Autotuning **Halide schedules** with OpenTuner

Jonathan Ragan-Kelley
(Stanford)

We are surrounded by computational cameras

Enormous opportunity,
demands extreme optimization

parallelism & locality limit
performance and energy

We are surrounded by computational cameras

Enormous opportunity,
demands extreme optimization
parallelism & locality limit
performance and energy

Camera: 8 Mpixels
(96MB/frame as *float*)

CPUs: 15 GFLOP/sec

GPU: 115 GFLOP/sec

We are surrounded by computational cameras

Enormous opportunity,
demands extreme optimization
parallelism & locality limit
performance and energy

Camera: 8 Mpixels
(96MB/frame as *float*)

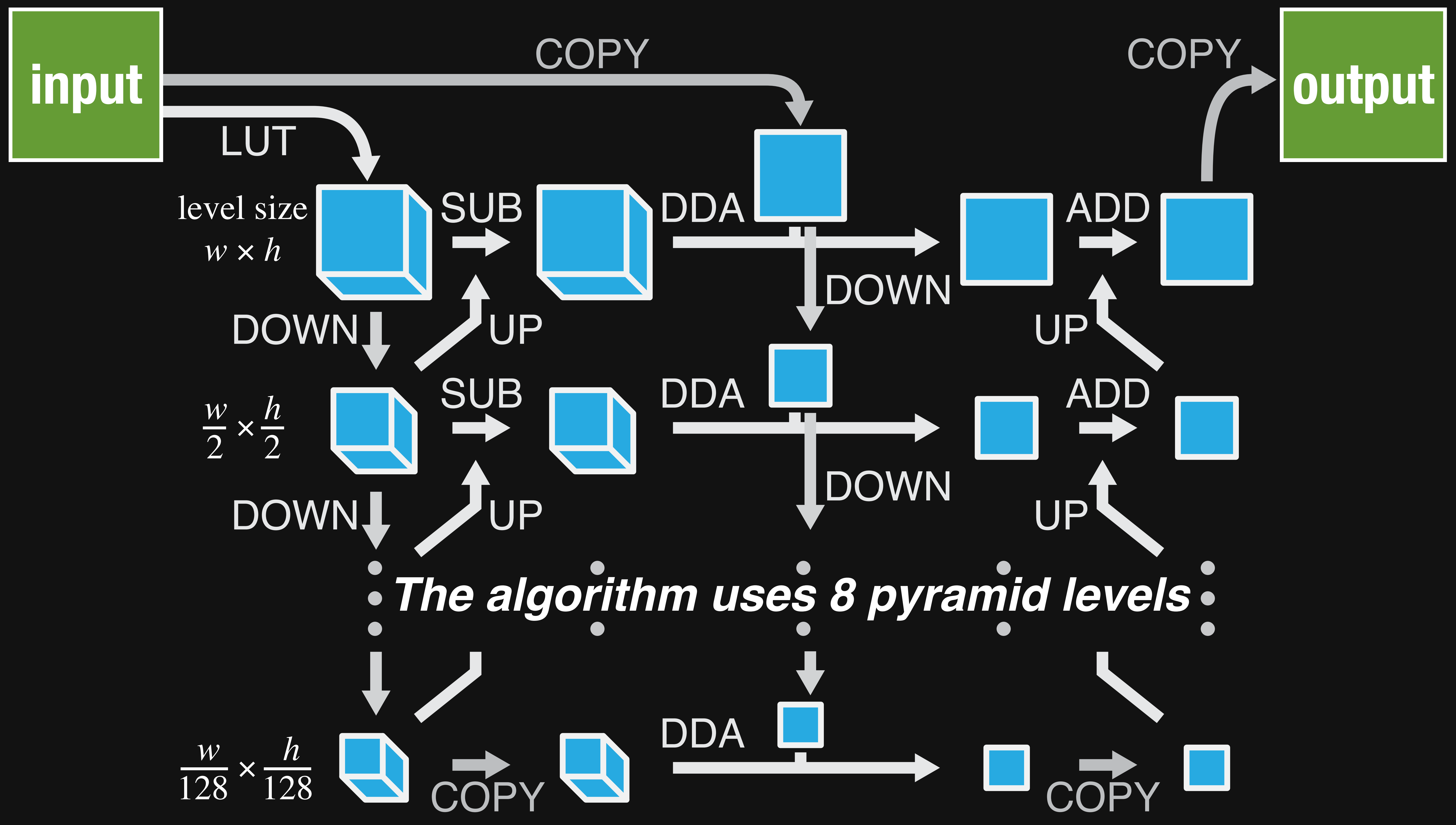
CPUs: 15 GFLOP/sec

GPU: 115 GFLOP/sec

Required arithmetic intensity > 40:1

A realistic pipeline: Local Laplacian Filters

[Paris et al. 2010, Aubry et al. 2011]



LUT: look-up table

$O(x,y,k) \leftarrow \mathrm{lut}(I(x,y) - k\sigma)$

ADD: addition

$O(x,y) \leftarrow I_1(x,y) + I_2(x,y)$

SUB: subtraction

$O(x,y) \leftarrow I_1(x,y) - I_2(x,y)$

UP: upsample

$T_1(2x,2y) \leftarrow I(x,y)$

$T_2 \leftarrow T_1 \otimes_x [1\ 3\ 3\ 1]$

$O \leftarrow T_2 \otimes_y [1\ 3\ 3\ 1]$

DOWN: downsample

$T_1 \leftarrow I \otimes_x [1\ 3\ 3\ 1]$

$T_2 \leftarrow T_1 \otimes_y [1\ 3\ 3\ 1]$

$O(x,y) \leftarrow T_2(2x,2y)$

DDA: data-dependent access

$k \leftarrow \mathrm{floor}(I_1(x,y) / \sigma)$

$\alpha \leftarrow (I_1(x,y) / \sigma) - k$

$O(x,y) \leftarrow (1-\alpha)\, I_2(x,y,k) + \alpha\, I_2(x,y,k+1)$

wide, deep, heterogeneous

stencils + stream processing

Local Laplacian Filters

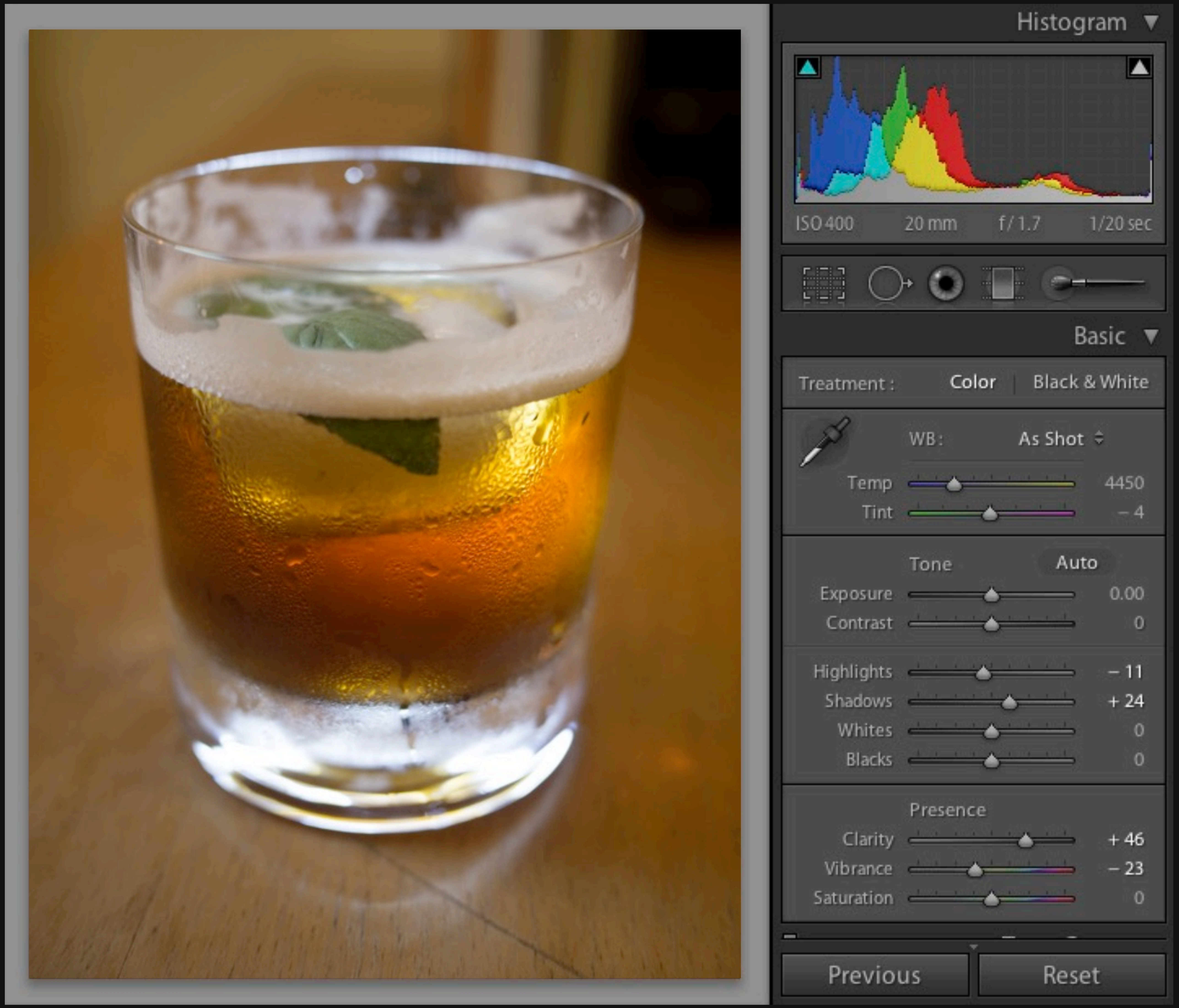
in Adobe Photoshop Camera Raw / Lightroom

