An introduction to Halide

Jonathan Ragan-Kelley (Stanford)
Andrew Adams (Google)
Dillon Sharlet (Google)

Today's agenda

Now: **the big ideas in Halide**

Later: **writing & optimizing real code**

Hello world (brightness)

Gaussian blur - *3x OpenCV*

Simple enhancement pipeline - *6x OpenCV*

break

MATLAB integration

IIR filter

CNN layers

break

GPU scheduling

Finally: **real-time HOG on a phone**

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

Today's methodology

C++ w/multithreading, SIMD
CUDA/OpenCL
OpenGL/RenderScript

Today's methodology

C++ w/multithreading, SIMD
CUDA/OpenCL
OpenGL/RenderScript

Optimization requires manually
transforming program & data structure
for locality and parallelism.

Today’s methodology

C++ w/multithreading, SIMD
CUDA/OpenCL
OpenGL/RenderScript

Optimization requires manually
transforming program & data structure
for locality and parallelism.

libraries don’t solve this:
BLAS, IPP, MKL, OpenCV
optimized kernels compose into
inefficient pipelines (no fusion)

Local Laplacian Filters

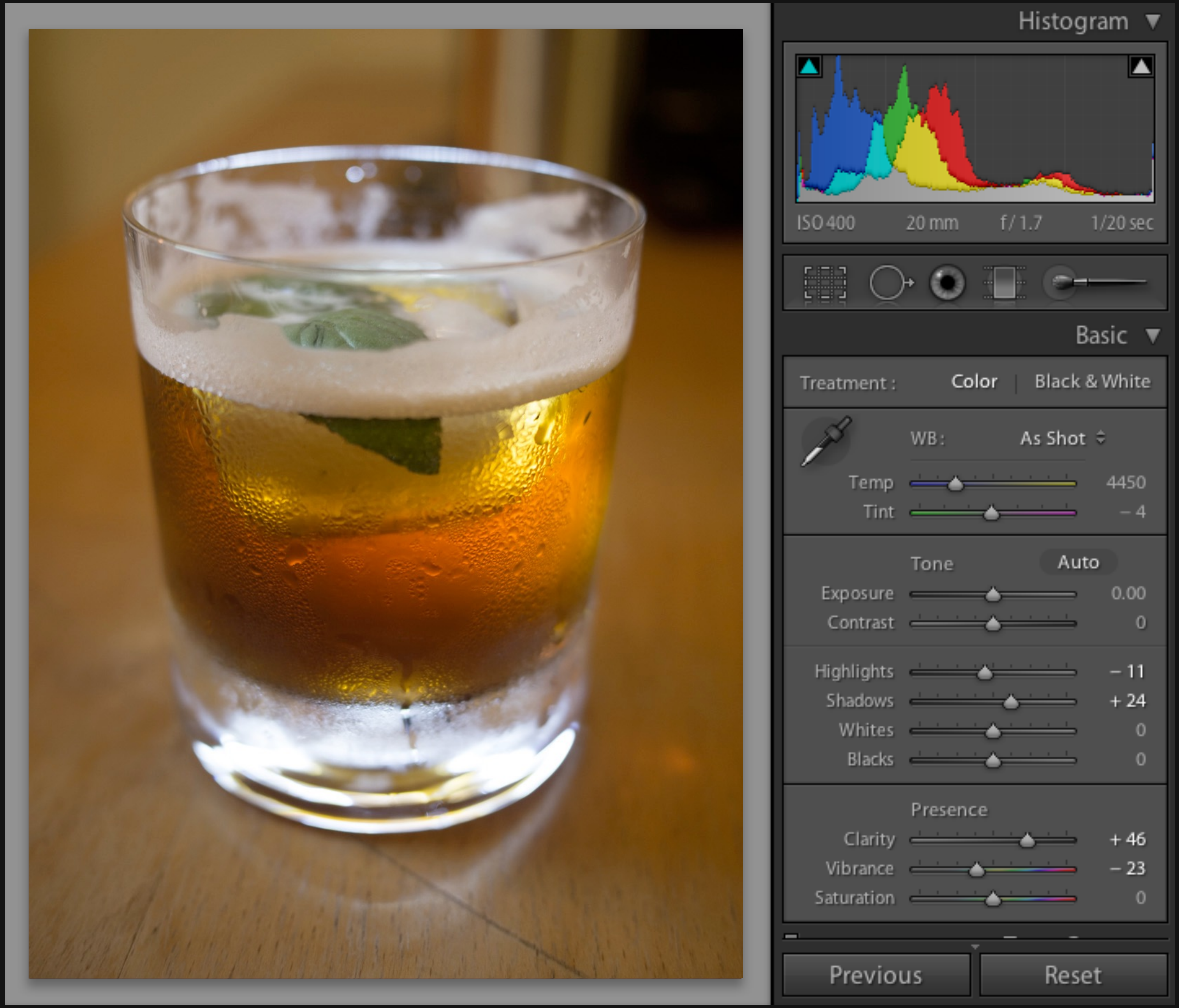
in Adobe Photoshop Camera Raw / Lightroom

1500 lines of expert-optimized C++

multi-threaded, SSE

3 months of work

10x faster *than reference C*



Local Laplacian Filters

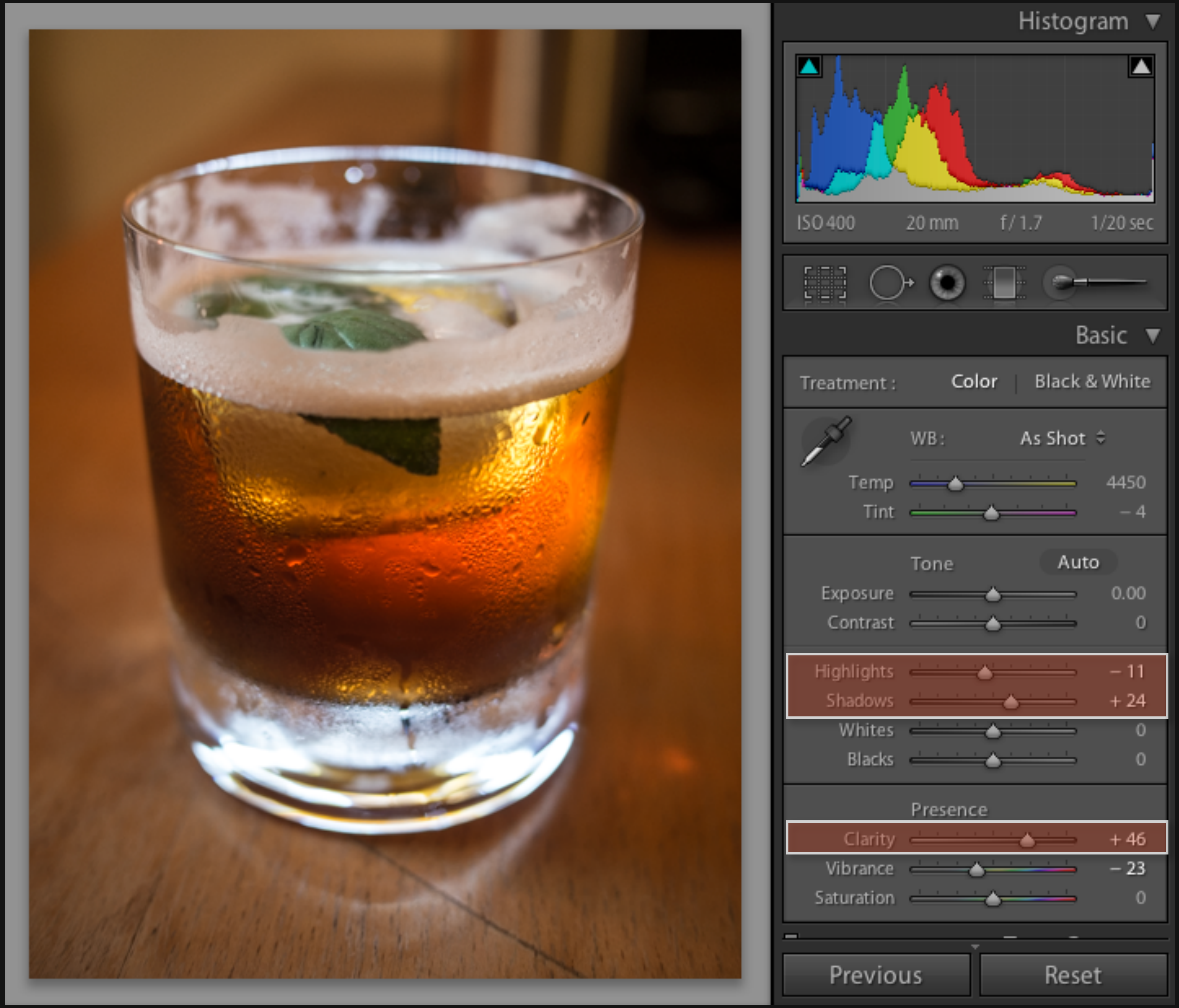
in Adobe Photoshop Camera Raw / Lightroom

1500 lines of expert-optimized C++

multi-threaded, SSE

3 months of work

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



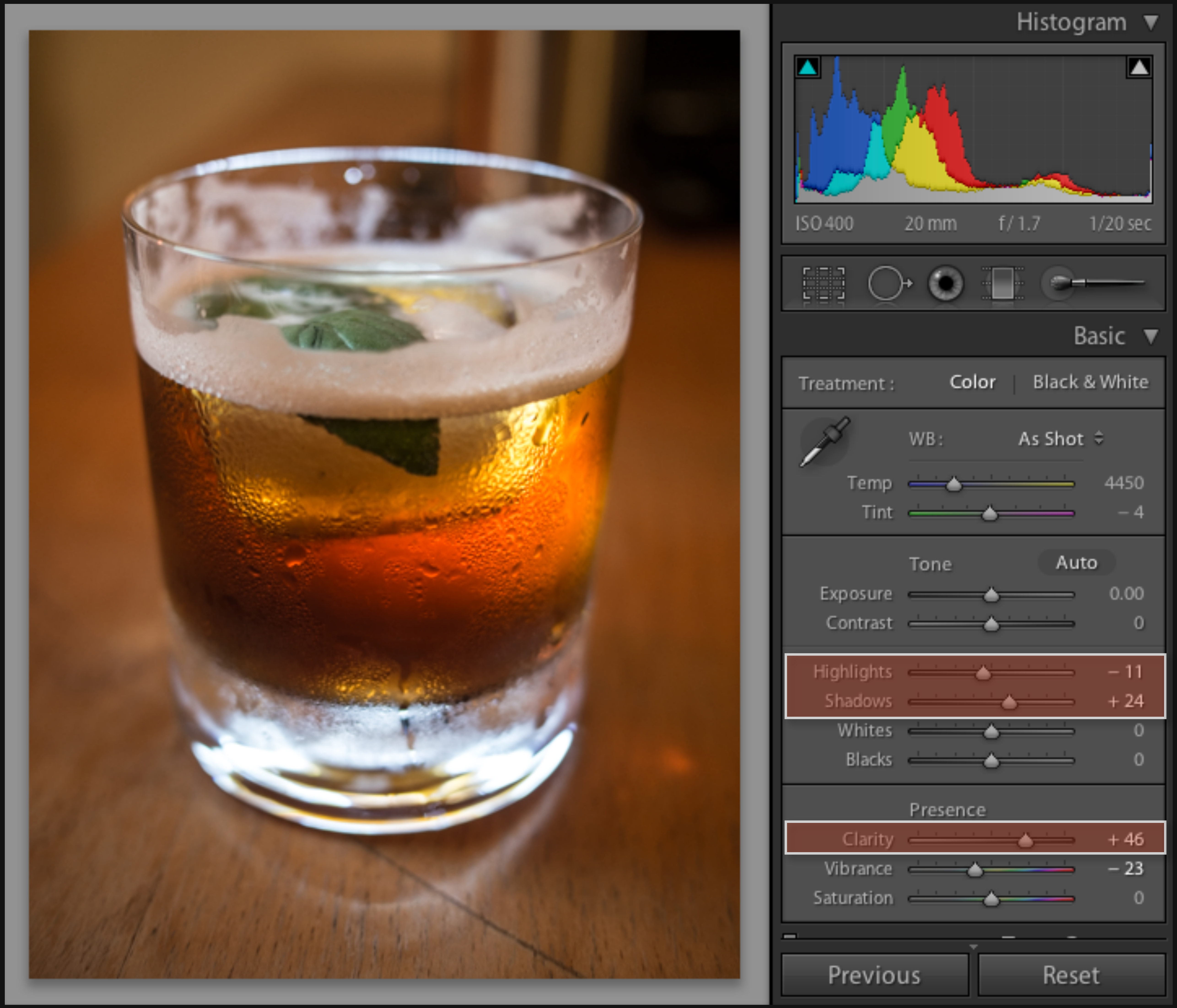
Local Laplacian Filters
in Adobe Photoshop Camera Raw / Lightroom

