

Domain-Extensible Compilers

Compilers should be domain-extensible:

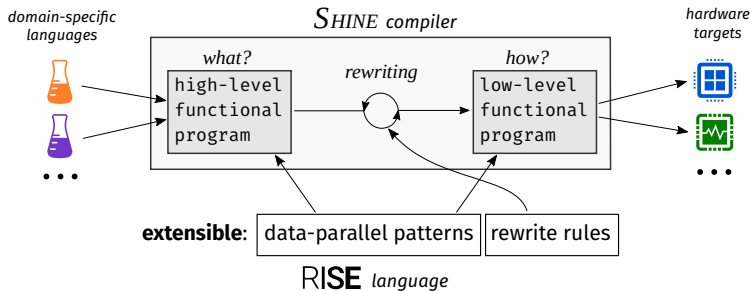
- + extensible set of abstractions and optimizations



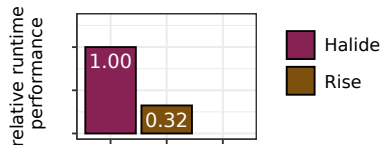
Domain-Extensible Compilers

Compilers should be domain-extensible:

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



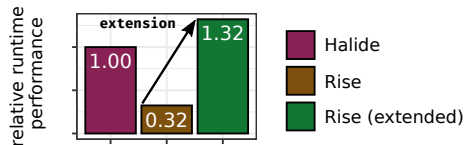
Is RISE Competitive with Domain-Specific Compilers?



important image processing pipeline optimizations are missing

[Koehler and Steuwer 2021 “Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs”]

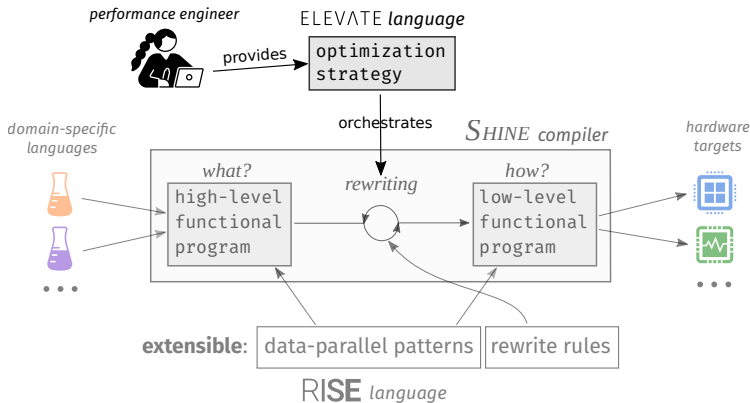
Is RISE Competitive with Domain-Specific Compilers?



*6 well-known image processing pipeline optimizations can be encoded as compositions of **RISE** rewrite rules*

[Koehler and Steuwer 2021 “Towards a Domain-Extensible Compiler: Optimizing an Image Processing Pipeline on Mobile CPUs”]

Orchestrating Compositions of Rewrite Rules



[Hagedorn et al. 2020 “Achieving high-performance the functional way: a functional pearl on expressing high-performance optimizations as rewrite strategies”]

Compilation Example

dot product

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)
```

Compilation Example

dot product

High-level RISE program

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

Rewriting

Compilation Example

dot product

High-level RISE program

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

Rewriting

Compilation Example

dot product

High-level RISE program

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

Rewriting

Compilation Example


dot product

High-level RISE program

```
def dot(a, b) = zip(a, b) ▷ map(x) ▷ 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

Compilation Example

dot product

High-level RISE program

```
def dot(a, b) = zip(a, b) ▷ map(x) ▷ 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)
```

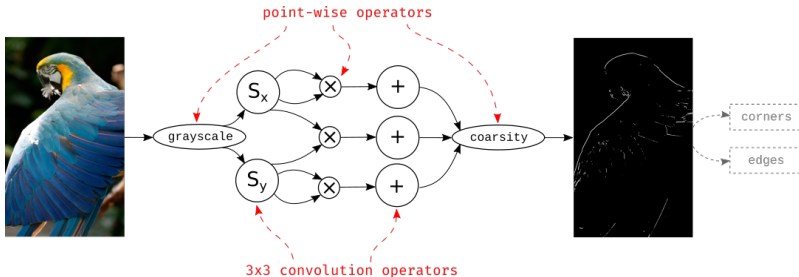
```
zip(a, b) ▷ reduceSeq(fun acc, x. acc + fst(x) × snd(x), 0)
```

