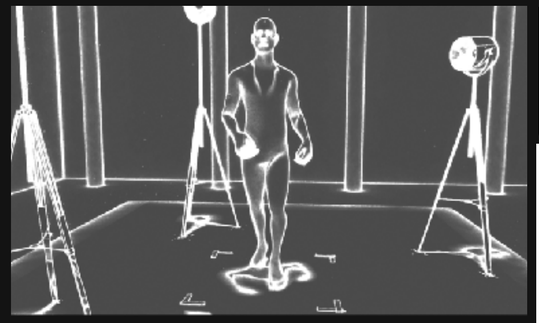
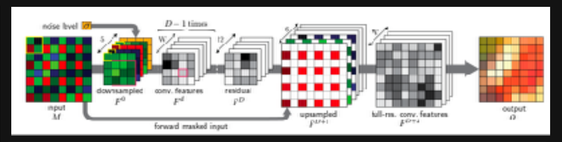
Halide: Decoupling Algorithms from Schedules for High Performance Image Processing

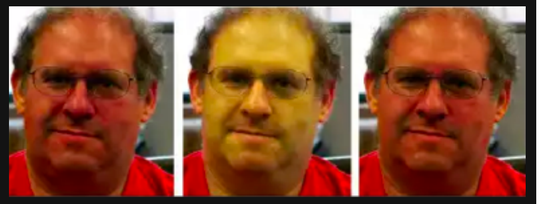
Frédo Durand
MIT & INRIA

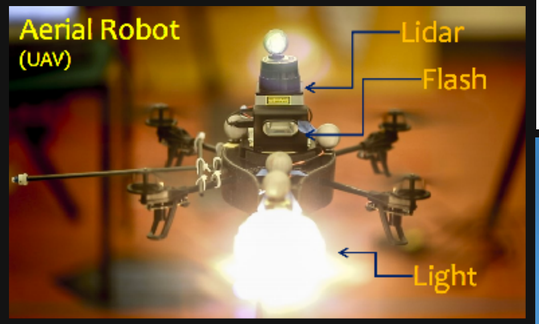
Our research in graphics, vision & photography

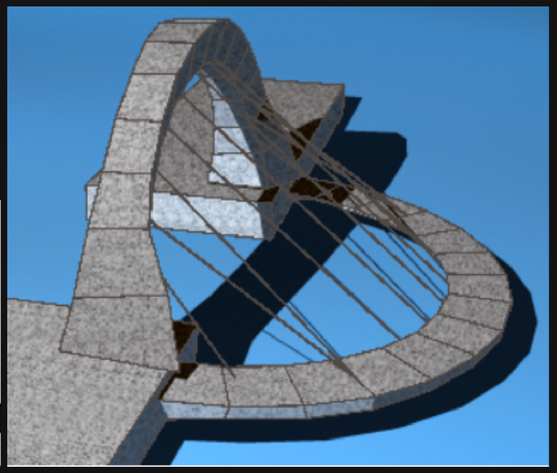
- Rendering
- Computational photography
- Revealing the invisible
- Graphics & learning
- HCI for visual authoring
- Architecture
- Systems for visual computing











Visiting

At INRIA until the end of July

Visiting Graphdeco team (Drettakis & Bousseau)

Drop by Y102 or email fredo@mit.edu

Halide

a language and compiler for image processing

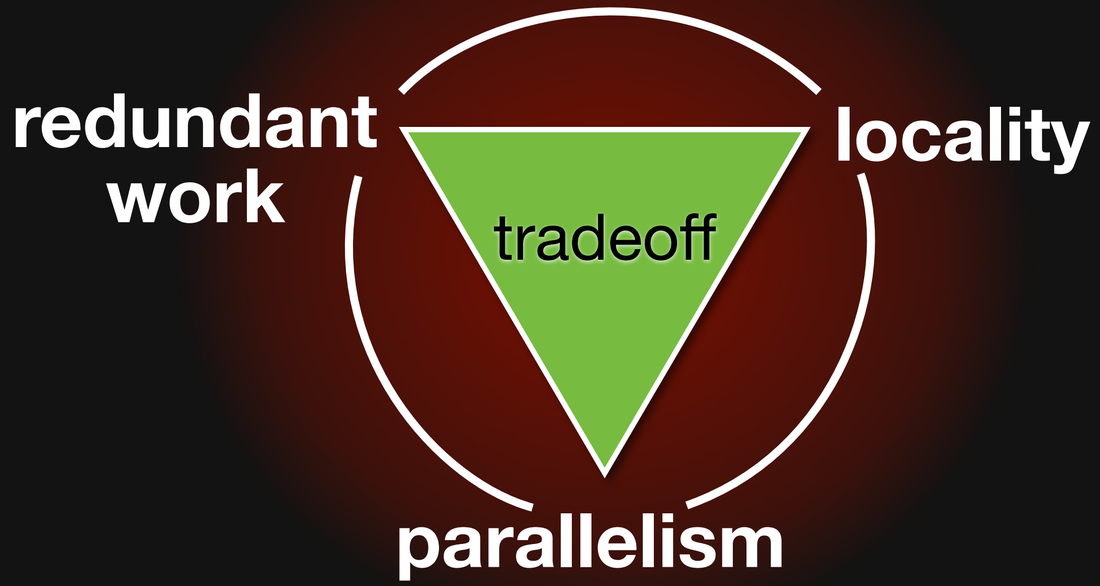
[SIGGRAPH 2012, PLDI 2013]

Jonathan *Ragan-Kelley, Andrew Adams, et al.*

Algorithm

Organization of computation

Hardware



Credit

Jonathan Ragan-Kelley

Andrew Adams

Also: Dillon Sharlet, Saman Amarasinghe, Sylvain Paris, Connelly Barnes, Ravi Mullapudi, Marc Levoy, Patricia Suriana, Shoaib Kamil, Gaurav Chaurasia, George Drettakis

Some work I wasn’t involved in

Most slides by Jonathan

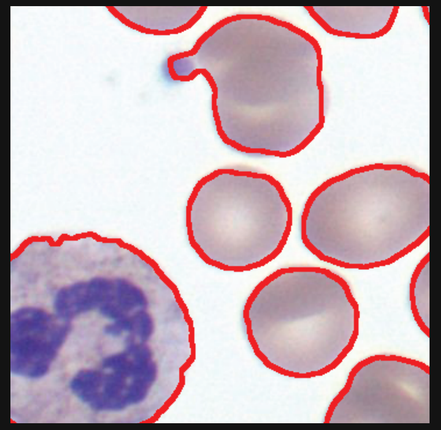
With a bit of portable Halide code you get

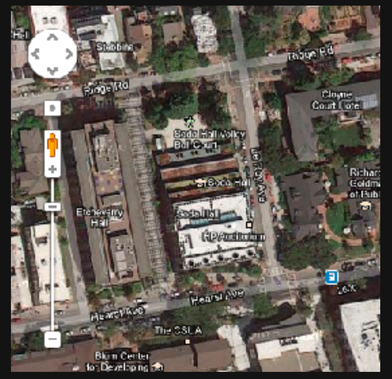
faster matrix multiply than Eigen
faster Gaussian blur than Intel Performance Primitives
faster Fourier transform than FFTW
faster Local Laplacian than Adobe's

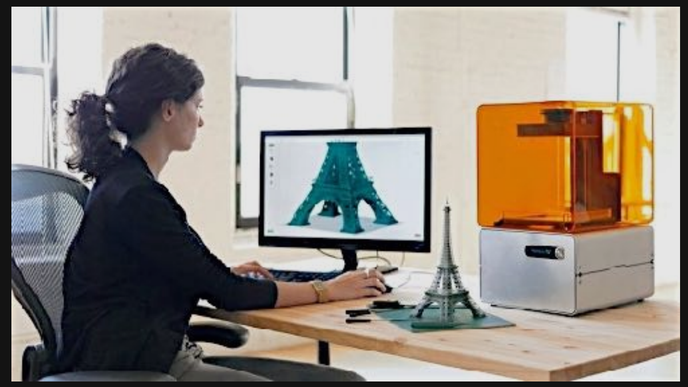
Plus

it runs on device and in the cloud
is supported on Linux, Windows, OSX, iOS, Android
compiles to x86, ARM, MIPS, native client, OpenCL, OpenGL, CUDA, JavaScript, RenderScript (ISPs soon)

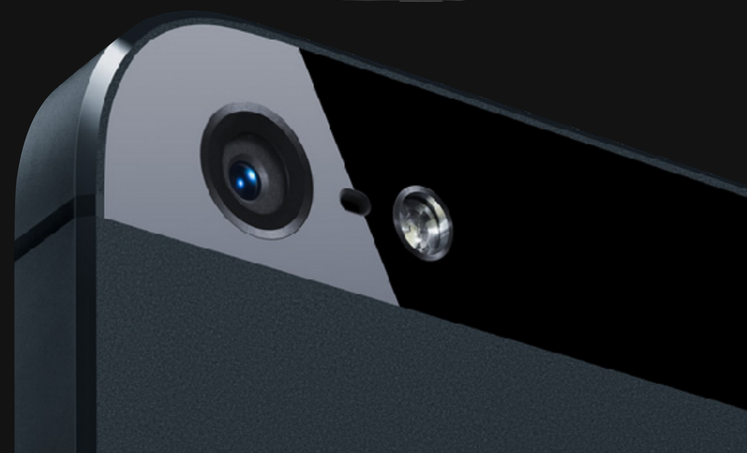
slide by Marc Levoy







Visual computing and spatiotemporal data are **everywhere**



Visual computing demands orders of magnitude more performance

Rendering:
insatiable demand for computation

Modern game

2 Mpixels

1 Mpolys

10 ms/frame

10 ms

5 hrs

Tintin, Avatar

8 Mpixels

5 Gpolys

5 hrs/frame

6 orders of magnitude more computation

images by Valve, Weta

The biggest data is visual

YouTube: 400 hrs uploaded / min

[Brewer et al. 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

transmit

LTE radio

50 Mbit/sec

1 W

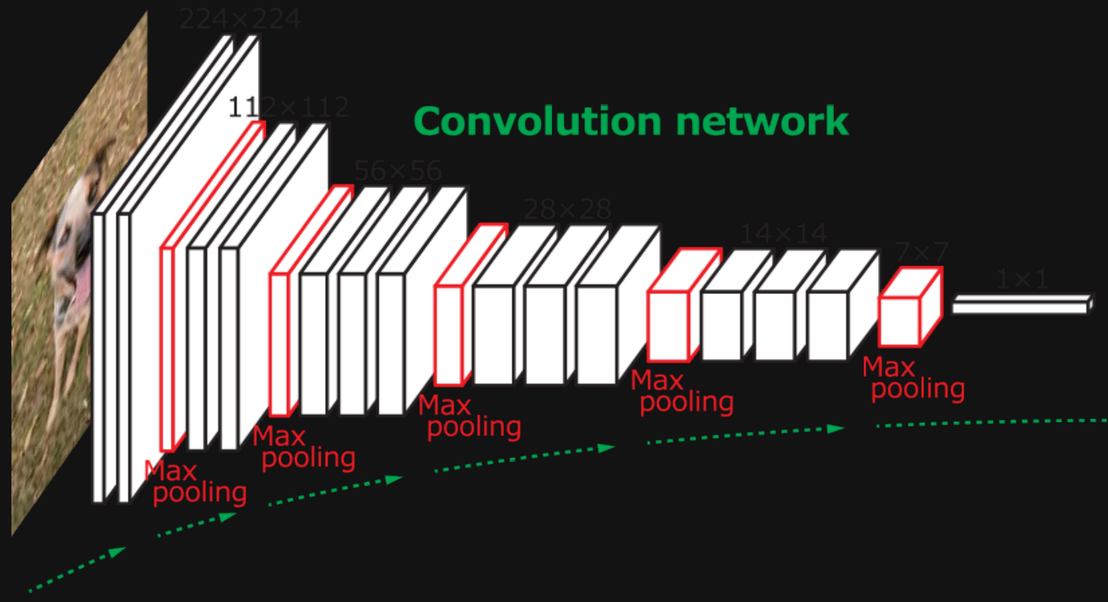
~1 J/frame

[Wu et al. 2012]

transmission power costs 1,000x capture

Visual data analysis is expensive

One object recognition neural net:
250 Watt GPU → 0.05 megapixels at video rate



(Caffe + cuDNN on K40c running VGG16)

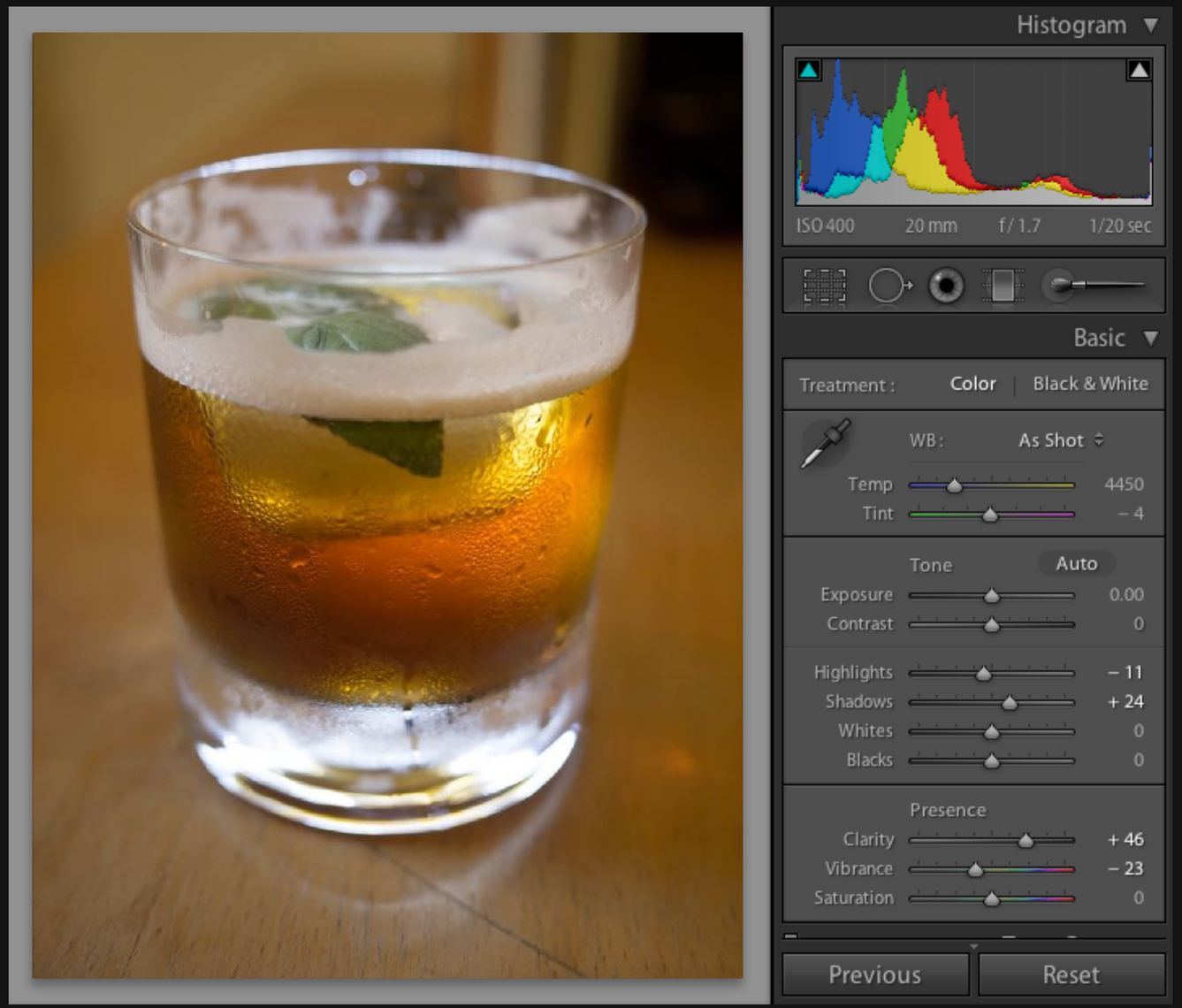
[Simonyan & Zisserman 2015]
image: Noh et al. 2015

Your data-intensive problem here…

Writing high-performance code is hard

Reference:
300 lines C++

Adobe: 1500 lines
3 months of work
10x faster



How can we increase performance and efficiency?

Parallelism

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

Locality

Data should move as little as possible.

[REVISE Moore's law context - if we want more perf/scale, we have two choices: use more HW (Moore growth or scale out), or more efficiently use same resources?]
The good news is: Moore's Law is continuing to give us exponentially more computational resources, at least