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

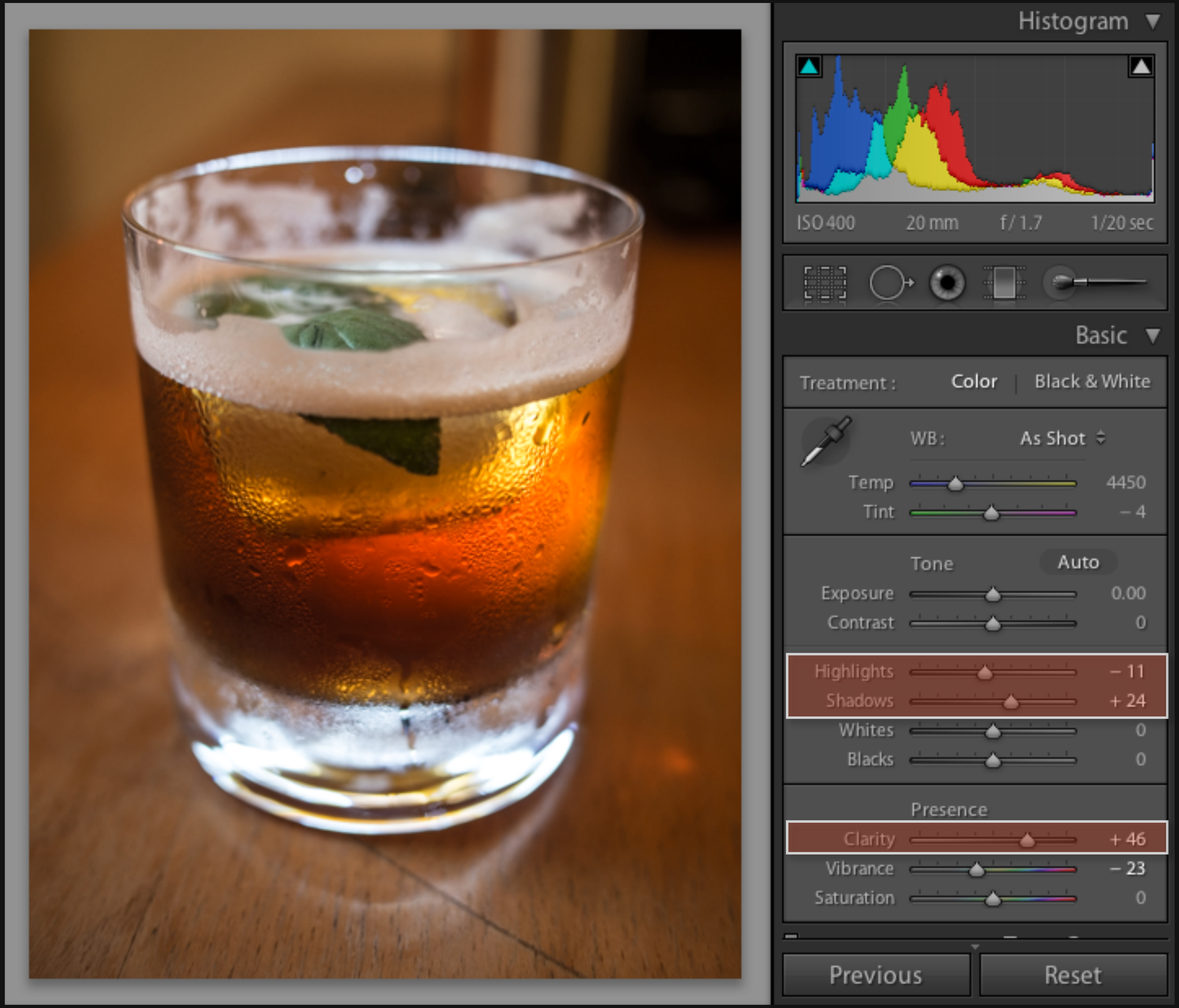
1500 lines of expert-optimized C++

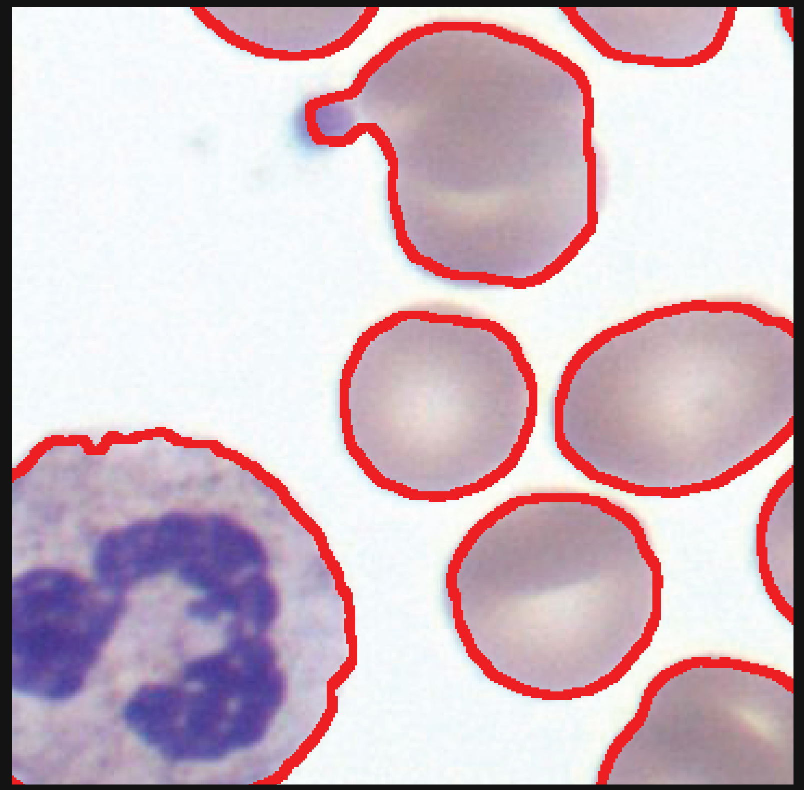
multi-threaded, SSE

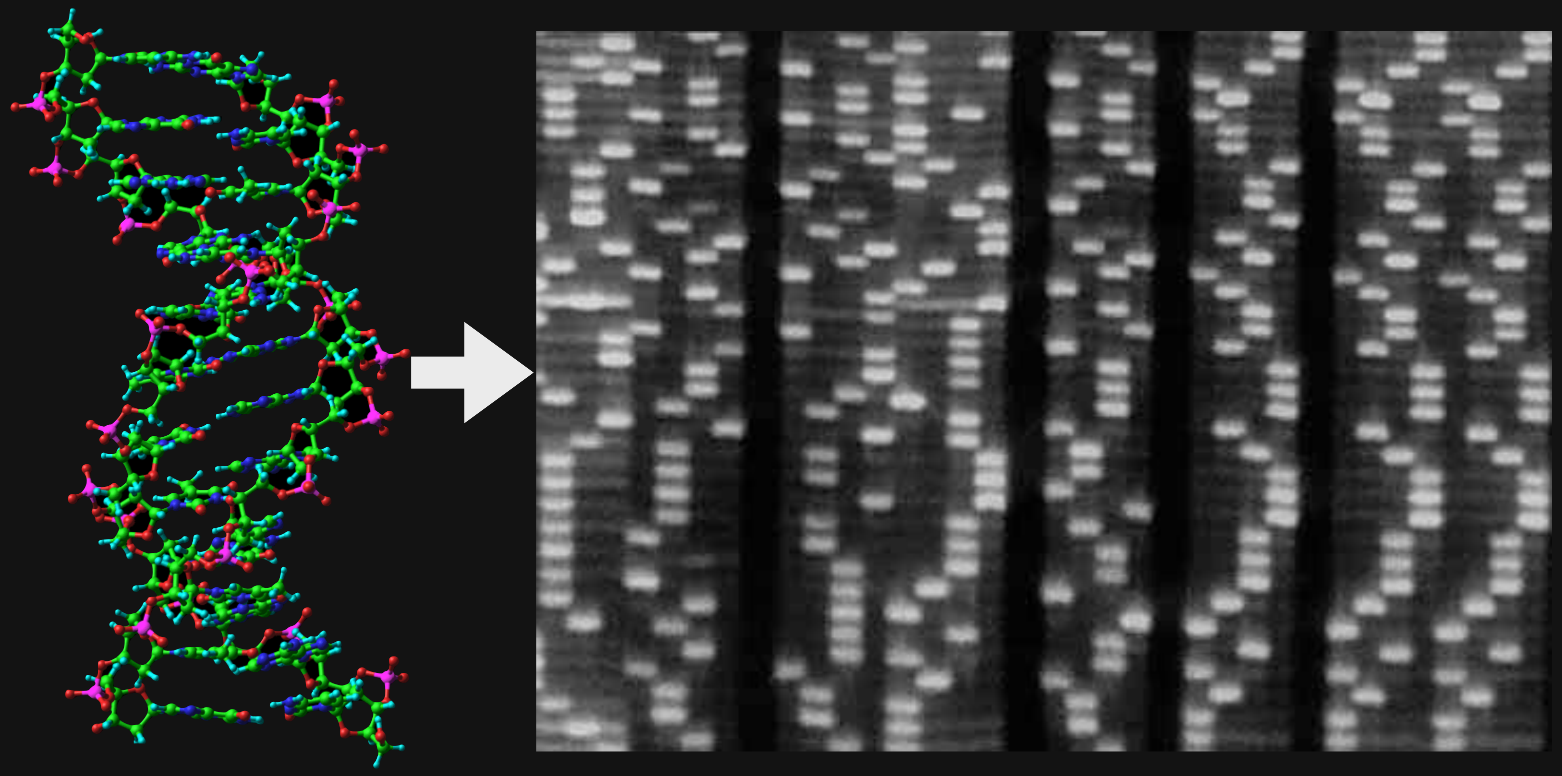
3 months of work

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

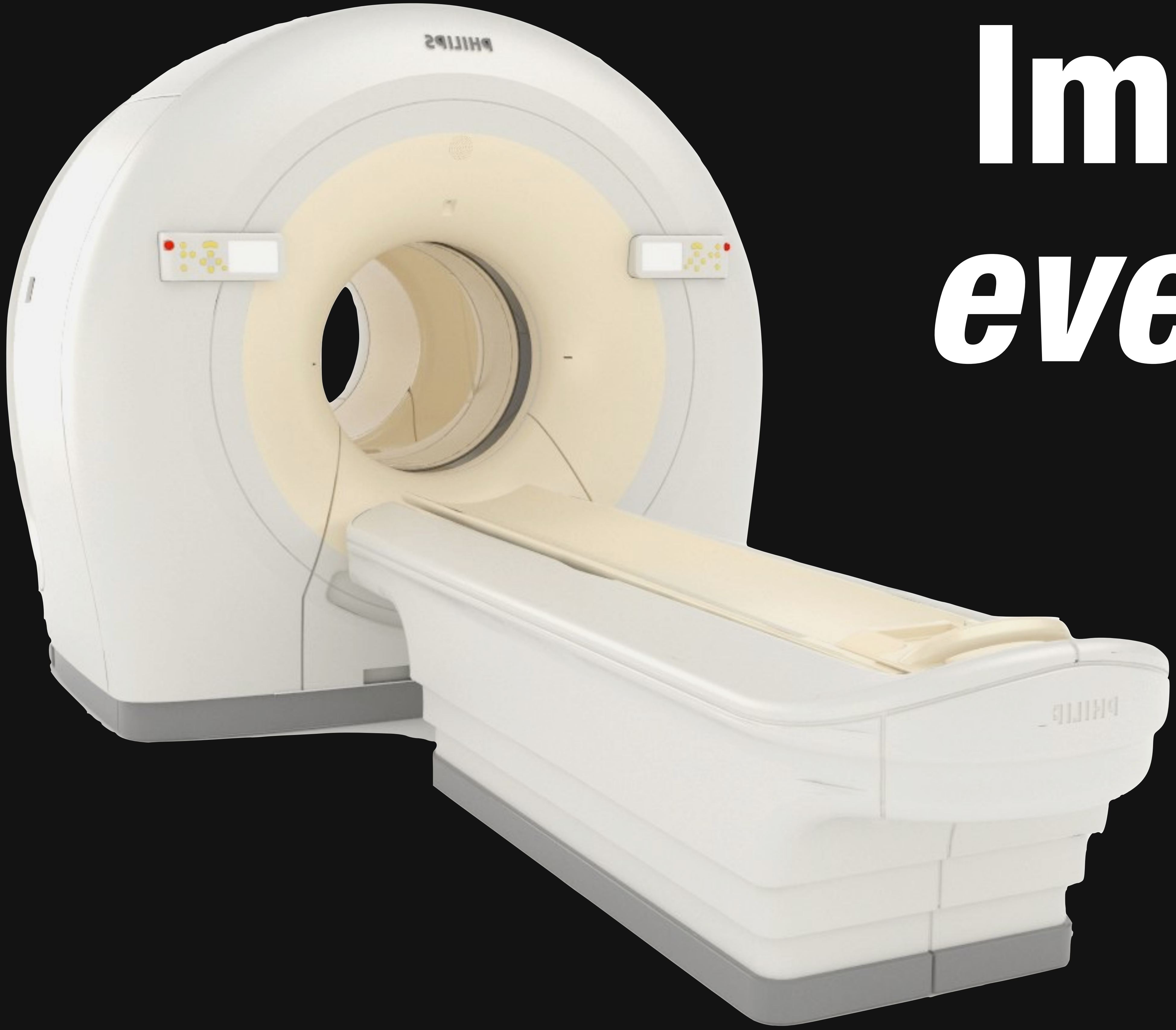
Just writing in C isn't nearly enough!