1500 lines of expert-optimized C++

multi-threaded, SSE

3 months of work

10x faster *than reference C*



Local Laplacian Filters

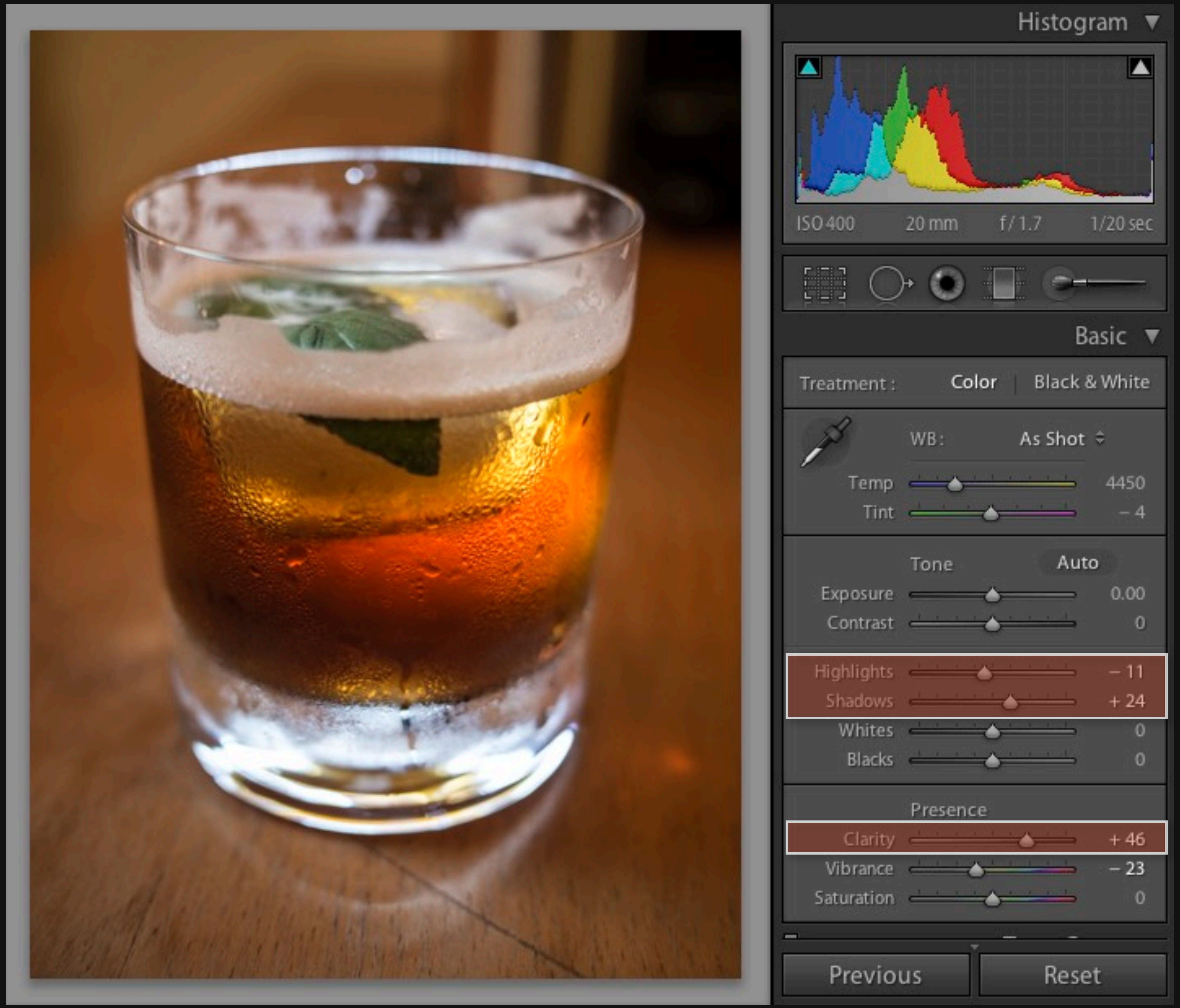
in Adobe Photoshop Camera Raw / Lightroom

1500 lines of expert-optimized C++

multi-threaded, SSE

3 months of work

10x faster *than reference C*



Local Laplacian Filters

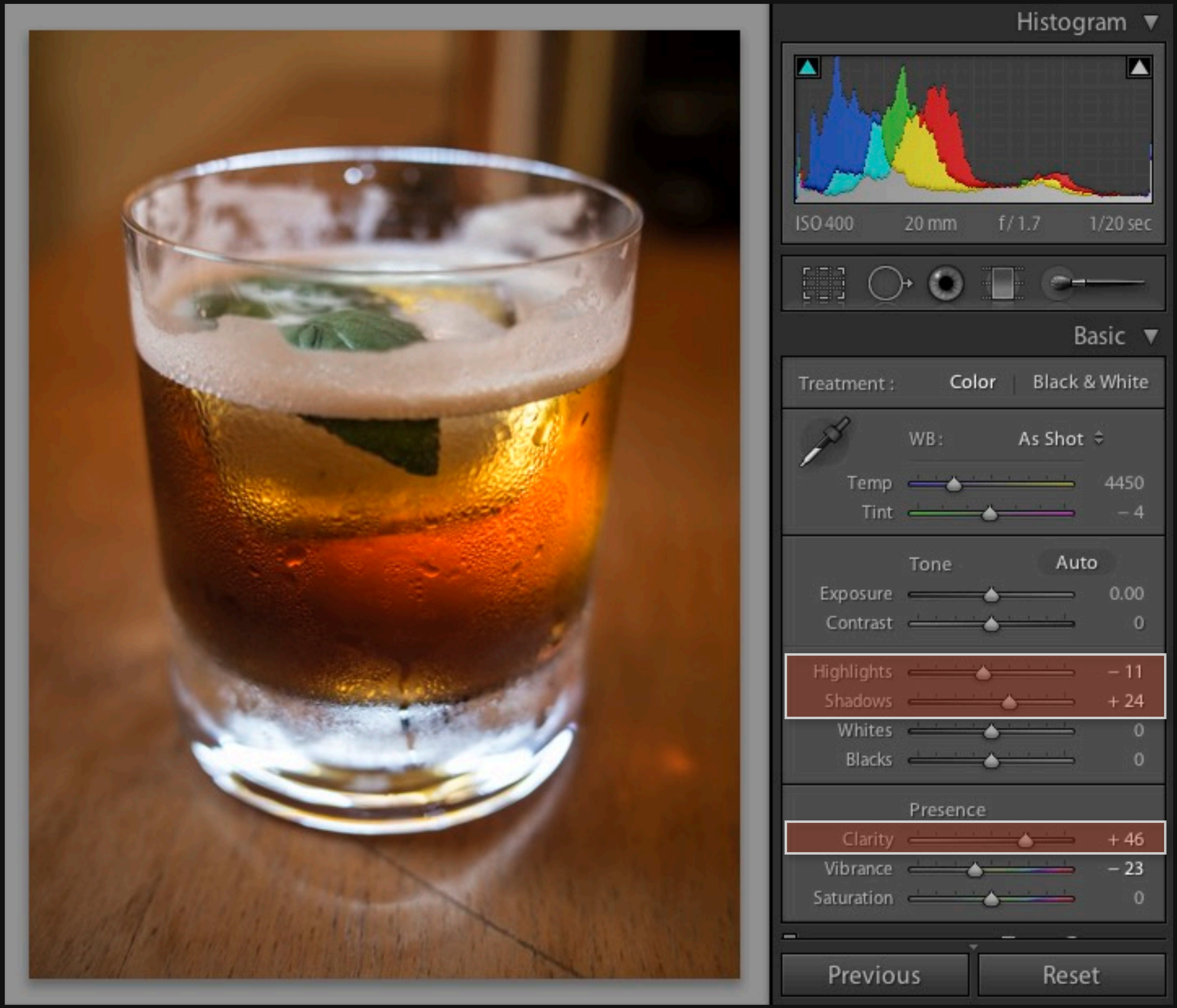
in Adobe Photoshop Camera Raw / Lightroom