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

Example: two-stage box blur

Horizontal blur followed by vertical blur

```
for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;

for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

Output of 1st blur can’t fit in cache

First pixels gone when needed for 2nd blur

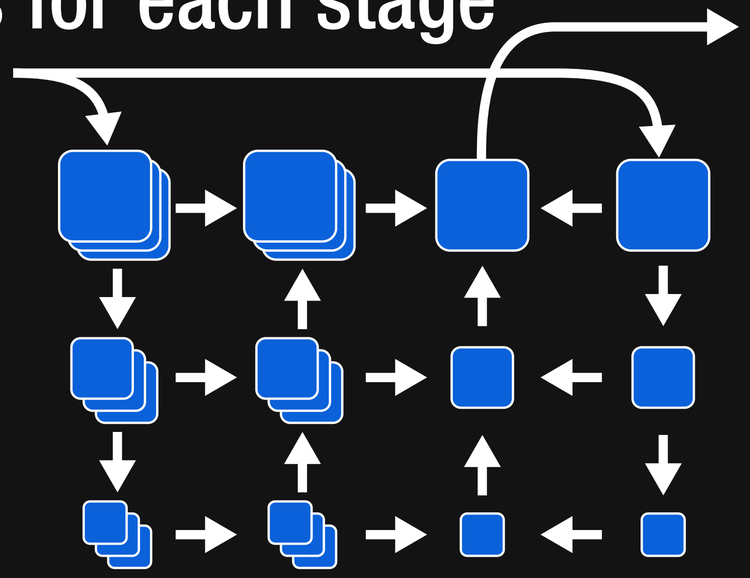
Image processing is wide and deep

Wide: data (images) are large

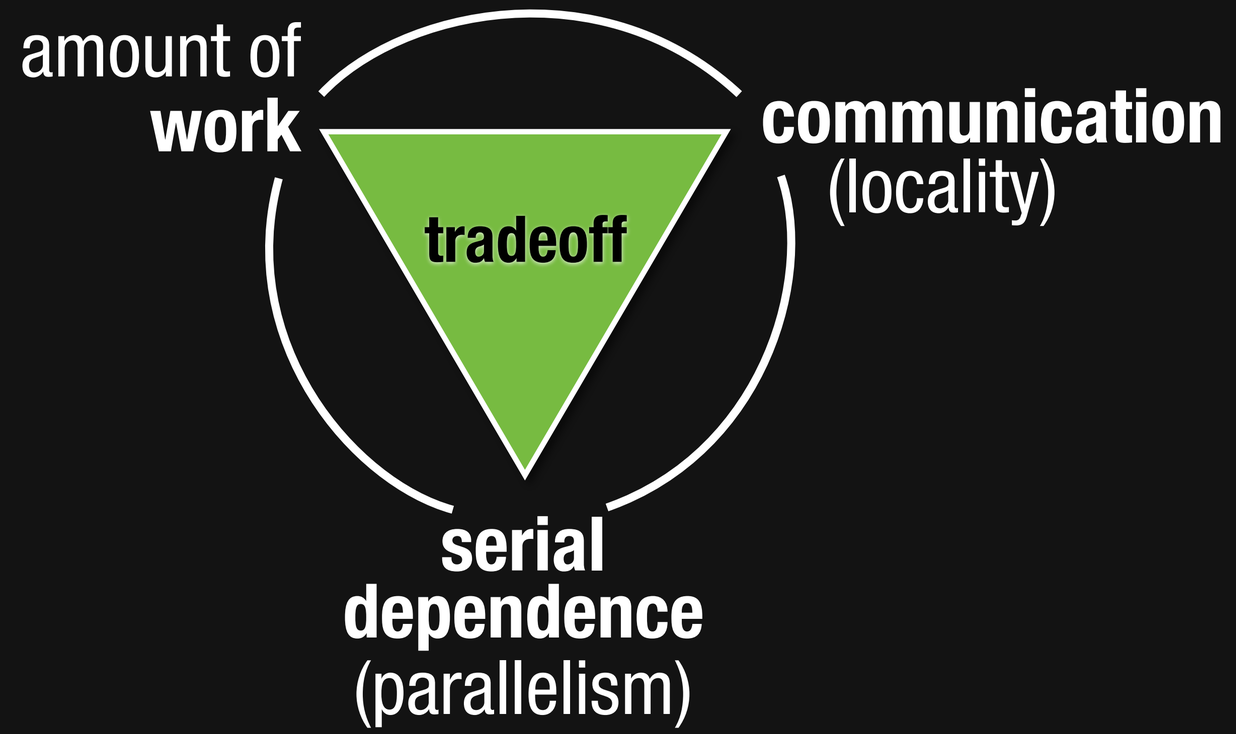
don’t fit in cache

Deep: pipelines or graphs contains many stages

incur cache misses for each stage



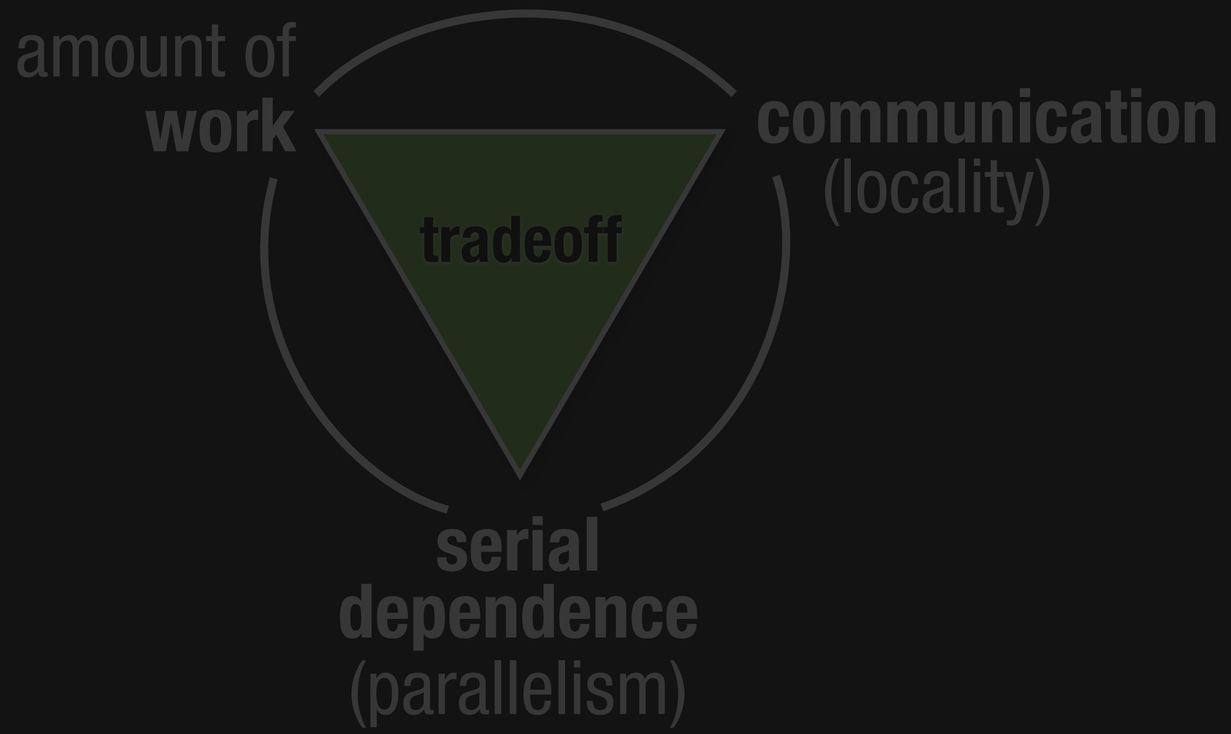
Message #1: Performance is a tension between three issues



Where does performance come from?

Program

Hardware



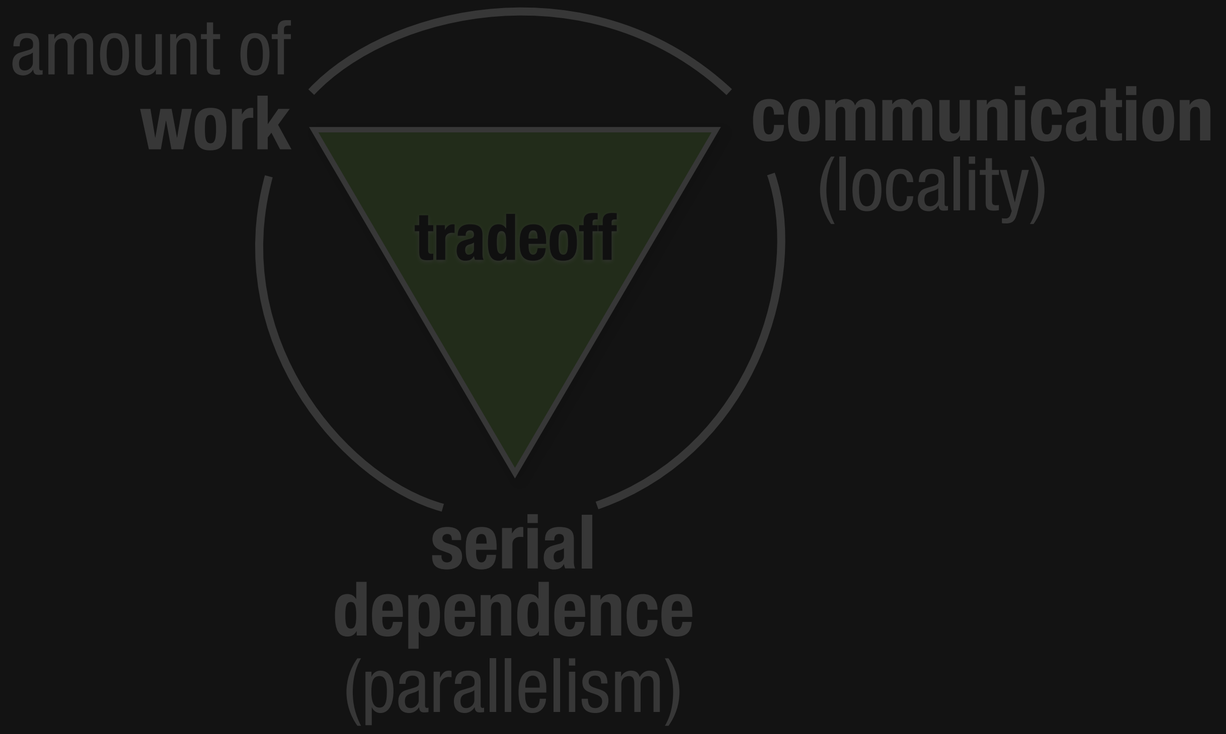
Message #2: organization of computation
is a first-class issue

Program:

Algorithm

Organization of computation

Hardware



Algorithm vs. Organization: 3x3 blur

```
for (int x = 0; x < input.width(); x++)
    for (int y = 0; y < input.height(); y++)
        blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
```

```
for (int x = 0; x < input.width(); x++)
    for (int y = 0; y < input.height(); y++)
        blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

Algorithm vs. Organization: 3x3 blur

```
for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;

for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

Same algorithm, different organization

One of them is 15x faster

Key challenge:
reorganize across two (or more) stages

```
for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;

for (int y = 0; y < input.height(); y++)
  for (int x = 0; x < input.width(); x++)
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

Example: fuse stages at tile granularity

```
for y_tile_index:
   for x_tile_index:
      for y_within_tile:
         for x_within_tile:
            tmp(y_within_tile, x_within_tile) =…
      for y_within_tile:
         for x_within_tile:
            out(y_tile_index*tile_height+y_within_tile,
               x_tile_index*tile_width+x_within_tile) =…
```

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

Traditional languages **conflate algorithm** & **organization**

```
void box_filter_3x3(const Image &in, Image &blurV) {
  __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      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(blurHPtr++, avg);
         inPtr += 8;
      }}
      blurHPtr = blurH;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blurV[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurHPtr+(2*256)/8);
          b = _mm_load_si128(blurHPtr+256/8);
          c = _mm_load_si128(blurHPtr++);
          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 &blurV) {
  Image blurH(in.width(), in.height());  // allocate blurH array

  for (int y = 0; y < in.height(); y++)
    for (int x = 0; x < in.width(); x++)
      blurH(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++)
      blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(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 &blurV) {
  __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      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(blurHPtr++, avg);
         inPtr += 8;
      }}
      blurHPtr = blurH;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blurV[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurHPtr+(2*256)/8);
          b = _mm_load_si128(blurHPtr+256/8);
          c = _mm_load_si128(blurHPtr++);
          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 &blurV) {
  __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      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(blurHPtr++, avg);
          inPtr += 8;
      }}
      blurHPtr = blurH;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blurV[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurHPtr+(2*256)/8);
          b = _mm_load_si128(blurHPtr+256/8);
          c = _mm_load_si128(blurHPtr++);
          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

Global reorganization breaks modularity

```
void box_filter_3x3(const Image &in, Image &blurV) {
  __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      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(blurHPtr++, avg);
         inPtr += 8;
      }}
      blurHPtr = blurH;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blurV[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurHPtr+(2*256)/8);
          b = _mm_load_si128(blurHPtr+256/8);
          c = _mm_load_si128(blurHPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

Shared outer loops

First blur

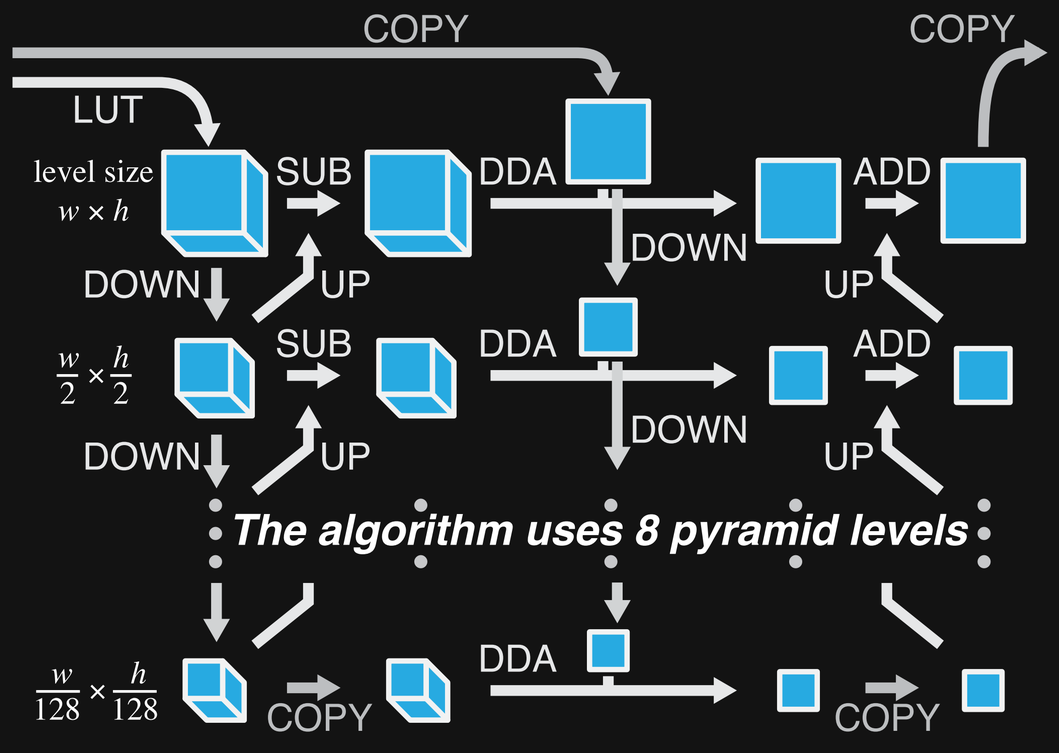
loop bounds depend on second blur (expansion by footprint)

Second blur

Global reorganization breaks modularity

Complex with many stages, full algebraic tree of filters

e.g. how deep do you fuse?
interdependency between stages

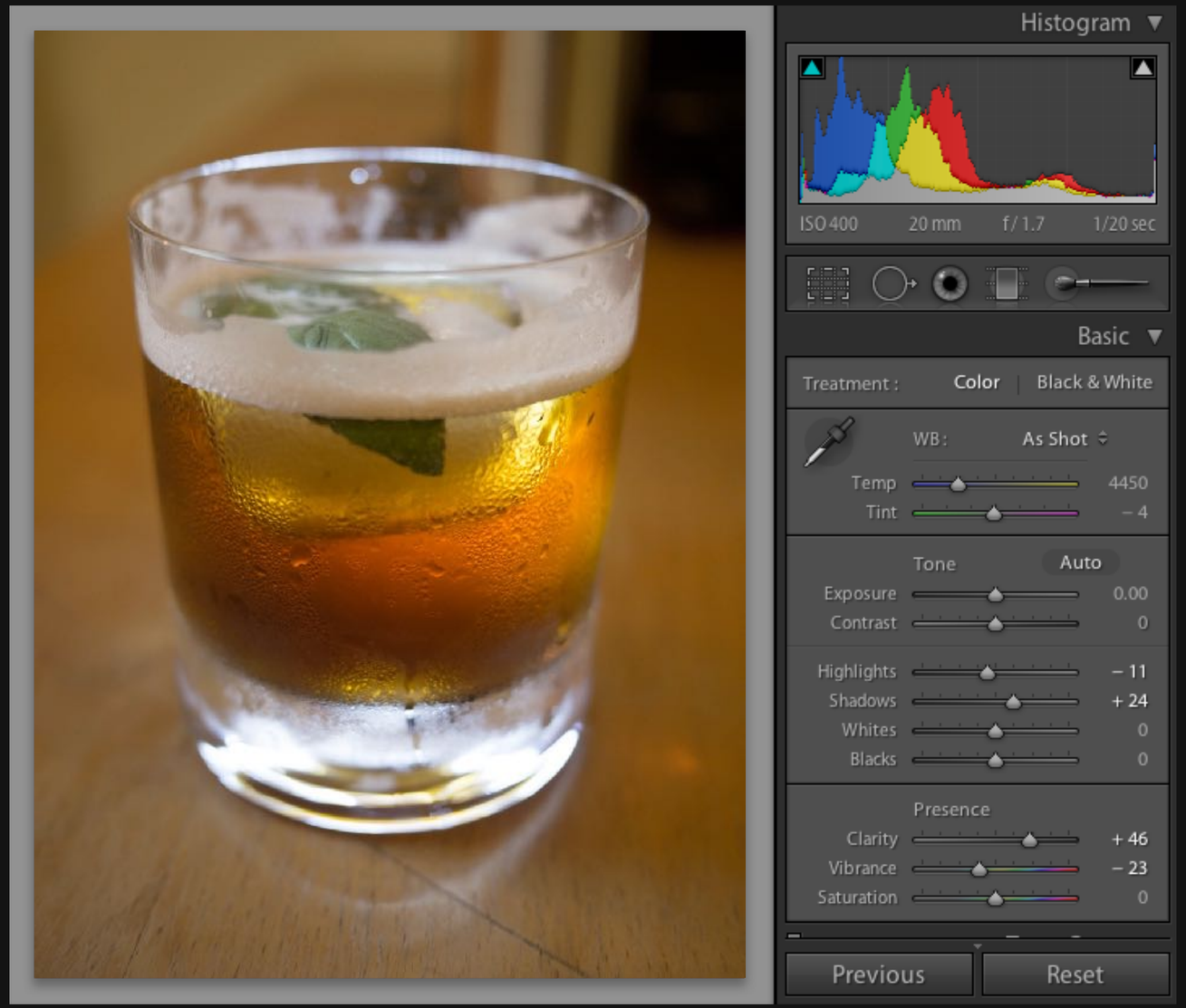


Reorganizing computation is painful

Reference:
300 lines C++

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

**Same algorithm,
Different organization**



(Re)organizing computation is hard

Optimizing parallelism, locality requires transforming program & data structure.

What transformations are *legal?*

What transformations are *beneficial?*

Implementing a tentative transformed version of a program requires tedious and error-prone manual effort

e.g. loop bounds, indexing, SIMD code

Libraries don’t solve this

e.g. BLAS, IPP, MKL, OpenCV, MATLAB

Optimized kernels compose into inefficient pipelines
(no fusion, bad cache coherence)

Halide's answer:
decouple algorithm from schedule

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

Related work*

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]
Affine partitioning [Lim & Lam 1999]

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

[Holzman 1988], [Shantzis 1994], [Eliot 2001]
PixelBender, CoreImage

**a tiny sample.*
Thousands have come before us.

Why a new language?

Analysis on legacy languages is hard/impossible

e.g. variable aliasing

We need to be able to analyze dependencies

to know what needs to be available to compute what
for loop bound and temp array size computation

Will also make code more concise

Embedded in C++ for easy integration

The algorithm defines pipelines as pure functions

Pipeline stages are functions from coordinates to values

Execution order and storage are unspecified
no explicit loops or arrays

3x3 blur as a Halide algorithm:

```
blurH(x, y) = (input(x-1, y) + input(x, y) + input(x+1, y))/3;
blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

Halide algorithm with declarations and call

```
Func blurH, blurV;
Var x, y, xi, yi;

blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

Image<float> output = blurV.realize(in.width(), in.height());
```

Halide is a purely Functional language

No side effect, no environments!

Define your algorithm as a pipeline of pure functions

Loops are NOT specified in the Halide algorithm

That’s the job of the schedule

Domain scope
of the programming model

All computation is over **regular grids**.

not Turing complete

Only **feed-forward pipelines**
Iterative computations are a (partial) escape hatch.

Iteration must have **bounded depth.**

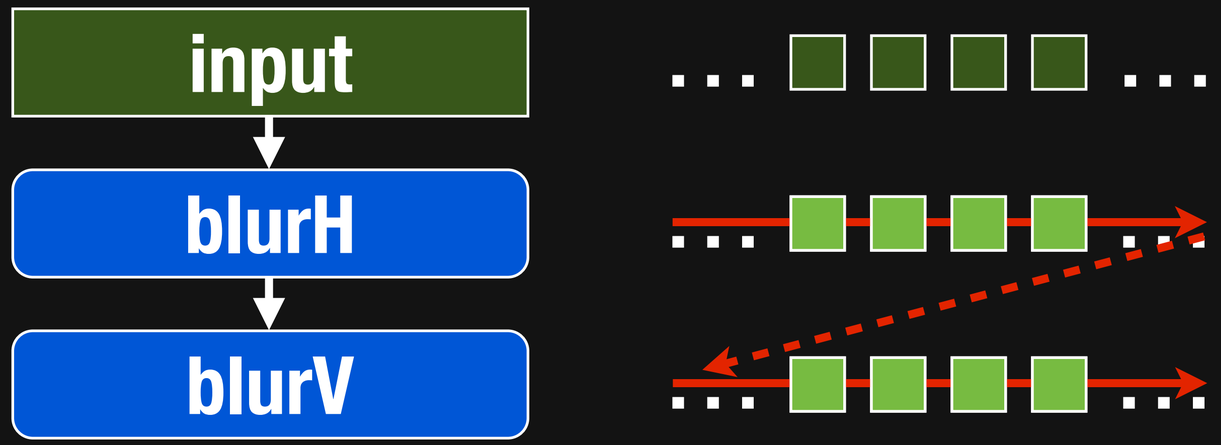
Dependence must be inferable.
User-defined clamping can impose tight bounds, when needed.

Long, heterogeneous pipelines.
Complex graphs, deeper than traditional stencil computations.

How can we organize this computation?

Organizing a data-parallel pipeline

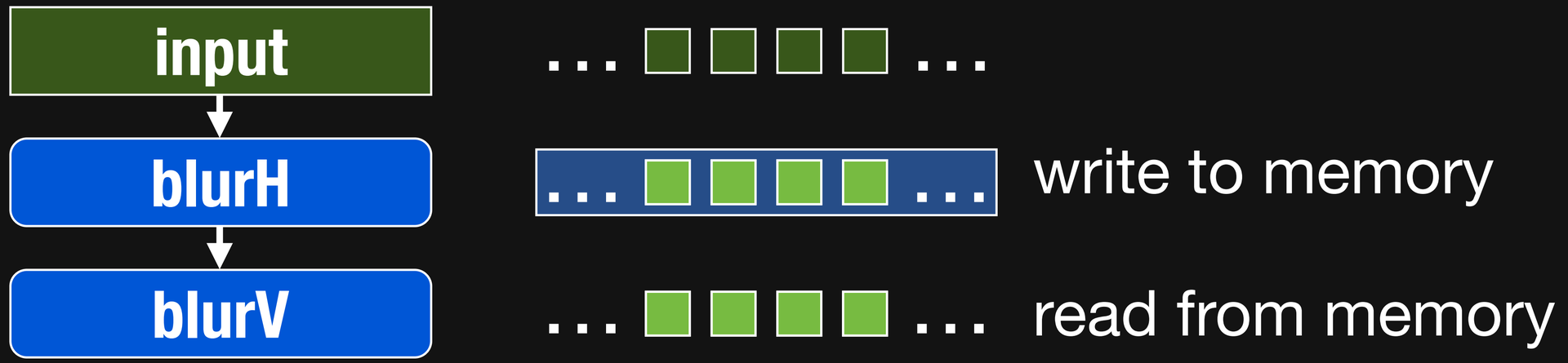
Simple loops execute **breadth-first** across stages

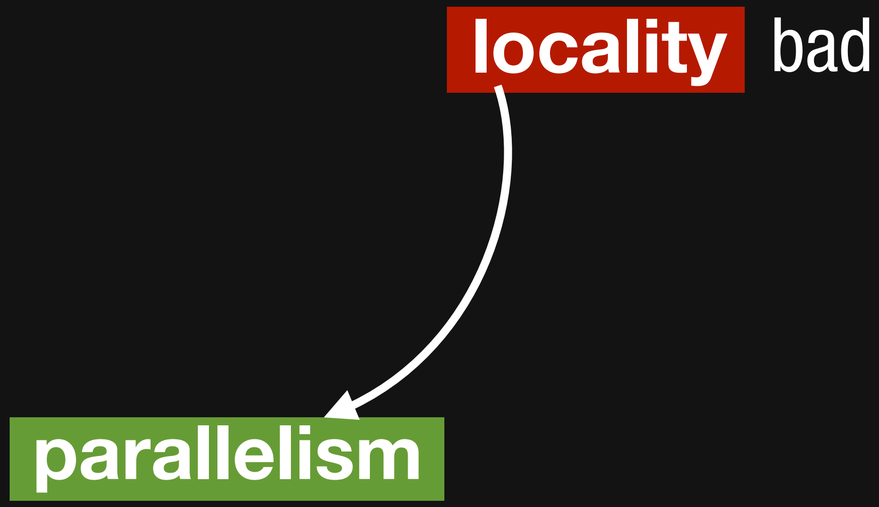


Simple loops execute **breadth-first** across stages

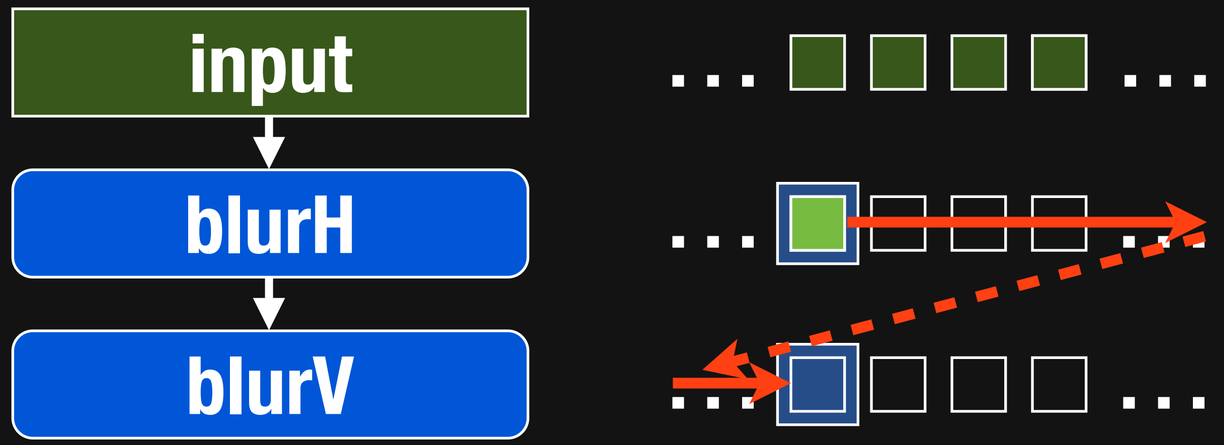
parallelism easy within each stage

Breadth-first execution **sacrifices locality**





Breadth-first execution **sacrifices locality**

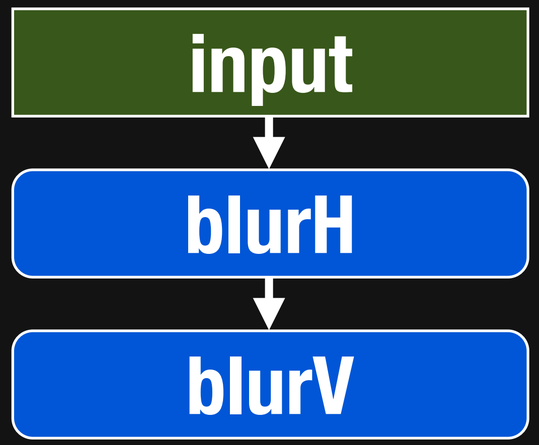


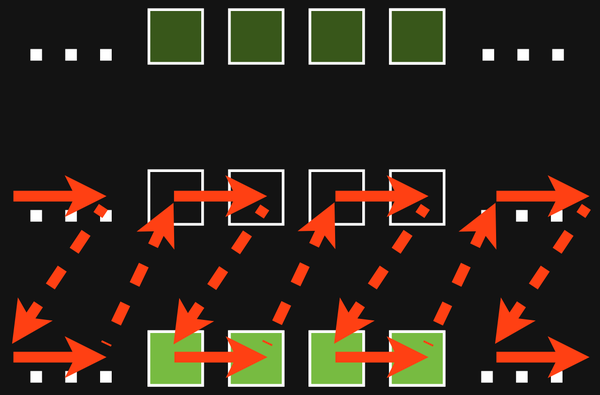
locality is a function of **reuse distance**

locality → **parallelism**

Interleaved execution (fusion) **improves locality**

*fusion **globally interleaves** computation*



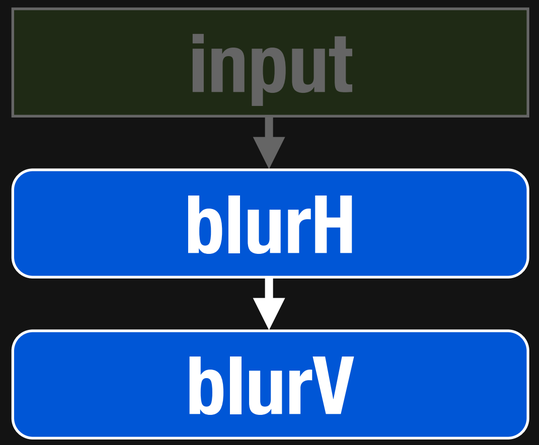


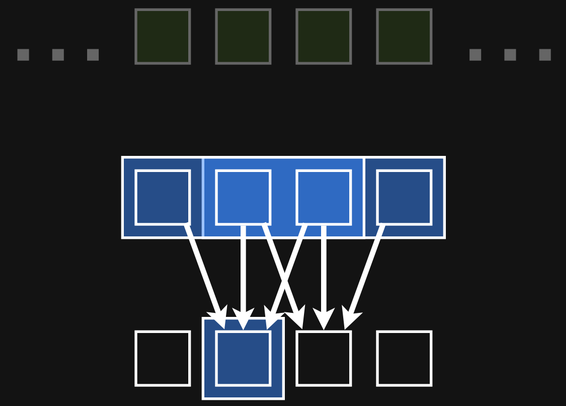
reduce reuse distance from **producer** to **consumer**

locality

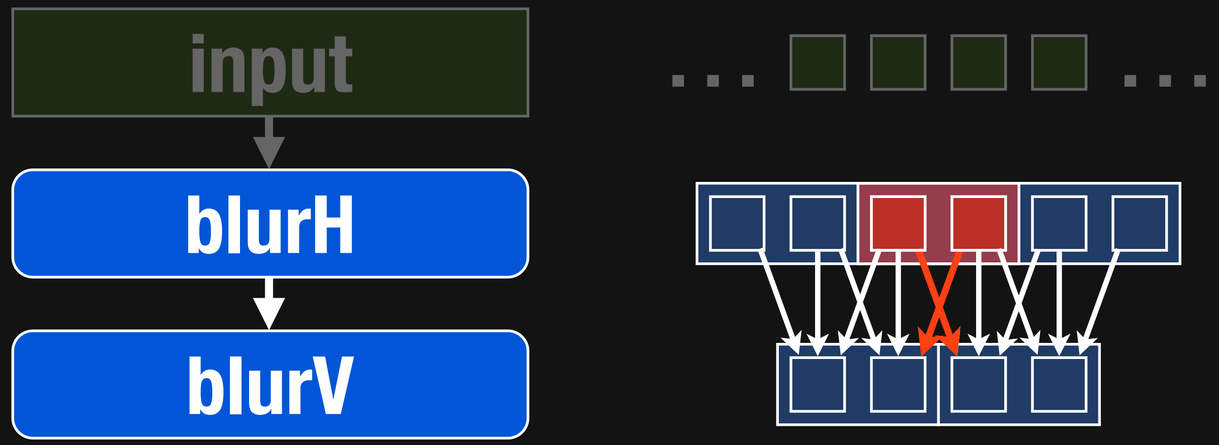
parallelism

Understanding dependencies

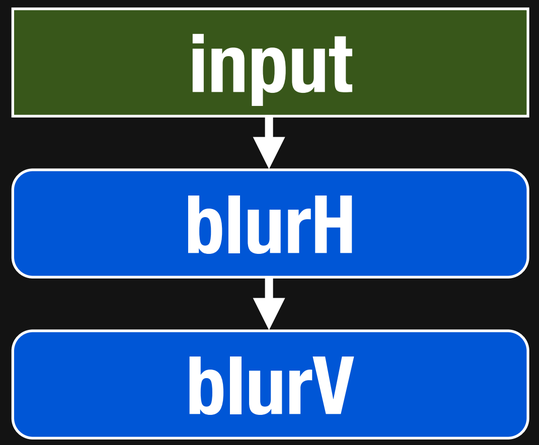


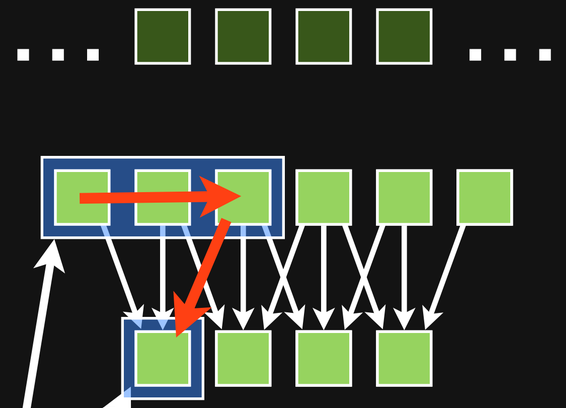


Stencils have overlapping dependencies



Sliding window execution **sacrifices parallelism**





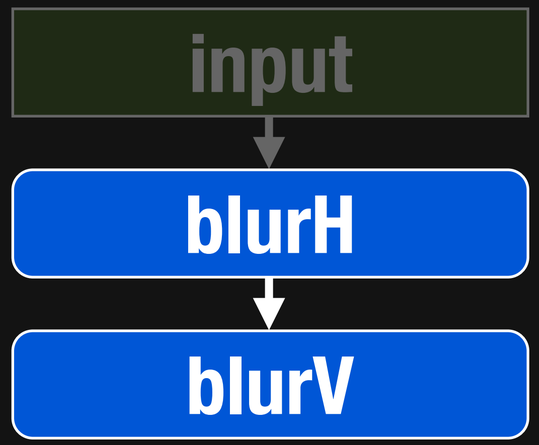
fixed order, constrains parallelism

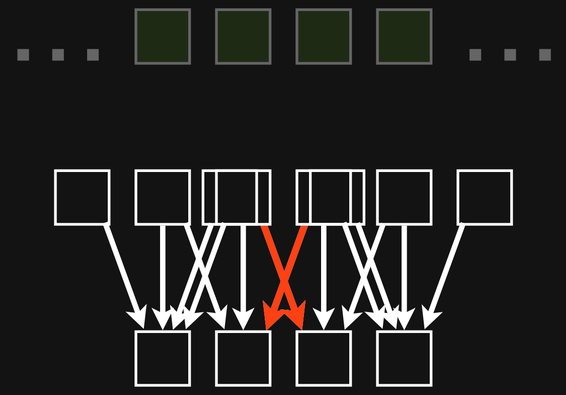
short reuse distance

locality

parallelism bad because serial dependence

Breaking dependencies with tiling

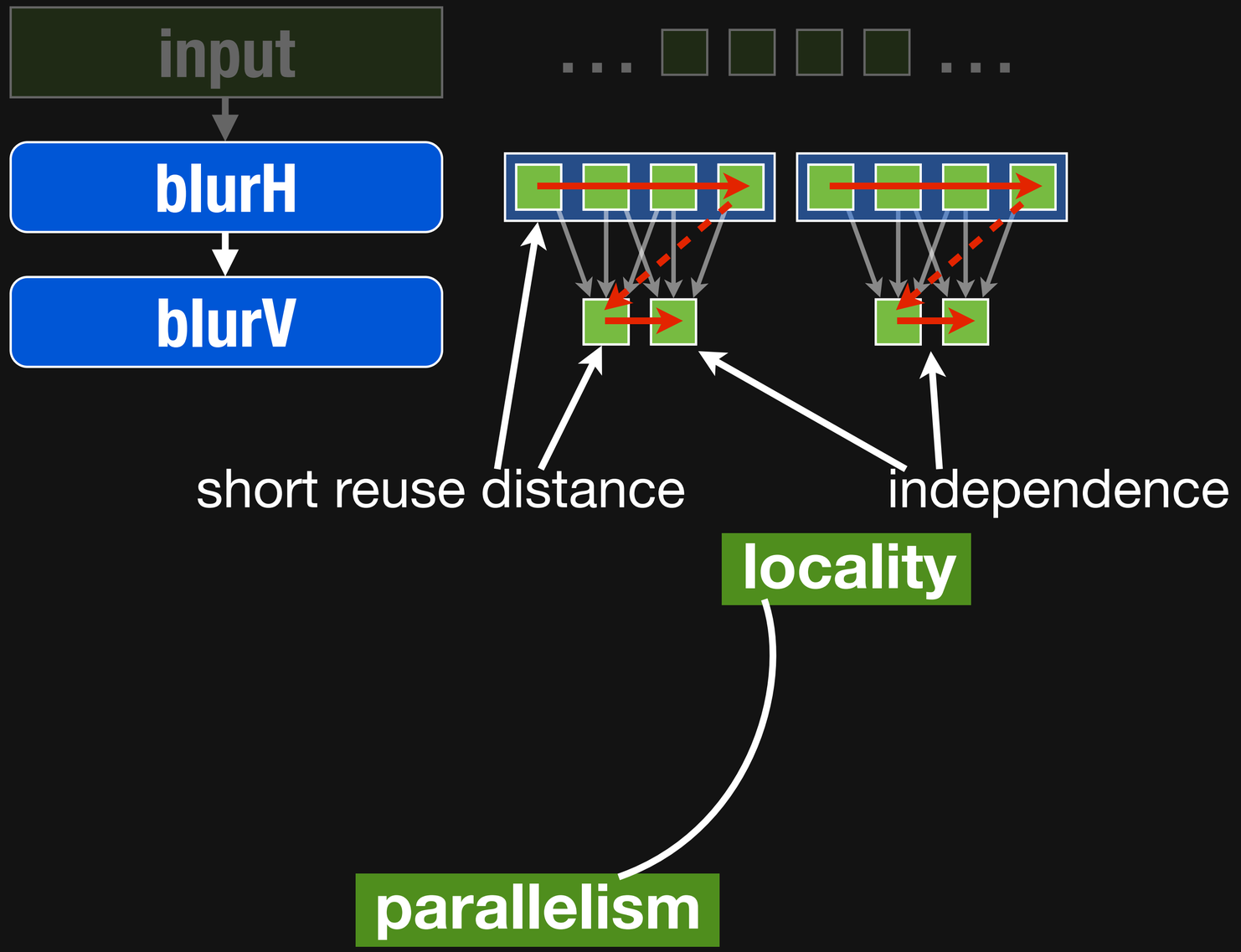




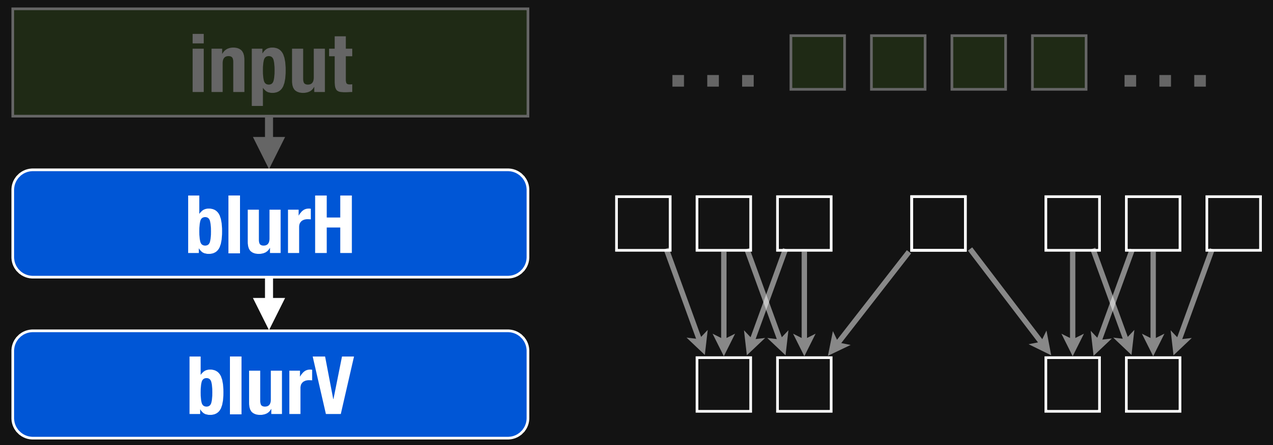
locality

parallelism

Decoupled tiles optimize **parallelism & locality**



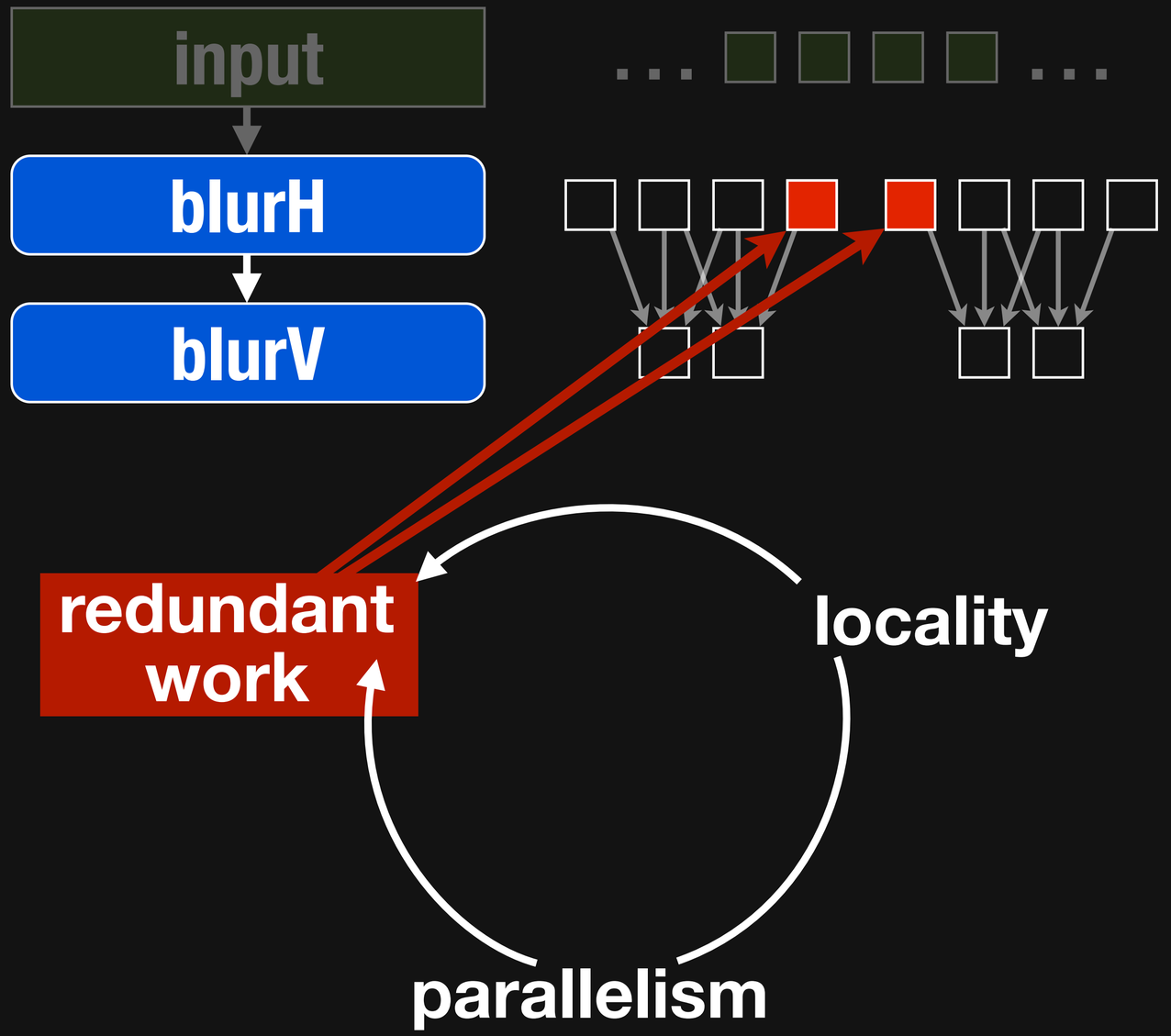
Breaking dependencies introduces **redundant work**



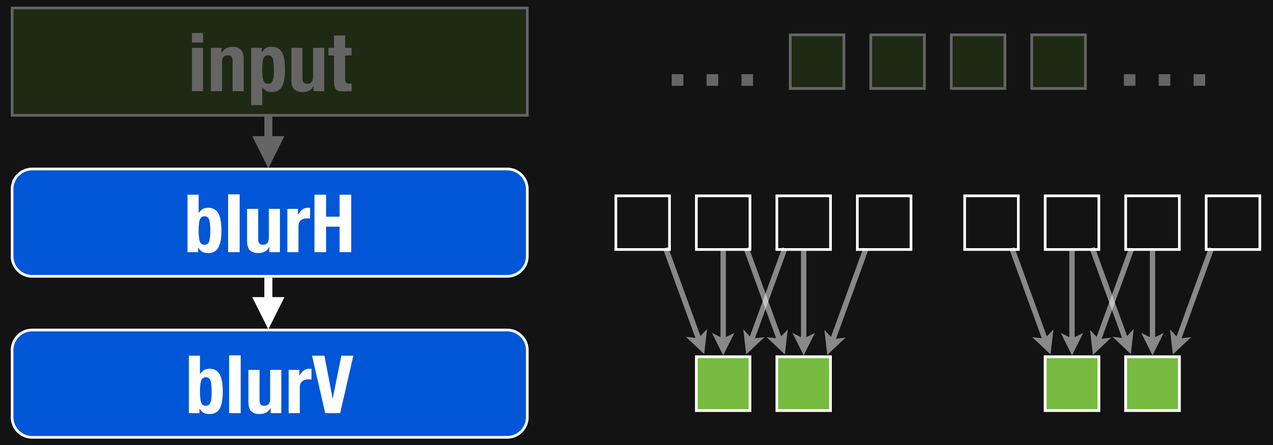
locality

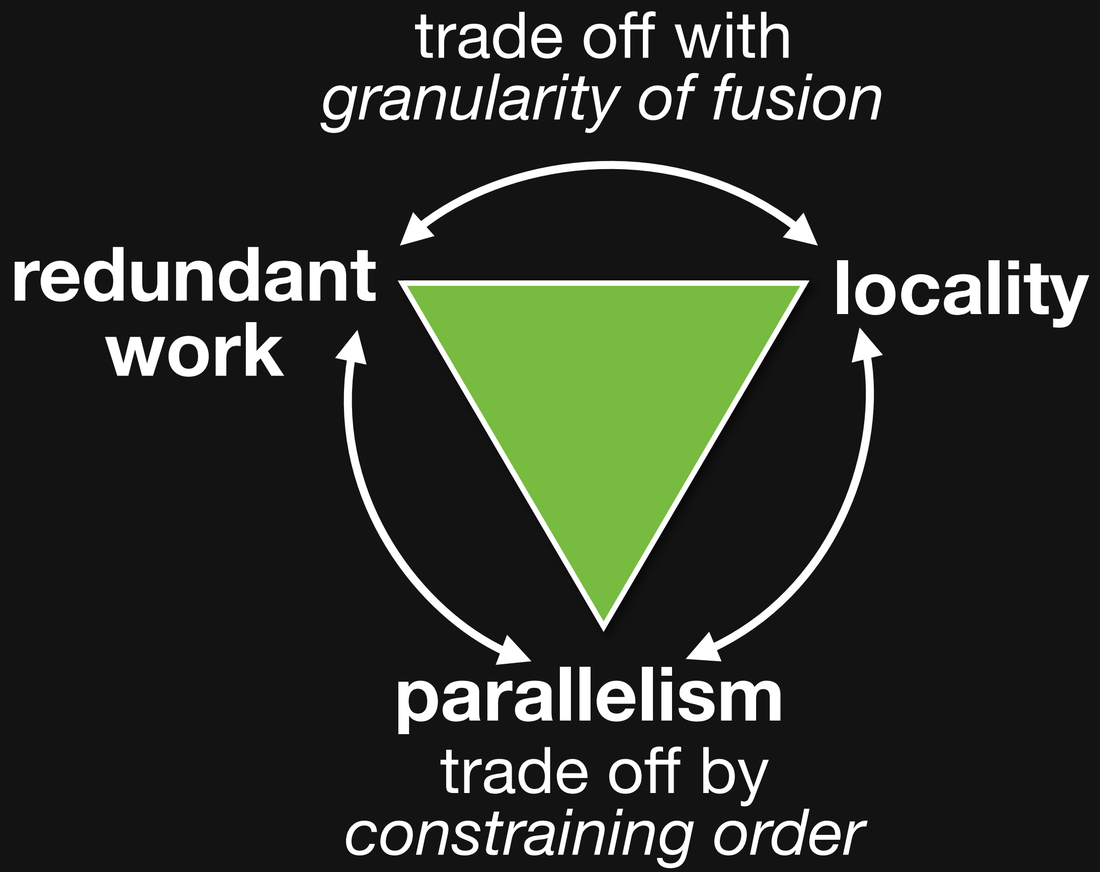
parallelism

Breaking dependencies introduces **redundant work**

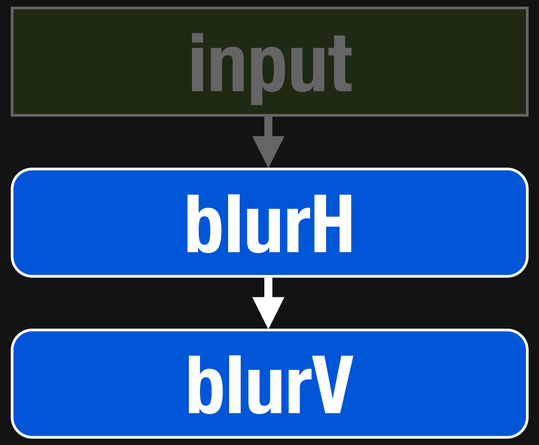


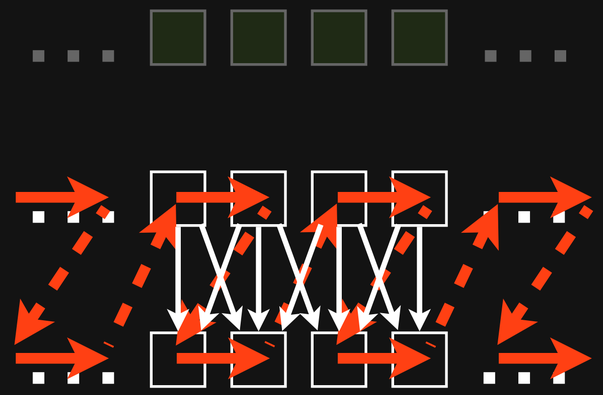
Message #1: Performance is a tension between three issues





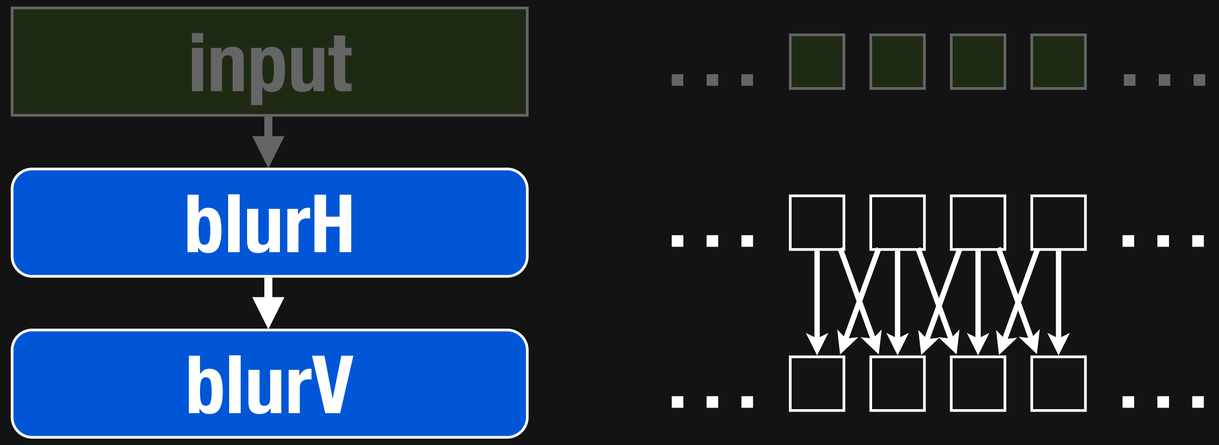
Message #2: algorithm vs. organization





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

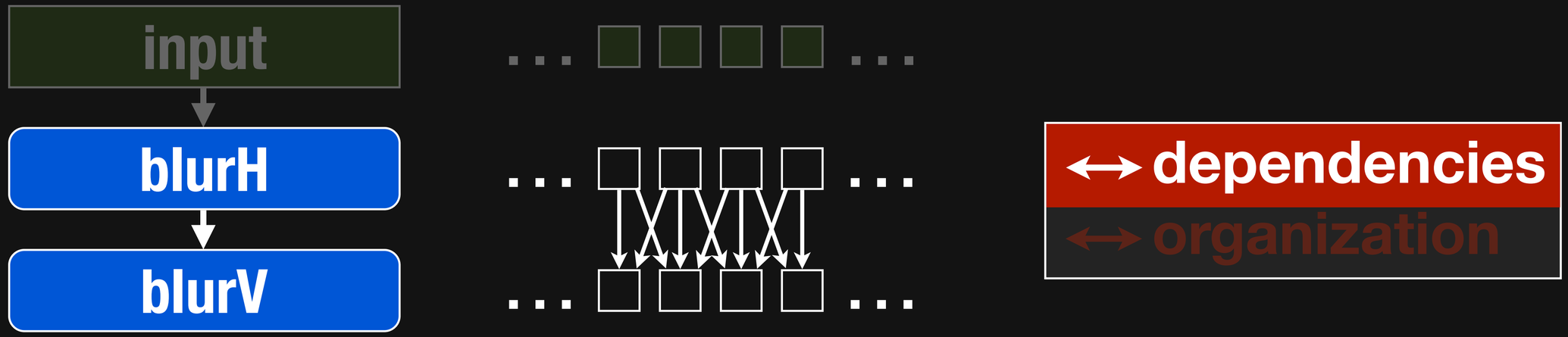
Dependencies limit choices of organization



↔ dependencies

↔ organization

Dependencies limit choices of organization



Hence need for domain-specific language

functional (no side effect, no variable aliasing)

understanding of data structure

analysis of indexing and tile bound expansion

A language of schedules

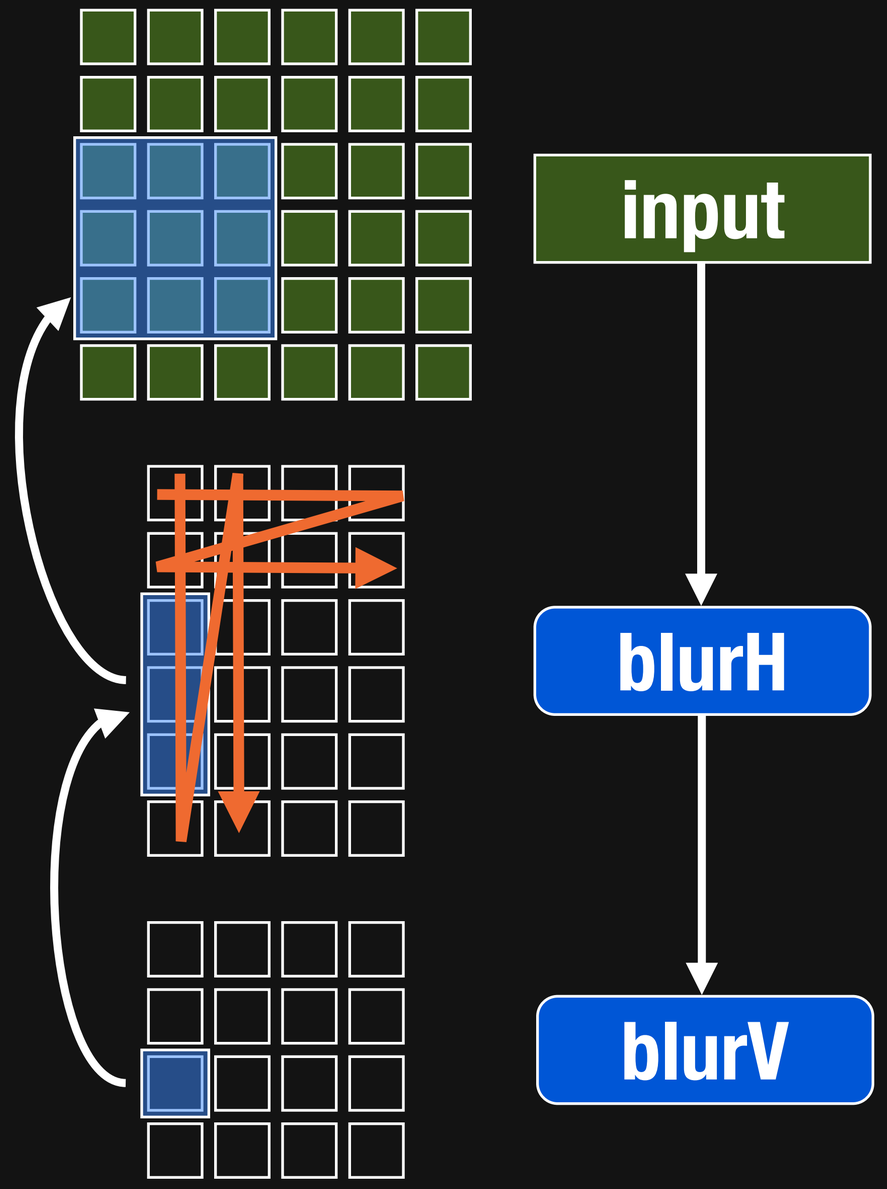
The schedule defines intra-stage order, inter-stage interleaving

For each stage:

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

2) **When** (at what granularity) should we **compute** its **inputs**?

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



Recall Halide separable box blur