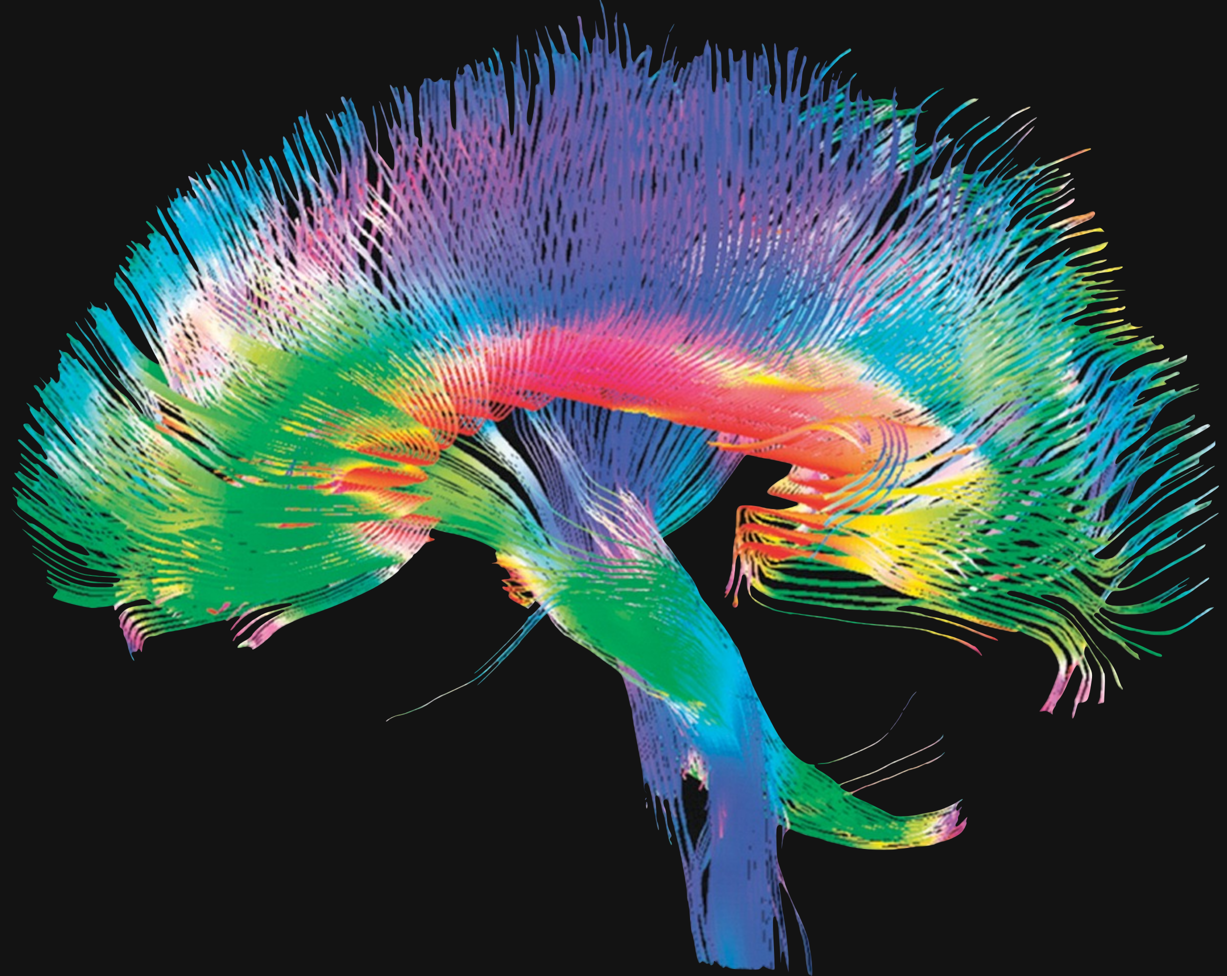


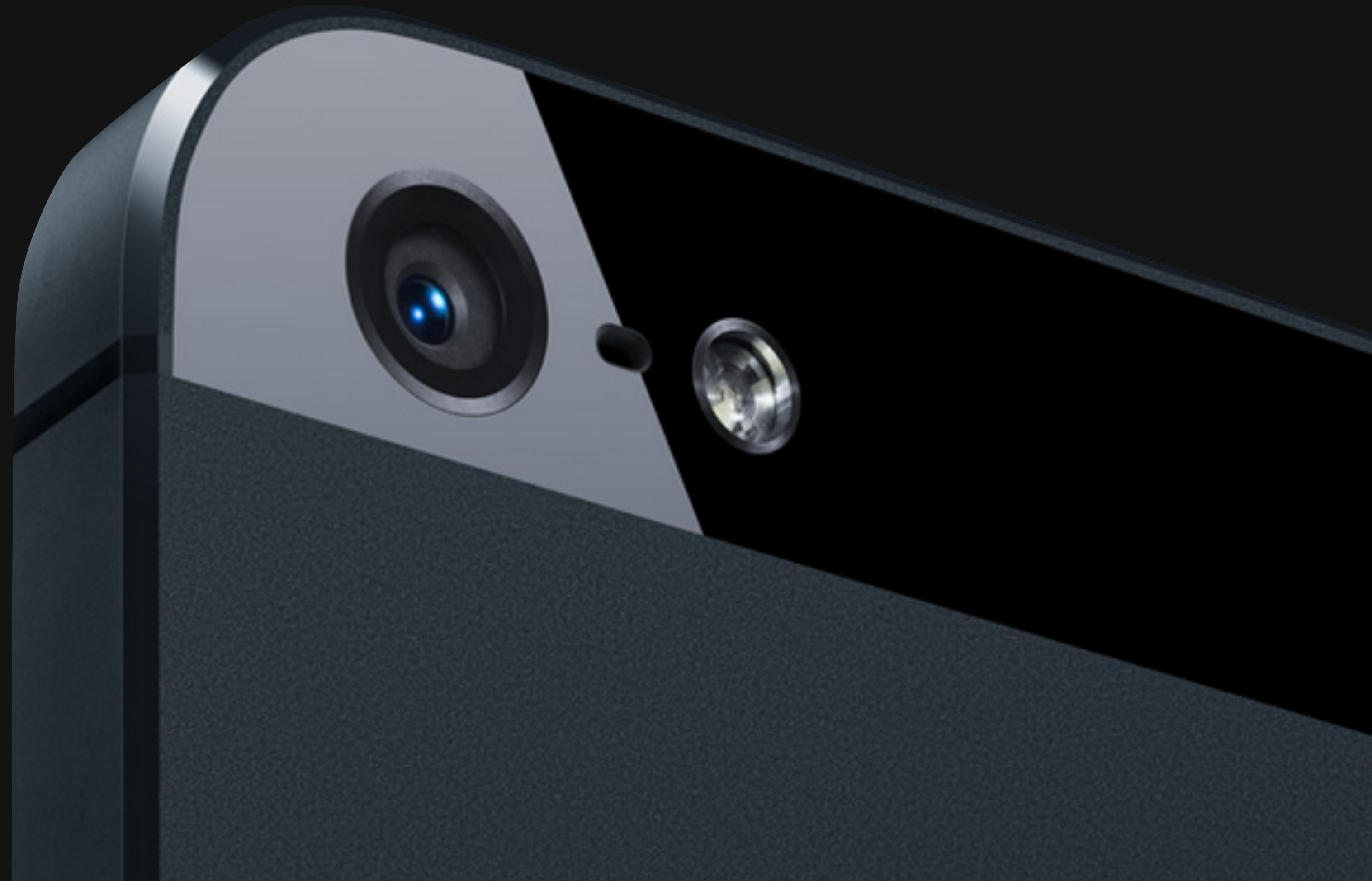




Imaging is *everywhere*







A simple example: 3x3 blur

```
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height());  // allocate blurx array

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
}
```

Hand-optimized C++

9.9 → 0.9 ms/megapixel

```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurxPtr = blurx;
      for (int y = -1; y < 32+1; y++) {
        const uint16_t *inPtr = &(in[yTile+y][xTile]);
        for (int x = 0; x < 256; x += 8) {
         a = _mm_loadu_si128((__m128i*)(inPtr-1));
         b = _mm_loadu_si128((__m128i*)(inPtr+1));
         c = _mm_load_si128((__m128i*)(inPtr));
         sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
         avg = _mm_mulhi_epi16(sum, one_third);
         _mm_store_si128(blurxPtr++, avg);
         inPtr += 8;
      }}
      blurxPtr = blurx;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blury[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurxPtr+(2*256)/8);
          b = _mm_load_si128(blurxPtr+256/8);
          c = _mm_load_si128(blurxPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

11x faster
(quad core x86)

Tiled, fused

Vectorized

Multithreaded

Redundant computation

Near roof-line optimum

Halide's answer: *decouple* algorithm from schedule

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

Easy for programmers to build pipelines
simplifies algorithm code
improves modularity

Easy for programmers to specify & explore optimizations
fusion, tiling, parallelism, vectorization
can't break the algorithm

Easy for the compiler to generate fast code

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*:

```
Var x, y; Func blurx, blury;
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;
```

Domain scope
of the programming model

All computation is over **regular grids**.

Only **feed-forward pipelines**

Recursive/reduction computations are a (partial) escape hatch.

Recursion must have **bounded depth.**

Domain scope
of the programming model

All computation is over **regular grids**.

not Turing complete

- Only **feed-forward pipelines**
  - Recursive/reduction computations are a (partial) escape hatch.
- **Recursion** must have **bounded depth.**

Halide's answer: *decouple* algorithm from schedule

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

Easy for programmers to build pipelines
simplifies algorithm code
improves modularity

Easy for programmers to specify & explore optimizations
fusion, tiling, parallelism, vectorization
can't break the algorithm

Easy for the compiler to generate fast code

Halide's answer: *decouple* algorithm from schedule

Algorithm: ***what*** is computed

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

Easy for programmers to build pipelines

simplifies algorithm code

improves modularity

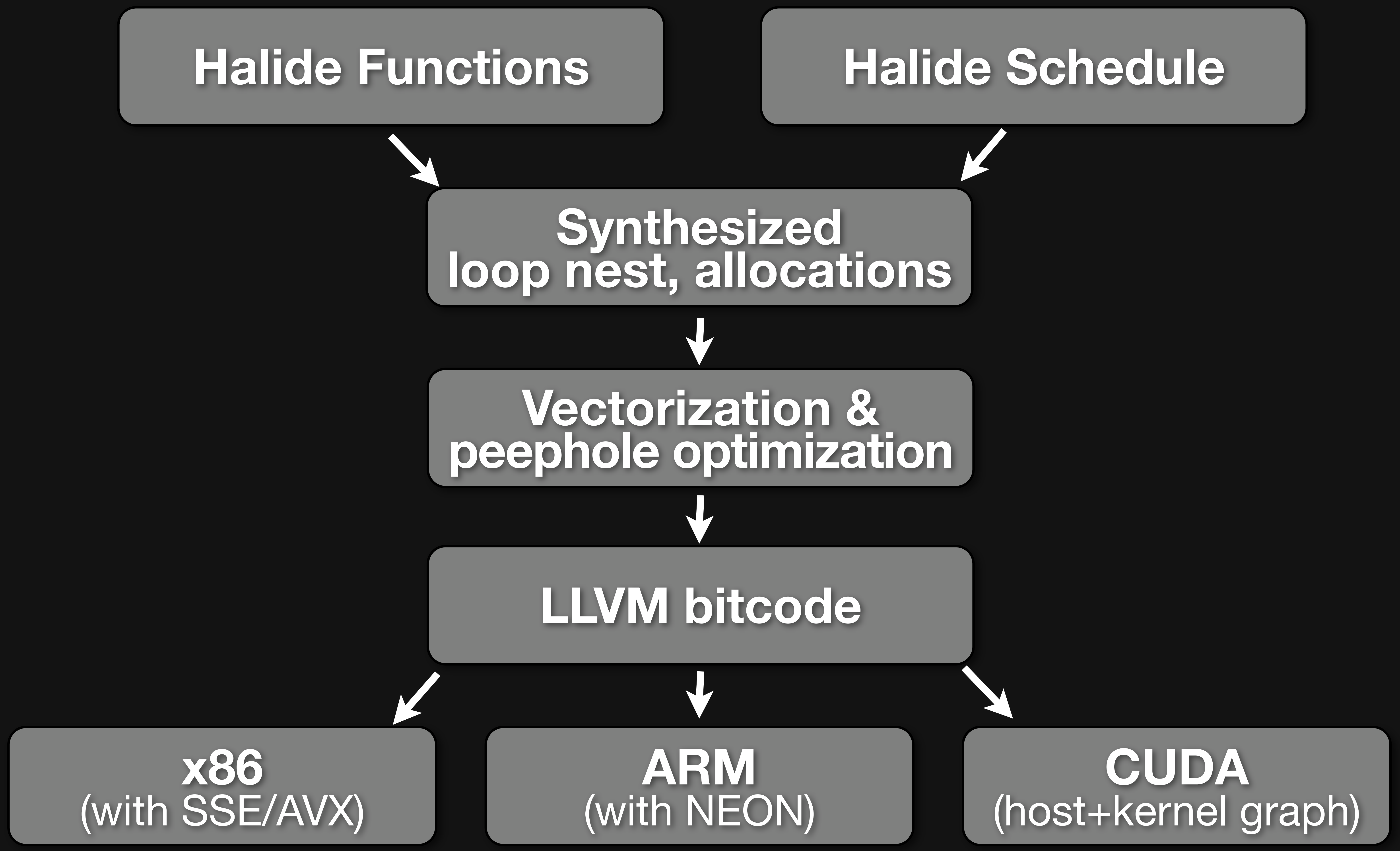
Easy for programmers to specify & explore optimizations

fusion, tiling, parallelism, vectorization

can't break the algorithm

Easy for the compiler to generate fast code

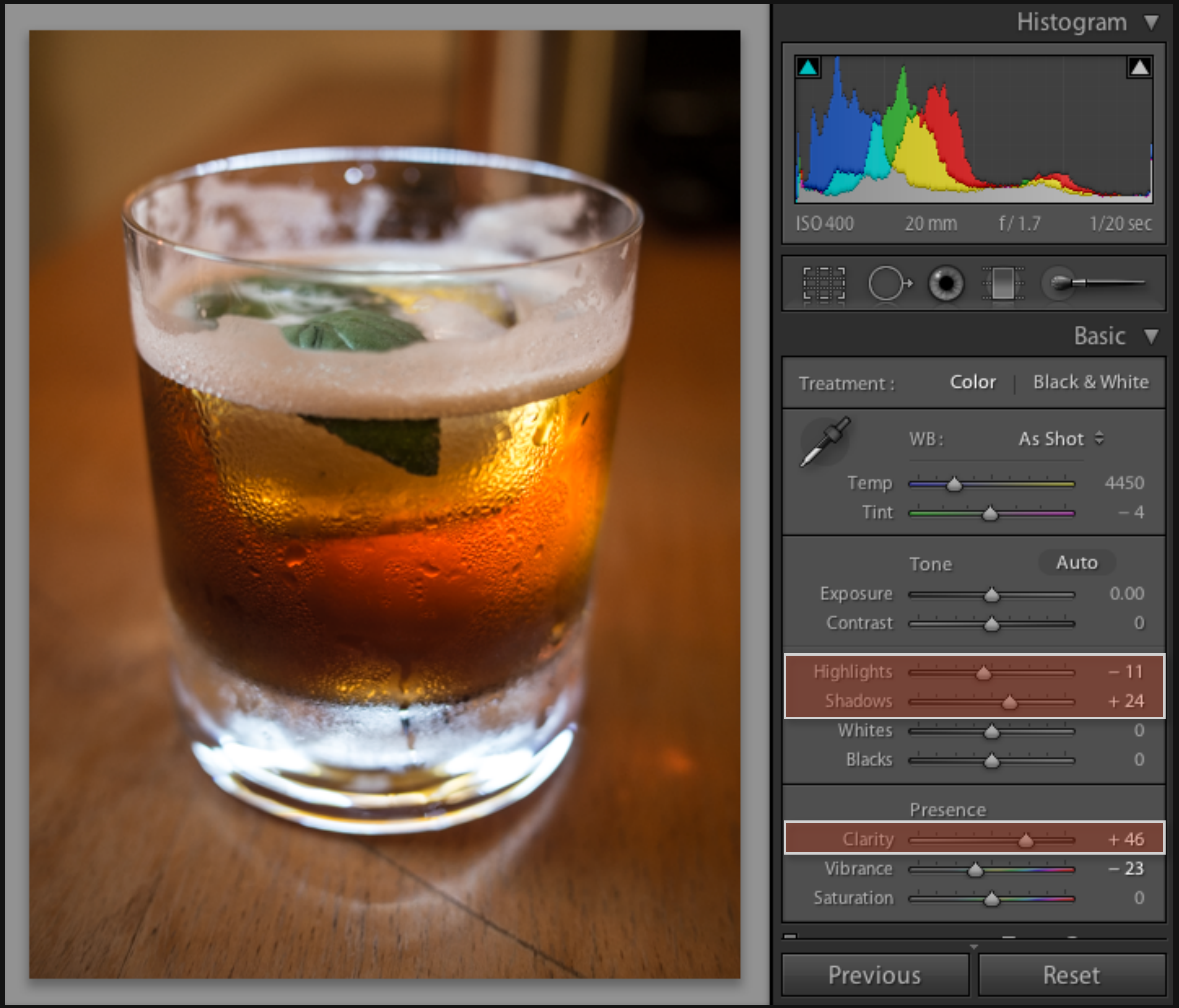
The Halide Compiler



The Halide Compiler

Local Laplacian Filters

prototype for Adobe Photoshop Camera Raw / Lightroom



Local Laplacian Filters

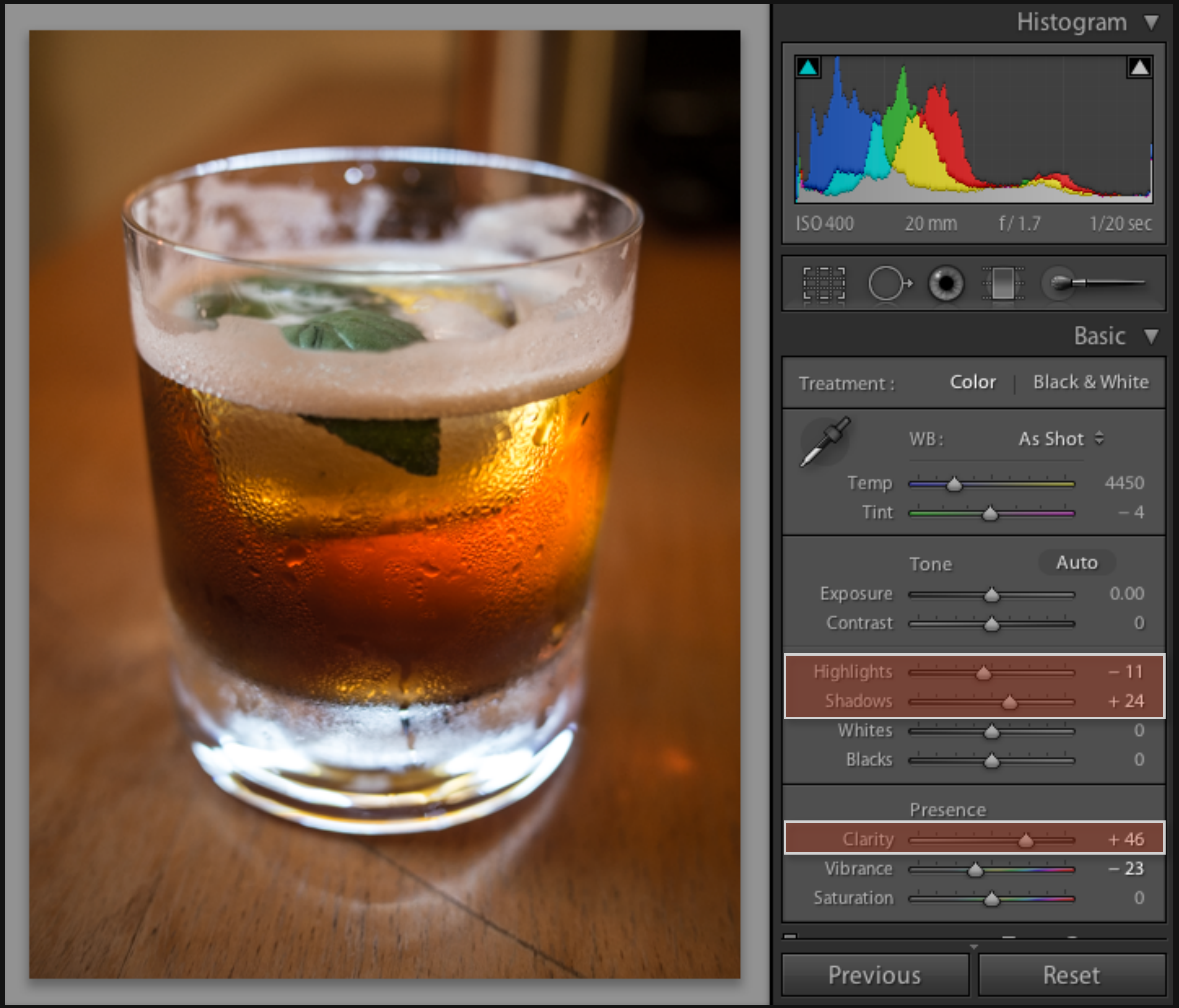
prototype for Adobe Photoshop Camera Raw / Lightroom

Reference: 300 lines C++

Adobe: 1500 lines

3 months of work

10x faster *(vs. reference)*



Local Laplacian Filters

prototype for Adobe Photoshop Camera Raw / Lightroom

Reference: 300 lines C++

