Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines

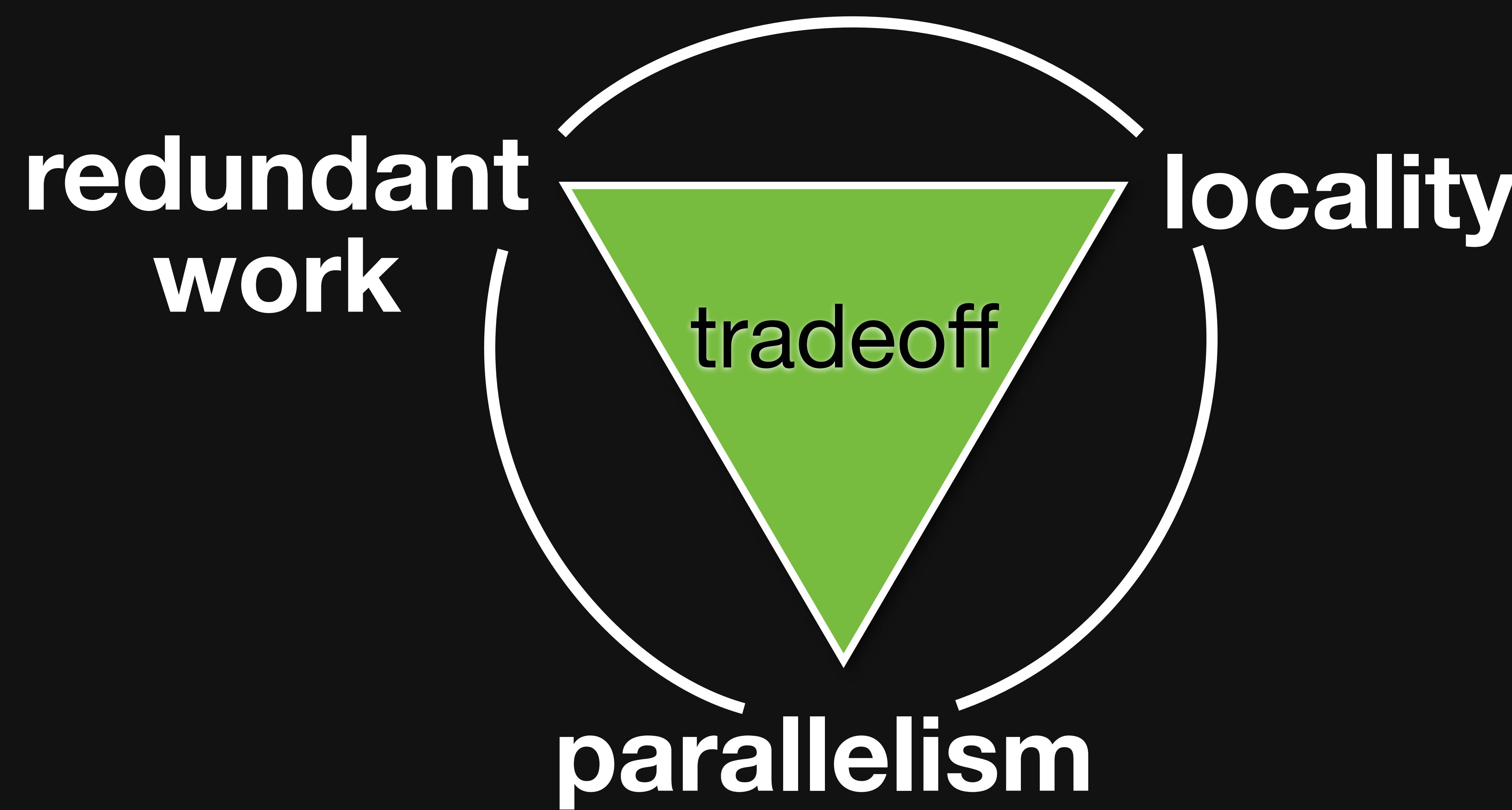
Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, Connely Barnes, Fredo Durand
MIT CSAIL, Adobe, Stanford

High Performance Image Processing

Parallelism
"Moore's law" growth will require exponentially more parallelism.

Locality
Data should move as little as possible.

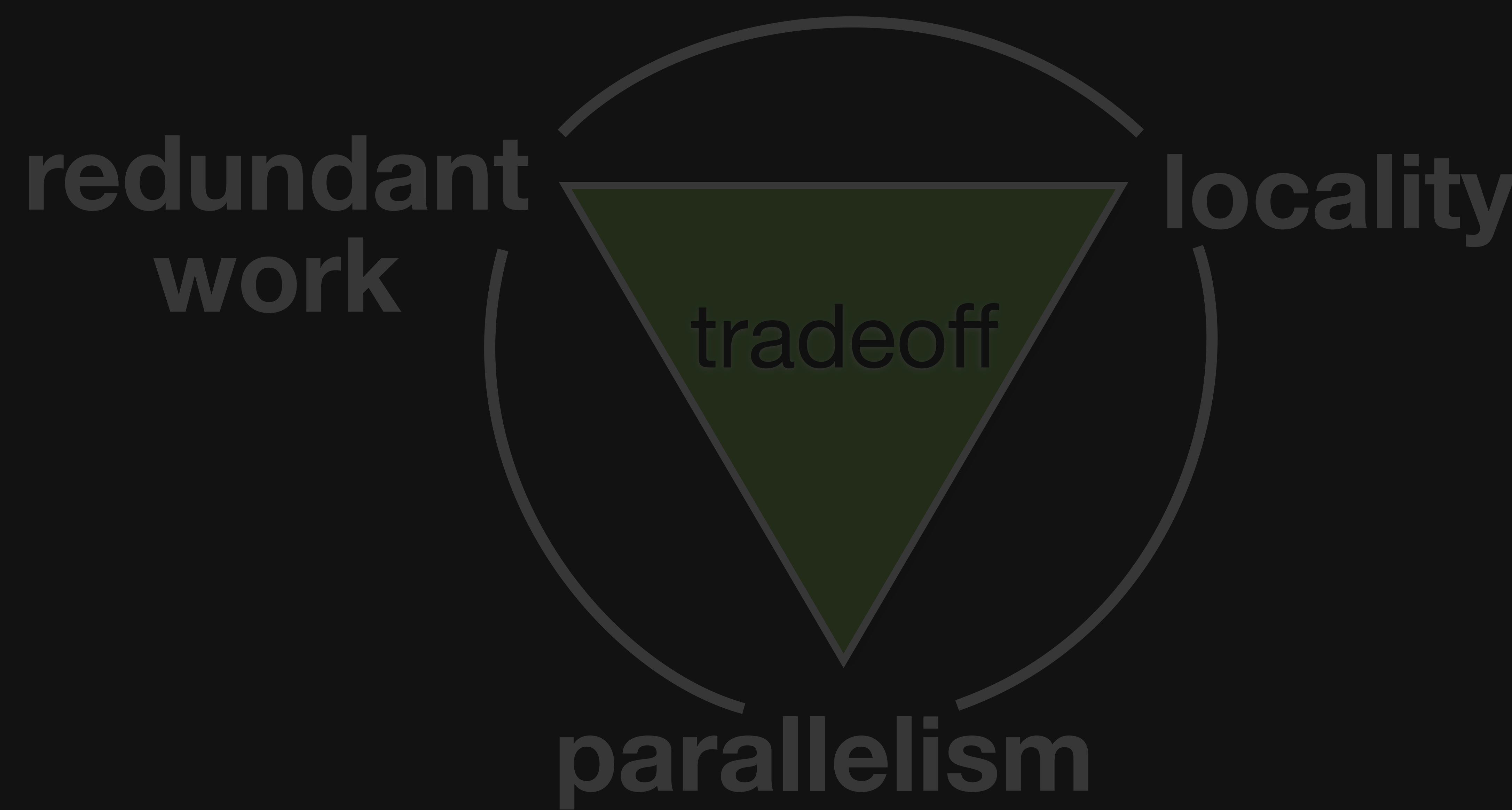
Message #1: Performance requires complex tradeoffs



Where does performance come from?

