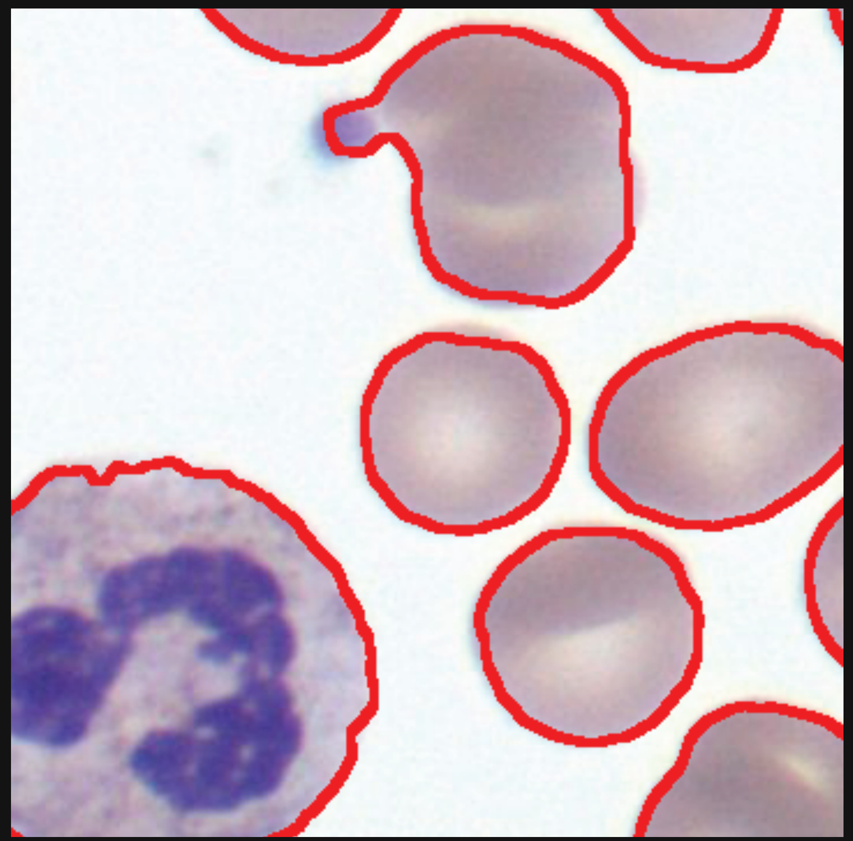
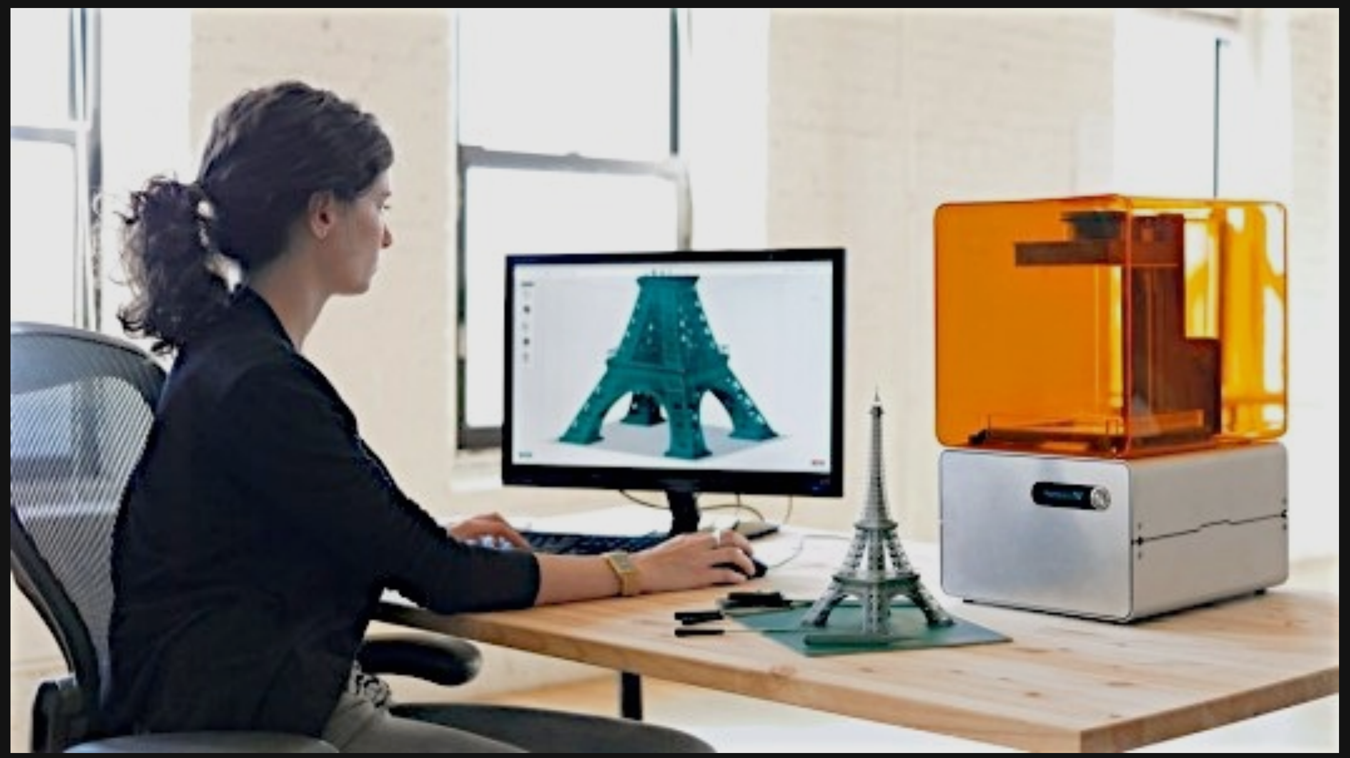
Lecture 24:

High-Performance Image Processing & Halide

Computer Graphics & Imaging
UC Berkeley CS184/284A





Visual computing and spatiotemporal data are **everywhere**

Visual computing demands orders of magnitude more computation

The biggest data is visual

YouTube: 400 hrs uploaded / min
[Brewer 2016]
1.5 Terapixels/sec

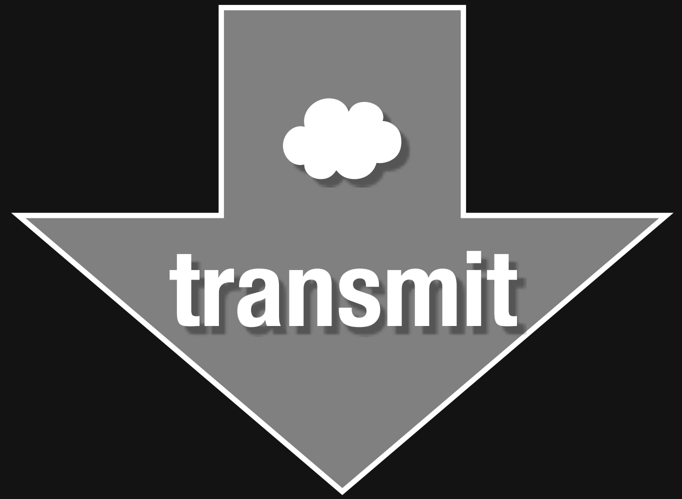
250 M surveillance cameras,
2.5 B cell phone cameras, …

Pervasive sensing: "the cloud" is not enough
data transfer ≫ capture

Sensor + Read out

5 Mpixels

~1 mJ/frame



LTE radio

50 Mbit/sec

1 W

~1 J/frame

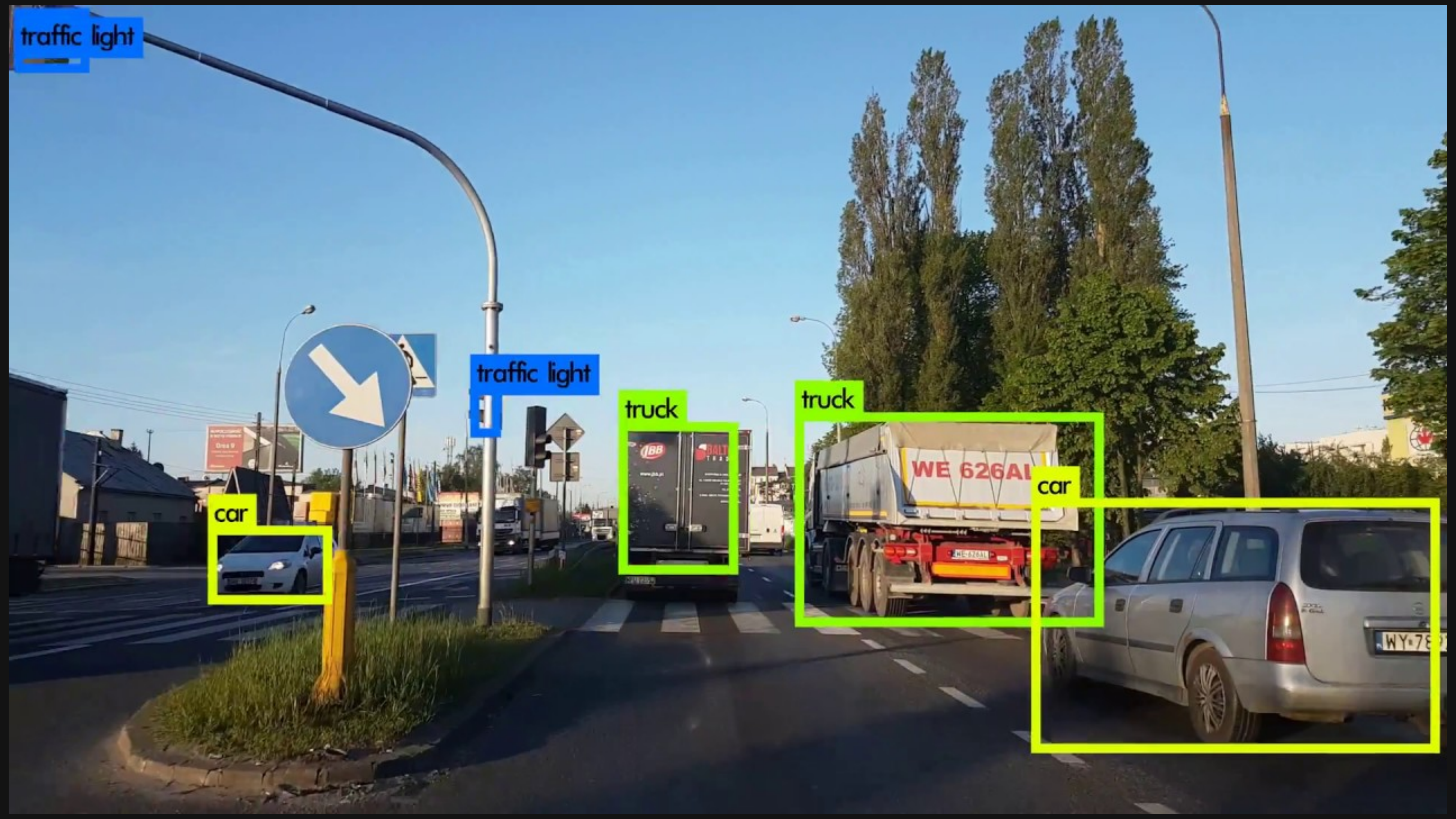
[Wu et al. 2012]

transmission power costs 1,000x capture

Visual data analysis is expensive

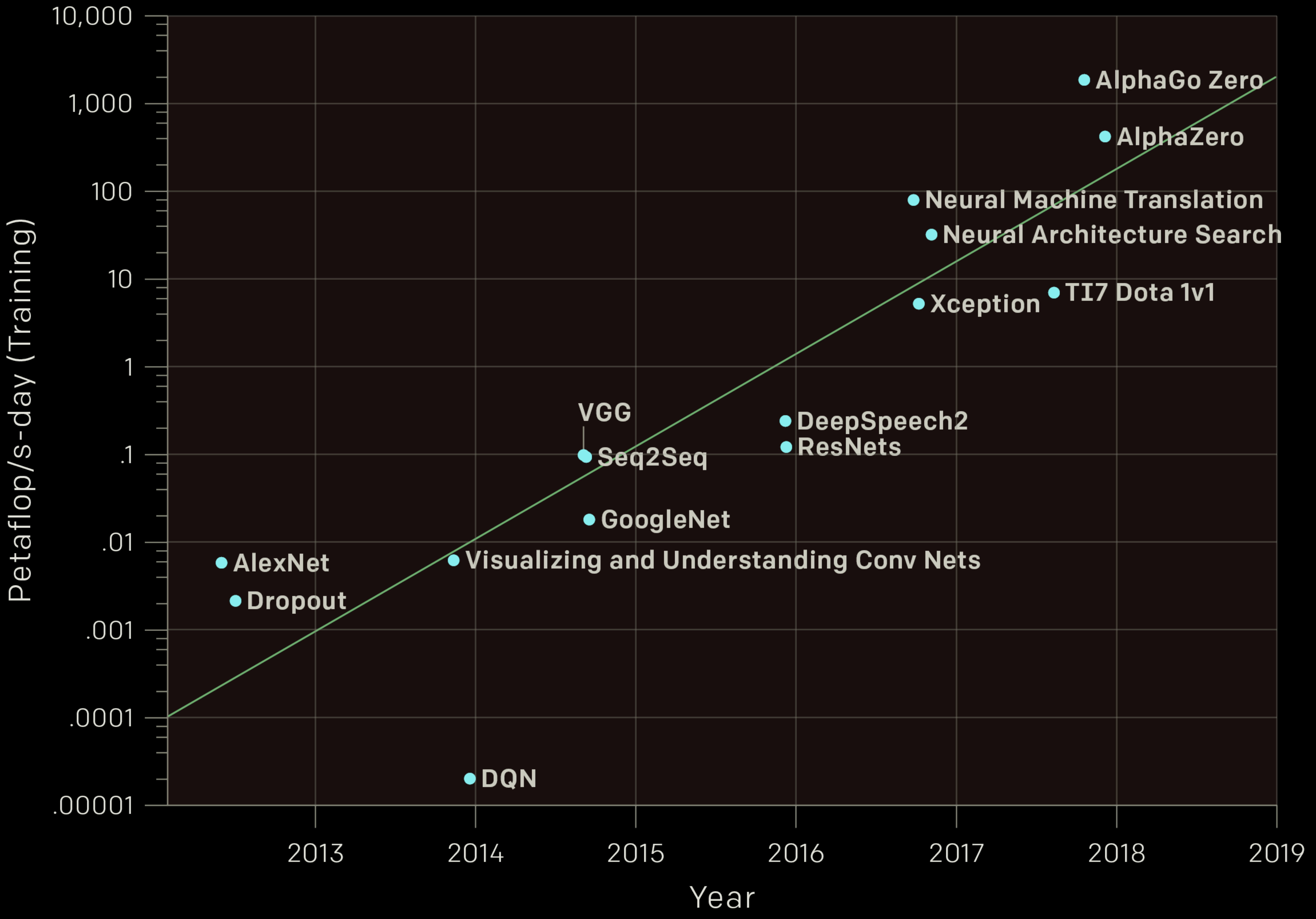
One object detection neural net:

250 Watt GPU → 0.25 megapixels at video rate



(yolo-v3 on Tesla V100)

AlexNet to AlphaGo Zero: A 300,000x Increase in Compute

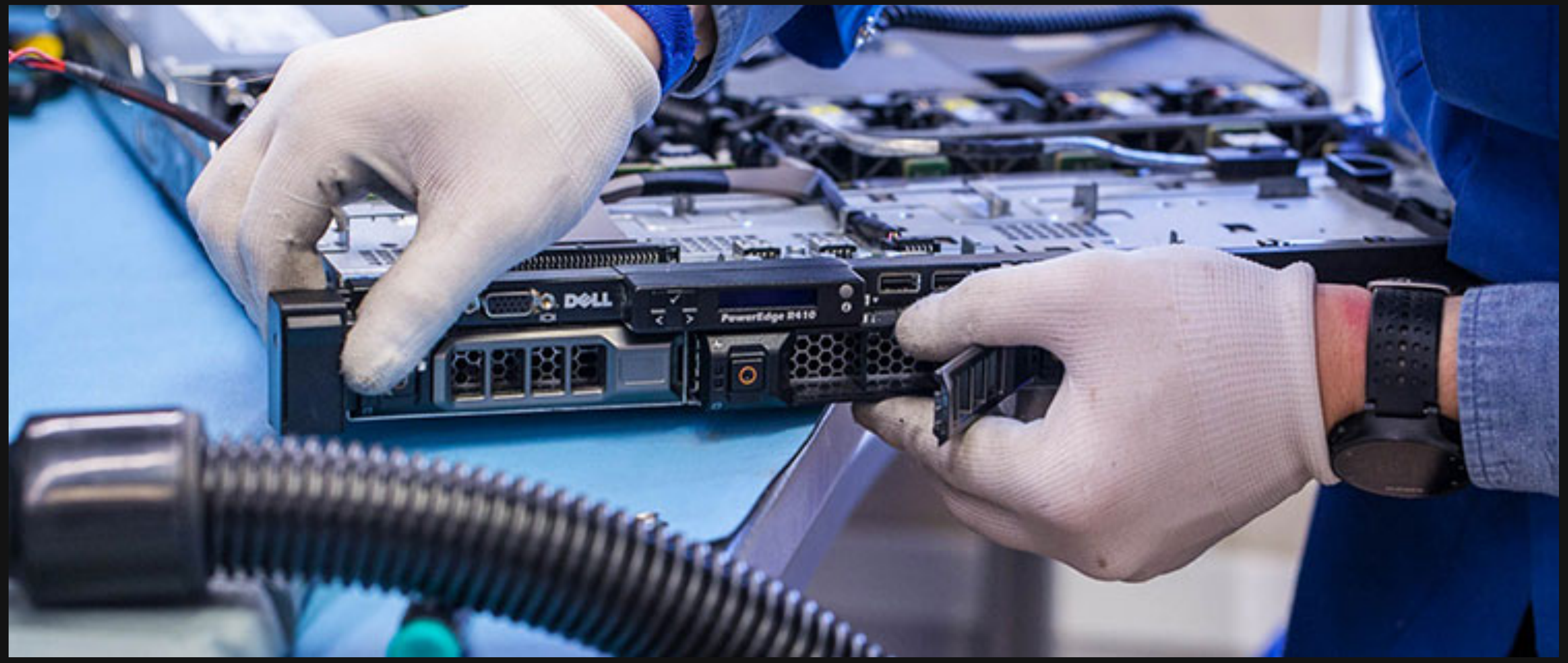


Source: OpenAI

Programmer productivity has exploded

C++

ORACLE



1990s

python

React

Firebase

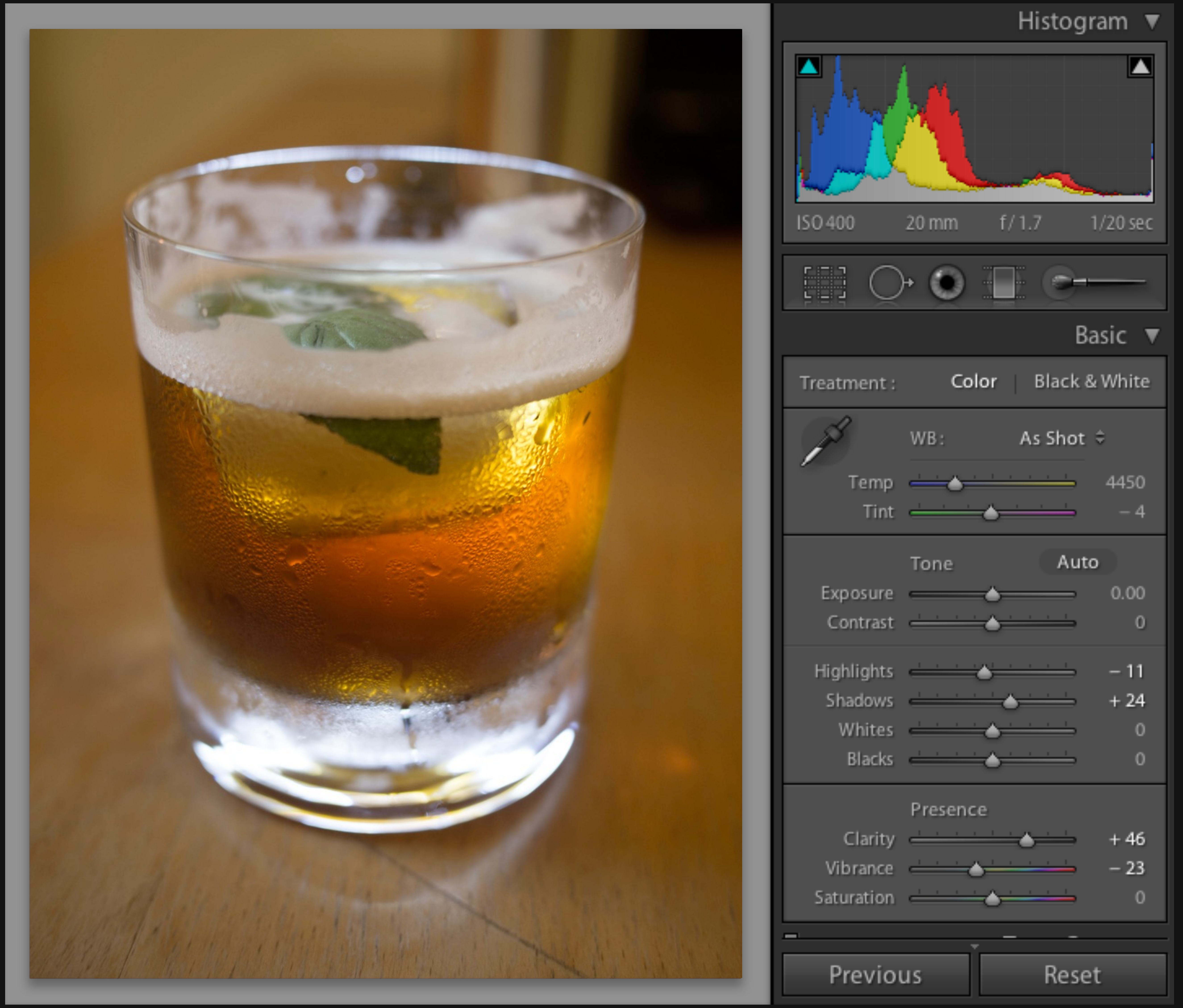
amazon web services

2010s

Building high-performance systems is harder than ever

Reference:
200 lines C++

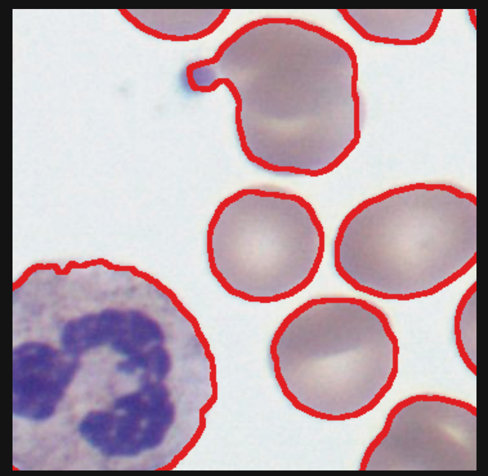
Adobe: 1500 lines
3 months of work
10x faster



My research:

Compilers, systems, architectures, and algorithms for high-performance graphics & visual computing.



Reorganize computations & data.

- Simpler programs
- Order of magnitude faster
- Scalable on future architectures

applications

algorithms

data structures

languages

compilers

hardware

How can we increase performance and efficiency?

Parallelism

"Moore's law" scaling requires exponentially more parallelism.

Locality

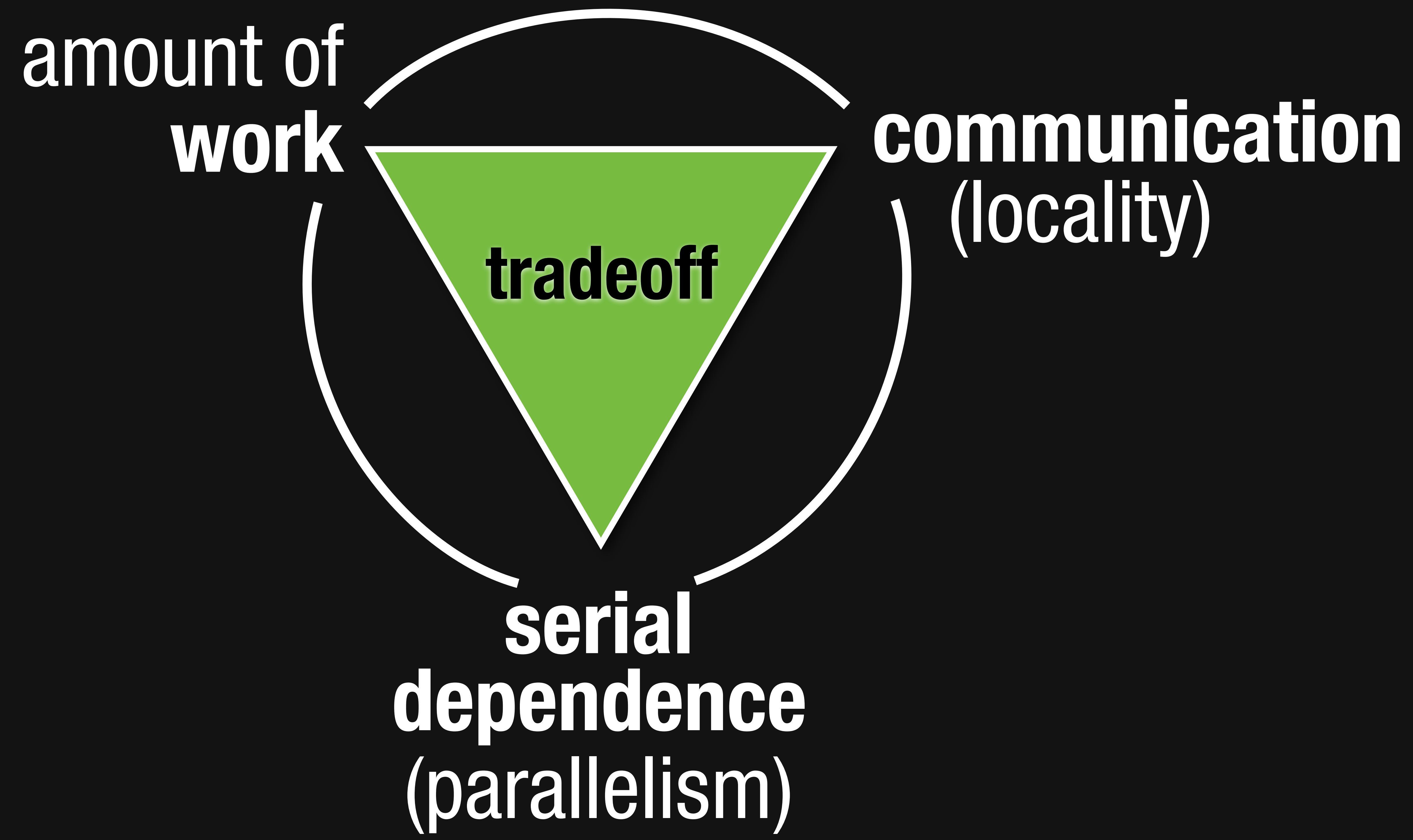
Data should move as little as possible.