```
// The algorithm - no storage, order
 blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

Halide scheduling co-language example

0.9 ms/megapixel

```
// The algorithm - no storage, order
 blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

 // The schedule - defines order, locality; implies storage
 blurV.tile(tx, ty, xi, yi, 256, 32)
       .parallel(ty);

 blurH.compute_at(blurV, tx);
```

Compute 2nd blur in tiles

Parallelize over tile rows

Halide scheduling co-language example

0.9 ms/megapixel

```
// The algorithm - no storage, order
 blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
 blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

 // The schedule - defines order, locality; implies storage
 blurV.tile(tx, ty, xi, yi, 256, 32)
       .parallel(ty);

 blurH.compute_at(blurV, tx);
```

Compute 2nd blur in tiles
Parallelize over tile rows

Compute first blur at granularity of tiles of second blur

i.e., compute all the blurH values needed to compute one tile of blurV

Given algorithm , The schedule specifies the organization of computation and storage

Boils down to specifying nested (possibly parallel) loops

e.g. schedule in previous slide:

```
blurV.tile(tx, ty, xi, yi, 256, 32).parallel(y);
blurH.compute_at(blurV, tx);
```

```
paralel_for ty:
   for tx:
      for yi:
         for xi:
            tmp(xi, yi) =…
      for yi:
         for xi:
            out(ty*tile_height+yi, tx*tile_width+xi) =…
```

Halide takes care of loops synthesis, bound expansion, indexing, allocation

Meaning of compute_at

Specify at which loop of blurV we insert the code for blurH

e.g. schedule in previous slide:

```
blurV.tile(tx, ty, xi, yi, 256, 32).parallel(y);
blurH.compute_at(blurV, tx);
```

```
paralel_for ty:
    for tx:
        for yi:
            for xi:
                tmp(xi, yi) =…
        for yi:
            for xi:
                out(ty*tile_height+yi, tx*tile_width+xi) =…
```

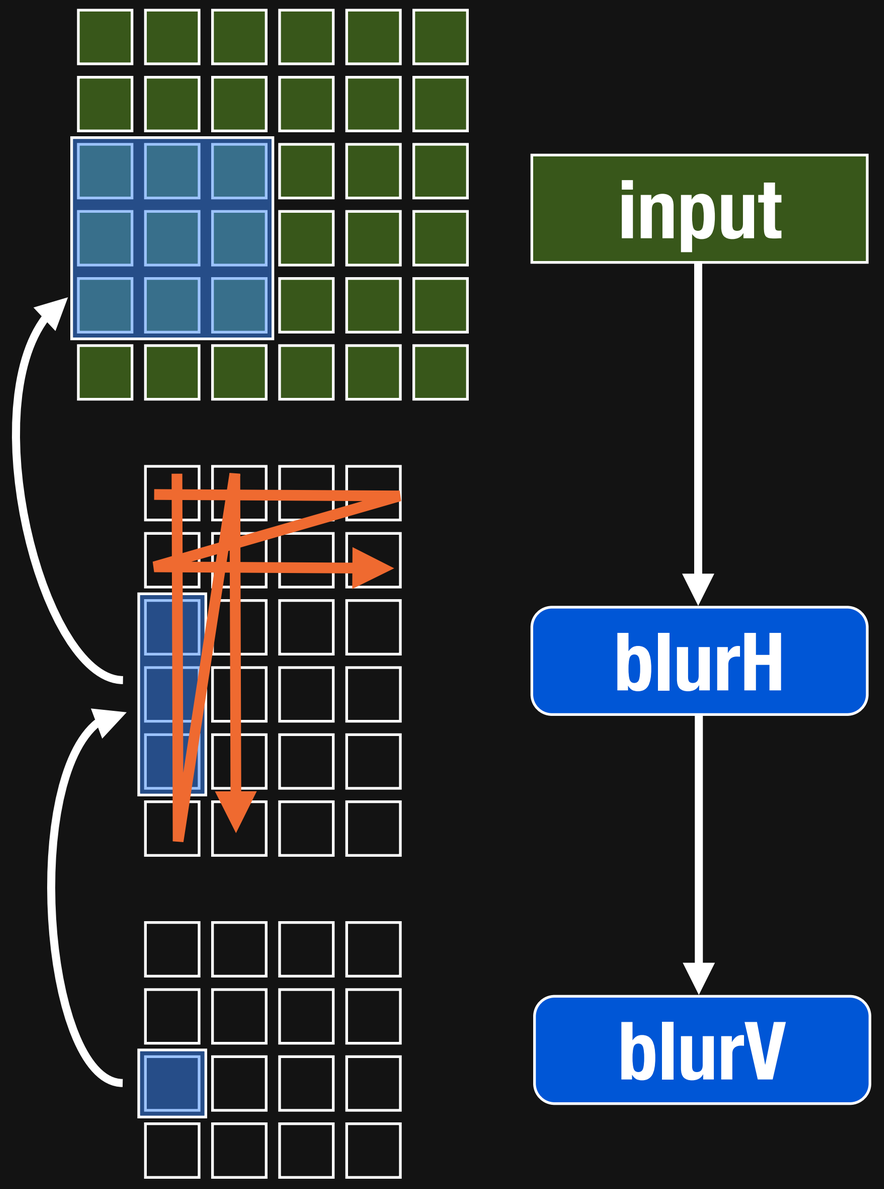
The schedule defines intra-stage order, inter-stage interleaving

For each stage:

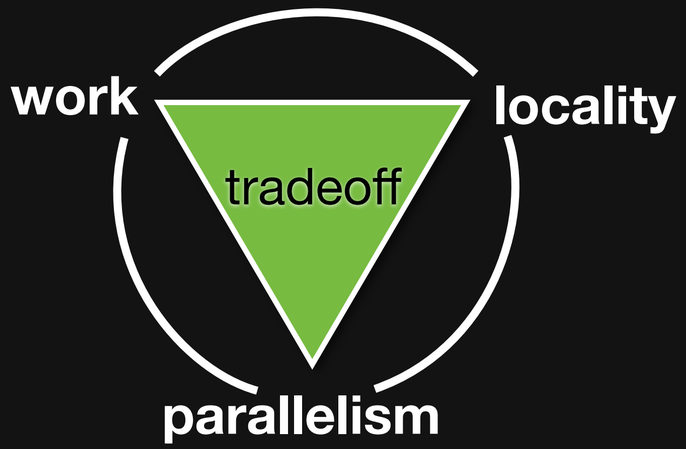
1) In **what order** should we **compute** its **values**?