Program

Hardware



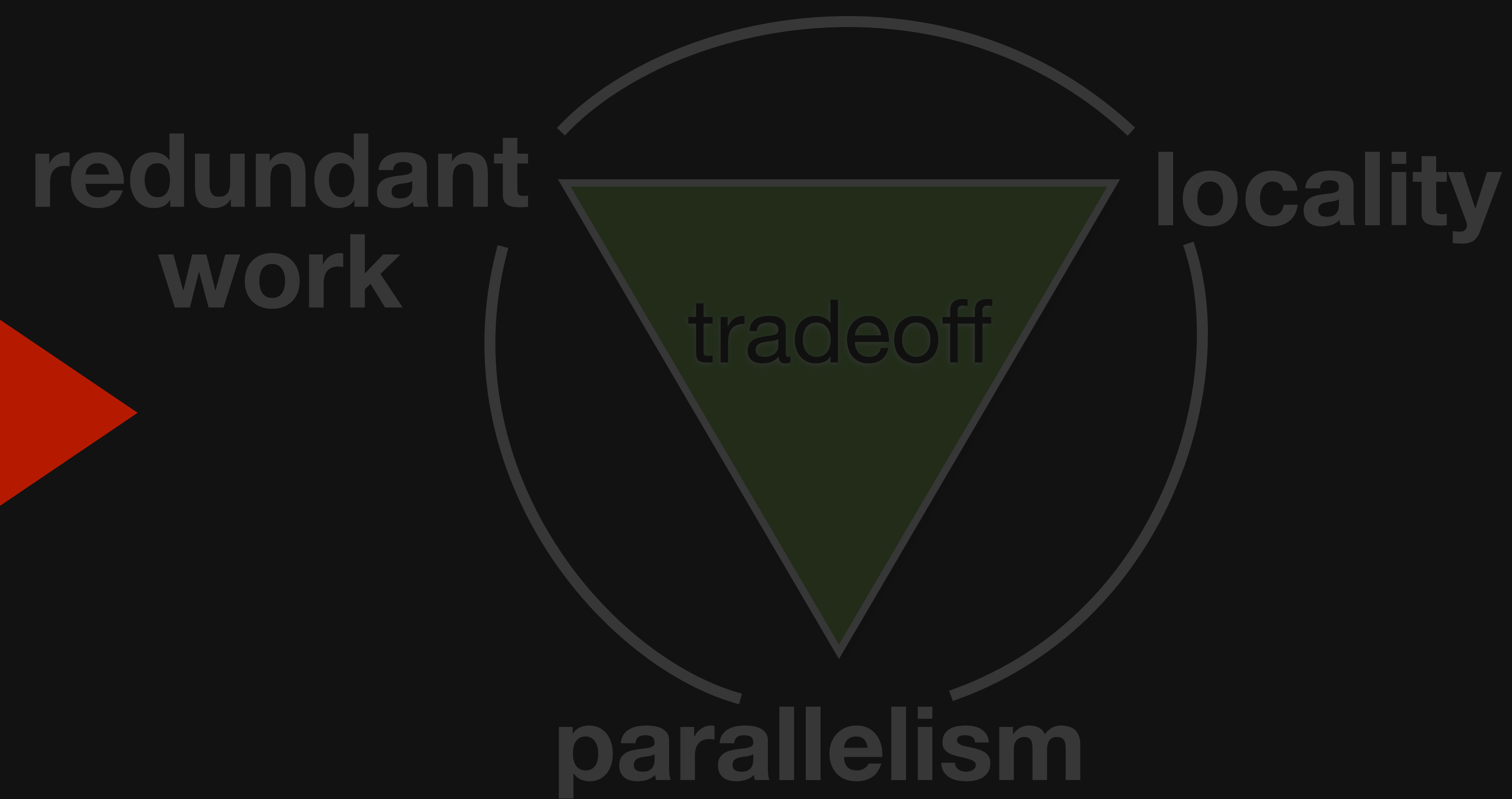
Message #2: organization of computation
is a first-class issue

Program:

Algorithm

Organization of computation

Hardware



Algorithm vs. Organization: 3x3 blur

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

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

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

Algorithm vs. Organization: 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;
}
```

Same algorithm, different organization

One of them is 15x faster

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

Same algorithm, different organization

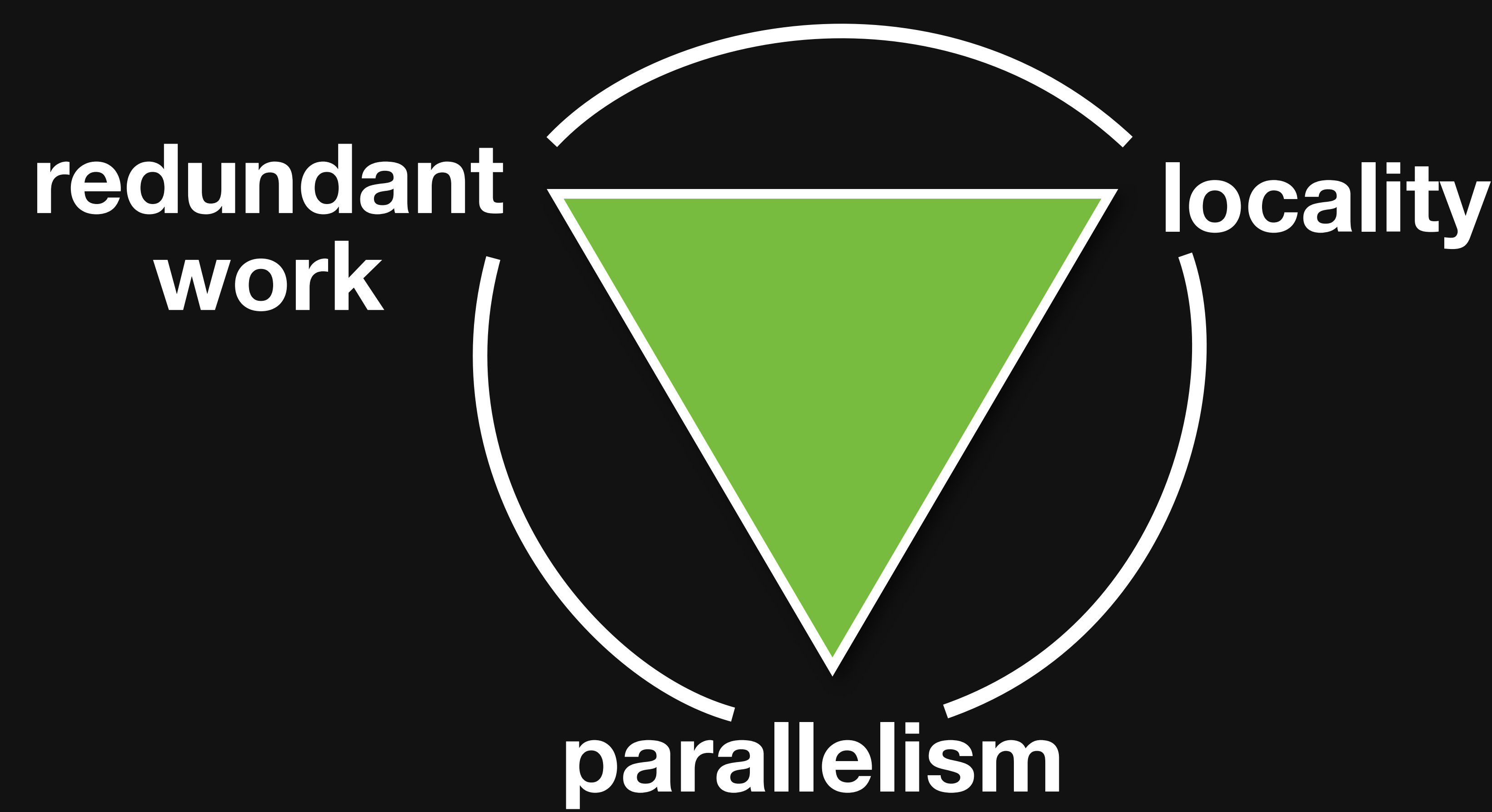
```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile = 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);
}}}}}
```