Communication dominates computation
in both energy and time

Operation (32-bit operands)	**Energy/Op** (28 nm)	***Cost*** *(vs. ALU)*
ALU op	1 pJ	-
Load from SRAM	5 pJ	*5x*
Move 10mm on-chip	32 pJ	*32x*
Send off-chip	500 pJ	*500x*
Send to DRAM	1 nJ	*1,000x*
Send over LTE	> 50 μJ	*50,000,000x*

data from John Brunhaver, Bill Dally, Mark Horowitz

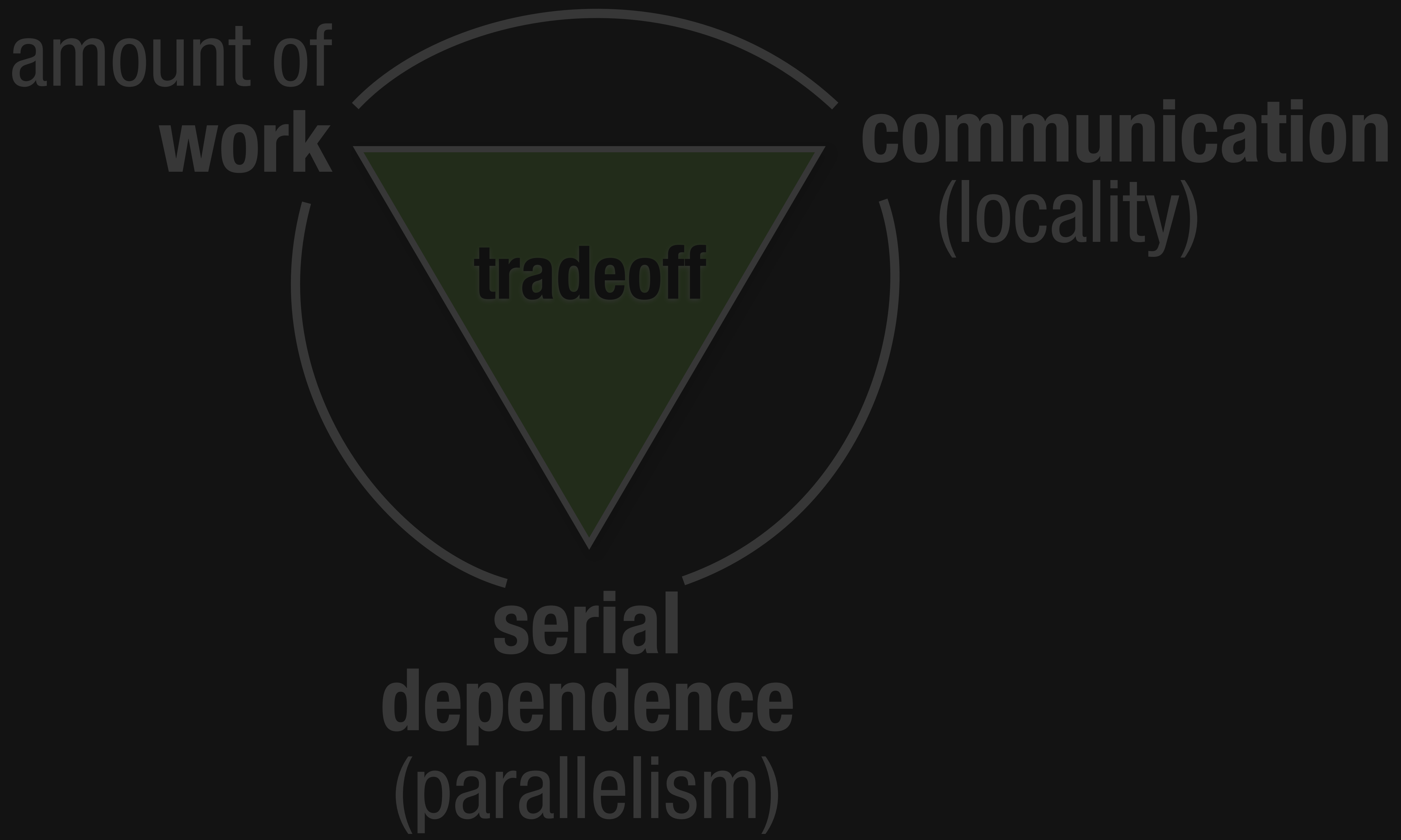
Message #1: Performance requires complex tradeoffs



Where does performance come from?

Program

Hardware



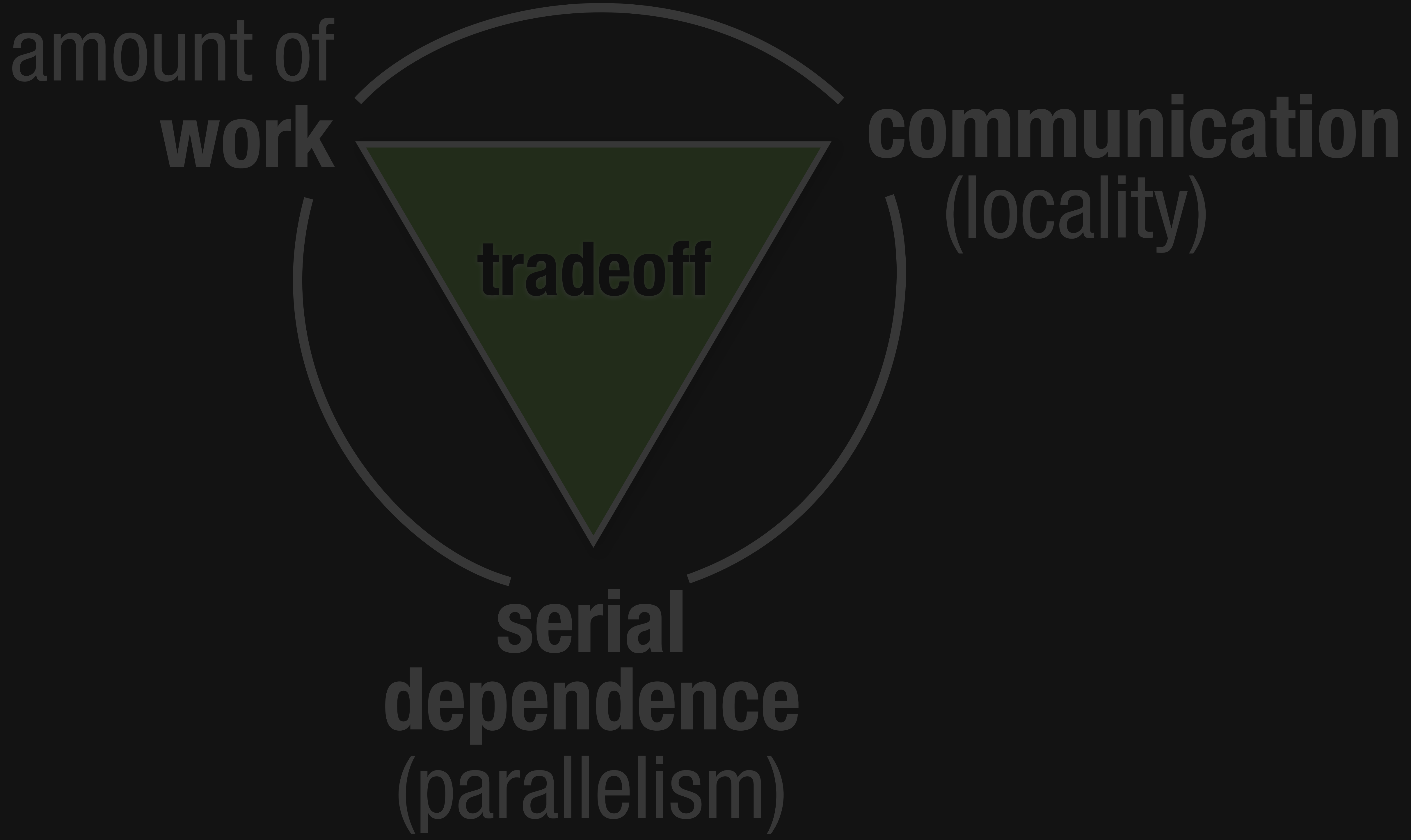
Message #2: organization of computation
is a first-class issue

Program:

Algorithm

Organization of computation

Hardware

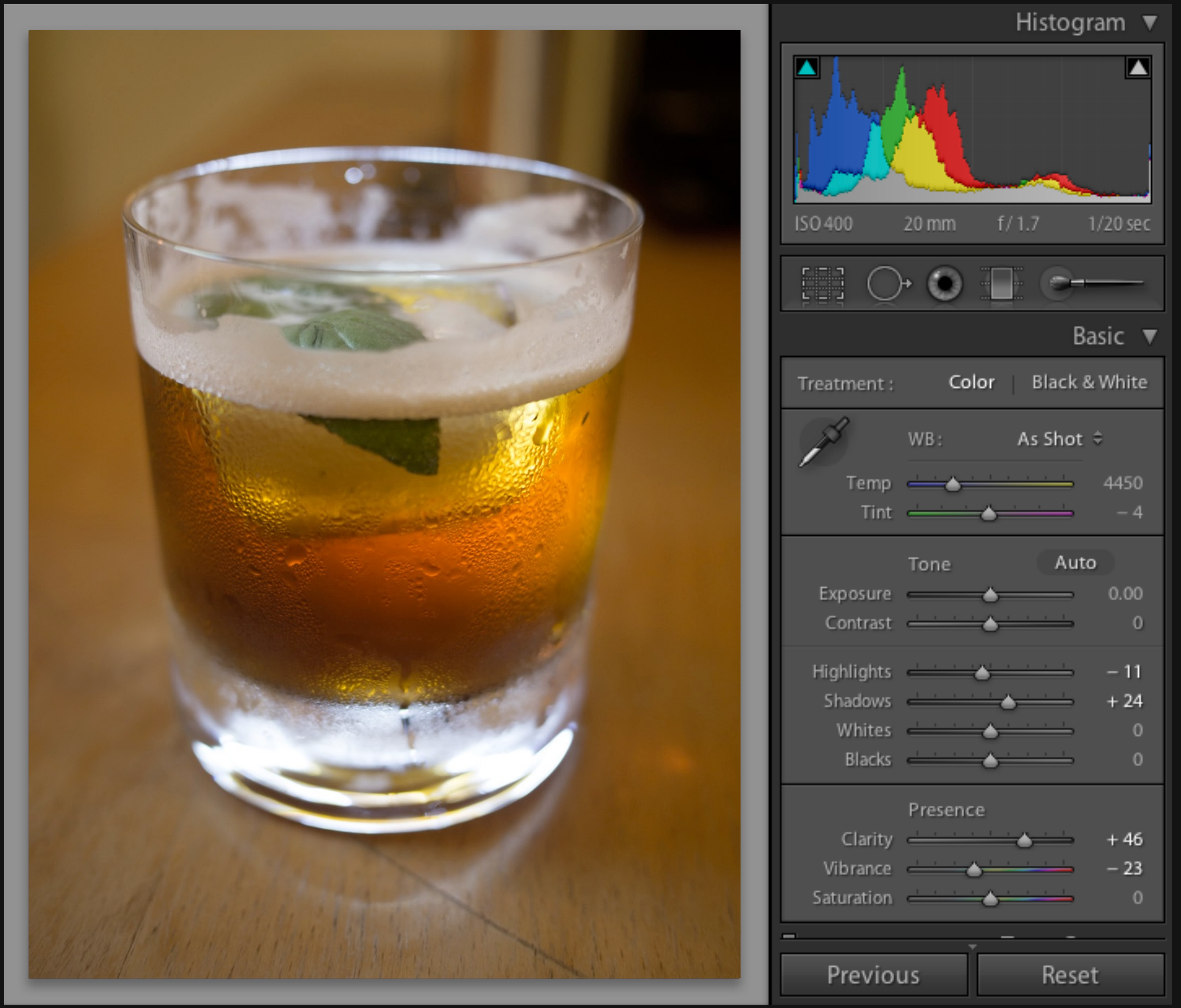


Reorganizing computation is painful

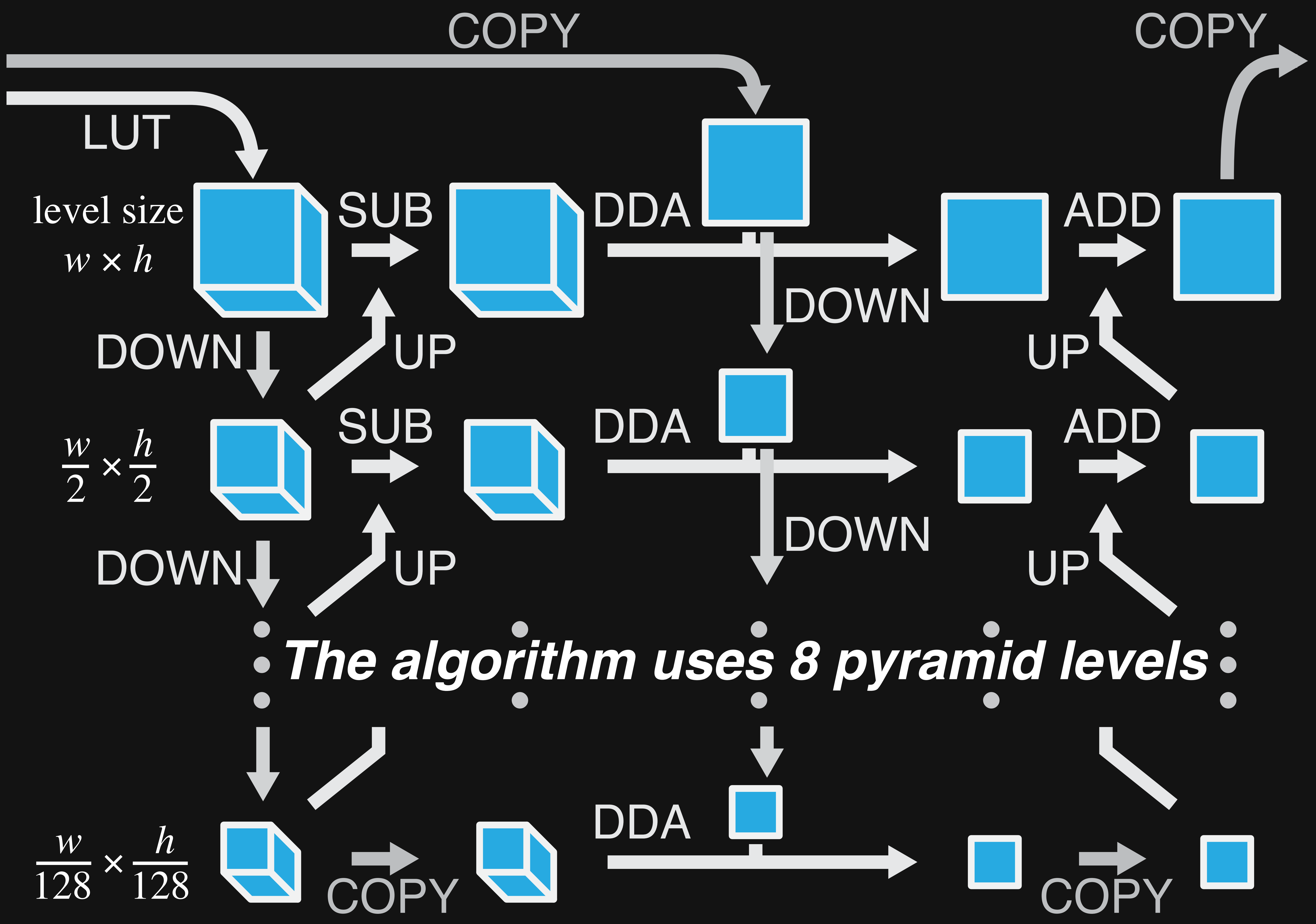
Reference:
300 lines C++

Adobe: 1500 lines
3 months of work
10x faster (vs. reference)

Same algorithm,
Different organization



Global reorganization breaks modularity



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

(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

a new language & compiler for image processing

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

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

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

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

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

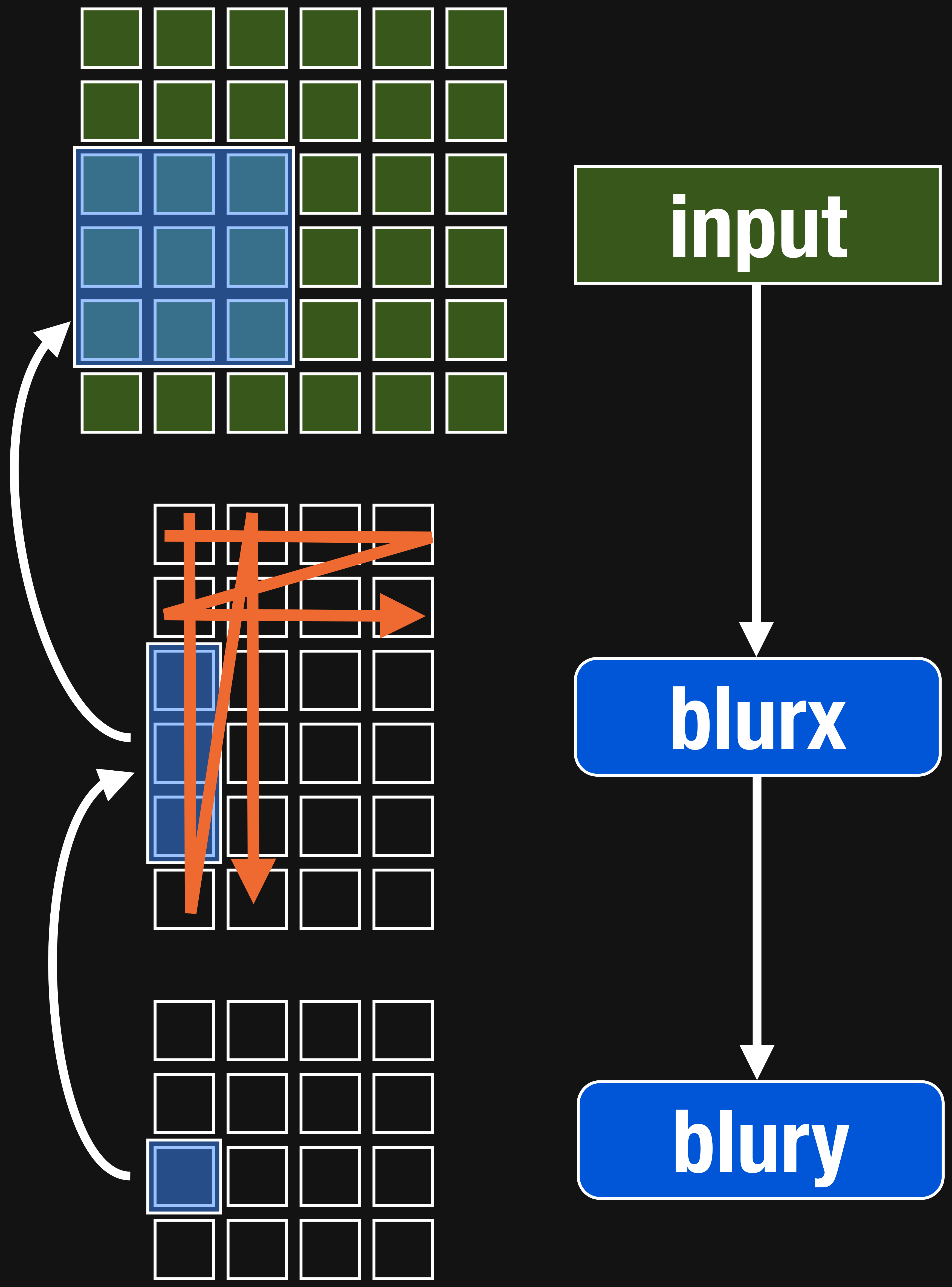
The schedule defines intra-stage order, inter-stage interleaving

For each stage:

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

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

This is a **co-language** for scheduling choices.



The **Schedule** defines a **loop nest** to compute the pipeline

```
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;
```

```
blury.tile(x, y, xo, yo, xi, yi, 32, 32);
```

```
// for each tile
for blury.yo:
  for blury.xo:
  // for pixel in tile
  for blury.yi:
    for blury.xi:
      compute blury
```

The **Schedule** defines a **loop nest** to compute the pipeline

```
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;
```

```
blury.tile(x, y, xo, yo, xi, yi, 256, 32);
blurx.compute_at(blury, xo);
```

compute *here*

```
// for each tile
for blury.yo:
  for blury.xo:
  // for pixel in tile
  for blury.yi:
    for blury.xi:
      compute blury
```

The **Schedule** defines a **loop nest** to compute the pipeline

```
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;
```

```
blury.tile(x, y, xo, yo, xi, yi, 256, 32);
blurx.compute_at(blury, xo);

// for each tile
for blury.yo:
  for blury.xo:
  // for pixel in required tile
  for blurx.y:
    for blurx.x:
      compute blurx
  // for pixel in tile
  for blury.yi:
    for blury.xi:
      compute blury
```

compute *here*

The **Schedule** defines a **loop nest** to compute the pipeline

```
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;
```

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

```
// for each tile
parallel for blury.yo:
  for blury.xo:
  // for pixel in required tile
  for blurx.y:
    vec for blurx.x:
      compute blurx<8>
  // for pixel in tile
  for blury.yi:
    for blury.xi:
      compute blury
```

Prior work*

*a tiny sample. Thousands have come before us.

Streaming languages

Ptolemy [Buck et al. 1993]
StreamIt [Thies et al. 2002]
Brook [Buck et al. 2004]

Loop optimization

Systolic arrays [Gross & Lam 1984]
Polyhedral model [Ancourt & Irigoin 1991, Amarasinghe & Lam 1993]

Parallel work scheduling

Cilk [Blumhofe et al. 1995]
NESL [Blelloch et al. 1993]

Region-based languages

ZPL [Chamberlain et al. 1998]
Chapel [Callahan et al. 2004]

Stencil optimization & DSLs

[Frigo & Strumpen 2005]
[Krishnamoorthy et al. 2007]
[Kamil et al. 2010]

Mapping-based languages & DSLs

SPL/SPIRAL [Püschel et al. 2005]
Sequoia [Fatahalian et al. 2006]

Shading languages

RSL [Hanrahan & Lawson 1990]
Cg, HLSL [Mark et al. 2003; Blythe 2006]

Image processing systems

[Shantzis 1994], [Levoy 1994]
PixelBender, CoreImage

Domain scope
of the programming model

All computation is over **regular grids** (up to 4D).

not Turing complete

Only **feed-forward pipelines**

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

Recursion must have **bounded depth.**

Long, heterogeneous pipelines.

Complex graphs, deeper than traditional stencil computations.

Roadmap

1. **Fundamental transformations**
   for stencil pipelines

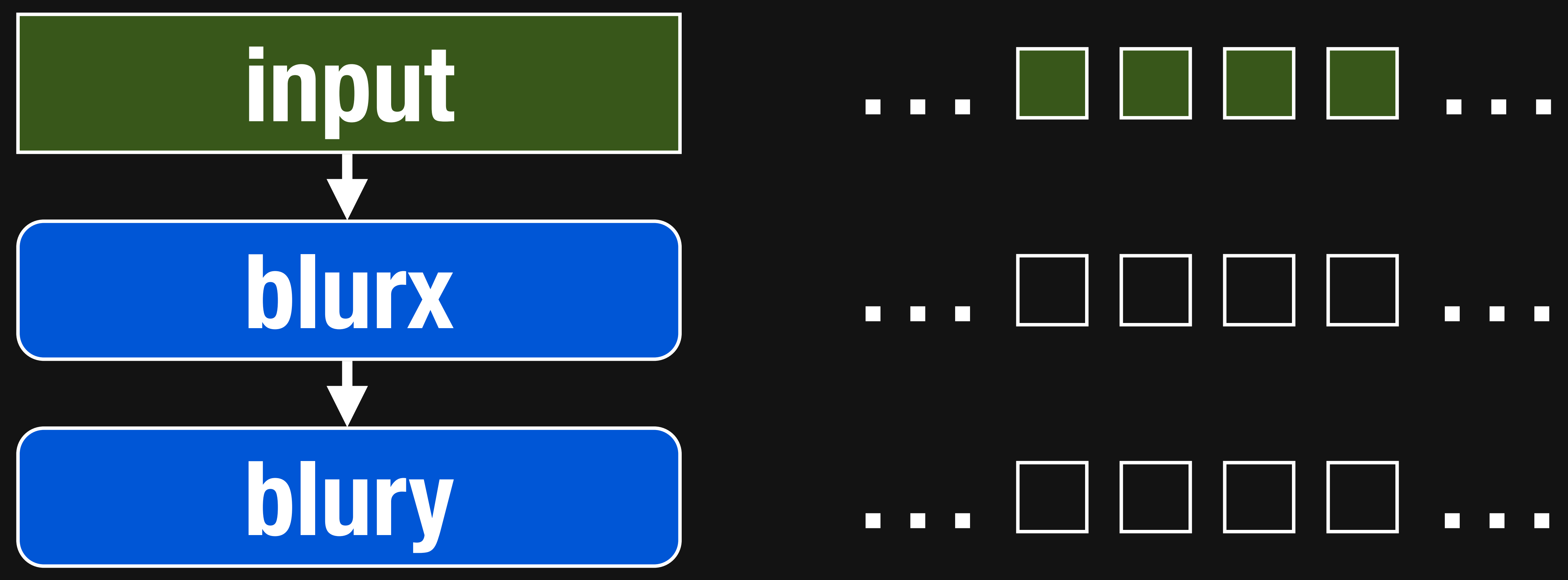
2. **Halide's unified model of scheduling**

3. **Results** on real image processing pipelines

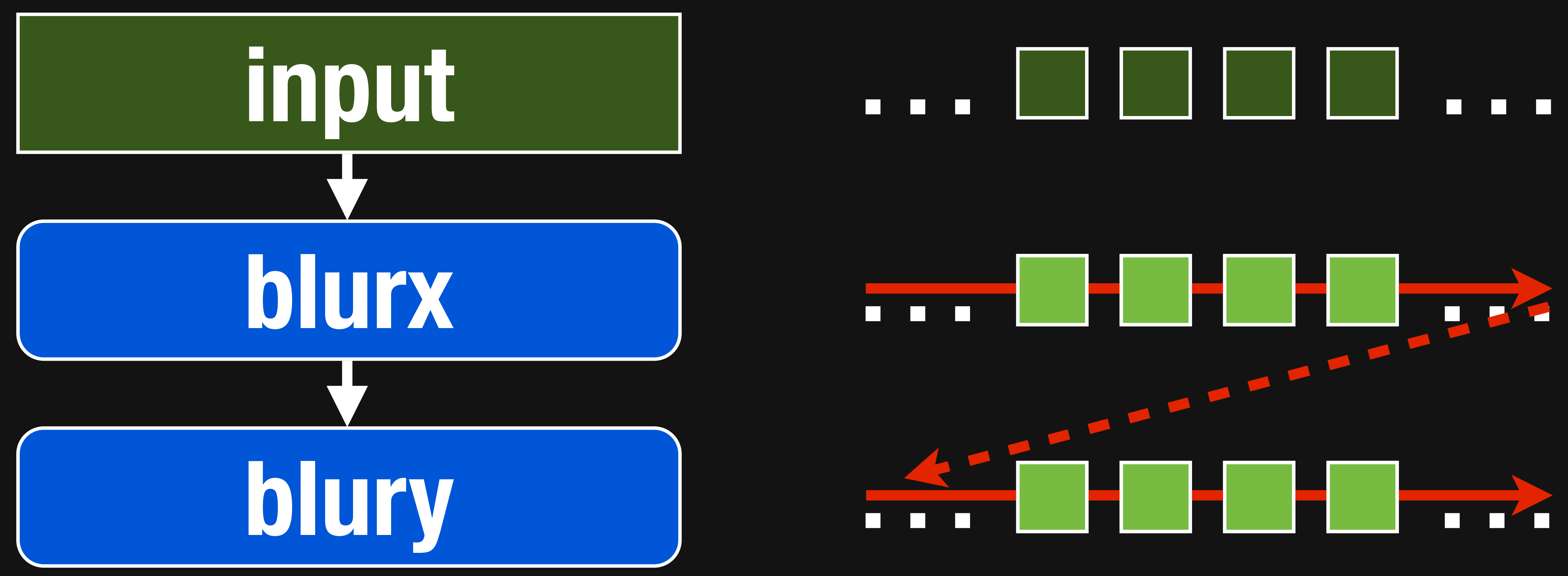
Roadmap

1. **Fundamental transformations** for stencil pipelines
2. **Halide's unified model of scheduling**
3. **Results** on real image processing pipelines

