Halide

a language and compiler for high performance image processing

CS448h
Oct. 20, 2015

We are surrounded by computational cameras

Enormous opportunity,
demands extreme optimization

parallelism & locality limit
performance and energy

We are surrounded by computational cameras

Enormous opportunity,
demands extreme optimization

parallelism & locality limit
performance and energy

Camera: 12 Mpixels
(144MB/frame as *float*)

CPUs: 15 GFLOP/sec

GPU: 115 GFLOP/sec

We are surrounded by computational cameras

Enormous opportunity,
demands extreme optimization

parallelism & locality limit
performance and energy

Camera: 12 Mpixels
(144MB/frame as *float*)

CPUs: 15 GFLOP/sec

GPU: 115 GFLOP/sec

Required arithmetic intensity > 40:1

Today’s methodology

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

Today's methodology

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

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

Today’s methodology

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

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

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

Key challenge: reorganize computations & data

Simpler programs

Order of magnitude faster

Scalable on future architectures

Simpler, Faster, Scalable

Reference: 300 lines C++

Adobe: 1500 lines

3 months of work

10x faster (vs. reference)

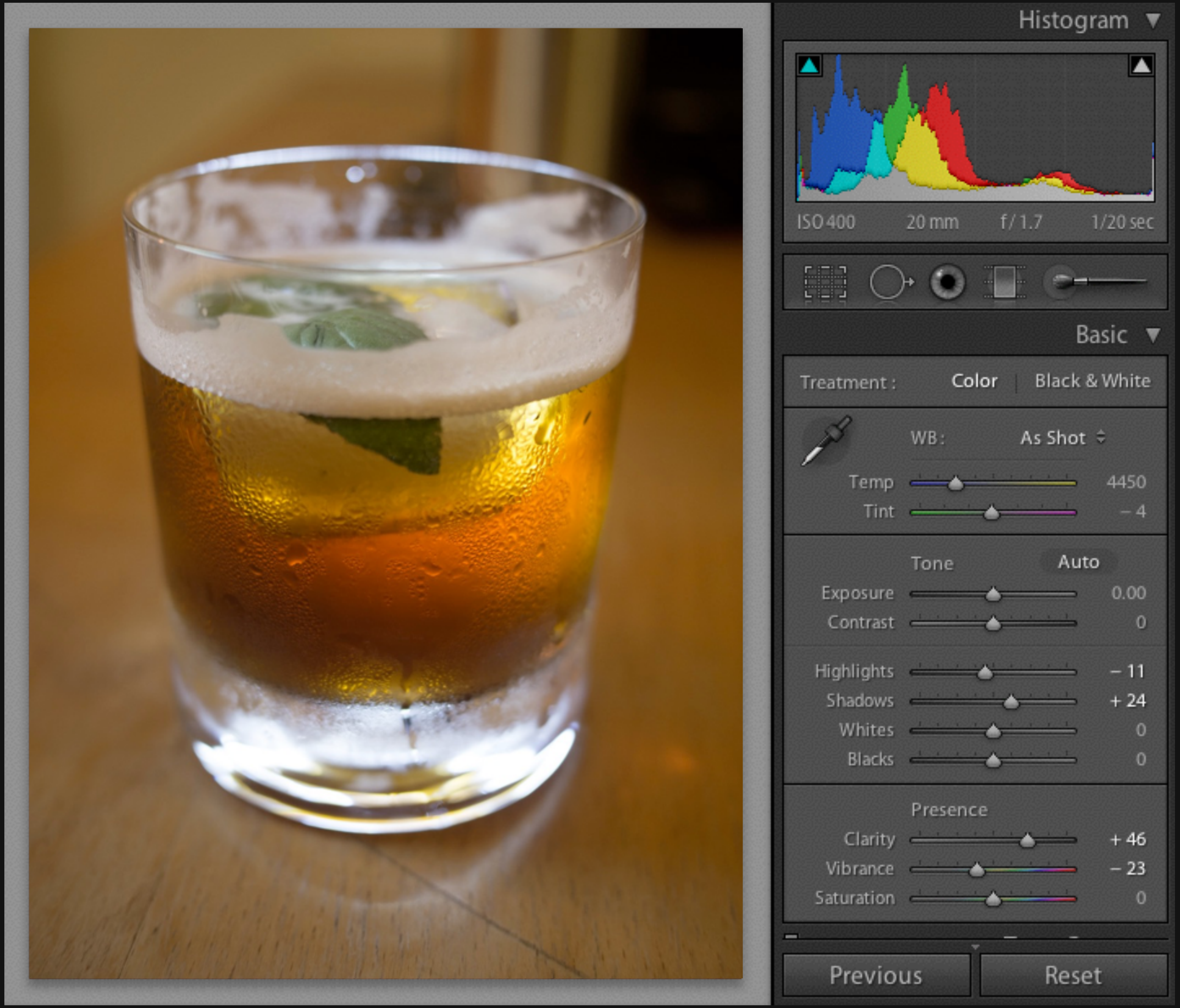
Halide: 60 lines

1 intern-day

20x faster (vs. reference)

2x faster (vs. Adobe)

GPU: 90x faster (vs. reference)



Simpler, Faster, Scalable

Reference: 300 lines C++

Adobe: 1500 lines

3 months of work

10x faster (vs. reference)

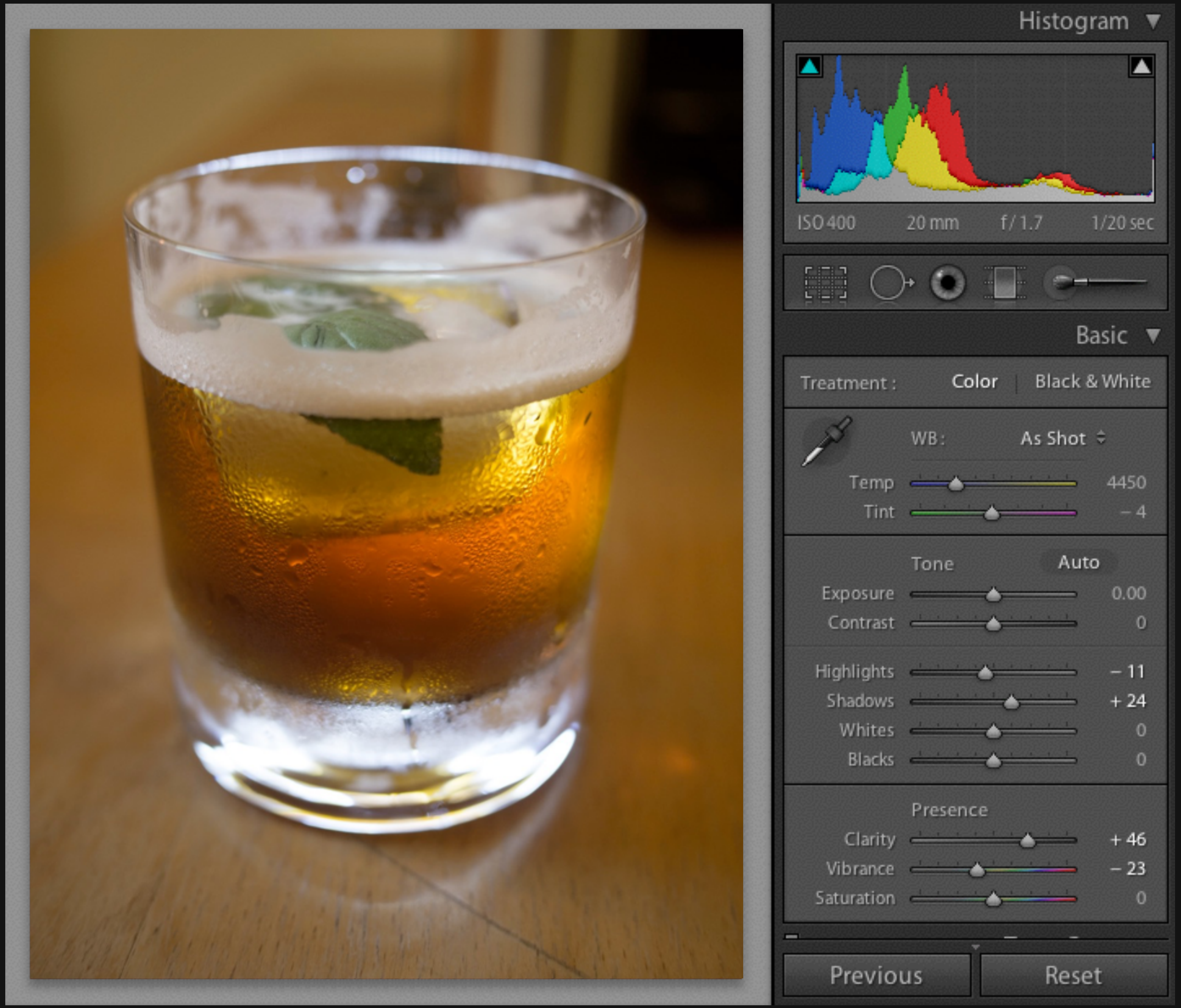
Halide: 60 lines

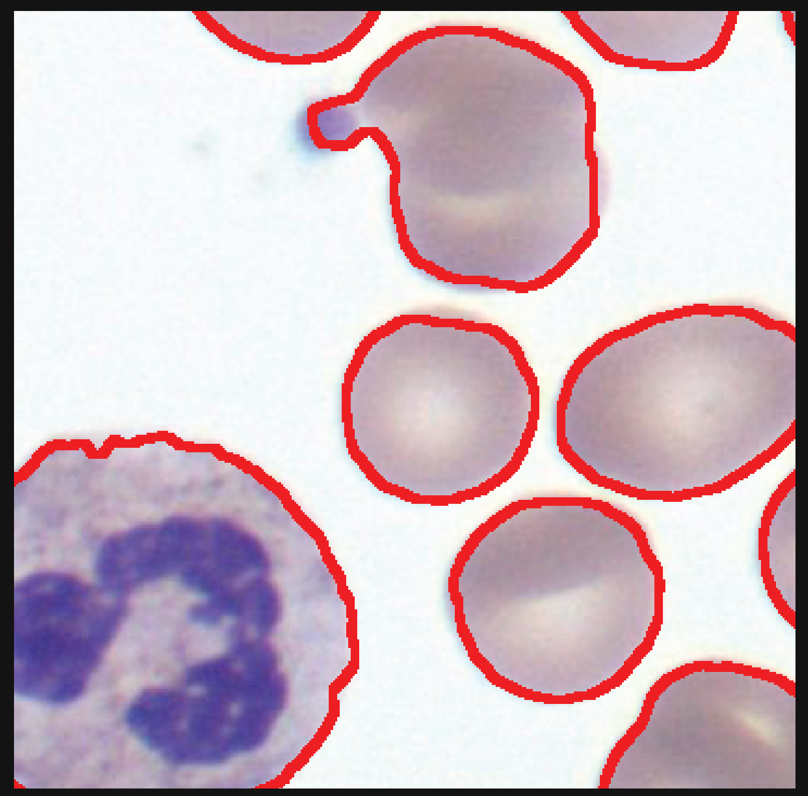
1 intern-day

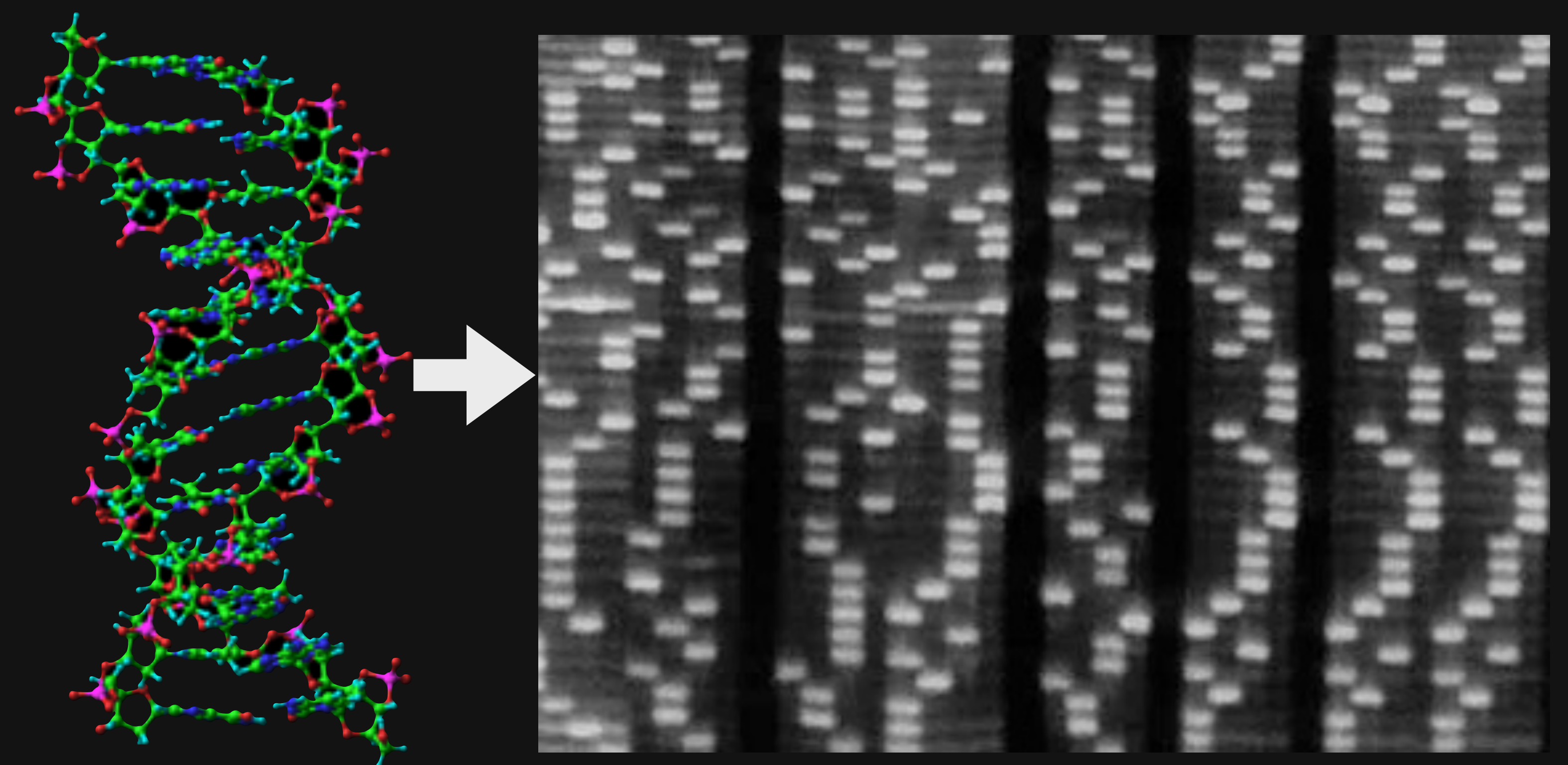
20x faster (vs. reference)

2x faster (vs. Adobe)

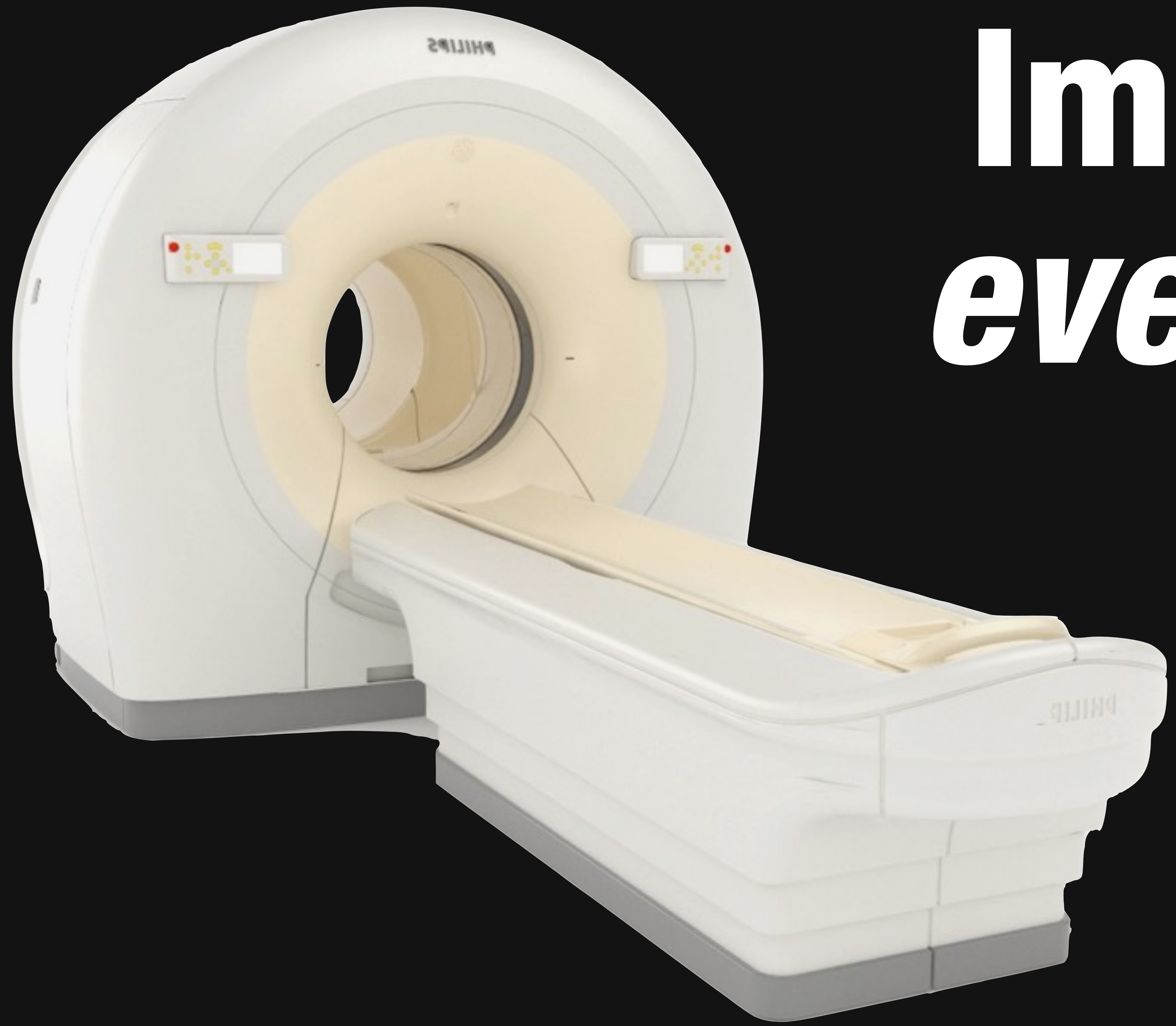
GPU: 90x faster (vs. reference)

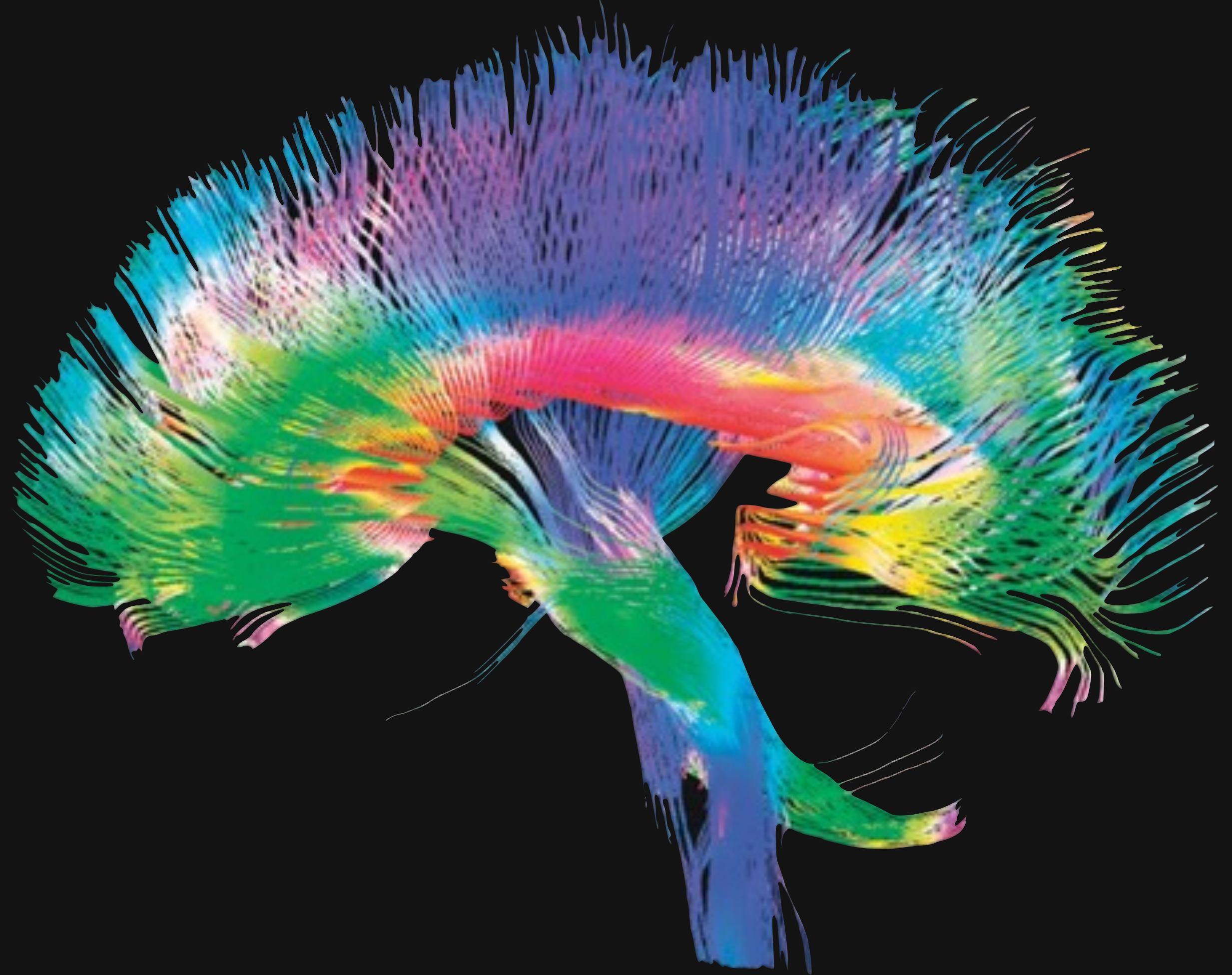


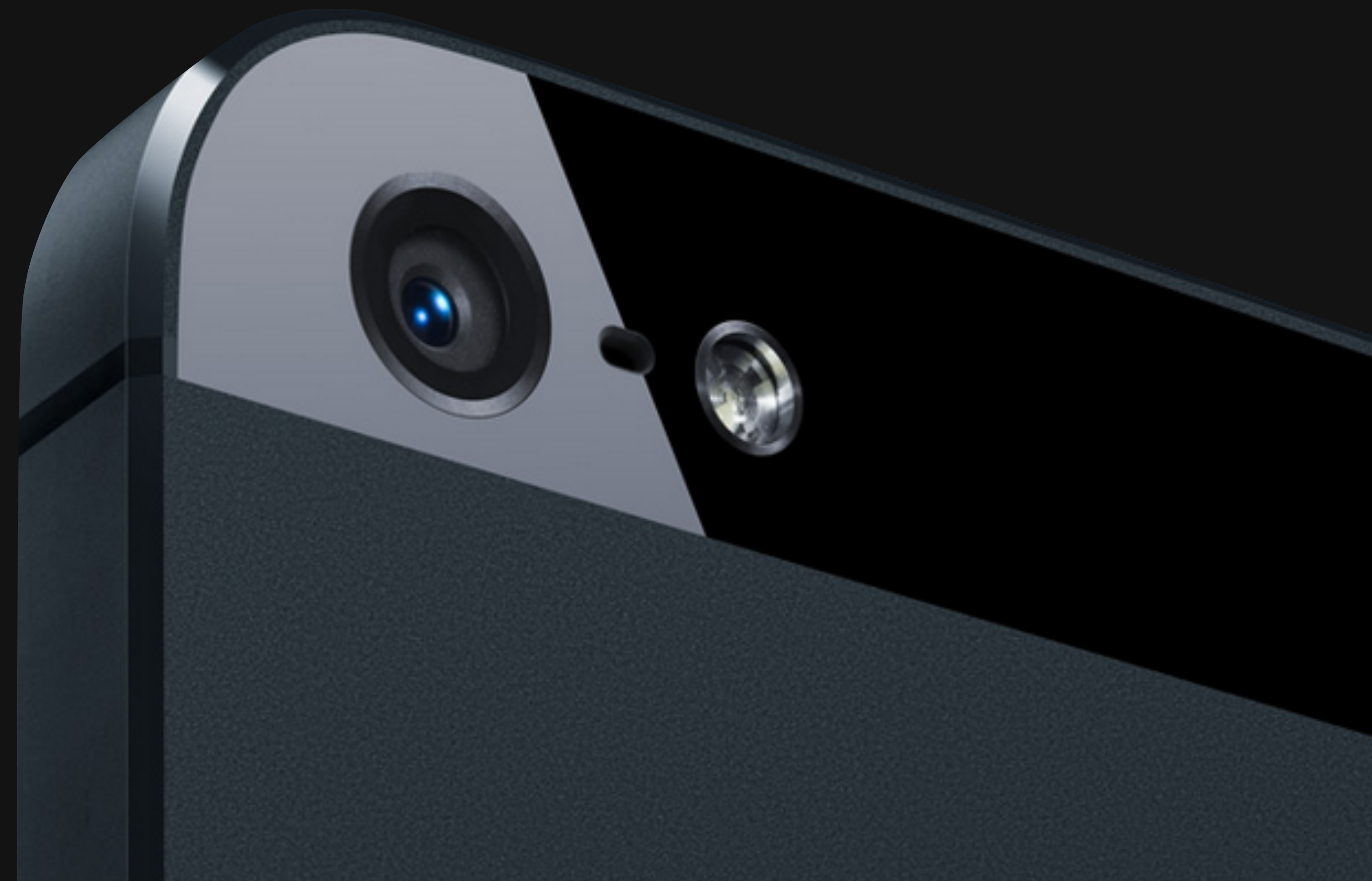




Imaging is *everywhere*







How can we scale image processing computation?