```
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;
}
```

For a given algorithm, organize to optimize:



(Re)organizing computation is hard

Optimizing parallelism, locality requires
transforming program & data structure.

What transformations are *legal*?

What transformations are *beneficial*?

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

Halide's answer: *decouple* algorithm from schedule

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

Easy for programmers to build pipelines

Easy to specify & explore optimizations
manual or automatic search

Easy for the compiler to generate fast code

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 *schedule*:

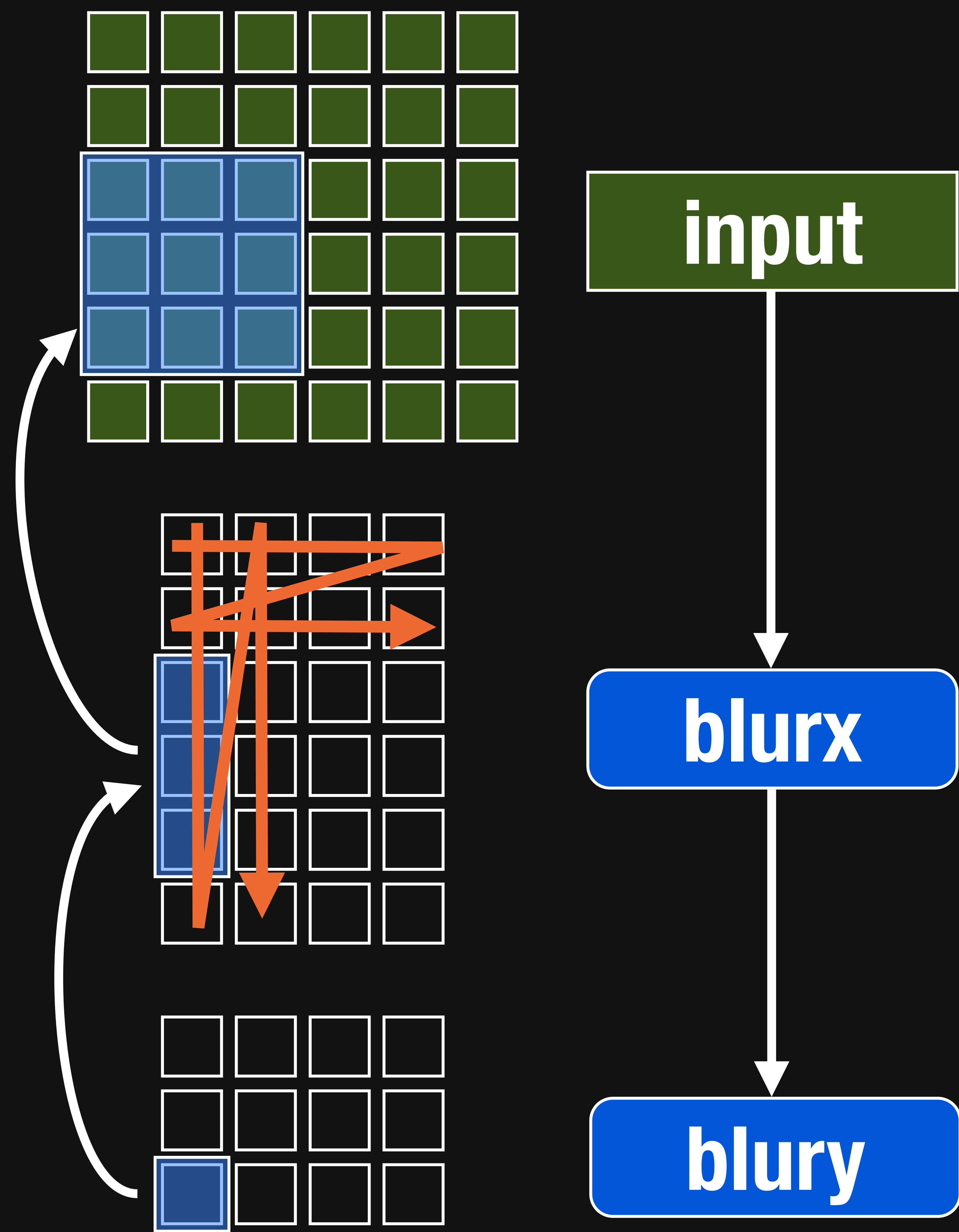
```
blury.tile(x, y, xi, yi, 256, 32).vectorize(xi, 8).parallel(y);
blurx.compute_at(blury, x).store_at(blury, x).vectorize(x, 8);
```

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

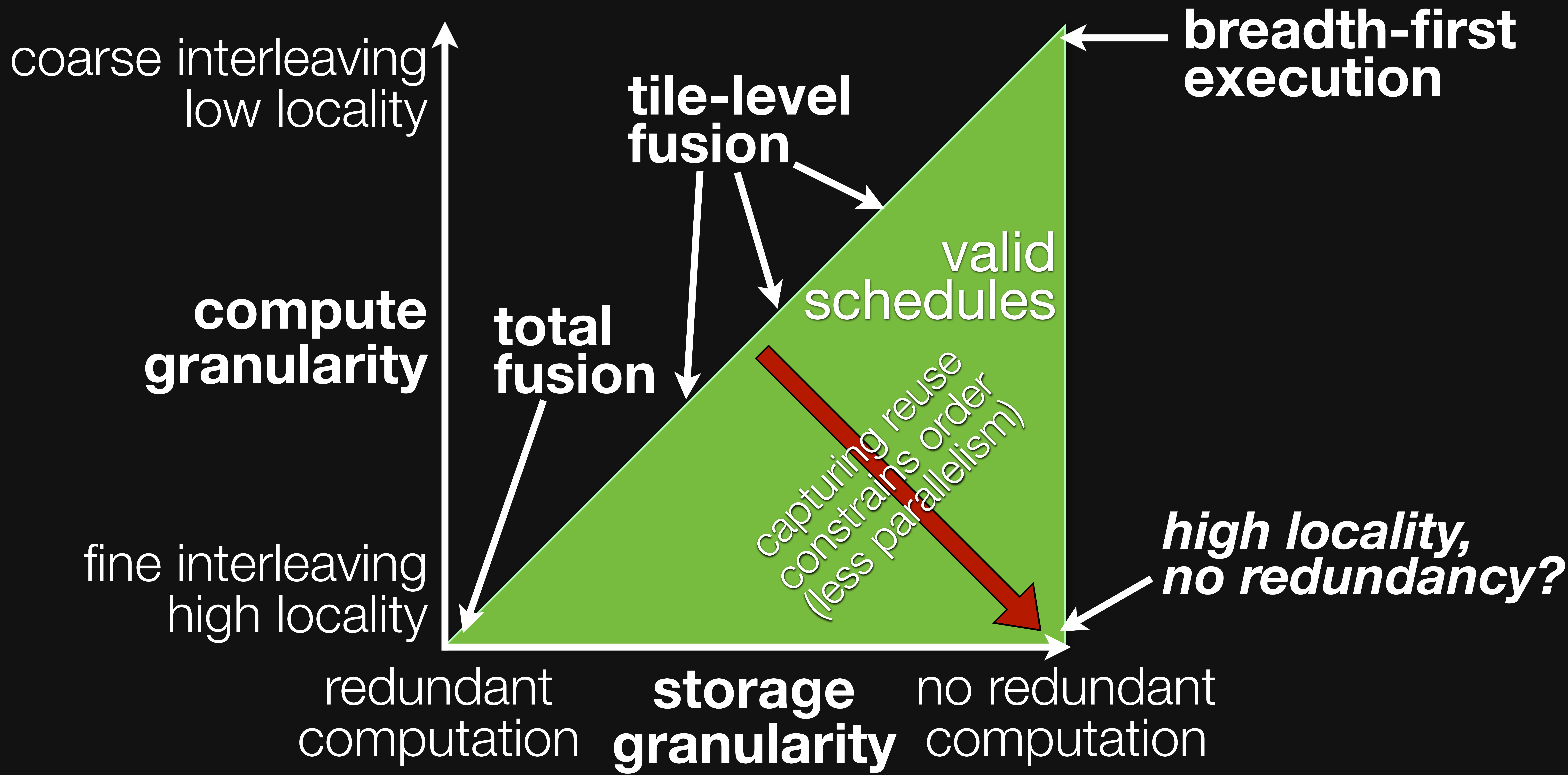
For each stage:

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

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



Tradeoff space modeled by **granularity of interleaving**



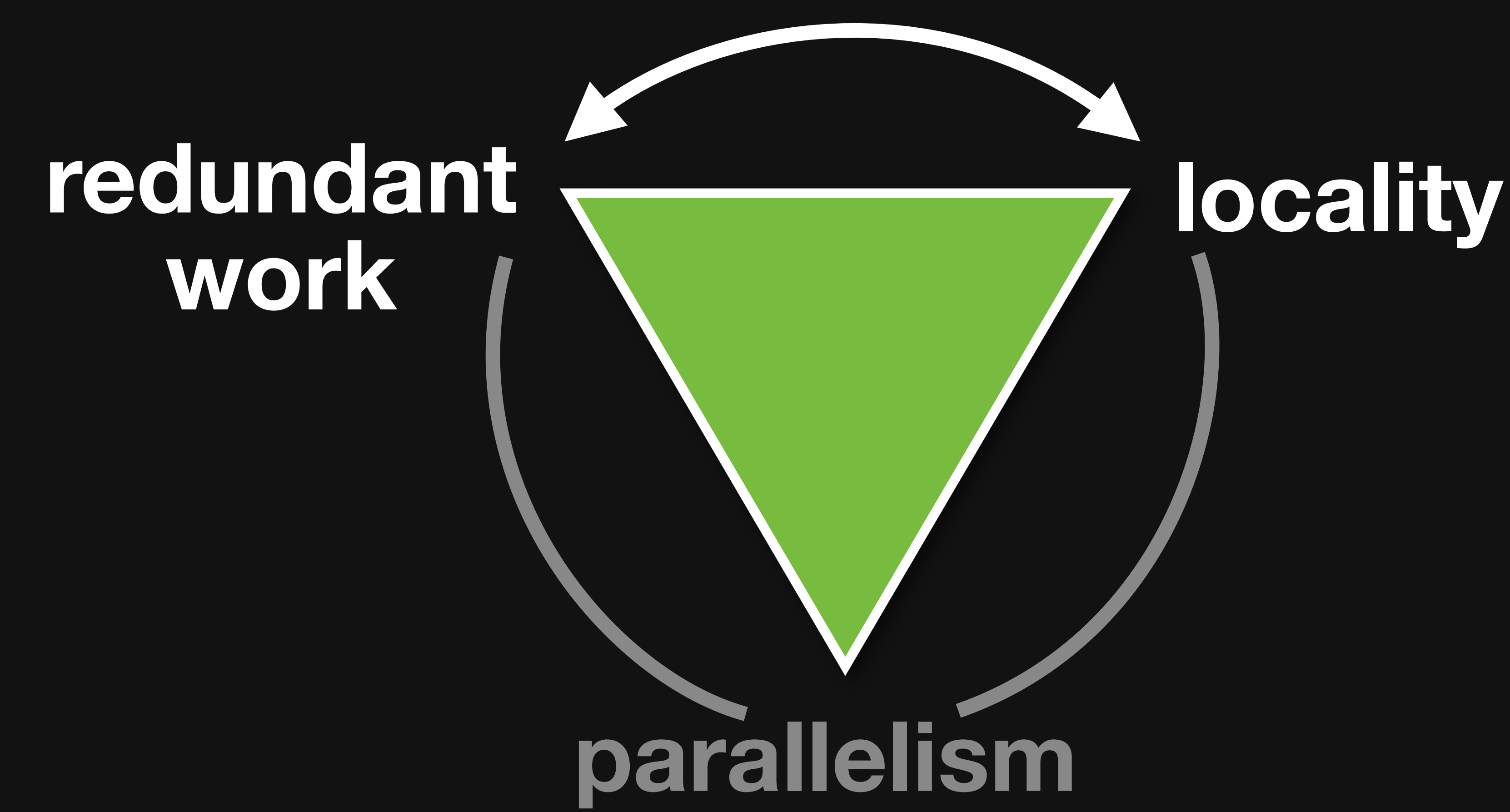
The schedule defines producer-consumer interleaving

Fine-grained fusion optimizes locality for point-wise operations

Breadth-first execution minimizes recomputation for large kernels

Tile-level fusion **trades off locality vs. recomputation for stencils**

Halide's *schedule* is designed to span these tradeoffs



Schedule primitives **compose** to create many organizations

Halide

0.9 ms/megapixel