Adobe: 1500 lines

3 months of work

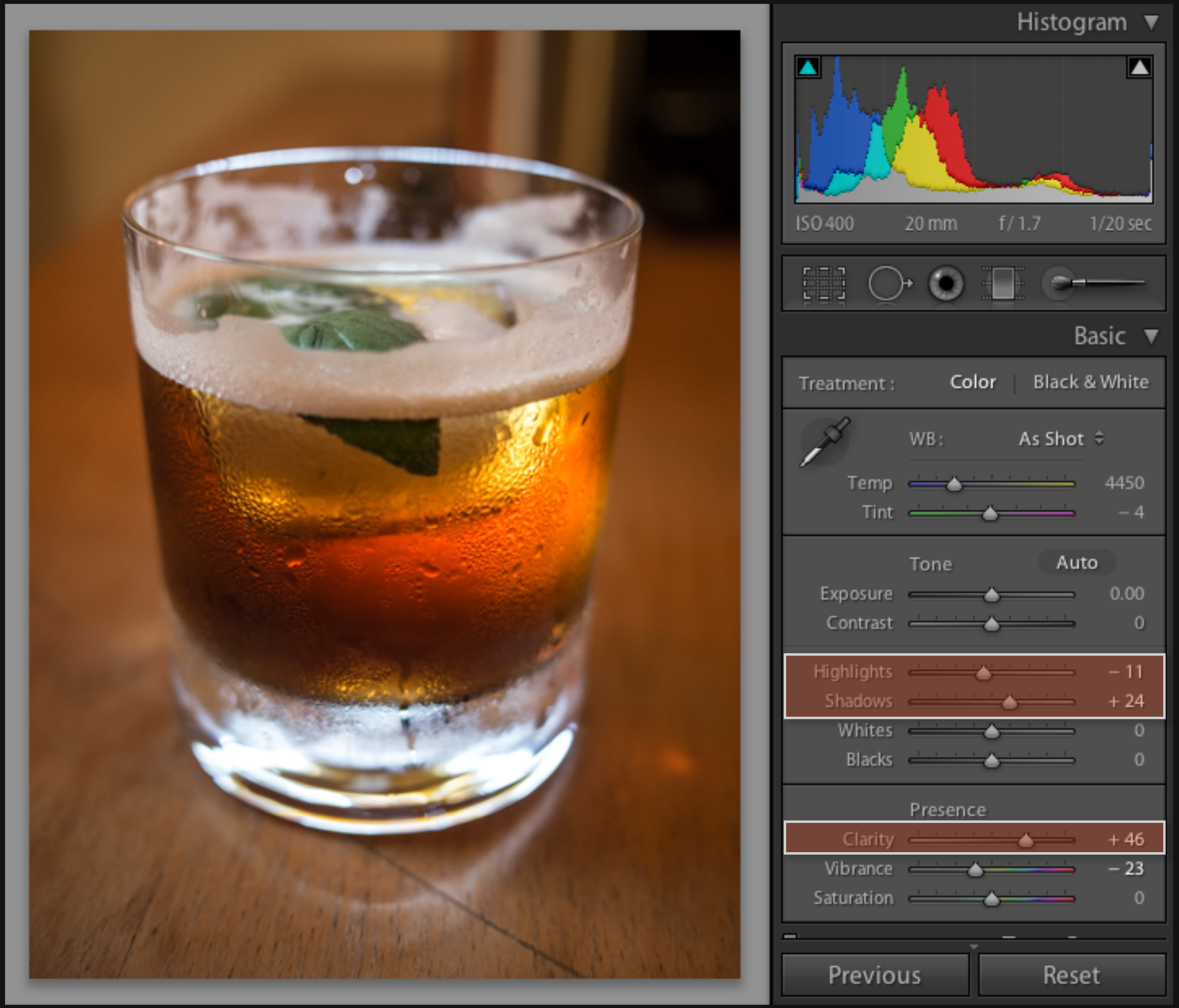
10x faster *(vs. reference)*

Halide: 60 lines

1 intern-day

20x faster (vs. reference)

2x faster (vs. Adobe)



Local Laplacian Filters

prototype for Adobe Photoshop Camera Raw / Lightroom

Reference: 300 lines C++

Adobe: 1500 lines

3 months of work

10x faster *(vs. reference)*

Halide: 60 lines

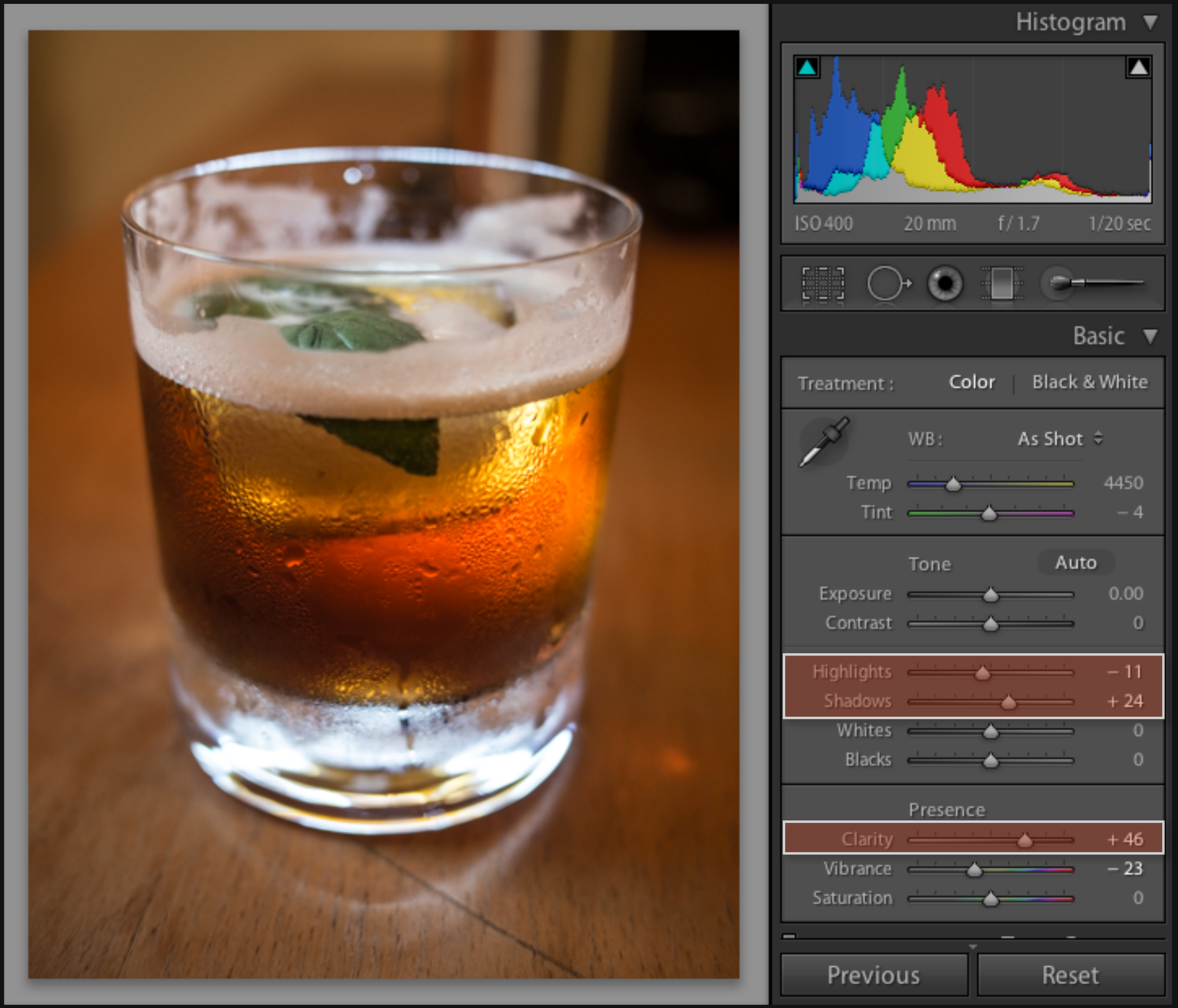
1 intern-day

20x faster (vs. reference)

2x faster (vs. Adobe)

GPU: 90x faster (vs. reference)

9x faster (vs. Adobe)



Local Laplacian Filters

prototype for Adobe Photoshop Camera Raw / Lightroom

Reference: 300 lines C++

Adobe: 1500 lines

3 months of work

10x faster *(vs. reference)*

Halide: 60 lines

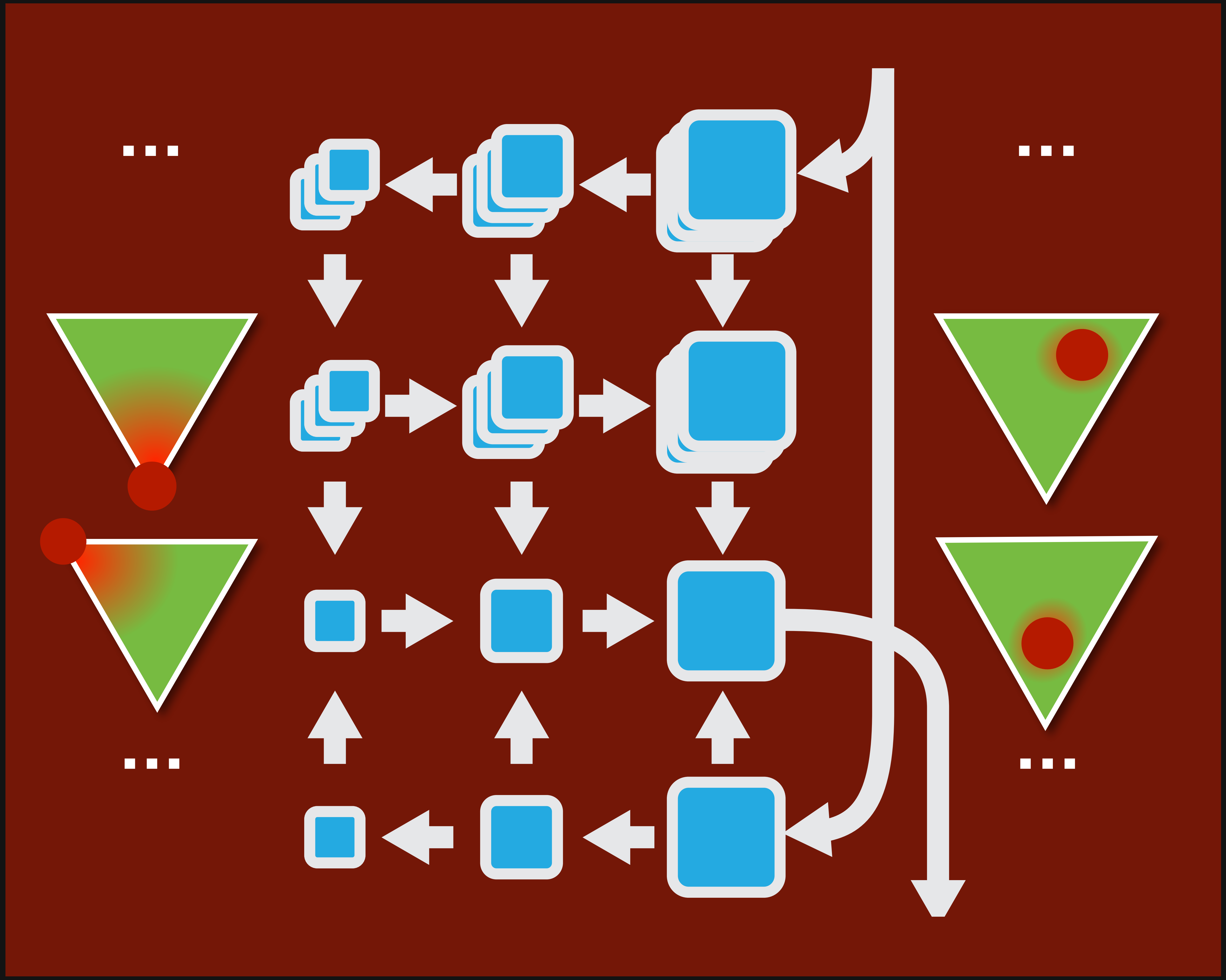
1 intern-day

20x faster (vs. reference)

2x faster (vs. Adobe)

GPU: 90x faster (vs. reference)

9x faster (vs. Adobe)



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 ×
Gaussian Blur	1.5 ×	5 ×
FFT (vs. FFTW)	1.5 ×	*10s*
BLAS (vs. Eigen)	1 ×	*100s*

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 ×

Current status

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

Google

65 active developers

> **200** pipelines

10s of kLOC in production

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

HDR+
Glass
Nexus devices

>20 companies
on Halide-Dev

***n* secret/unannounced projects**

Today's agenda

the big ideas in Halide

Now: **writing & optimizing real code**

Hello world (brightness)

Gaussian blur - *3x OpenCV*

Simple enhancement pipeline - *6x OpenCV*

break

MATLAB integration

IIR filter

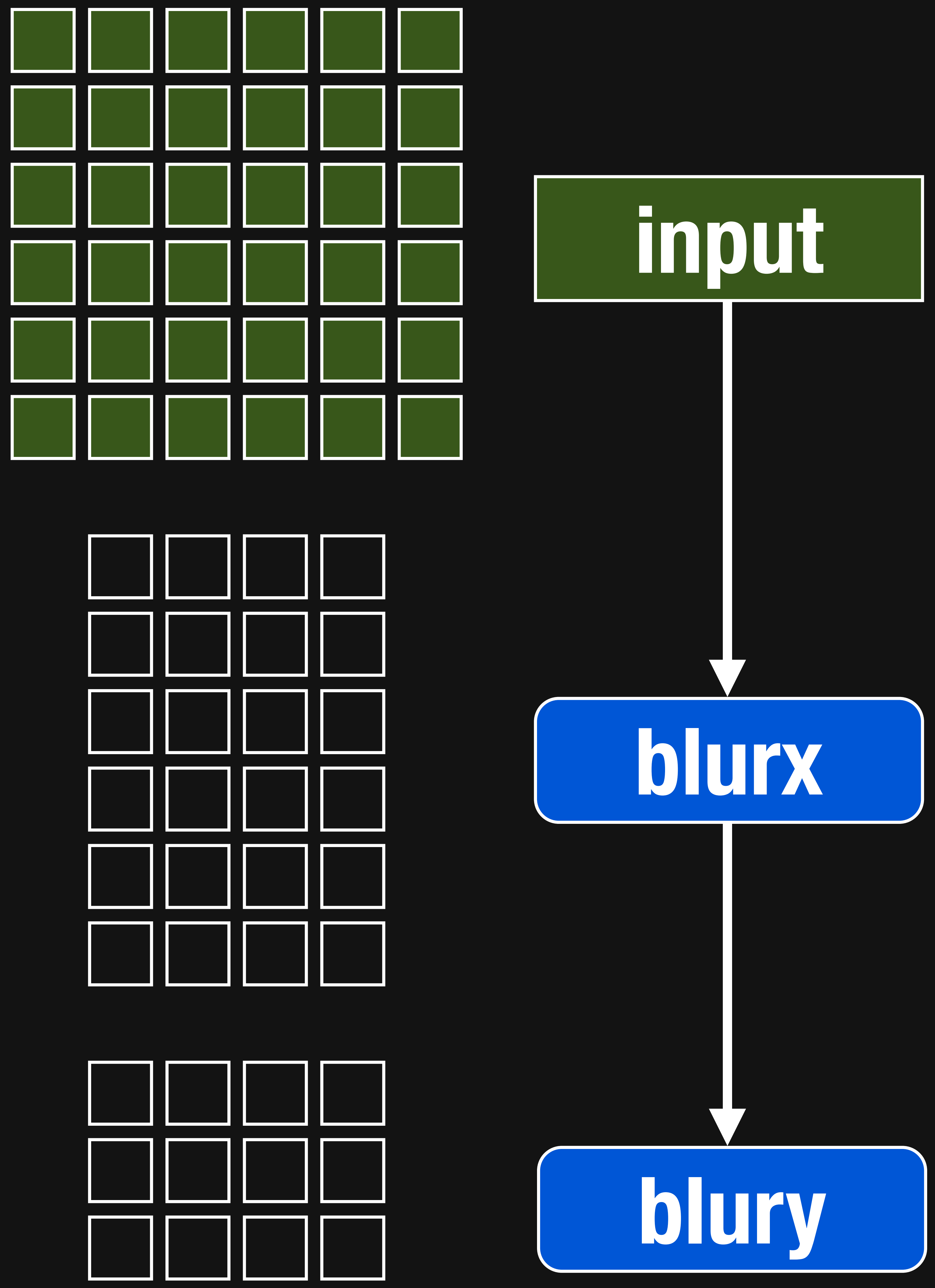
CNN layers

break

GPU scheduling

Finally: **real-time HOG on a phone**

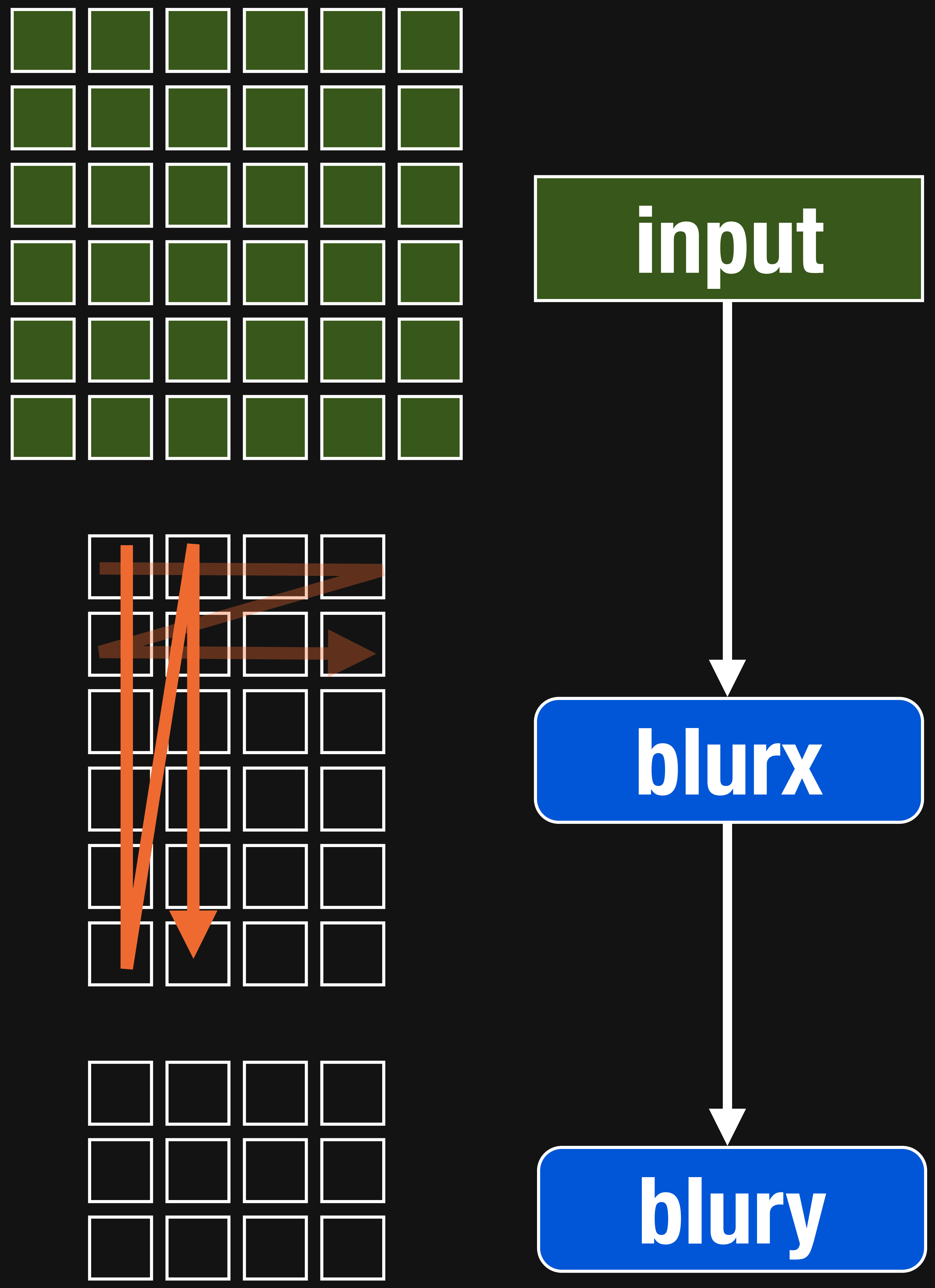
The schedule defines intra-stage order, inter-stage interleaving



