Optimizing Image Processing Pipelines with a Domain-Extensible Compiler

Thomas Kœhler Michel Steuwer

Intra-Systems Seminar — October 2020

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

Halide Development Roadmap #5055

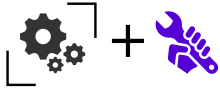
Open **abadams** opened this issue on Jun 19 · 44 comments

- How do we keep Halide maintainable over time?
- 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?

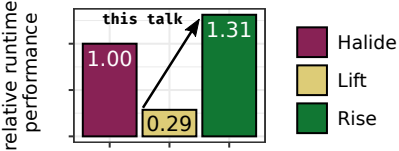
https://github.com/halide/Halide/issues/5055

Introduction

Domain-extensible compilers such as Lift



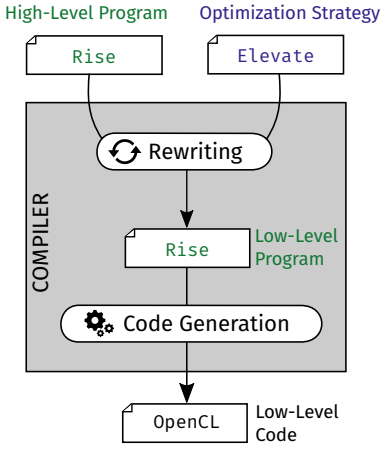
+ extensible set of abstractions and optimizations
- competitive with domain-specific compilers?



Well-known image processing optimizations are missing in both Lift and Halide.

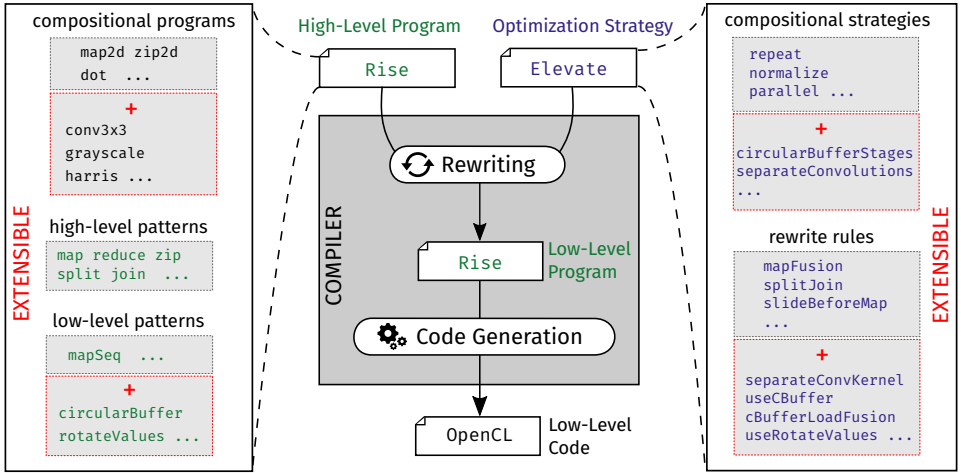
Rise and Elevate

Overview



Rise and Elevate

Overview



Rise and Elevate

Dot product example

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)
```

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

Low-Level Rise Program

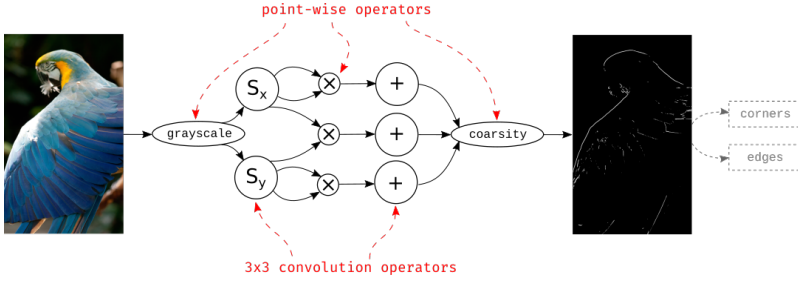
↓

```
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

The Harris Operator

Our Case Study



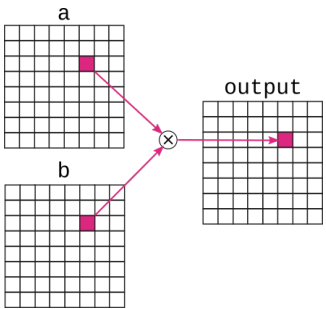
The Harris corner (and edge) detector is a well established image processing pipeline

The Harris Operator

In Rise

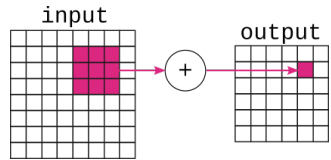
High-level point-wise operator

```
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)))
```



Optimizations

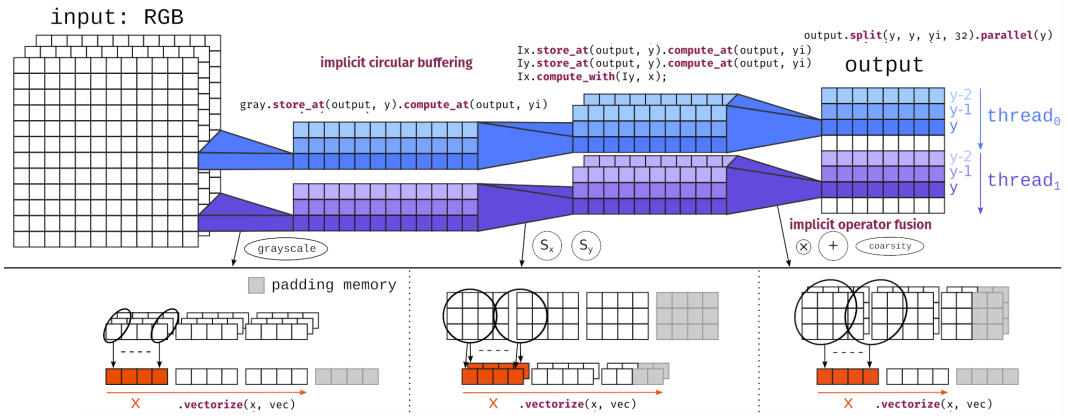
Halide Reference

Harris schedule 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);
```

Optimizations

Halide Reference



Optimizations

In Rise and Elevate

Harris after `fuseOperators`

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

ELEVATE optimization strategy

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

Optimizations

In Rise and Elevate

Harris during `splitPipelines(32)`