How can we scale image processing computation?

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

How can we scale image processing computation?

Parallelism
“Moore’s law” growth will require exponentially more parallelism.

Locality
Data should move as little as possible.

How can we scale image processing computation?

Parallelism
"Moore's law" growth will require 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

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

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*

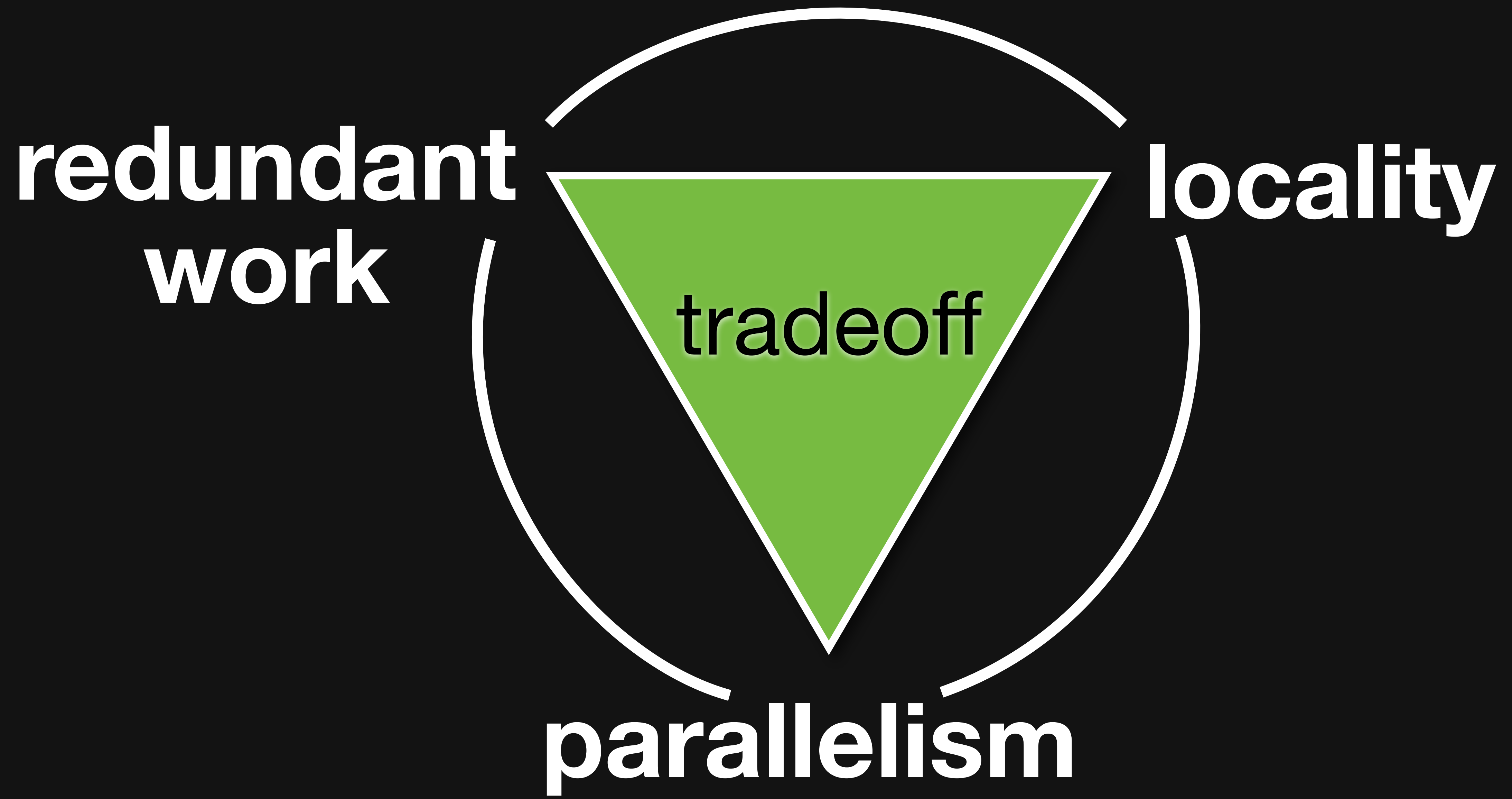
data from John Brunhaver, Bill Dally, Mark Horowitz

Communication dominates computation in both energy and time

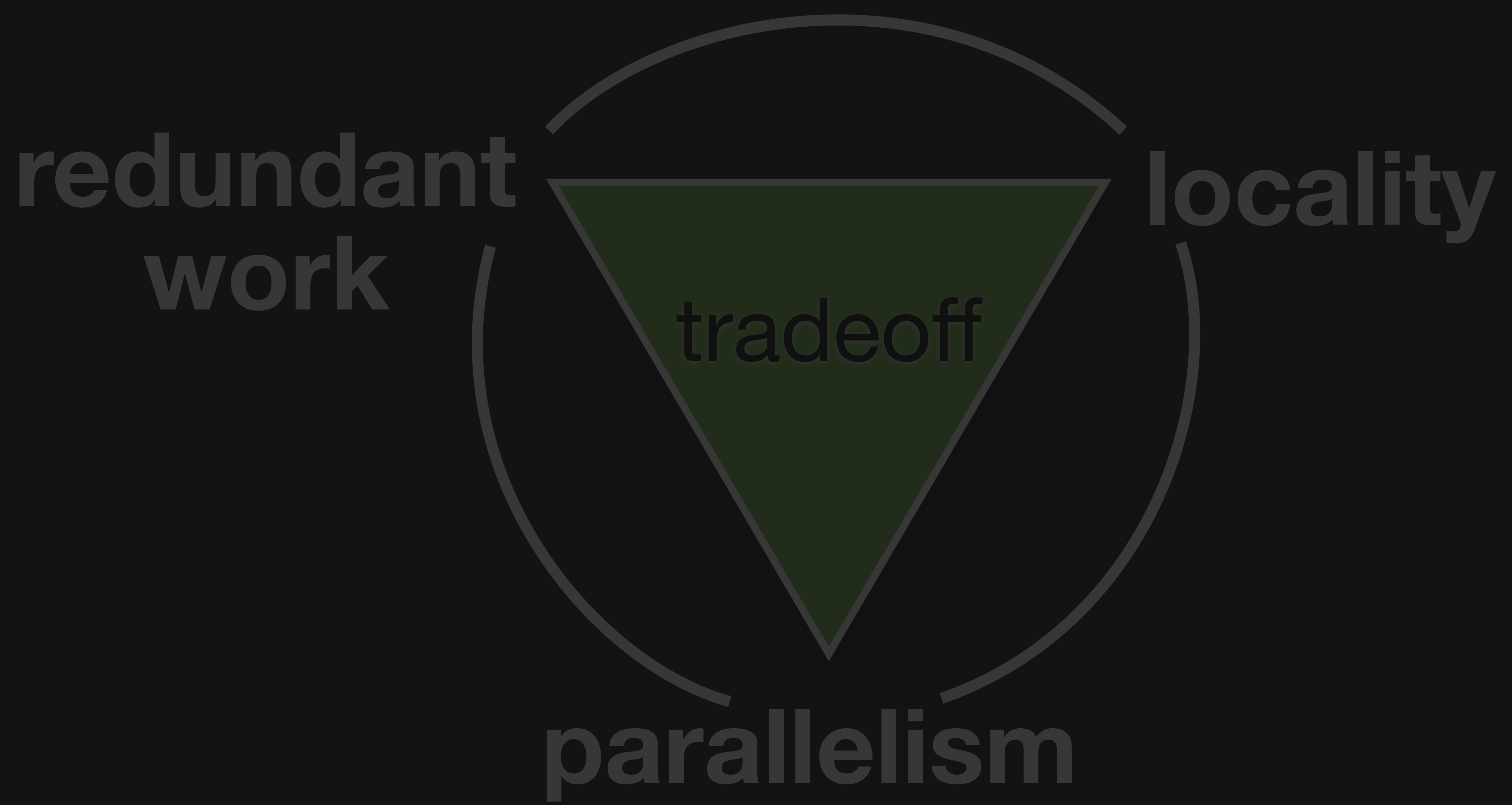
Operation (32-bit operands)	Energy/Op (28 nm)	*Cost* *(vs. ALU)*
ALU op	1 pJ	-
Load from SRAM	5 pJ	*5x*
Move 10mm on-chip	32 pJ	*32x*
Send off-chip	500 pJ	*500x*
Send to DRAM	1 nJ	*1,000x*
Send over LTE	>50 μJ	*50,000,000x*

data from John Brunhaver, Bill Dally, Mark Horowitz

Message #1: Performance requires complex tradeoffs



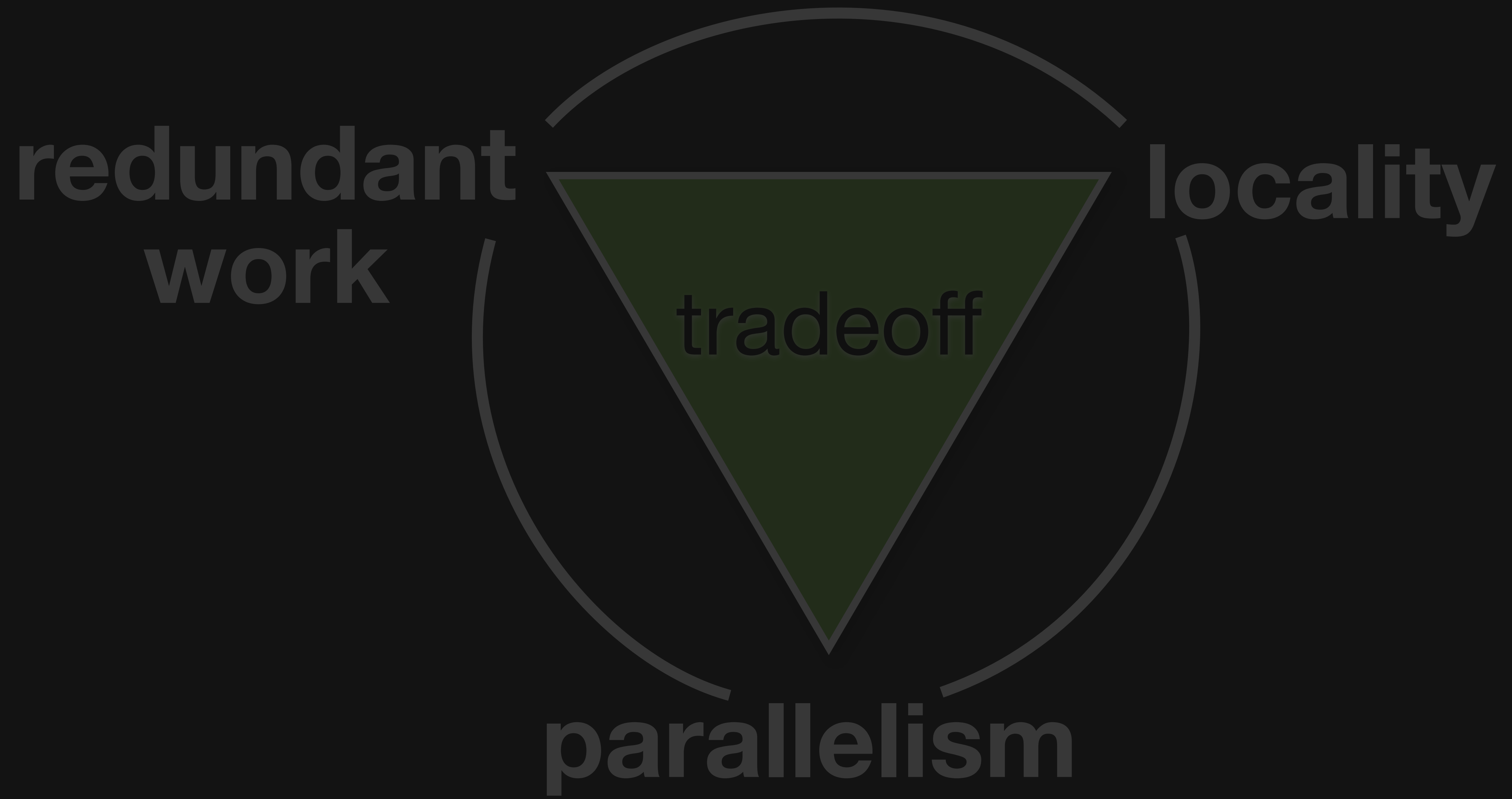
Where does performance come from?



Where does performance come from?

Program

Hardware

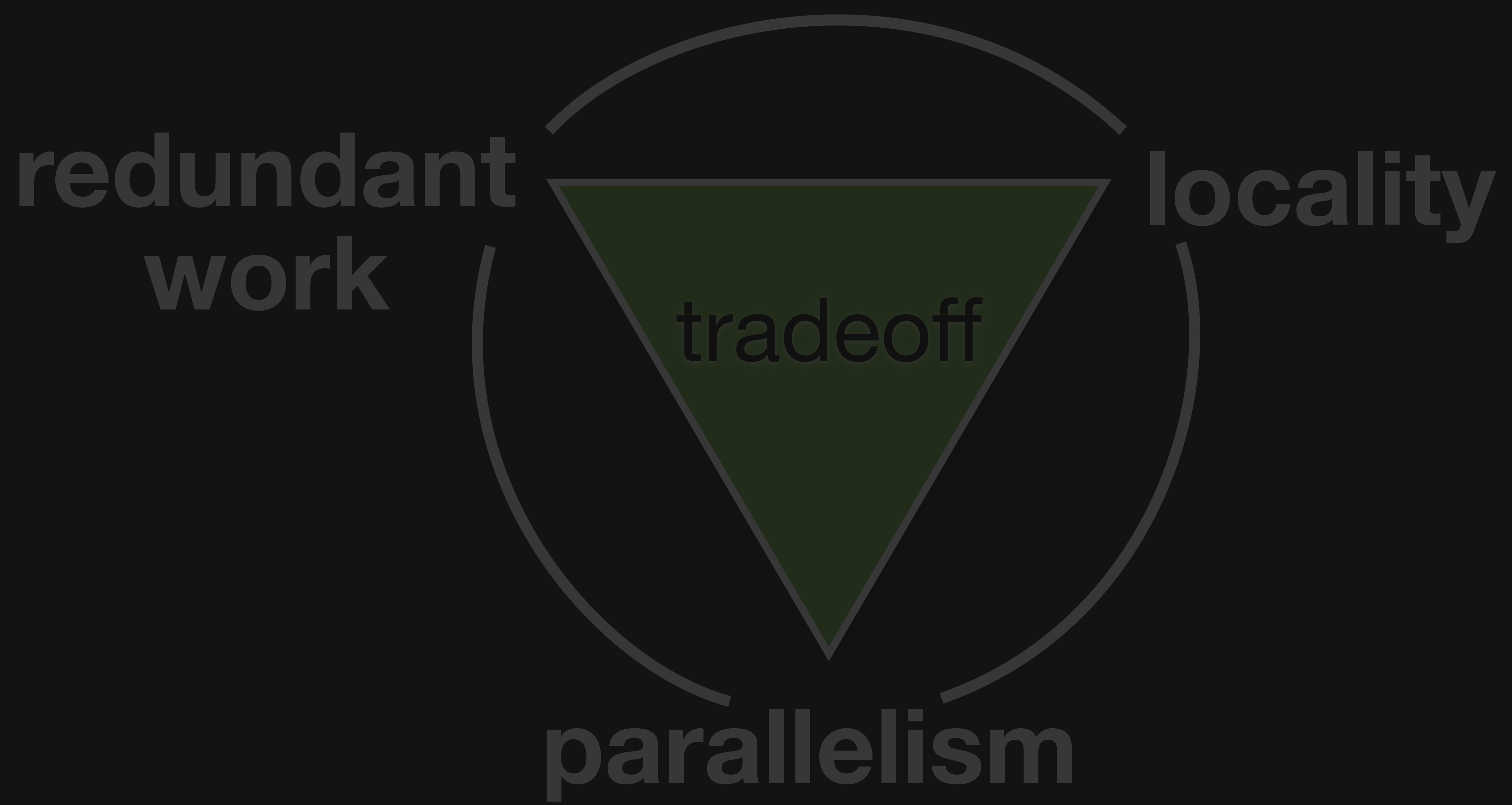


Message #2: organization of computation
is a first-class issue

Program:

Program

Hardware



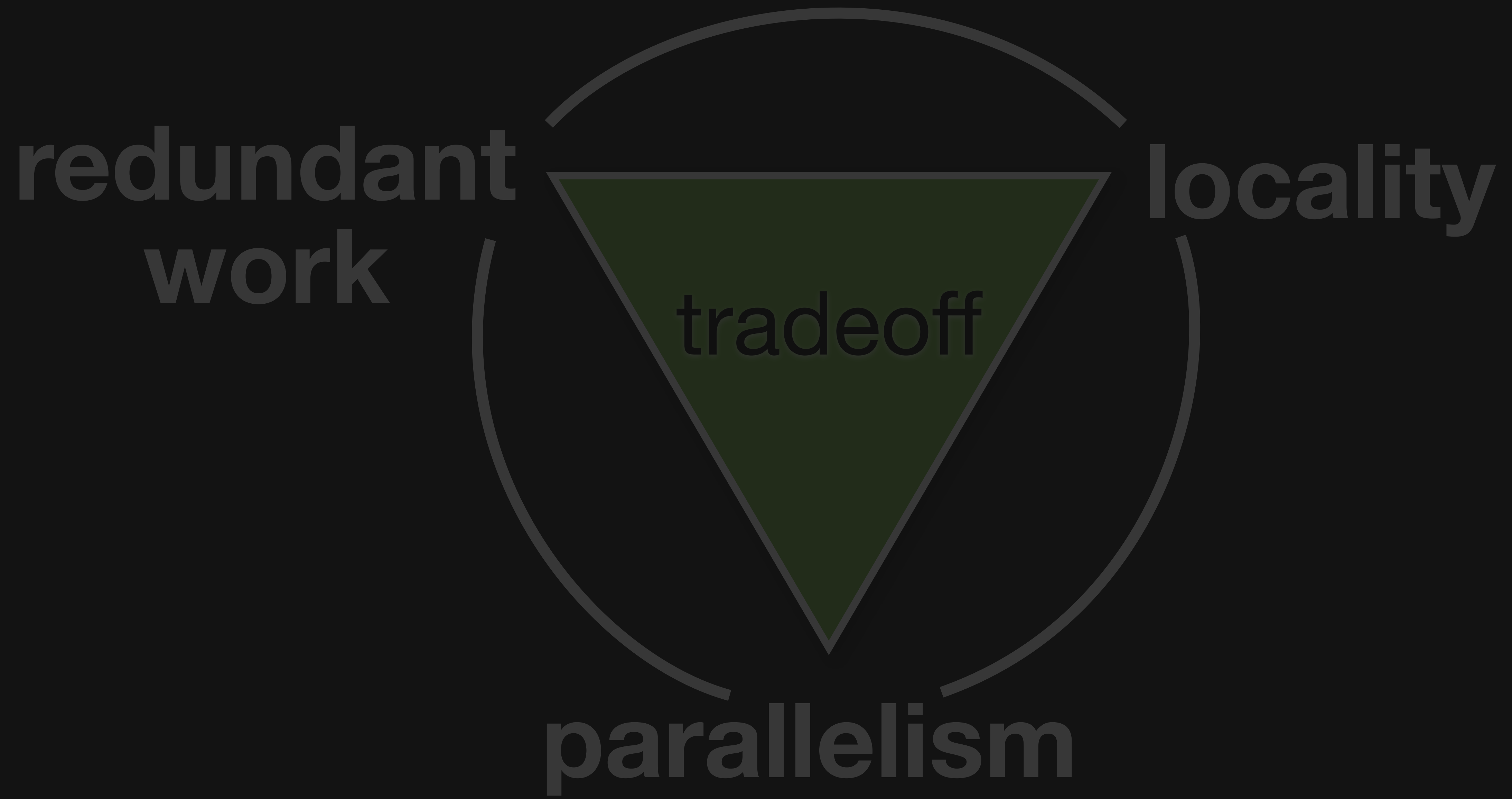
Message #2: organization of computation
is a first-class issue

Program:

Algorithm

Organization of computation

Hardware



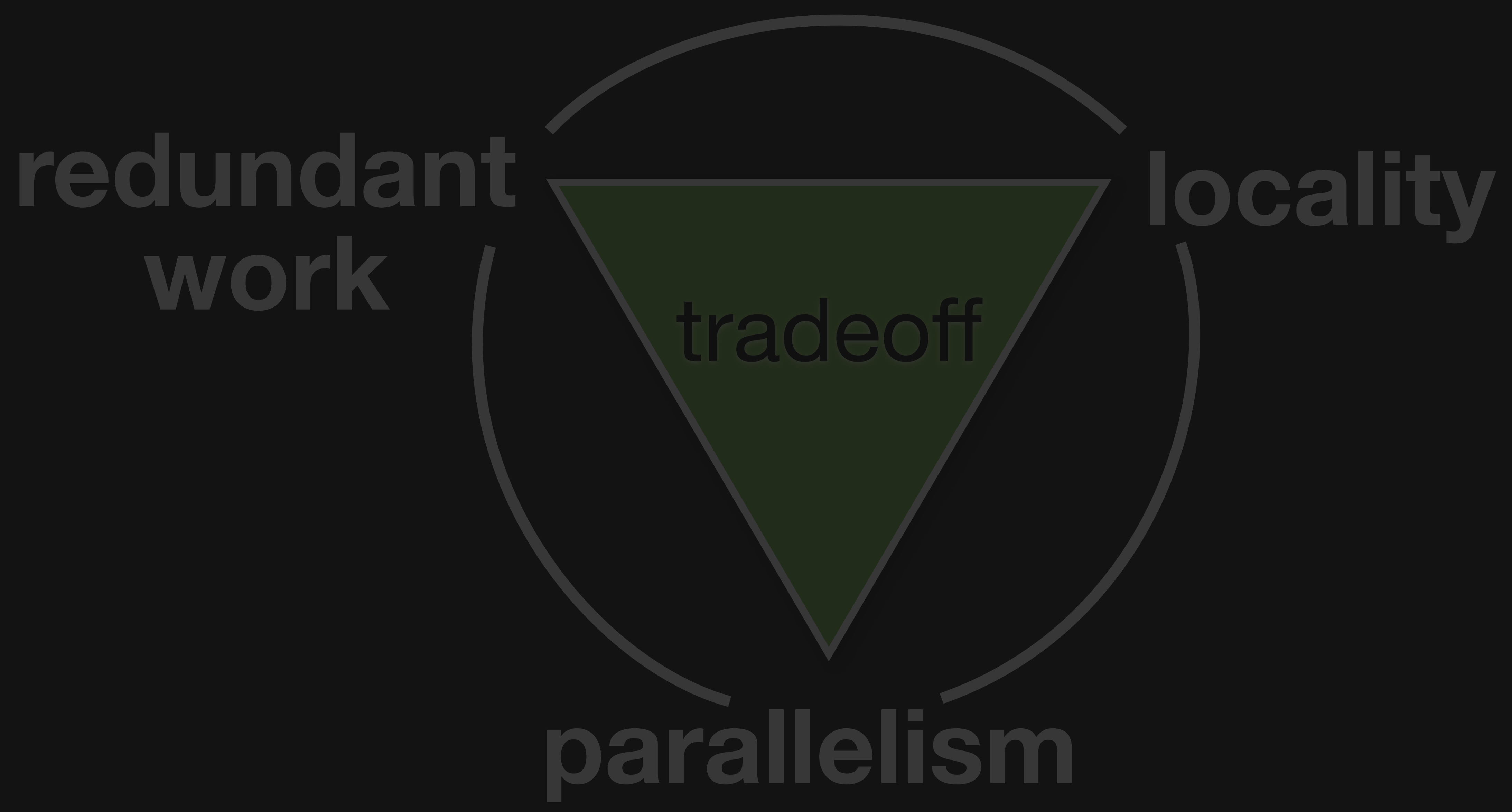
Message #2: organization of computation
is a first-class issue

Program:

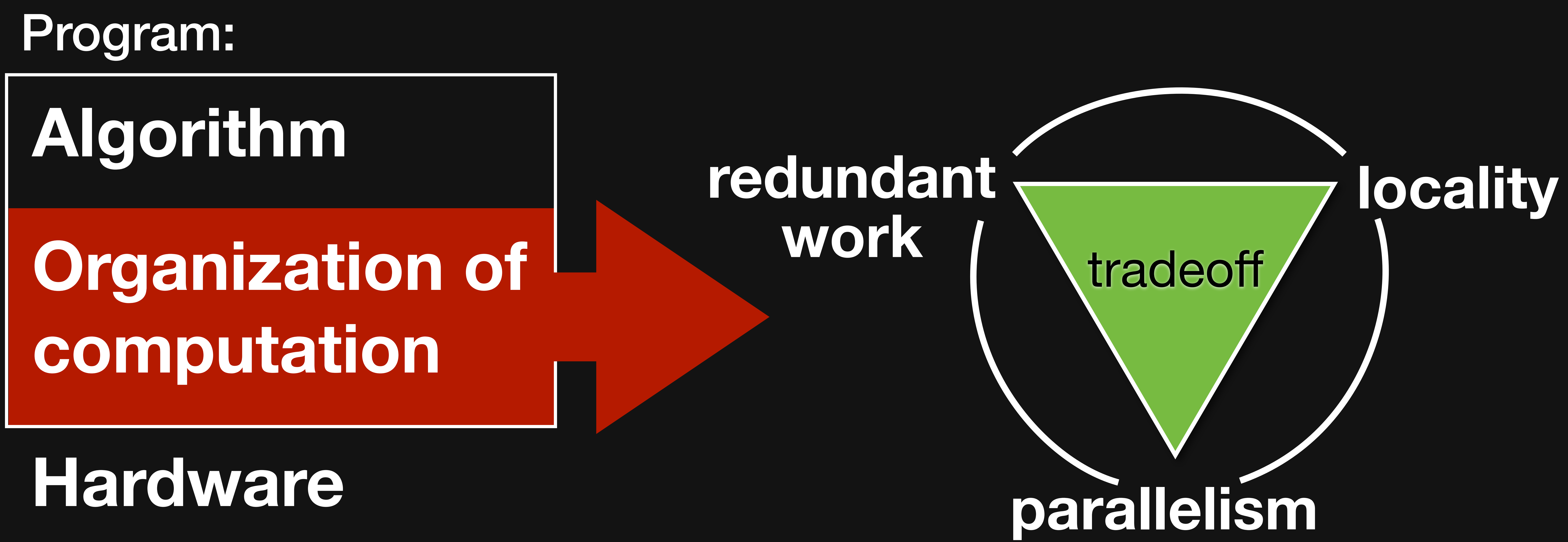
Algorithm

Organization of computation

Hardware



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



Halide

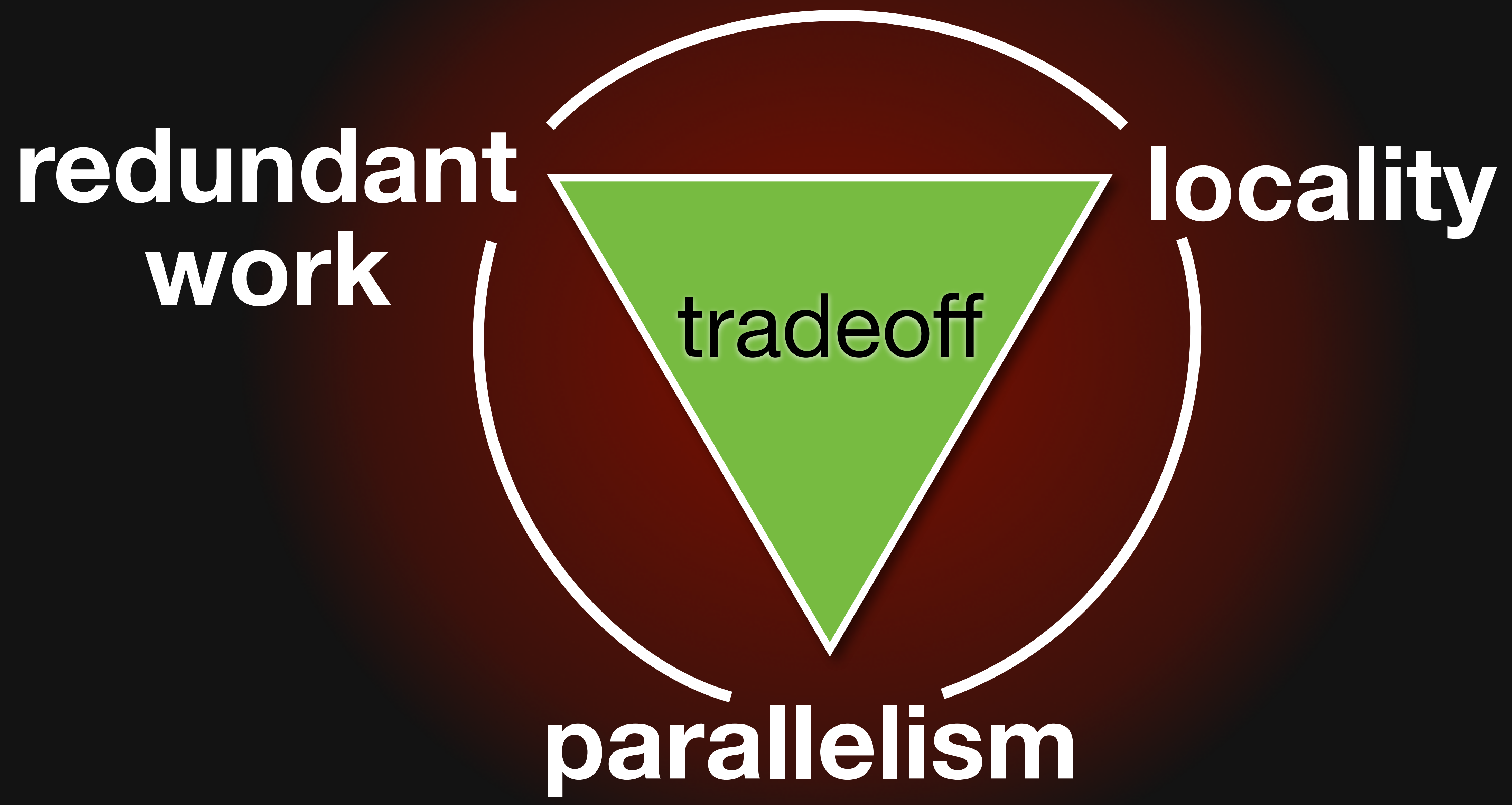
a language and compiler for image processing

[SIGGRAPH 2012, PLDI 2013]
joint work with Andrew Adams, et al.

Algorithm

Organization of computation

Hardware



Algorithm vs. Organization: 3x3 blur

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

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

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

Algorithm vs. Organization: 3x3 blur

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

  for (int 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;
}
```

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

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*?

(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

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

Halide

a new language & compiler for image processing

1. Decouple *algorithm* from *schedule*

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

2. Single, unified model for *all* schedules

Halide

a new language & compiler for image processing

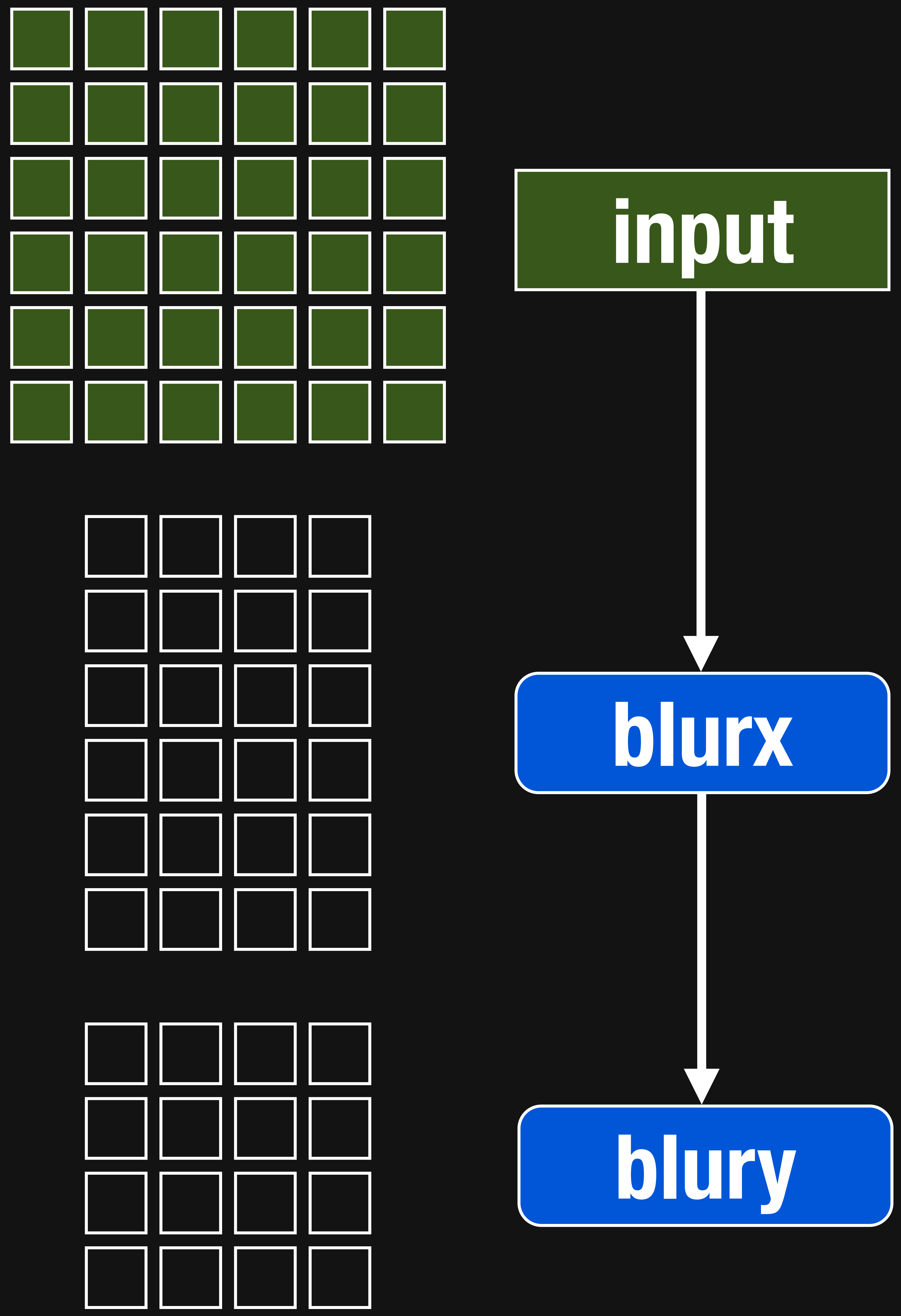
1. **Decouple *algorithm* from *schedule***

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

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

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

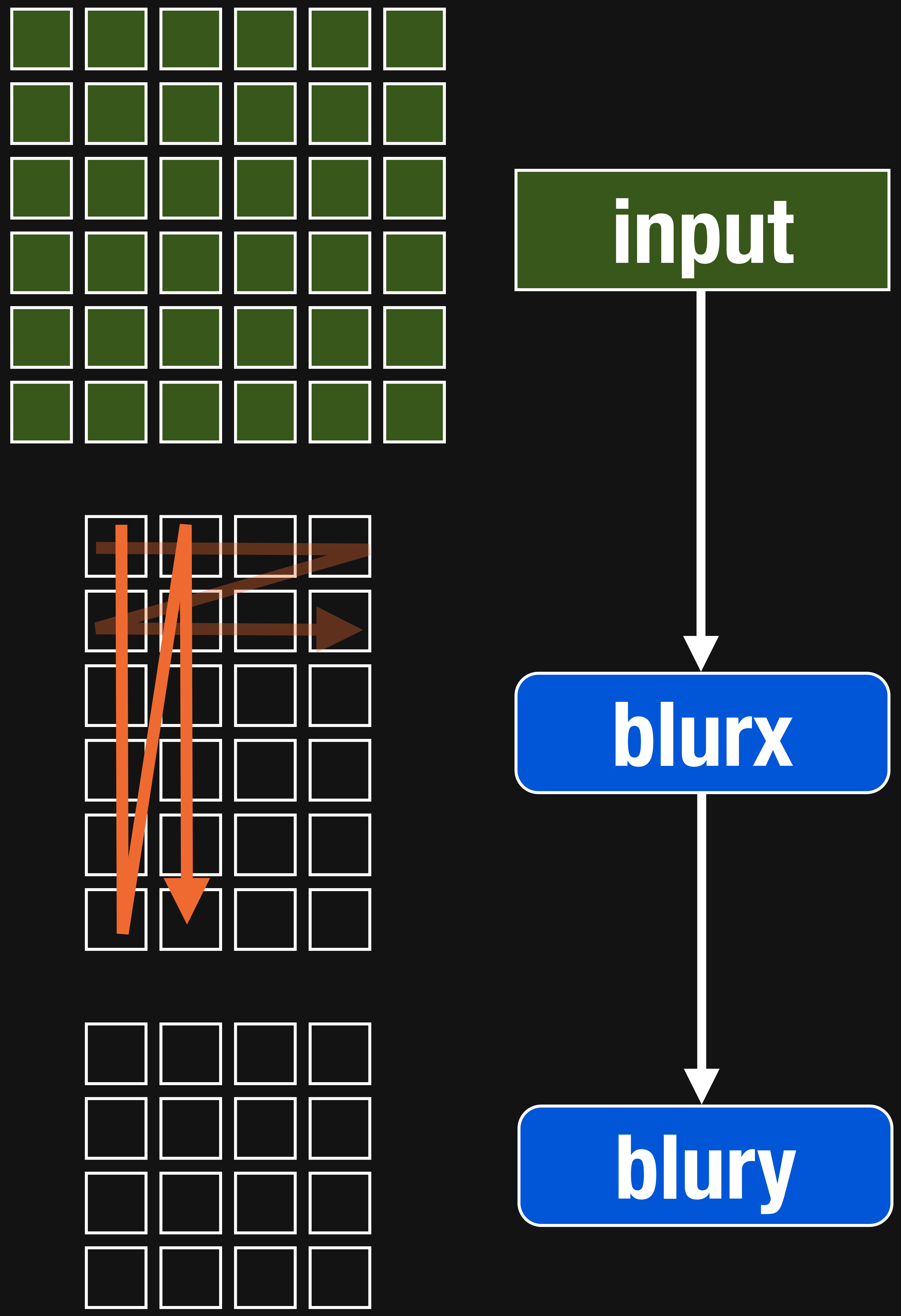
The schedule defines intra-stage order, inter-stage interleaving



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

For each stage:

1) In what order should we compute its values?

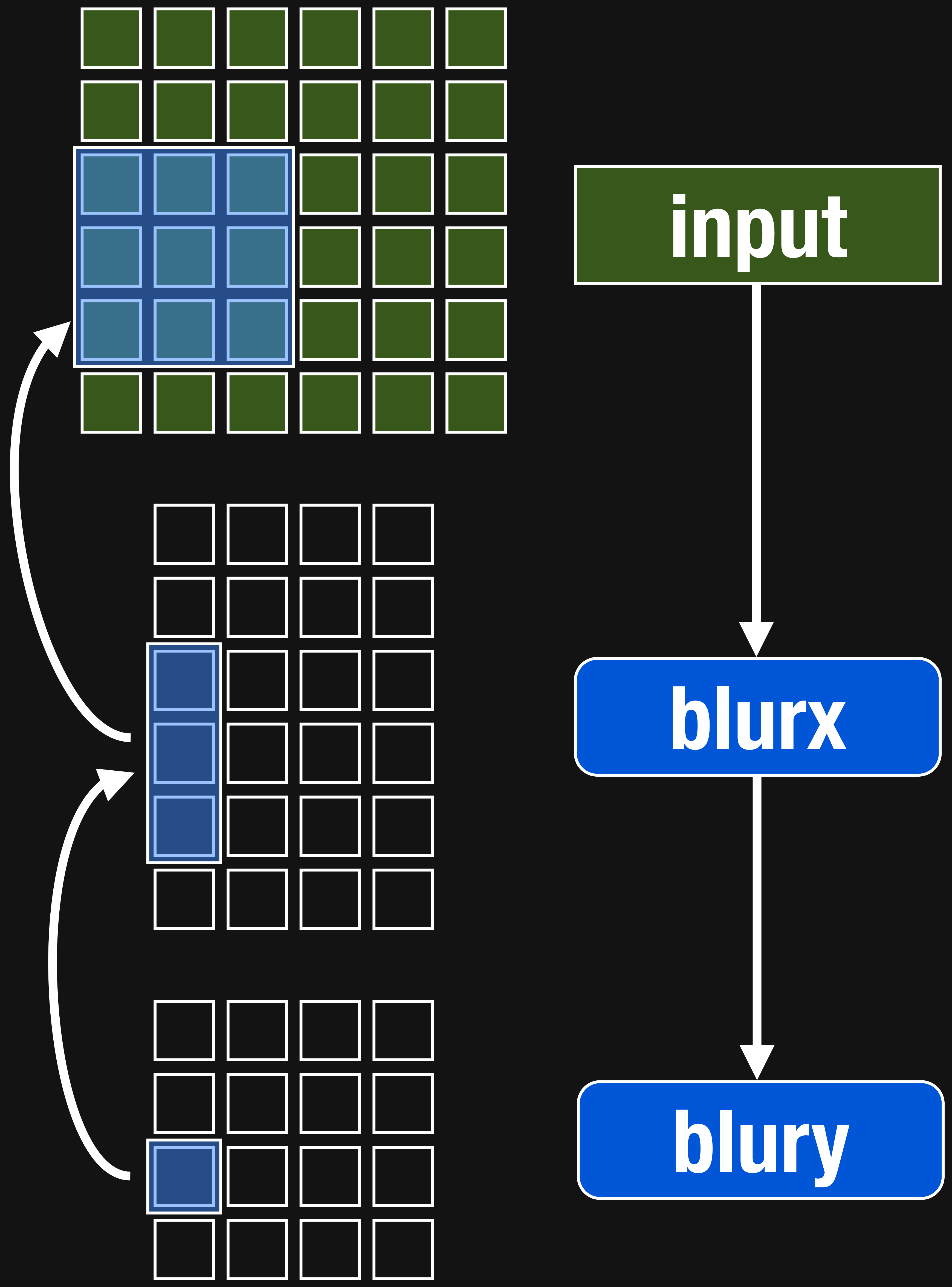


The schedule defines intra-stage order, inter-stage interleaving

For each stage:

1) In what order should we compute its values?

2) When should we compute its inputs?



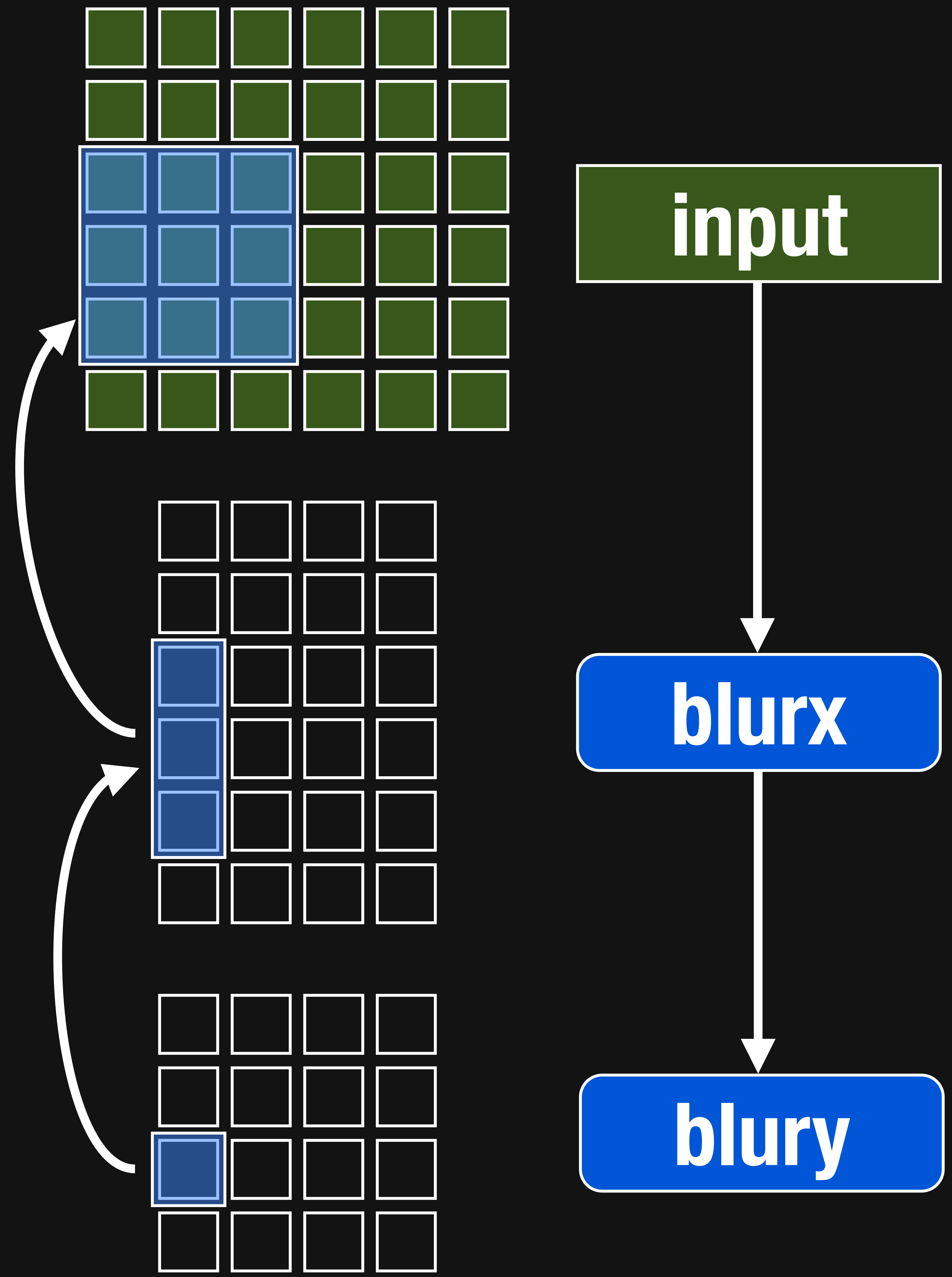
The schedule defines intra-stage order, inter-stage interleaving

For each stage:

1) In what order should we compute its values?

2) When should we compute its inputs?

This is a co-language for scheduling choices.



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

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;
```

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);
```

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);
```

```
// 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 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 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);
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
```

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).

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.

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

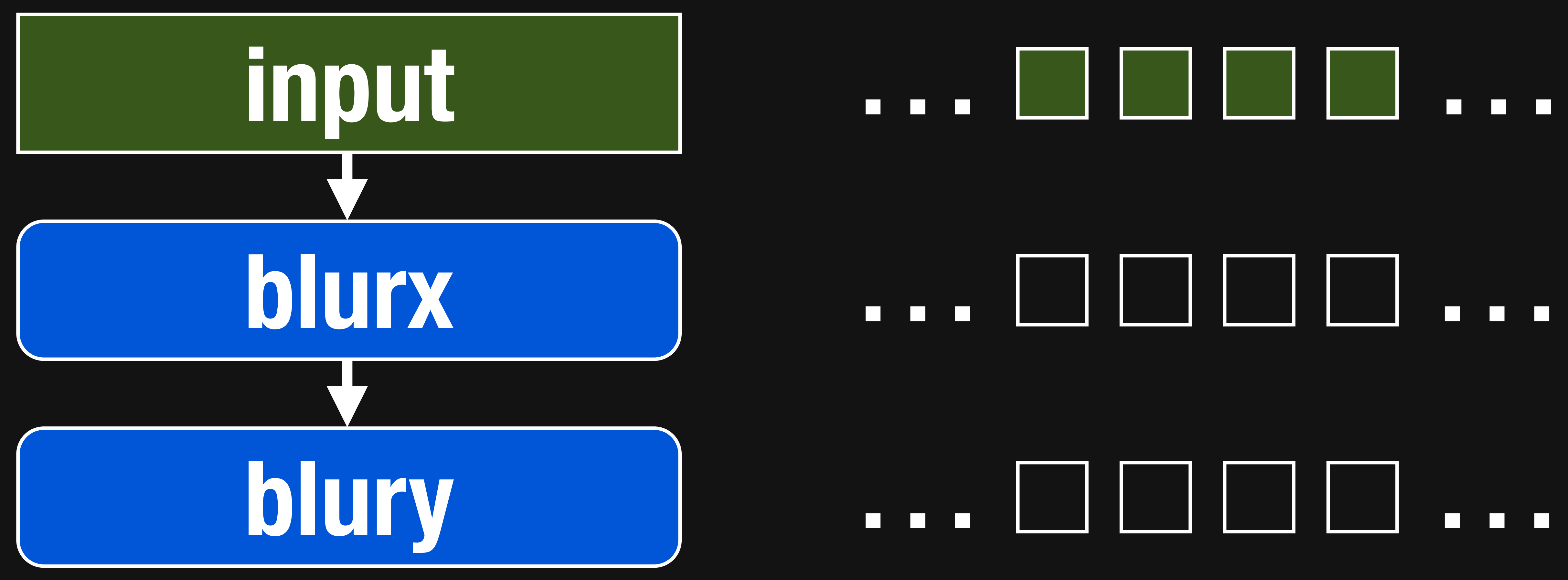
Roadmap

1. **Fundamental transformations**
   for stencil pipelines

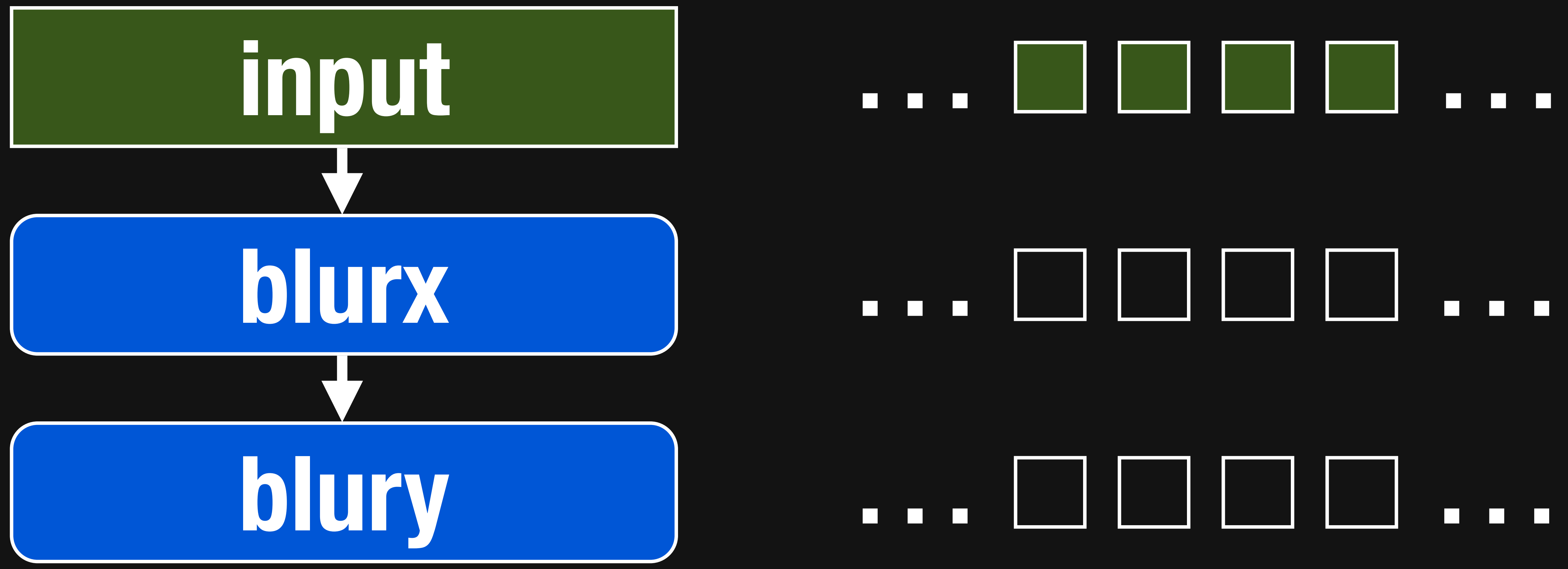
2. **Halide's unified model of scheduling**