```
Func box_filter_3x3(Func in) {
  Func blurx, blury;
  Var x, y, xi, yi;

  // The algorithm - no storage, order
  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;

  // The schedule - defines order, locality; implies storage
  blury.tile(x, y, xi, yi, 256, 32)
       .vectorize(xi, 8).parallel(y);
  blurx.compute_at(blury, x).store_at(blury, x).vectorize(x, 8);

  return blury;
}
```

C++

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);
}}}}}
```

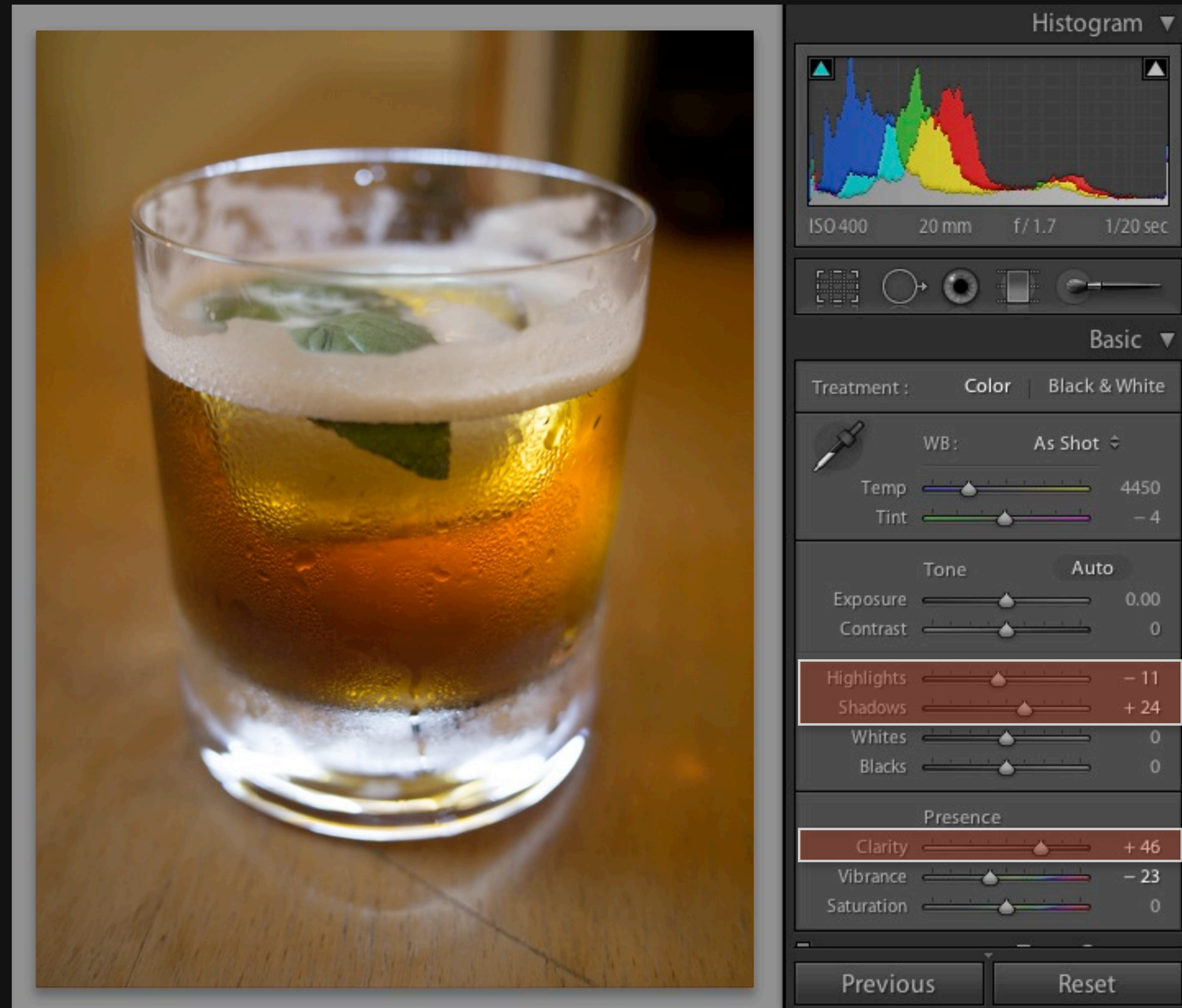
Local Laplacian Filters

prototype for Adobe Photoshop Camera Raw / Lightroom

Adobe: 1500 lines of expert-optimized C++
multi-threaded, SSE
3 months of work
10x faster *than original C++*

Halide: 60 lines
1 intern-day

2x faster on CPU,
7x faster on GPU



Local Laplacian Filters

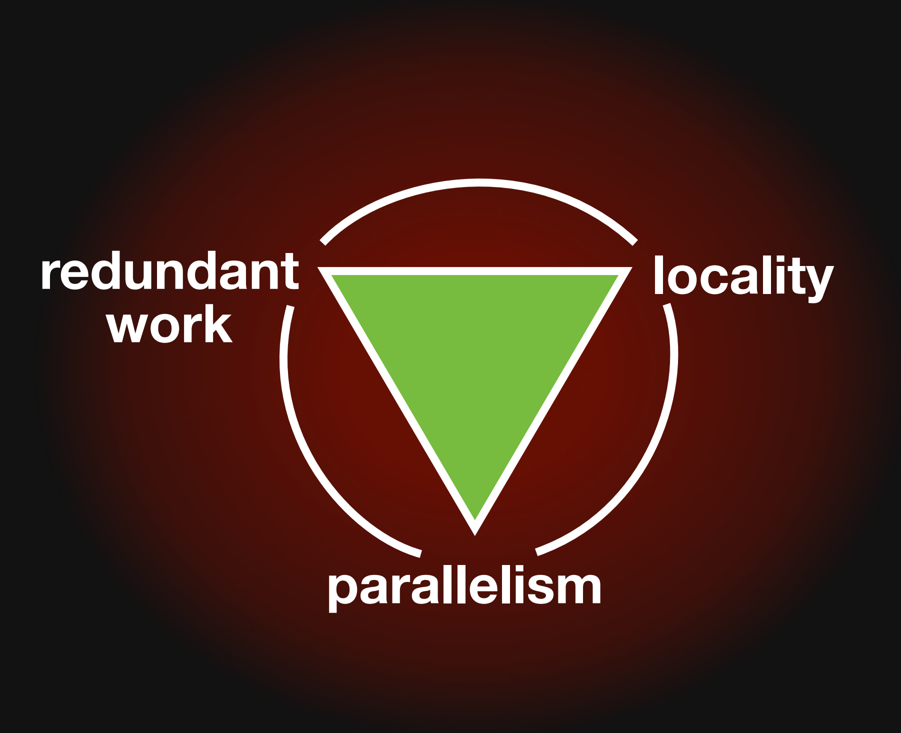
Adobe: 1500 lines of
expert-optimized C++
multi-threaded, SSE
3 months of work
10x faster *than original C++*

Halide: 60 lines
1 intern-day

2x faster on CPU,
7x faster on GPU

The Halide language & compiler

Decouples **algorithm** from **organization** through a **scheduling co-language** to navigate fundamental **tradeoffs**.