1500 lines of expert-optimized C++

multi-threaded, SSE

3 months of work

***10x faster** than reference C*



Local Laplacian Filters

in Adobe Photoshop Camera Raw / Lightroom

Adobe

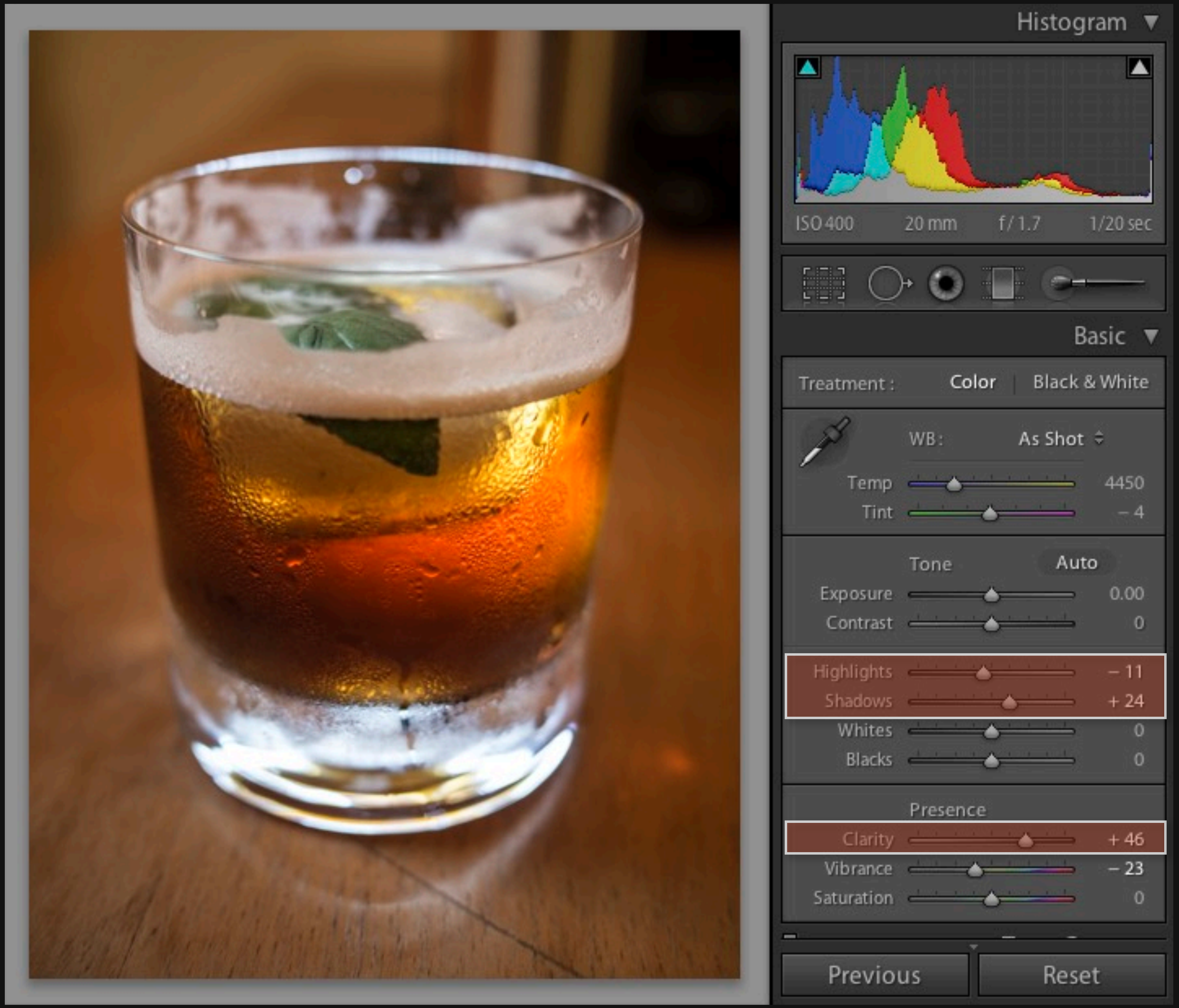
1500 lines of expert-optimized C++

multi-threaded, SSE

3 months of work

10x faster *than reference C*

2x slower *than another organization (which they couldn't find)*



Halide

a new language & compiler for image processing

Halide

a new language & compiler for image processing

1. **Decouple *algorithm* from *schedule***

   Algorithm: *what* is computed
   Schedule: *where* and *when* it's computed

Halide

a new language & compiler for image processing

1. Decouple *algorithm* from *schedule*

Algorithm: *what* is computed
Schedule: *where* and *when* it's computed

we want to autotune this

The algorithm defines pipelines as pure functions

Pipeline stages are *functions* from coordinates to values

Execution order and storage are unspecified

The algorithm defines pipelines as pure functions

Pipeline stages are *functions* from coordinates to values

Execution order and storage are unspecified

3x3 blur as a Halide *algorithm*:

```
blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
```

Halide

a new language & compiler for image processing

1. **Decouple *algorithm* from *schedule***

   Algorithm: *what* is computed

   Schedule: *where* and *when* it's computed

Halide

a new language & compiler for image processing

1. **Decouple *algorithm* from *schedule***

    Algorithm: *what* is computed
    Schedule: *where* and *when* it's computed

2. **Single, unified model for *all* schedules**

Halide

a new language & compiler for image processing

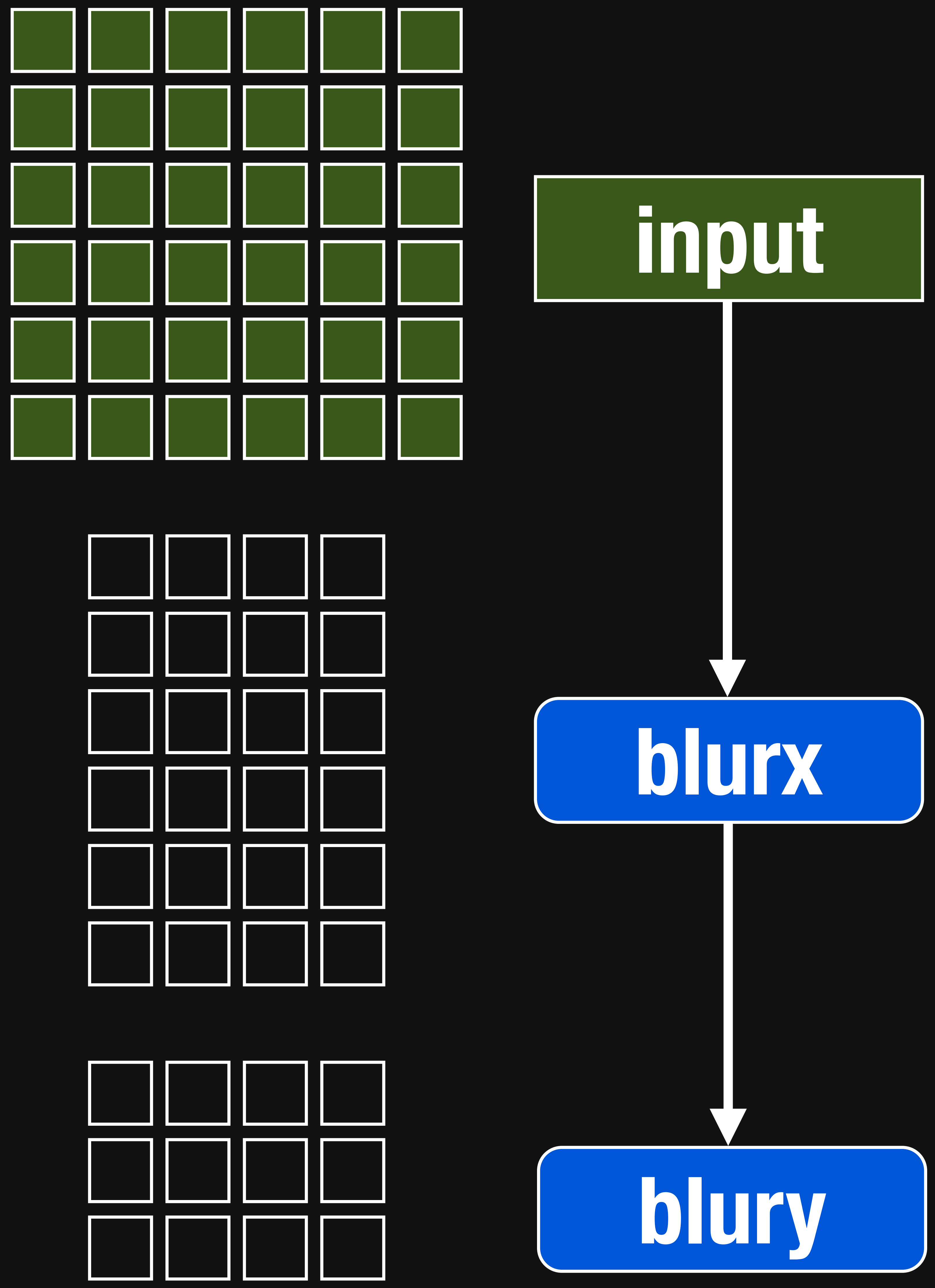
1. Decouple *algorithm* from *schedule*

Algorithm: *what* is computed
Schedule: *where* and *when* it's computed

2. Single, unified model for *all* schedules

Simple enough to search, expose to user
Powerful enough to beat expert-tuned code