Low-Level
RISE Program

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

Low-Level
C Code

Harris Case Study



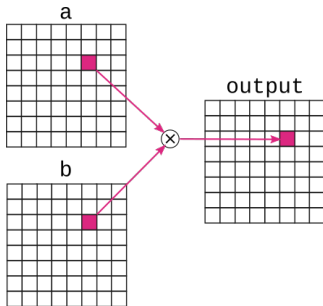
The Harris corner (and edge) detector is a well established image processing pipeline

How do we represent these operators in RISE?

Harris Case Study

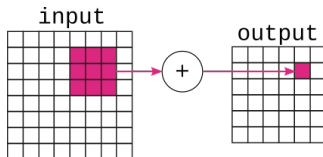
High-level point-wise operator

```
def  $\times_{2D}$ (a, b: [n] [m] f32): [n] [m] f32 =  
  zip2d(a, b)  $\triangleright$  map2d( $\times$ )
```



High-level convolution operator

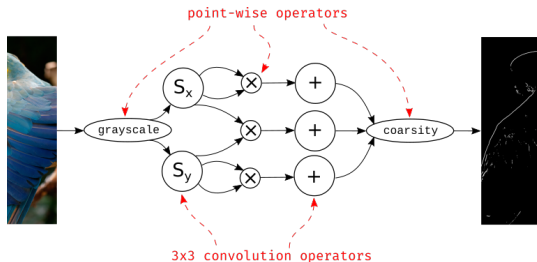
```
def  $+_{3\times 3}$ : [n+2] [m+2] f32  $\rightarrow$  [n] [m] f32 =  
  slide2d(3, 1, 3, 1)  $\triangleright$  map2d(fun w. reduce(+,  $\odot$  join(w)))
```



Harris Case Study

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)
```



Reference Optimizations

CPU schedule for Harris

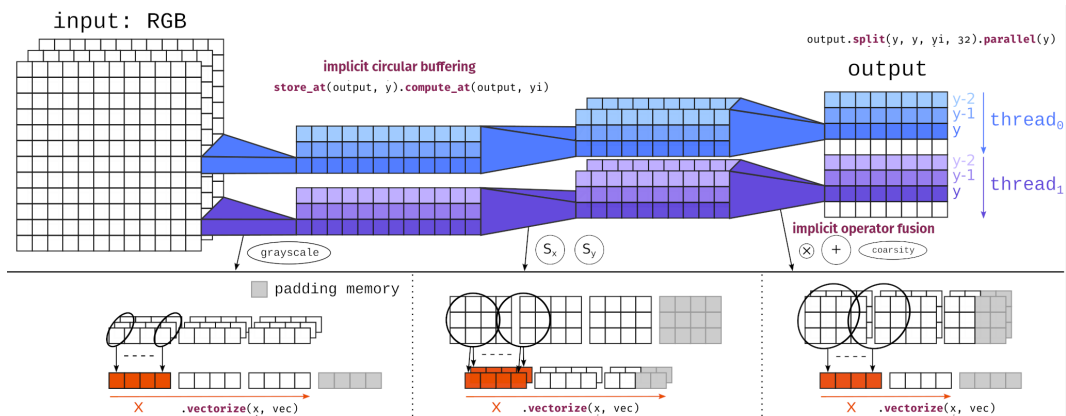
from the Halide GitHub repository

```
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 }
```

Reference Optimizations



Reference Optimizations

In RISE and ELEVATE

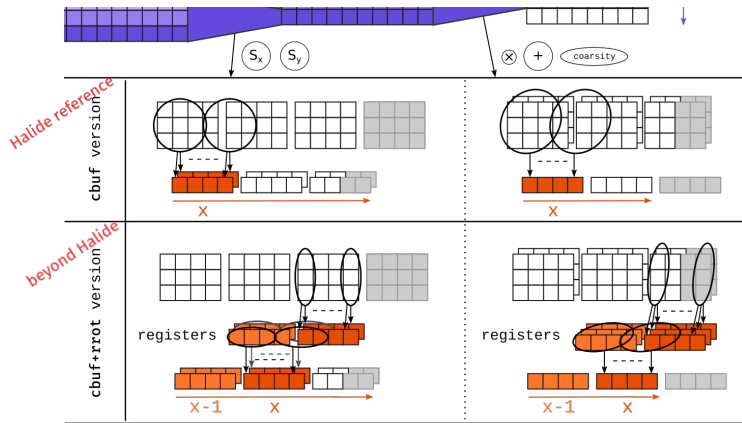
ELEVATE optimization strategy