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

For each stage:

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

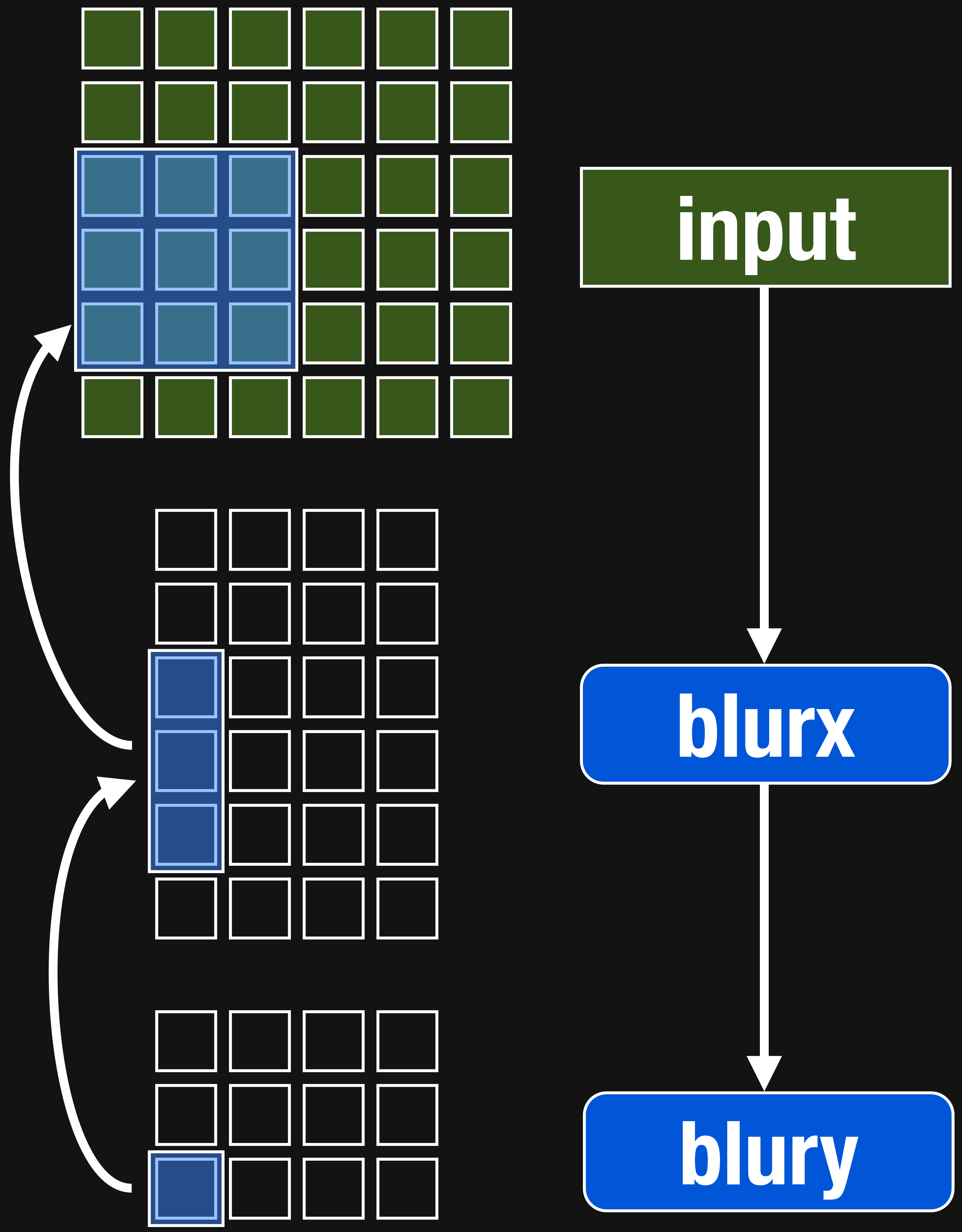


The schedule defines intra-stage order, inter-stage interleaving

For each stage:

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

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



The schedule defines order & parallelism within stages

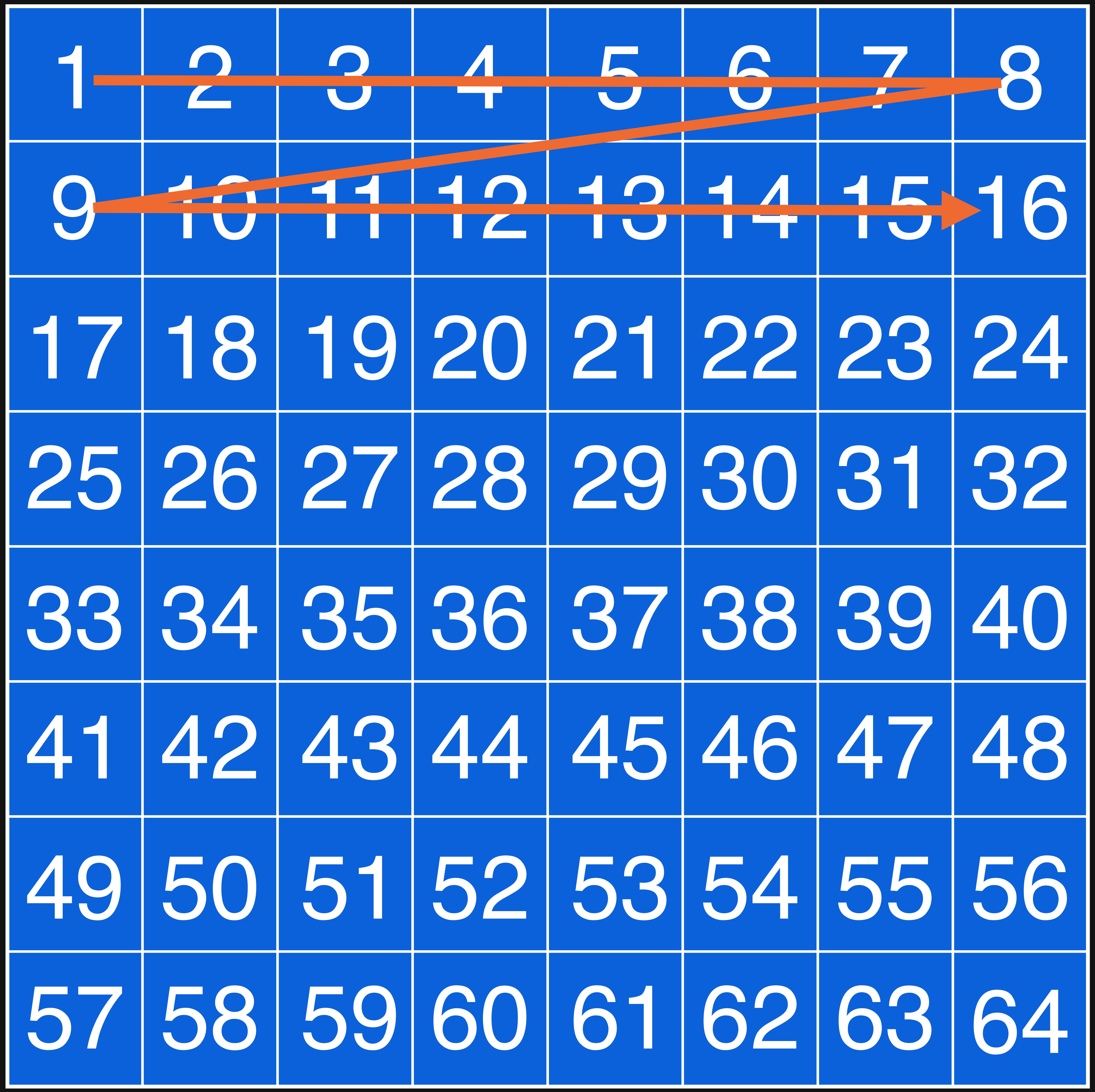
The schedule defines order & parallelism within stages

Serial y,
Serial x

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48
49	50	51	52	53	54	55	56
57	58	59	60	61	62	63	64

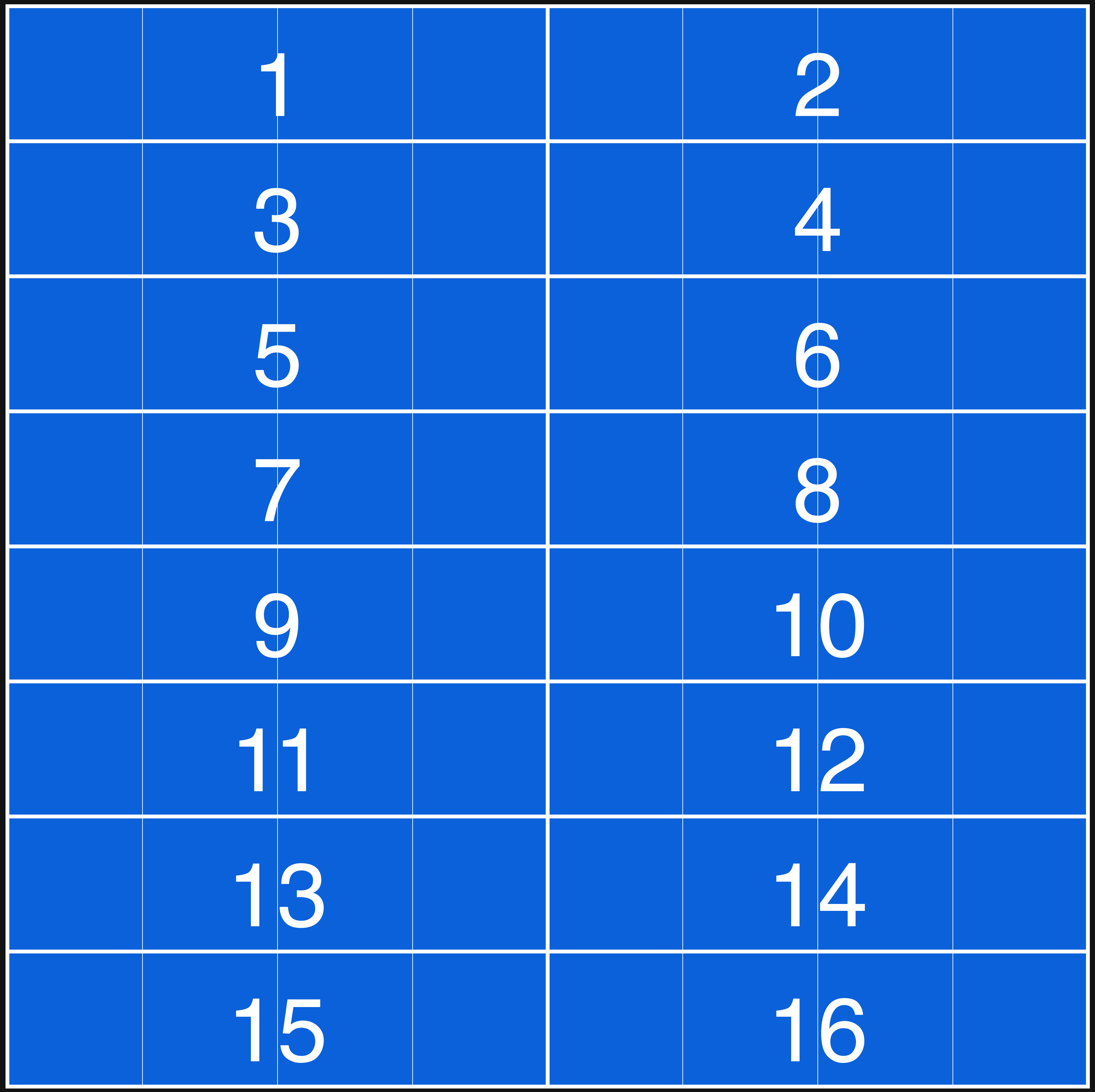
The schedule defines order & parallelism within stages

Serial y,
Serial x



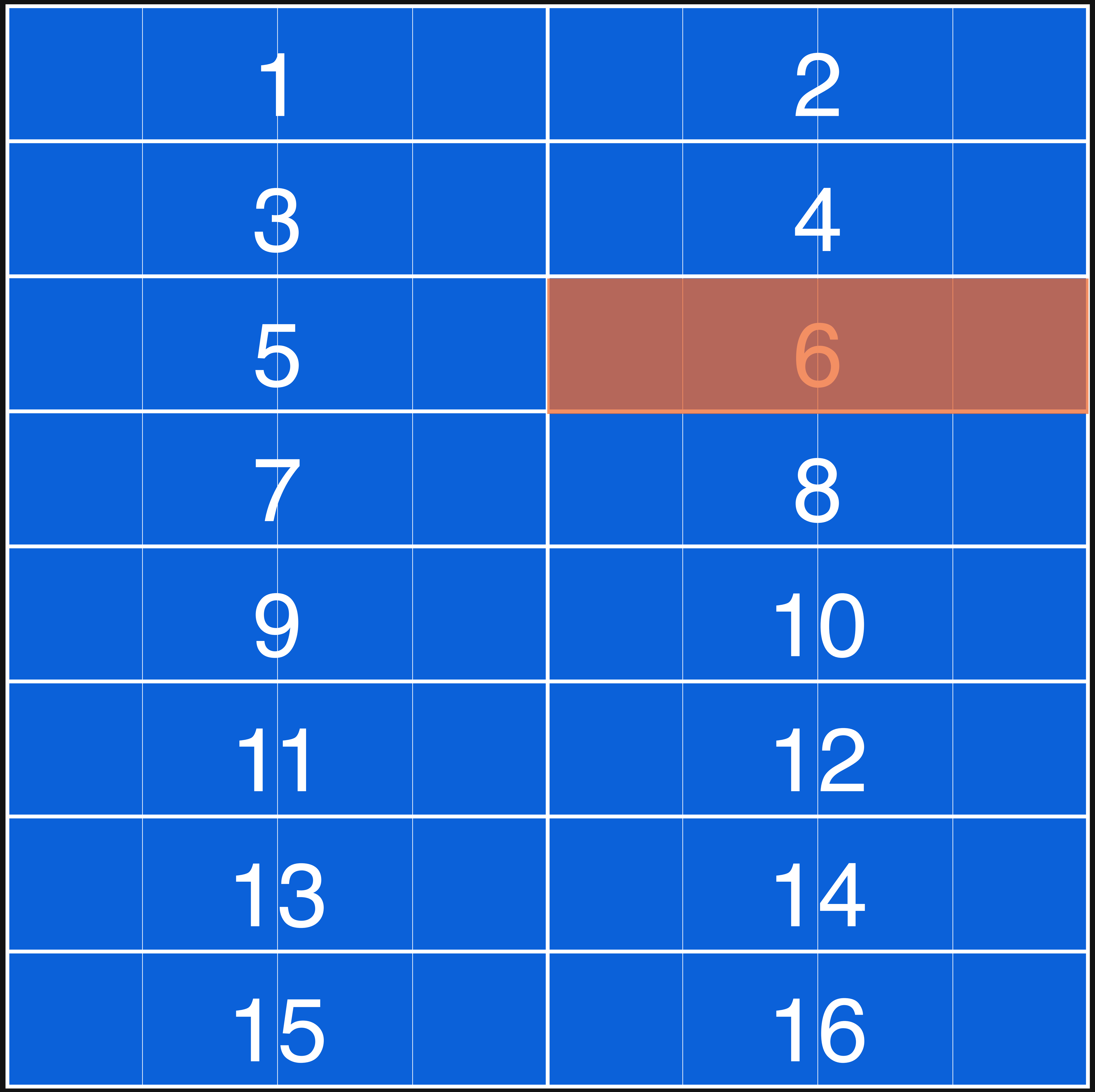
The schedule defines order & parallelism within stages

Serial y,
Vectorize x by 4



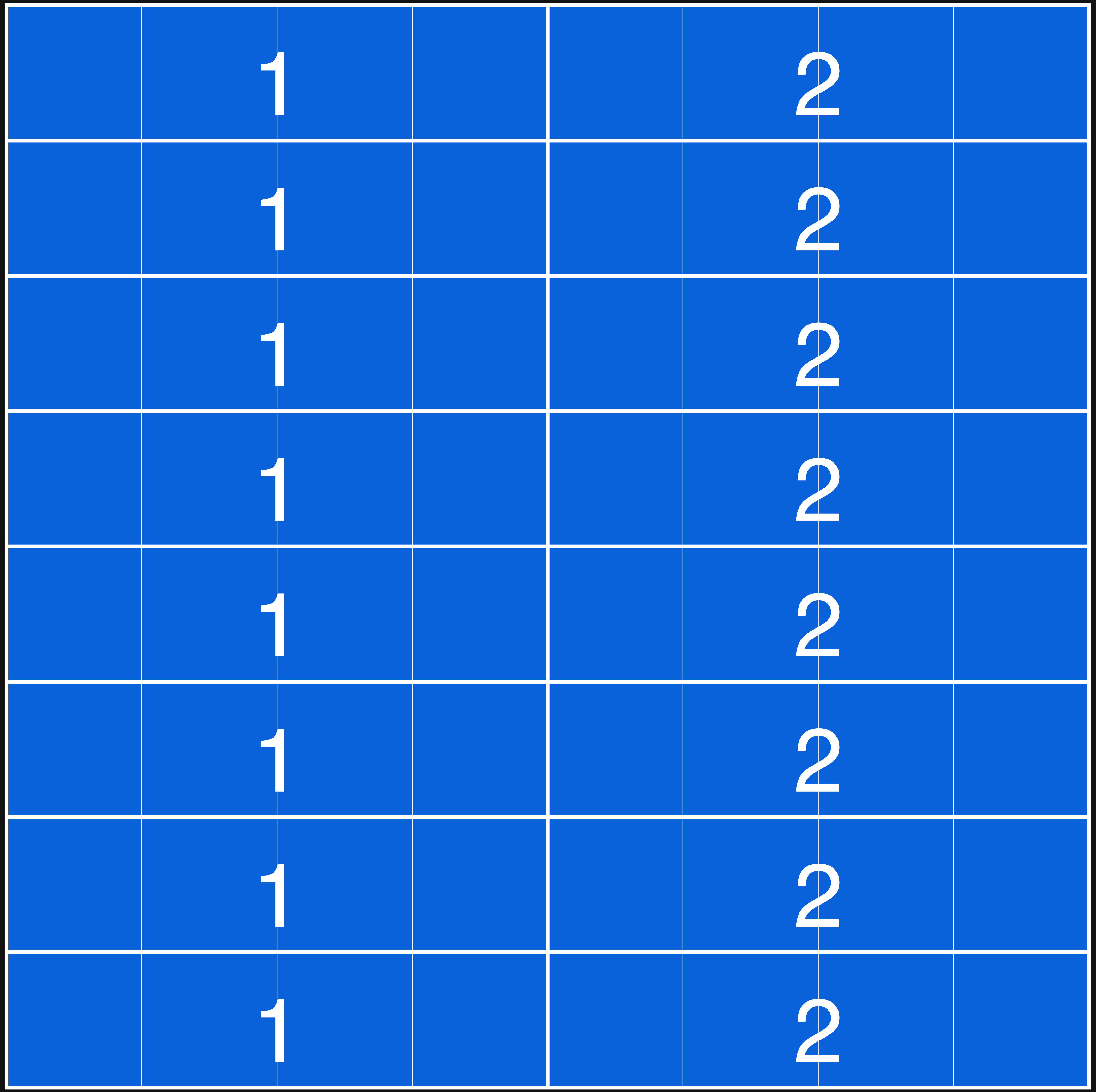
The schedule defines order & parallelism within stages

Serial y,
Vectorize x by 4



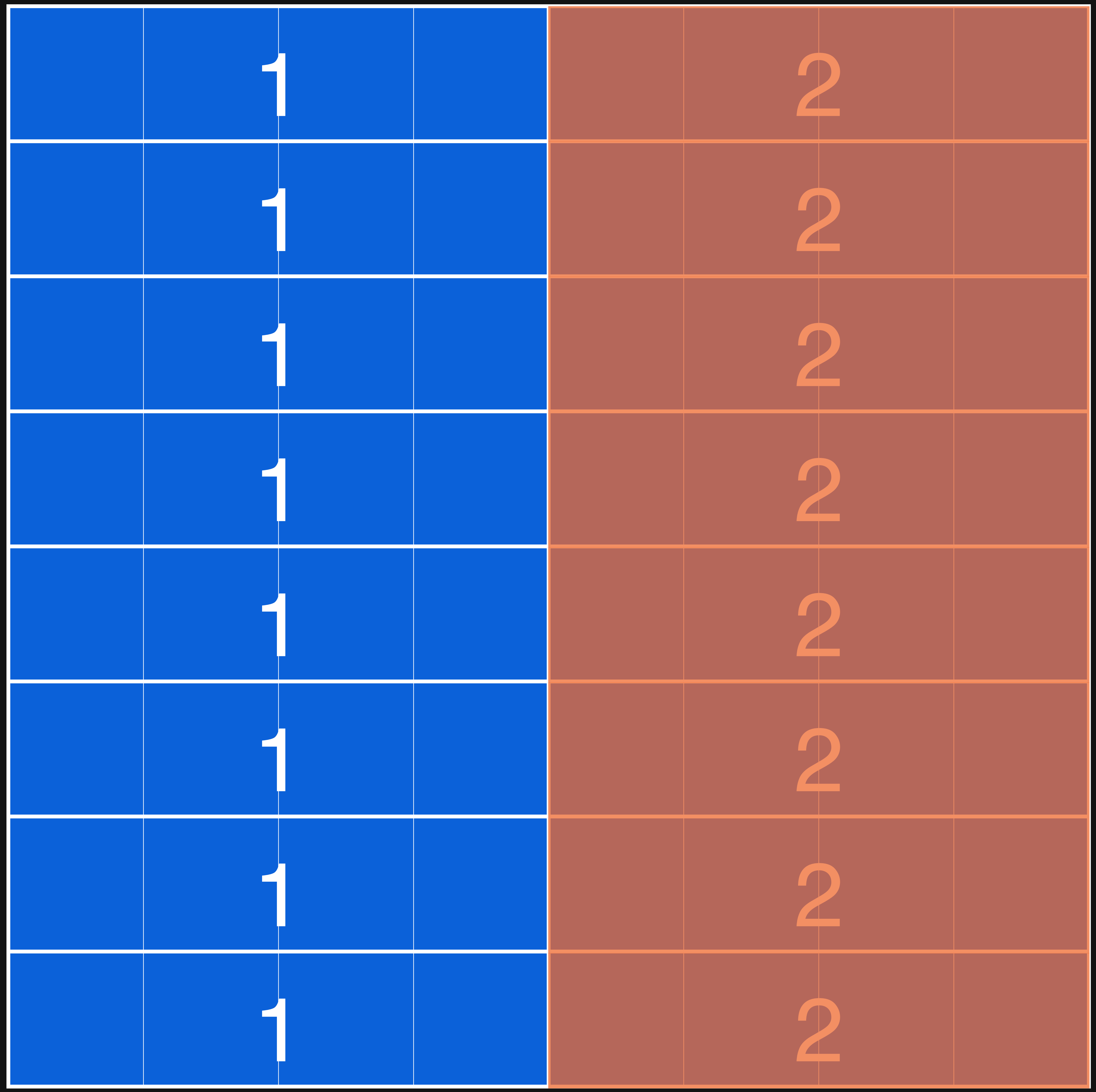
The schedule defines order & parallelism within stages

Parallel y,
Vectorize x by 4



The schedule defines order & parallelism within stages

Parallel y,
Vectorize x by 4



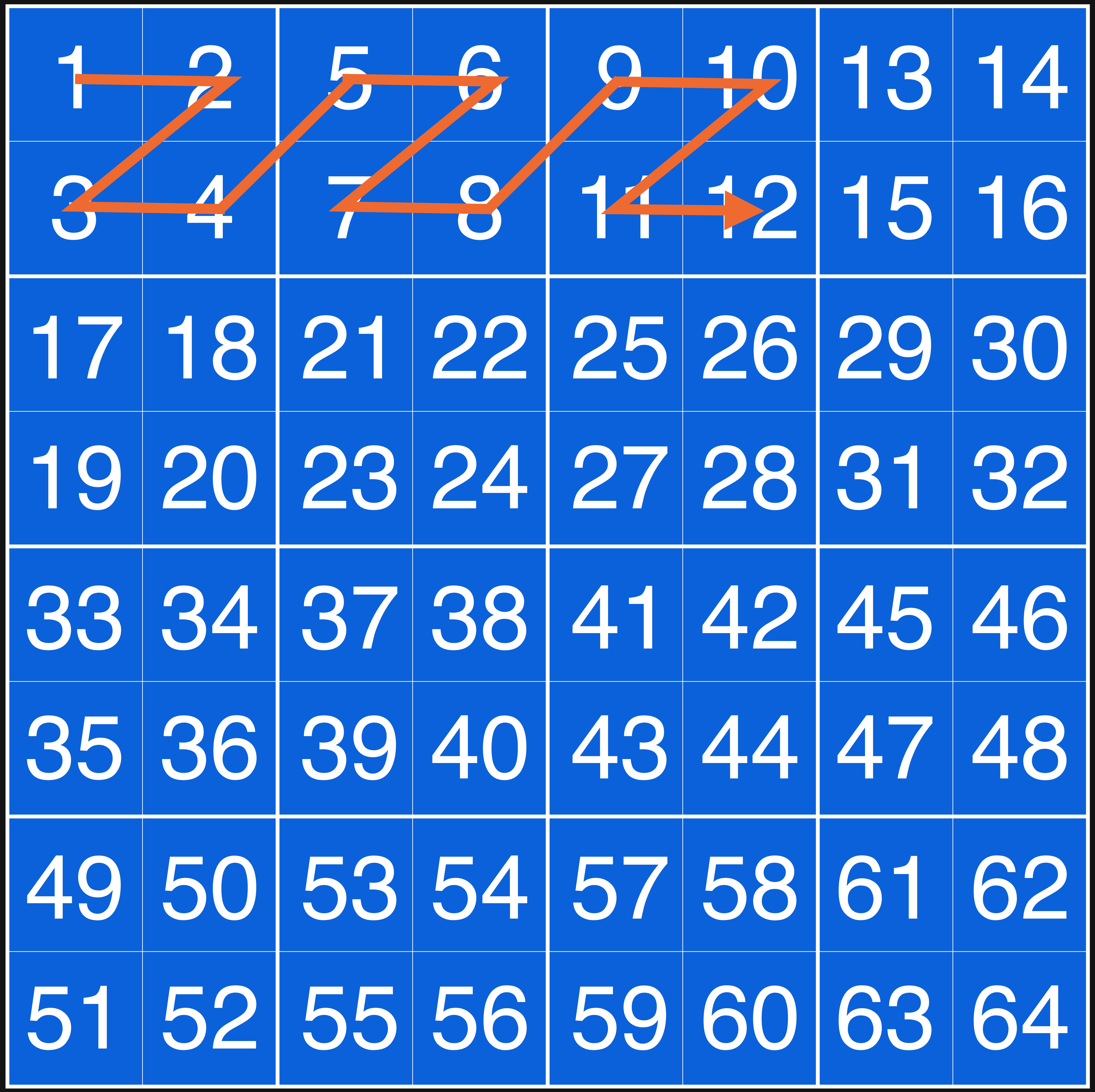
The schedule defines order & parallelism within stages

Split x by 2**,**
Split y by 2**.**

1	2	5	6	9	10	13	14
3	4	7	8	11	12	15	16
17	18	21	22	25	26	29	30
19	20	23	24	27	28	31	32
33	34	37	38	41	42	45	46
35	36	39	40	43	44	47	48
49	50	53	54	57	58	61	62
51	52	55	56	59	60	63	64

The schedule defines order & parallelism within stages

Split x by 2,
Split y by 2.
Serial y$_{\text{outer}}$,
Serial x$_{\text{outer}}$,
Serial y$_{\text{inner}}$,
Serial x$_{\text{inner}}$



Domain order defines a **loop nest** for each function

Domain order defines a **loop nest** for each function