```
map(grayLine)  ▷ slide(3,1) ▷
map(sobelLine) ▷ slide(3,1) ▷
split(32) ▷
map(map(coarsityLine)) ▷
join
```

ELEVATE optimization strategy

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

Optimizations

In Rise and Elevate

Harris after `splitPipelines(32)`

```
slide(32+4, 32) ▷ map(
  map(grayLine)  ▷ slide(3,1) ▷
  map(sobelLine) ▷ slide(3,1) ▷
  map(coarsityLine)
) ▷ join
```

ELEVATE optimization strategy

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

Optimizations

In Rise and Elevate

Harris after `parallel`

```
slide(32+4, 32) ▷ mapGlobal(
  map(grayLine)  ▷ slide(3,1) ▷
  map(sobelLine) ▷ slide(3,1) ▷
  map(coarsityLine)
) ▷ join
```

Elevate optimization strategy

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

Optimizations

In Rise and Elevate

Harris after `circularBufferStages`

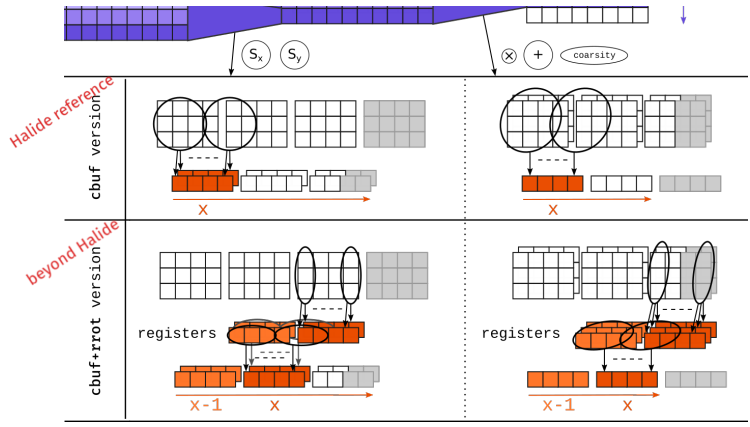
```
slide(32+4, 32) ▷ mapGlobal(
  circularBuffer(global, 3, grayLine) ▷
  circularBuffer(global, 3, sobelLine) ▷
  mapSeq(coarsityLine)
) ▷ join
```

Elevate optimization strategy

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

Optimizations beyond Halide

Overview



Optimizations beyond Halide

In Rise and Elevate

Convolution before `separateConvolutions`

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

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

ELEVATE optimization strategy

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

Optimizations beyond Halide

In Rise and Elevate

Convolution during `separateConvolutions`

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

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

ELEVATE optimization strategy

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

Optimizations beyond Halide

In Rise and Elevate

Convolution after `separateConvolutions`

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

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

ELEVATE optimization strategy

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

Optimizations beyond Halide

In Rise and Elevate

Convolution after `rotateValuesAndConsumeLines`

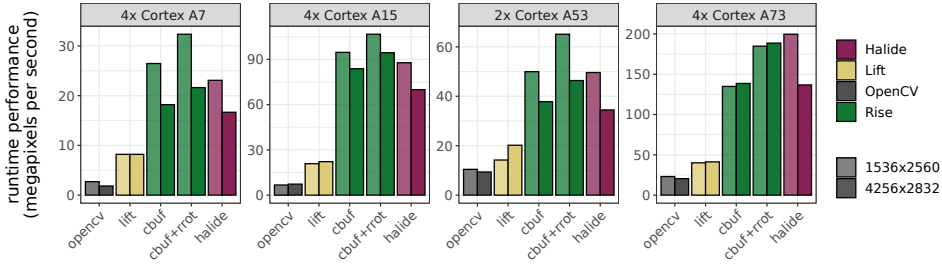
```
nbhV ▷ transpose ▷ map(dot(wV))
     ▷ rotateValues(private, 3) ▷ mapSeq(dot(wH))
```

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

ELEVATE optimization strategy

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

Experimental Evaluation



- All compilers outperform the OpenCV library *(Rise by up to 16×)*
- Rise outperforms Lift by up to 4.5×
- Without register rotation, Rise is on par with Halide
- With register rotation, Rise is faster than Halide by 30-40%

Conclusion

Harris Operator case study on ARM CPUs

- Our domain-extensible compiler can reproduce the optimized Halide schedule with user-defined optimization strategies.
- The achieved performance is on par with the highly optimized Halide compiler, which is specifically built for image processing pipelines.
- Our compiler is able to reach higher performance through additional optimizations that are not expressible in a Halide schedule, showing the benefit of extensibility.

Conclusion

Harris Operator case study on ARM CPUs

- Our domain-extensible compiler can reproduce the optimized Halide schedule with user-defined optimization strategies.
- The achieved performance is on par with the highly optimized Halide compiler, which is specifically built for image processing pipelines.
- Our compiler is able to reach higher performance through additional optimizations that are not expressible in a Halide schedule, showing the benefit of extensibility.

---

thomas.koehler@thok.eu
thok.eu

Thanks!

rise-lang.org
elevate-lang.org