Simpler programs

Faster than hand-tuned code

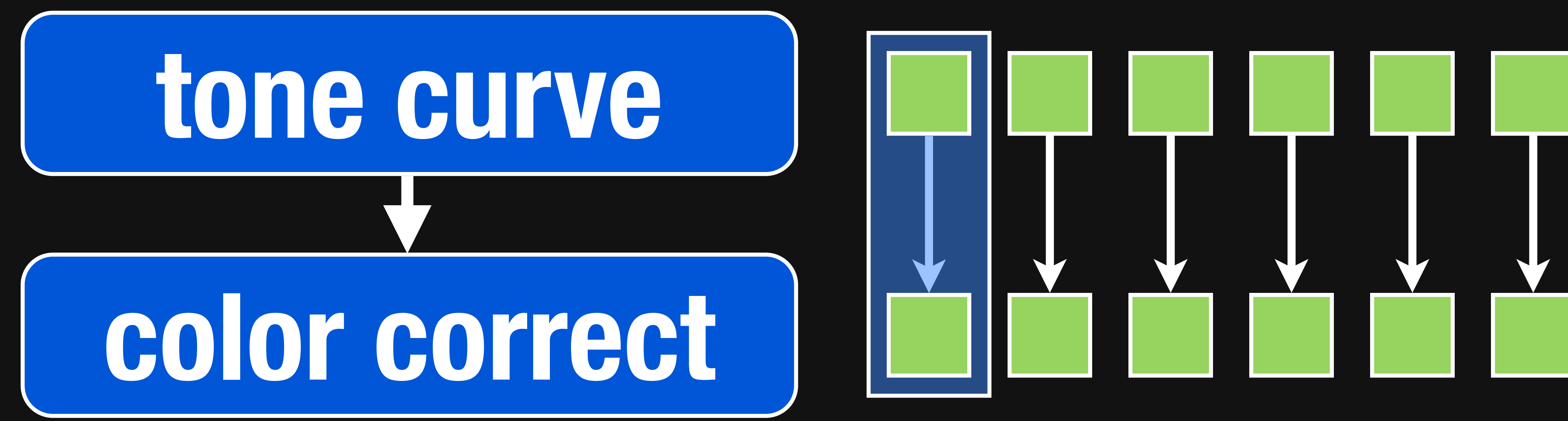
Scalable across architectures

Fredo XXX: remind two (three) big messages

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

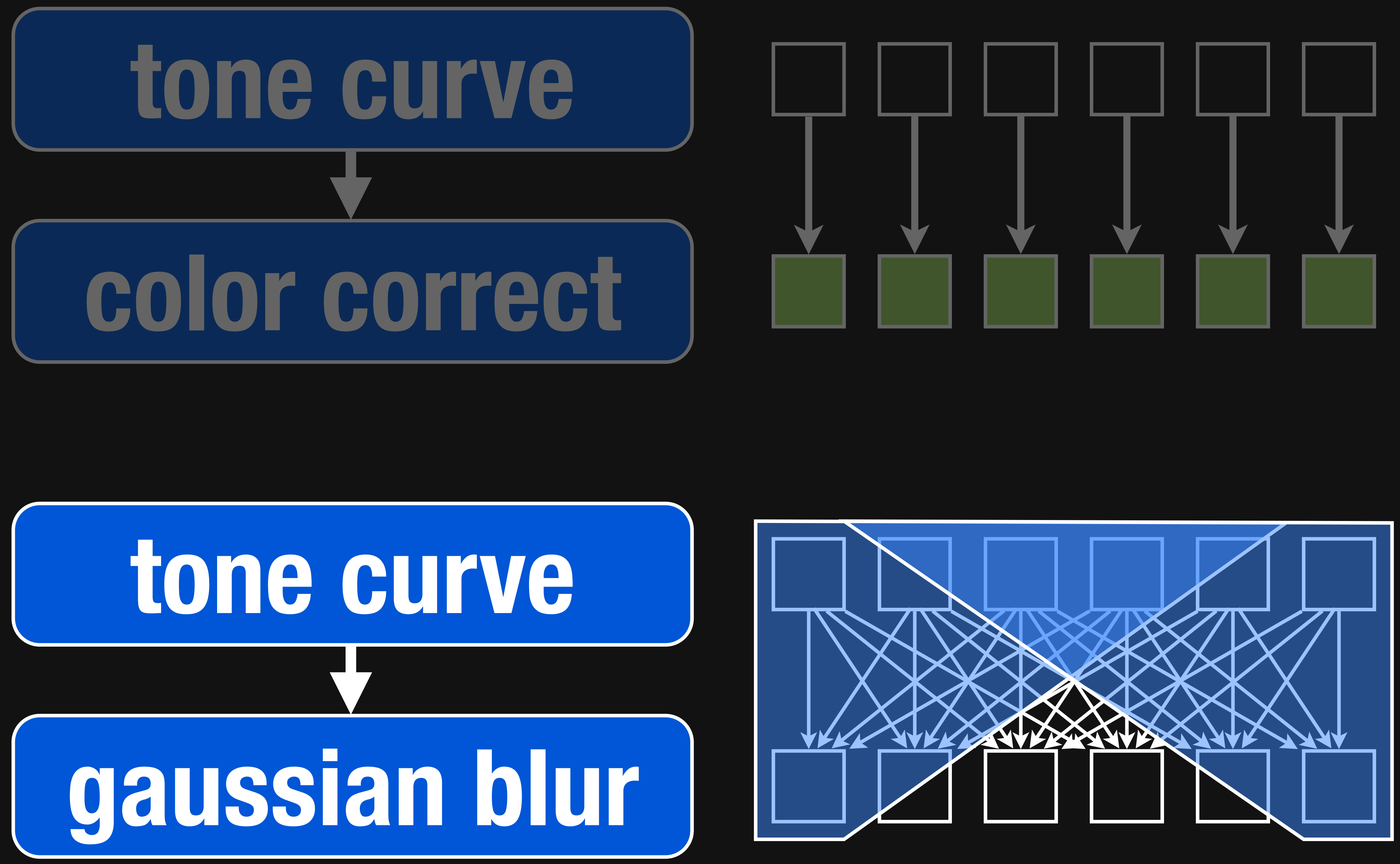
The schedule defines producer-consumer interleaving

Fine-grained fusion optimizes locality for point-wise operations



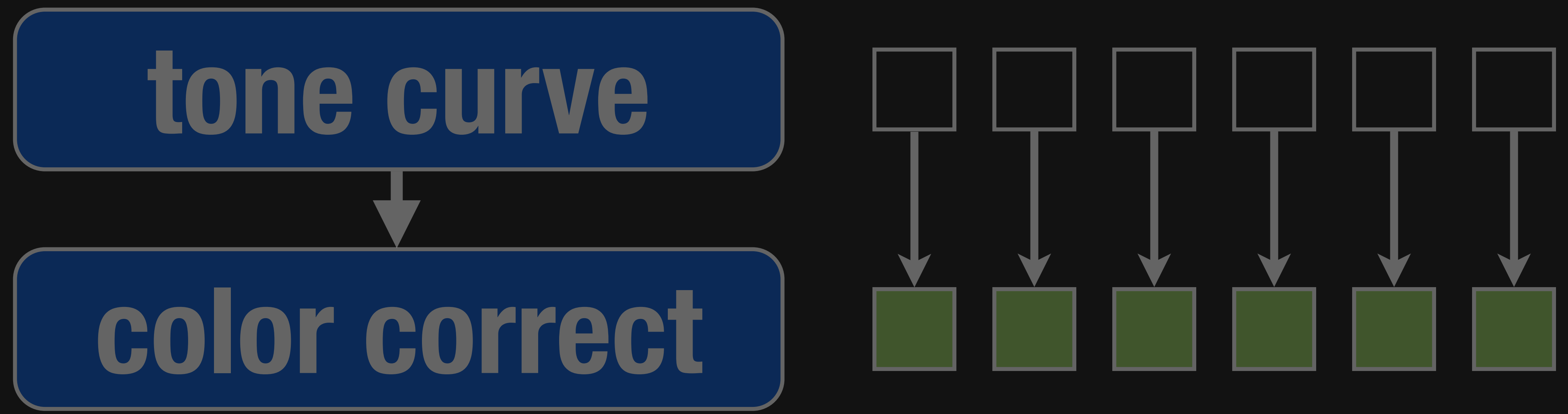
The schedule defines producer-consumer interleaving