```
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



Optimizations beyond Halide

In RISE and ELEVATE

ELEVATE optimization strategy

```
strategy cbuf+rrrotVersion =  
  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+rrrotVersion =  
  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+rrrotVersion =  
  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+rrrotVersion =  
  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)))
```

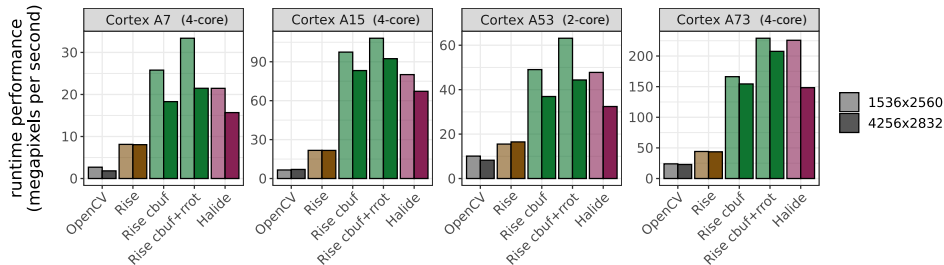
$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} \begin{bmatrix} -1 & 0 & 1 \end{bmatrix}$$

```
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** improved by up to 4.5×
- ▶ **RISE** cbuf is roughly on par with Halide
- ▶ **RISE** cbuf+rrot is faster than Halide by up to 40%

Summary

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.

Summary

Harris Operator case study on ARM CPUs

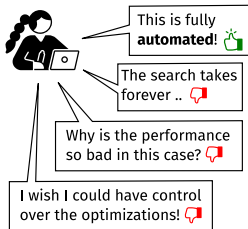
- ▶ We reproduced an optimized Halide schedule by defining compositional ELEVATE optimization strategies; by extending and re-using **RISE** patterns.
- ▶ The achieved performance is on par with the highly optimized Halide compiler, which is specifically built for image processing pipelines.
- ▶ We reached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

But, ELEVATE optimization strategies are difficult to write!

Deciding how to Apply Rewrite Rules

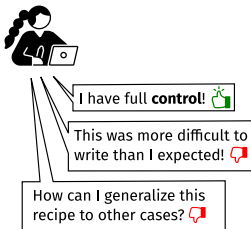
Fully automated search?

e.g. heuristic search,
equality saturation, ...

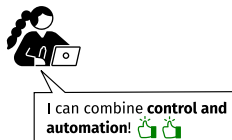


Manually written recipe?

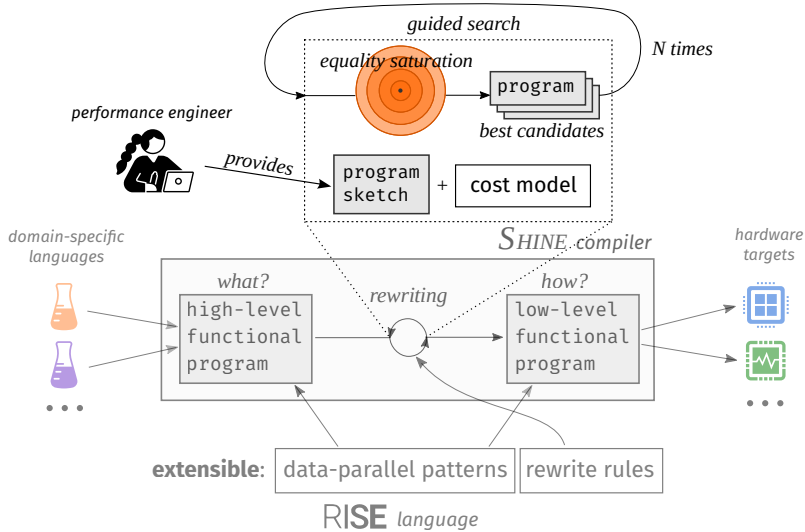
e.g. Halide/TVM schedules,
Elevate strategies, ...



Guided search!



Guided Equality Saturation via Sketching



Optimizing Matrix Multiplication with Sketching