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

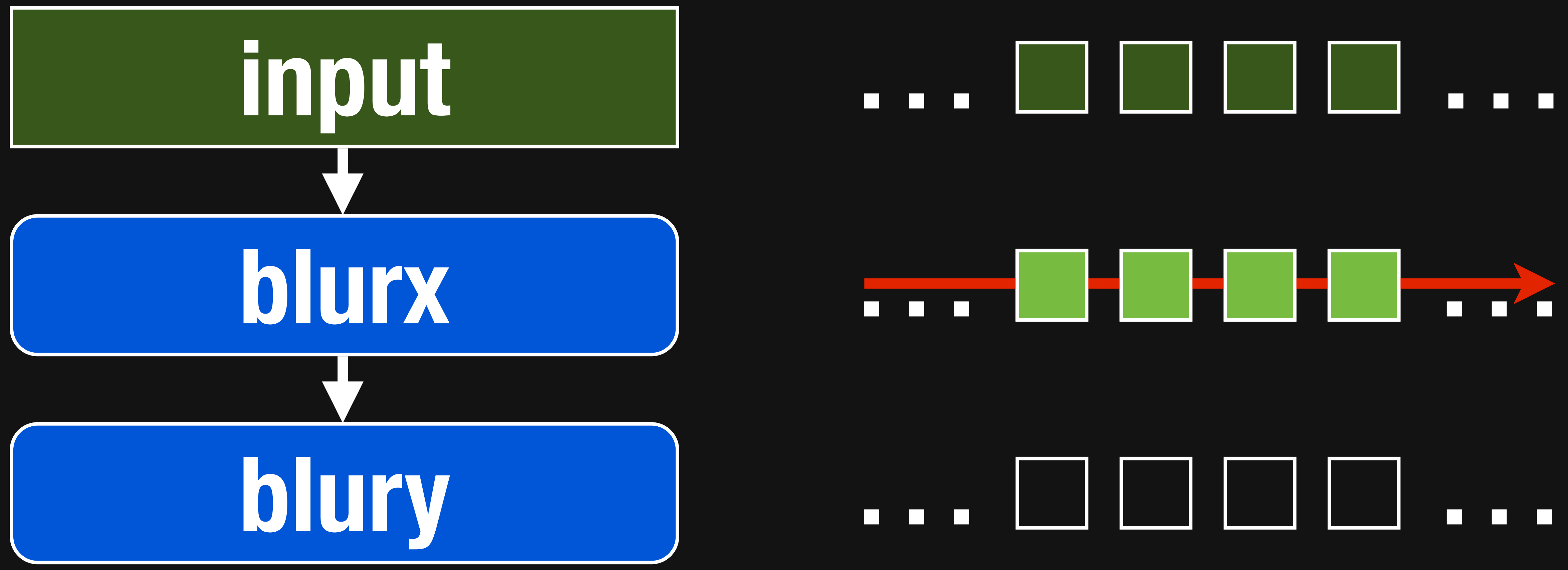
Organizing a data-parallel pipeline



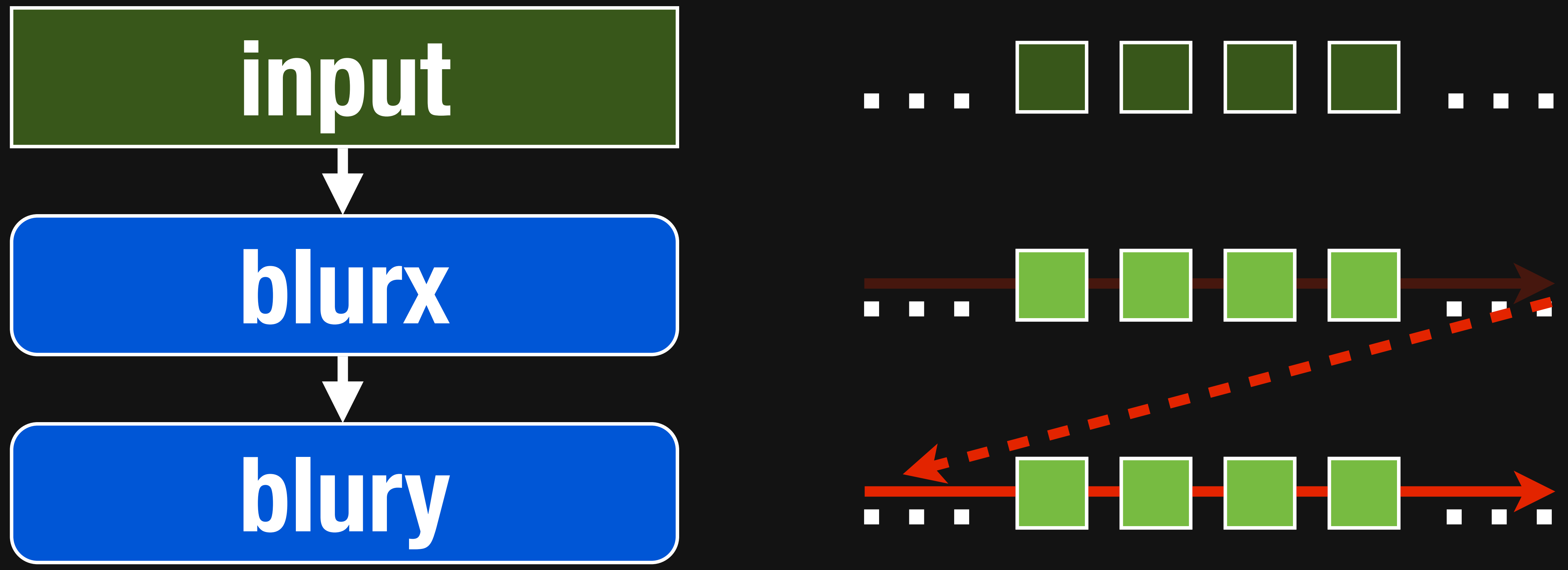
Simple loops execute **breadth-first** across stages



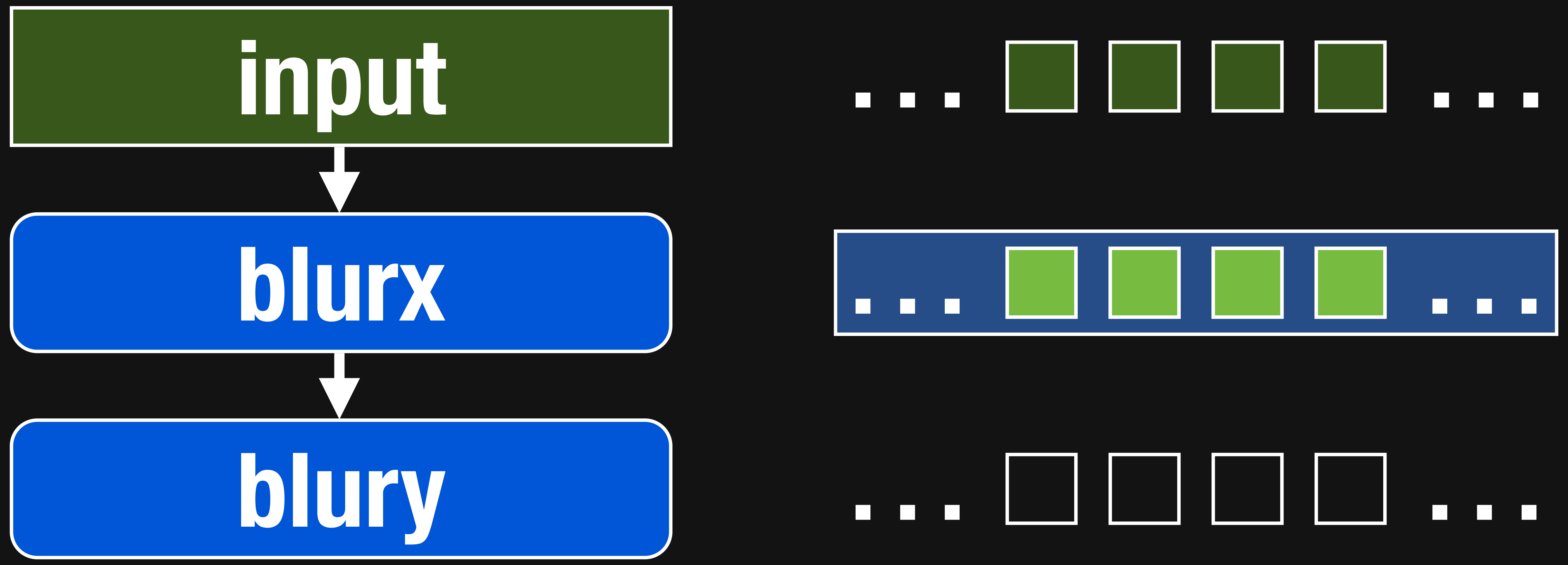
Simple loops execute **breadth-first** across stages



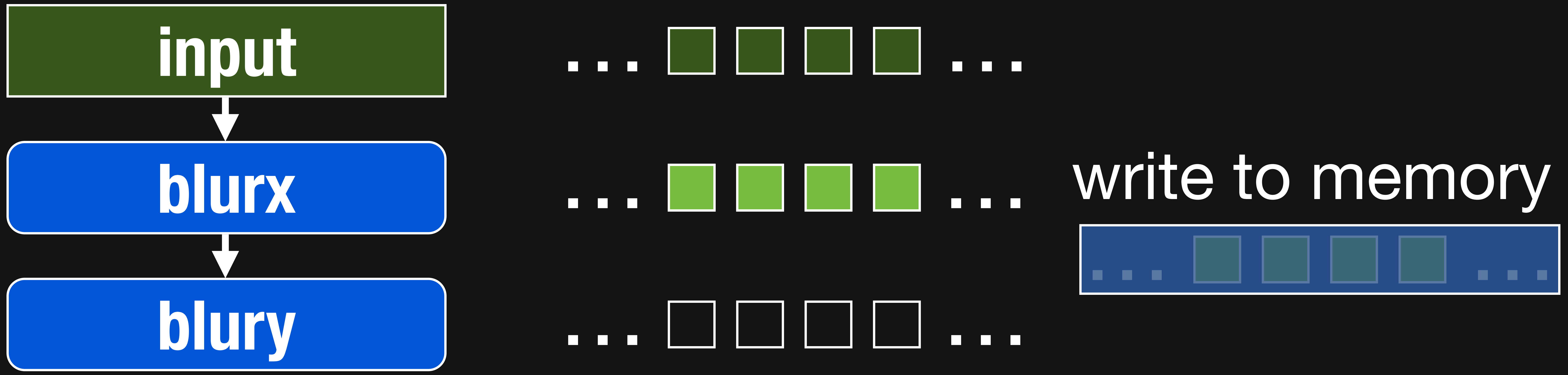
Simple loops execute **breadth-first** across stages



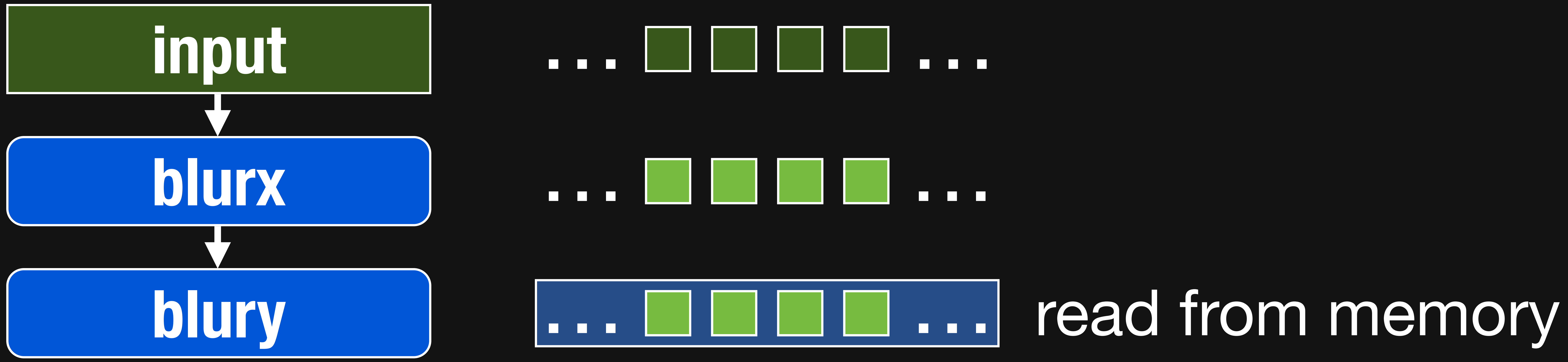
Simple loops execute **breadth-first** across stages



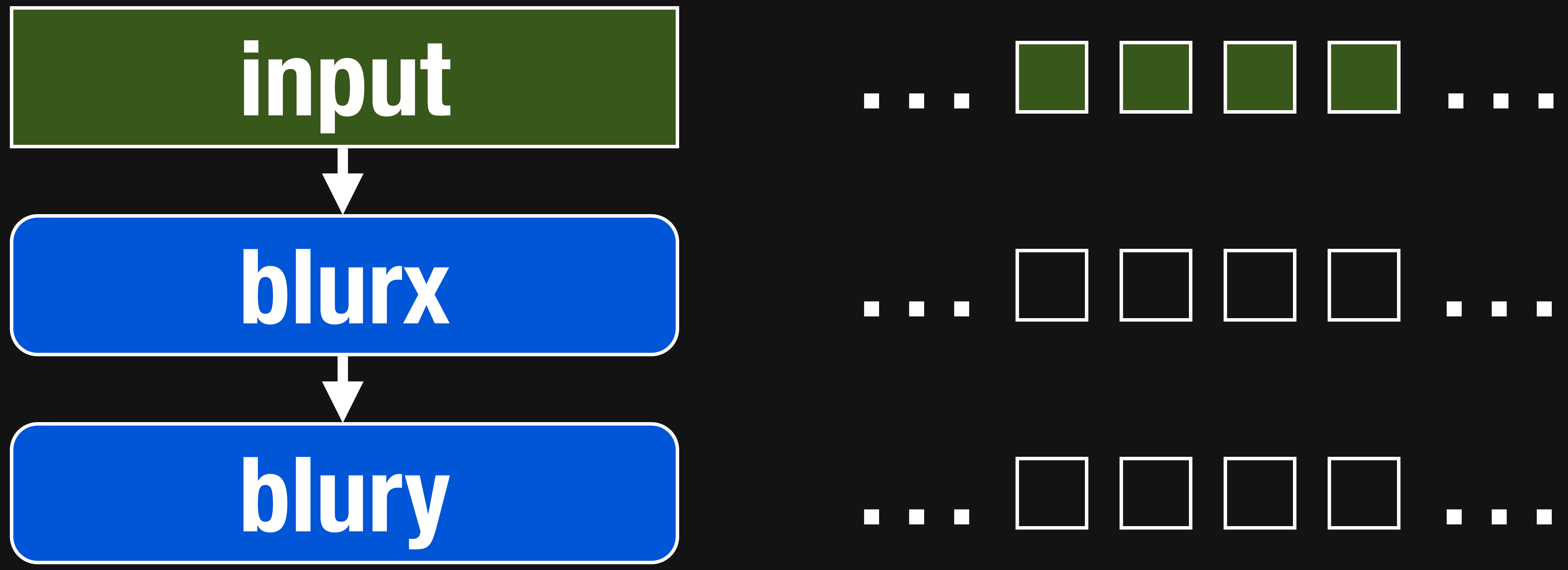
Simple loops execute **breadth-first** across stages



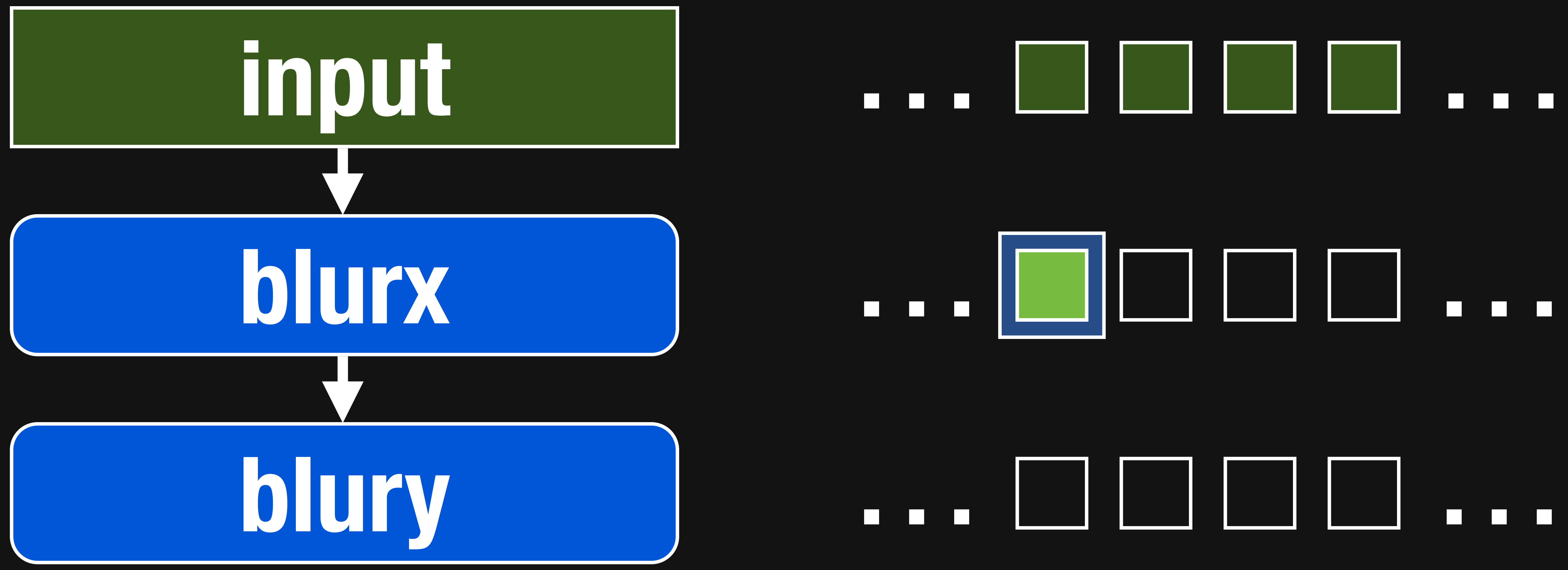
Simple loops execute **breadth-first** across stages



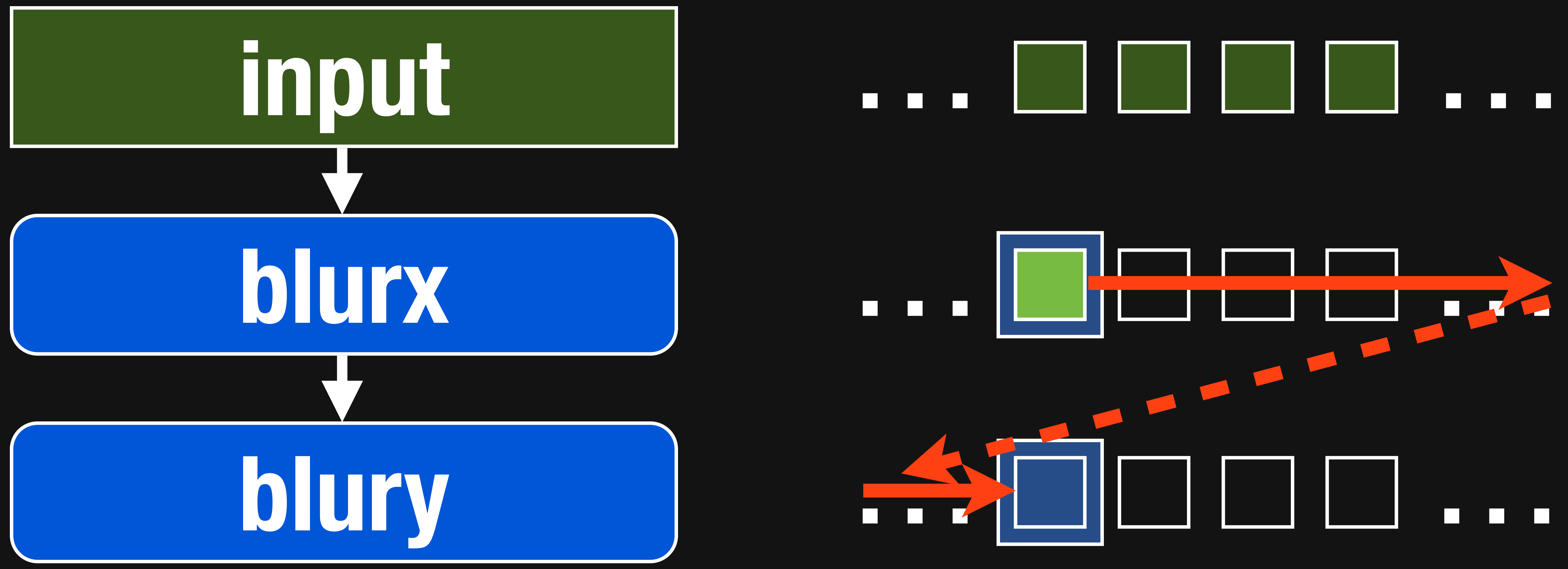
Breadth-first execution sacrifices locality



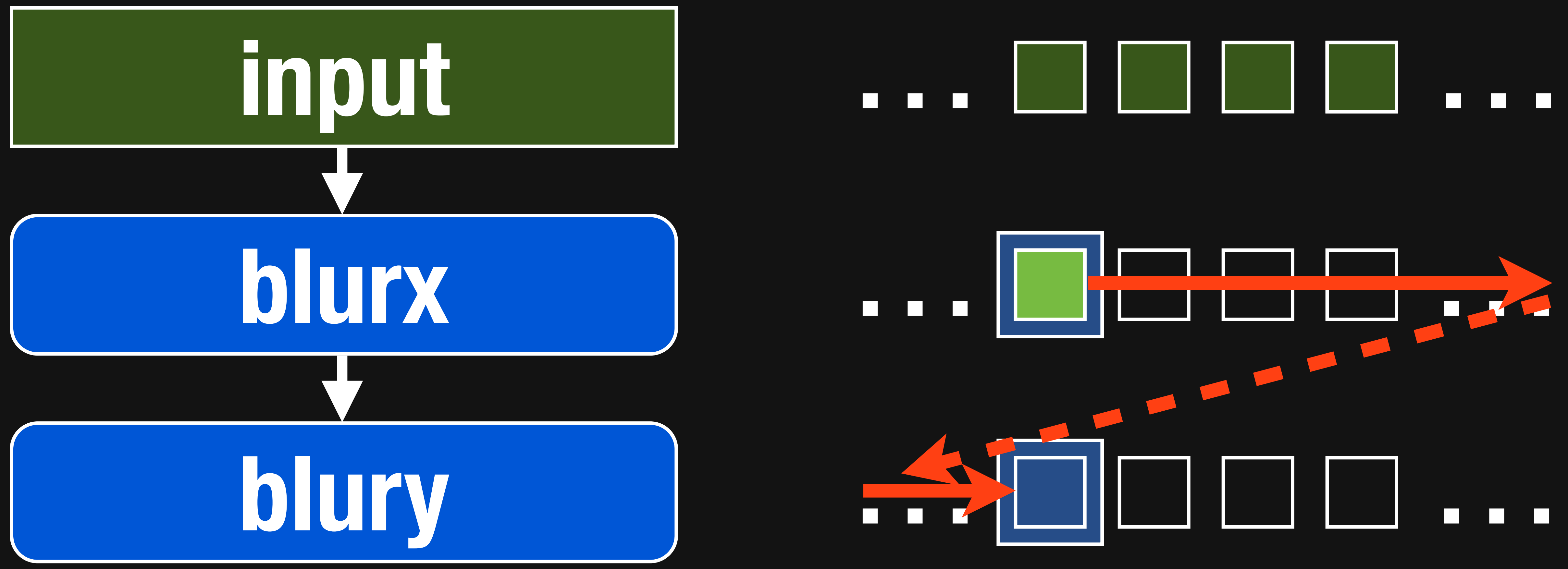
Breadth-first execution sacrifices locality