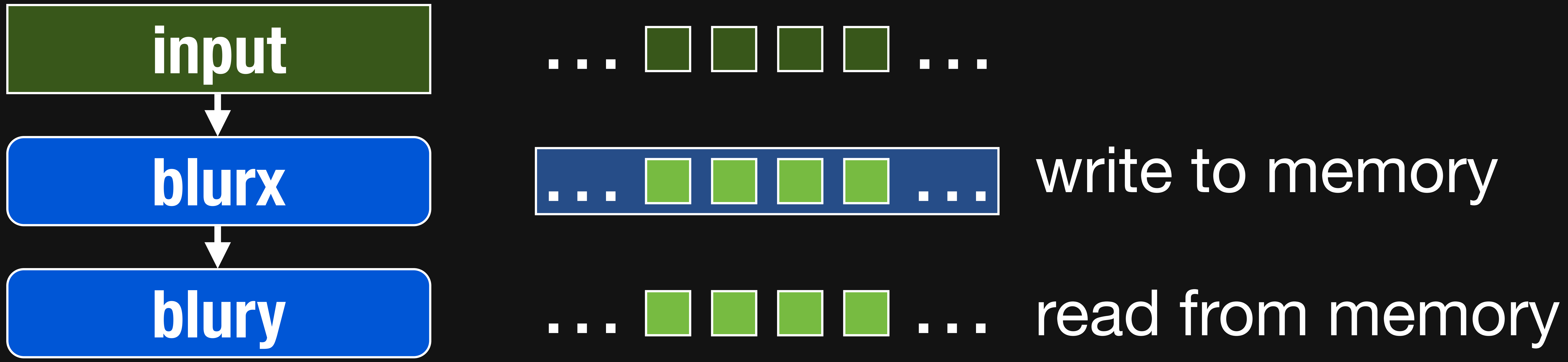
Organizing a data-parallel pipeline



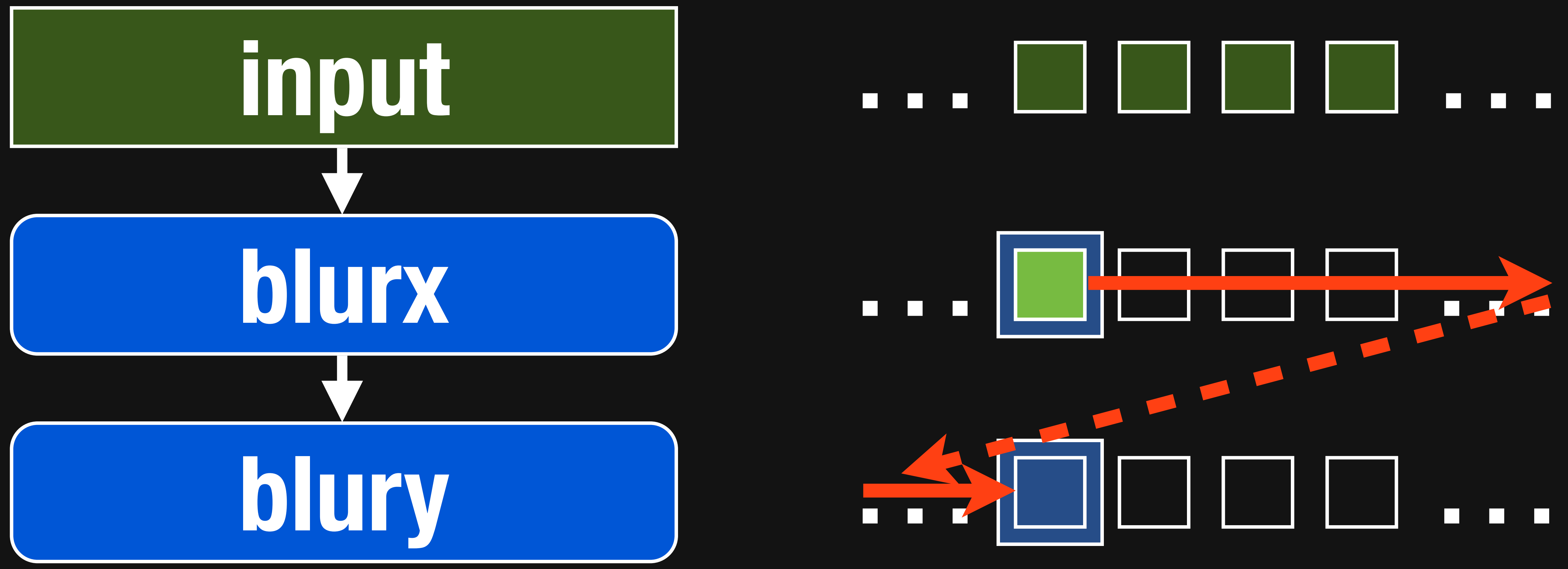
Simple loops execute **breadth-first** across stages



Simple loops execute **breadth-first** across stages

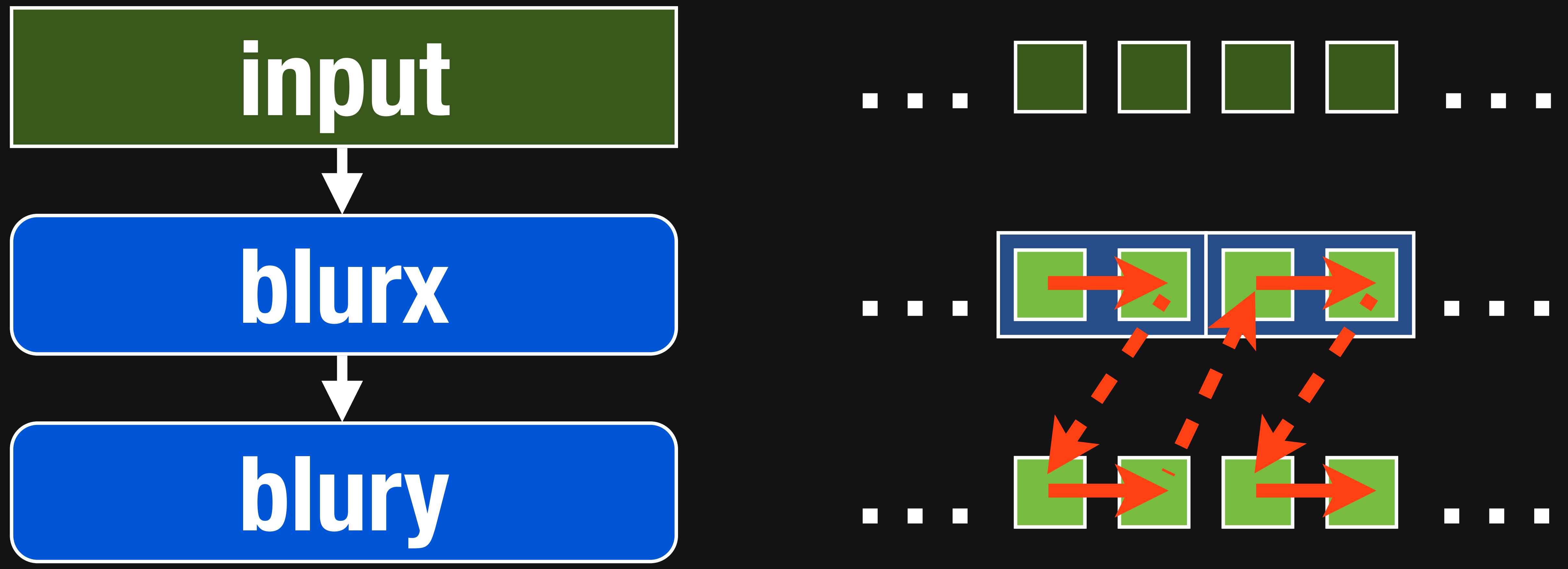


Breadth-first execution sacrifices locality



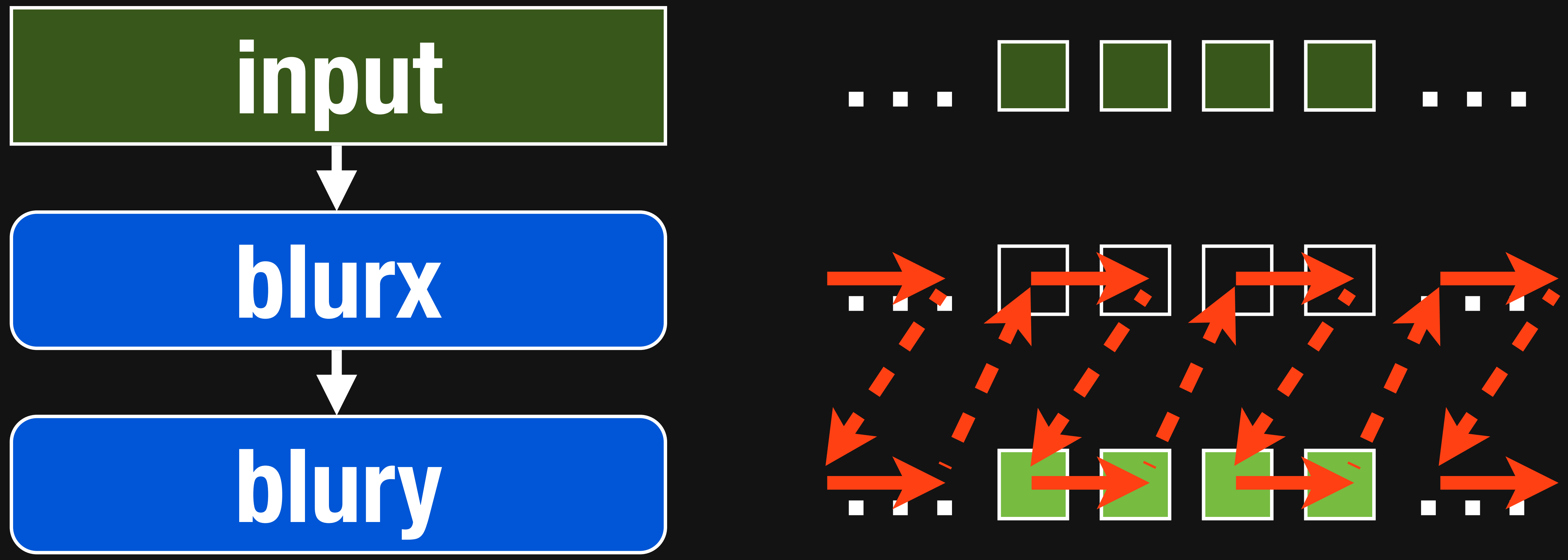
locality is a
function of
reuse distance

Interleaved execution improves locality



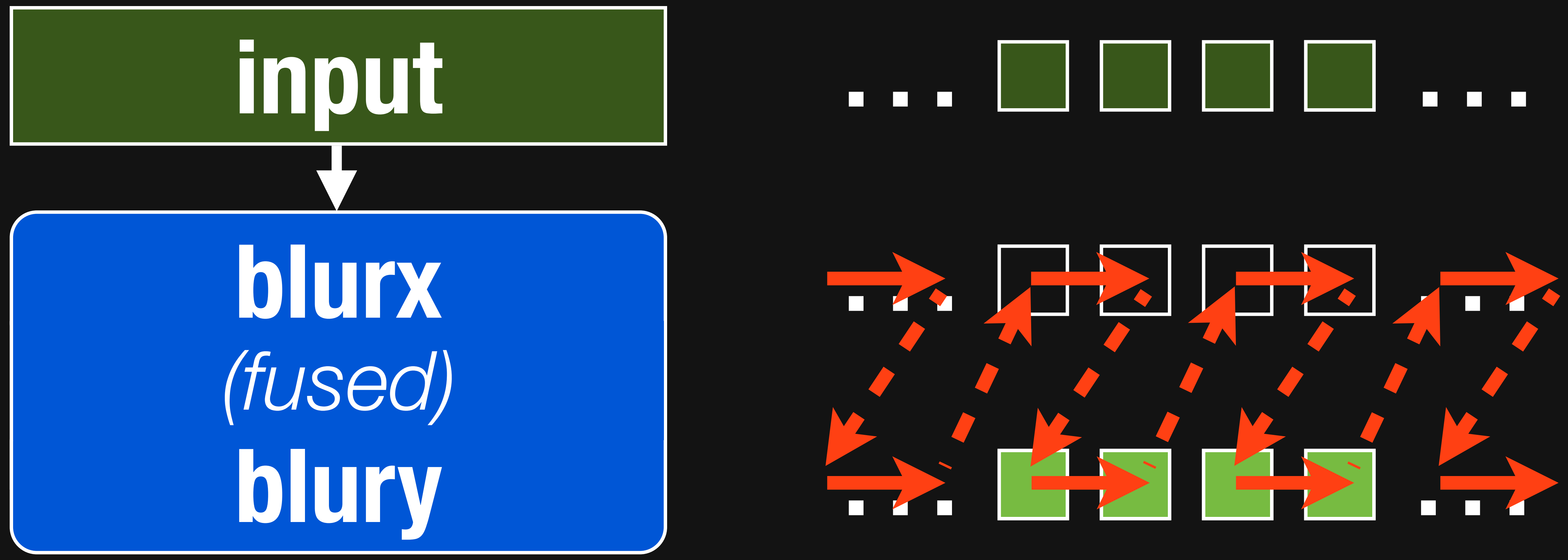
(simplified)

Interleaved execution improves locality



reduce reuse distance from **producer** to **consumer**

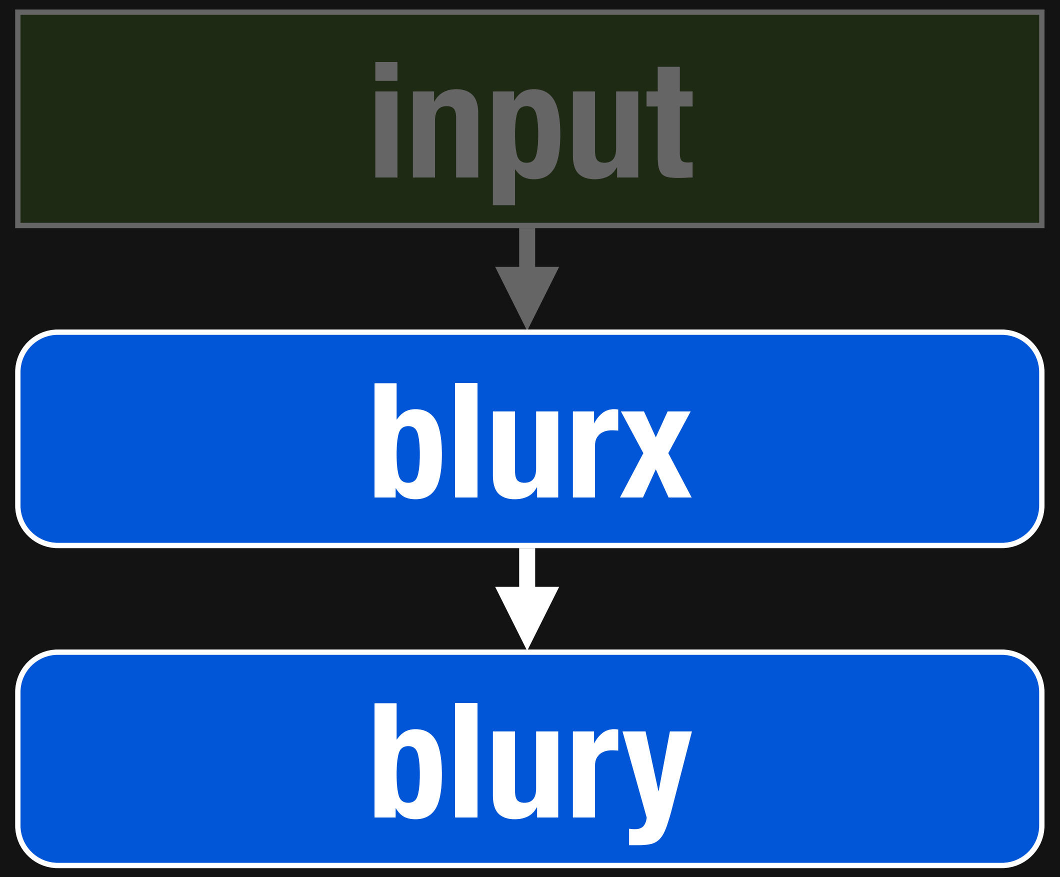
Fusion improves locality

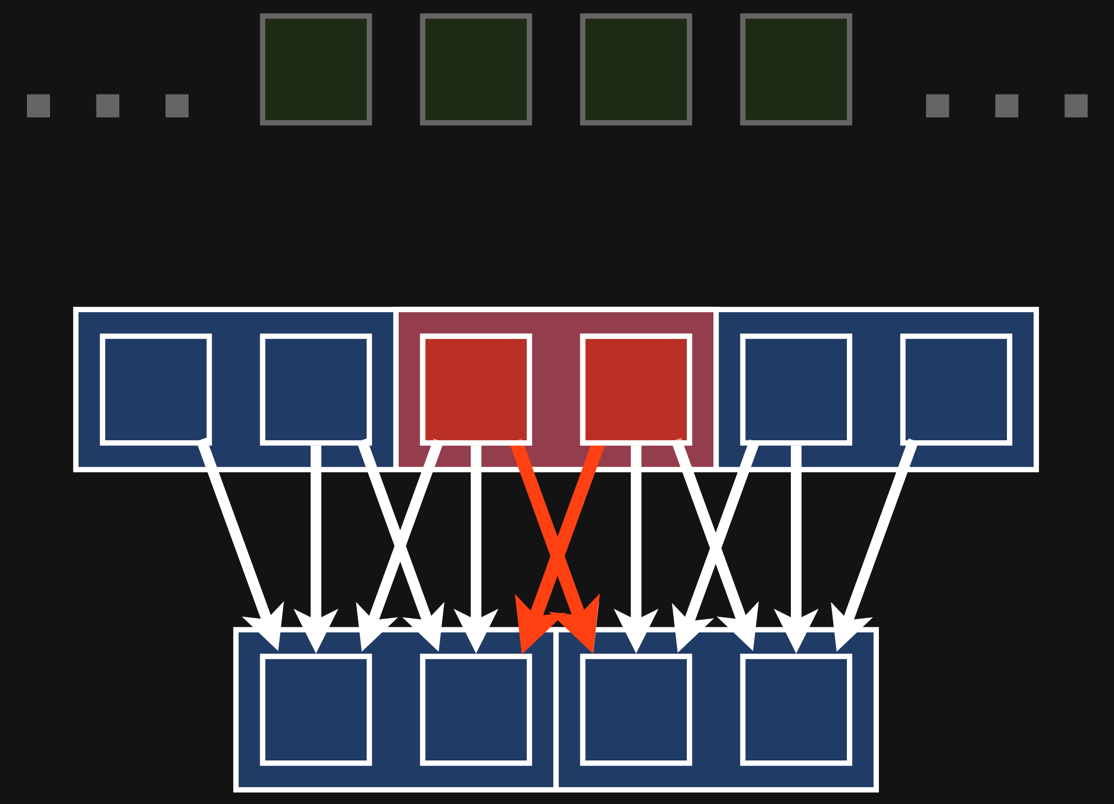


fusion **globally interleaves** computation

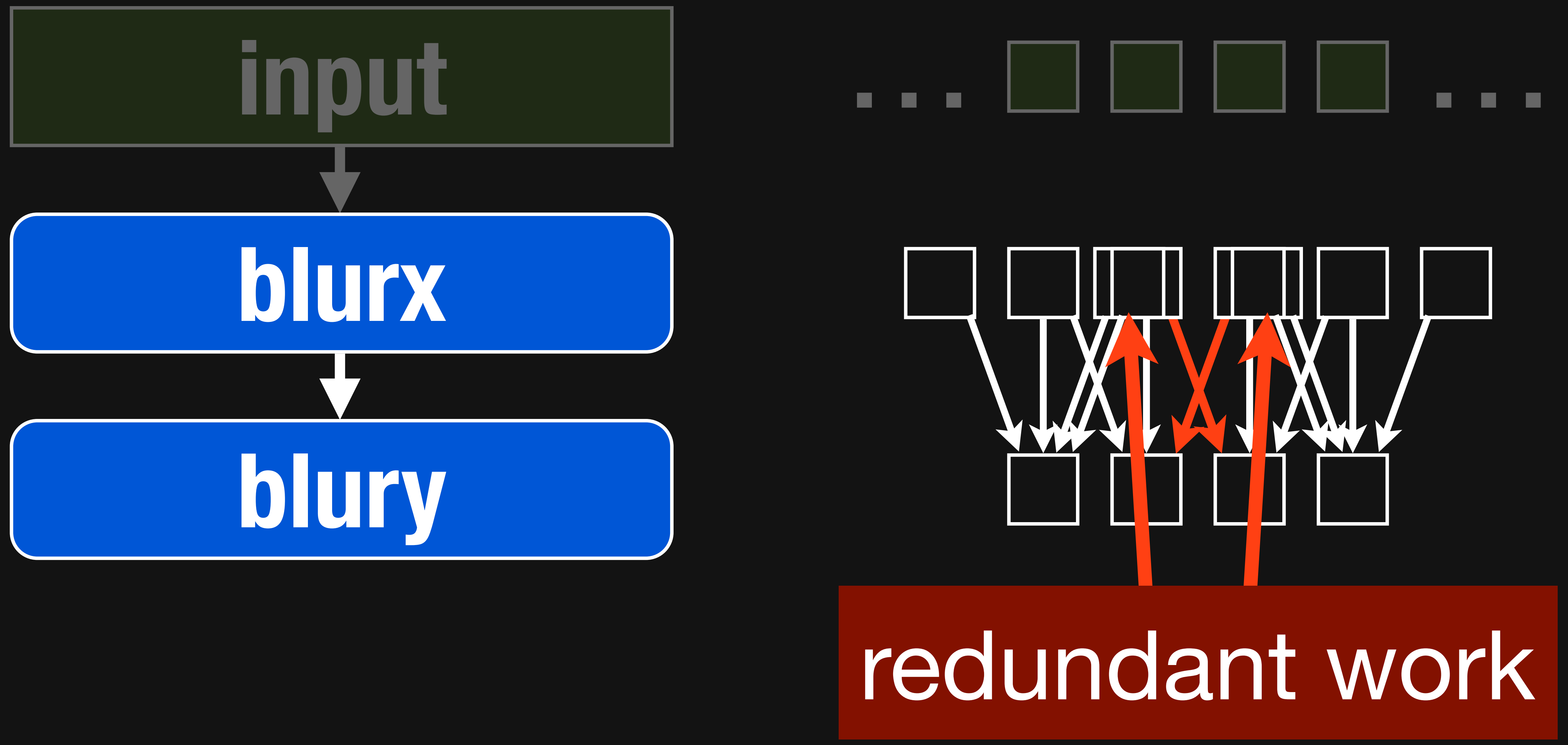
Understanding dependencies

Stencils have overlapping dependencies

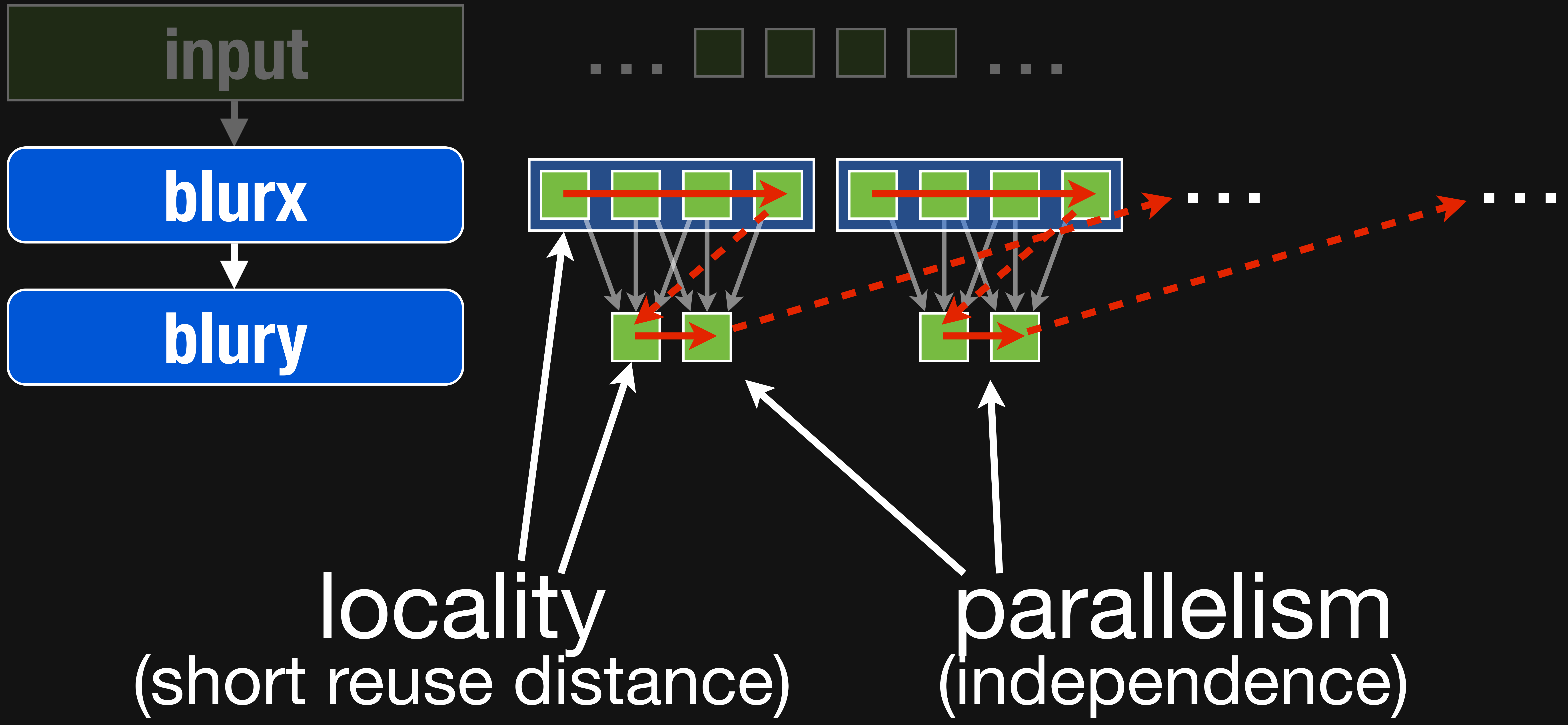




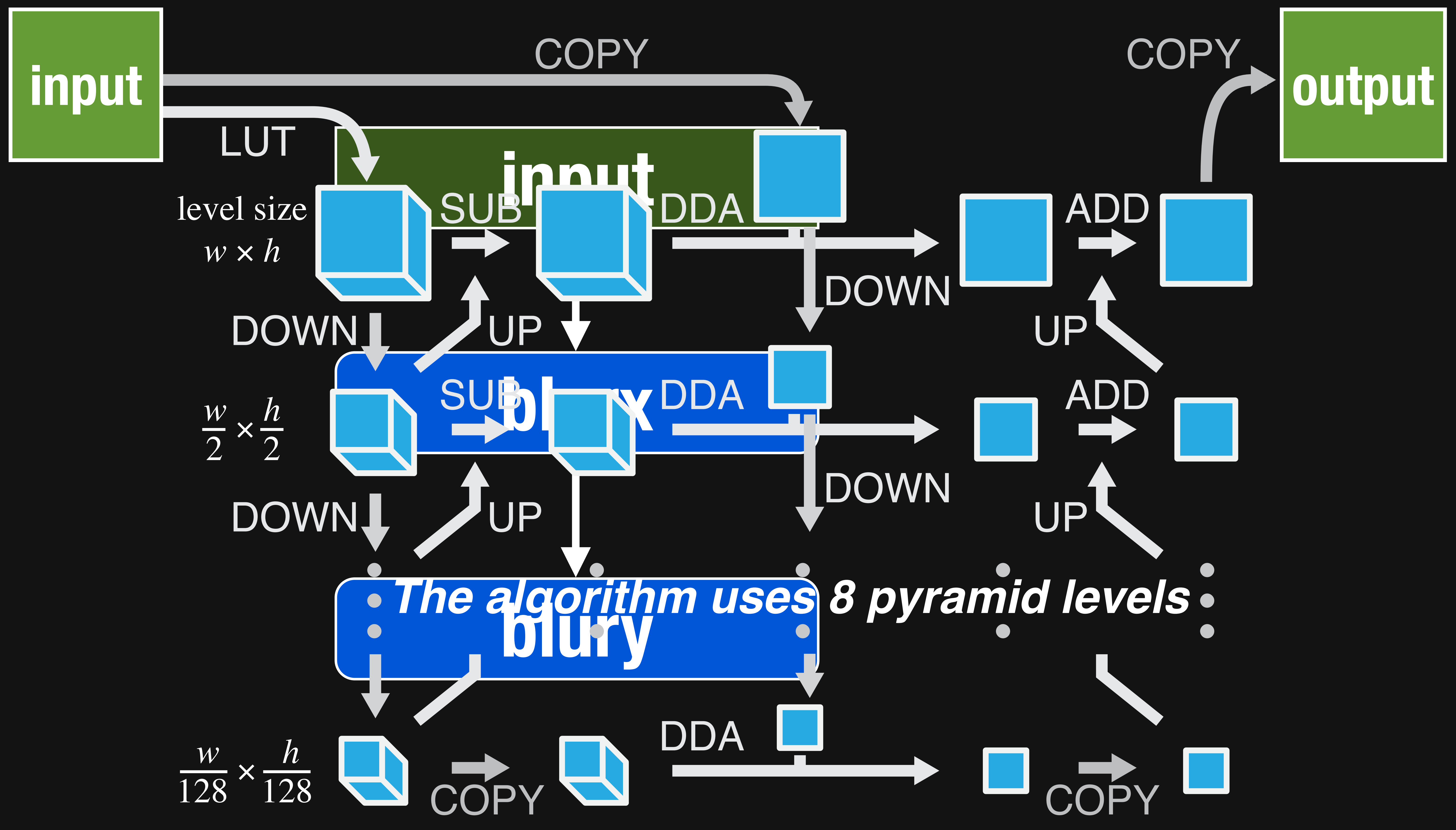
Breaking dependencies introduces redundant work