```
brighten(x, y, c) = …
```

Domain order defines a **loop nest** for each function

```
brighten(x, y, c) = …
```

Default:

Serial c,
Serial y,
Serial x

```
for c:
  for y:
    for x:
      brighten(...) = ...
```

Parallel marks a loop to be multithreaded

```
brighten(x, y, c) = …

brighten.parallel(y)
```

```
for c:
  parallel y:
    for x:
      brighten(...) = ...
```

Parallel marks a loop to be multithreaded

```
brighten(x, y, c) = …

brighten.parallel(y)
        .vectorize(x, 8)
```

```
for c:
  parallel y:
    for x:
      vectorized x.v in [0,7]:
        brighten(...) = ...
```

Parallel marks a loop to be multithreaded

```
brighten(x, y, c) = …
```

```
brighten.parallel(y)
        .unroll(x, 4)
```

```
for c:
  parallel y:
    for x:
      unrolled x.v in [0,3]:
        brighten(...) = ...
```

Parallel marks a loop to be multithreaded

brighten(x, y, c) = …

for c:
  for y_o:
    for y_i in [0,63]:
      for x:
        brighten(...) = ...

brighten.split(y, y_o, y_i, 64)

Parallel marks a loop to be multithreaded

brighten(x, y, c) = …

brighten.split(y, y_o, y_i, 64)
        .reorder(c, y_o)

for y_o:
  for c:
    for y_i in [0,63]:
      for x:
        brighten(...) = ...

Today's agenda

the big ideas in Halide

writing & optimizing real code

Now: Hello world (brightness)

Gaussian blur - *3x OpenCV*

Simple enhancement pipeline - *6x OpenCV*

break

MATLAB integration

IIR filter

CNN layers

break

GPU scheduling

Finally: **real-time HOG on a phone**

Today's agenda

the big ideas in Halide

writing & optimizing real code

Now:
- Hello world (brightness)
- Gaussian blur
- Simple enhancement pipeline - *6x OpenCV*

break

- MATLAB integration
- IIR filter
- CNN layers

break

- GPU scheduling

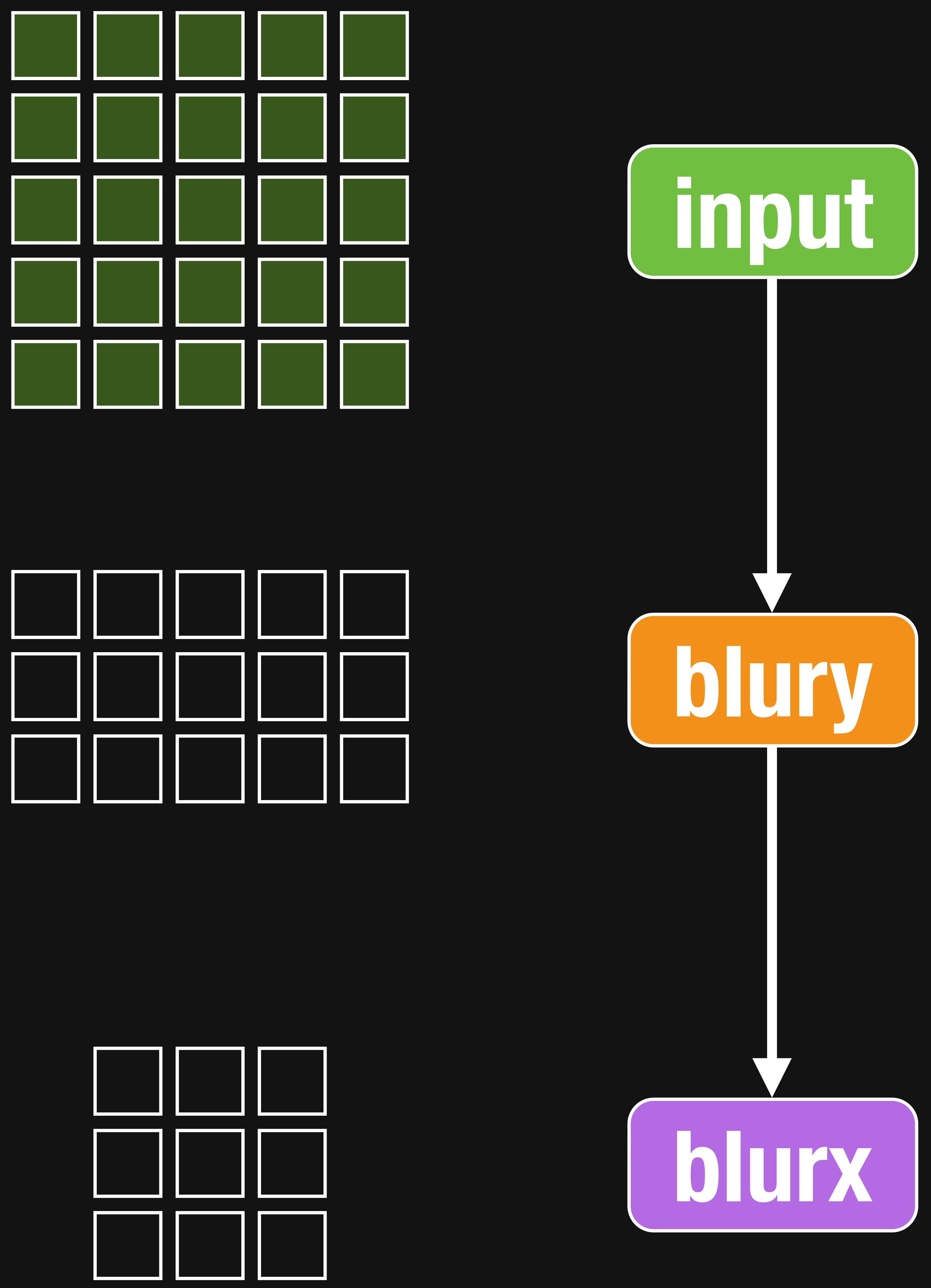
Finally: **real-time HOG on a phone**

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

For each stage:

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

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

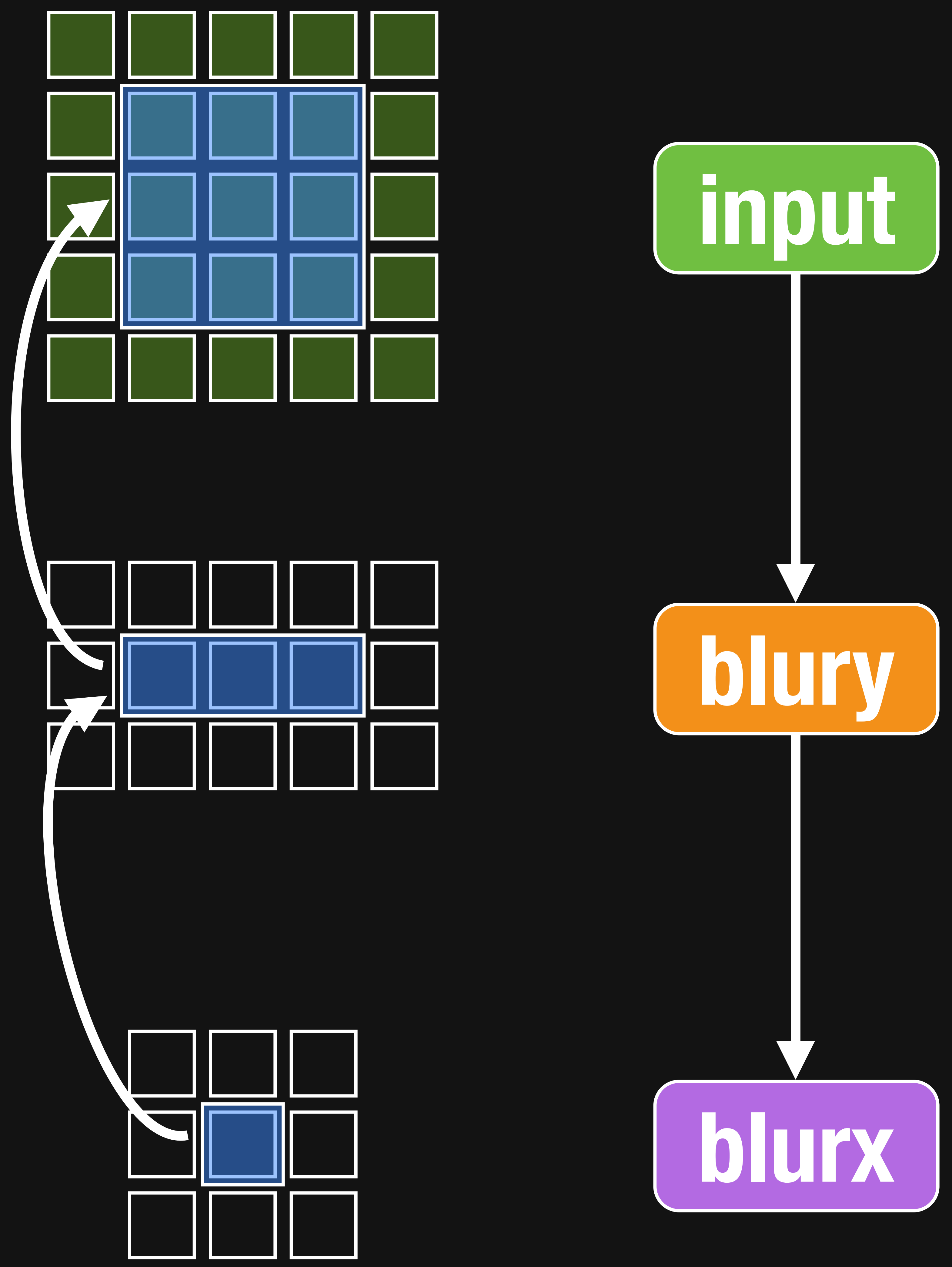


The schedule defines intra-stage order, inter-stage interleaving

For each stage:

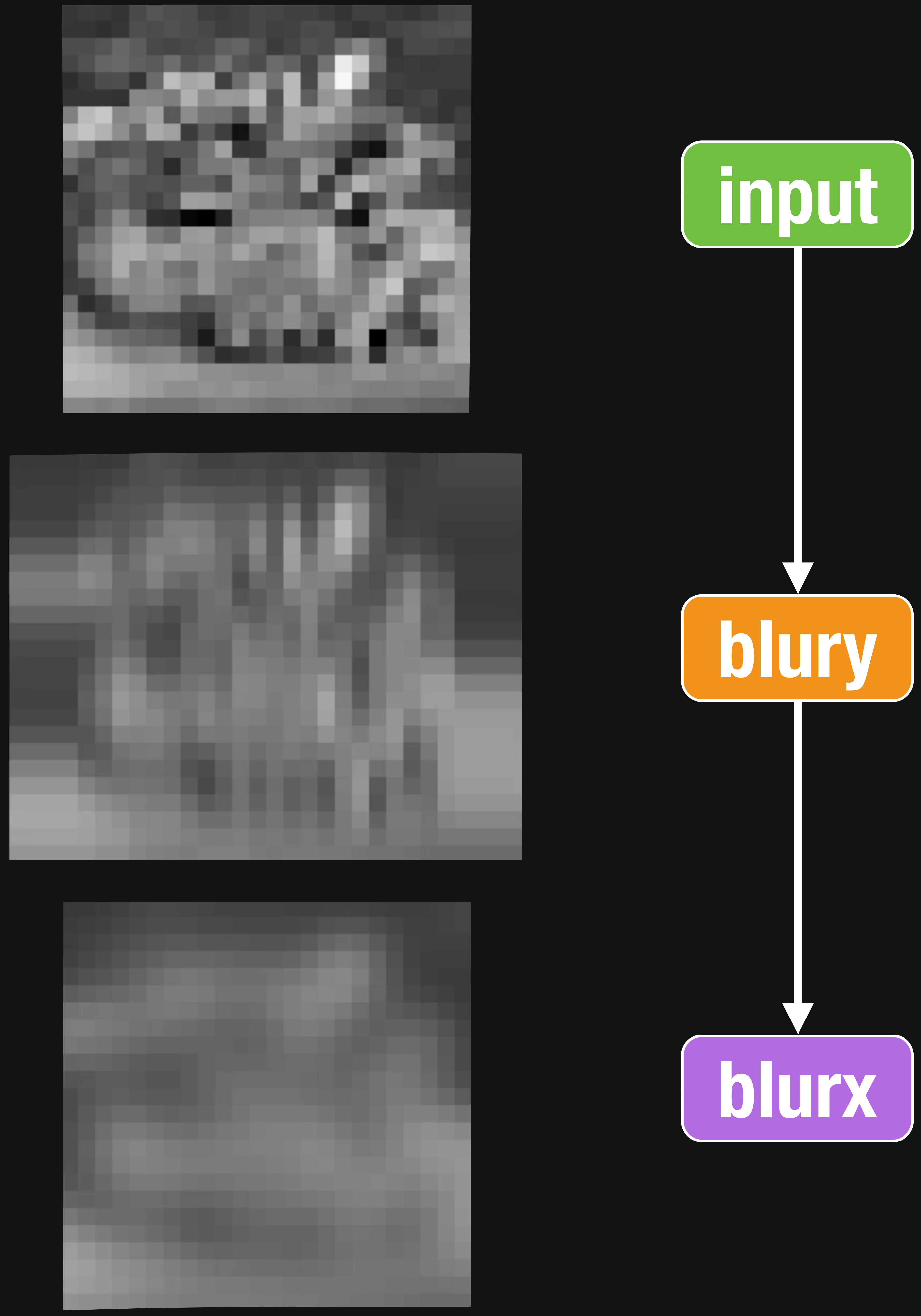
1) In **what order** should we **compute** its **values**?

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

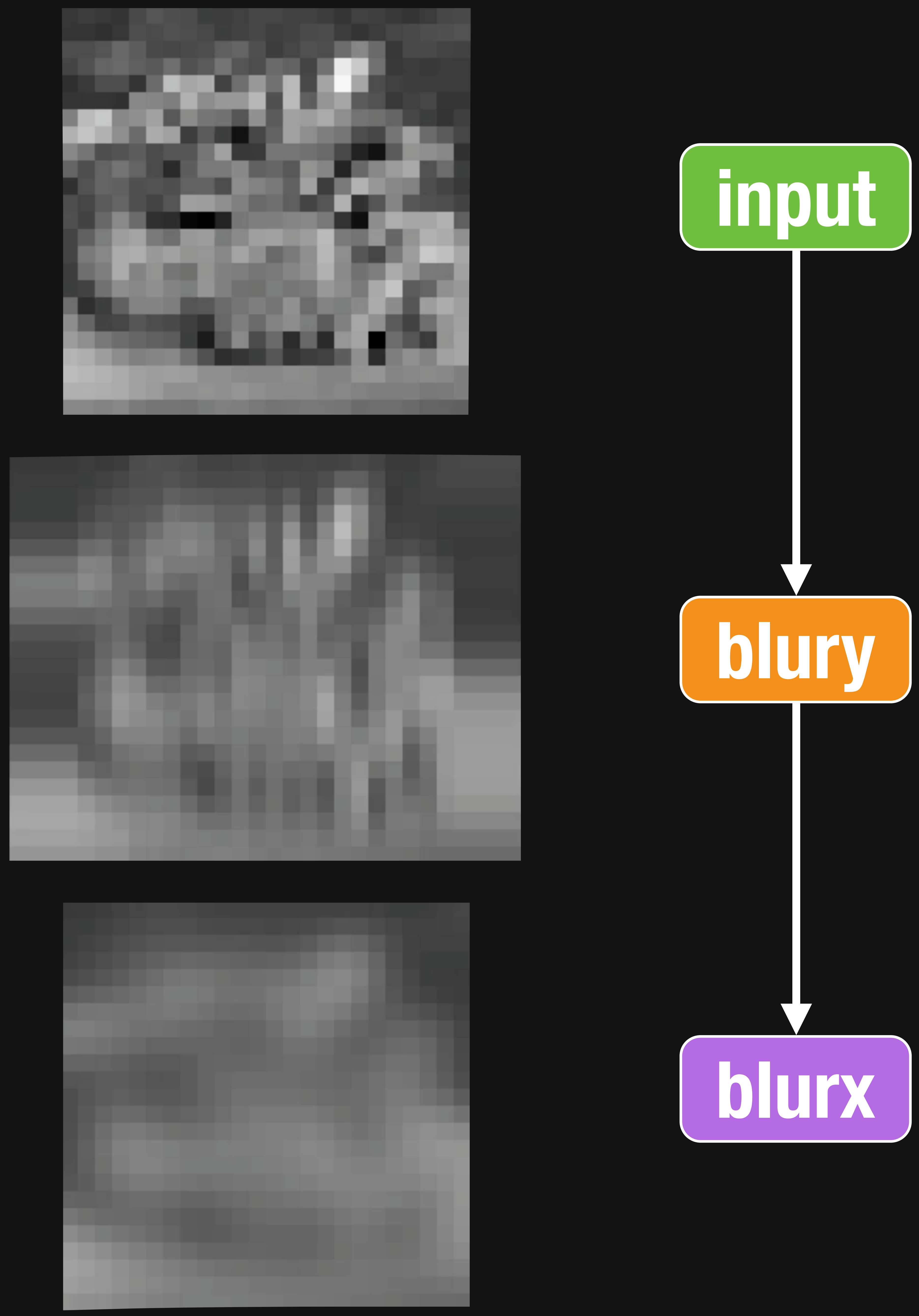


Organizing the algorithm
as a **data-parallel pipeline**
& **loops**

Organizing the algorithm
as a **data-parallel pipeline**
& **loops**



Organizing the algorithm as a **data-parallel pipeline** & **loops**

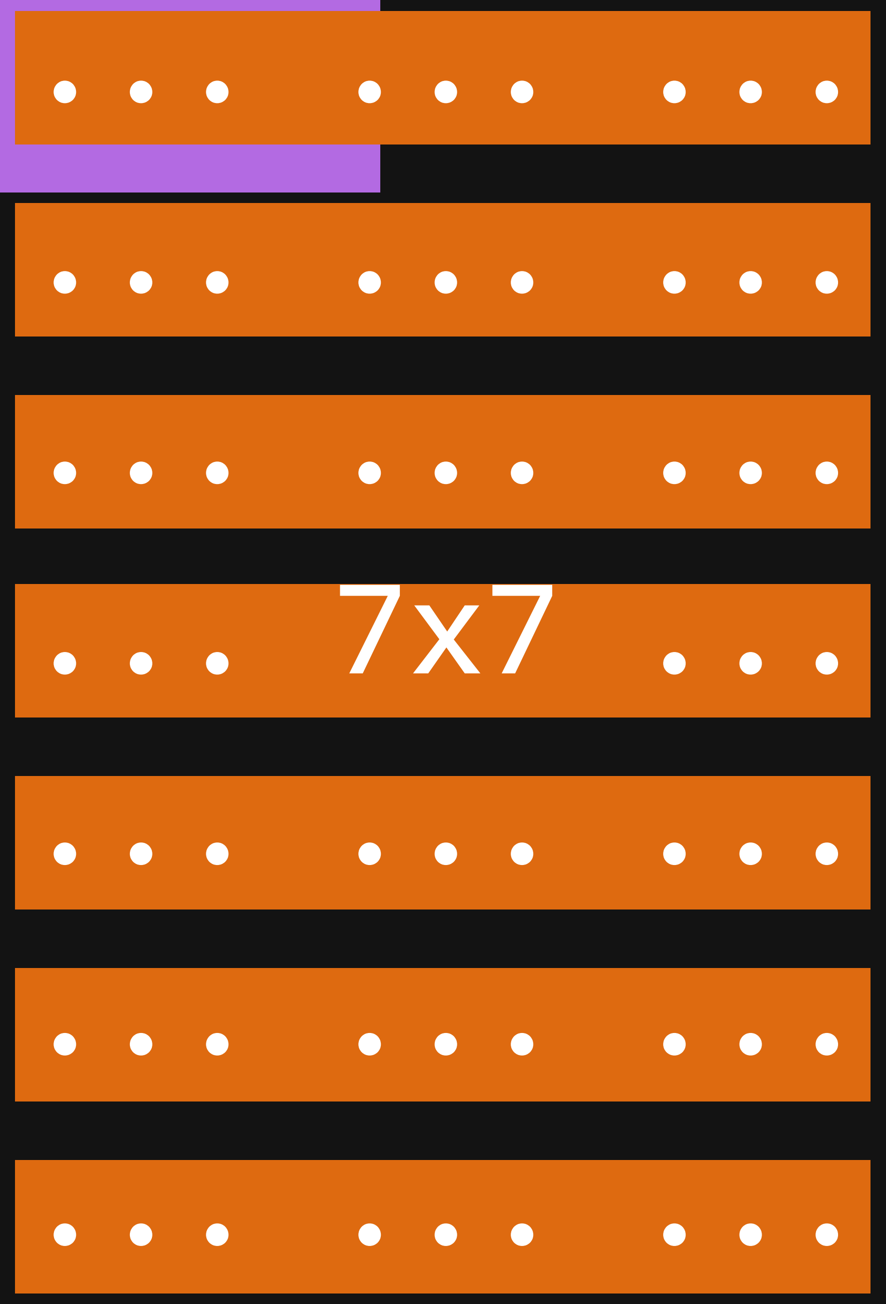