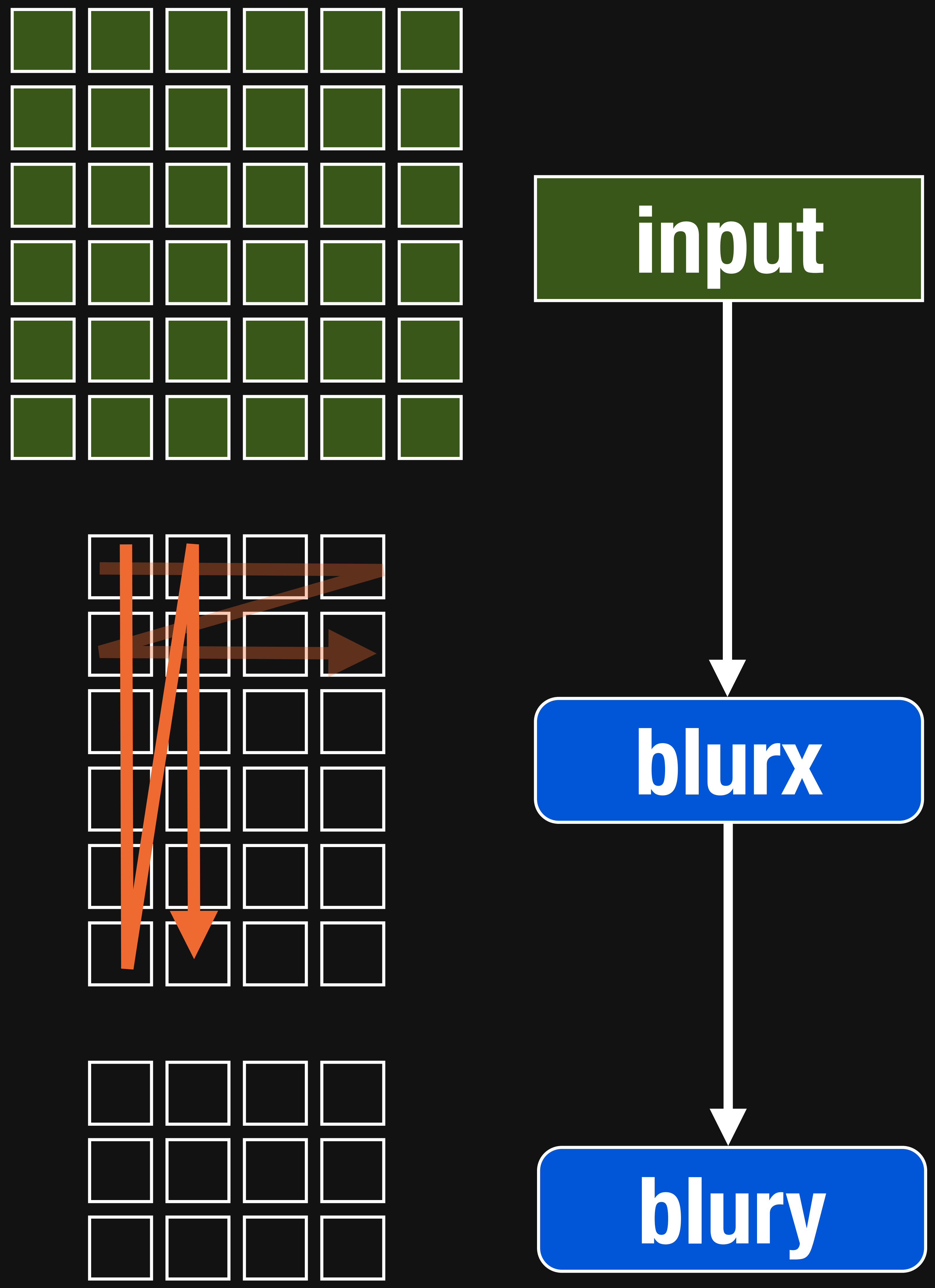
The schedule defines intra-stage order, inter-stage interleaving



The schedule defines intra-stage order, inter-stage interleaving

For each stage:

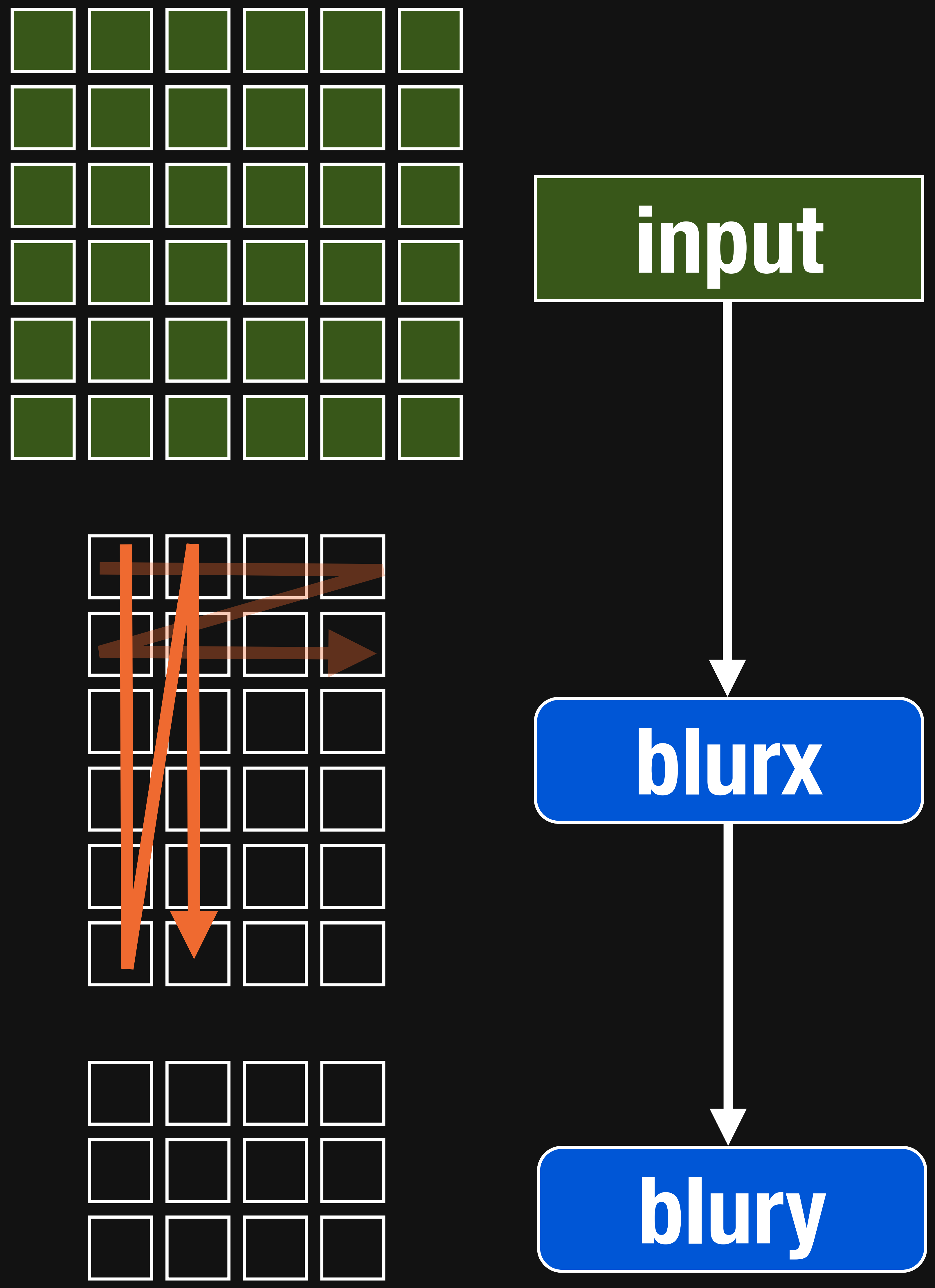
1) In **what order** should we **compute** its **values**?