Decoupled tiles optimize parallelism & locality



Organization requires global tradeoffs



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

DDA: data-dependent access
$k \leftarrow \mathrm{floor}(I_1(x,y) / \sigma)$
$\alpha \leftarrow (I_1(x,y) / \sigma) - k$
$O(x,y) \leftarrow (1-\alpha)\, I_2(x,y,k) + \alpha\, I_2(x,y,k+1)$

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
$T_1 \leftarrow I \otimes_x [1\ 3\ 3\ 1]$
$T_2 \leftarrow T_1 \otimes_y [1\ 3\ 3\ 1]$
$O(x,y) \leftarrow T_2(2x,2y)$

local Laplacian filters
[Paris et al. 2010, Aubry et al. 2011]

3x3 box filter

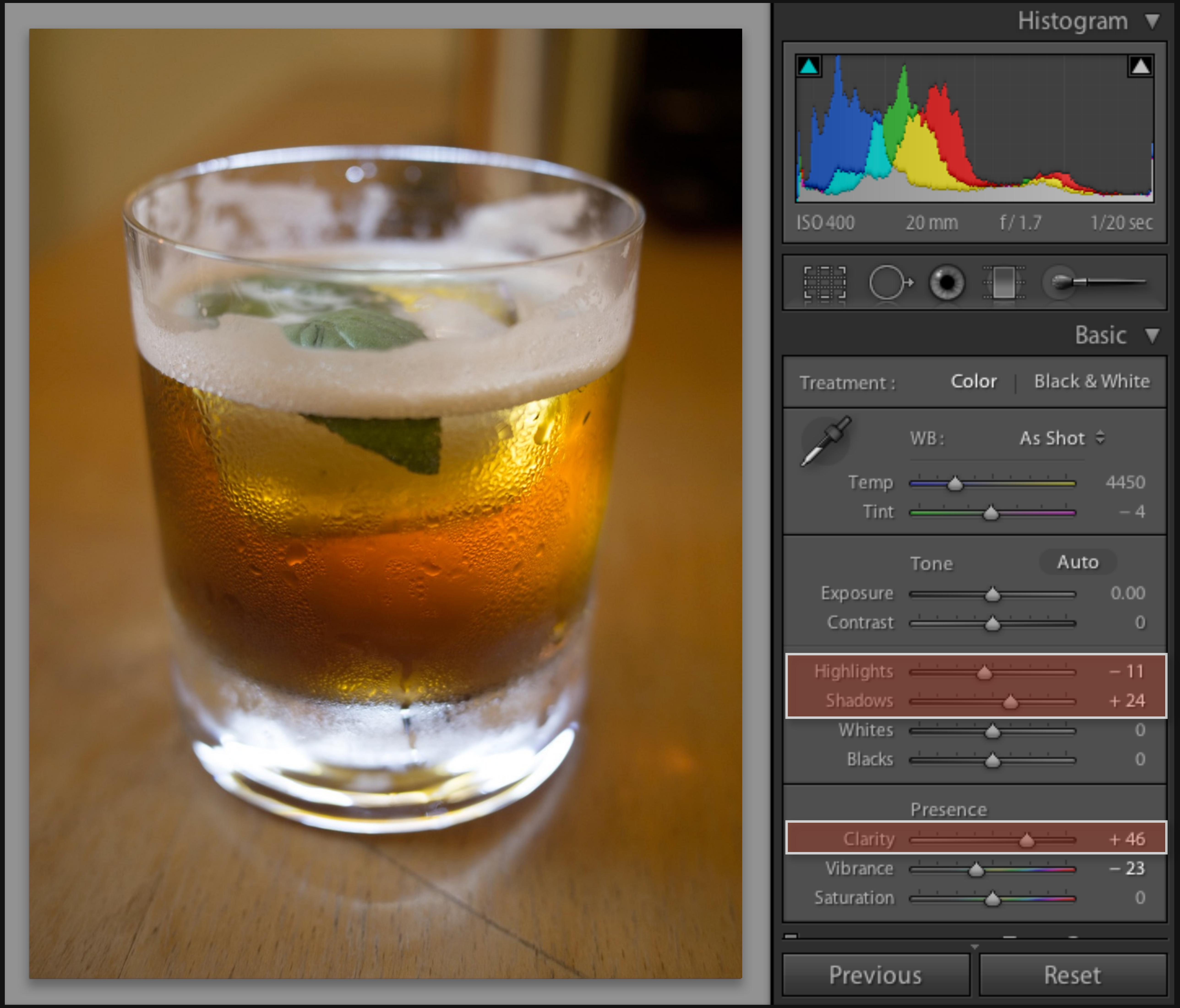
Local Laplacian Filters

prototype for Adobe Photoshop Camera Raw / Lightroom

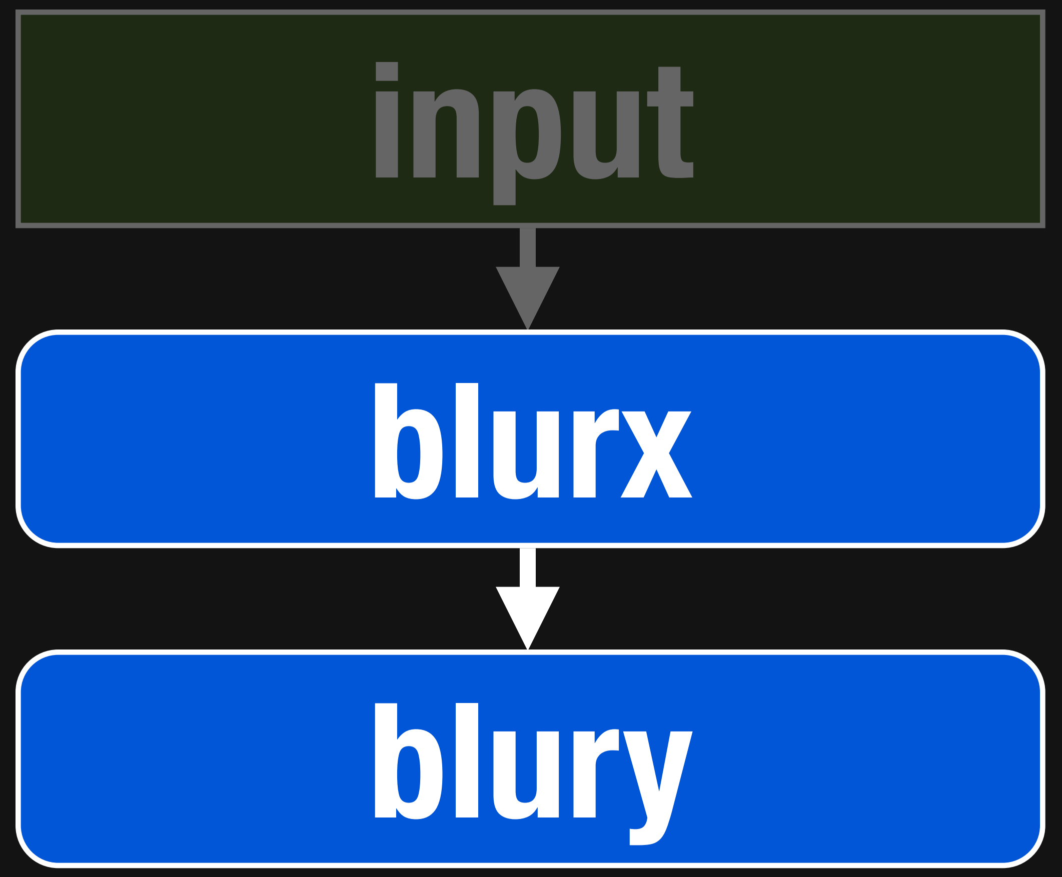
Adobe: 1500 lines of
expert-optimized C++
multi-threaded, SSE
3 months of work
***10x faster** than original C++*

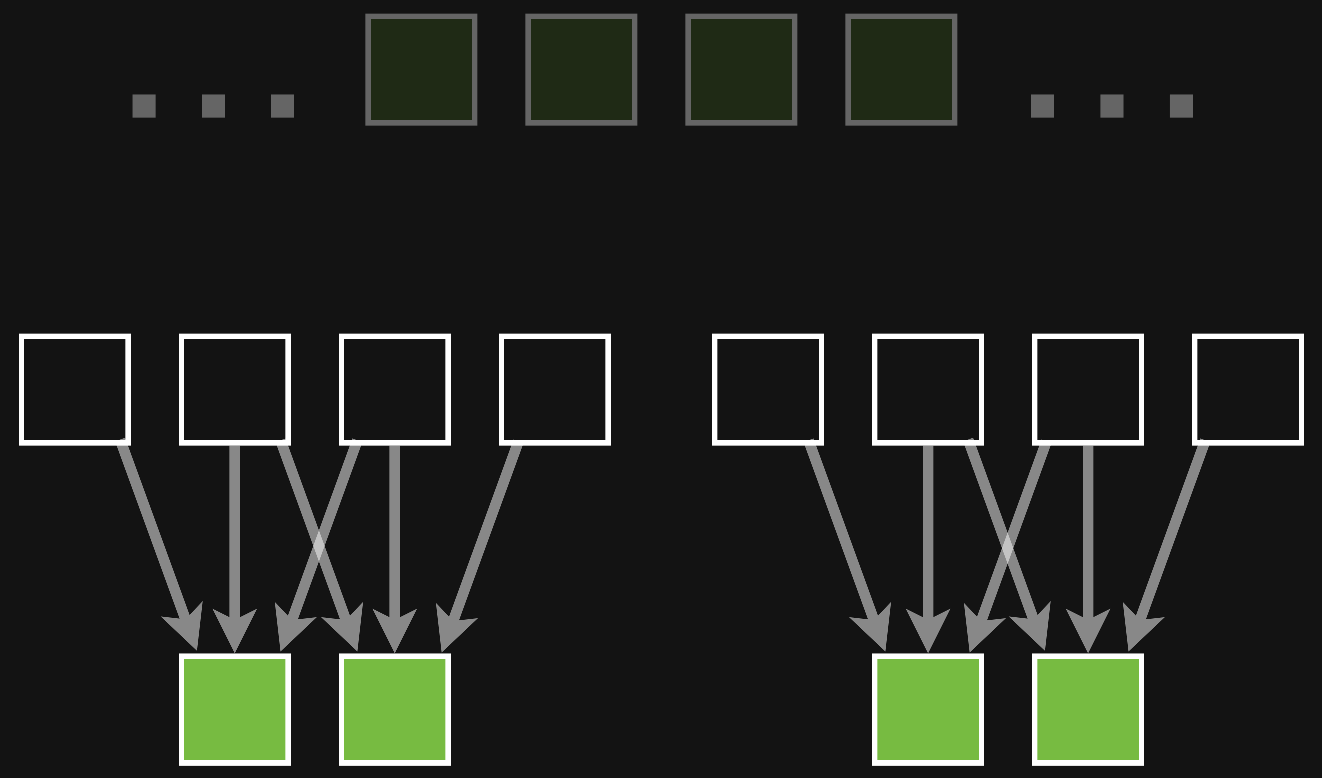
Halide: 60 lines
1 intern-day

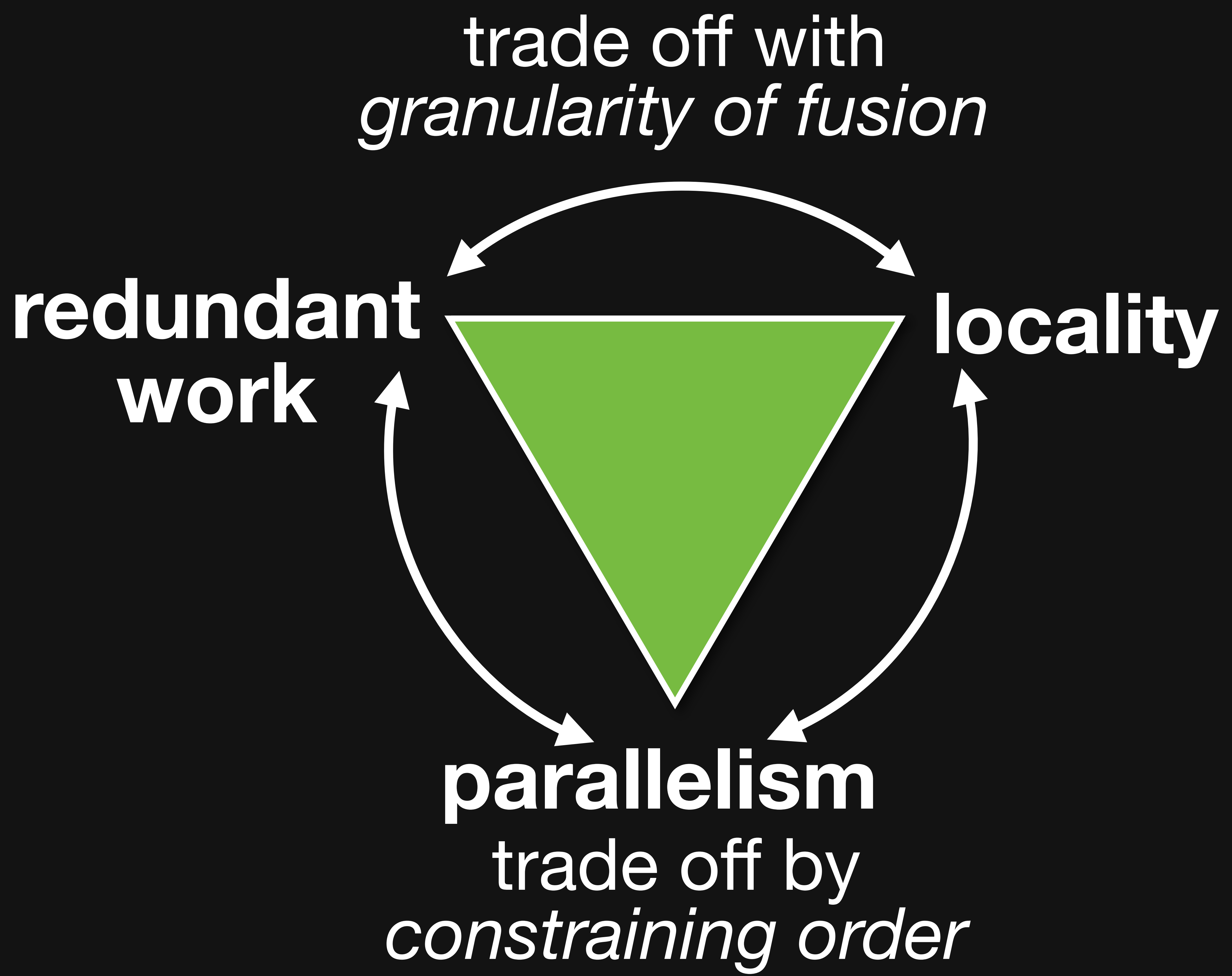
Halide vs. Adobe:
2x faster on same **CPU,**
10x faster on **GPU**



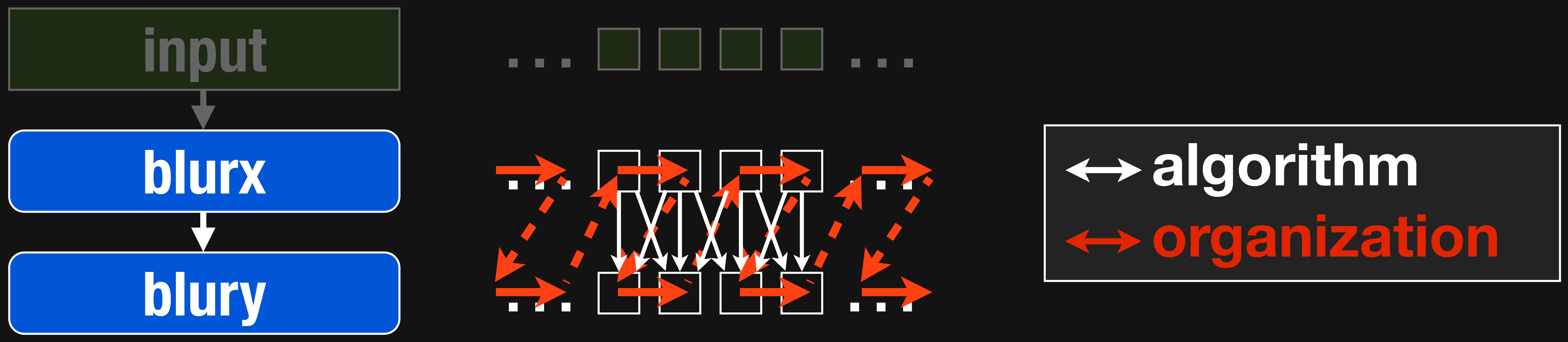
Message #1: performance requires tradeoffs





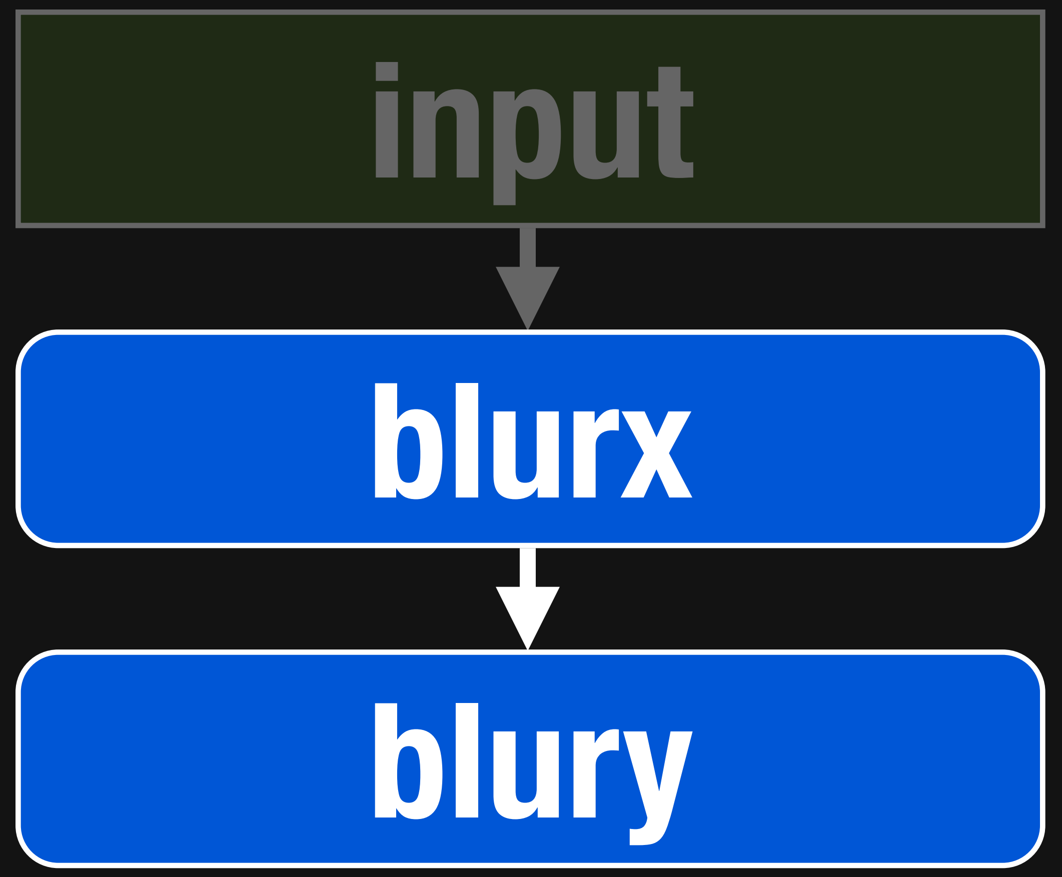


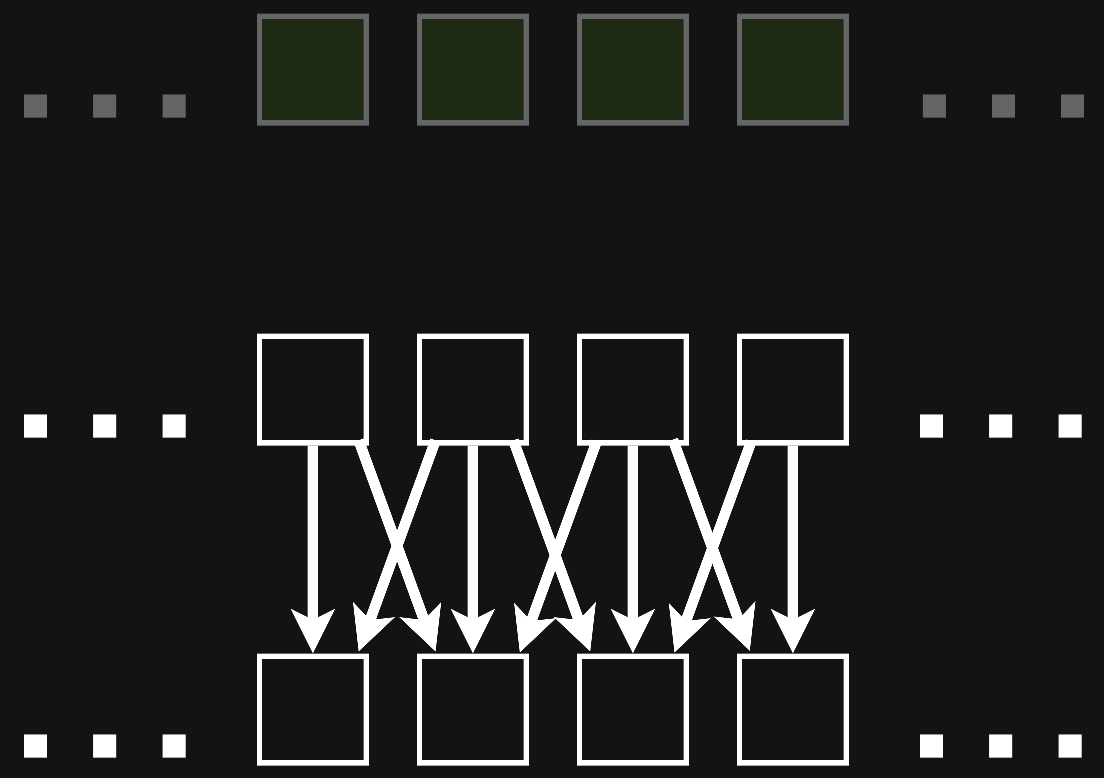
Message #2: algorithm vs. organization



order and interleaving
radically alter performance
of the *same algorithm*

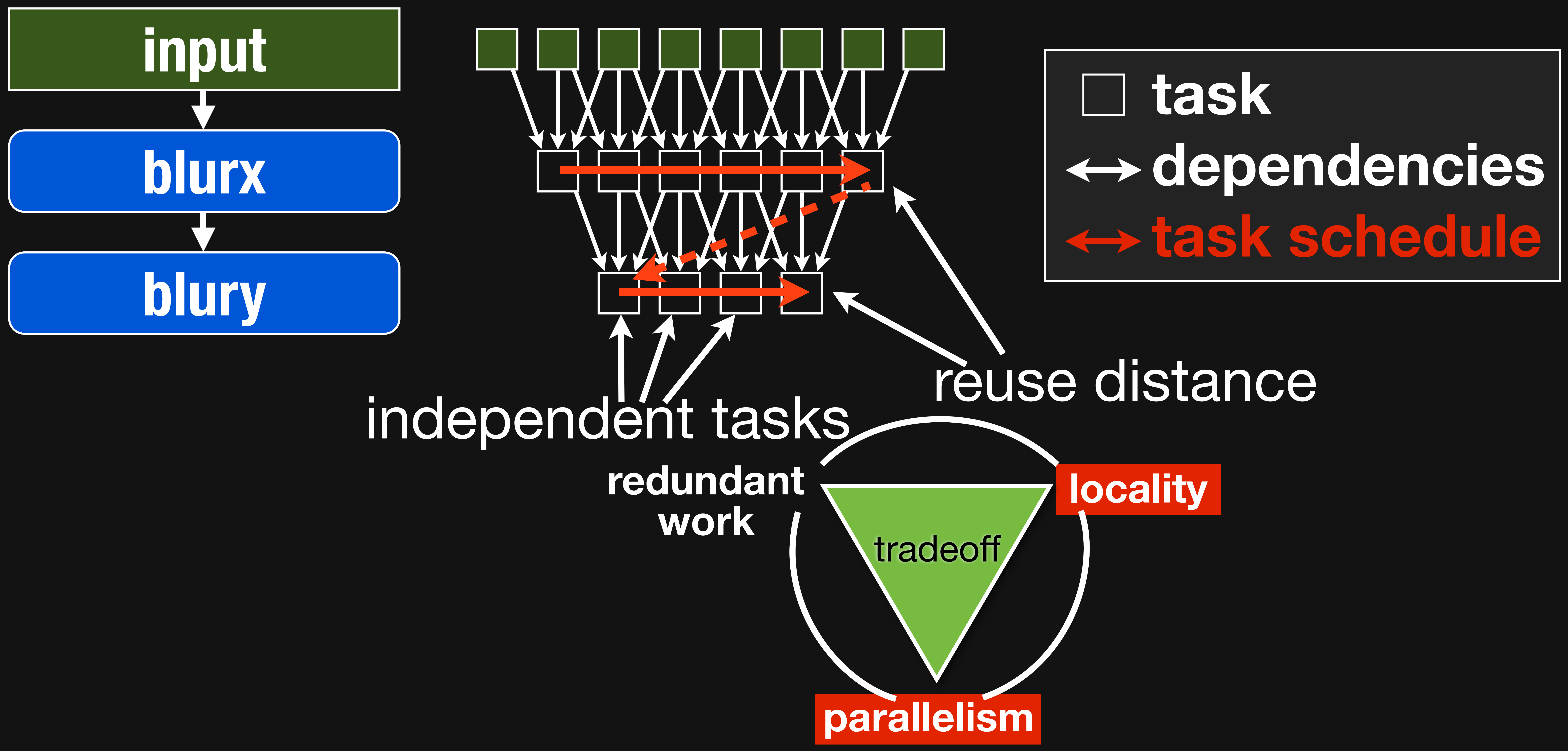
Message #3: dependencies limit choices of organization