Fine-grained fusion optimizes locality for point-wise operations

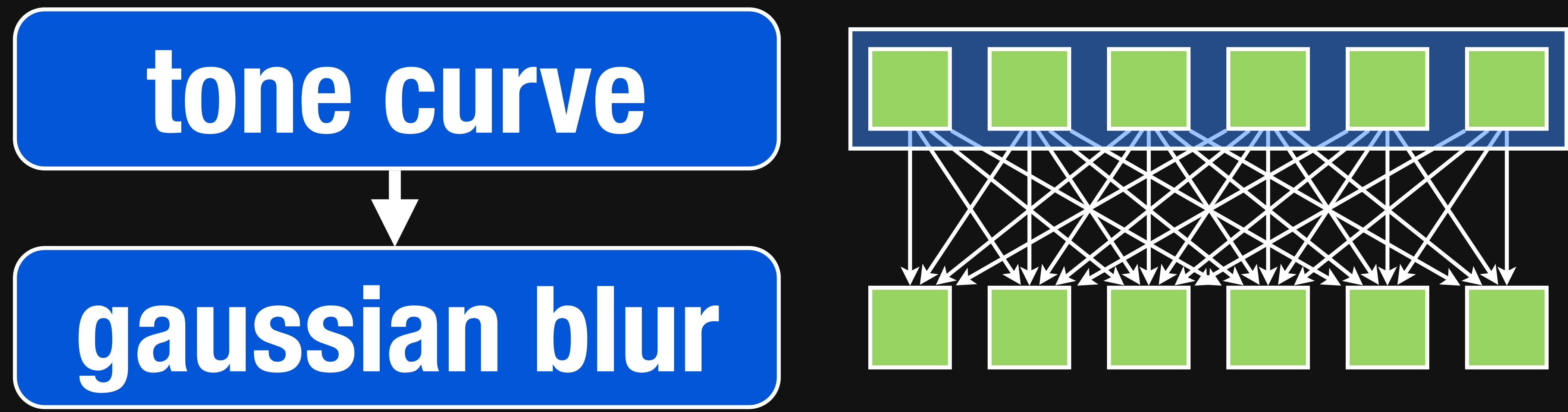


The schedule defines producer-consumer interleaving

Fine-grained fusion optimizes locality for point-wise operations



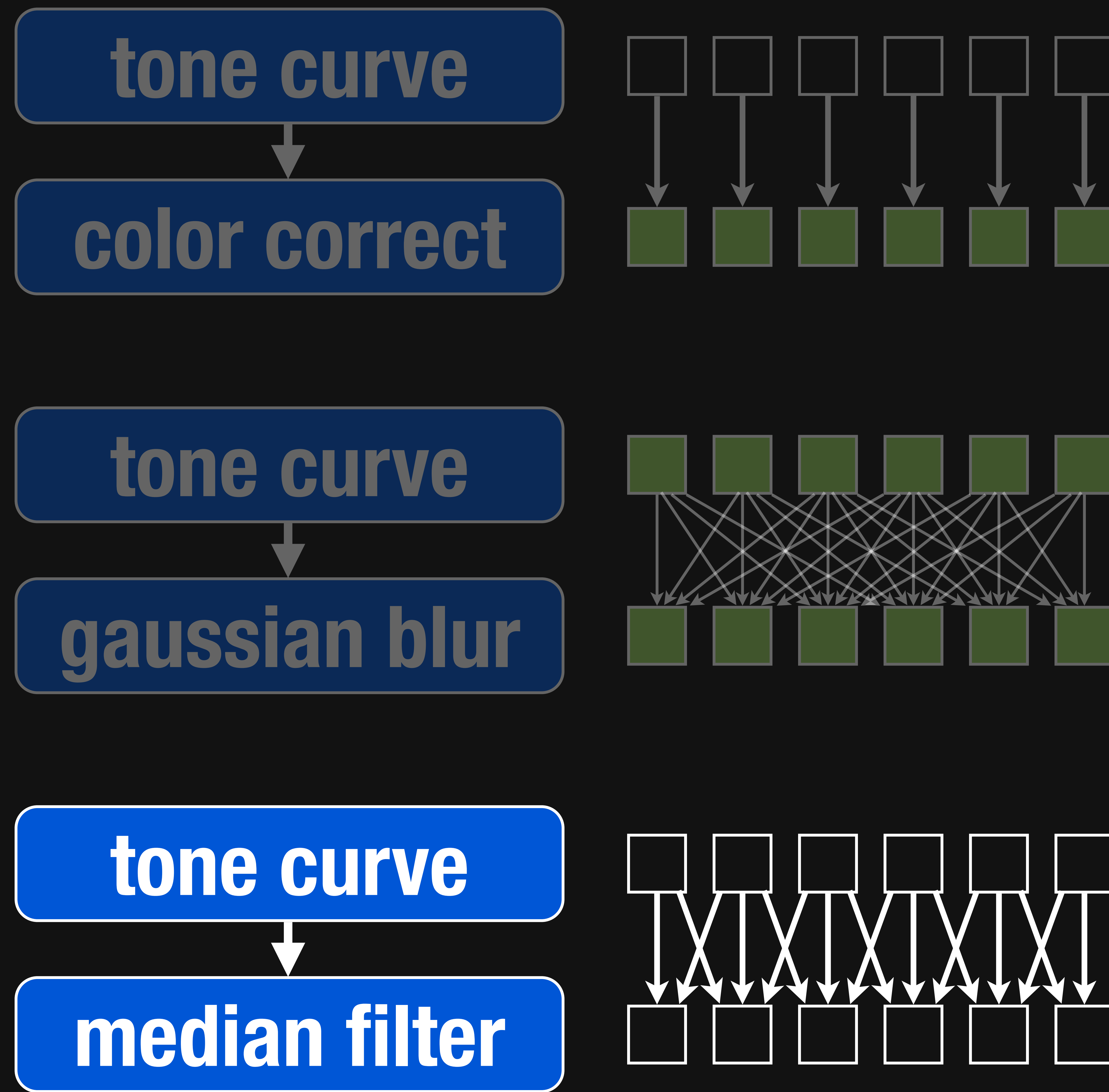
Breadth-first execution minimizes recomputation for large kernels



The schedule defines producer-consumer interleaving

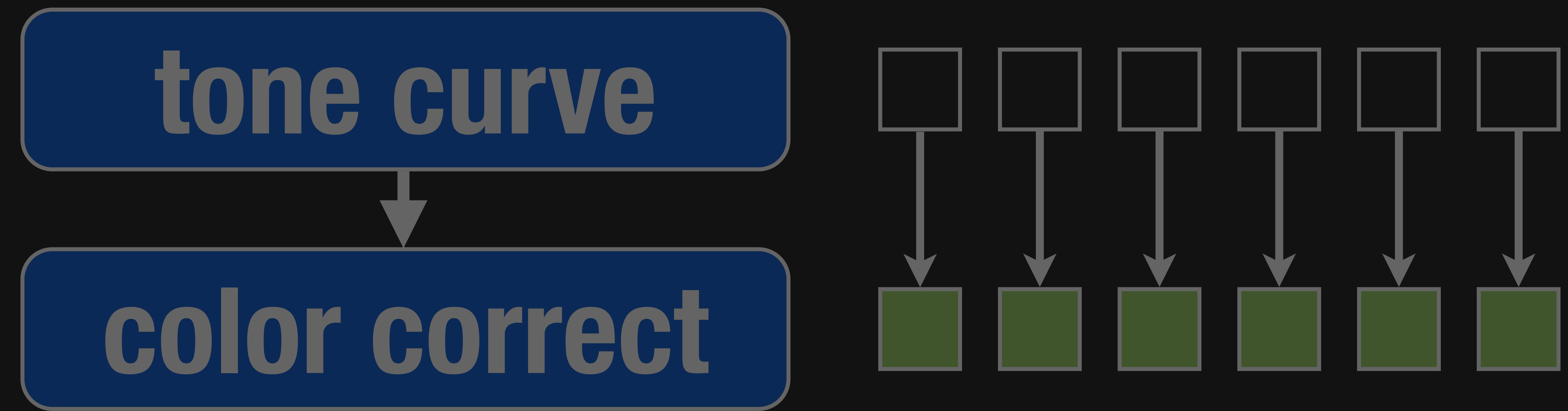
Fine-grained fusion optimizes locality for point-wise operations

Breadth-first execution minimizes recomputation for large kernels

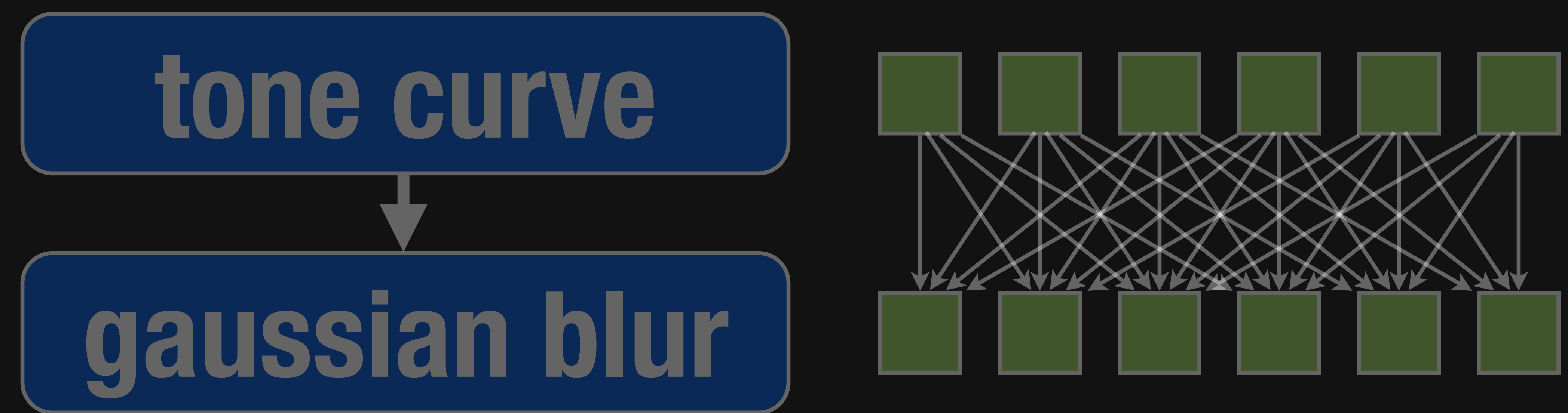


The schedule defines producer-consumer interleaving

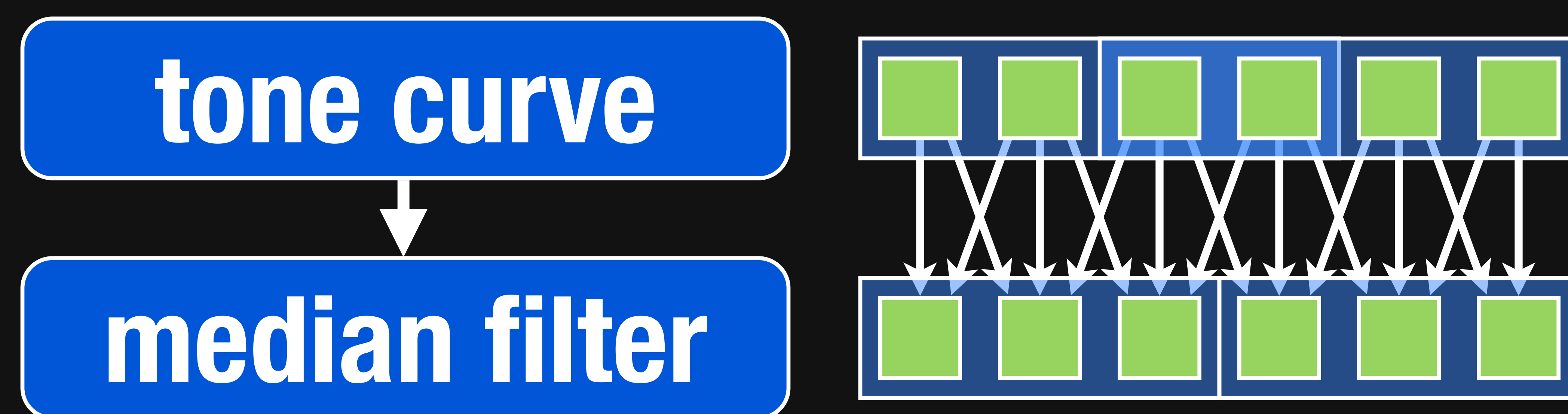
Fine-grained fusion optimizes locality for point-wise operations