The schedule defines intra-stage order, inter-stage interleaving

For each stage:

1) In **what order** should we **compute** its **values**?
split, tile, reorder, vectorize, unroll loops

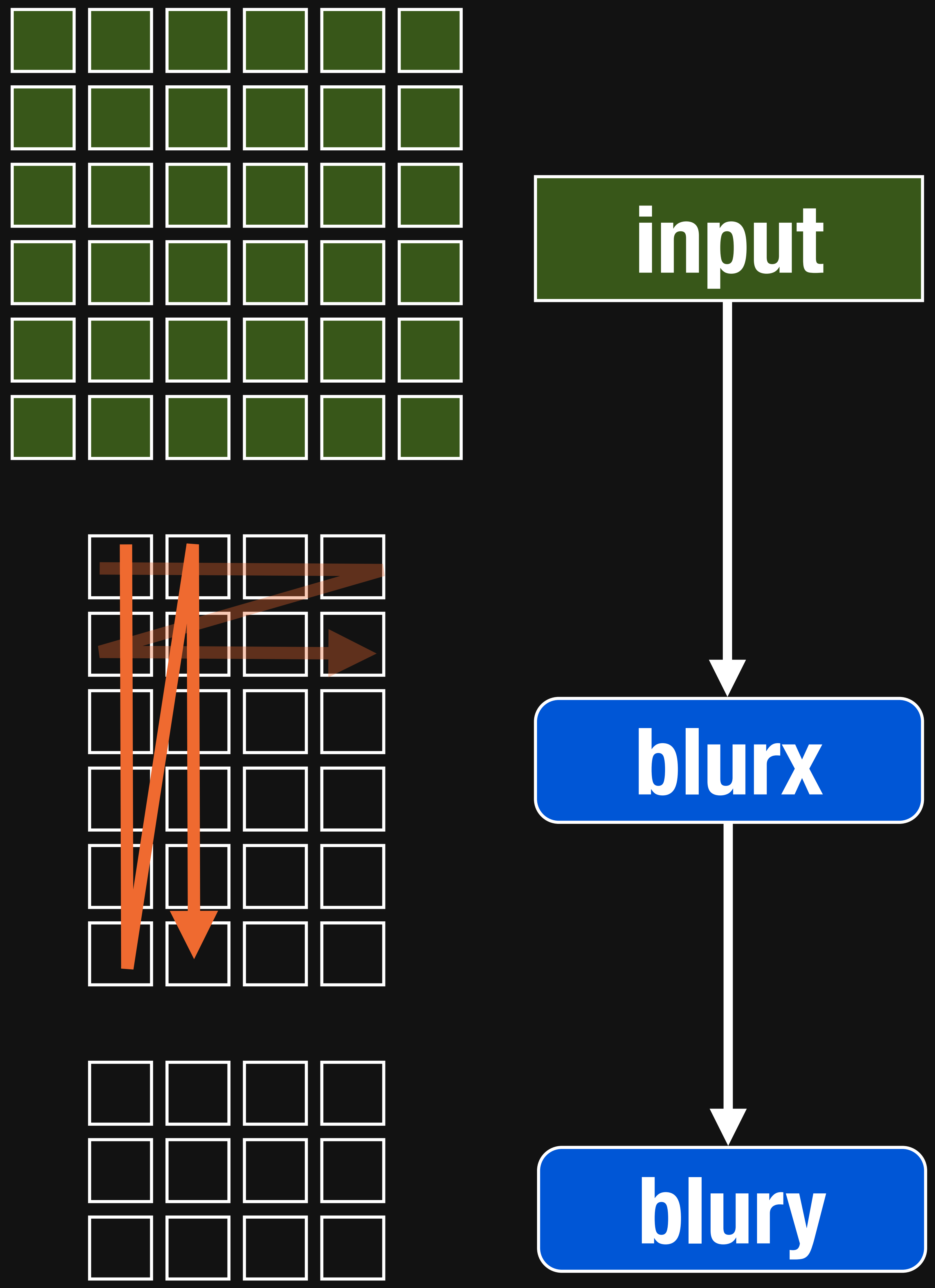


The schedule defines intra-stage order, inter-stage interleaving

For each stage:

1) In **what order** should we **compute** its **values**?
   split, tile, reorder, vectorize, unroll loops

2) **When** should we **compute** its **inputs**?

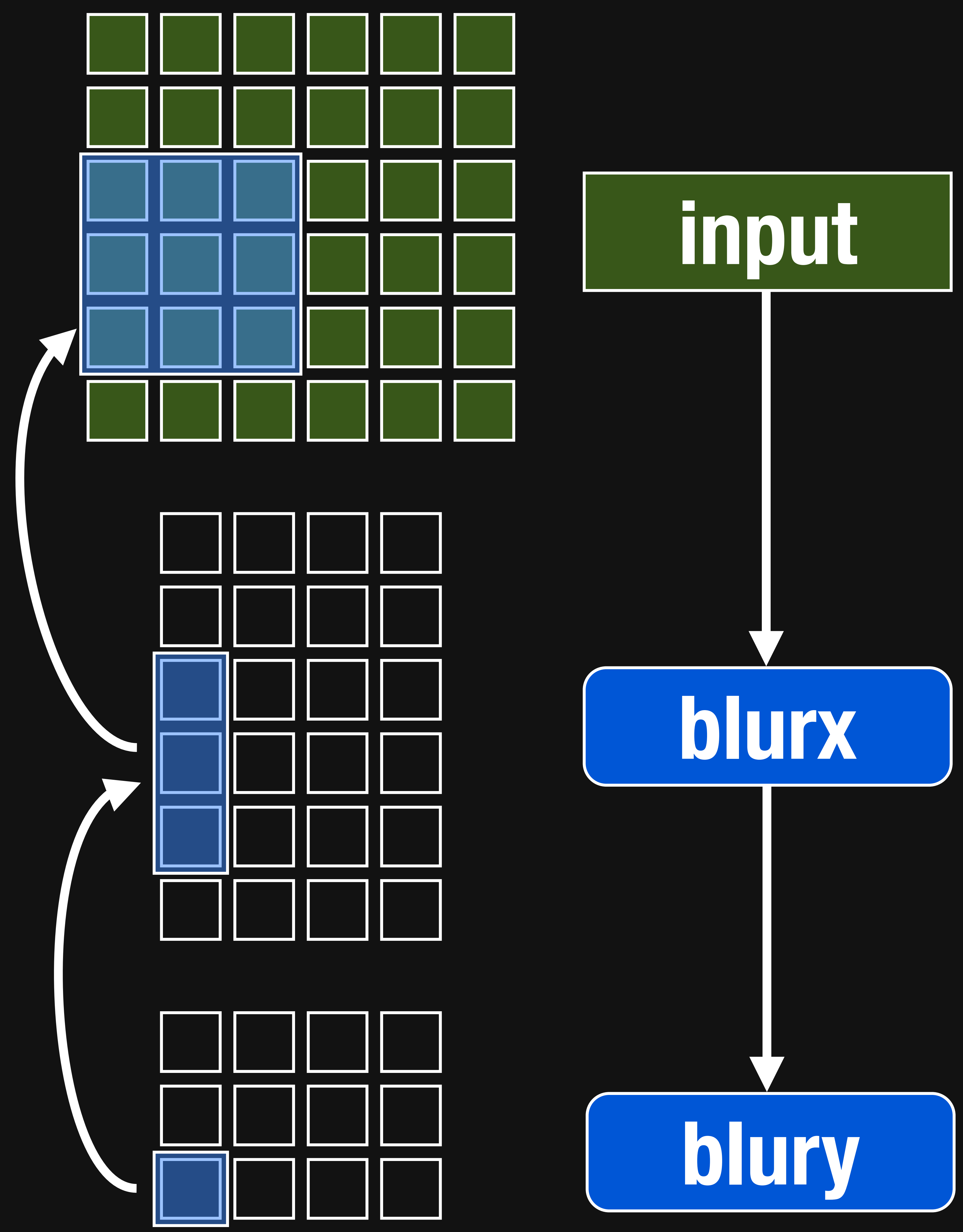


The schedule defines intra-stage order, inter-stage interleaving

For each stage:

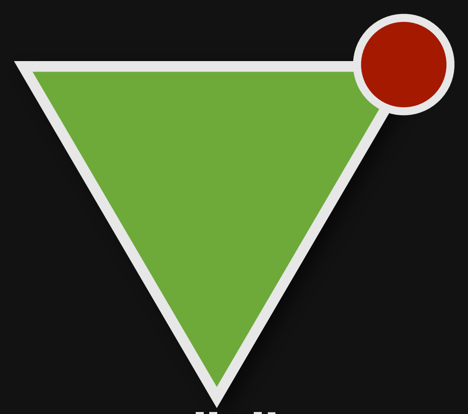
1) In **what order** should we **compute** its **values**?
   split, tile, reorder, vectorize, unroll loops

2) **When** should we **compute** its **inputs**?
   level in loop nest of consumers at which to compute each producer