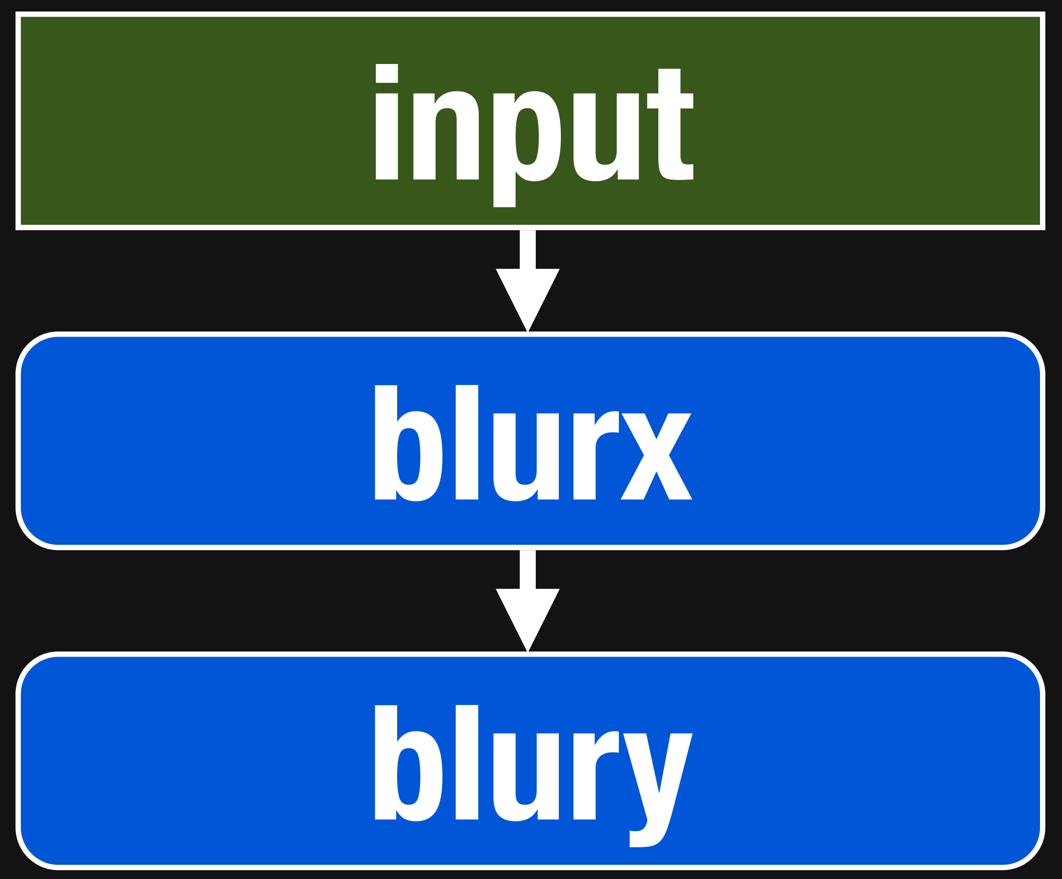


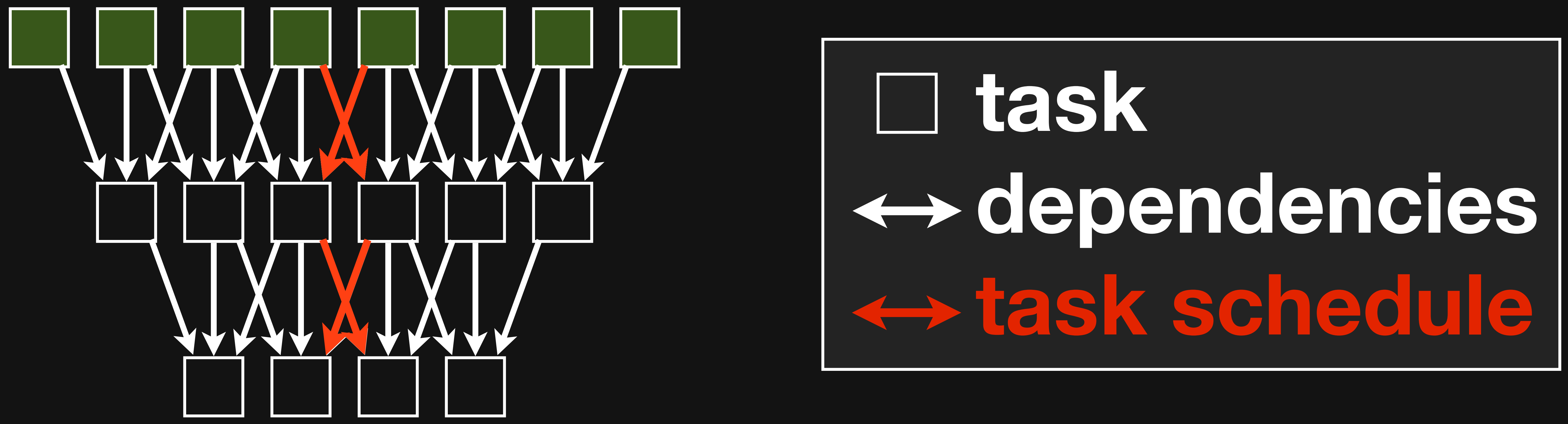
↔ dependencies
↔ organization

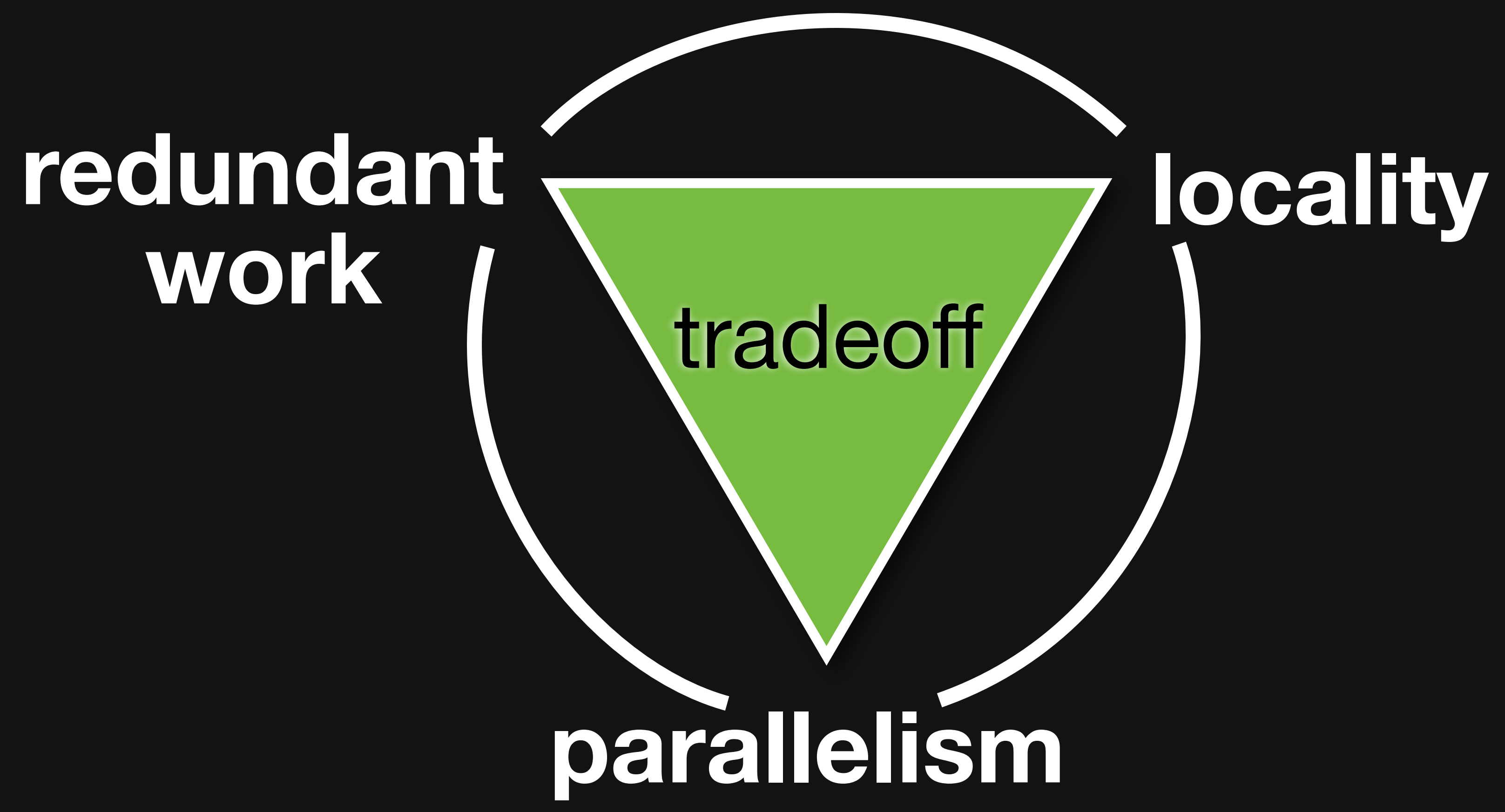
This is a general **task graph**



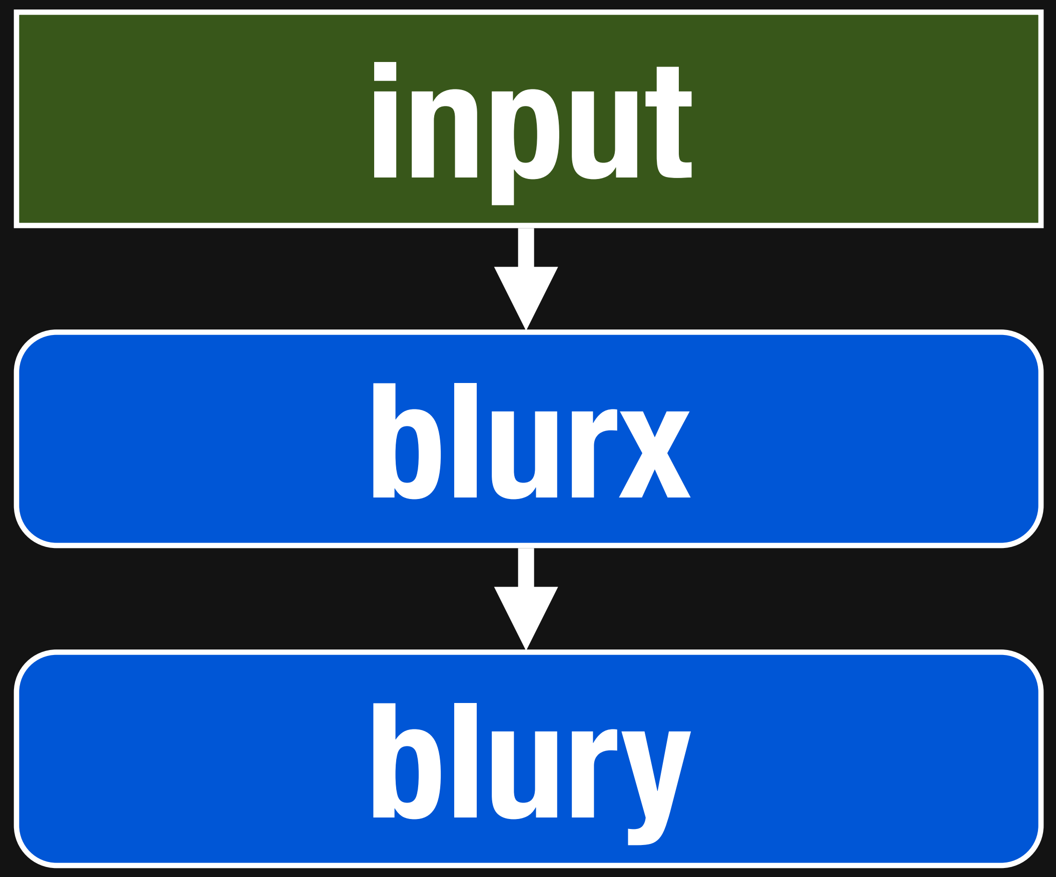
This is a general **task graph**

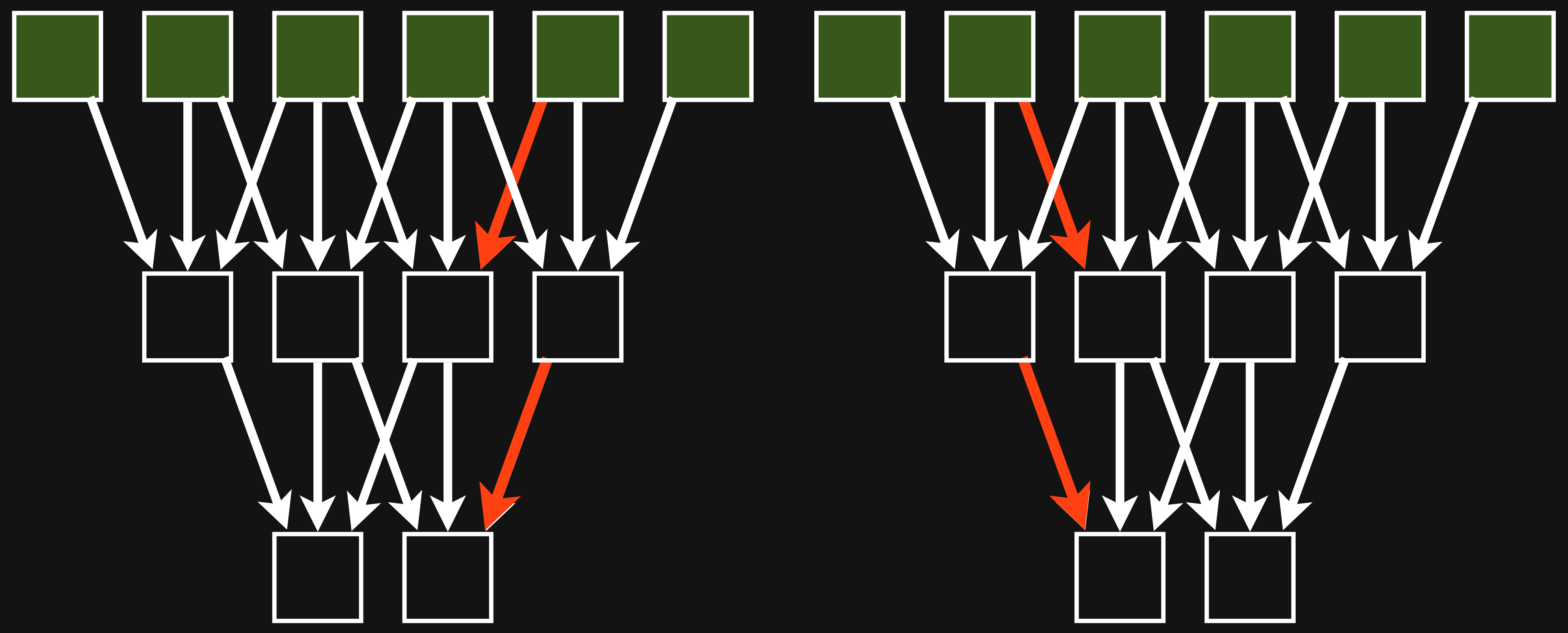


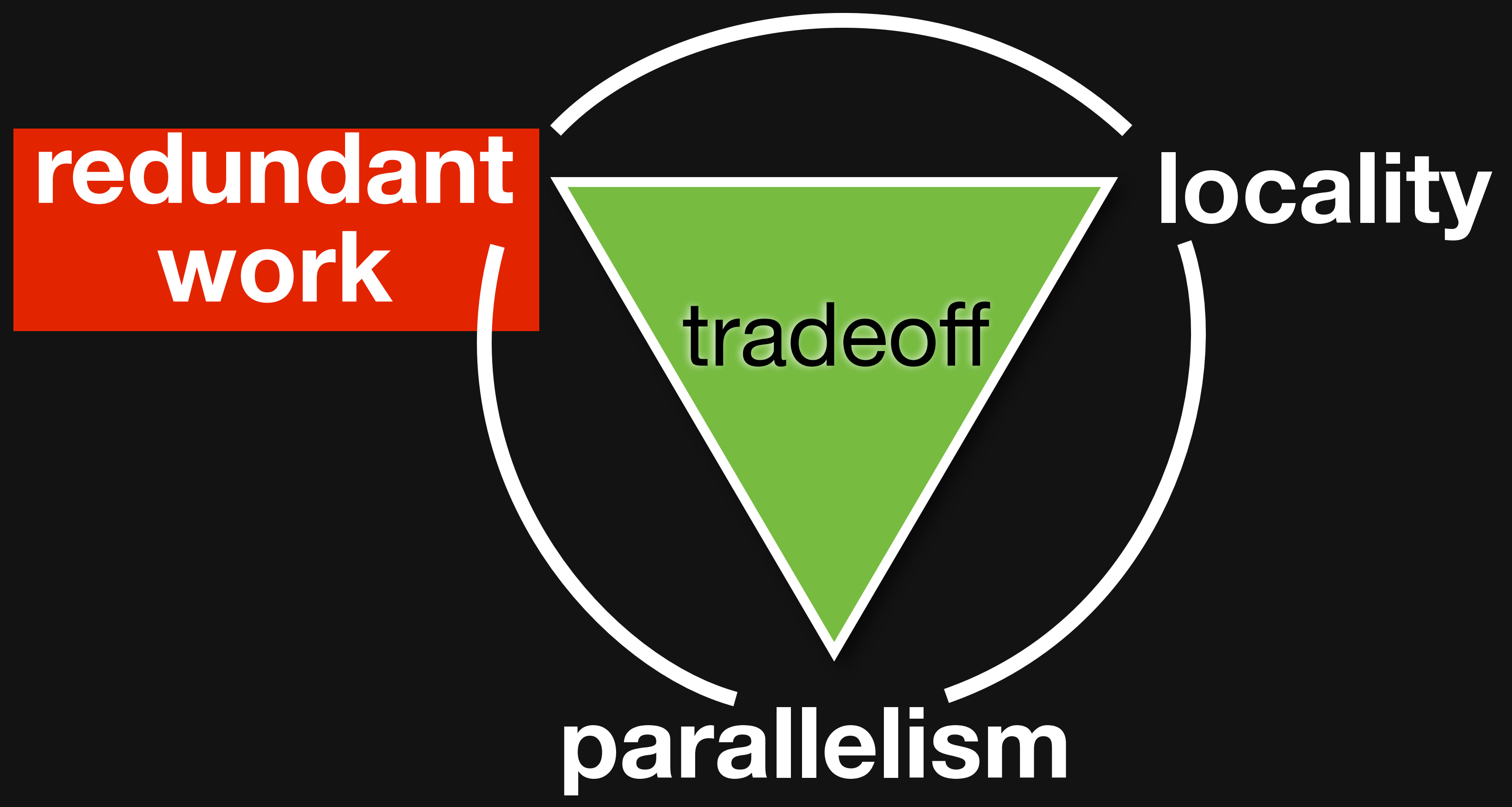




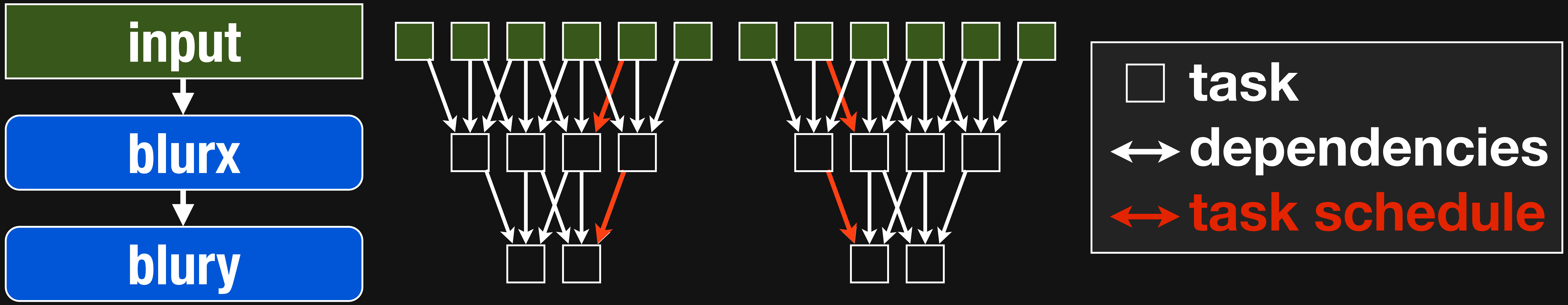
This is a general **task graph**

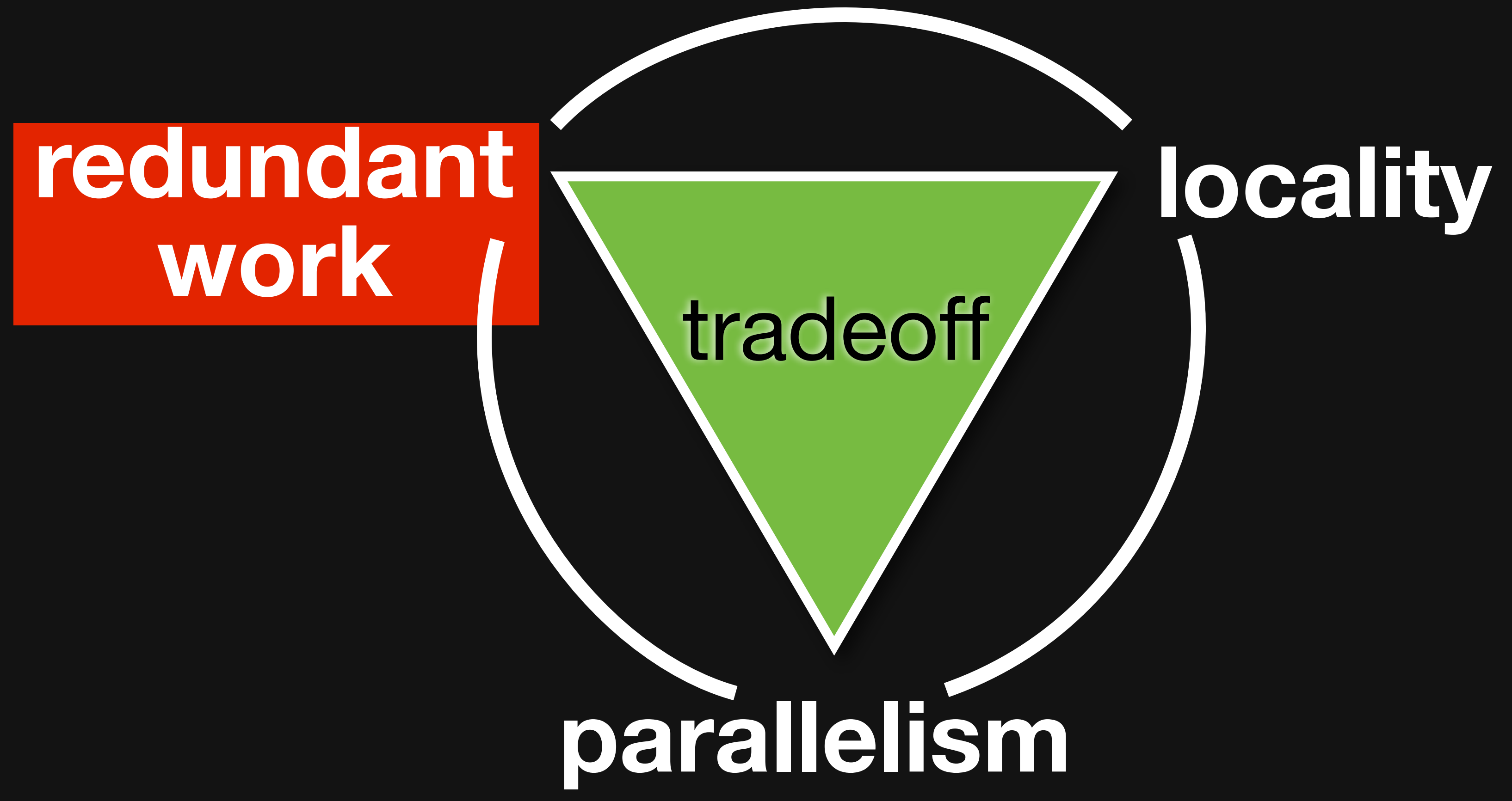






This is a general **task graph**





Traditional languages **conflate algorithm** & **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;
}
```

not readable

architecture-specific

hard to change organization
or algorithm

Optimized 3x3 blur in C++

```
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);
}}}}}
```

parallelism

distribute across threads

SIMD parallel vectors

Optimized 3x3 blur in C++

```
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);
}}}}}
```

parallelism

distribute across threads
SIMD parallel vectors

locality

reorganize computation:
fuse two blurs,
compute in tiles

The effect of organization on performance

	Performance (vs. root baseline)
Breadth-first	1 ×
Breadth-first + parallel	4 ×
Interleaving alone	0.8 ×
Interleaving + parallel	11.5 ×

Same algorithm, different organization

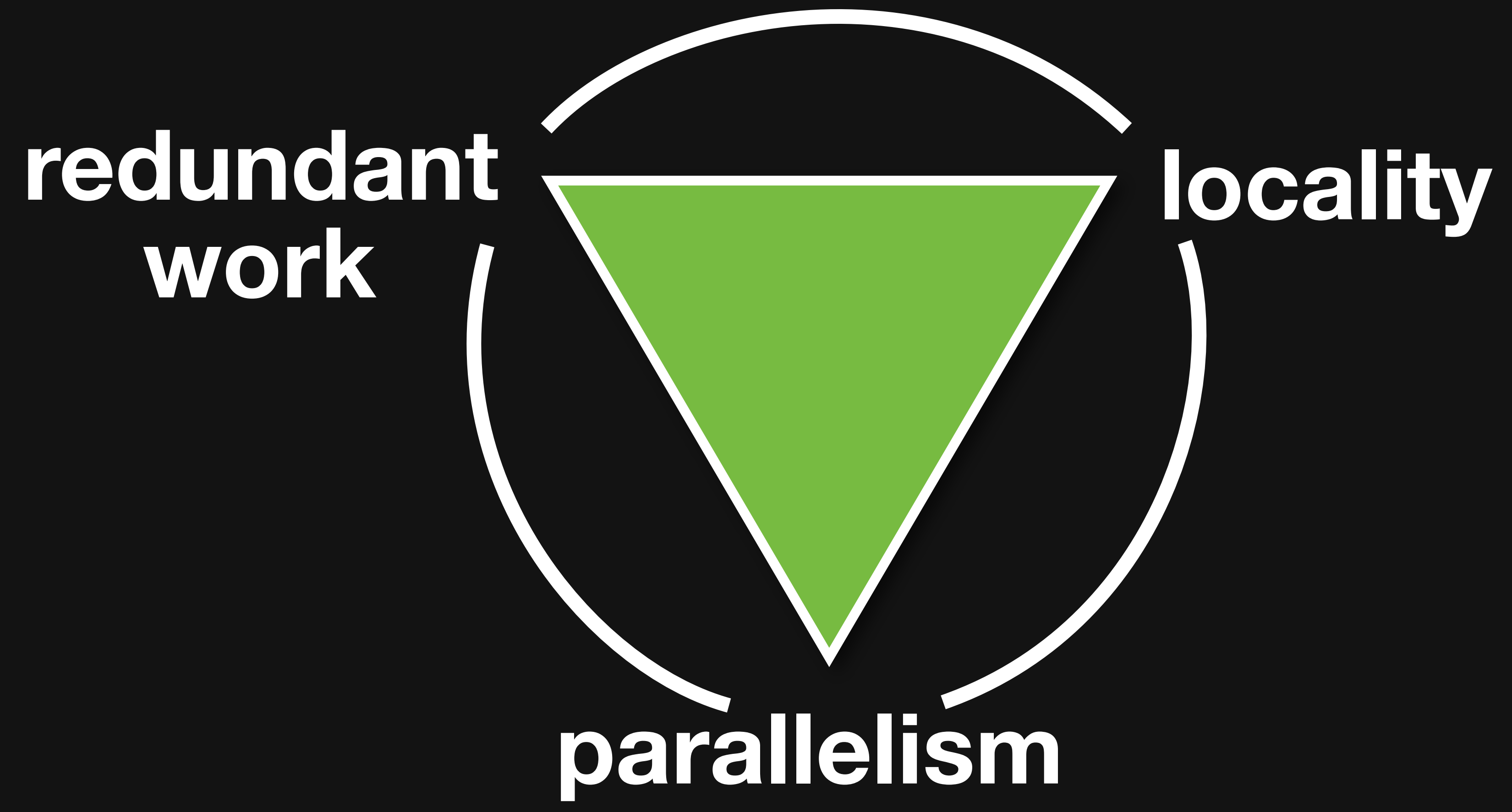
```
void box_filter_3x3(const Image &in, Image &blury) {
  __m128i one_third = _mm_set1_epi16(21846);
  #pragma omp parallel for
  for (int yTile = 0; yTile < in.height(); yTile += 32) {
    __m128i a, b, c, sum, avg;
    __m128i blurx[(256/8)*(32+2)]; // allocate tile blurx array
    for (int xTile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