```
map (fun aRow. map (fun bCol. dot aRow bCol) transpose b) a
```

→*

```
join (map (map join) (map transpose
  map (map fun x2.
    reduceSeq (fun x3. fun x4.
      reduceSeq (fun x5. fun x6.
        map (map (fun x7. (fst x7) +
          (fst (snd x7)) × (snd (snd x7)))
          (map (fun x7. zip (fst x7) (snd x7)) (zip x5 x6)))
        (transpose (map transpose
          (snd (unzip (map unzip map (fun x5.
            zip (fst x5) (snd x5))
            (zip x3 x4)))))))
        (generate (fun x3. generate (fun x4. 0)))
        transpose (map transpose x2))
      (map (map (map (map (split 4))))
        (map transpose
          (map (map (fun x2. map (map (zip x2)
            (split 32 (transpose b))))
            split 32 a)))))))
```

```
containsMap(m /32,
  containsMap(n /32,
    containsReduceSeq(k /4,
      containsReduceSeq(4,
        containsMap(32,
          containsMap(32, ?))))))
```

- 0 intermediate sketch: not found after minutes **X**

Optimizing Matrix Multiplication with Sketching

```
map (fun aRow. map (fun bCol. dot aRow bCol) transpose b) a
```

→*

```
join (map (map join) (map transpose
  map (map fun x2.
    reduceSeq (fun x3. fun x4.
      reduceSeq (fun x5. fun x6.
        map (map (fun x7. (fst x7) +
          (fst (snd x7)) × (snd (snd x7)))
          (map (fun x7. zip (fst x7) (snd x7)) (zip x5 x6)))
        (transpose (map transpose
          (snd (unzip (map unzip map (fun x5.
            zip (fst x5) (snd x5))
            (zip x3 x4)))))))
        (generate (fun x3. generate (fun x4. 0)))
        transpose (map transpose x2))
      (map (map (map (map (split 4))))
        (map transpose
          (map (map (fun x2. map (map (zip x2)
            (split 32 (transpose b))))
            split 32 a)))))))
```

```
containsMap(m /32,
  containsMap(32,
    containsMap(n /32,
      containsMap(32,
        containsReduceSeq(k /4,
          containsReduceSeq(4, ?))))))
```