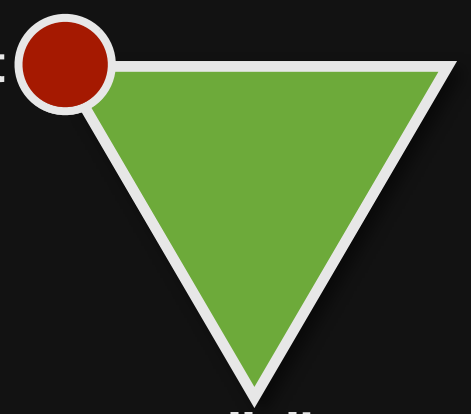


Schedule primitives **compose** to create many organizations

```
blur_x.compute_at_root()
```

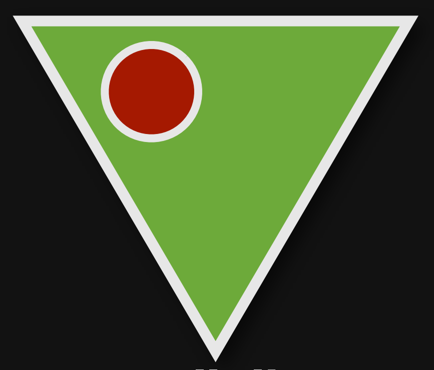



```
blur_x.compute_at(blury, x)
```

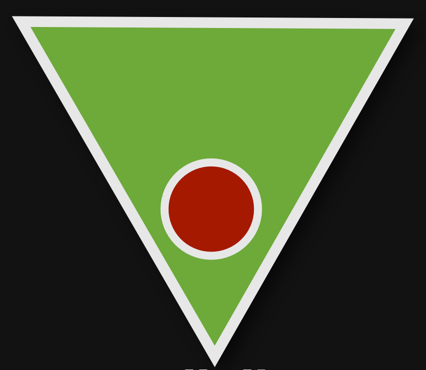



```
blur_x.compute_at(blury, x)
      .store_at_root()
```

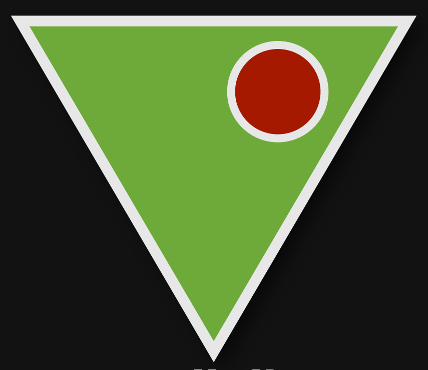

```
blur_x.compute_at(blury, x)
      .vectorize(x, 4)
blur_y.tile(x, y, xi, yi, 8, 8)
      .parallel(y)
      .vectorize(xi, 4)
```




```
blur_x.compute_at(blury, y)
      .store_at_root()
      .split(x, x, xi, 8)
      .vectorize(xi, 4)
      .parallel(x)
blur_y.split(x, x, xi, 8)
      .vectorize(xi, 4)
      .parallel(x)
```




```
blur_x.compute_at(blury, y)
      .store_at(blury, yi)
      .vectorize(x, 4)
blur_y.split(y, y, yi, 8)
      .parallel(y)
      .vectorize(x, 4)
```



Schedule primitives **compose** to create many organizations

A trivial Halide program

```
// The algorithm - no storage, order
a(x, y) = in(x, y);
b(x, y) = a(x, y);
c(x, y) = b(x, y);
```

A trivial Halide program

```
// The algorithm - no storage, order
a(x, y) = in(x, y);
b(x, y) = a(x, y);
c(x, y) = b(x, y);
```

Schedules are complex

split

reorder / reorder_storage

vectorize / parallel

compute_at / store_at

```
// generated schedule
a.split(x, x, x0, 4)
 .split(y, y, y1, 16)
 .reorder(y1, x0, y, x)
 .vectorize(y1, 4)
 .compute_at(b, y);
b.split(x, x, x2, 64)
 .reorder(x2, x, y)
 .reorder_storage(y, x)
 .vectorize(x2, 8)
 .compute_at(c, x4);
c.split(x, x, x4, 8)
 .split(y, y, y5, 2)
 .reorder(x4, y5, y, x)
 .parallel(x)
 .compute_root();
```

A trivial Halide program

```
// The algorithm - no storage, order
a(x, y) = in(x, y);
b(x, y) = a(x, y);
c(x, y) = b(x, y);
```

```
// generated schedule
a.split(x, x, x0, 4)
 .split(y, y, y1, 16)
 .reorder(y1, x0, y, x)
 .vectorize(y1, 4)
 .compute_at(b, y);
b.split(x, x, x2, 64)
 .reorder(x2, x, y)
 .reorder_storage(y, x)
 .vectorize(x2, 8)
 .compute_at(c, x4);
c.split(x, x, x4, 8)
 .split(y, y, y5, 2)
 .reorder(x4, y5, y, x)
 .parallel(x)
 .compute_root();
```

Schedules are complex

split

reorder / reorder_storage

vectorize / parallel

compute_at / store_at

A simple schedule (interleaving only)

Schedule:

```
a.compute_at(b, y);
b.compute_at(c, x);
c.compute_root();
```

Synthesized loop nest:

```
for c.x:
  for b.x:
    for b.y:
      for a.x:
        for a.y:
          a[a.x, a.y] = in[a.x, a.y]
      b[b.x, b.y] = a[b.x, b.y]
  for c.y:
    c[c.x, c.y] = b[c.x, c.y]
```

A naive representation

Direct schedule encoding:

```
a.compute_at(b, y);
b.compute_at(c, x);
c.compute_root();
```

A naive representation

Direct schedule encoding:

```
a.compute_at(b, y);
b.compute_at(c, x);
c.compute_root();
```

8 placement locations

compute_at(a, x)
compute_at(a, y)
compute_at(b, x)
compute_at(b, y)
compute_at(c, x)
compute_at(c, y)
compute_root()
inline

A naive representation

Direct schedule encoding:

```
a.compute_at(b, y);
b.compute_at(c, x);
c.compute_root();
```

8 placement locations

compute_at(a, x)
compute_at(a, y)
compute_at(b, x)
compute_at(b, y)
compute_at(c, x)
compute_at(c, y)
compute_root()
inline

3 functions to place

(a, b, c)

A naive representation

Direct schedule encoding:

```
a.compute_at(b, y);
b.compute_at(c, x);
c.compute_root();
```

8 placement locations

compute_at(a, x)
compute_at(a, y)
compute_at(b, x)
compute_at(b, y)
compute_at(c, x)
compute_at(c, y)
compute_root()
inline

3 functions to place

(a, b, c)

8^3 = 512 possible schedules

A naive representation doesn't work

Most of the space is meaningless

474 of 512 schedules are invalid

Exponentially worse for large programs

Poor search space locality

small changes radically restructure the generated loops

Fails completely for nontrivial programs

A naive representation doesn't work

Most of the space is meaningless

474 of 512 schedules are invalid

Exponentially worse for large programs

Poor search space locality

small changes radically restructure the generated loops

Fails completely for nontrivial programs

```
for c.x:
  for b.x:
    for b.y:
      for a.x:
        for a.y:
          compute a()
      compute b()
  for c.y:
    compute c()
```

A naive representation doesn't work

Most of the space is meaningless

474 of 512 schedules are invalid

Exponentially worse for large programs

Poor search space locality

small changes radically restructure the generated loops

```
for c.x:
  for b.x:
    for b.y:
      for a.x:
        for a.y:
          compute a()
      compute b()
  for c.y:
    compute c()
```

A naive representation doesn't work

Most of the space is meaningless

474 of 512 schedules are invalid

Exponentially worse for large programs

Poor search space locality

small changes radically restructure the generated loops

Fails completely for nontrivial programs

```
for c.x:
  for b.x:
    for b.y:
      for a.x:
        for a.y:
          compute a()
      compute b()
  for c.y:
    compute c()
```