Halide's answer:
decouple algorithm from schedule

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

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*:

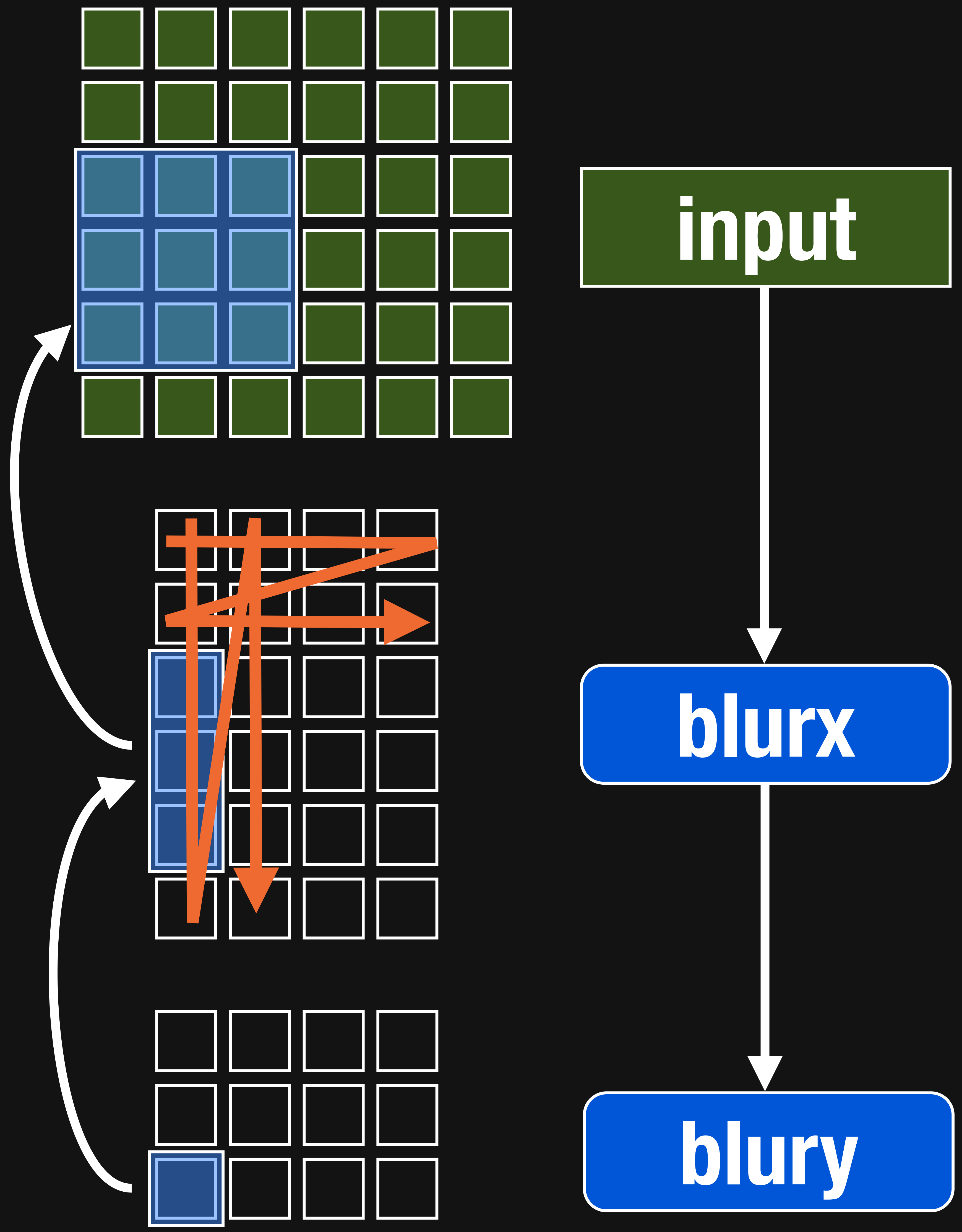
```
blurx(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blury(x, y) = (blurx(x, y-1) + blurx(x, y) + blurx(x, y+1))/3;
```

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

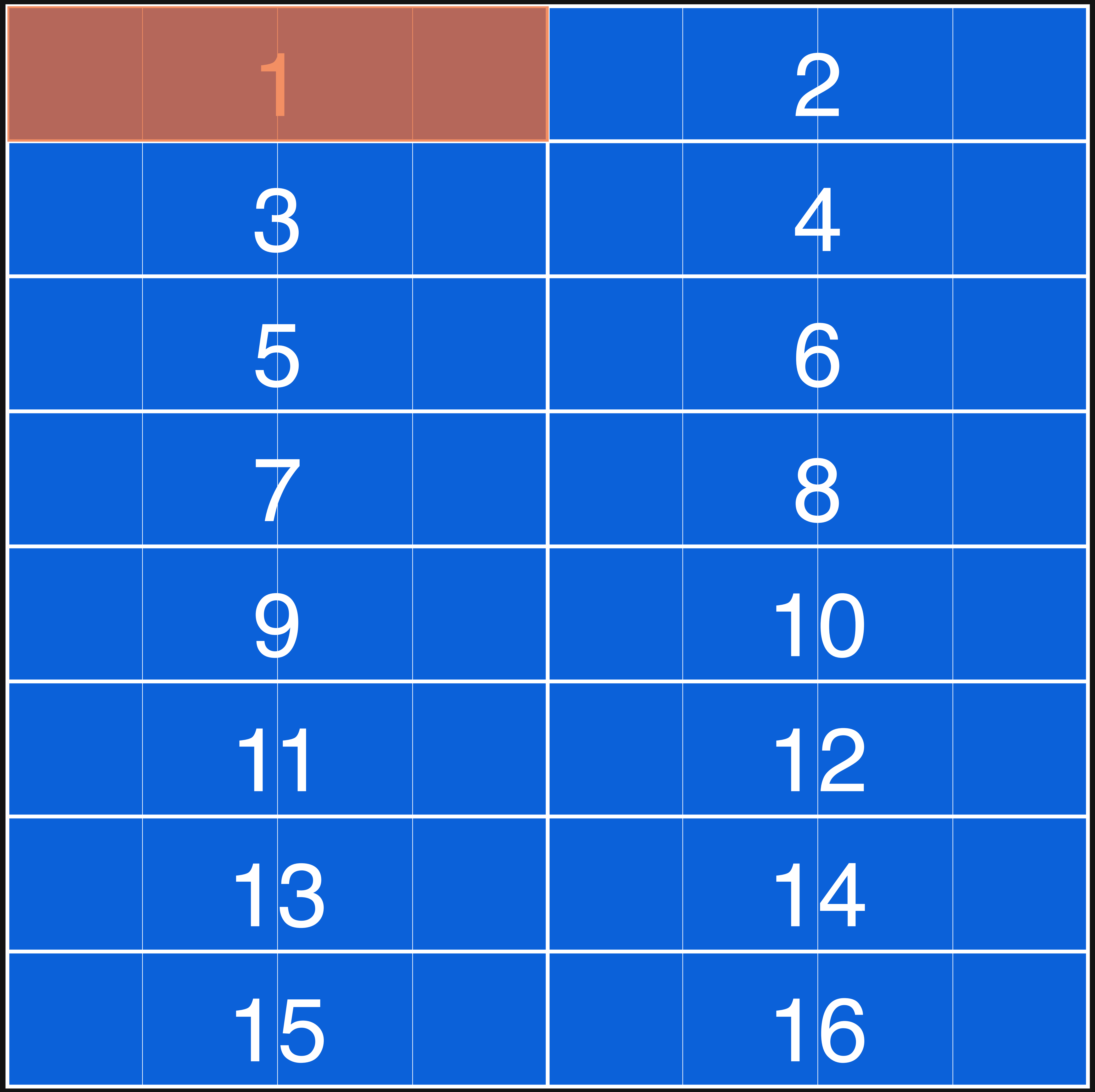
The schedule defines order & parallelism within stages

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

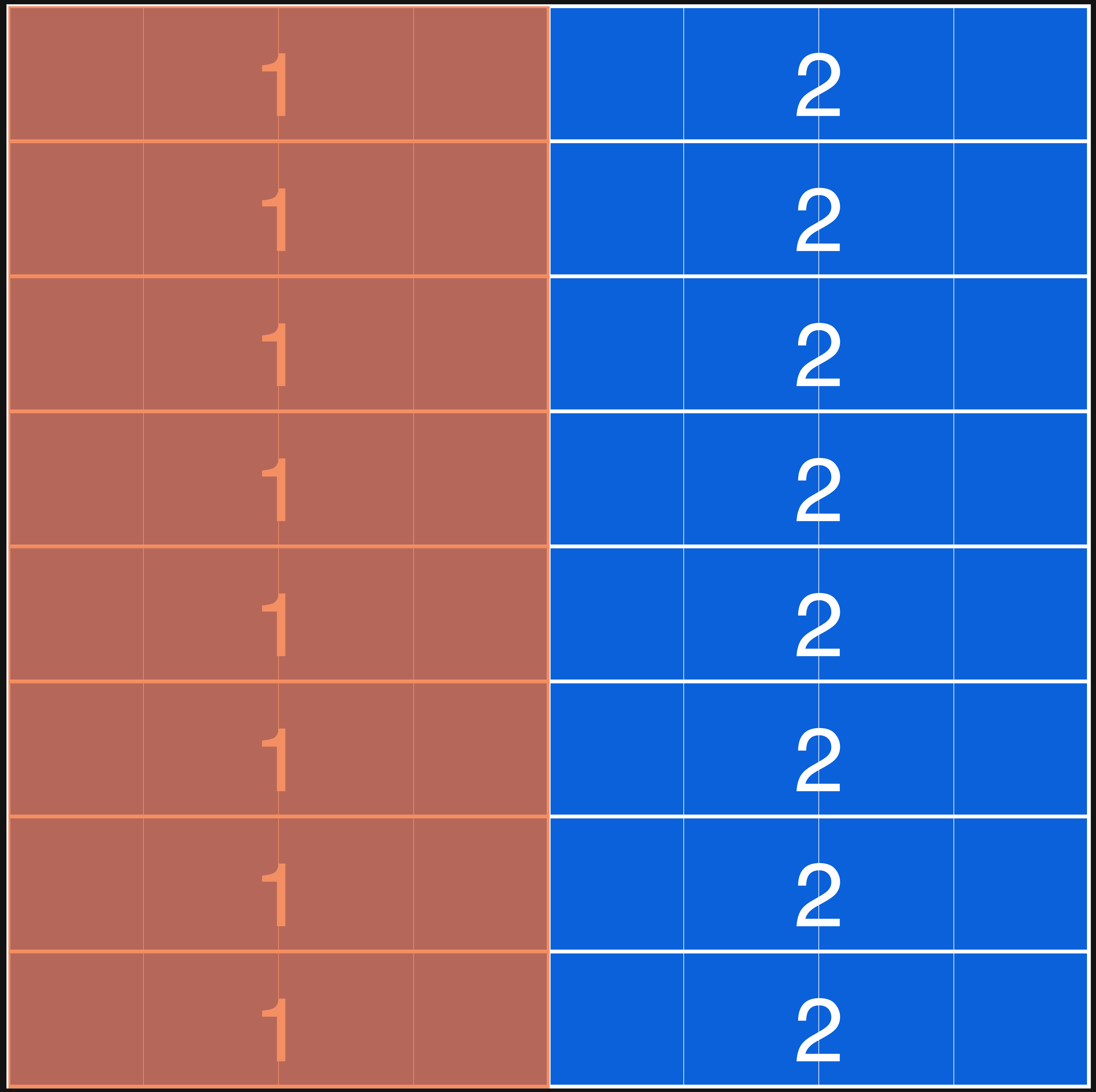
The schedule defines order & parallelism within stages

Serial y,
Vectorize x by 4



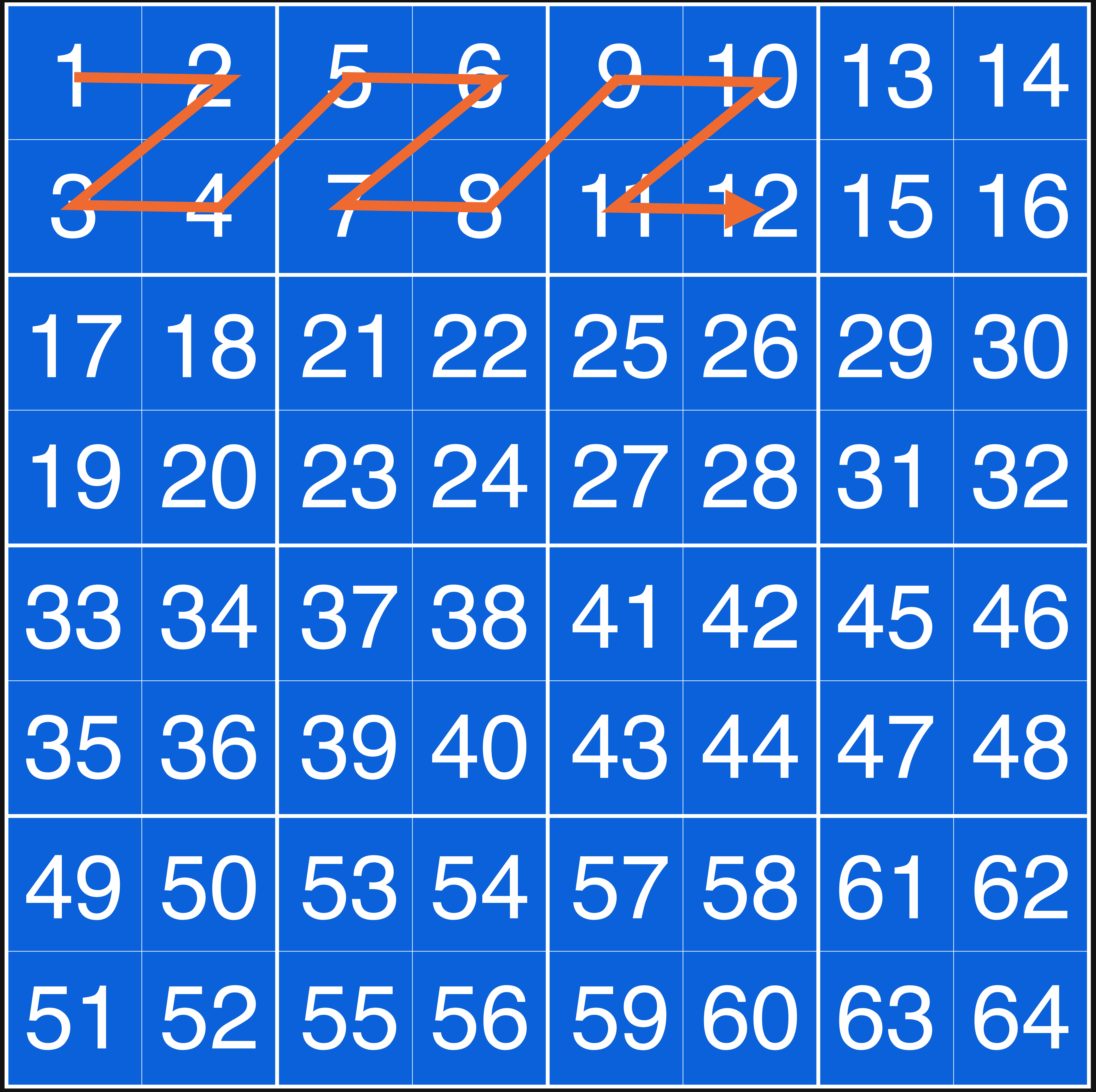
The schedule defines order & parallelism within stages

Parallel y,
Vectorize x by 4



The schedule defines order & parallelism within stages

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



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

Serial y,
Serial x

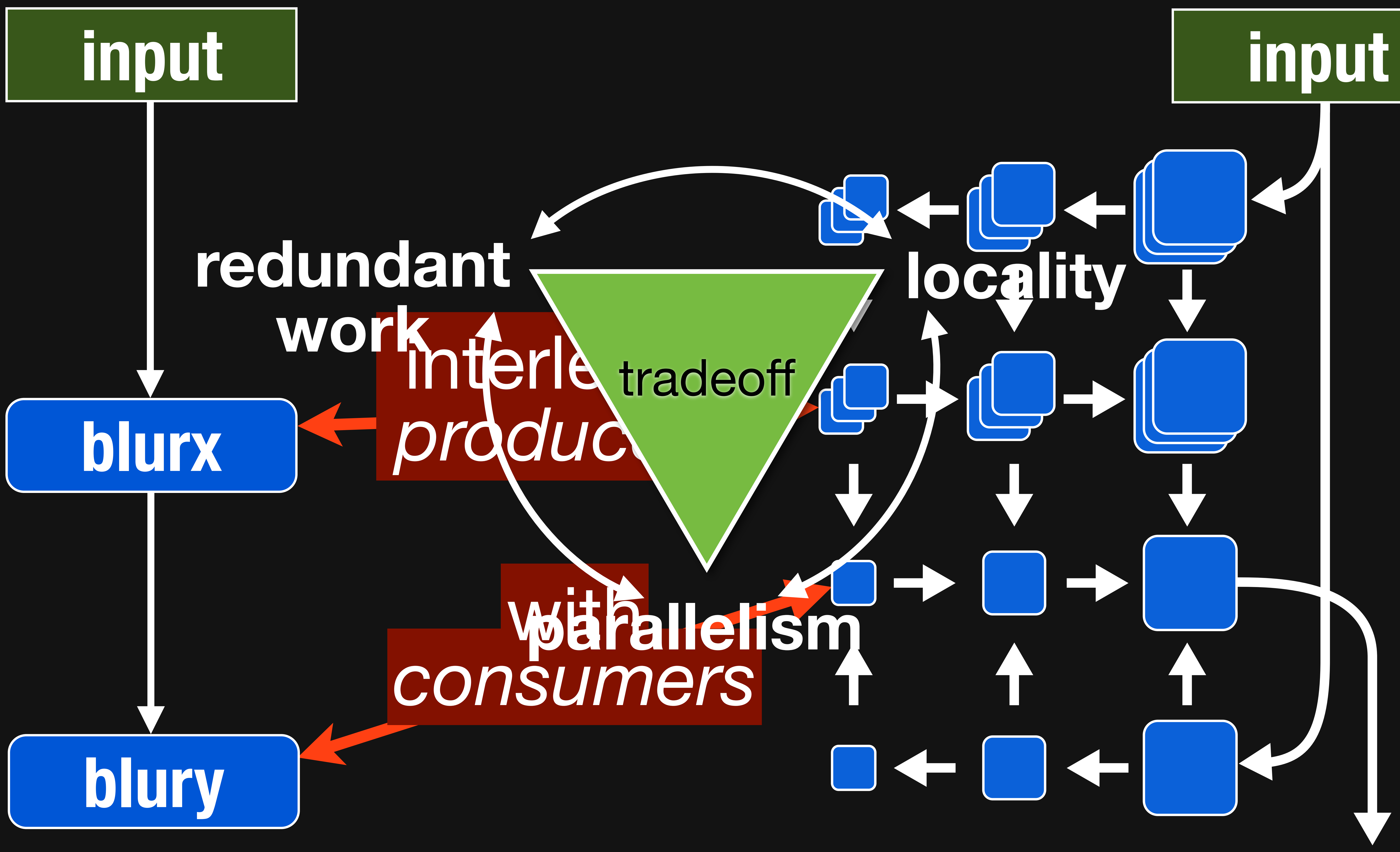
```
for (y : y_min..y_max)
  for (x : x_min..x_max) {
    eval[ f(x, y) ]
  }
```

Split x by 4,
Split y by 4.
Parallel y_o,
Serial x_o,
Serial y_i,
Vectorize x_i by 4

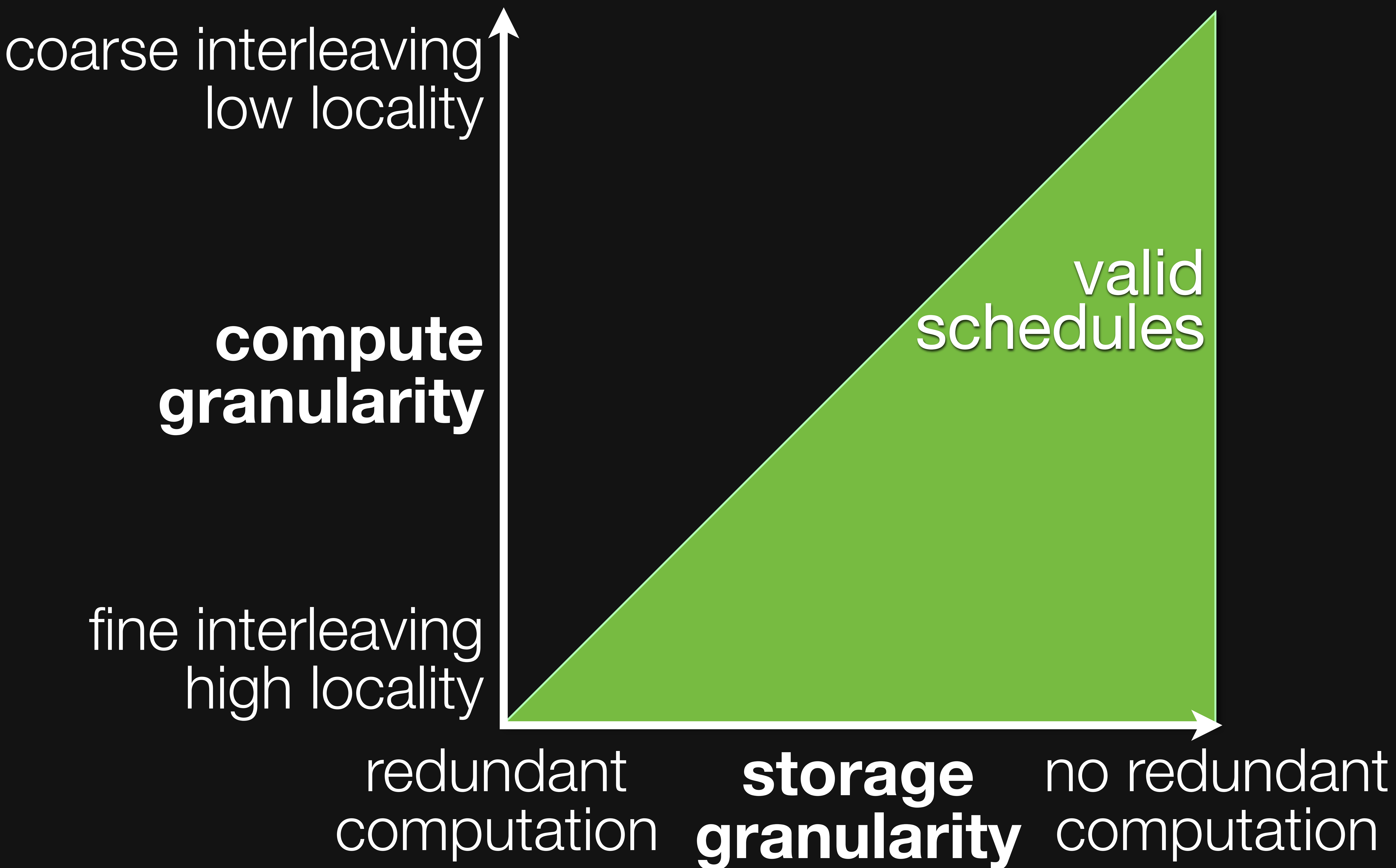
```
f.split(x, xo, xi, 4)
 .split(y, yo, yi, 4)
 .reorder(yo, xo, yi, xi)
 .parallel(yo)
 .vectorize(xi, 4)
```

```
parfor (y_o : y_o_min..y_o_max)
  for (x_o : x_o_min..x_o_max)
    for (y_i : y_i_min..y_i_max)
      simdfor (x_i : x_i_min..x_i_max by 4) {
        eval<4>[ f(x_o*4+x_i, y_o*4+y_i) ]
      }
```

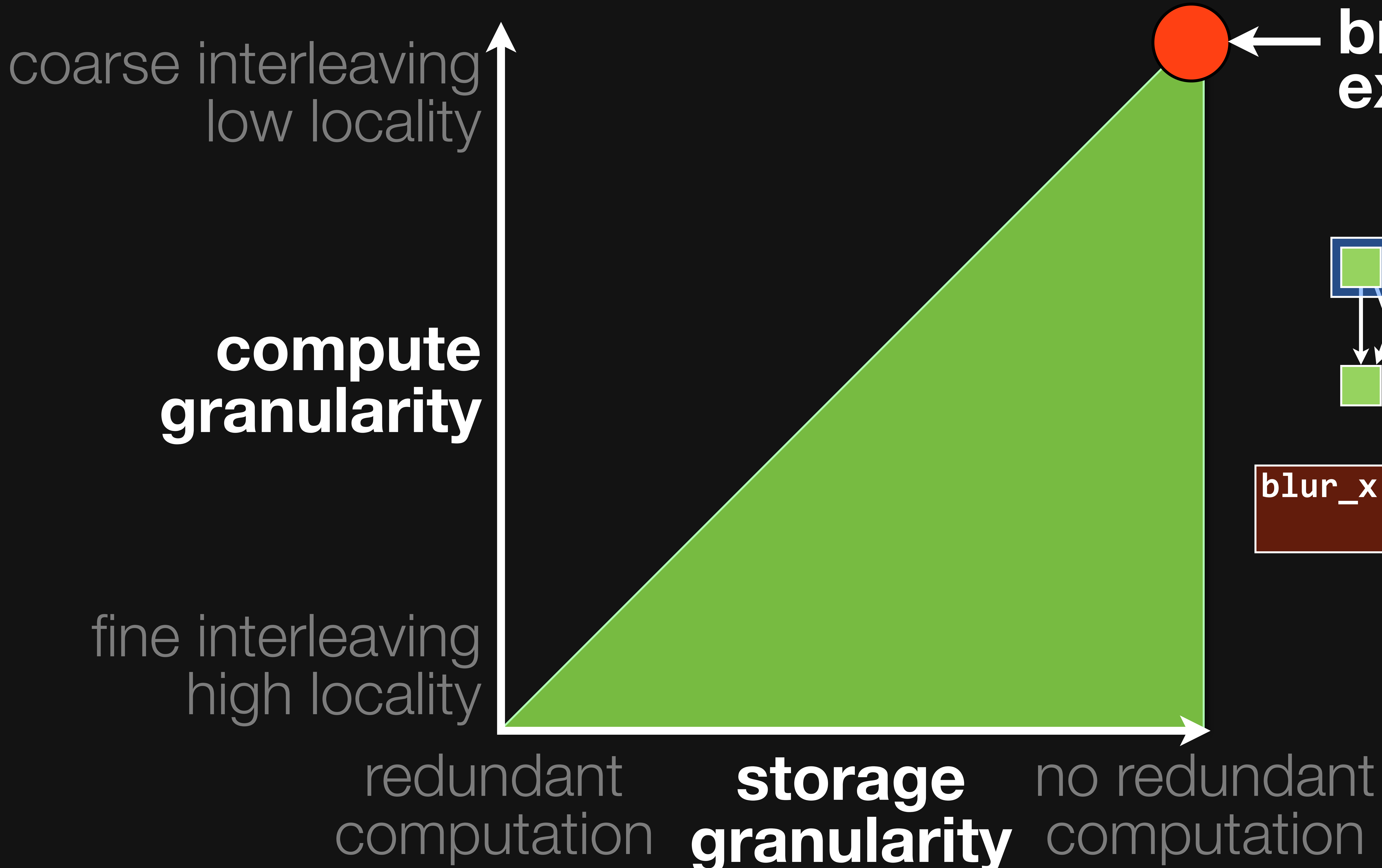
The schedule defines producer-consumer interleaving