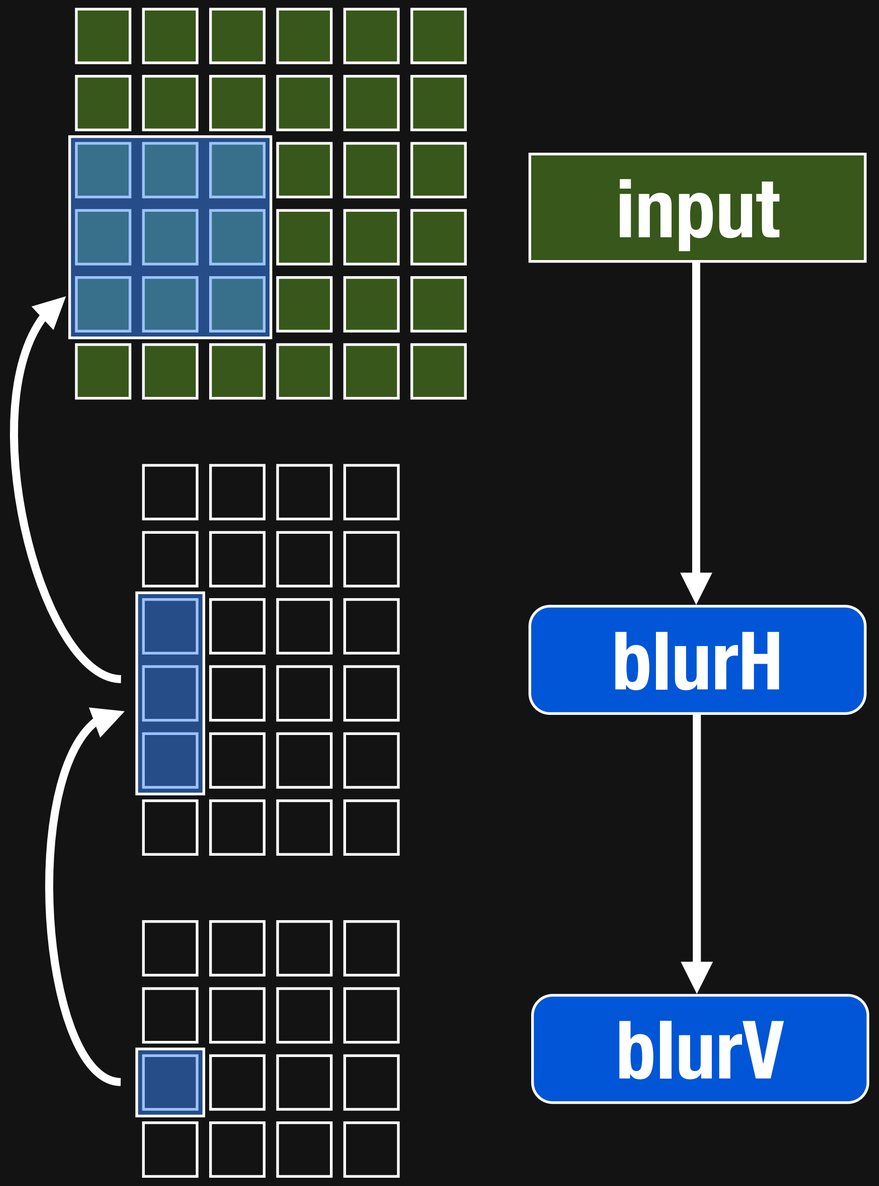
2) **When** (at what granularity) should we **compute** and store its **inputs**?

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

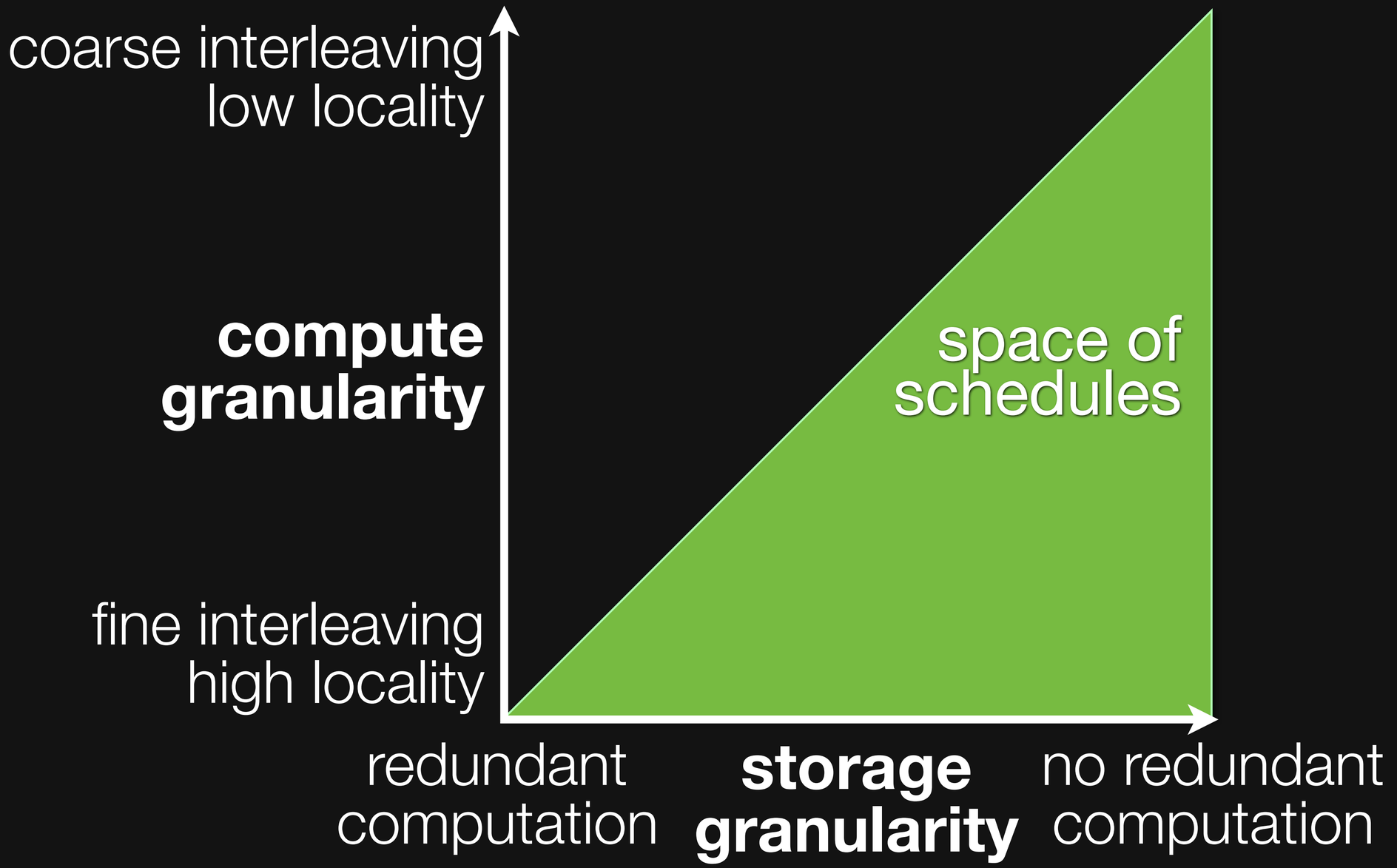


The schedule defines intra-stage order, inter-stage interleaving

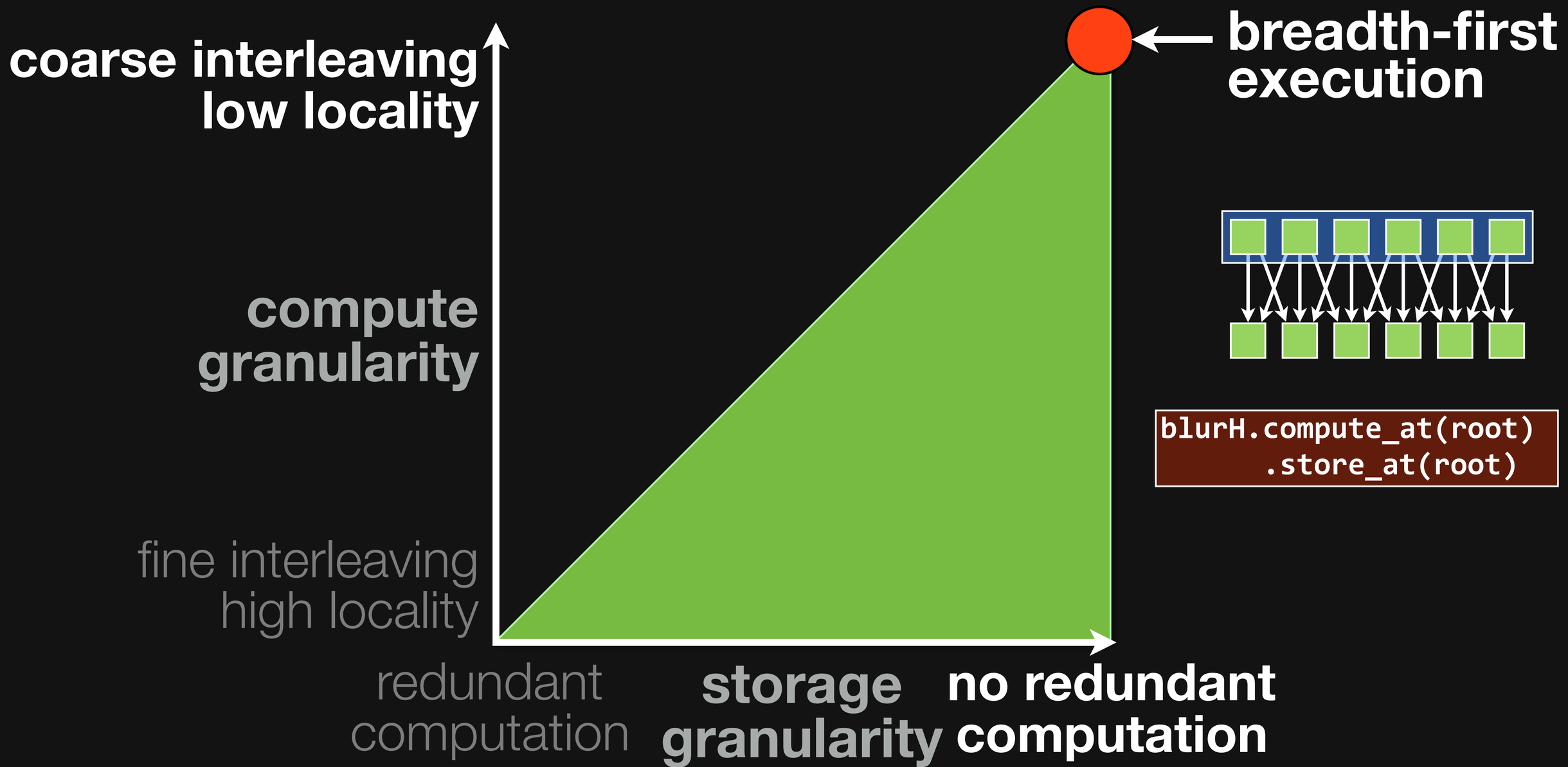




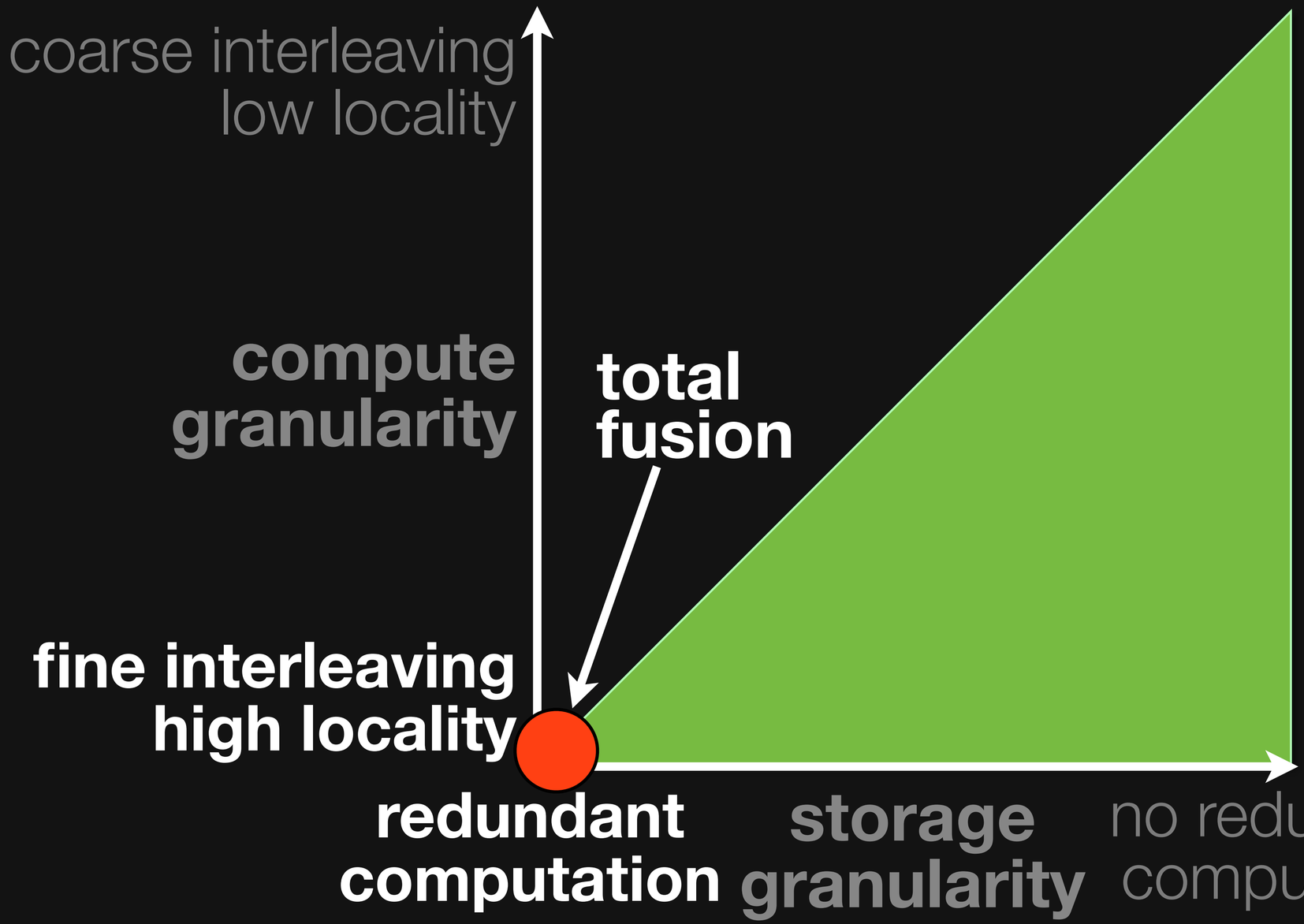
Tradeoff space modeled by **granularity of interleaving**



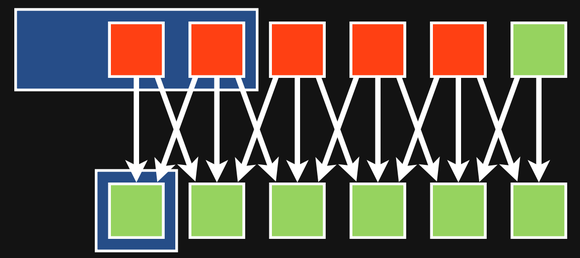
Tradeoff space modeled by **granularity of interleaving**



Tradeoff space modeled by **granularity of interleaving**



redundant work



```
blurH.compute_at(blurV, x)
     .store_at(blurV, x)
```

Tradeoff space modeled by **granularity of interleaving**

coarse interleaving
low locality

compute granularity

fine interleaving
high locality

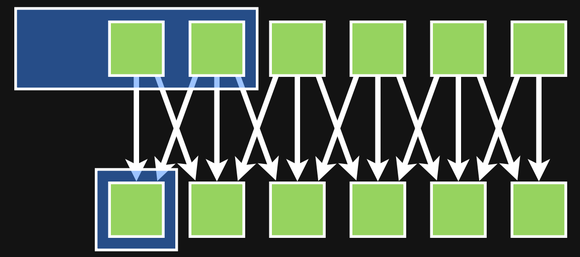
redundant computation

storage granularity

no redundant computation

capturing reuse constrains order (less parallelism)