Breadth-first execution sacrifices locality

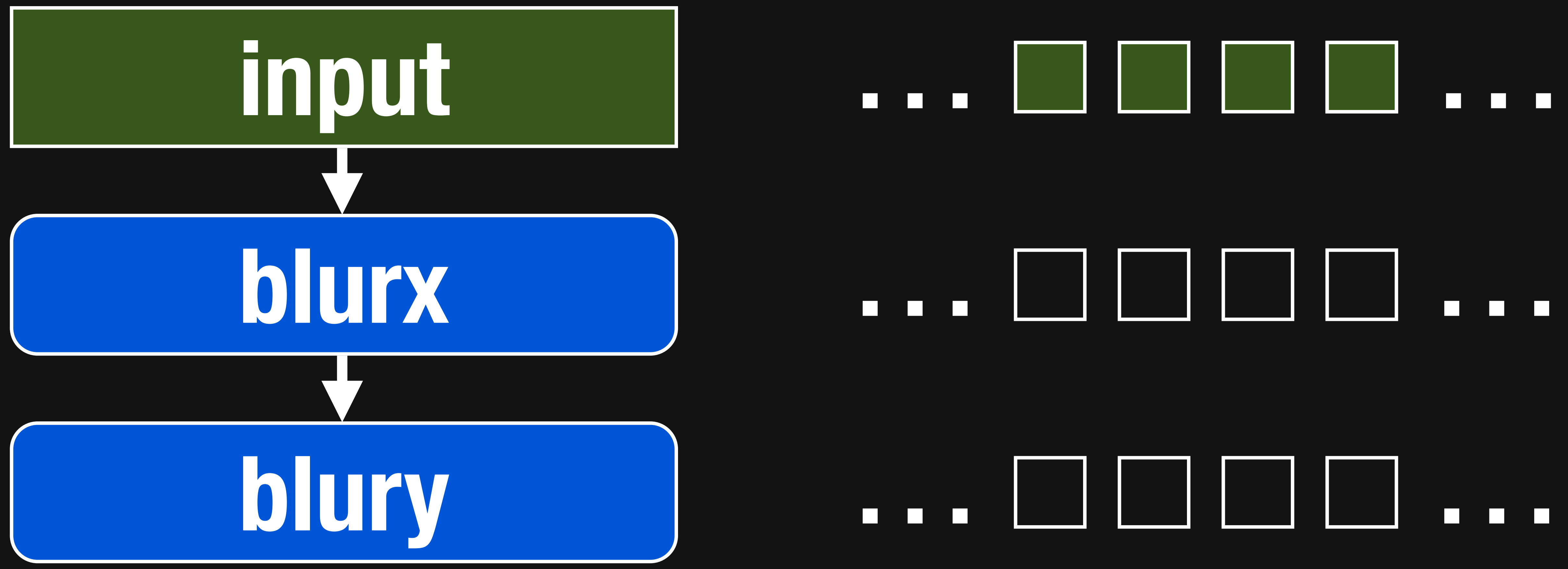


Breadth-first execution sacrifices locality

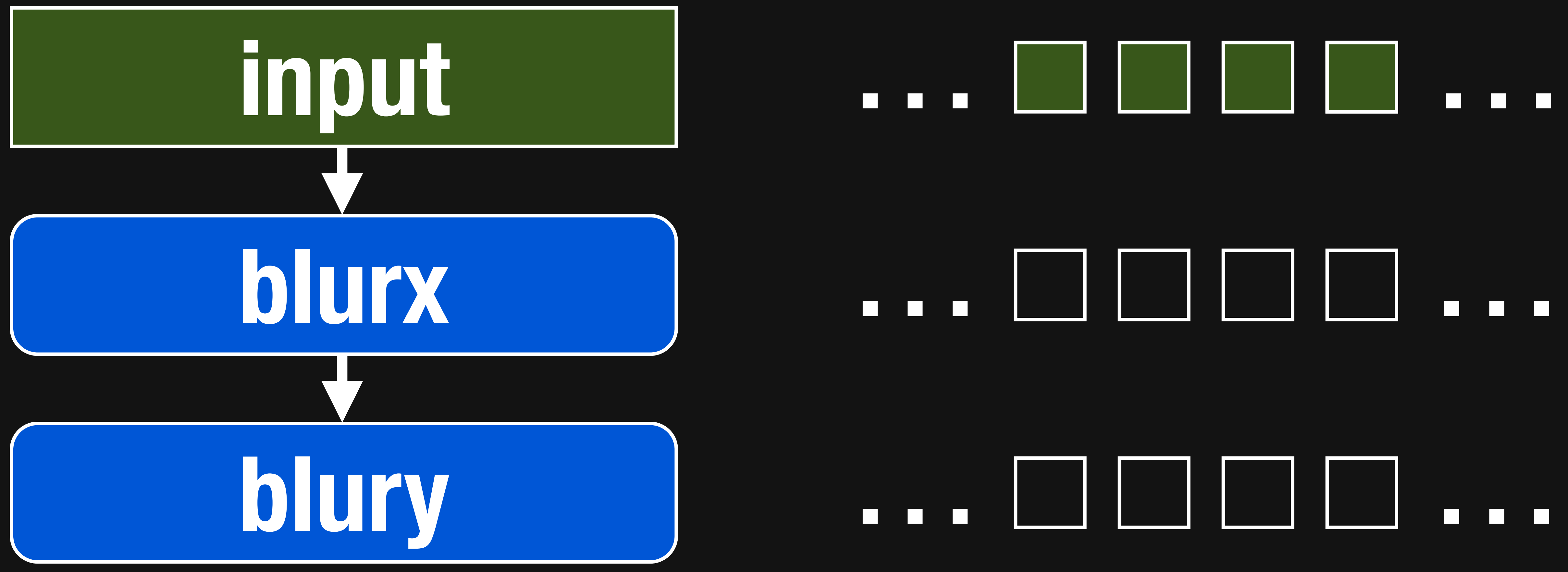


locality is a function of **reuse distance**

Interleaved execution improves locality

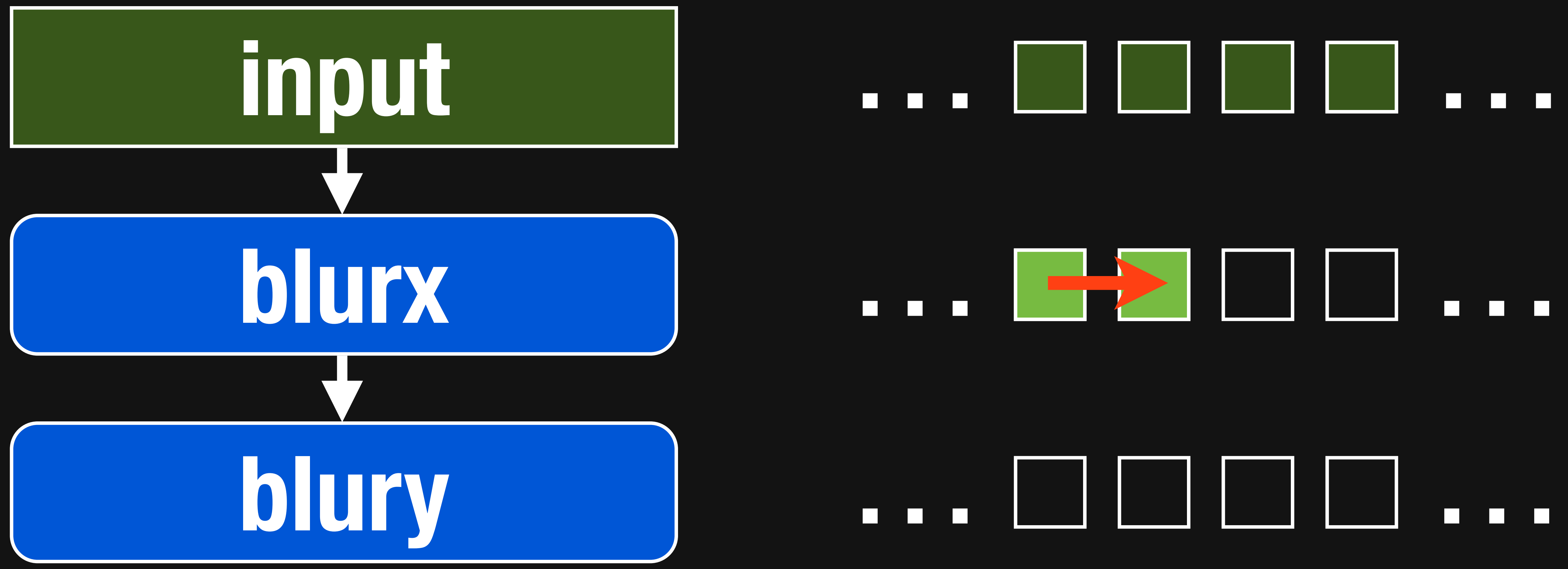


Interleaved execution improves locality



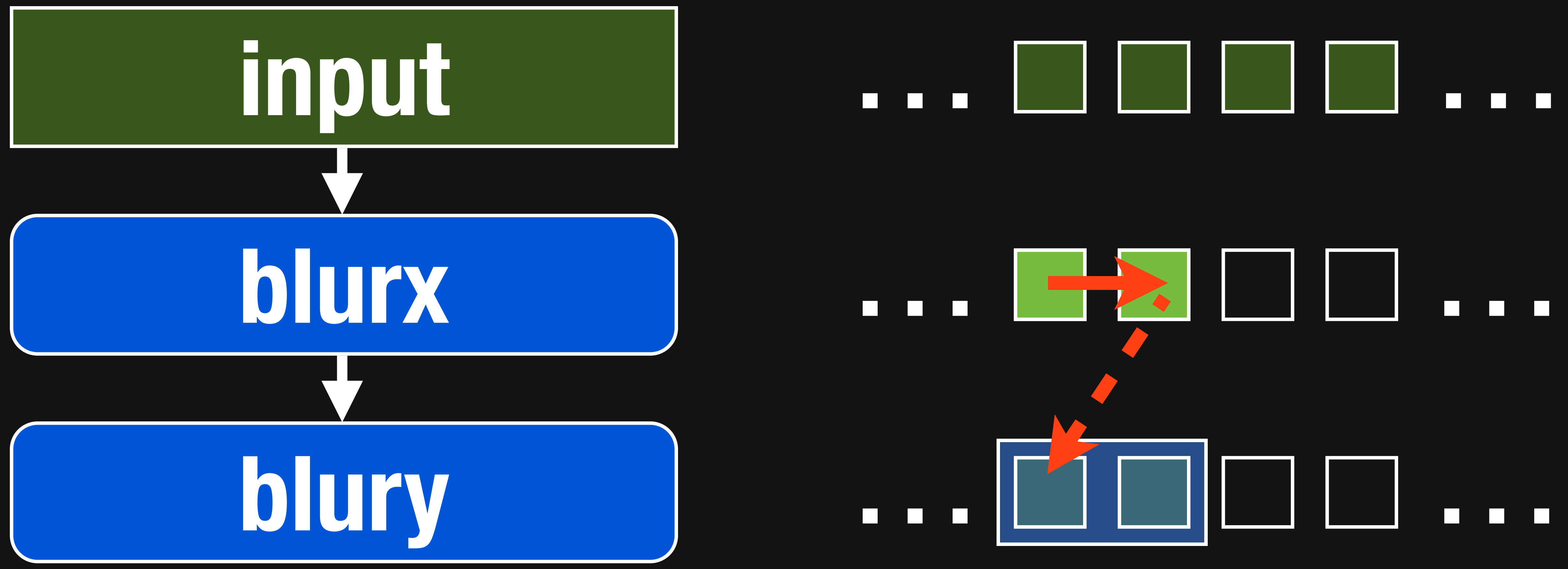
(simplified)

Interleaved execution improves locality



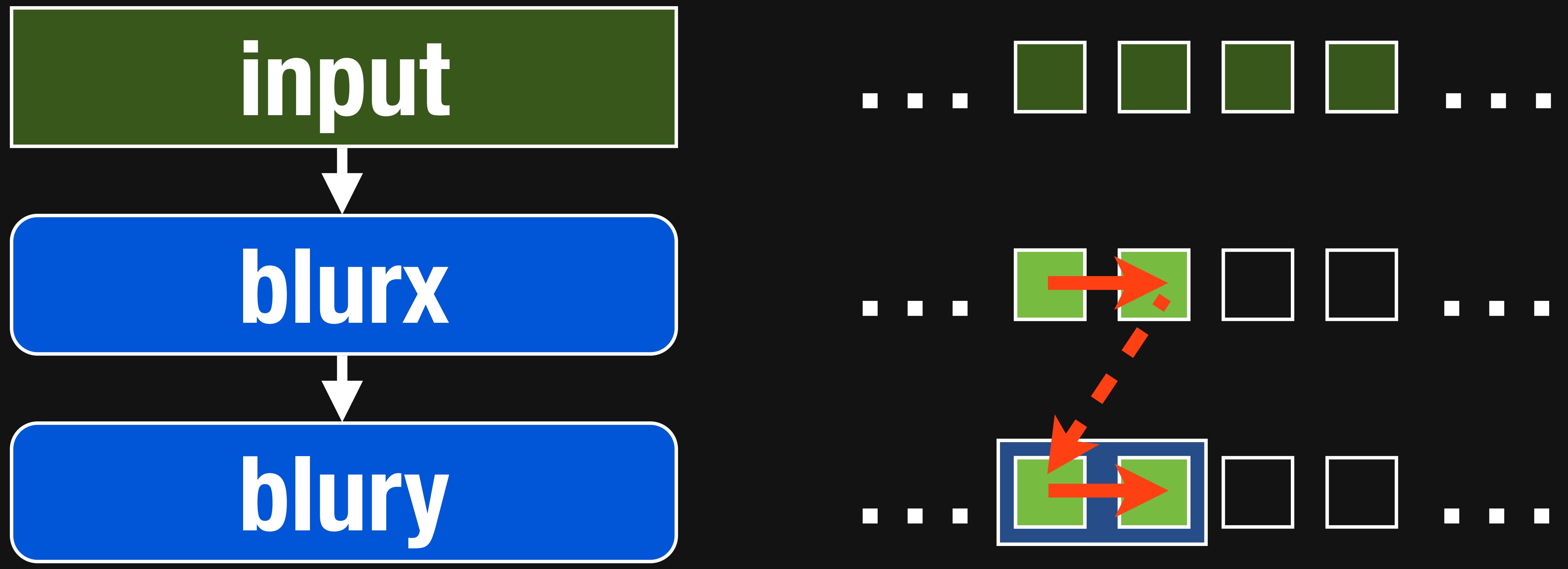
(simplified)

Interleaved execution improves locality



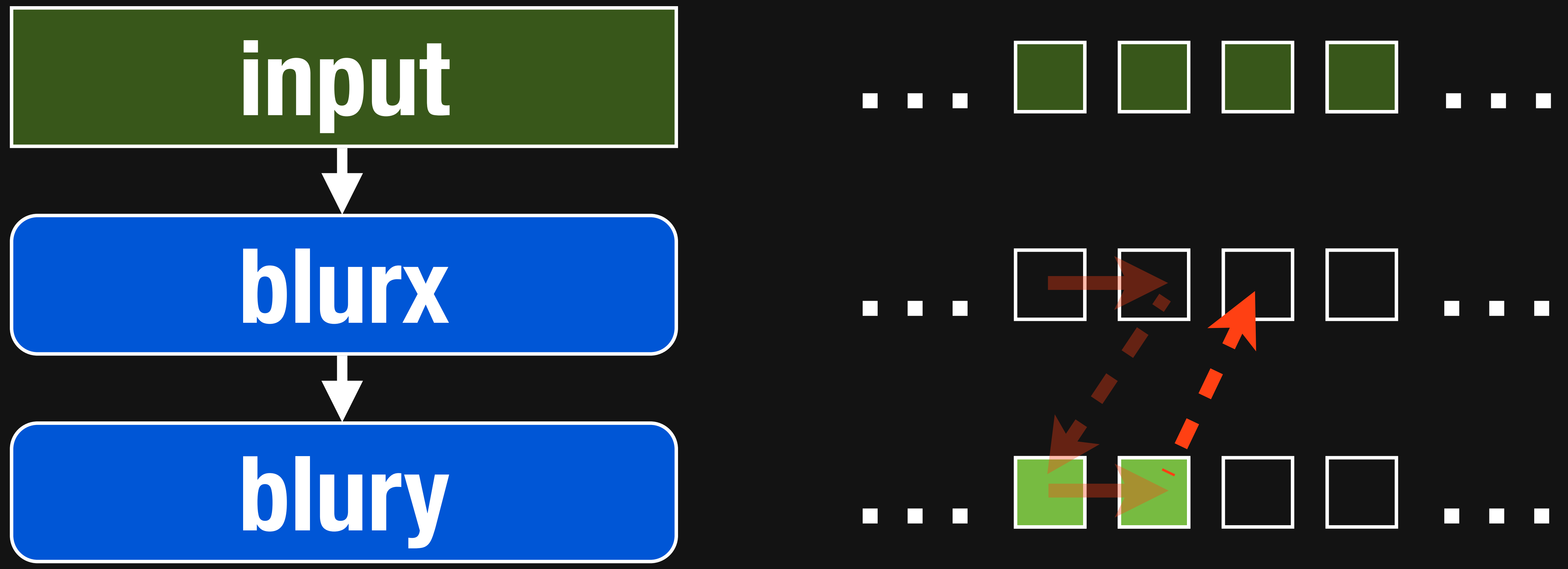
(simplified)

Interleaved execution improves locality



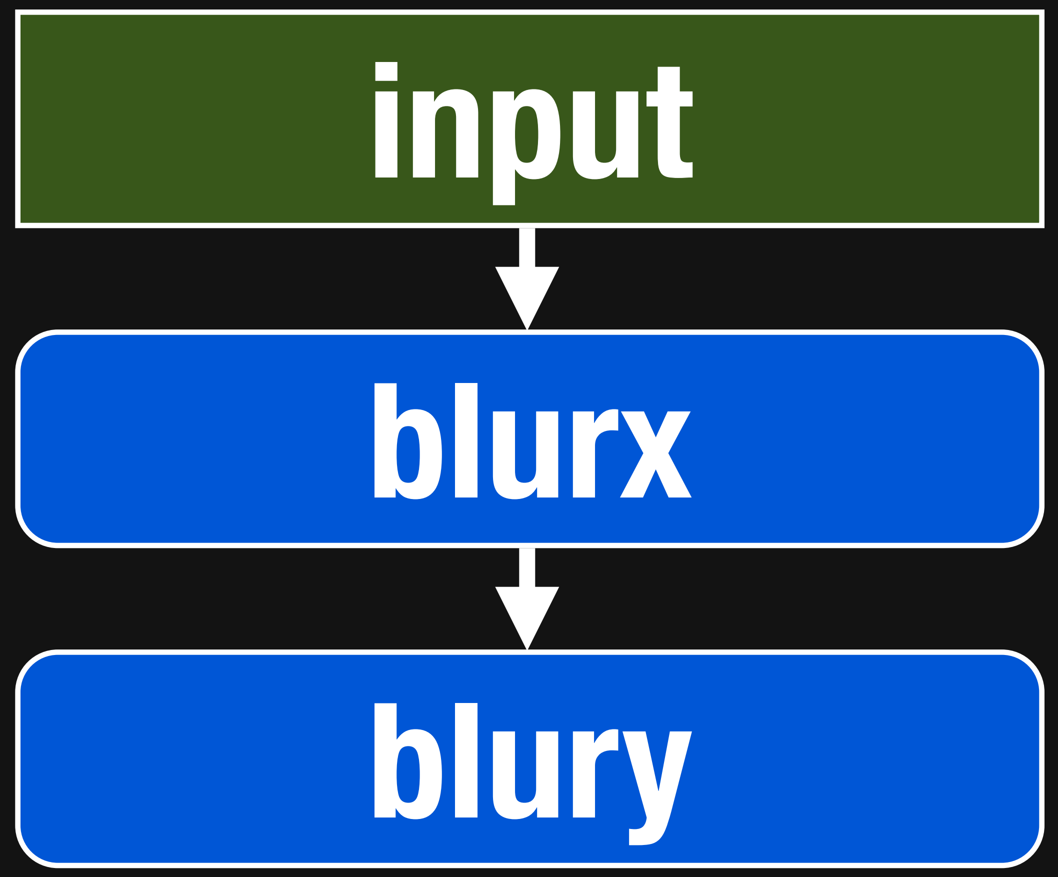
(simplified)

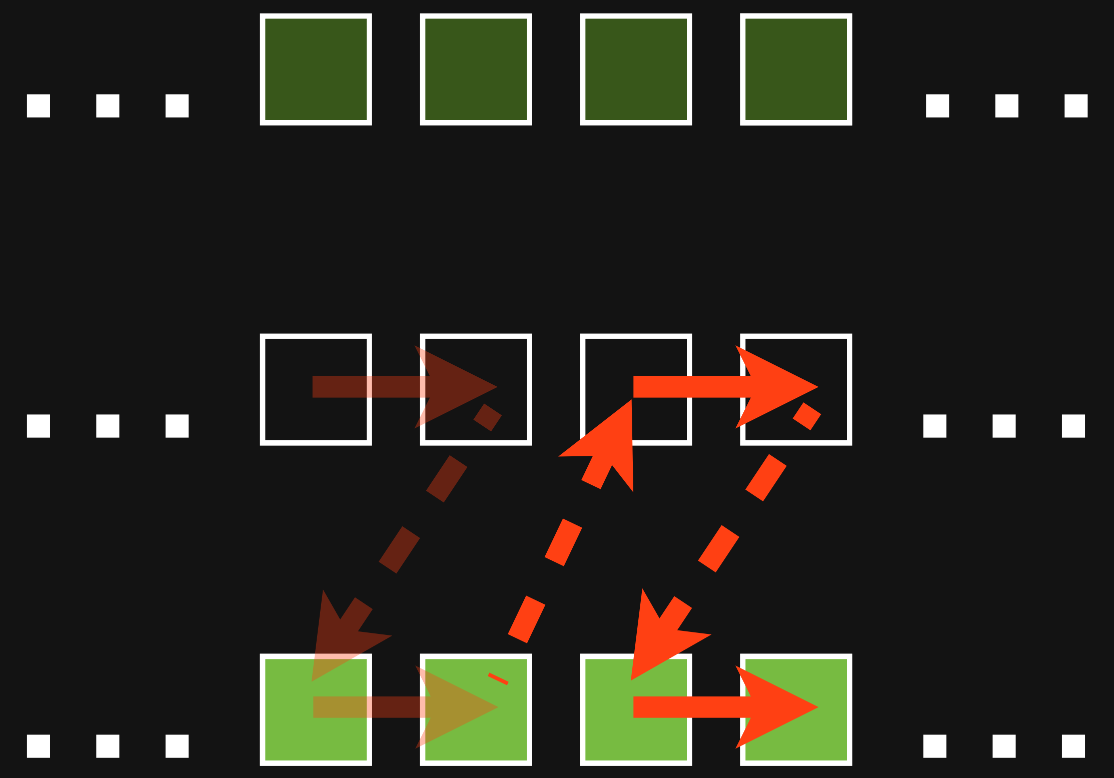
Interleaved execution improves locality



(simplified)

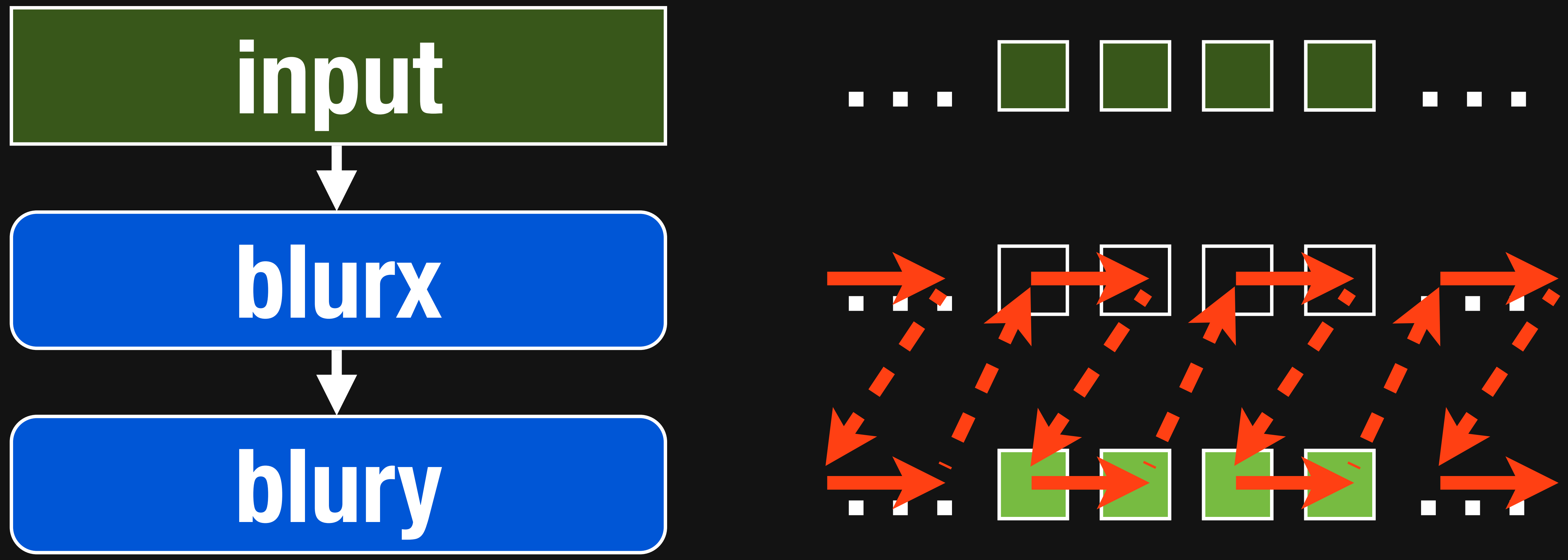
Interleaved execution improves locality



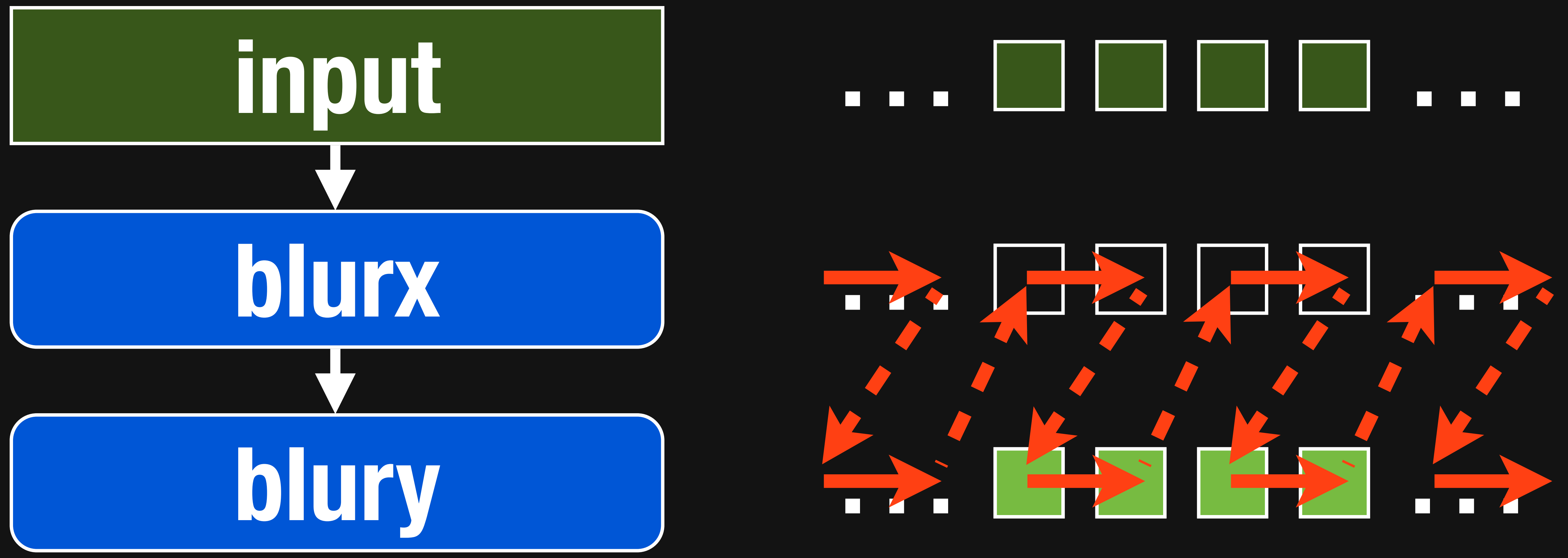


(simplified)