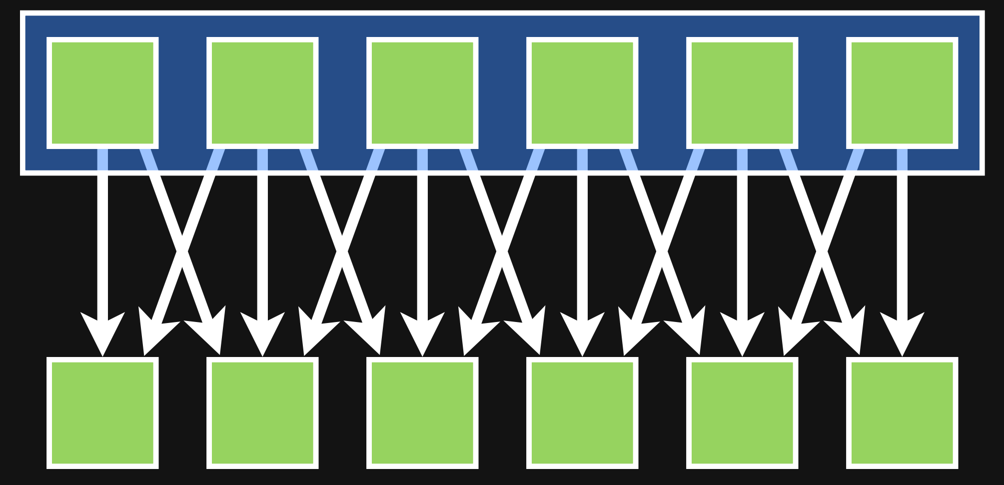
Tradeoff space modeled by **granularity of interleaving**



Tradeoff space modeled by **granularity of interleaving**

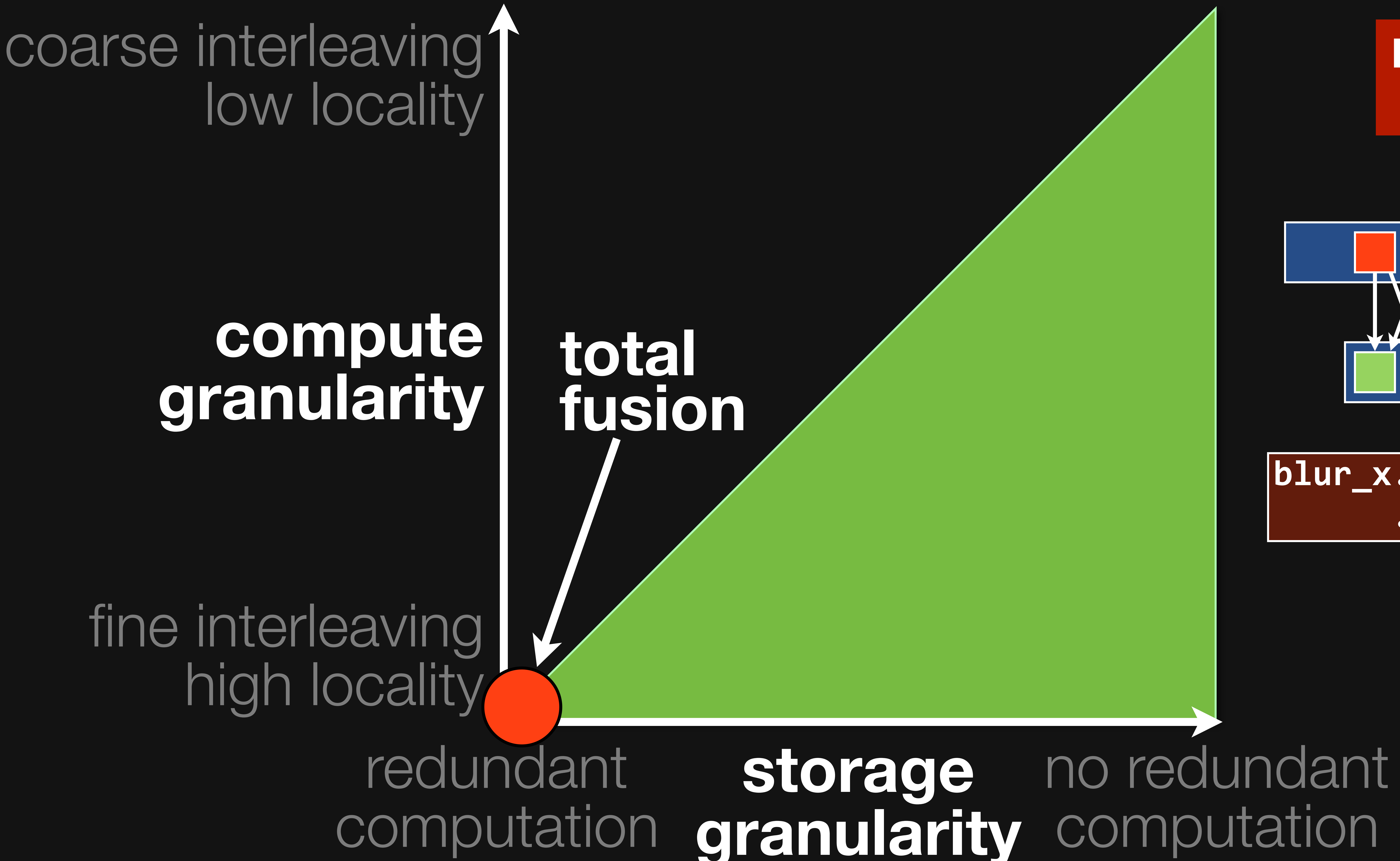


breadth-first execution

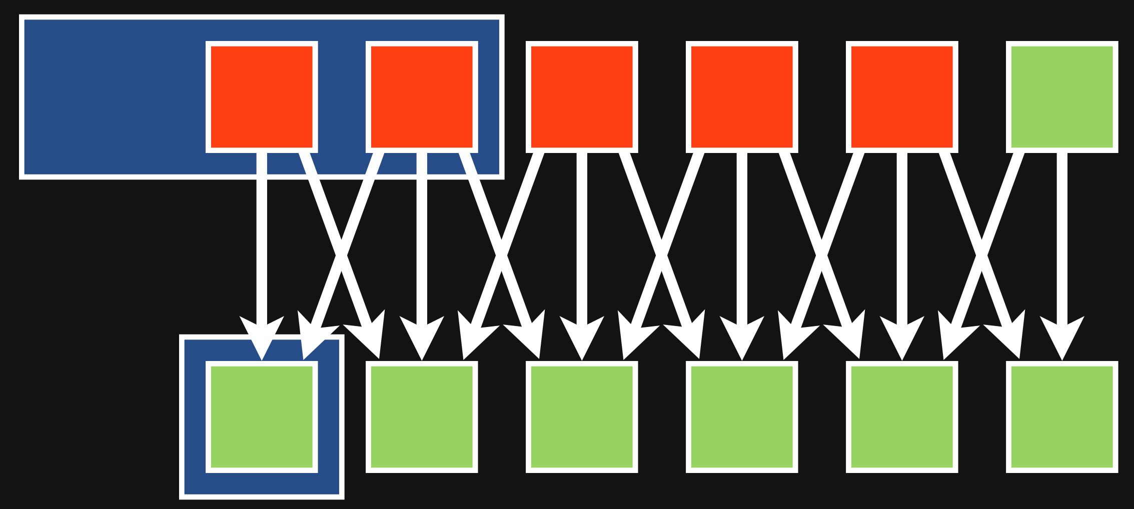


```
blur_x.compute_at(root)
      .store_at(root)
```

Tradeoff space modeled by **granularity of interleaving**

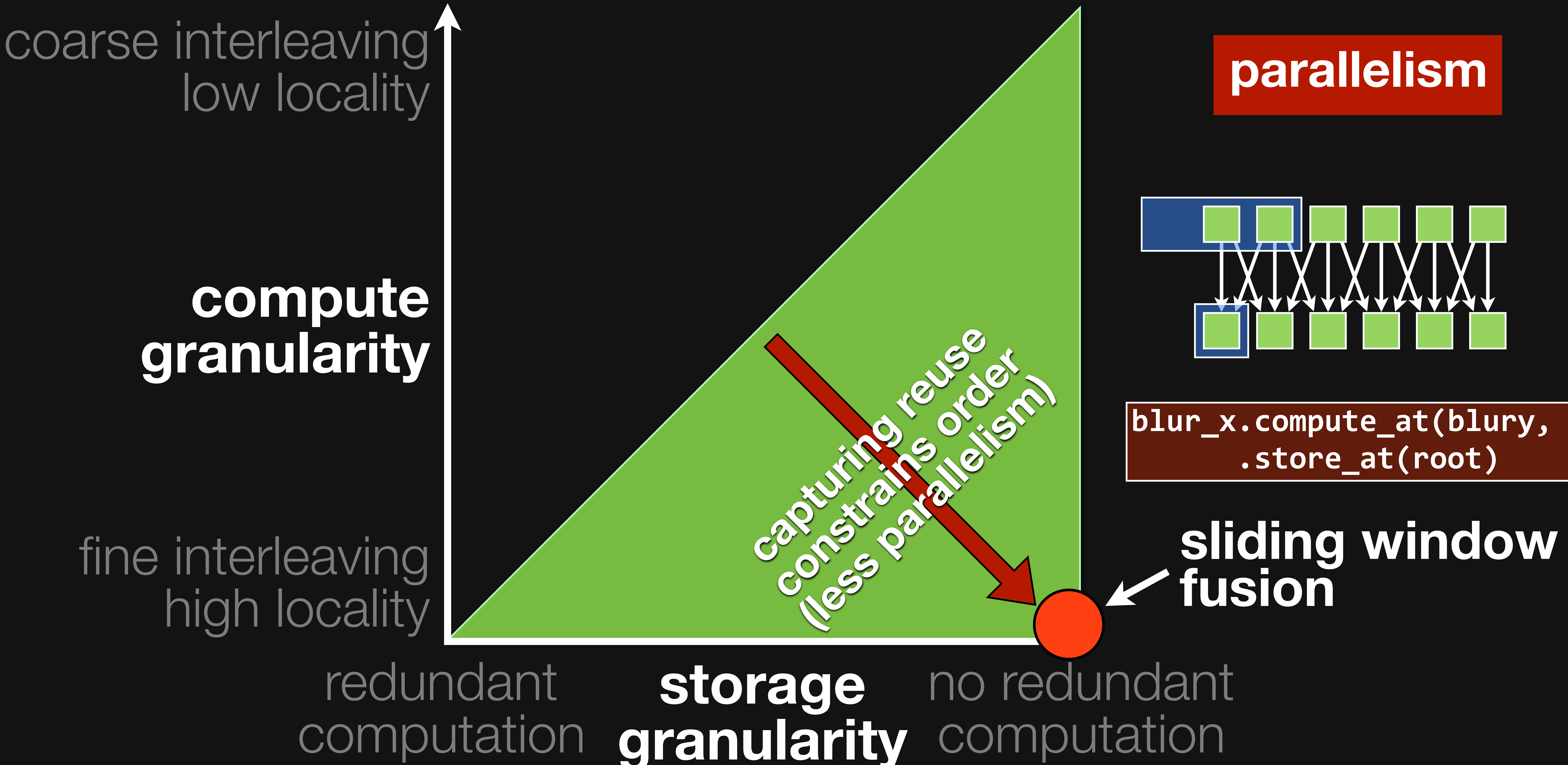


redundant work



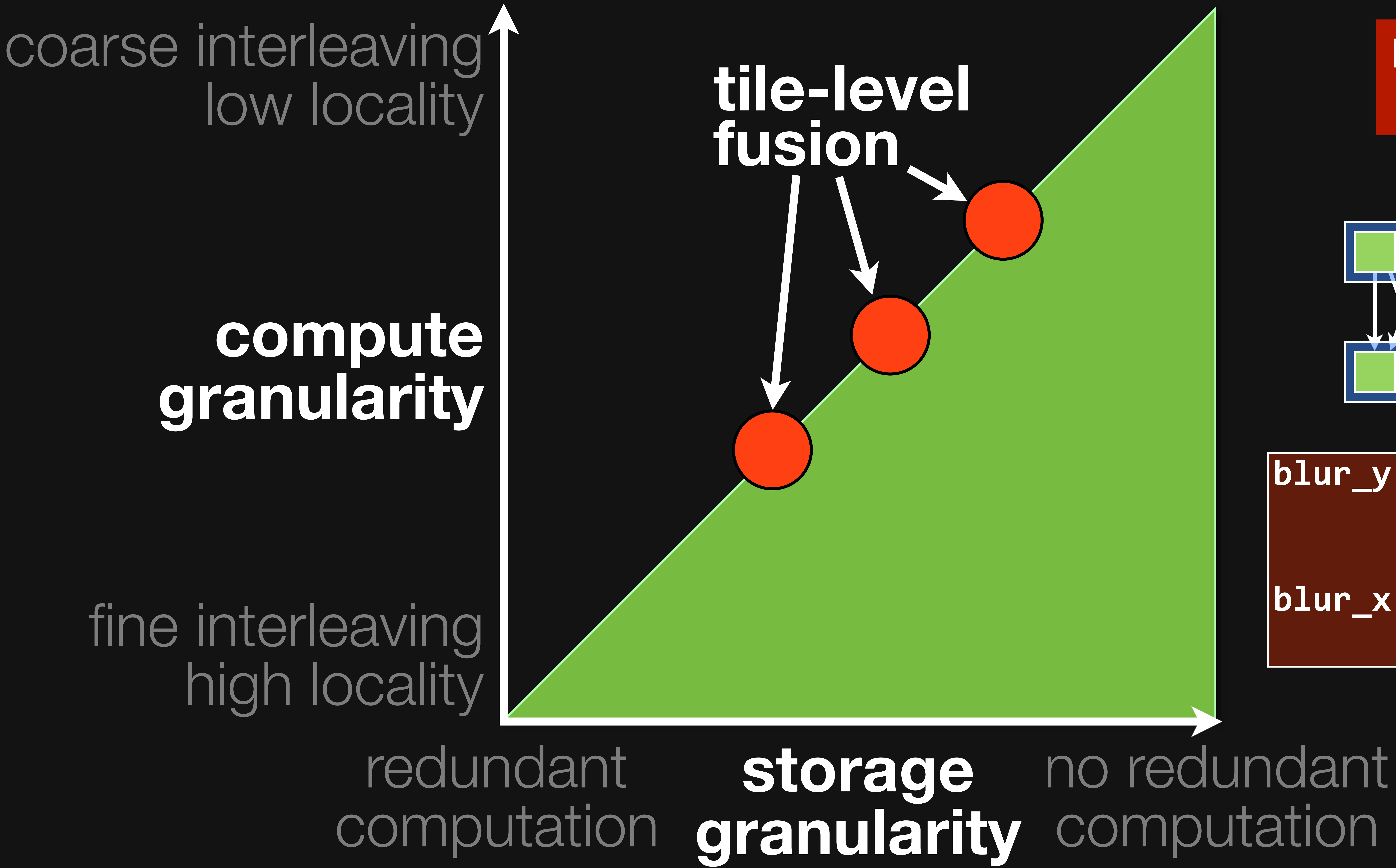
```
blur_x.compute_at(blury, x)
      .store_at(blury, x)
```

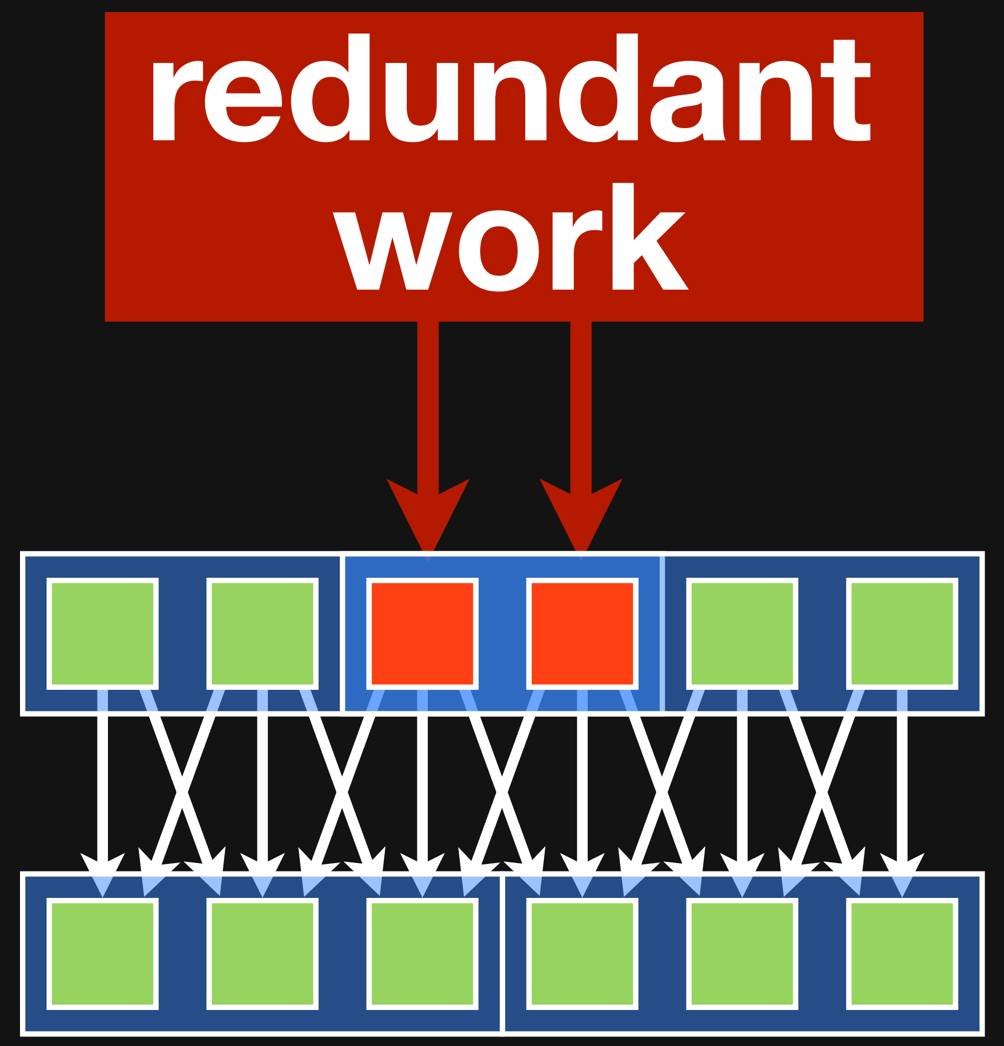
Tradeoff space modeled by **granularity of interleaving**




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

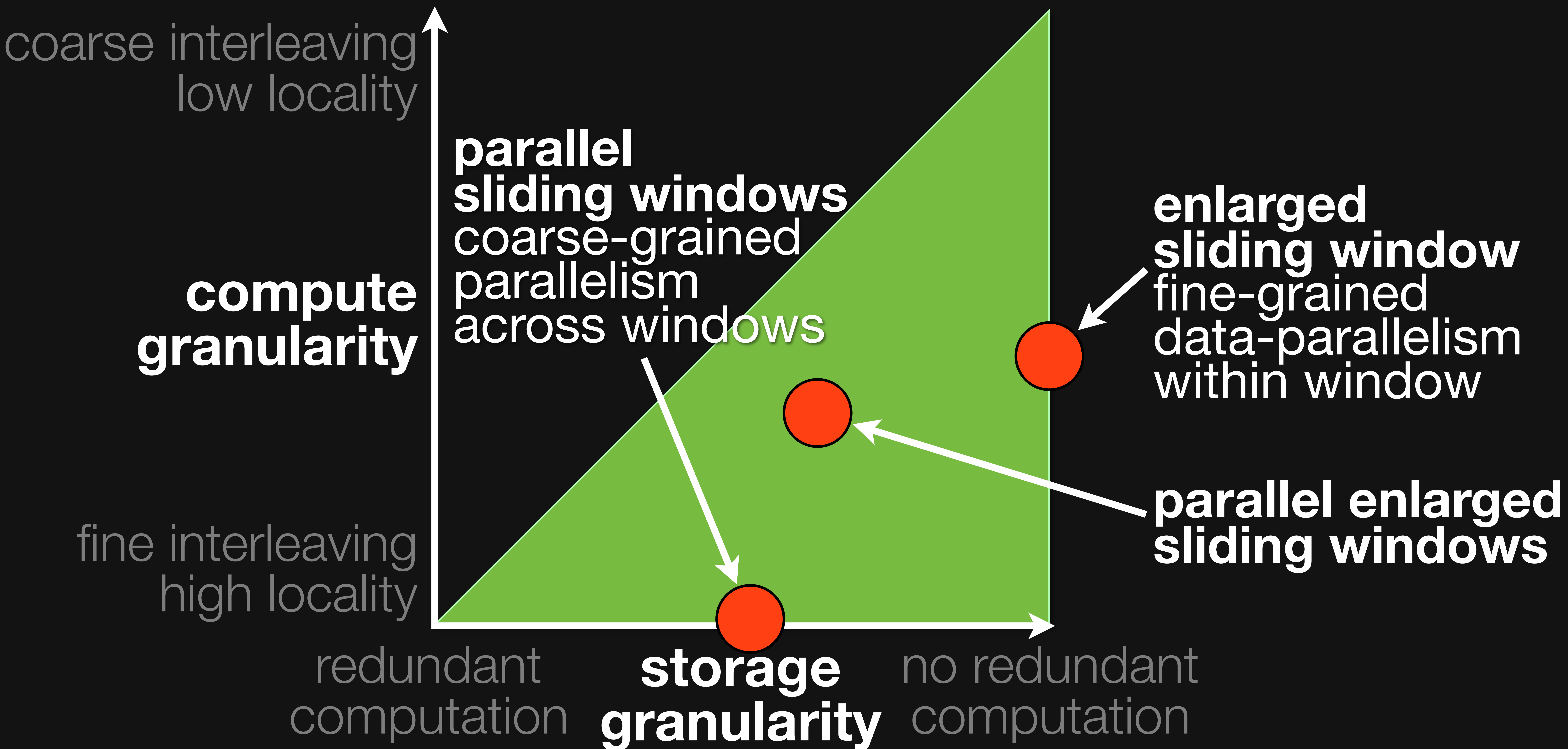
Tradeoff space modeled by **granularity of interleaving**






```
blur_y.tile(xo, yo,
            xi, yi,
            W,  H)
blur_x.compute_at(blury, xo)
      .store_at(blury, xo)
```

Tradeoff space modeled by **granularity of interleaving**



Schedule primitives **compose** to create many organizations

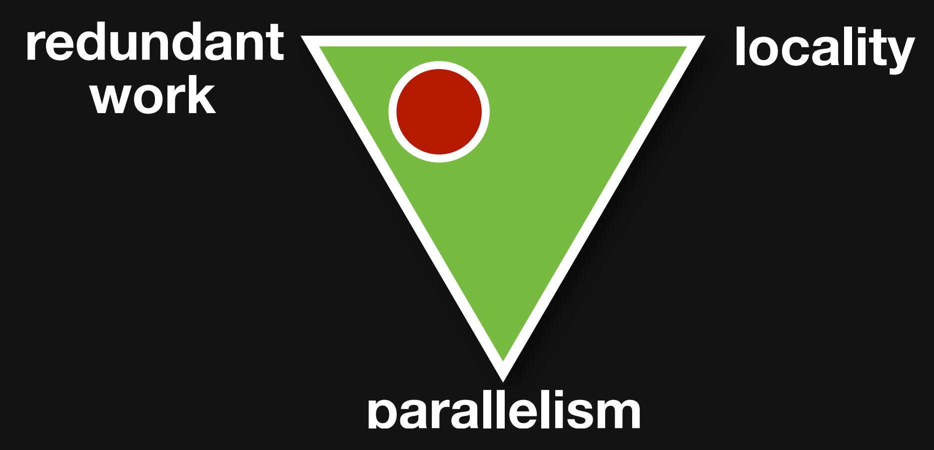

```
blur_x.compute_at(blury, y)
```

Schedule primitives **compose** to create many organizations

```
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);
}}}}}
```

```
blur_x.compute_at(blury, x)
      .vectorize(x, 4)

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



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;
}
```

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);
}}}}}
```

More language features
beyond the scope of this talk

Computed, data-dependent reads (gather)