A better representation

A better representation

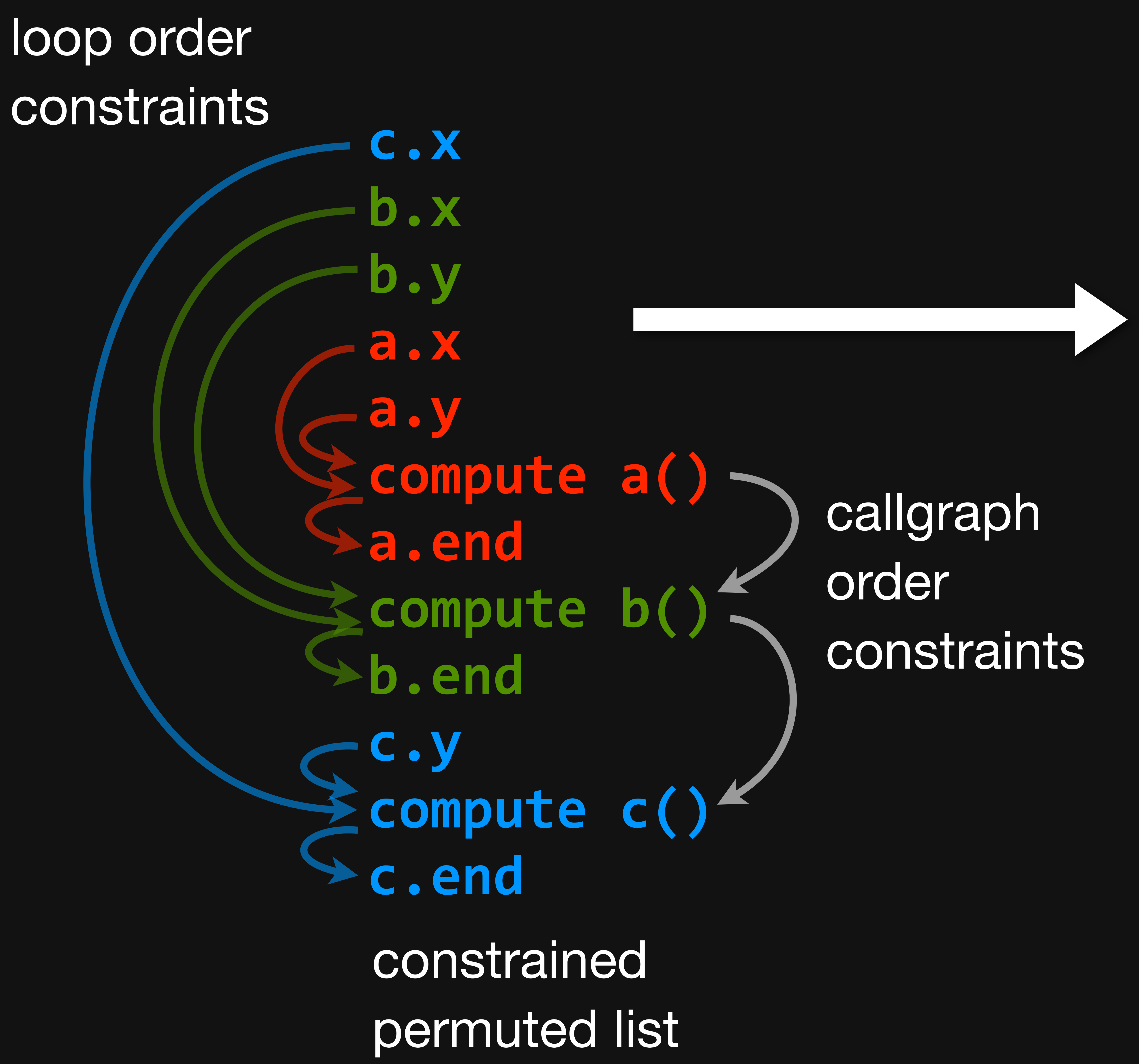
loop order
constraints

```
c.x
b.x
b.y
a.x
a.y
compute a()
a.end
compute b()
b.end
c.y
compute c()
c.end
```