```
compute blury:
  for ...:
    blury(...) = ...
```

```
compute blurx:
  for ...:
    blurx(...) = ...
```

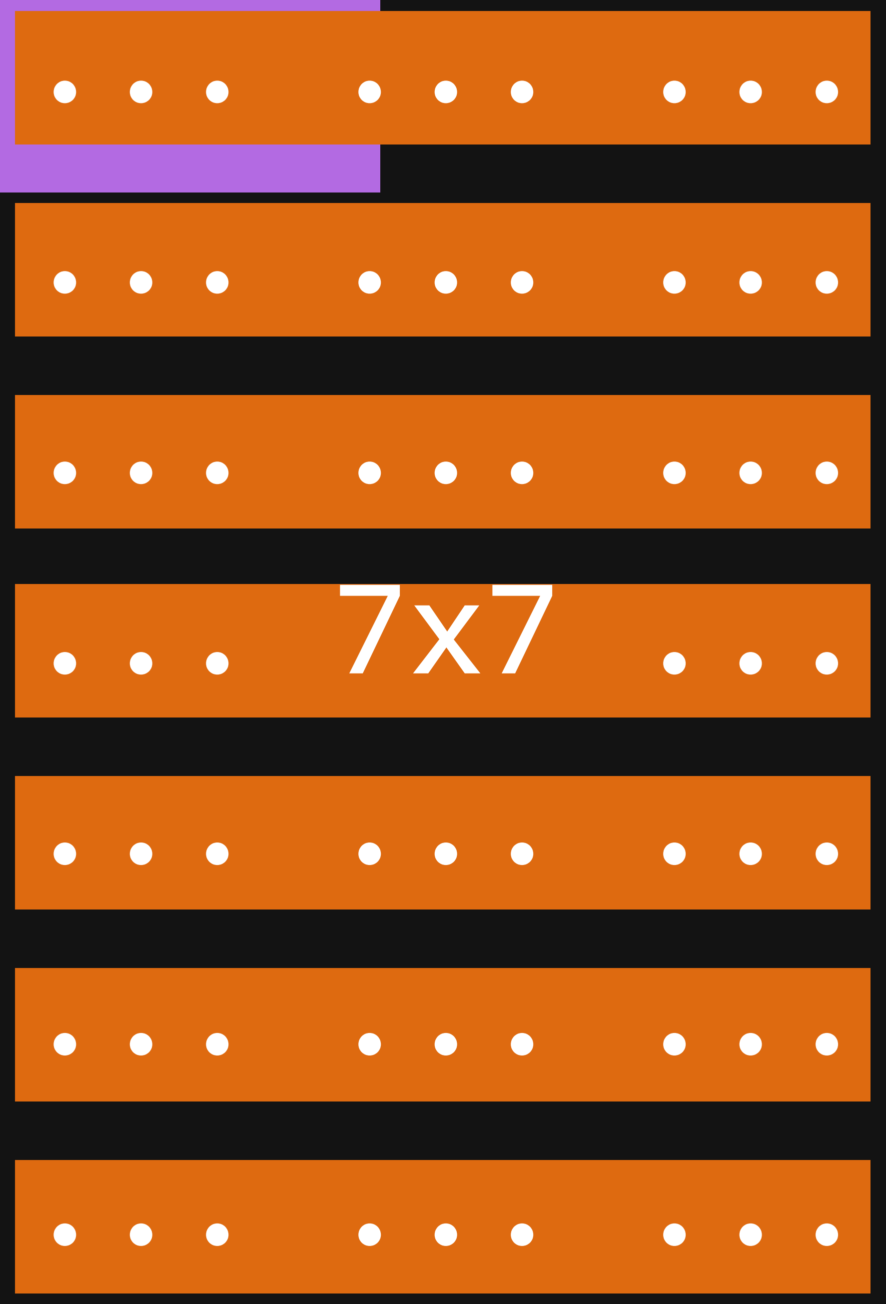
Inline maximizes **locality**, but also **recomputes values**


```
compute blurx:
  for c:
    for y:
      for x:
        blurx(...) =
```



Inline maximizes **locality**, but also **recomputes values**


```
compute blurx:
  for c:
    for y:
      for x:
        blurx(...) =
```



Compute root minimizes **recompute**, but also **locality**


```
compute blury:
  for c:
    for y:
      for x:
        blury(...) = ...
compute blurx:
  for c:
    for y:
      for x:
        blurx(...) = ...
```

Compute root minimizes **recompute**, but also **locality**


```
compute blury:
  for c:
    for y:
      for x:
        blury(...) = ...
compute blurx:
  for c:
    for y:
      for x:
        blurx(...) = ...
```

Compute at `blurx.y` interleaves scanlines for better **locality**


```
compute blurx:
  for c:
    for y:
      compute blury:
        for x:
          blury(...) = ...
      for x:
        blurx(...) = ...
```

Compute at `blurx.y` interleaves scanlines for better **locality**


```
compute blurx:
  for c:
    for y:
      compute blury:
        for x:
          blury(...) = ...
      for x:
        blurx(...) = ...
```

Today's agenda

the big ideas in Halide

writing & optimizing real code

Hello world (brightness)

Now: Gaussian blur - *3x OpenCV*

Simple enhancement pipeline - *6x OpenCV*

break

MATLAB integration

IIR filter

CNN layers

break

GPU scheduling

Finally: **real-time HOG on a phone**