Interleaved execution improves locality

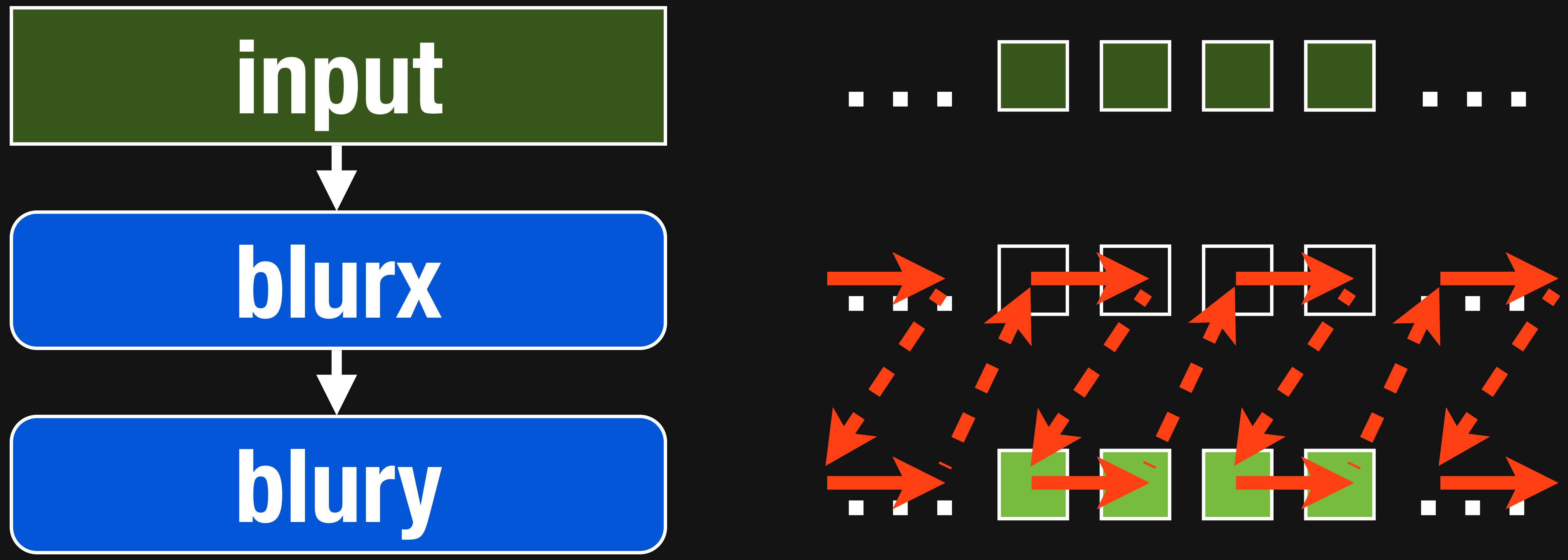


Interleaved execution improves locality

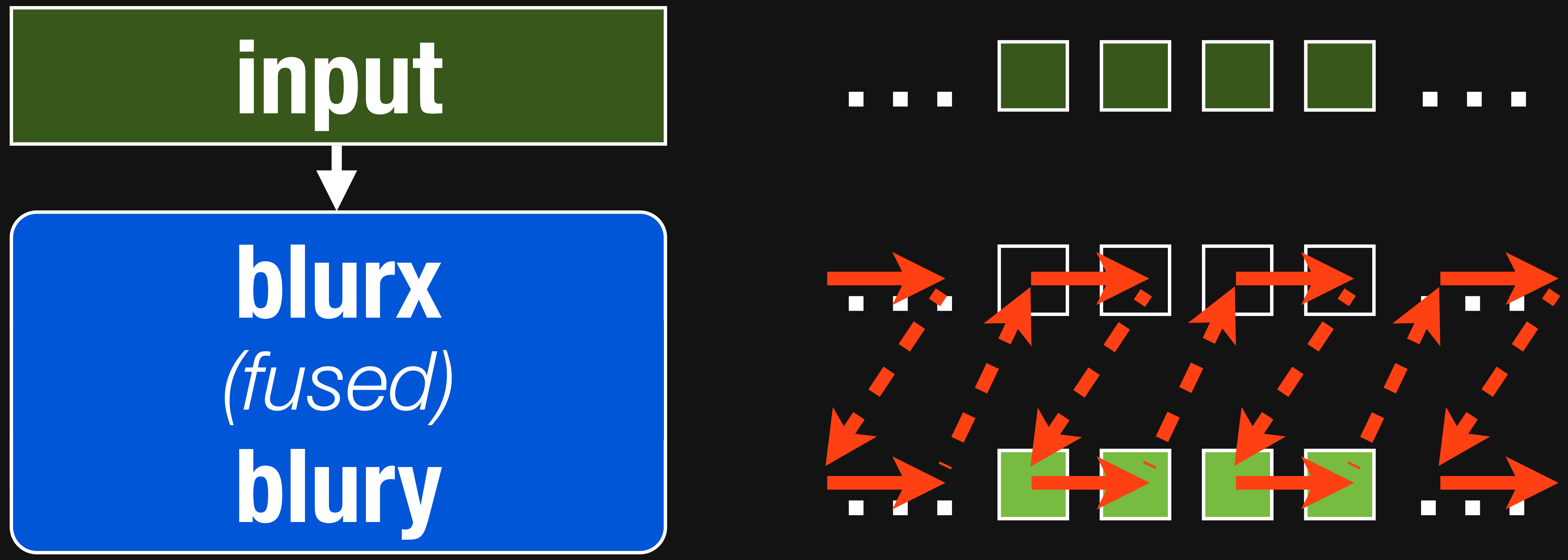


reduce reuse distance from **producer** to **consumer**

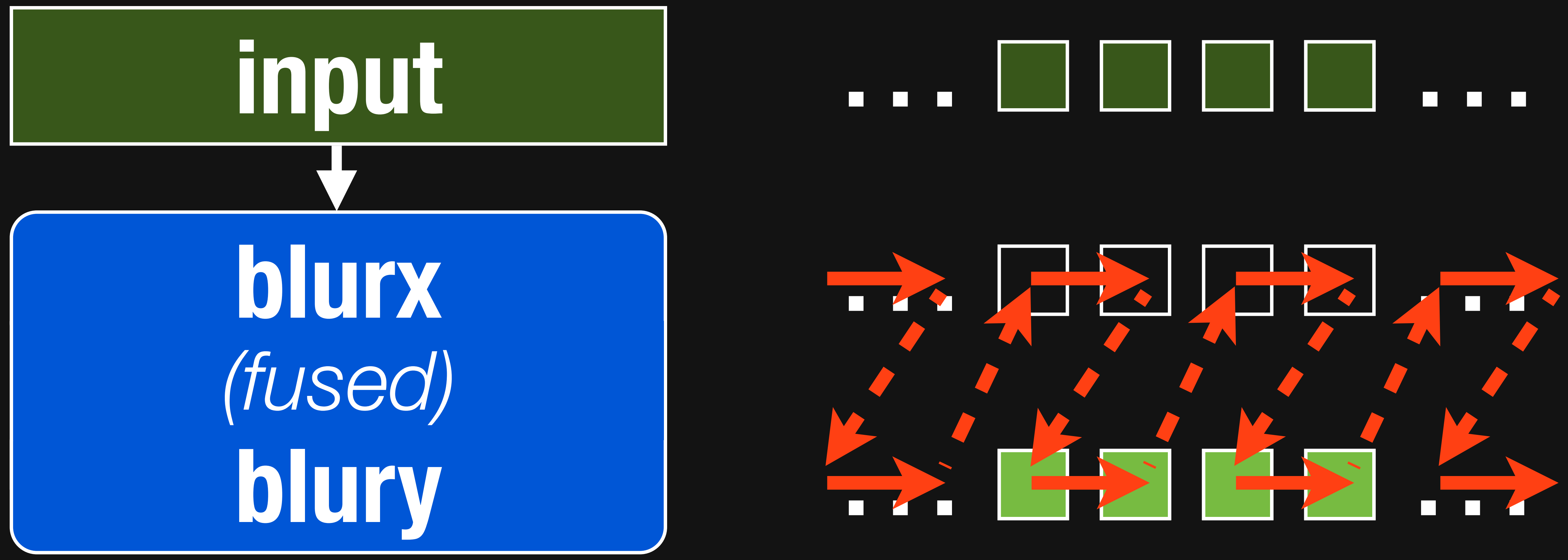
Fusion improves locality



Fusion improves locality



Fusion improves locality



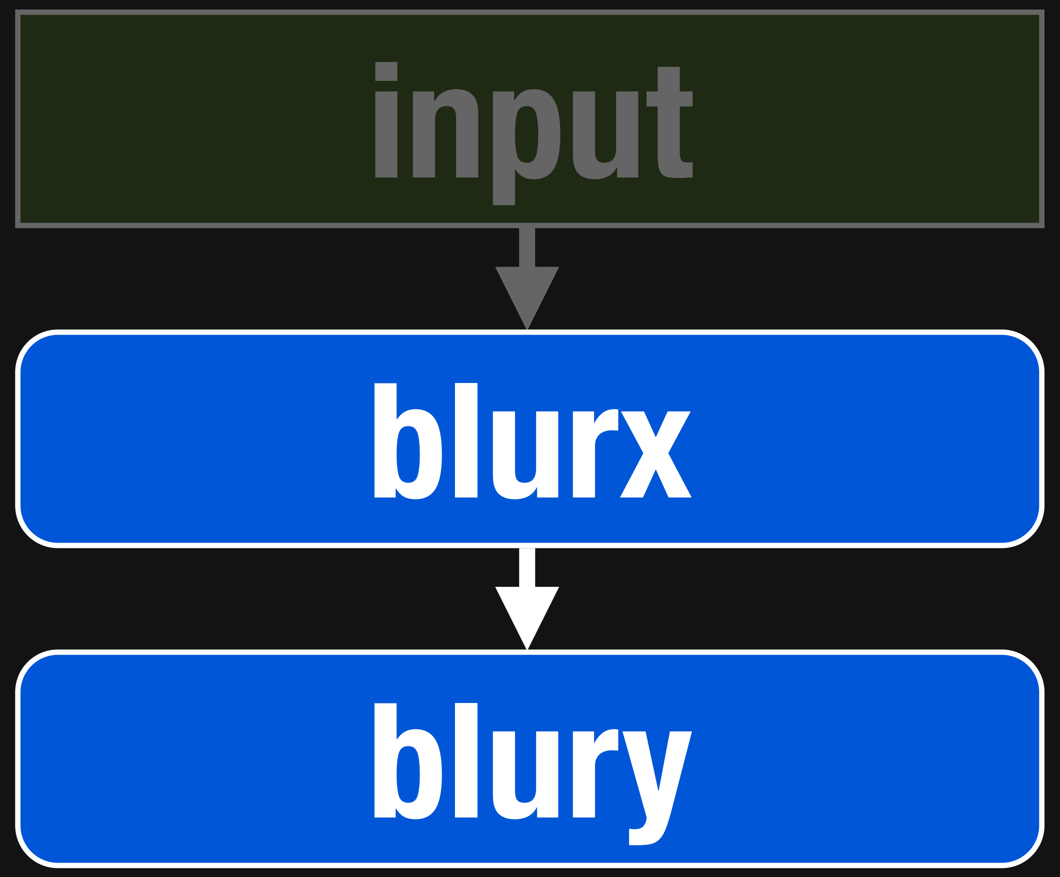
fusion **globally interleaves** computation