Breadth-first execution minimizes recomputation for large kernels

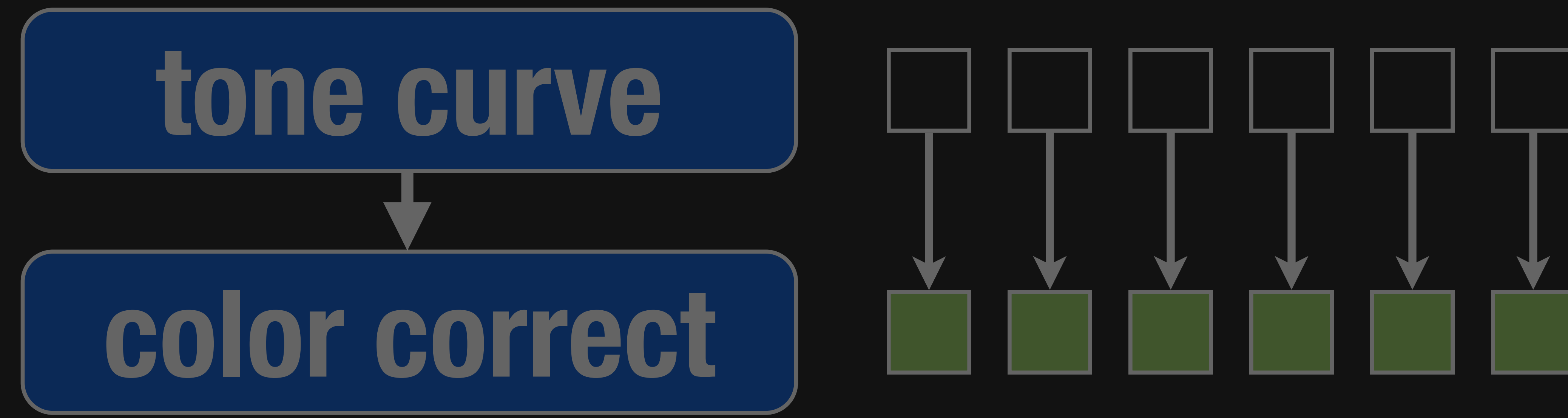


Tile-level fusion **trades off locality vs. recomputation for stencils**

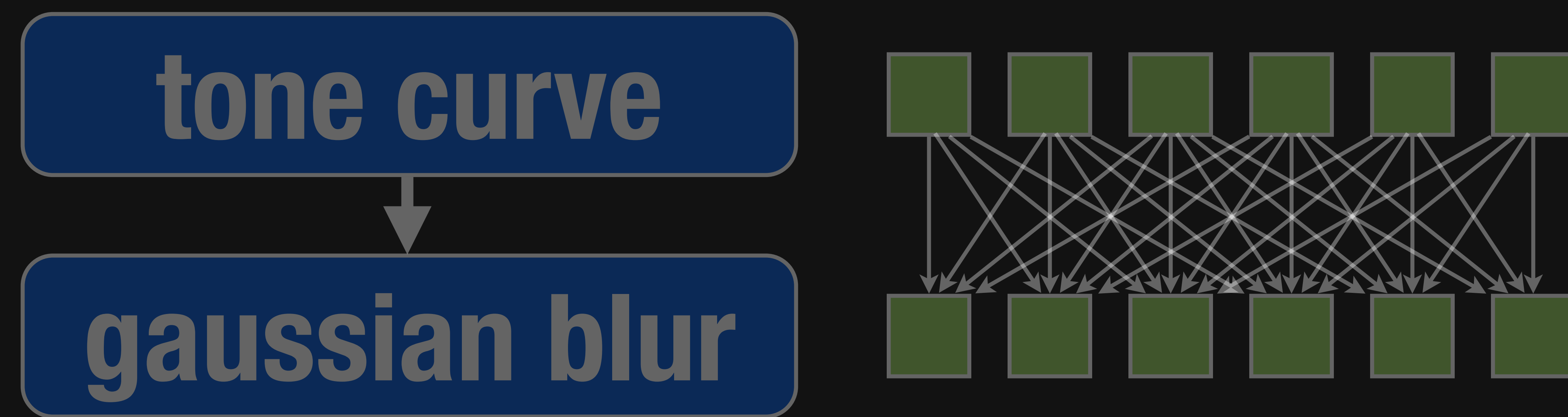


The schedule defines producer-consumer interleaving

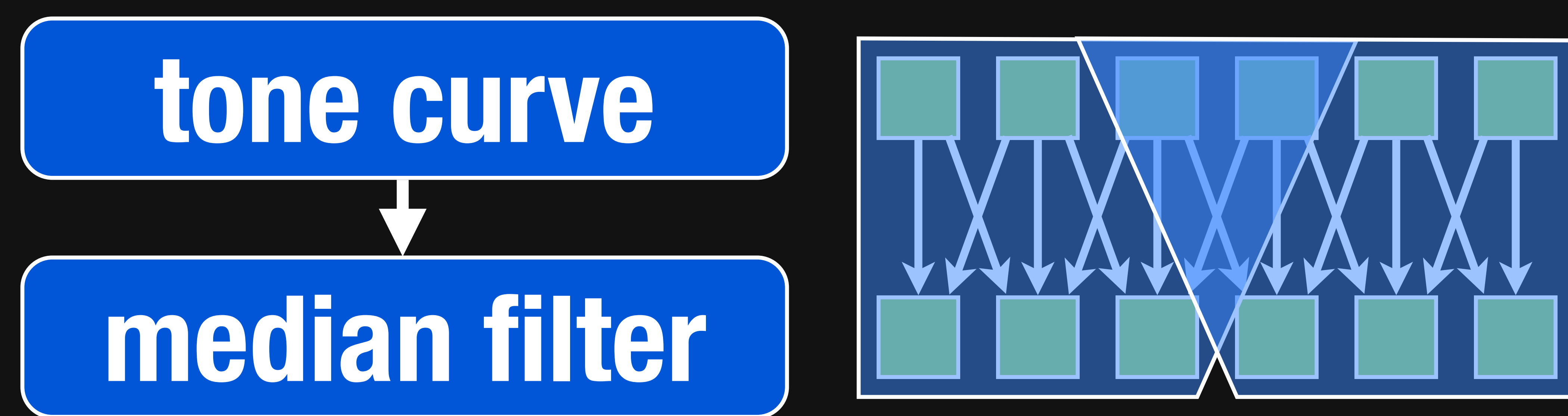
Fine-grained fusion optimizes locality for point-wise operations



Breadth-first execution minimizes recomputation for large kernels



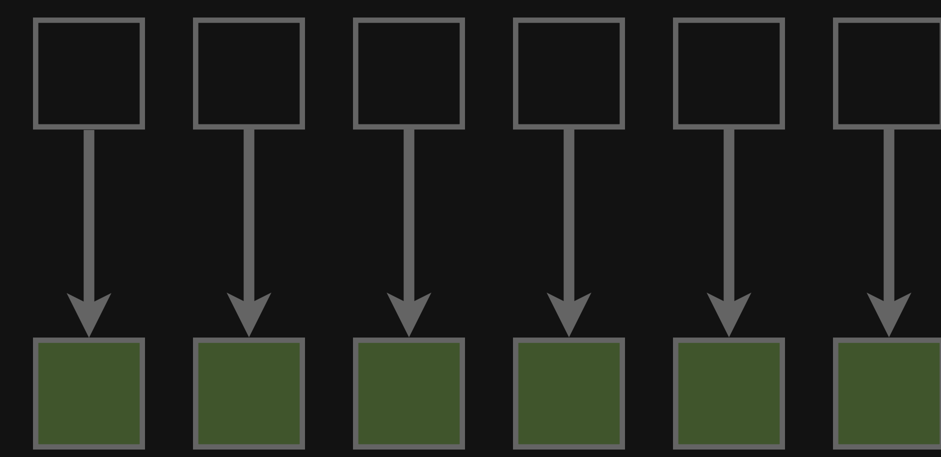
Tile-level fusion **trades off locality vs. recomputation for stencils**