```
containsMap(m /32,
  containsMap(n /32,
    containsReduceSeq(k /4,
      containsReduceSeq(4,
        containsMap(32,
          containsMap(32, ?))))))
```

- 1 intermediate sketch: found in seconds ✓

Conclusion

- ▶ We encode 6 well-known image processing pipeline optimizations as compositions of rewrite rules.
- ▶ We present guided equality saturation via sketching, to offer novel trade-offs between precise control and full automation of optimizations.

Conclusion

- ▶ We encode 6 well-known image processing pipeline optimizations as compositions of rewrite rules.
- ▶ We present guided equality saturation via sketching, to offer novel trade-offs between precise control and full automation of optimizations.

✉ thomas.koehler@thok.eu

🌐 thok.eu

Thanks!

We are open to collaboration!

🌐 rise-lang.org

🌐 elevate-lang.org

Equality Saturation

Which rewrite rule should be applied when, and where?

Explore all possibilities

[Tate et al. 2009 “Equality saturation: a new approach to optimization”]

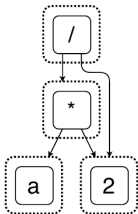
[Willsey et al. 2021 “egg: fast and extensible equality saturation”]

- + No need to decide which rewrite to apply next,
Decide which program variant you want in the end.
- Need to efficiently represent and rewrite many programs.

Equality Saturation

E-Graphs

$$(a * 2) / 2 \longrightarrow^* a$$

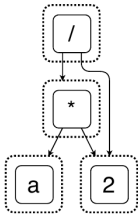


$$(a * 2) / 2$$

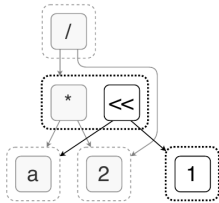
Equality Saturation

E-Graphs

$$(a * 2) / 2 \longrightarrow^* a$$



$$(a * 2) / 2$$

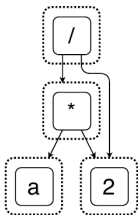


$$x * 2 \longrightarrow x \ll 1$$

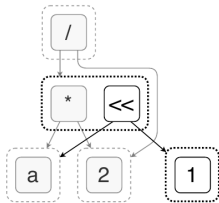
Equality Saturation

E-Graphs

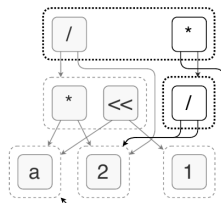
$$(a * 2) / 2 \longrightarrow^* a$$



$$(a * 2) / 2$$



$$x * 2 \longrightarrow x \ll 1$$

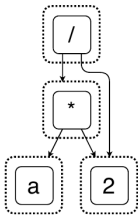


$$(x * y) / z \longrightarrow x * (y / z)$$

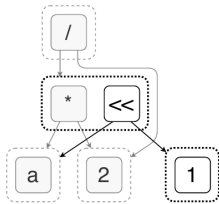
Equality Saturation

E-Graphs

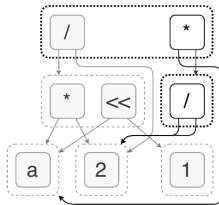
$$(a * 2)/2 \longrightarrow^* a$$



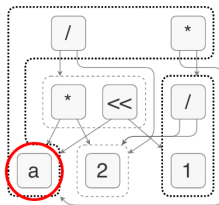
$$(a * 2)/2$$



$$x * 2 \longrightarrow x \ll 1$$



$$(x * y) / z \longrightarrow x * (y / z)$$



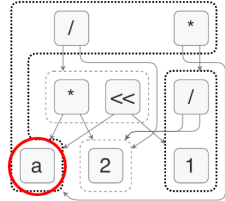
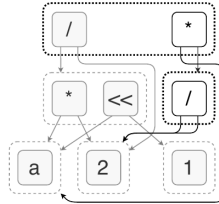
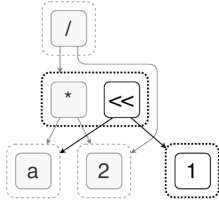
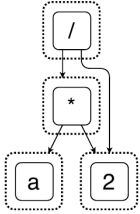
$$x / x \longrightarrow 1$$

$$x * 1 \longrightarrow x$$

Equality Saturation

E-Graphs

$$(a * 2) / 2 \longrightarrow^* a$$

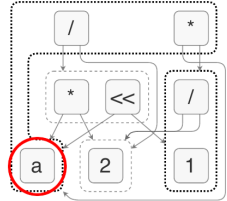
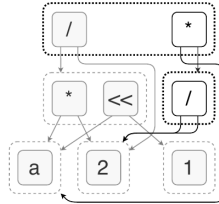
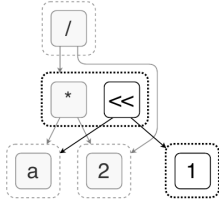
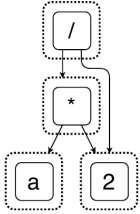


Congruence invariant: $a = b \implies f(a) = f(b)$

Equality Saturation

E-Graphs

$$(a * 2) / 2 \longrightarrow^* a$$



How does it work for functional programs?

Equality Saturation for RISE

$(\lambda x. b) e \longrightarrow b[e/x]$ (β -reduction)

$\lambda x. f x \longrightarrow f$ if x not free in f (η -reduction)

$\text{map } f (\text{map } g \text{ arg}) \longrightarrow \text{map } (\lambda x. f (g x)) \text{ arg}$ (map-fusion)

$\text{map } (\lambda x. f gx) \longrightarrow \lambda y. \text{map } f (\text{map } (\lambda x. gx) y)$ if x not free in f (map-fission)

How can we implement **substitution**, **predicates** and **name bindings**?

- ▶ State-of-the-art is very inefficient, trivial optimizations are out of reach.
- ▶ We made **substitution** order of magnitudes more efficient using a practical approximation.
- ▶ We made **name bindings** order of magnitudes more efficient using DeBruijn indices.