```
f(x) = g(floor(2.3*in(x)))
```

Computed, data-dependent *writes* (scatter)

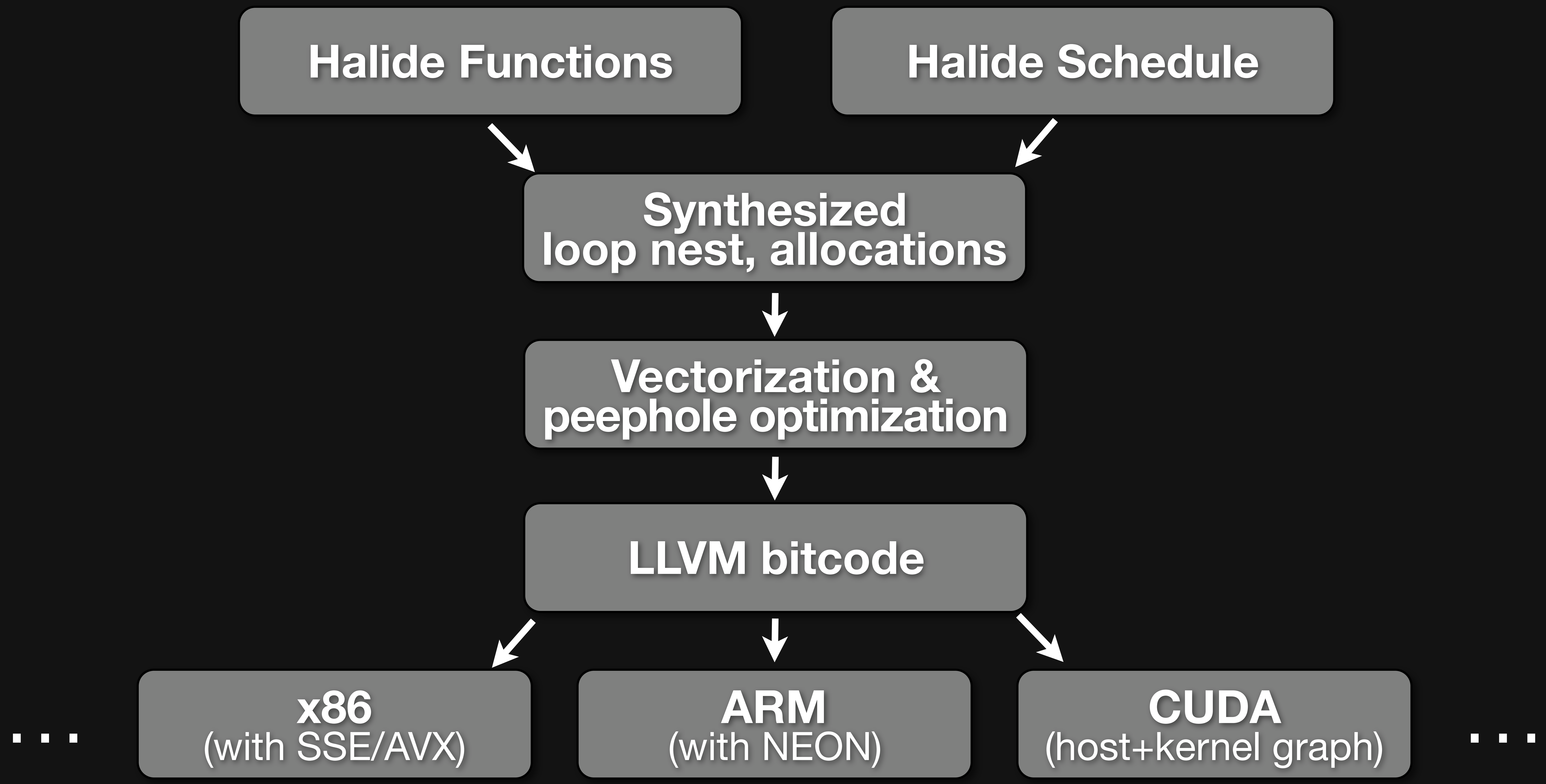
```
f(g(floor(2.3*in(x)))) = in(x)
```

Recursive functions (IIR convolution, scan)

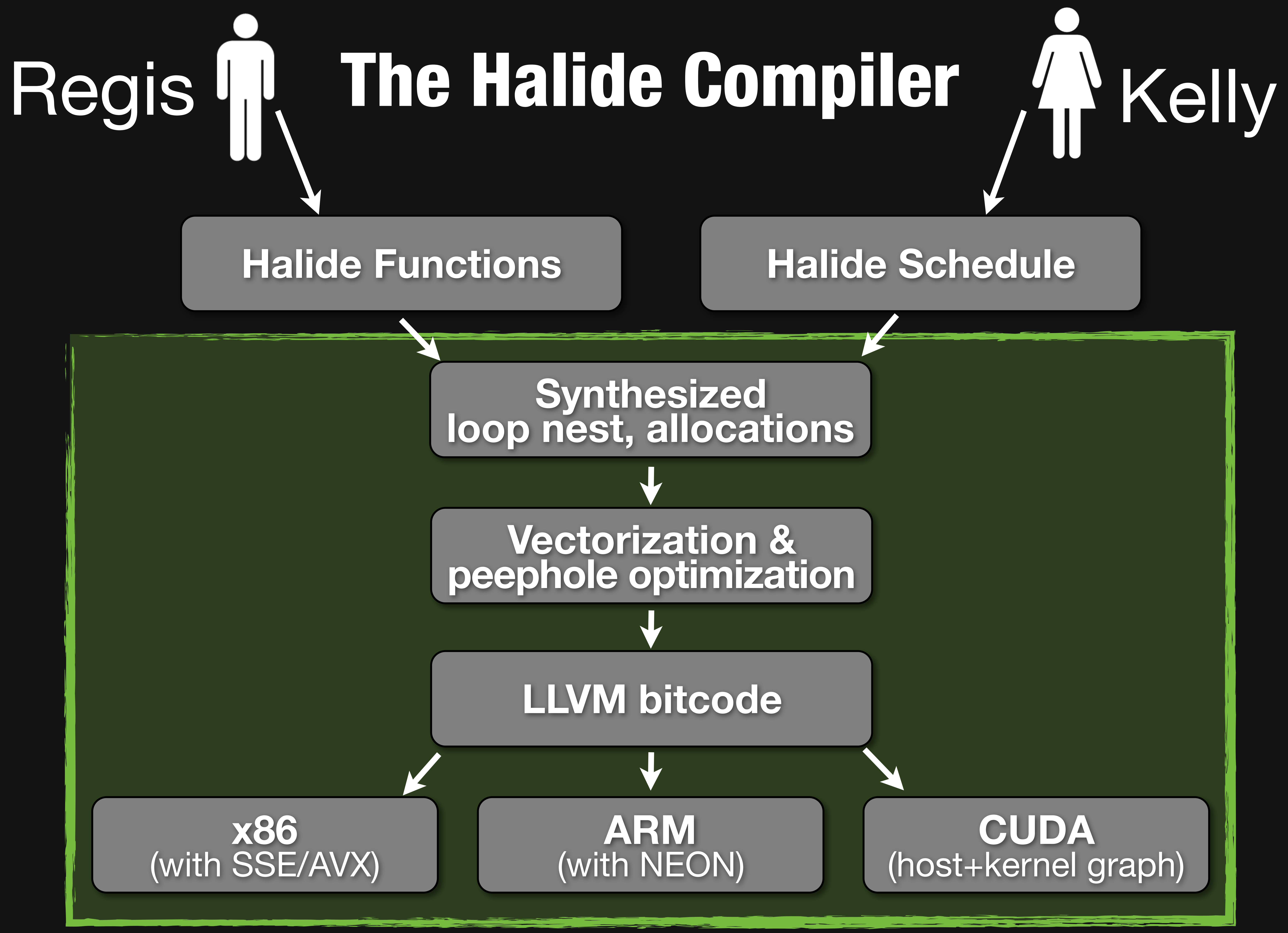
```
cdf(i) = cdf(i-1) + pdf(i)
```

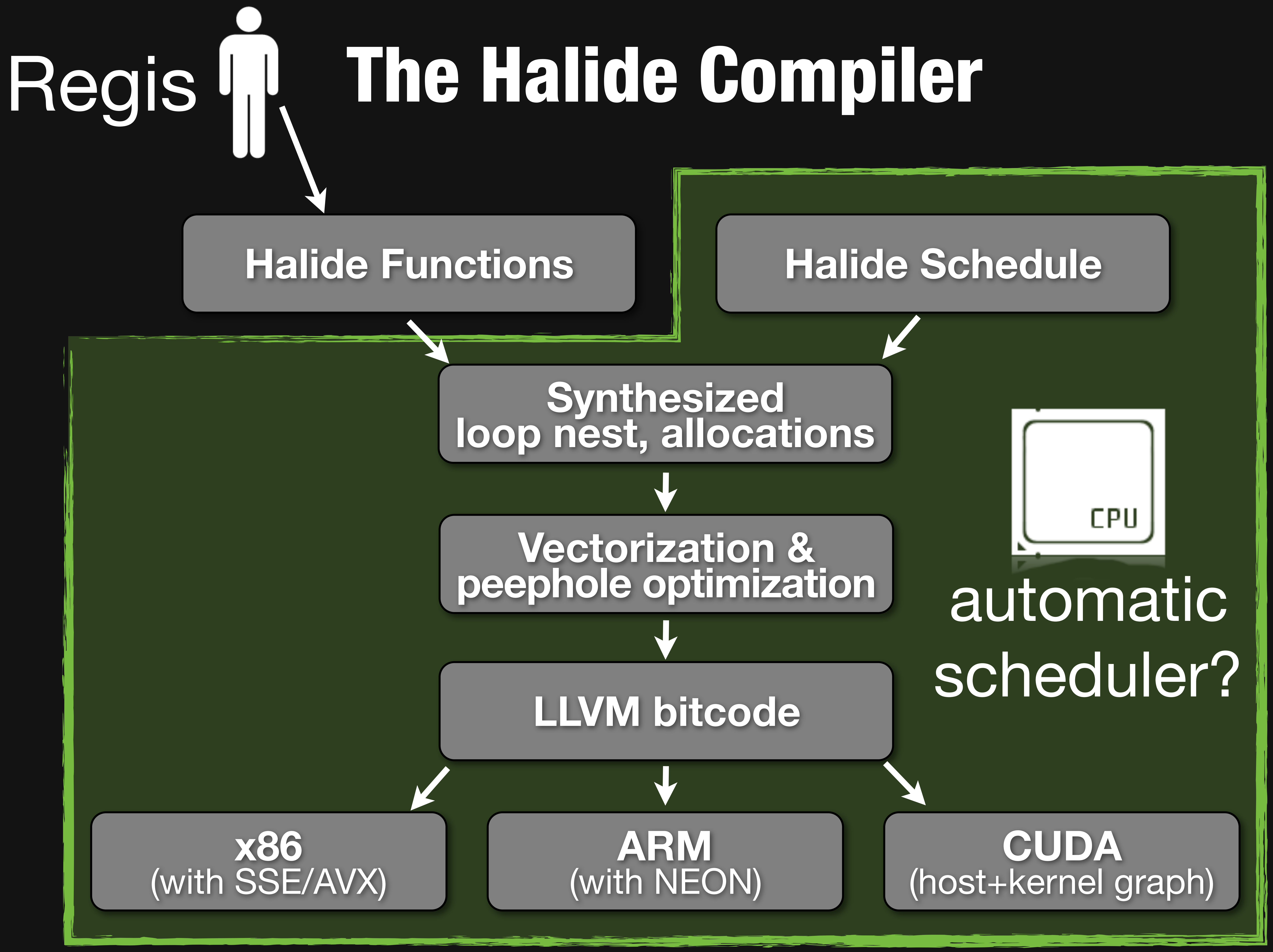
Reductions
histogram, etc.

The Halide Compiler



The Halide Compiler




Regis
The Halide Compiler
Halide Functions
Halide Schedule
Synthesized
loop nest, allocations
Vectorization &
peephole optimization
LLVM bitcode
x86
(with SSE/AVX)
ARM
(with NEON)
CUDA
(host+kernel graph)
CPU
automatic
scheduler?

How can we automatically find good organizations?

We need:

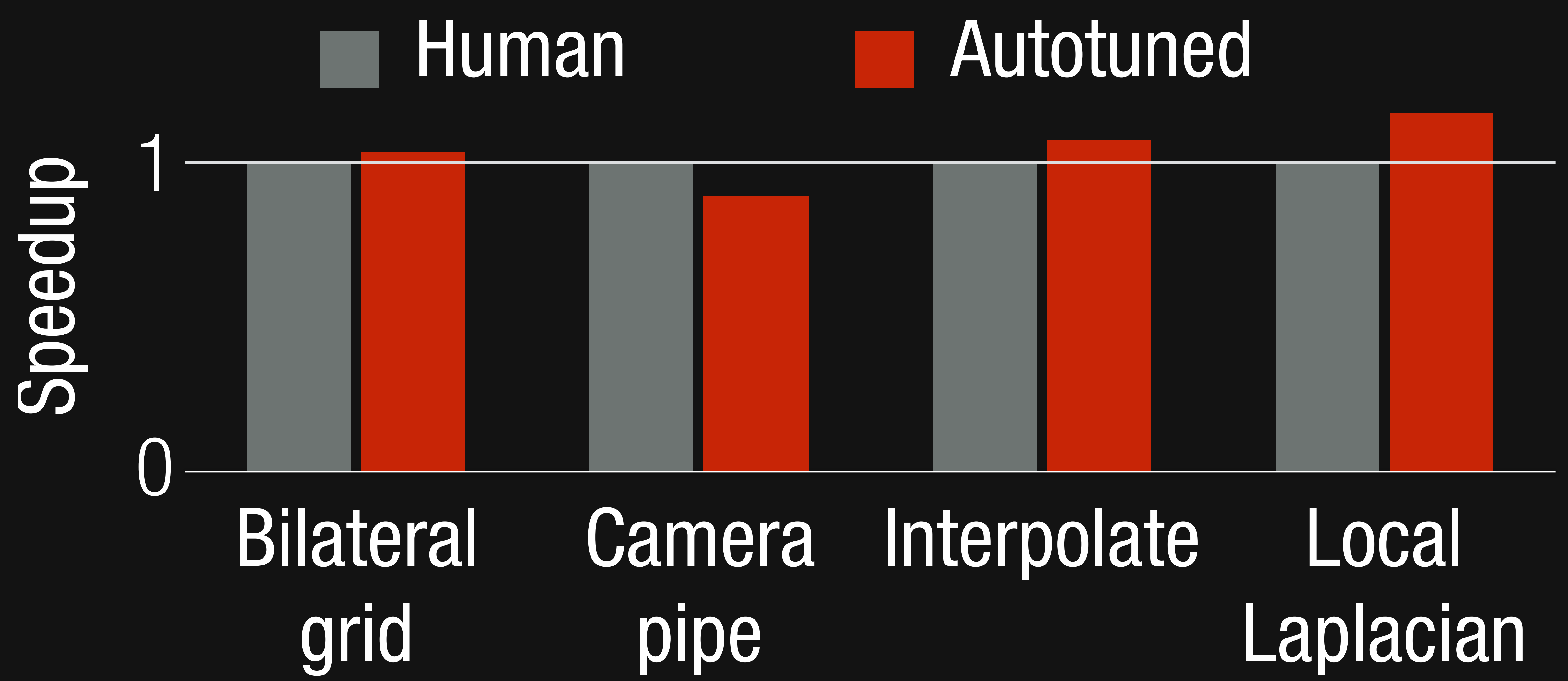
1. A **model** of the organization space
2. An **objective function**
3. An **algorithm** to find good organizations in the space

Autotuning: stochastic search over organizations

Model: **Halide schedules**

Objective: **measured benchmark runtime**

Algorithm: **hybrid heuristic search**



[Halide, PLDI 2013]

[PetaBricks, ASPLOS 2013]

[OpenTuner, PACT 2014]

Autotuning: stochastic search over organizations

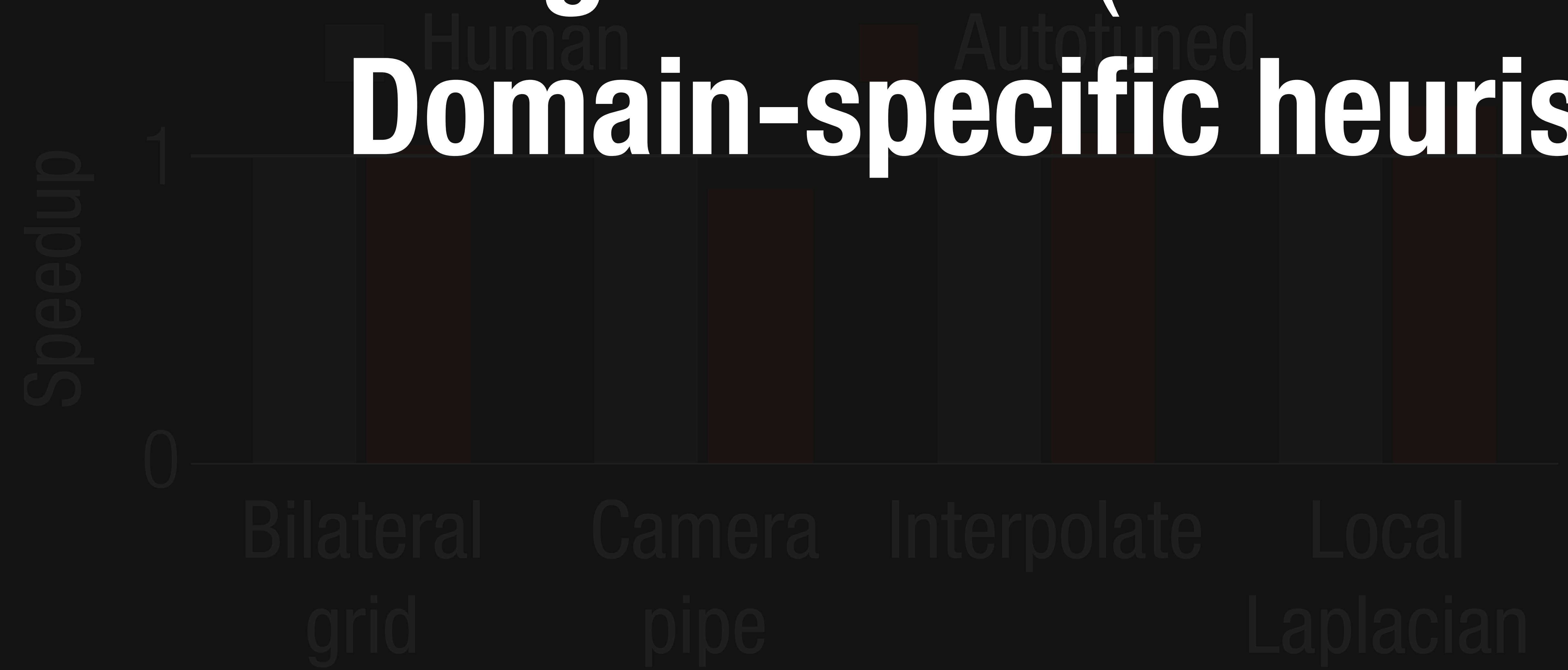
Model: **valid Halide schedules**

Requires:

Test harness + data (not just program code)

Long runtime (hours or days)

Domain-specific heuristics



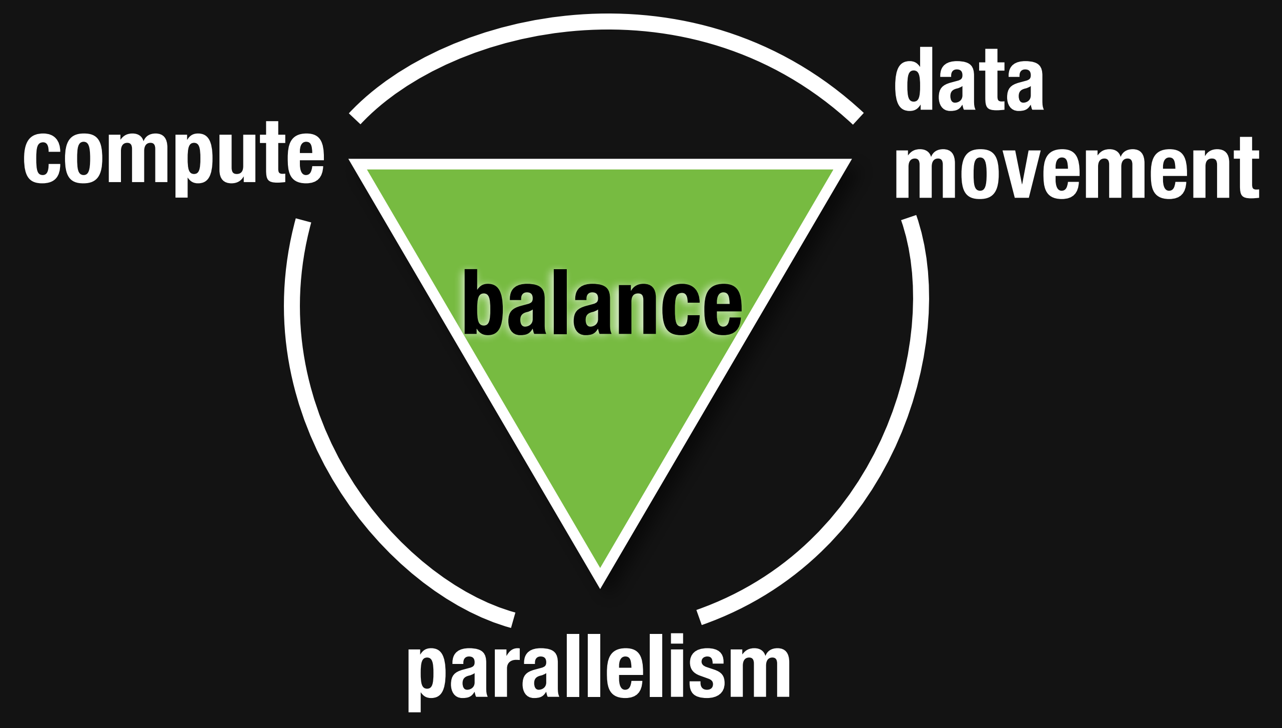
[Halide, PLDI 2013]

[PetaBricks, ASPLOS 2013]

[OpenTuner, PACT 2014]

Direct heuristic scheduling [SIGGRAPH 2016]

Model: **restricted Halide schedules**



Objective: **simple cost model**

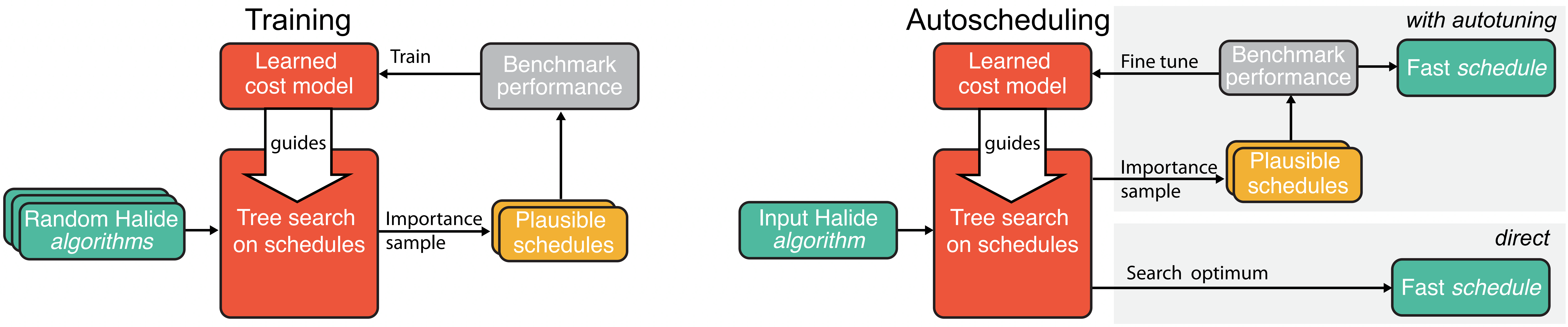
cost(alg, org)

Algorithm: **greedy clustering,**

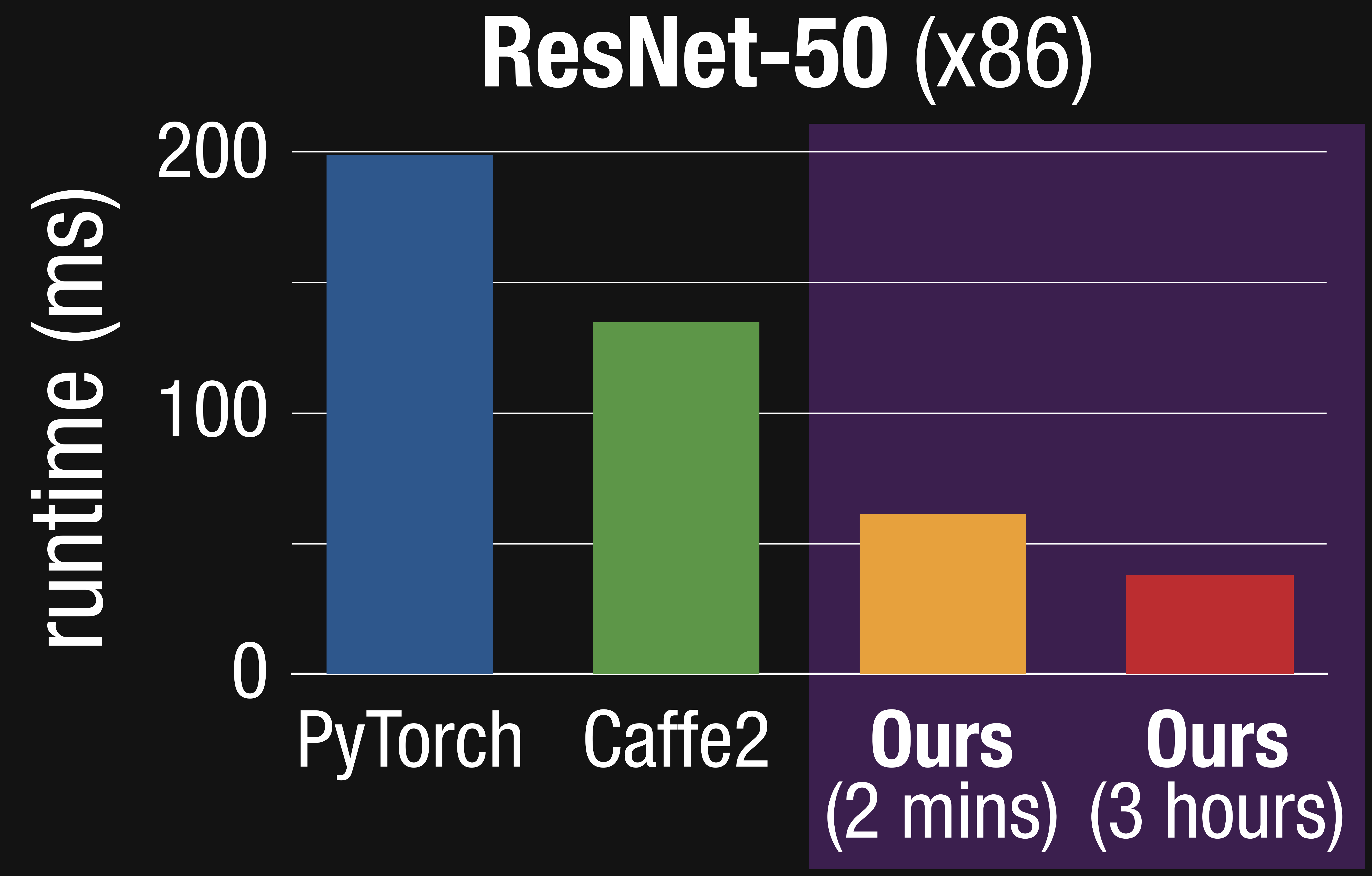
heuristic within each group

Good schedules in **seconds**

Scalable to very large programs

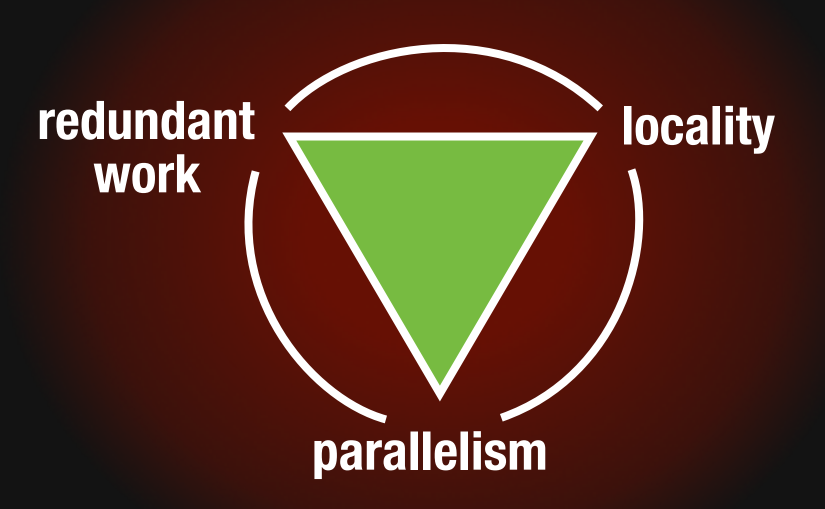

Training
Learned cost model
Train
Benchmark performance
guides
Random Halide algorithms
Tree search on schedules
Importance sample
Plausible schedules
Autoscheduling
with autotuning
Learned cost model
Fine tune
Benchmark performance
Fast schedule
guides
Importance sample
Plausible schedules
Input Halide algorithm
Tree search on schedules
direct
Search optimum
Fast schedule

Superhuman autoscheduling [in review]



The Halide language

Decouples **algorithm** from **organization**
structured as a scheduling language to navigate fundamental tradeoffs.



Simpler programs

Faster than hand-tuned code

Scalable across architectures

Real-world adoption

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

Adobe

Google

> 1800 pipelines
10s of kLOC in production

Microsoft Cognitive Toolkit

aka.ms/CognitiveToolkit

Microsoft

YouTube

Tensor Comprehensions

Qualcomm snapdragon

intel