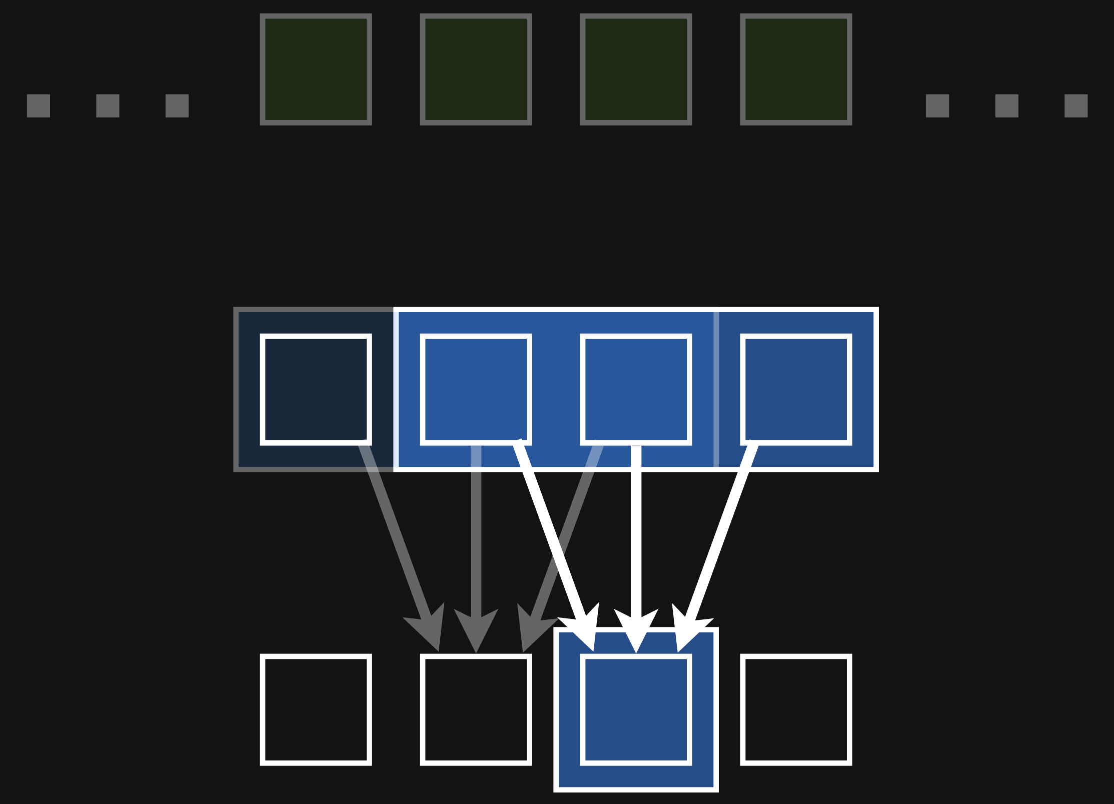
Understanding dependencies

Understanding dependencies

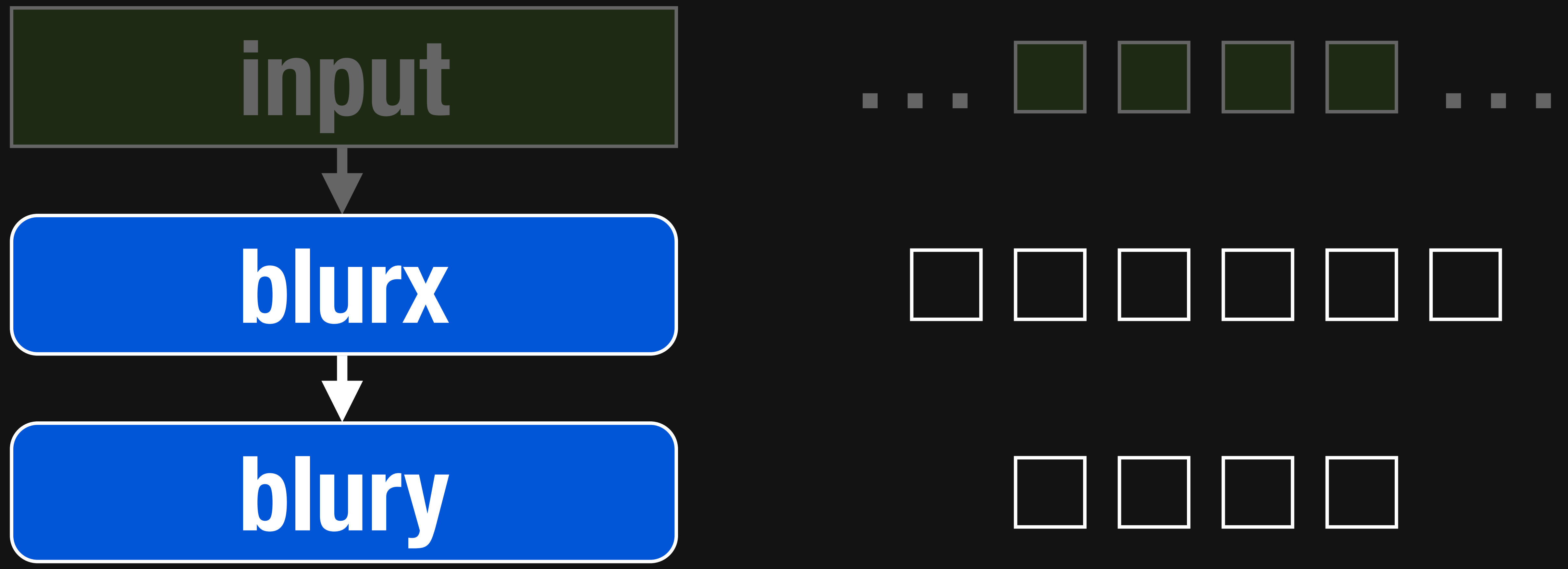
Understanding dependencies

Understanding dependencies

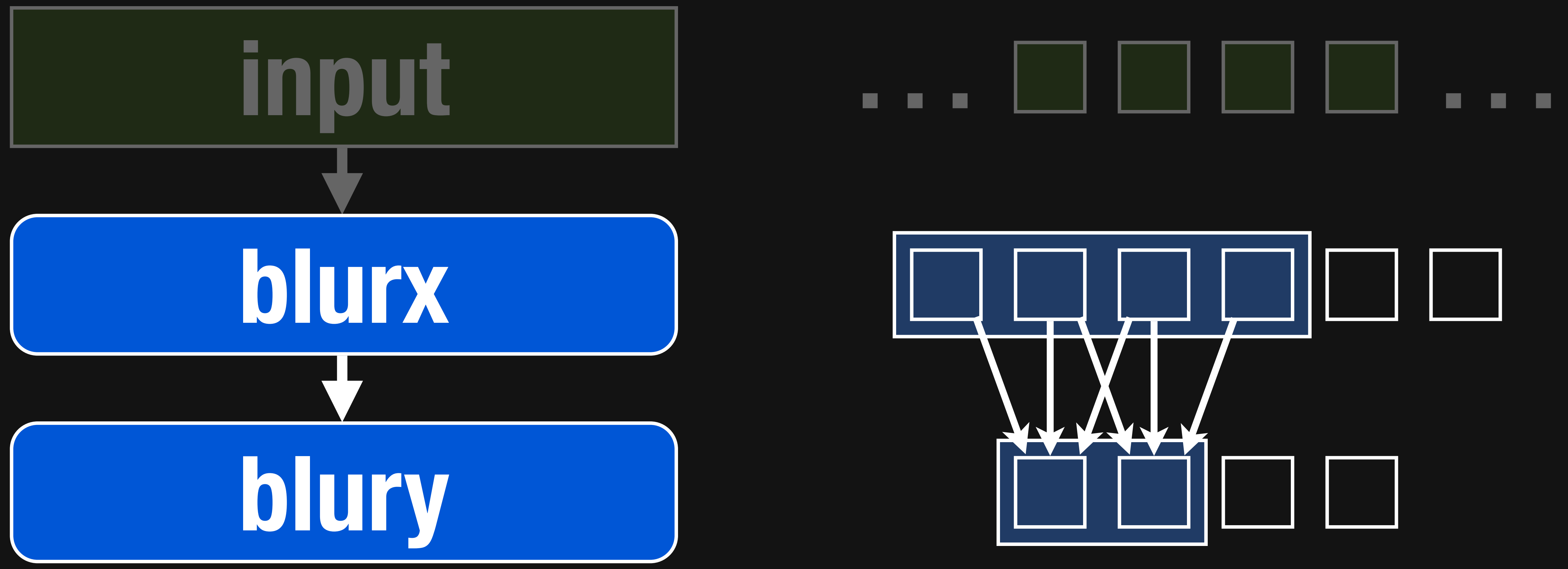




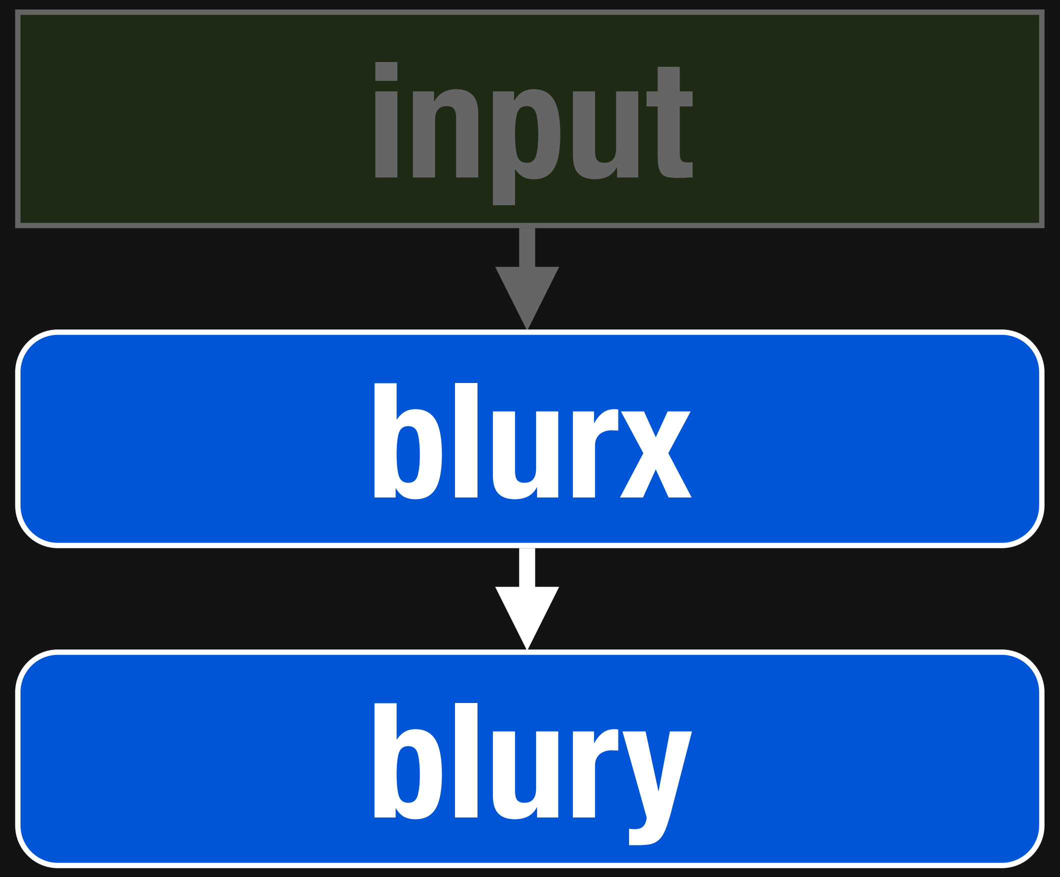
Stencils have overlapping dependencies

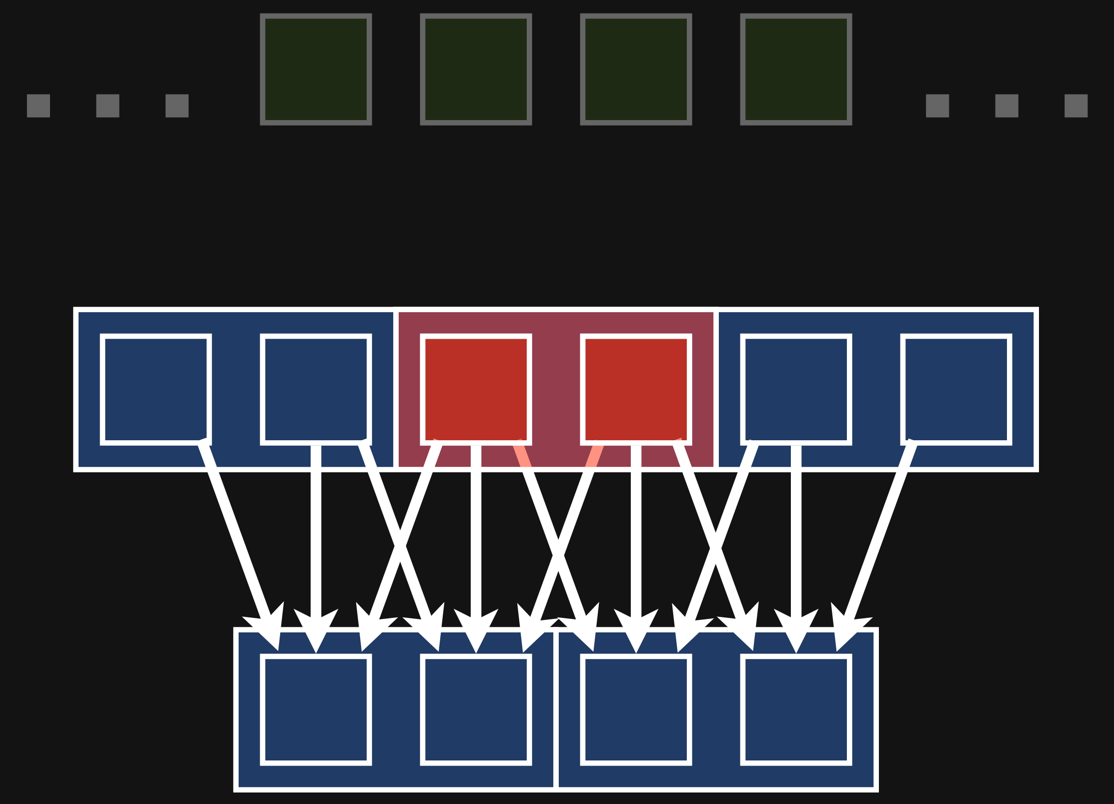


Stencils have overlapping dependencies

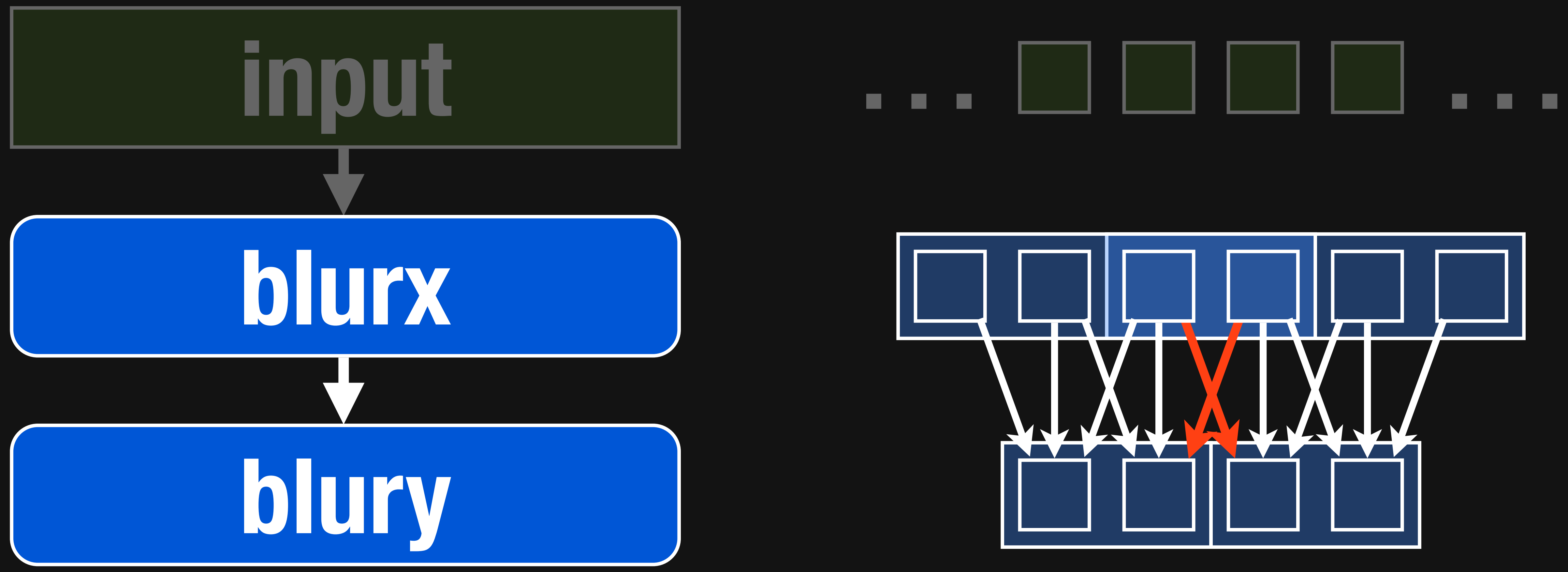


Stencils have overlapping dependencies

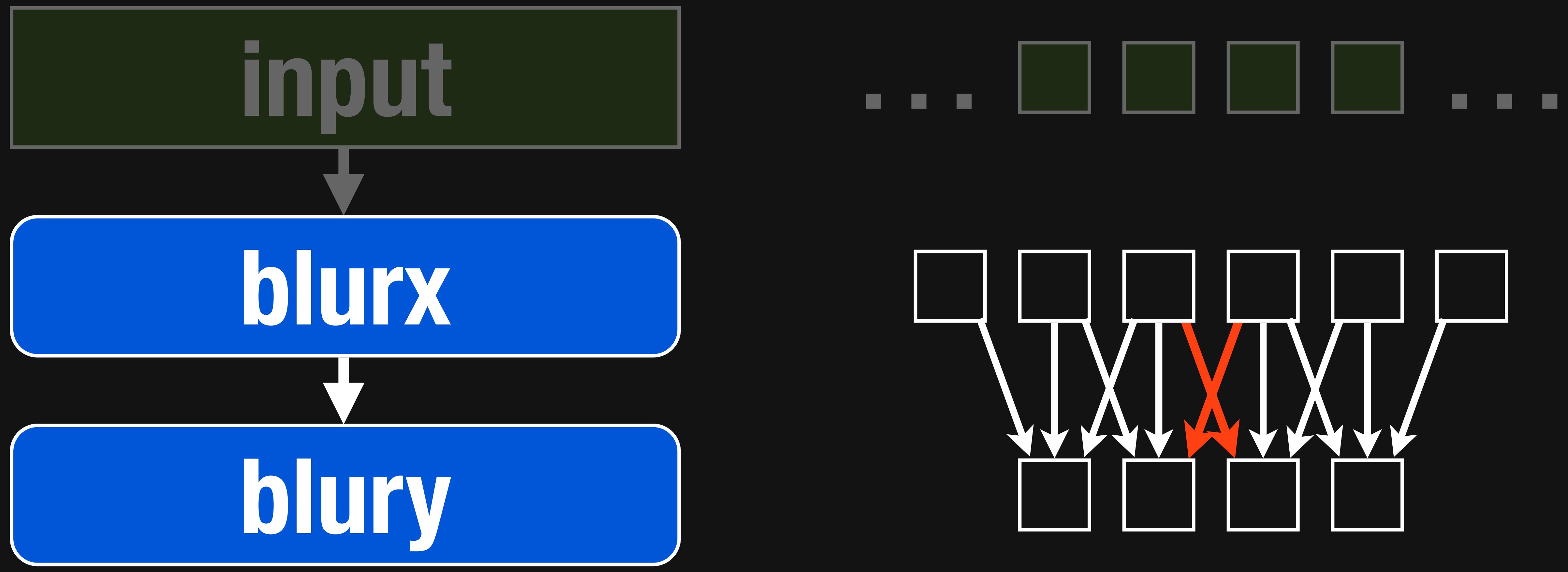




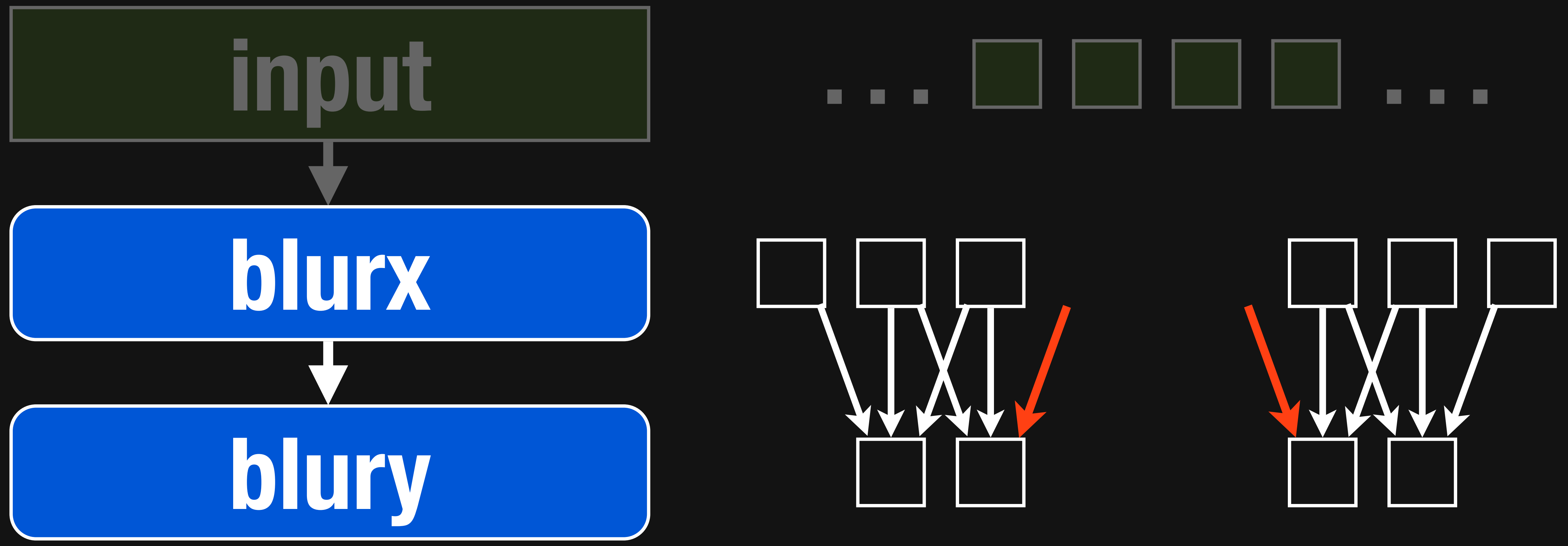
Stencils have overlapping dependencies



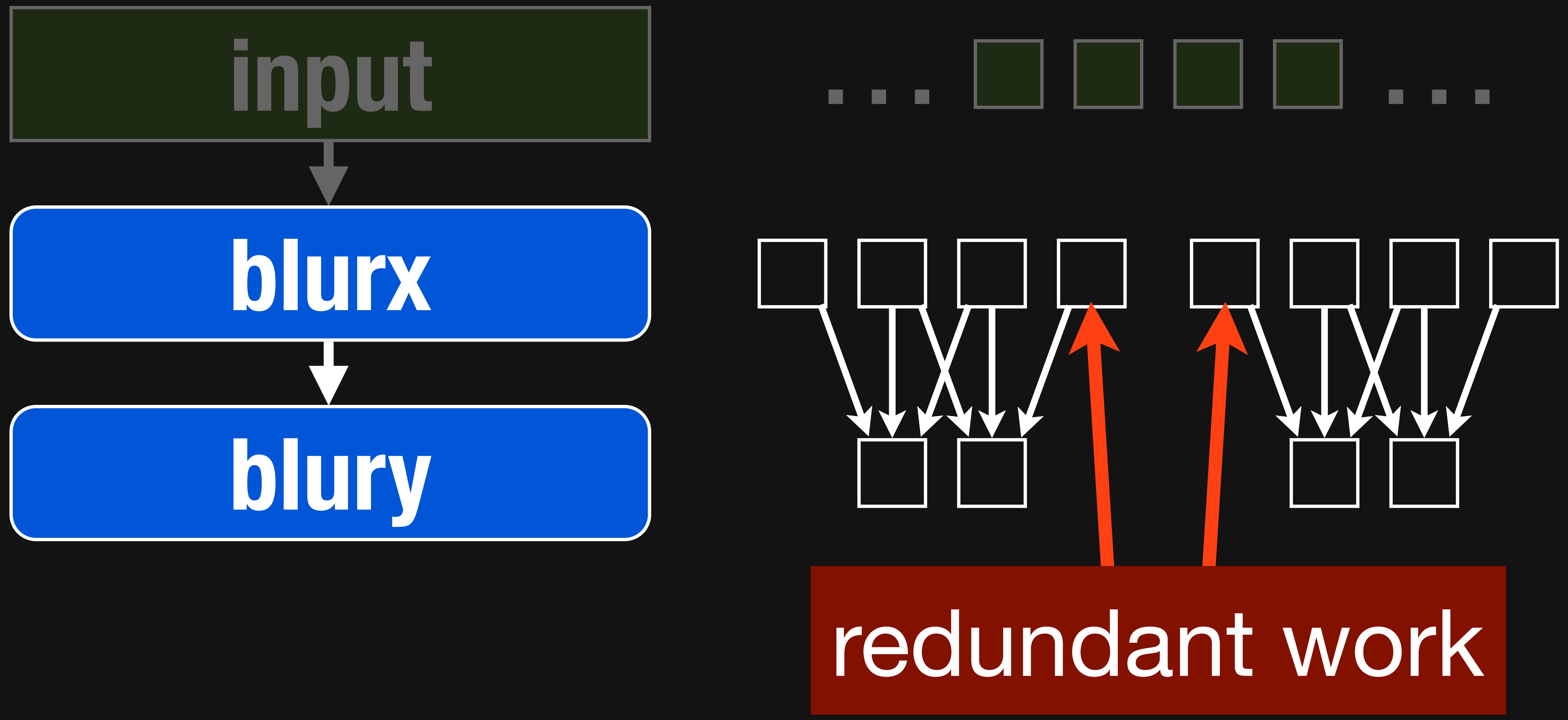
Breaking dependencies introduces redundant work



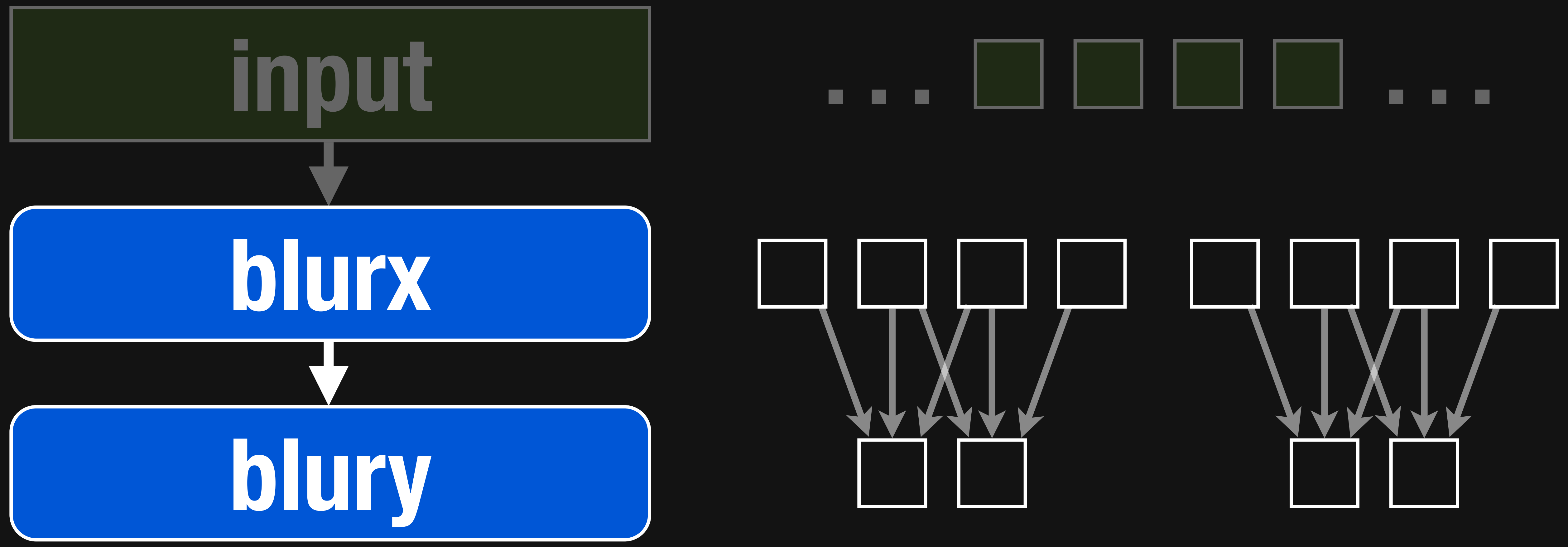
Breaking dependencies introduces redundant work



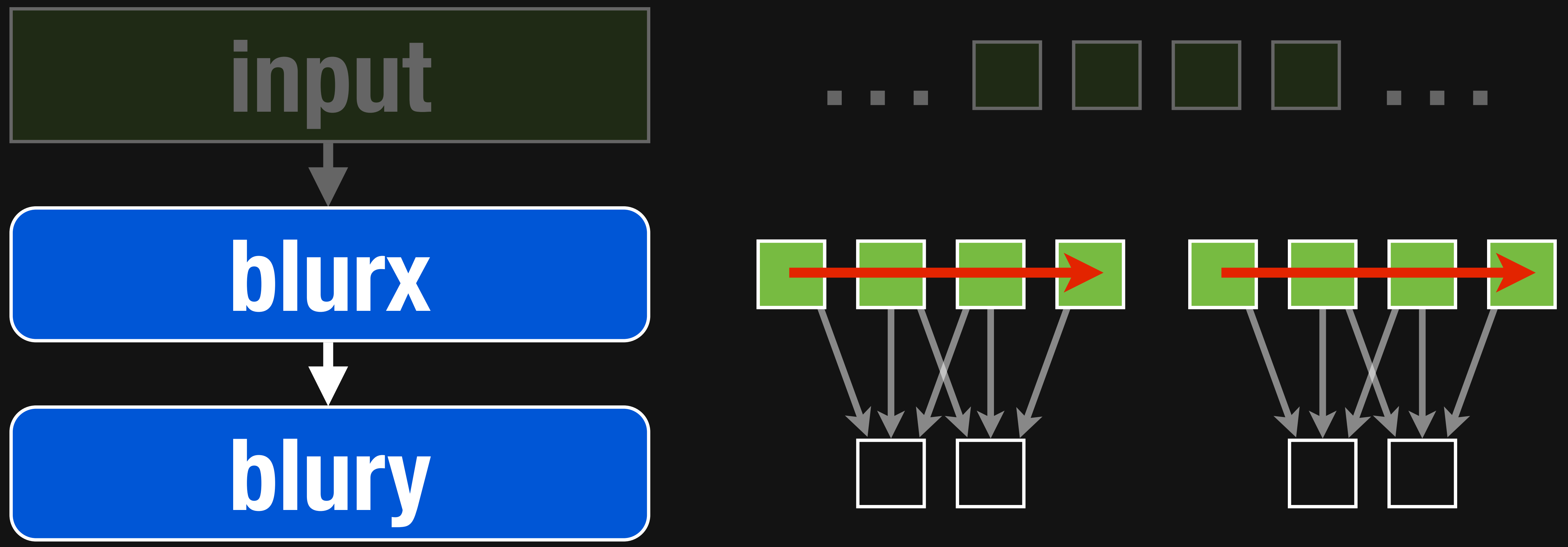
Breaking dependencies introduces redundant work



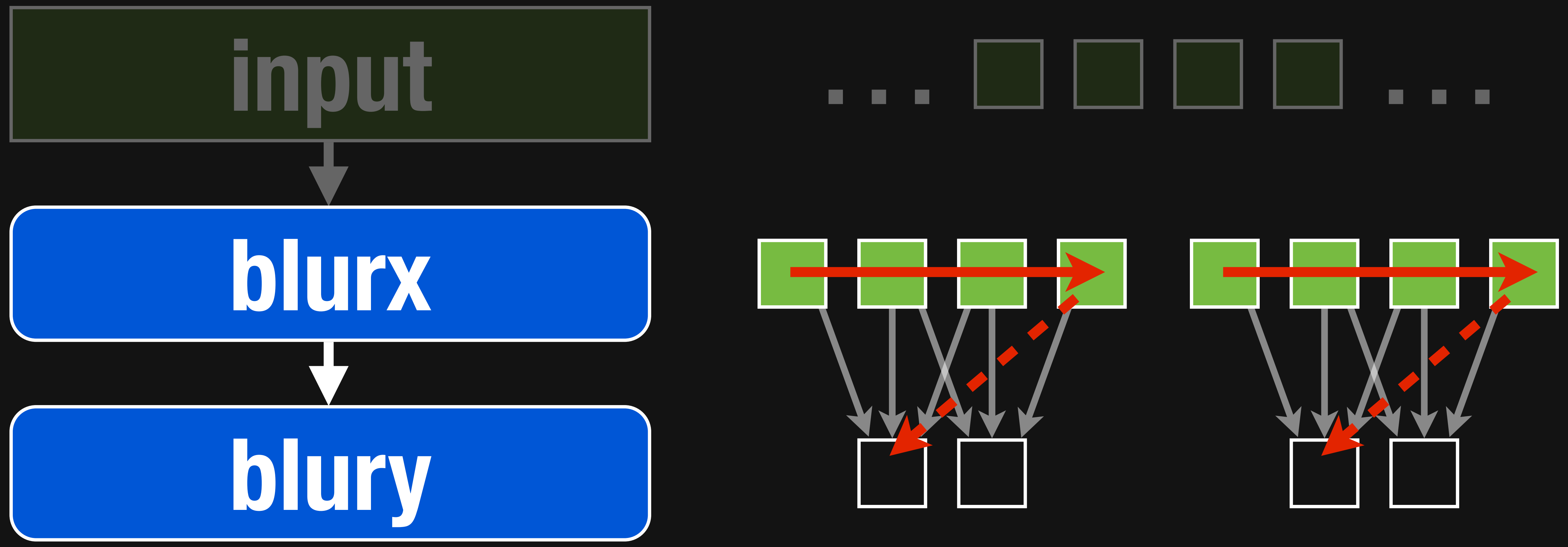
Decoupled tiles optimize parallelism & locality



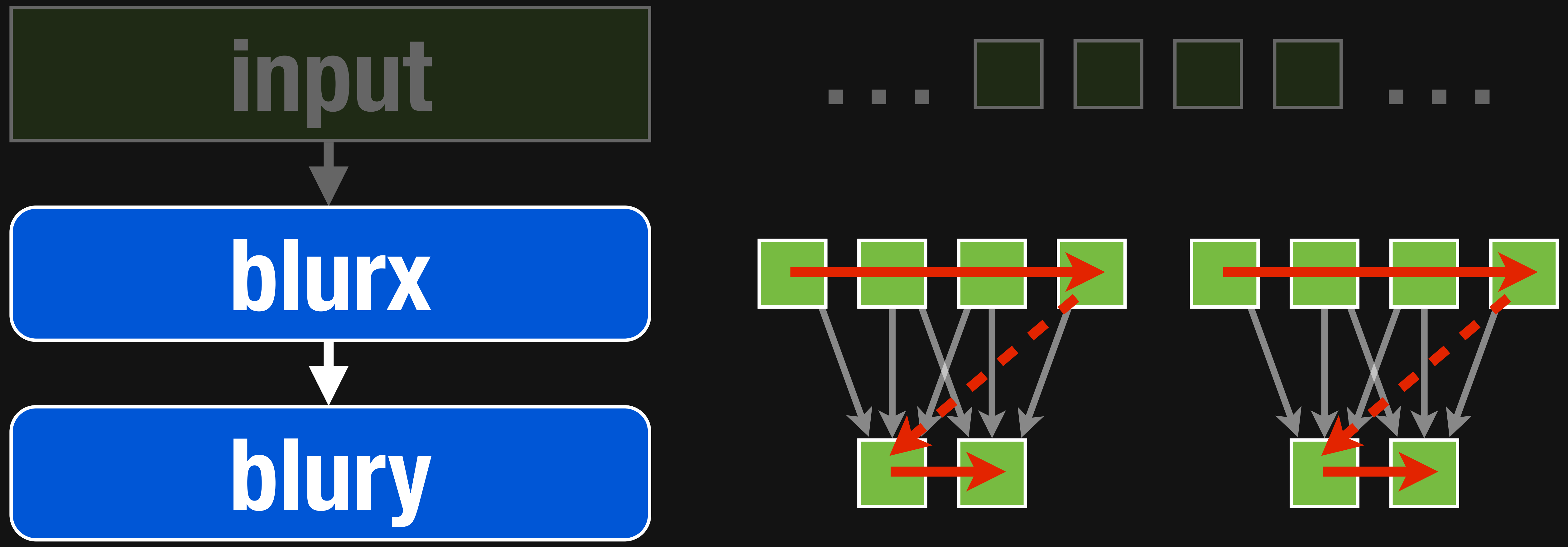
Decoupled tiles optimize parallelism & locality



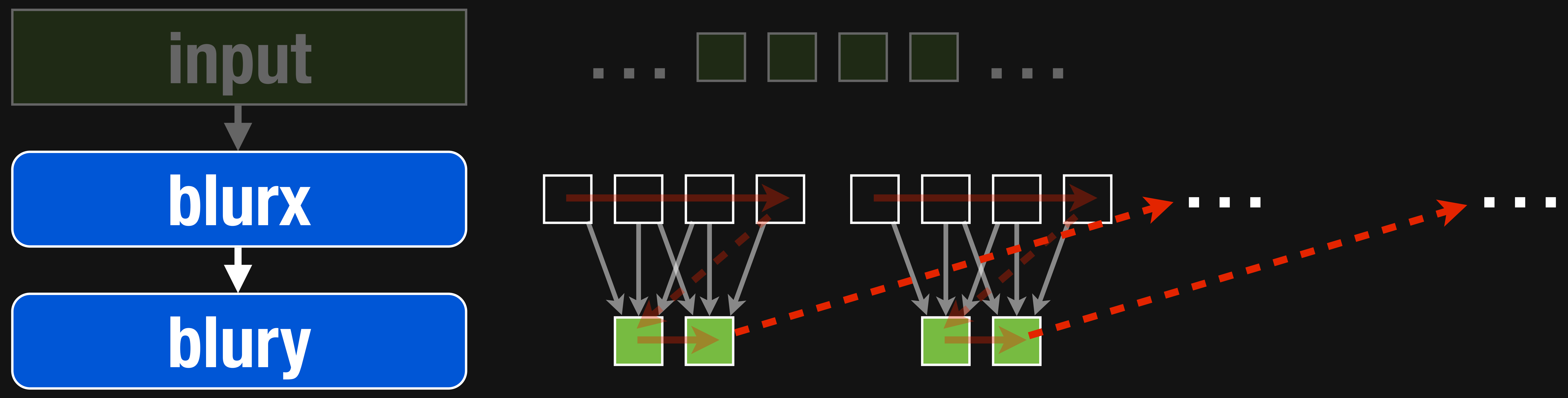
Decoupled tiles optimize parallelism & locality