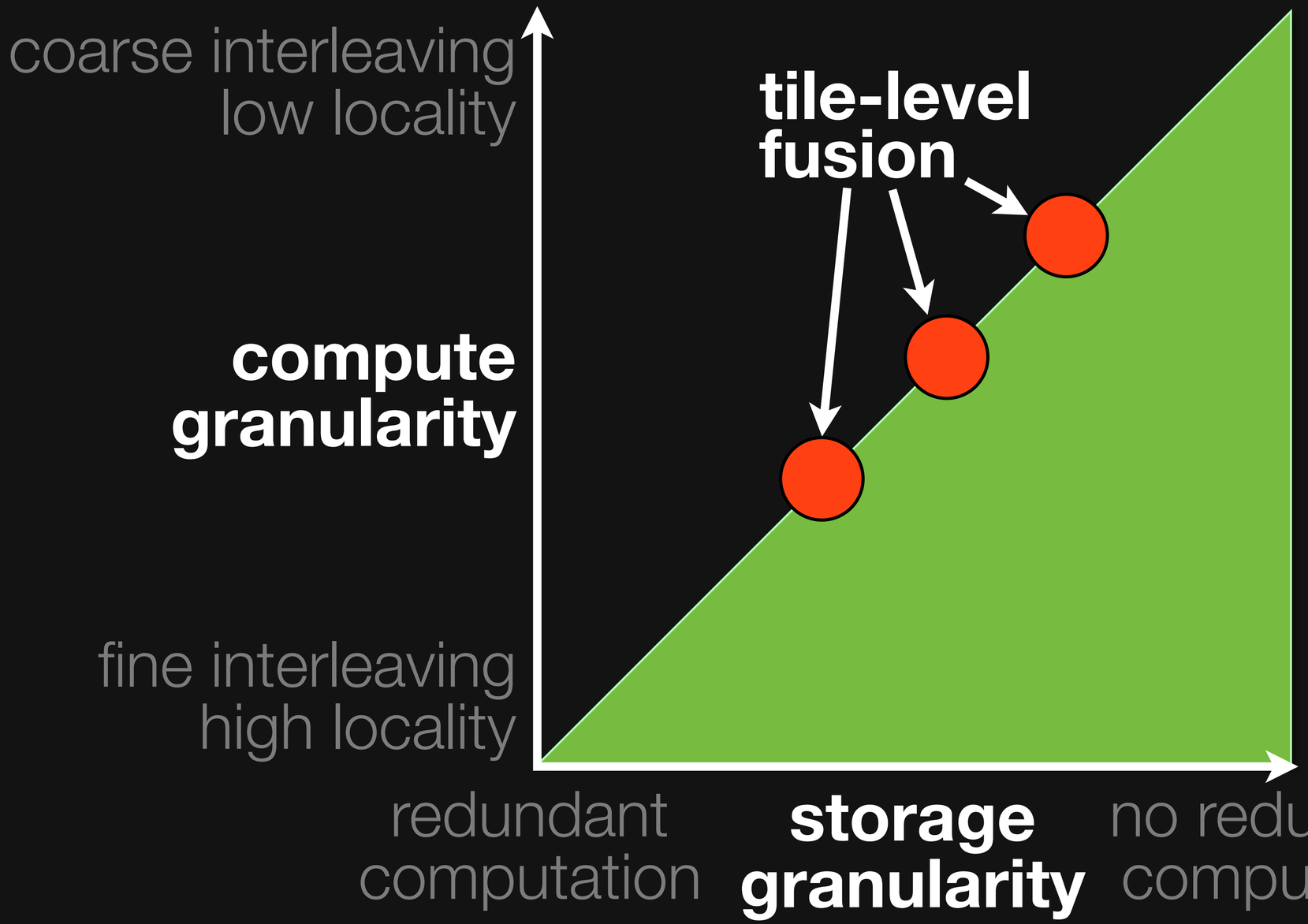
parallelism

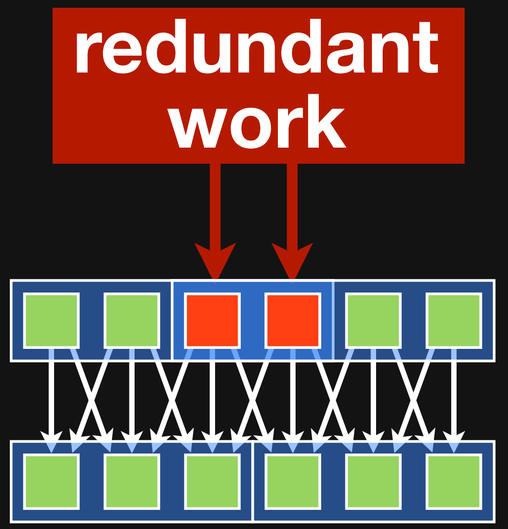


```
blurH.compute_at(blurV, x)
     .store_at(root)
```

sliding window
fusion

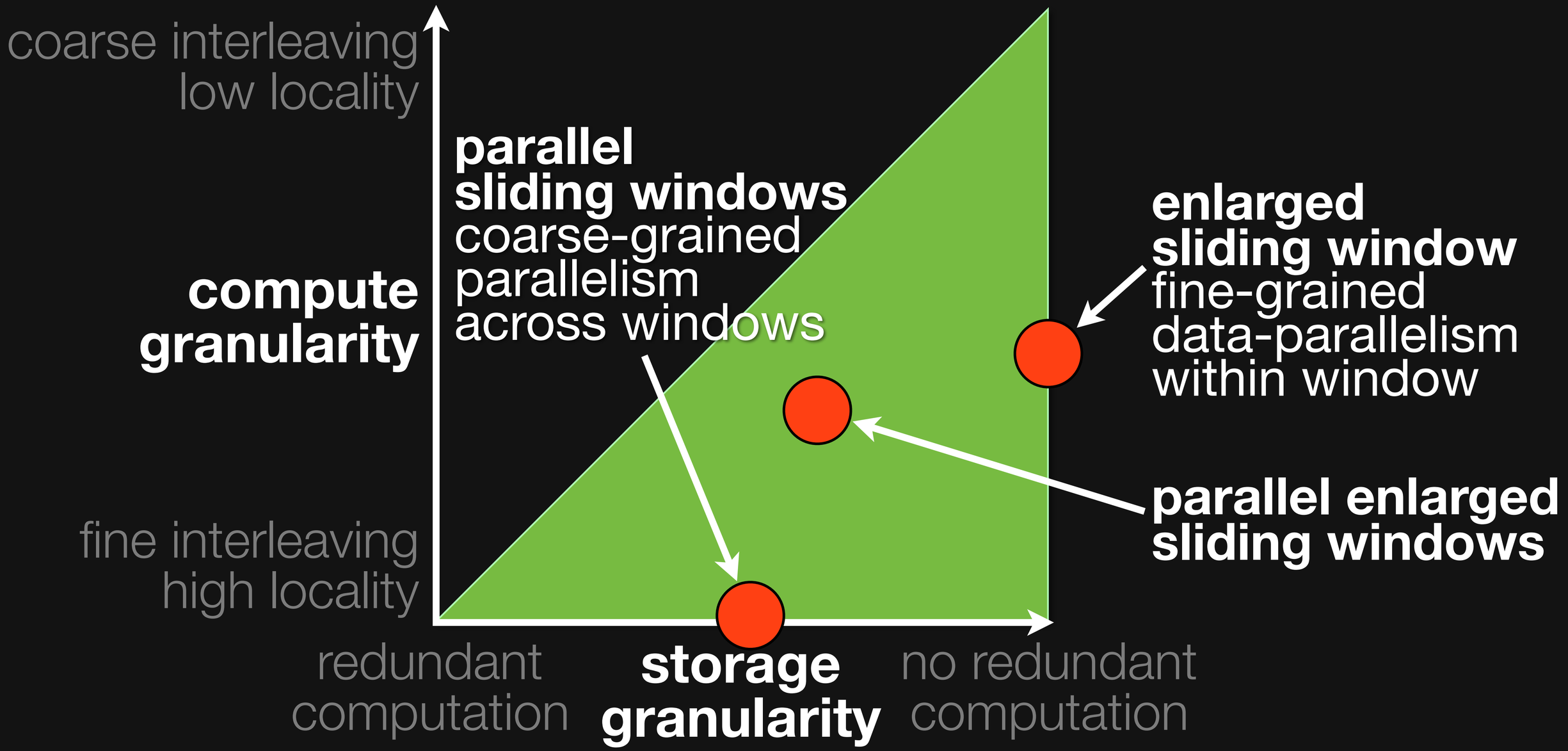
Tradeoff space modeled by **granularity of interleaving**



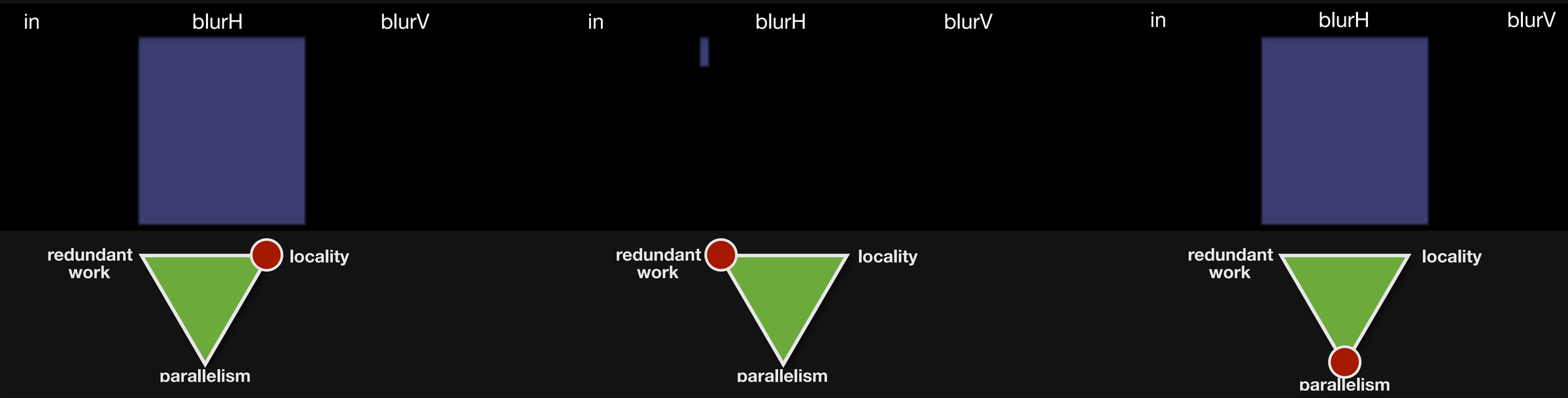



```
blurV.tile(tx, ty,
           xi, yi,
            W,  H)
blurH.compute_at(blurV, tx)
      .store_at(blurV, tx)
```

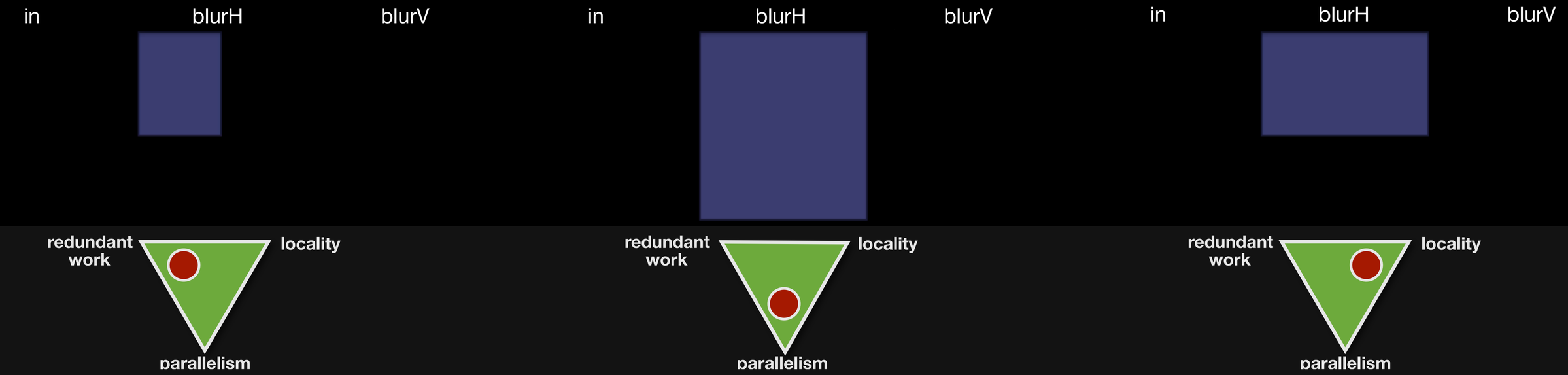
Tradeoff space modeled by **granularity of interleaving**



Schedule primitives **compose** to create many organizations




```
blurH.compute_at(blurV, y)
```



Example: box blur, 5x5, 35 MPixels on 12 cores

root first stage

took 1.99803357124 seconds

tile 256 x 256 + fuse

took 1.7552740097 seconds

tile 256 x 256 + parallel+vector

took 0.550438785553 seconds

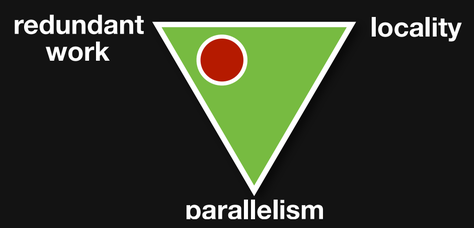
tile 256 x 256 + parallel+vector but without fusion

took 1.10970659256 seconds

Note the exact doubling (memory bound)

Schedule primitives **compose** to create many organizations

```
blurH.compute_at(blurV, x)
      .vectorize(x, 4)

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




```
void box_filter_3x3(const Image &in, Image &blurV) {
  __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      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(blurHPtr++, avg);
          inPtr += 8;
      }}
      blurHPtr = blurH;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blurV[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurHPtr+(2*256)/8);
          b = _mm_load_si128(blurHPtr+256/8);
          c = _mm_load_si128(blurHPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

Halide

0.9 ms/megapixel

```
Func box_filter_3x3(Func in) {
  Func blurH, blurV;
  Var x, y, xi, yi;

  // The algorithm - no storage, order
  blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
  blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

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

  return blurV;
}
```

Halide

0.9 ms/megapixel

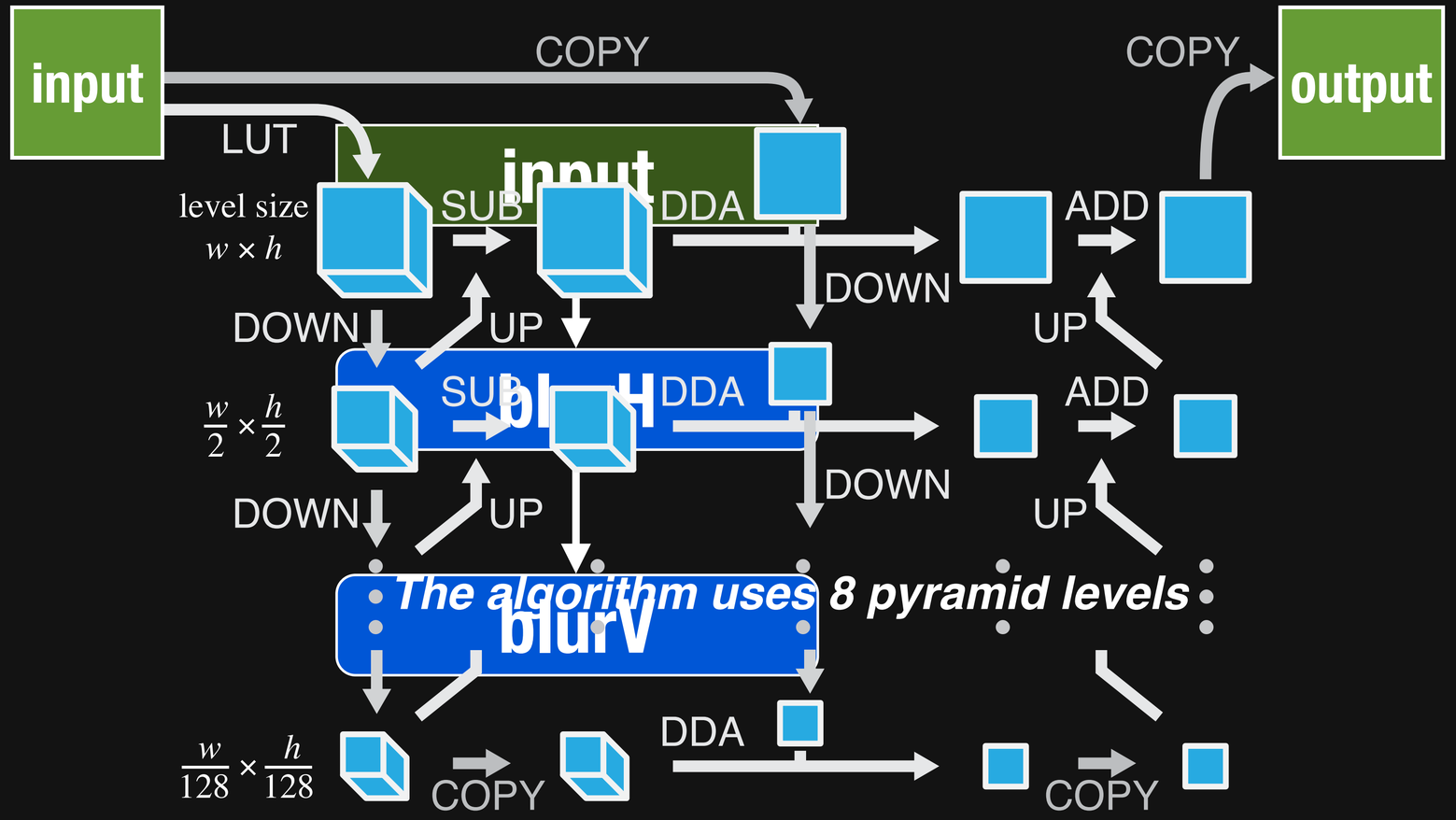
```
Func box_filter_3x3(Func in) {
  Func blurH, blurV;
  Var x, y, xi, yi;

  // The algorithm - no storage, order
  blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
  blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

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

  return blurV;
}
```

C++

0.9 ms/megapixel

```
void box_filter_3x3(const Image &in, Image &blurV) {
  __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 blurH[(256/8)*(32+2)]; // allocate tile blurH array
    for (int xTile = 0; xTile < in.width(); xTile += 256) {
      __m128i *blurHPtr = blurH;
      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(blurHPtr++, avg);
         inPtr += 8;
      }}
      blurHPtr = blurH;
      for (int y = 0; y < 32; y++) {
        __m128i *outPtr = (__m128i *)(&(blurV[yTile+y][xTile]));
        for (int x = 0; x < 256; x += 8) {
          a = _mm_load_si128(blurHPtr+(2*256)/8);
          b = _mm_load_si128(blurHPtr+256/8);
          c = _mm_load_si128(blurHPtr++);
          sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
          avg = _mm_mulhi_epi16(sum, one_third);
          _mm_store_si128(outPtr++, avg);
}}}}}
```

Organization requires global tradeoffs



LUT: look-up table

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

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

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

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

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

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

3x3 box filter

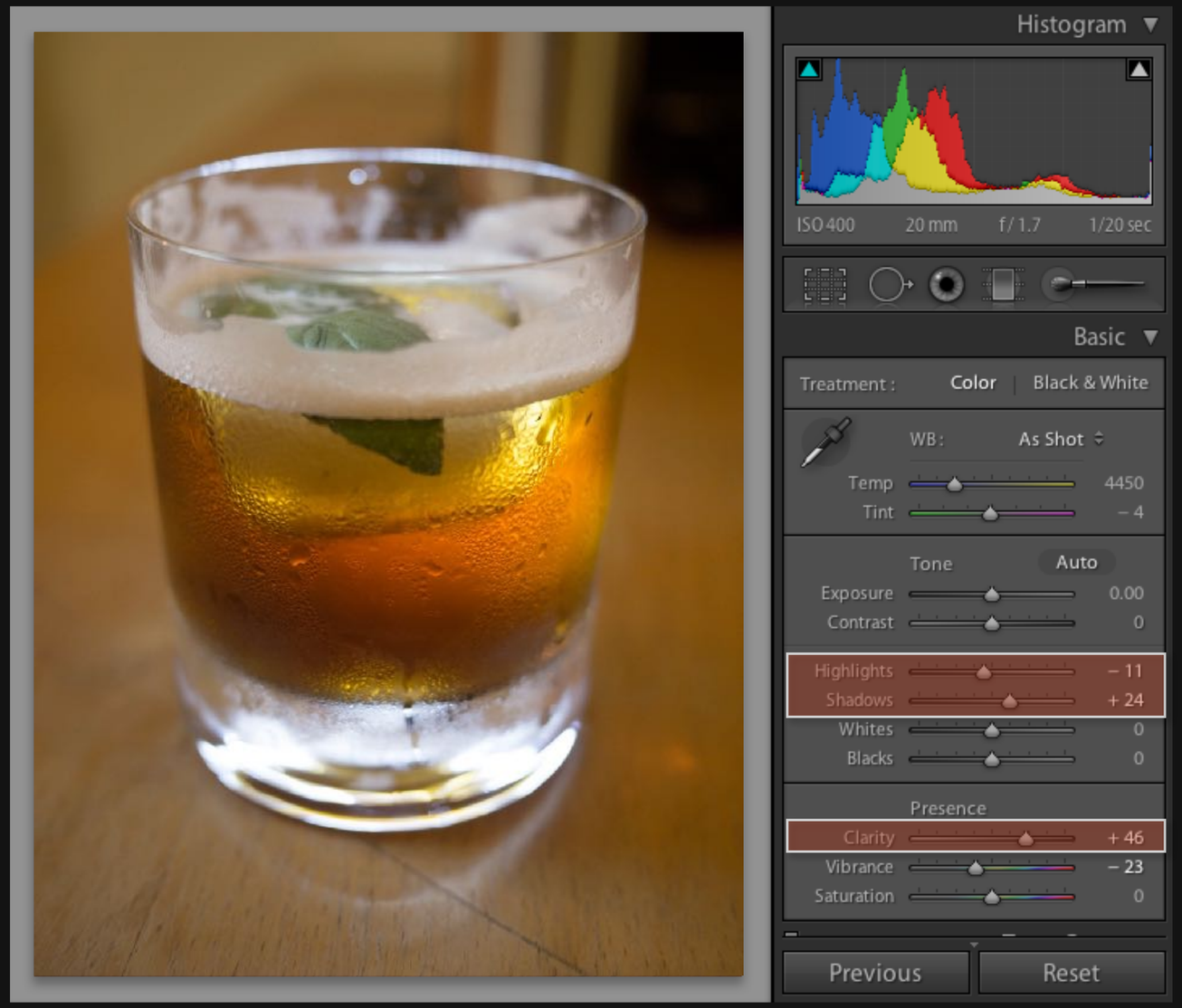
[Paris et al. 2010, Aubry et al. 2011]