callgraph
order
constraints

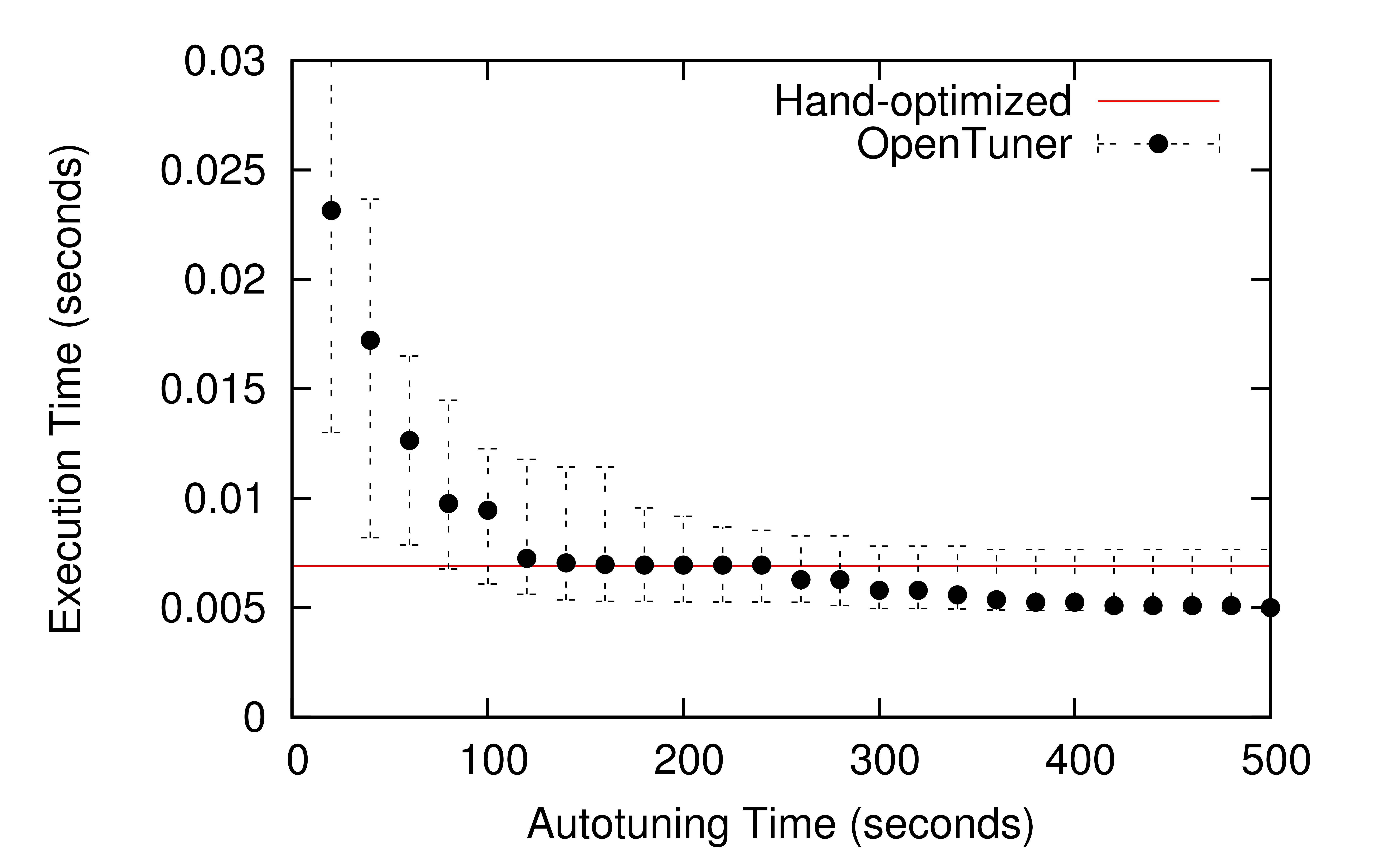
constrained
permuted list

A better representation

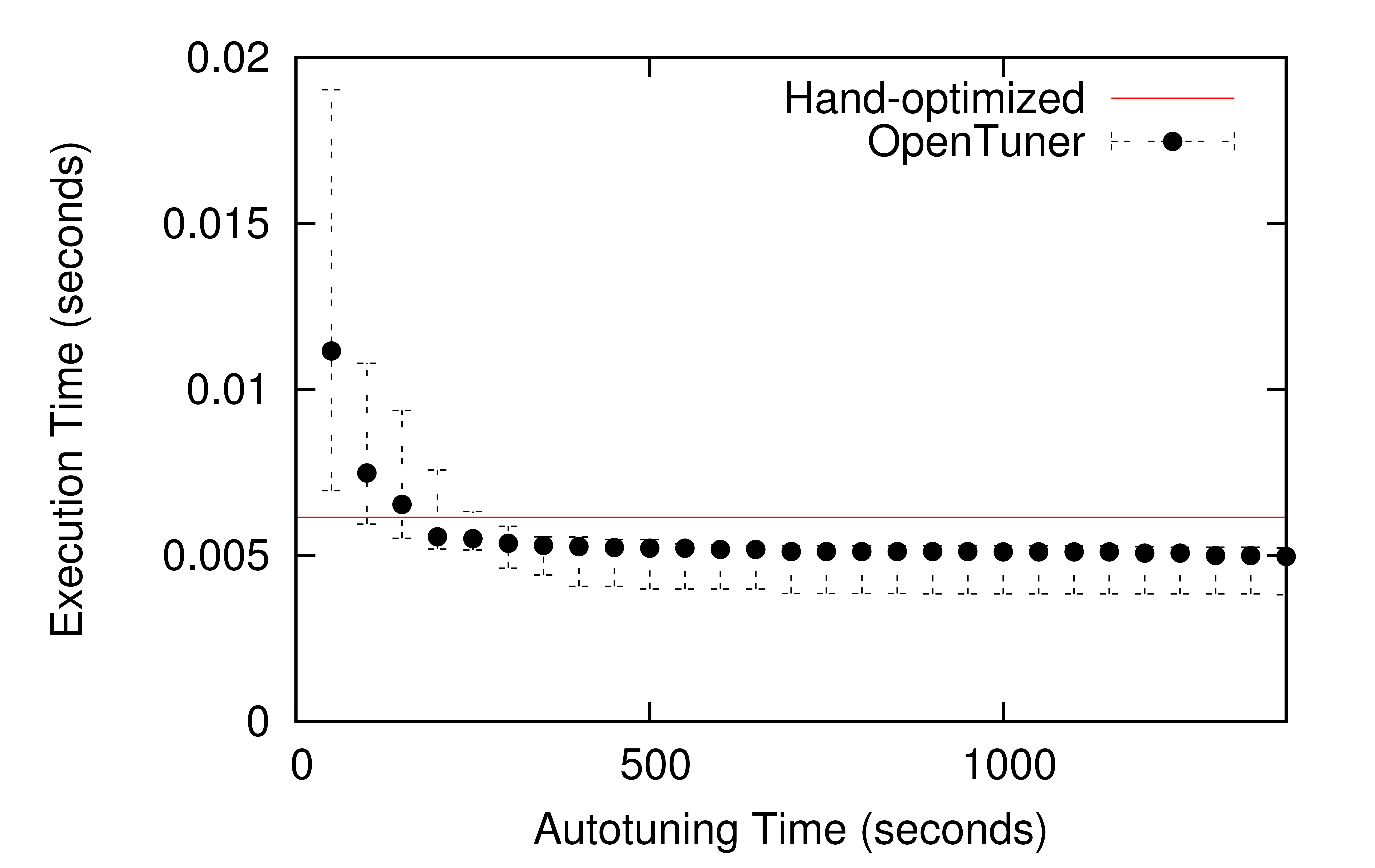



```
for c.x:
  for b.x:
    for b.y:
      for a.x:
        for a.y:
          compute a()
      compute b()
  for c.y:
    compute c()
```

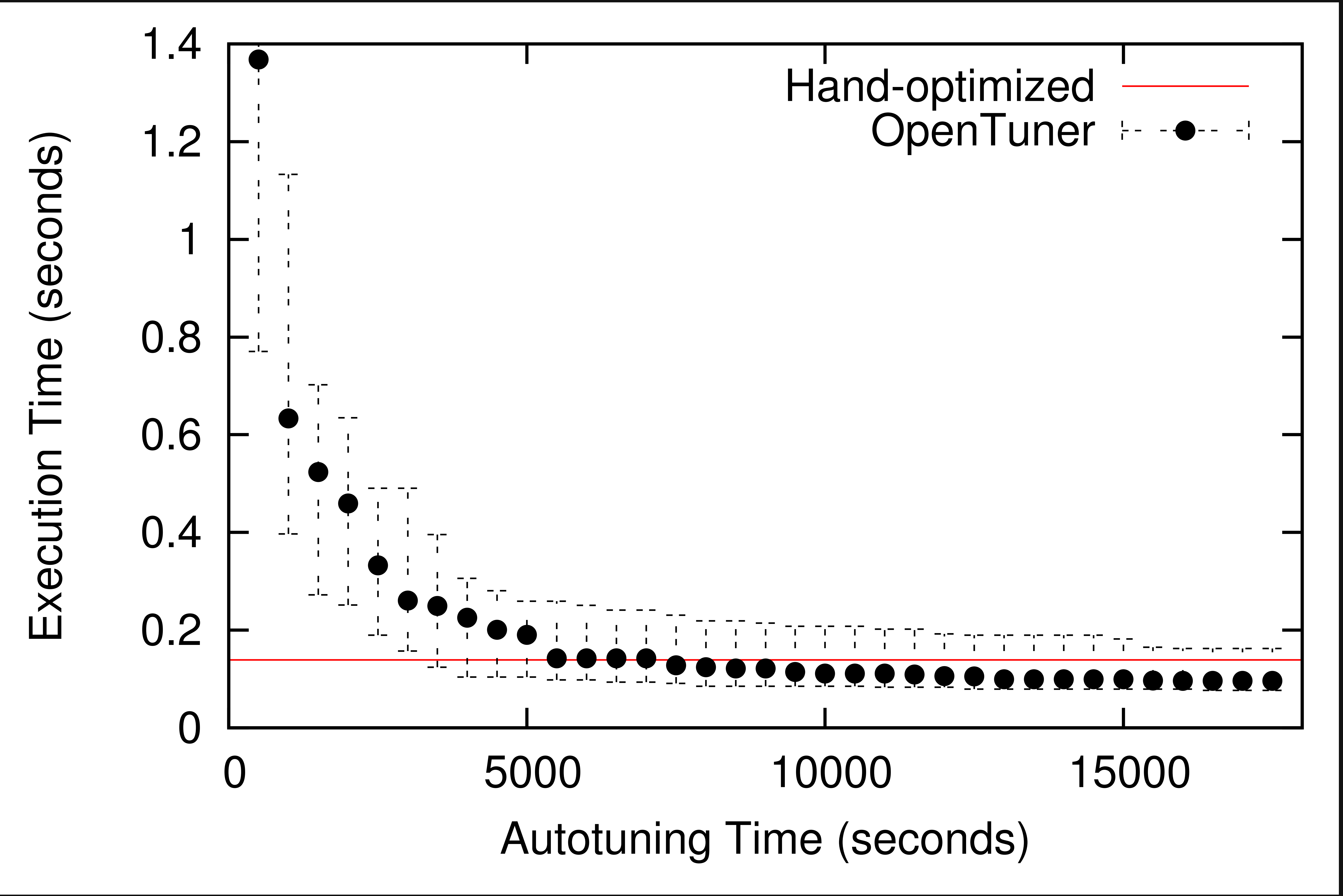
Results: *blur*



Results: *wavelet*



Results: *bilateral grid*



x86	Speedup	Factor shorter
Blur	1.2 ×	18 ×
Bilateral Grid	4.4 ×	4 ×
Camera pipeline	3.4 ×	2 ×
“Healing brush”	1.7 ×	7 ×
Local Laplacian	1.7 ×	5 ×

GPU	Speedup	Factor shorter
Bilateral Grid	2.3 ×	11 ×
“Healing brush”	*5.9** ×	*7** ×
Local Laplacian	*9** ×	*7** ×

ARM	Speedup	Factor shorter
Camera pipeline	1.1 ×	3 ×

x86	Speedup	Factor shorter
Blur	1.2 ×	18 ×
Bilateral Grid	4.4 ×	4 ×
Camera pipeline	3.4 ×	2 ×
“Healing brush”	1.7 ×	7 ×
Local Laplacian	1.7 ×	5 ×

GPU	Speedup	Factor shorter
Bilateral Grid	2.3 ×	11 ×
“Healing brush”	*5.9** ×	*7** ×
Local Laplacian	*9** ×	*7** ×

ARM	Speedup	Factor shorter
Camera pipeline	1.1 ×	3 ×

x86	Speedup	Factor shorter
Blur	1.2 ×	18 ×
Bilateral Grid	4.4 ×	4 ×
Camera pipeline	3.4 ×	2 ×
"Healing brush"	1.7 ×	7 ×
Local Laplacian	1.7 ×	5 ×

GPU	Speedup	Factor shorter
Bilateral Grid	2.3 ×	11 ×
"Healing brush"	*5.9** ×	*7** ×
Local Laplacian	*9** ×	*7** ×

ARM	Speedup	Factor shorter
Camera pipeline	1.1 ×	3 ×

Autotuning time:
(single node)

2 hrs to 2 days

85% within < 24 hrs

x86	Speedup	Factor shorter
Blur	1.2 ×	18 ×
Bilateral Grid	4.4 ×	4 ×
Camera pipeline	3.4 ×	2 ×
"Healing brush"	1.7 ×	7 ×
Local Laplacian	1.7 ×	5 ×

GPU	Speedup	Factor shorter
Bilateral Grid	2.3 ×	11 ×
"Healing brush"	*5.9** ×	*7** ×
Local Laplacian	*9** ×	*7** ×

ARM	Speedup	Factor shorter
Camera pipeline	1.1 ×	3 ×

In progress

new representation

smarter heuristic seed schedules

Autotuning time:
(single node)

2 hrs to 2 days

85% within < 24 hrs

Halide: current status

open source at http://halide-lang.org

Google
~ **50** developers
\> **10 kLOC** in production

G+ Photos *auto-enhance*
Data center
Android
Chrome (PNaCl)

HDR+
Glass
Nexus devices

Computational photography course (6.815)
60 undergrads