The schedule defines producer-consumer interleaving

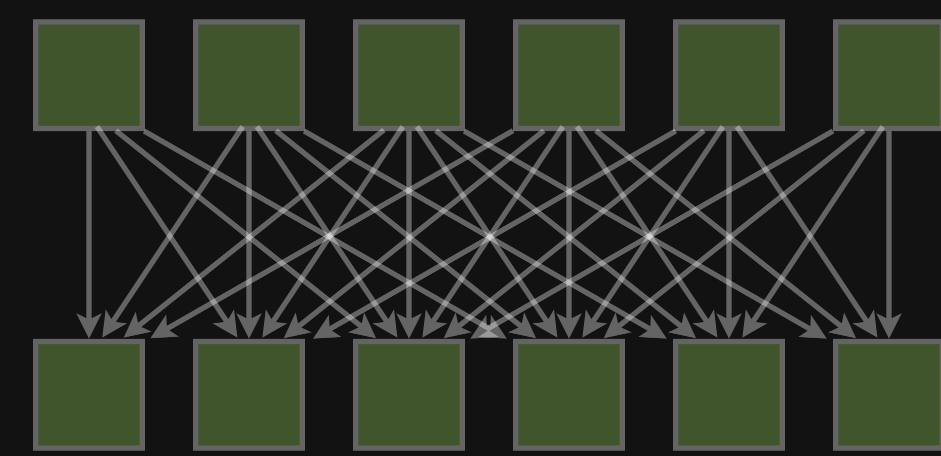
Fine-grained fusion optimizes locality for point-wise operations

tone curve → color correct



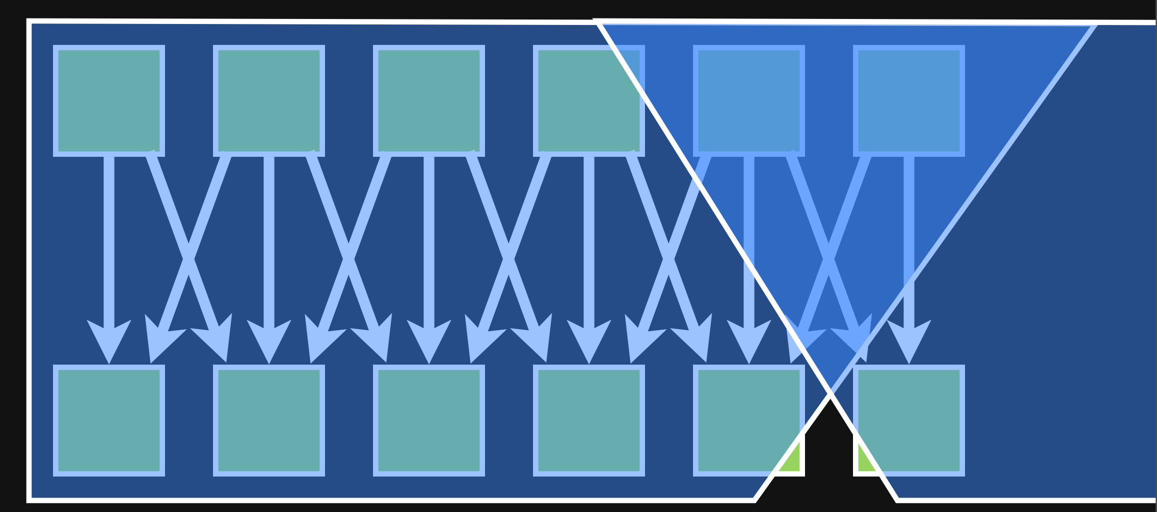
Breadth-first execution minimizes recomputation for large kernels

tone curve → gaussian blur



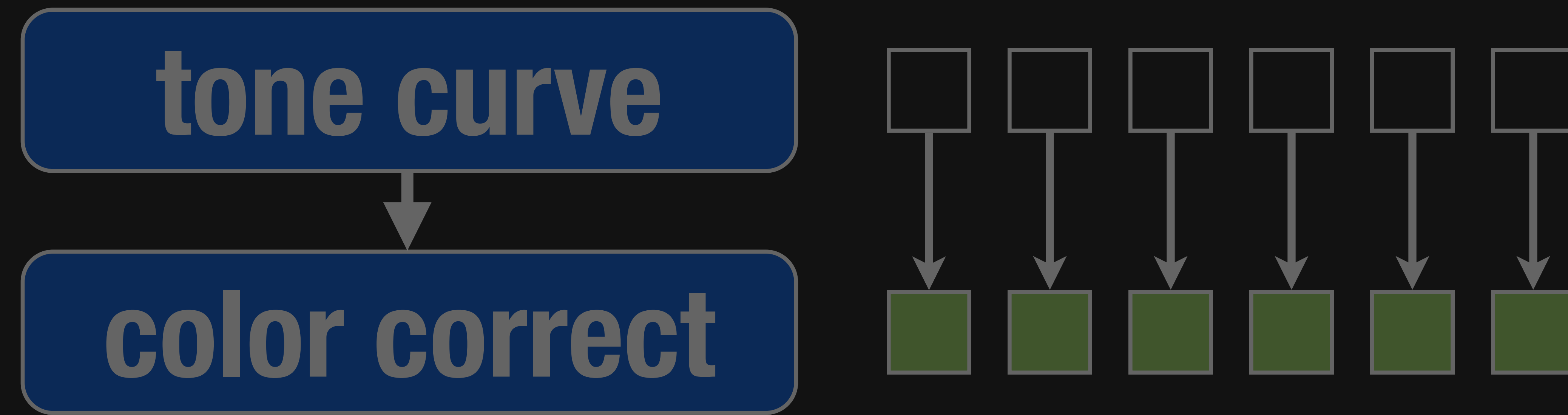
Tile-level fusion **trades off locality vs. recomputation for stencils**

tone curve → median filter

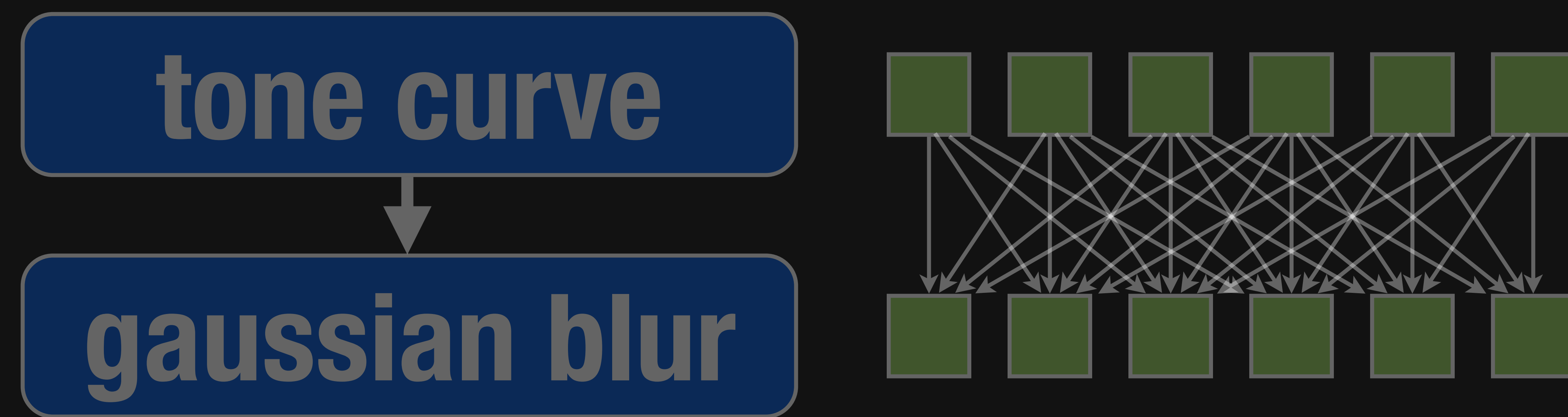


The schedule defines producer-consumer interleaving

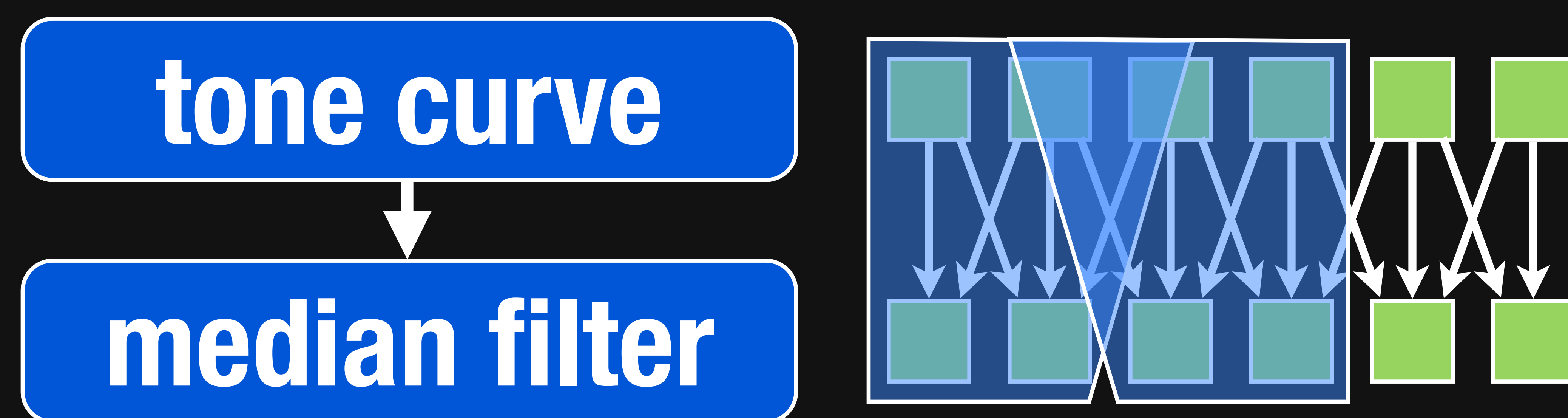
Fine-grained fusion optimizes locality for point-wise operations



Breadth-first execution minimizes recomputation for large kernels



Tile-level fusion **trades off locality vs. recomputation for stencils**



The schedule defines producer-consumer interleaving

Fine-grained fusion optimizes locality for point-wise operations

Breadth-first execution minimizes recomputation for large kernels

Tile-level fusion trades off locality vs. recomputation for stencils

Halide's *schedule* is designed to span these tradeoffs