Local Laplacian Filters
in Adobe Photoshop

Adobe: 1500 lines
expert-tuned 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**



Adobe: 1500 lines

expert-tuned C++

multi-threaded, SSE

3 months of work

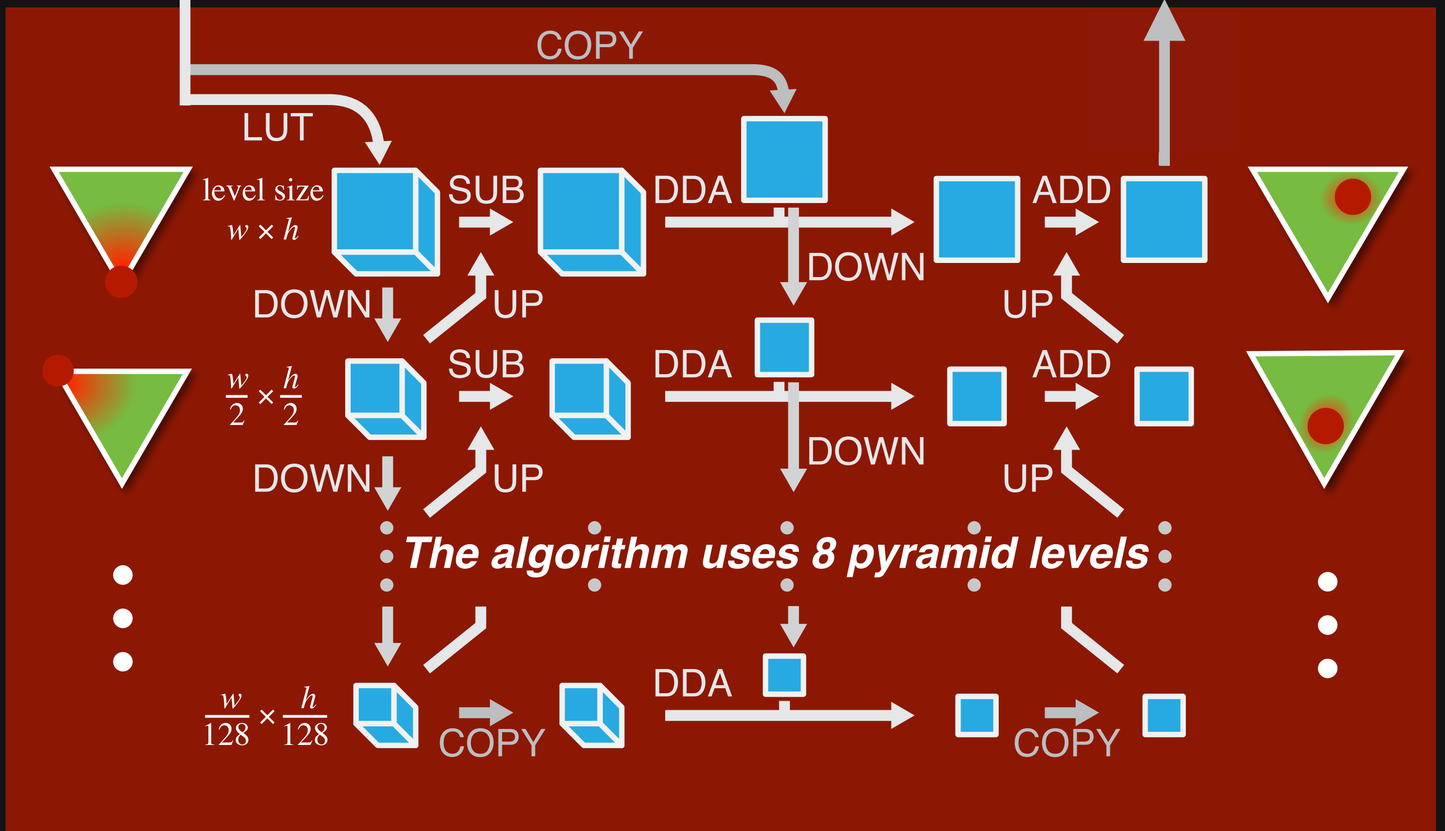
10x faster than original C++

Halide: 60 lines

1 intern-day

Halide vs. Adobe:

2x faster on same **CPU**



Adobe: 1500 lines

expert-tuned C++

multi-threaded, SSE

3 months of work

10x faster than original C++

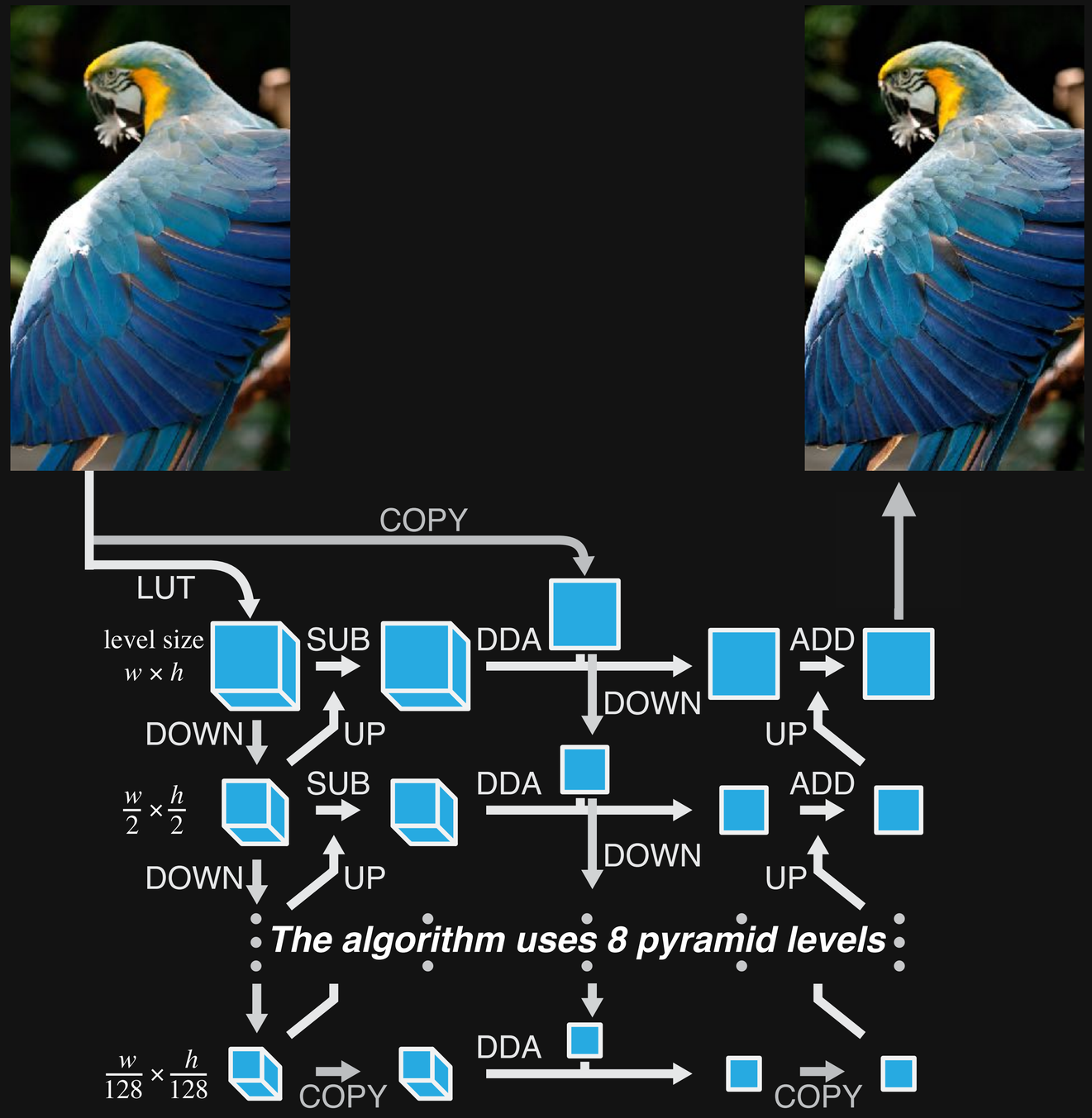
Halide: 60 lines

1 intern-day

Halide vs. Adobe:

2x faster on same **CPU**

10x faster** on **GPU



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.

In practice

Halide is embedded in C++

classes and operator overloading define program representation

e.g. A Func is encoded as an algebraic tree

= is overloaded to define algebraic tree

() is overloaded to specify symbolic variables

```
Func blurH, blurV;
Var x, y, xi, yi;

// The algorithm - no storage, order
blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;

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

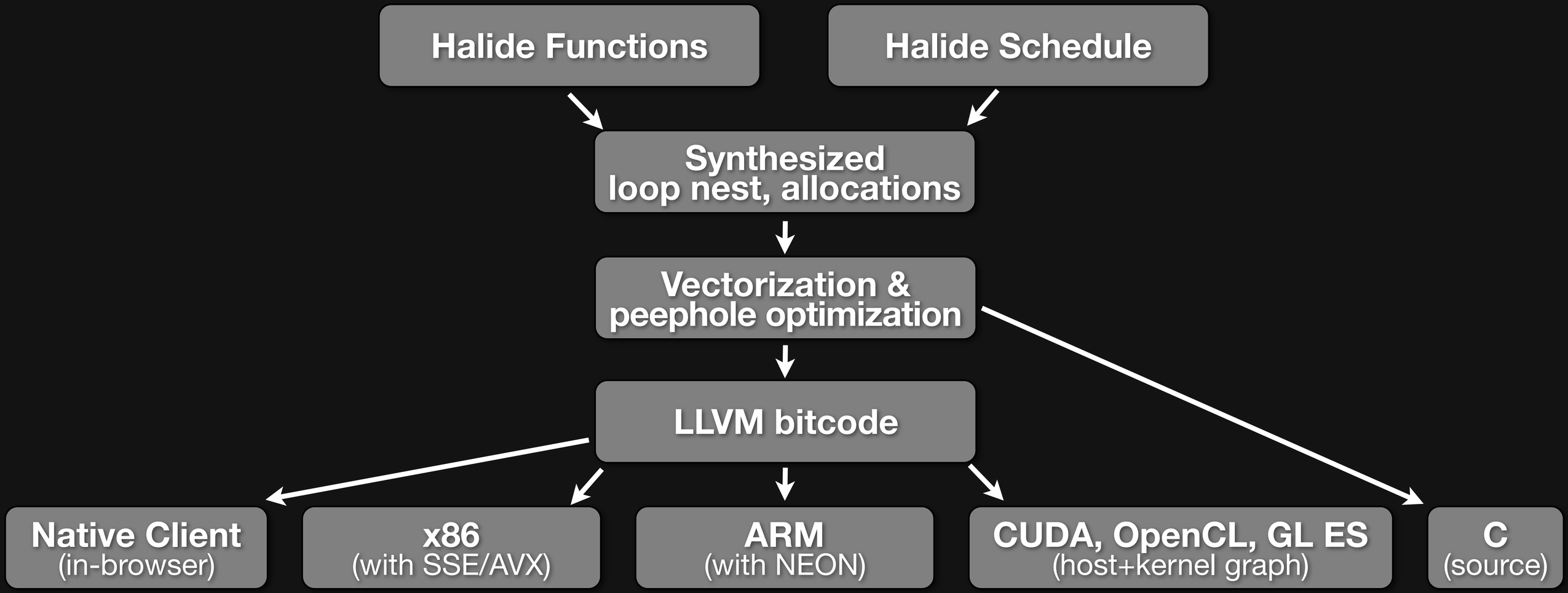
In practice

Halide is embedded in C++

Compiled as JIT or statically by calling Func.realize()

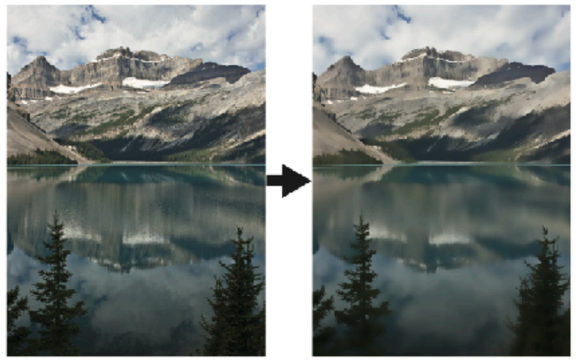
- Evaluate bound expansions (symbolic, interval analysis)
- Organize loop nests and temporary array allocation based on schedule
- Call LLVM for code gen and low-level optimization

The Halide Compiler



Top performance, concise and portable code

Figure 3: Summary of the code size and running time Halide implementations compared to hand-written reference implementations of four representative computational photography algorithms. The time measurements are reported in milliseconds to process the representative image (4–6 megapixels), except for the fast Fourier transform. The FFT experiment is for the tiled real-to-complex transform, and reports time in nanoseconds per 16 × 16 tile.



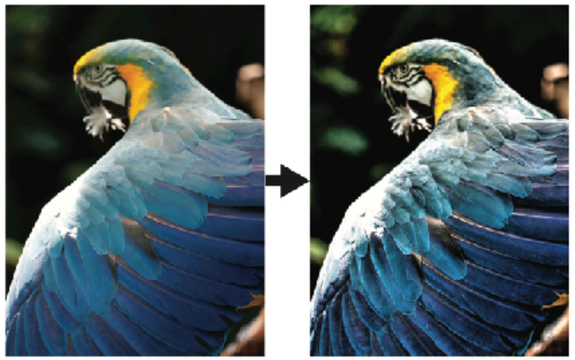
Bilateral grid

Reference C++: 122 lines
Quad core x86: 150ms

CUDA C++: 370 lines
GTX 980: 2.7ms

Halide algorithm: 34 lines
schedule: 6 lines
Quad core x86: 14ms

GPU schedule: 6 lines
GTX 980: 2.3ms

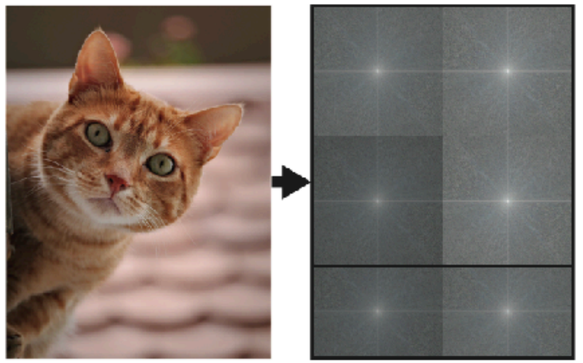


Local Laplacian filters

C++, OpenMP+iIPP: 262 lines
Quad core x86: 210ms

Halide algorithm: 62 lines
schedule: 11 lines
Quad core x86: 92ms

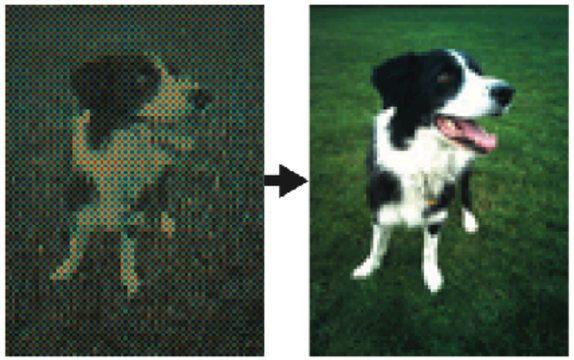
GPU schedule: 9 lines
GTX 980: 23ms



Fast Fourier transform

FFTW: *thousands*
Quad core x86: 384ns
Quad core ARM: 5960ns

Halide algorithm: 350 lines
schedule: 30 lines
Quad core x86: 250ns
Quad core ARM: 1250ns



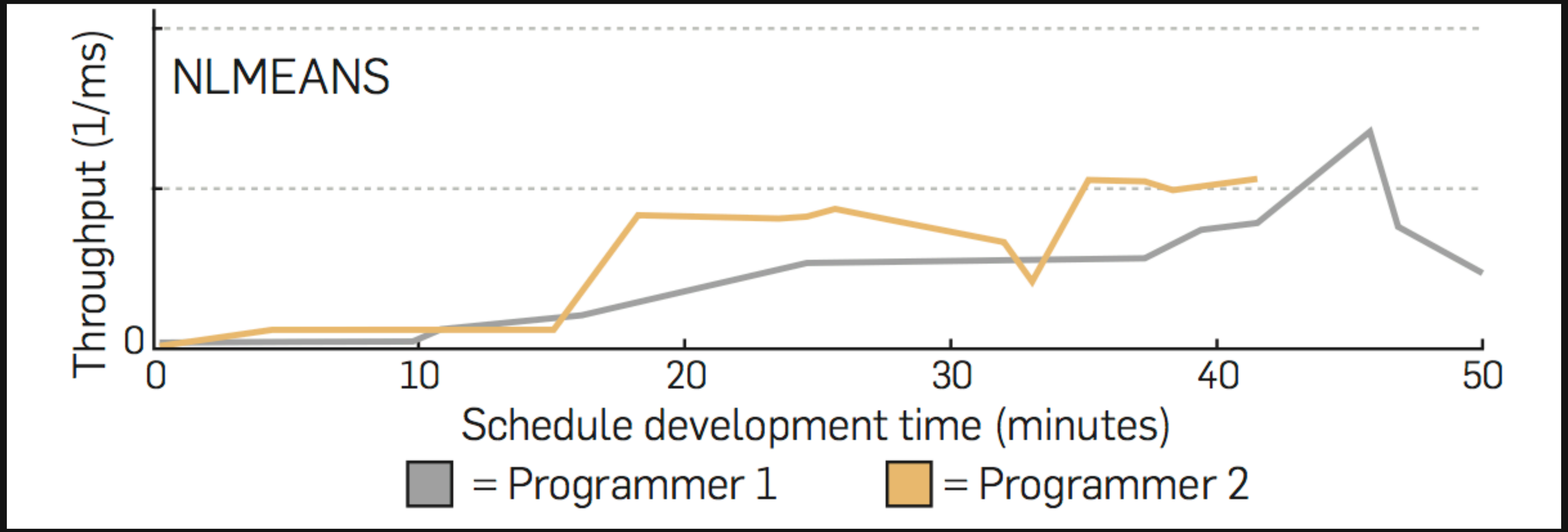
Camera pipeline

Optimized assembly: 463 lines
ARM core: 39ms

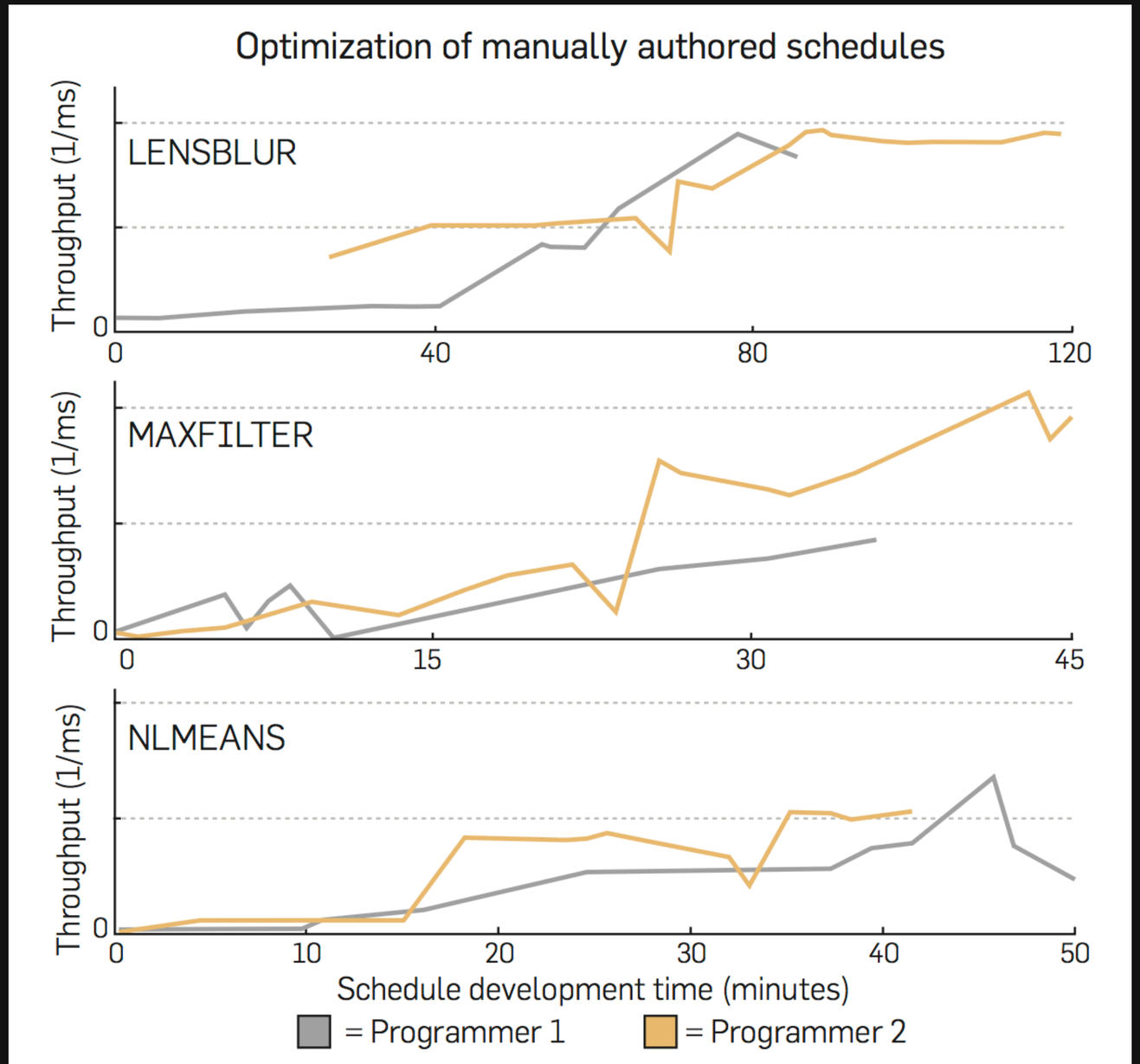
Halide algorithm: 170 lines
schedule: 50 lines
ARM core: 41ms

DSP schedule: 70 lines
Hexagon 680: 15ms

Schedule exploration by pros



Schedule exploration by pros



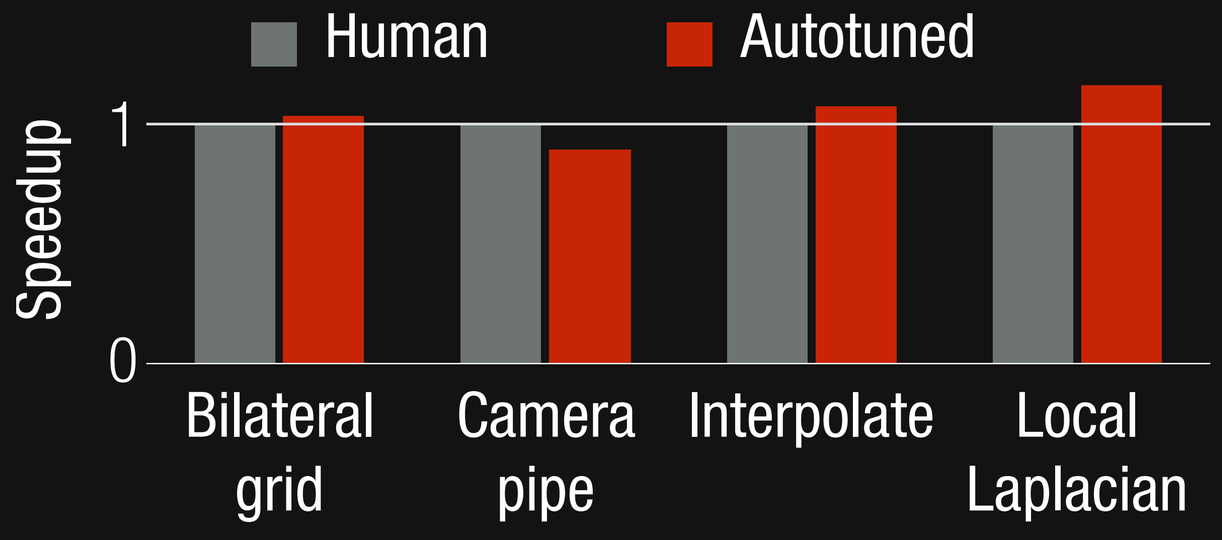
How can we automatically find good organizations?

Autotuning: stochastic search over organizations

Model: **Halide schedules**

Objective: **measured benchmark runtime**

Algorithm: **hybrid heuristic search**

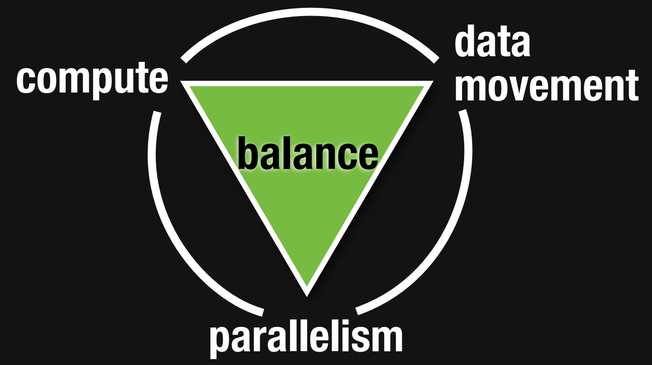


[Halide, PLDI 2013]

Direct heuristic scheduling [SIGGRAPH 2016]

Ravi Mullapudi et al, not my work

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

Mathematical transformations enable new organization

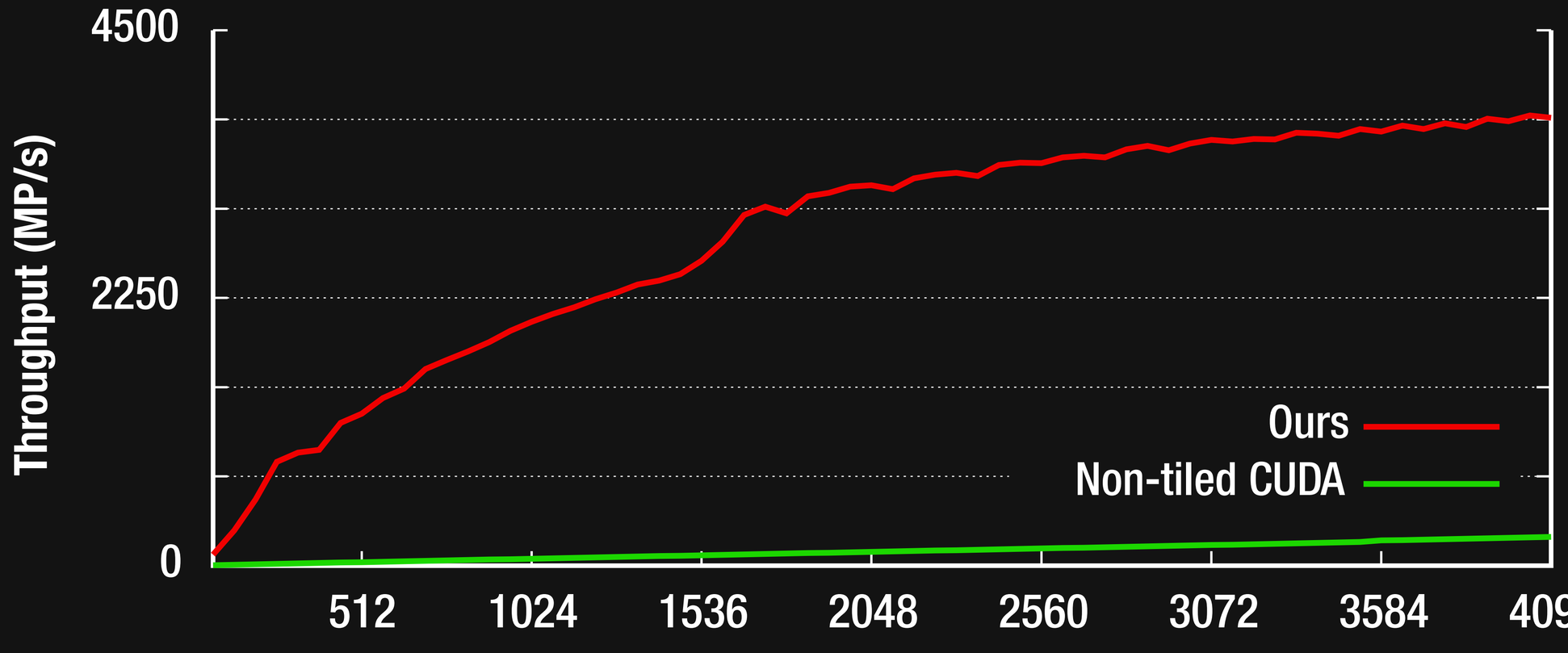
[HPG 2015]

with Chaurasia, Drettakis, Paris, Ragan-Kelley

Linear recursive filters

parallelism

locality



Reduction scheduling [Suriana et al. CGO 2017]

State-of-the-art results on CNNs

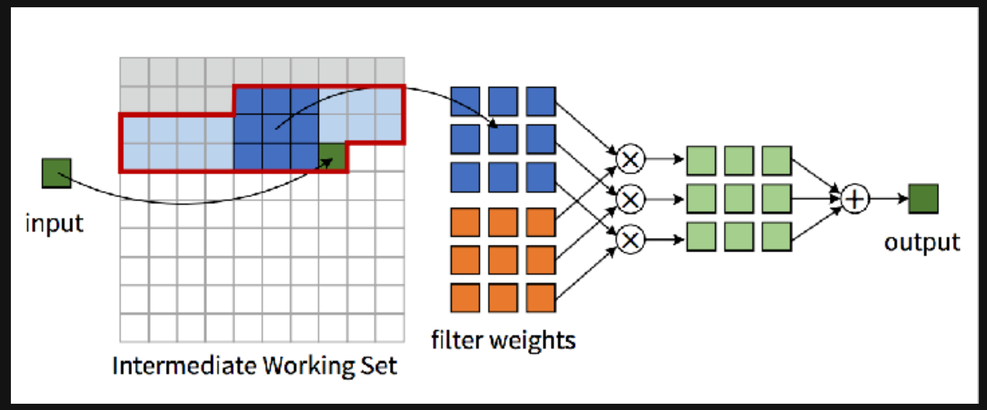
(on ARM, x86, and Hexagon DSP especially).

Matches or beats commercial hand-tuned inner kernels with a few lines of code.

Compilation to FPGA [Pu et al.]

compiling Halide directly to ARM+FPGA SoC.

TACO 2017 paper



Real-world adoption

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

Adobe

Google

Google

> 200 pipelines

10s of kLOC in production

manual schedules

YouTube

Halide touches every video uploaded.

65B frames/day

Should you use it?

Yes if your problem is deep, wide and your data structures and computations are regular

Images, volumes, arrays in general
Less clear for iterative linear stencil methods (not deep)

Check out SimIt for less regular data/computation (e.g. physical simulation on meshes)

Some of the benefits of Halide

Keeps algorithm clean and orthogonal to schedule

Systematic organization of scheduling/performance

Automatically does low-level stuff for you

indexing logic, even when image is not divisible by the tile size
tile expansion inference
vectorization, translation to CUDA

Enables quick exploration of possible schedules

How skilled do you need to be?

No need to understand vector unit assembly

Some mental model of what performance entails

I was able to write Halide code and get a 10-20x speedup

And I have never written SIMD or GPU code

How to get started?

http://halide-lang.org/

Good tutorials

http://stellar.mit.edu/S/course/6/fa15/6.815/

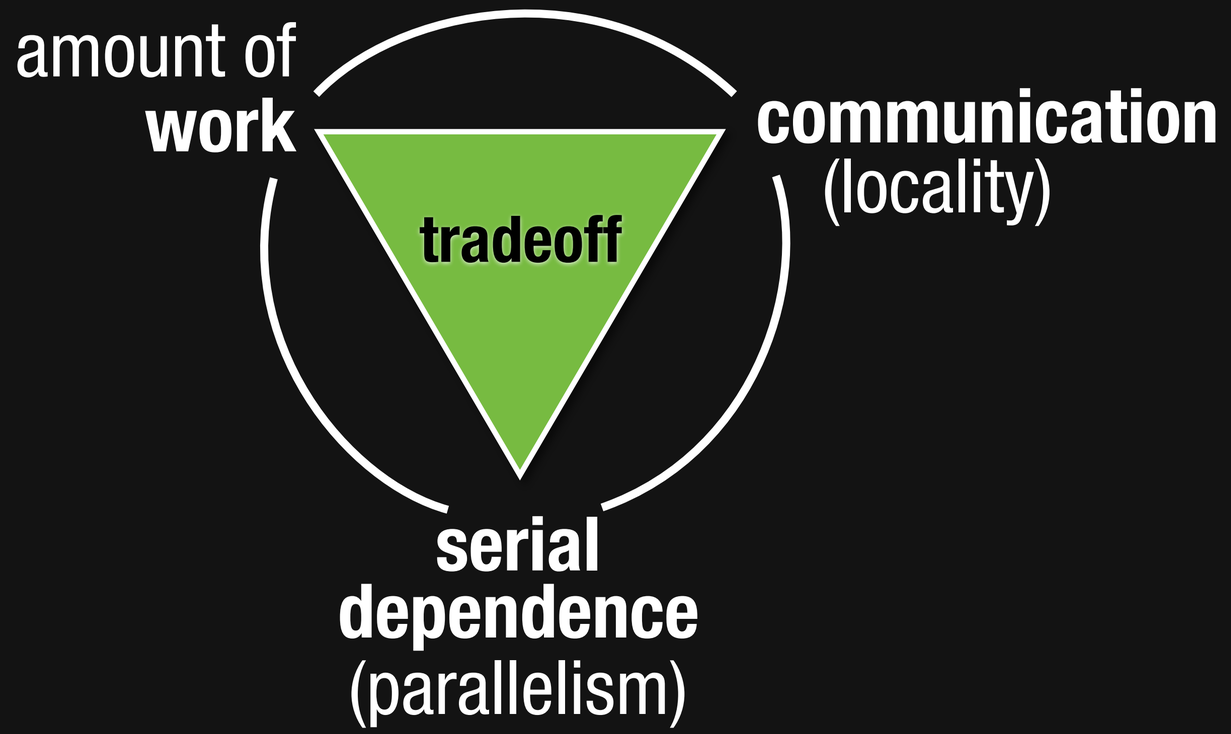
Organization of computation is a first-class issue

Program:

Algorithm

Organization of computation

Hardware



Domain specific languages are powerful

Own and understand data structure and dependence

Make organization of computation orthogonal

Conciseness (secondary)

Can ensure correctness

Thank you!

Acknowledgements

Jonathan Ragan-Kelley

Andrew Adams

Connelly Barnes

Sylvain Paris

Marc Levoy

Saman Amarasinghe

Patricia Suriana

Shoaib Kamil,

Ravi Mullapudi

Shoaib Kamil

Dillon Sharlet

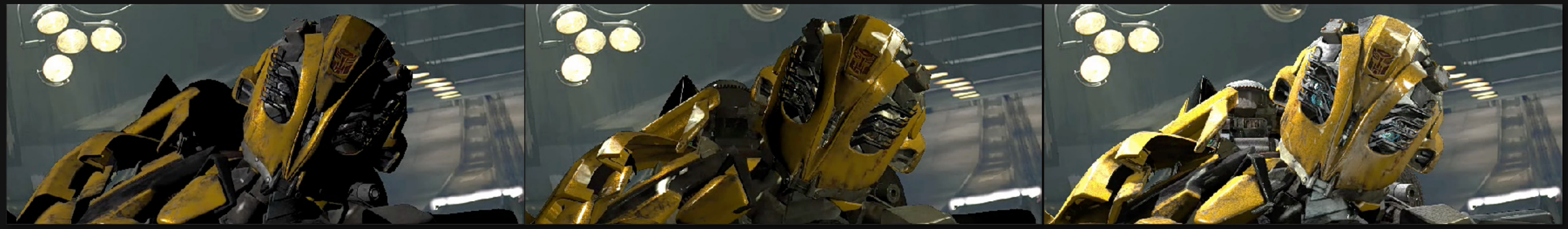
Kayvon Fatahalian

Zalman Stern

Gaurav Chaurasia

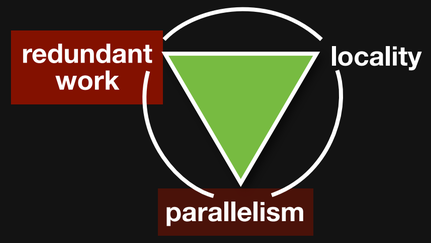
George Drettakis

The **Lightspeed** Automatic Interactive Lighting Preview System
[SIGGRAPH 2007]



Algorithm
Organization
Hardware

Most computations & data are redundant



Our **compiler** automatically **reorganizes** and **parallelizes**, based on analysis of **dependencies**

Accelerate previews **10,000x**
interactive, instead of hours

The **Lightspeed** Automatic Interactive Lighting Preview System

[SIGGRAPH 2007]

INDUSTRIAL LIGHT & MAGIC

LUCASFILM Ltd

Used by dozens of artists on over 16 films, including:

- Transformers 2
- Rango
- Terminator 4
- Star Trek
- Iron Man 2
- Star Tours
- Spiderwick Chronicles
- Avatar
- Indiana Jones 4
- Transformers 3
- Battleship
- Red Tails
- Cowboys & Aliens
- The Avengers

Decoupled Sampling for Graphics Pipelines

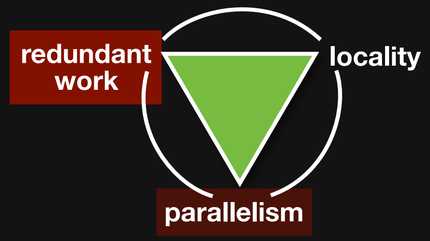
[ACM TOG 30(3), *Presented at SIGGRAPH 2011*]

Shading every visible sample is **redundant**.

Visibility and shading have complicated **dependence**.

Extend **Direct3D** graphics pipeline to capture **reuse** by tracking **dependence**.

Algorithm
Organization
Hardware



Tested real games in full GPU simulator.
3-40x less shading, similar quality.

A Hierarchical **Volumetric Shadow** Algorithm for Single Scattering

[SIGGRAPH Asia 2010]

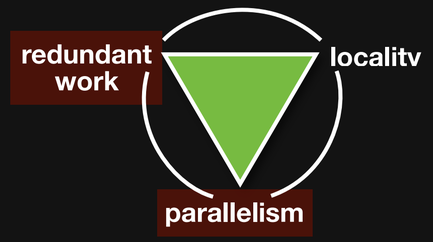
Incremental integration along epipolar slices using partial sum trees

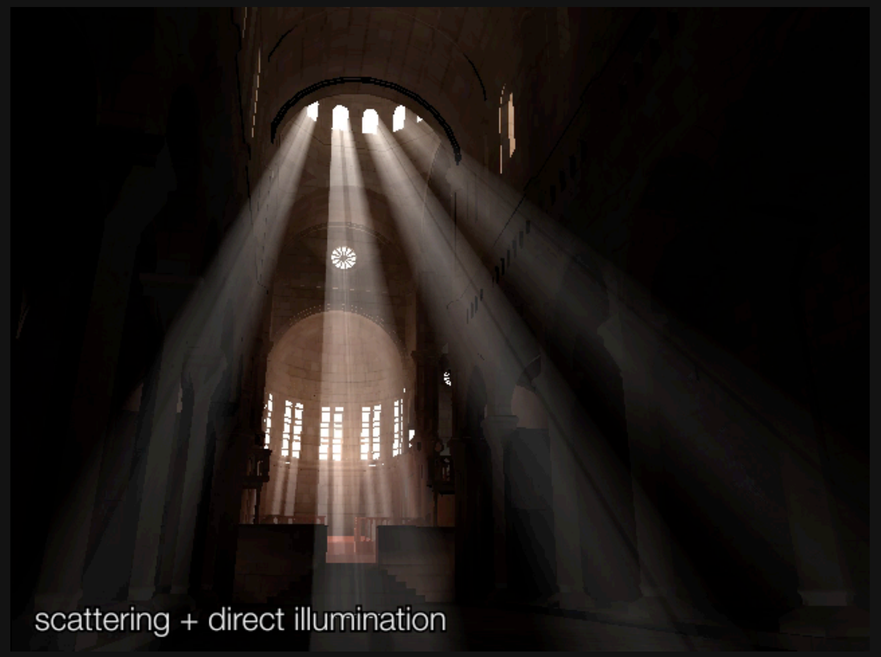
asymptotically faster than prior algorithms

Algorithm

Organization

Hardware





17-120x acceleration.
Ray marching quality, in real-time.