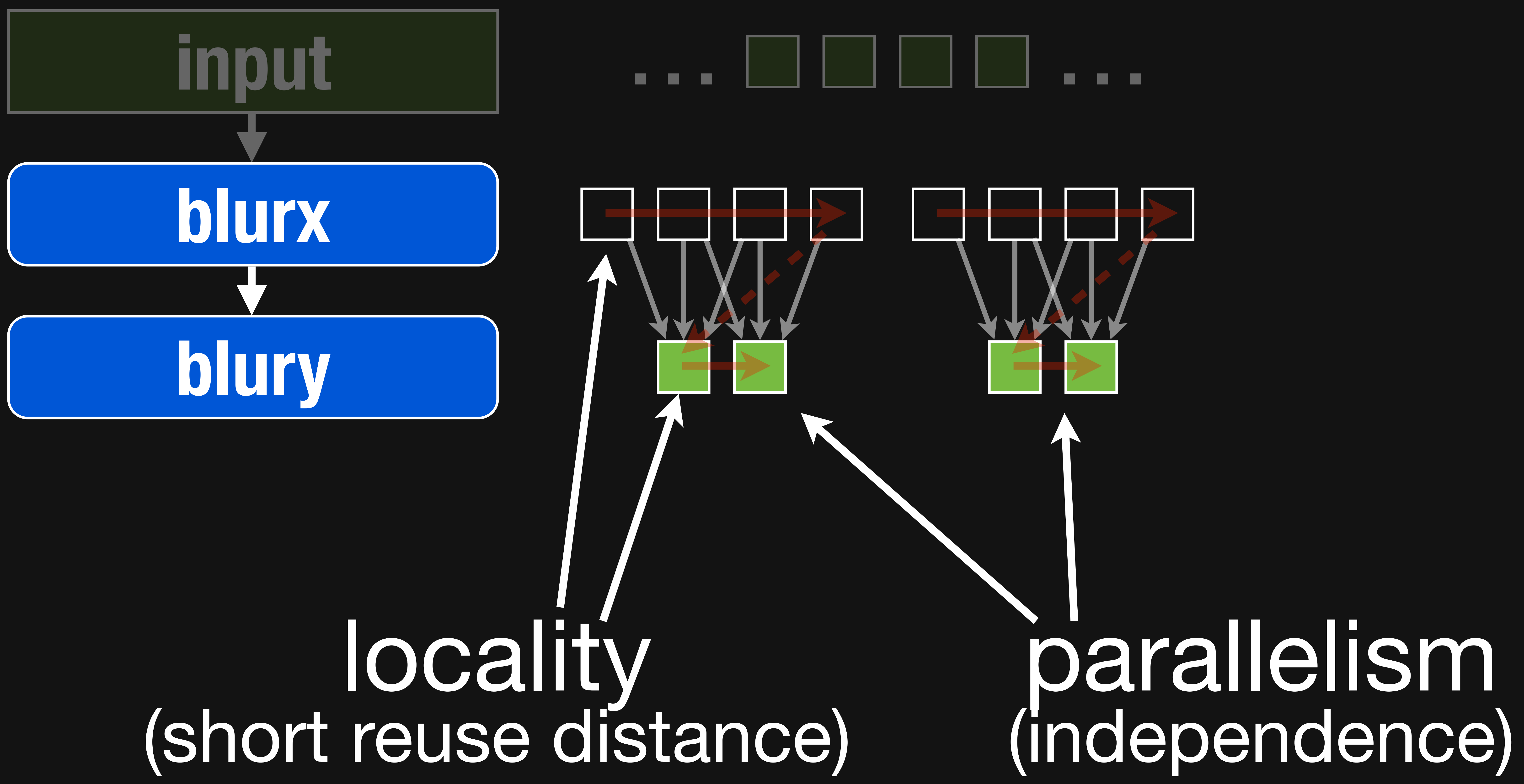
Decoupled tiles optimize parallelism & locality



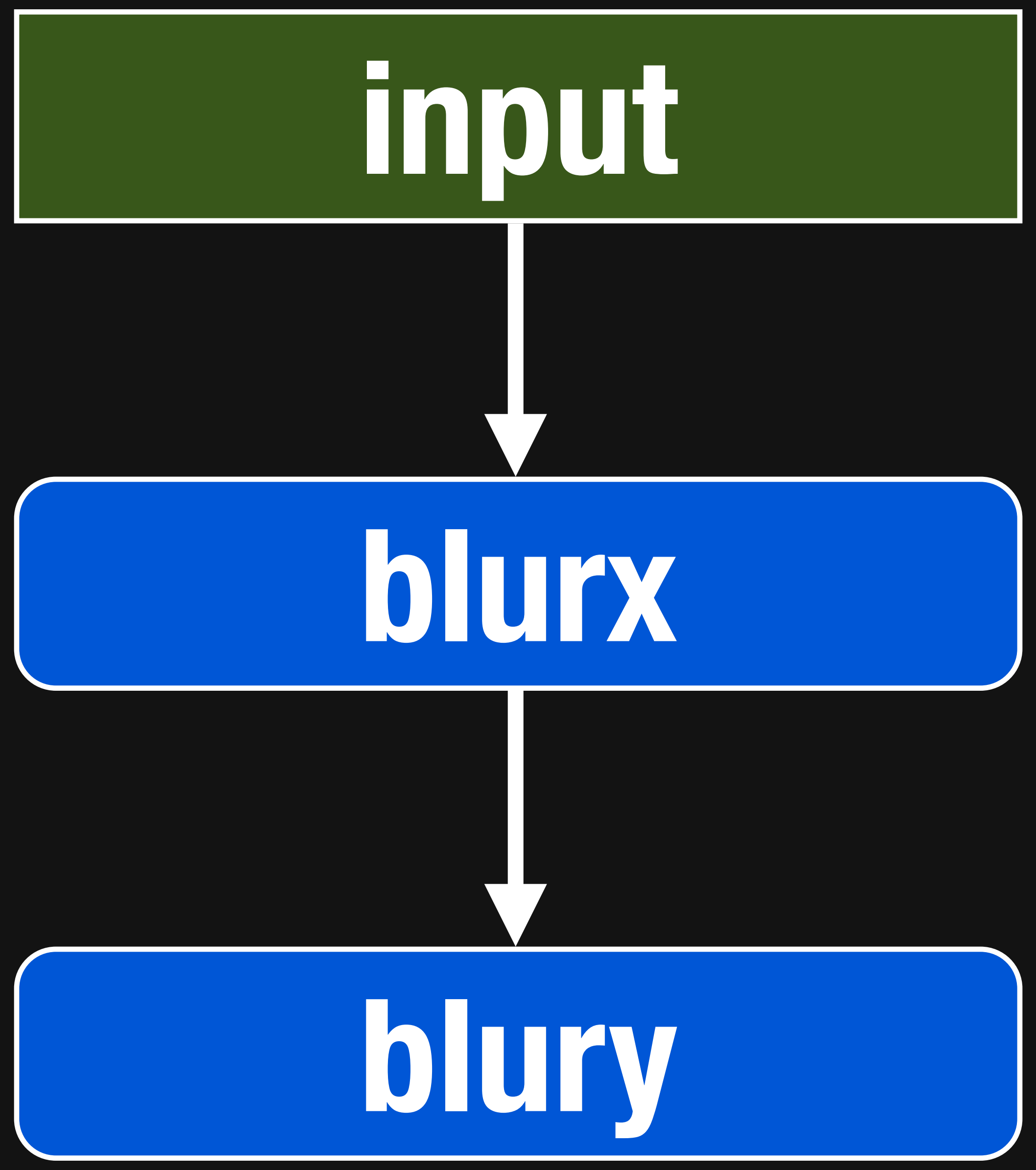
Decoupled tiles optimize parallelism & locality



Decoupled tiles optimize parallelism & locality

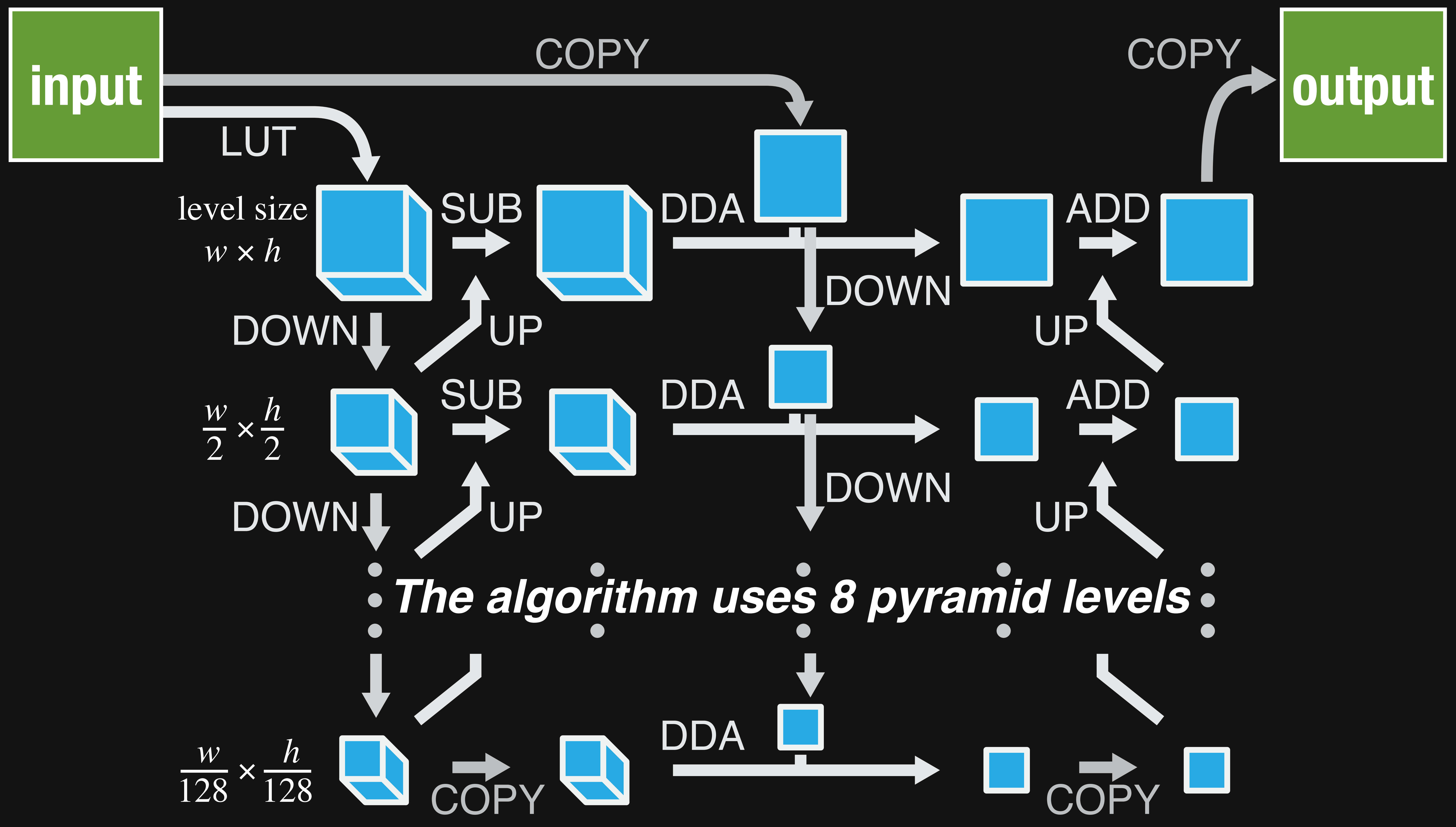


Organization requires global tradeoffs



3x3 box filter

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

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

UP: upsample

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

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

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

DOWN: downsample

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

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

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

DDA: data-dependent access

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

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

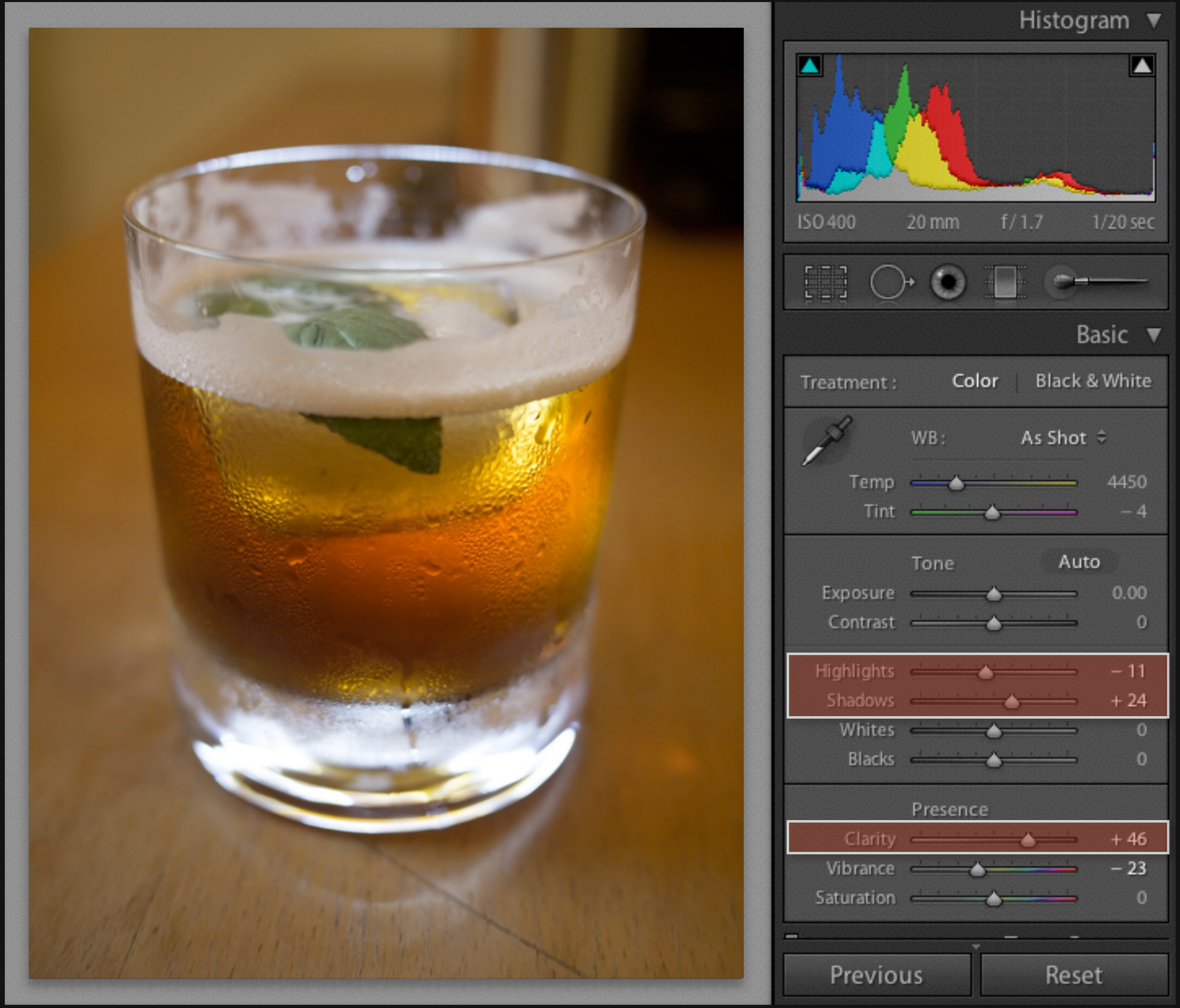
$O(x,y) \leftarrow (1-\alpha)\, I_2(x,y,k) + \alpha\, I_2(x,y,k+1)$

local Laplacian filters

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

Local Laplacian Filters

prototype for Adobe Photoshop Camera Raw / Lightroom



Local Laplacian Filters

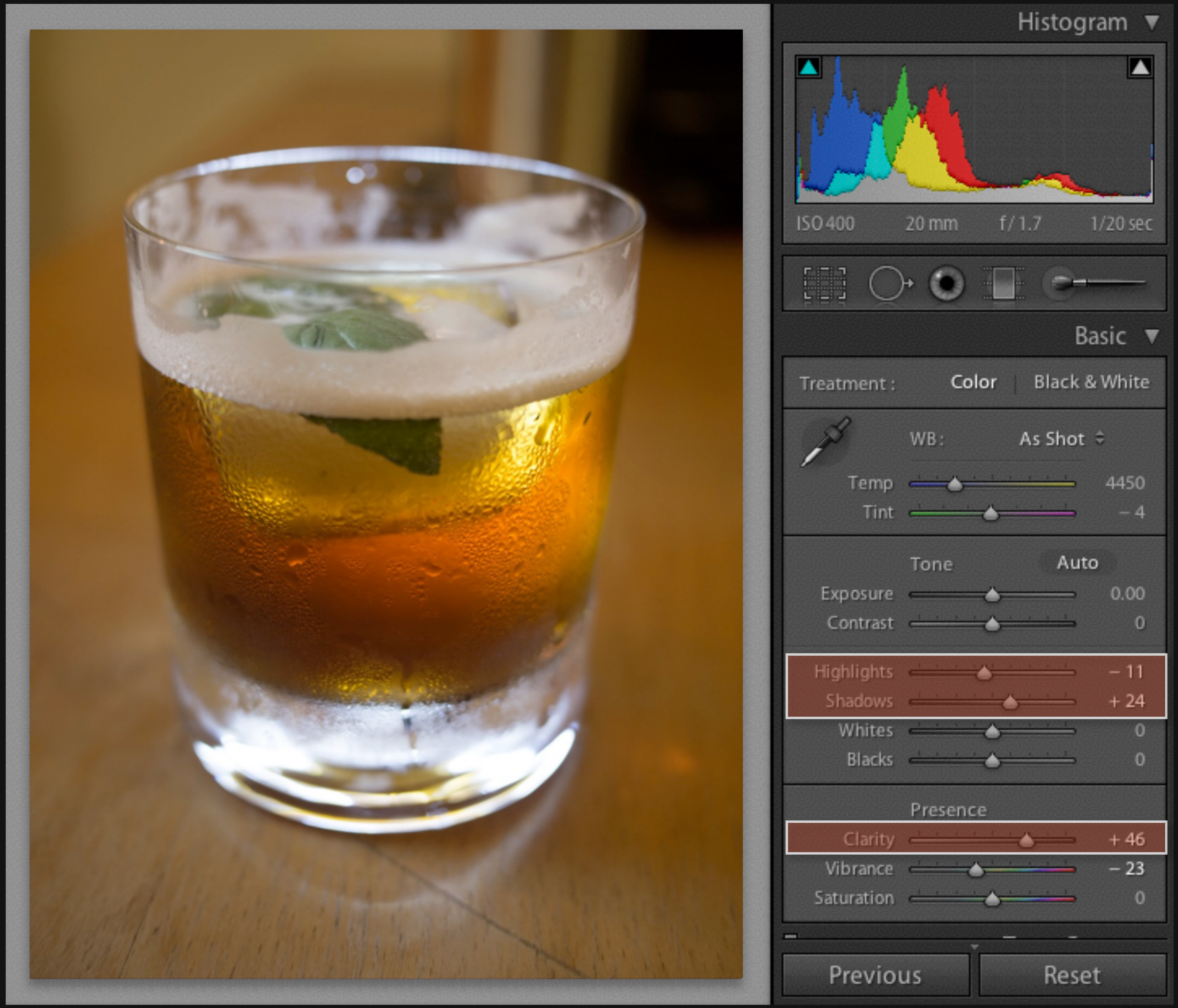
prototype for Adobe Photoshop Camera Raw / Lightroom

Adobe: 1500 lines of expert-optimized C++

multi-threaded, SSE

3 months of work

10x faster than original C++



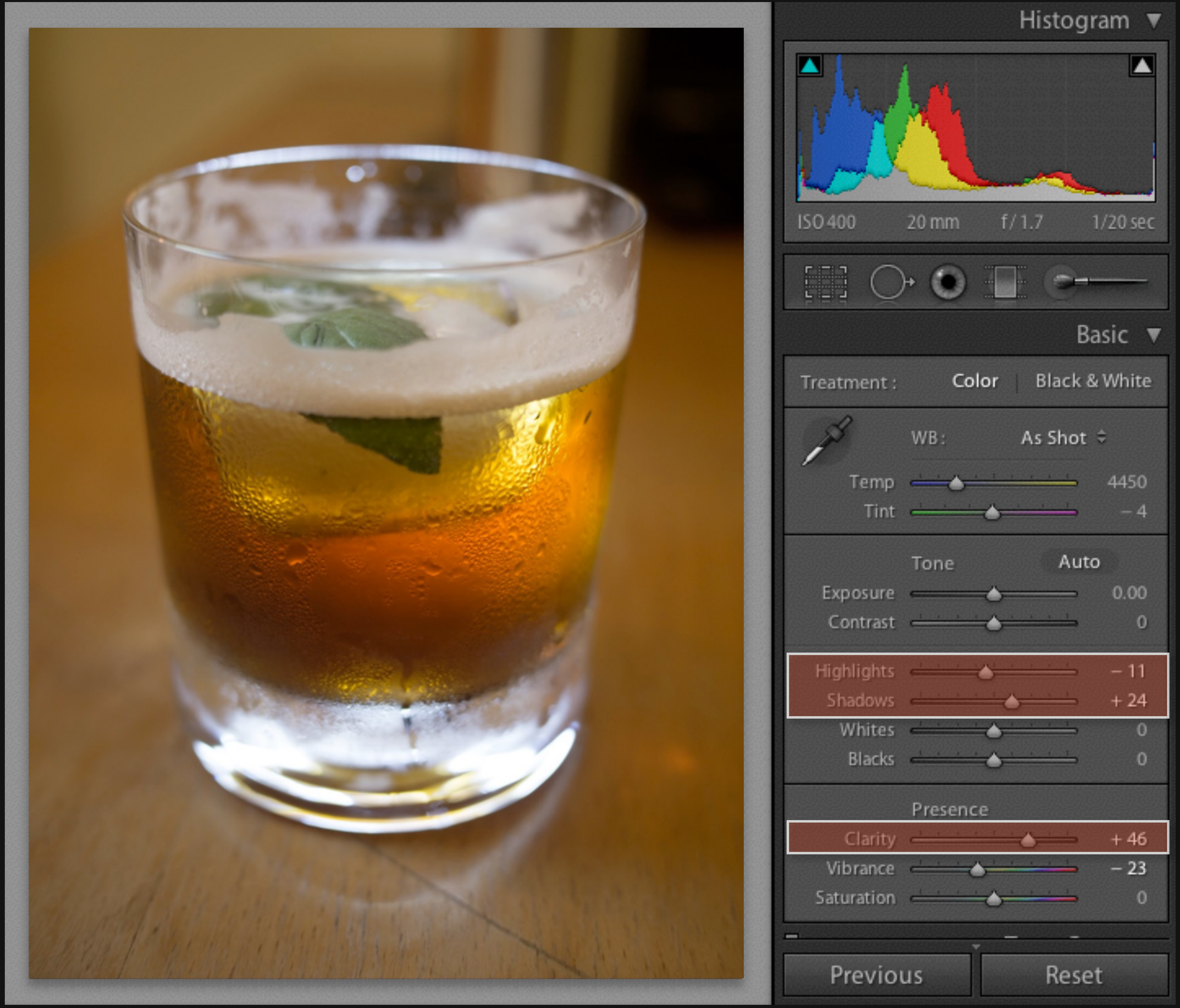
Local Laplacian Filters

prototype for Adobe Photoshop Camera Raw / Lightroom

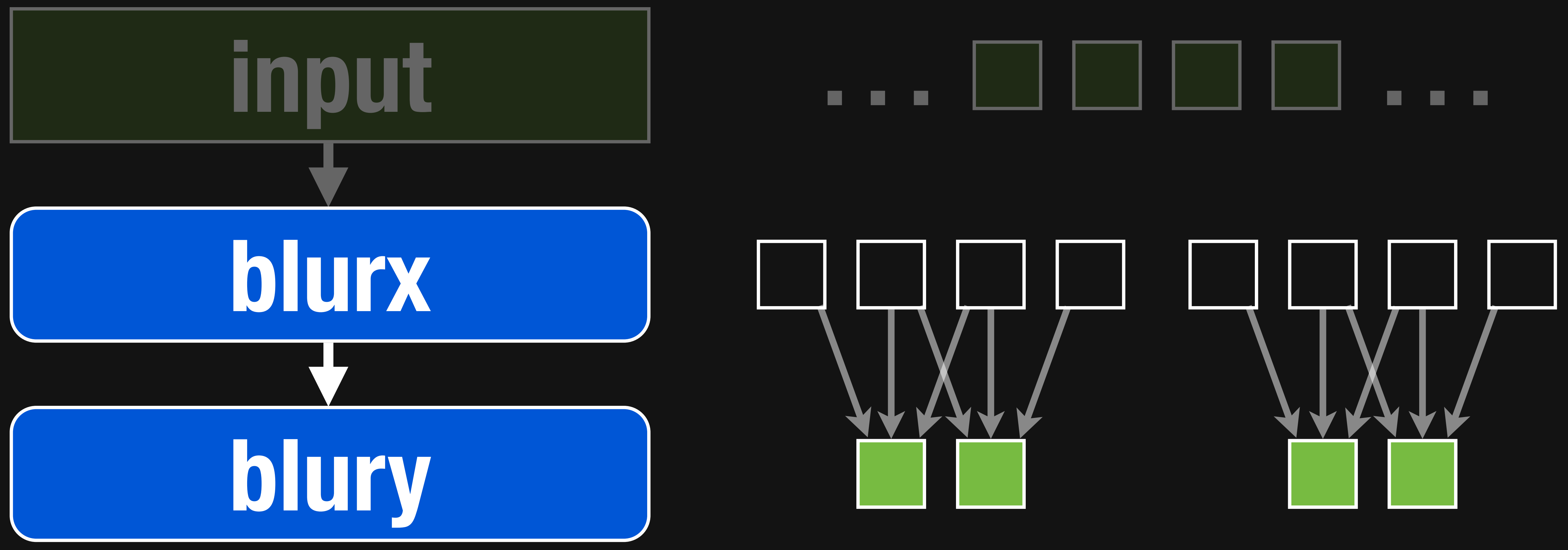
Adobe: 1500 lines of expert-optimized C++
multi-threaded, SSE
3 months of work
10x faster than original C++

Halide: 60 lines
1 intern-day

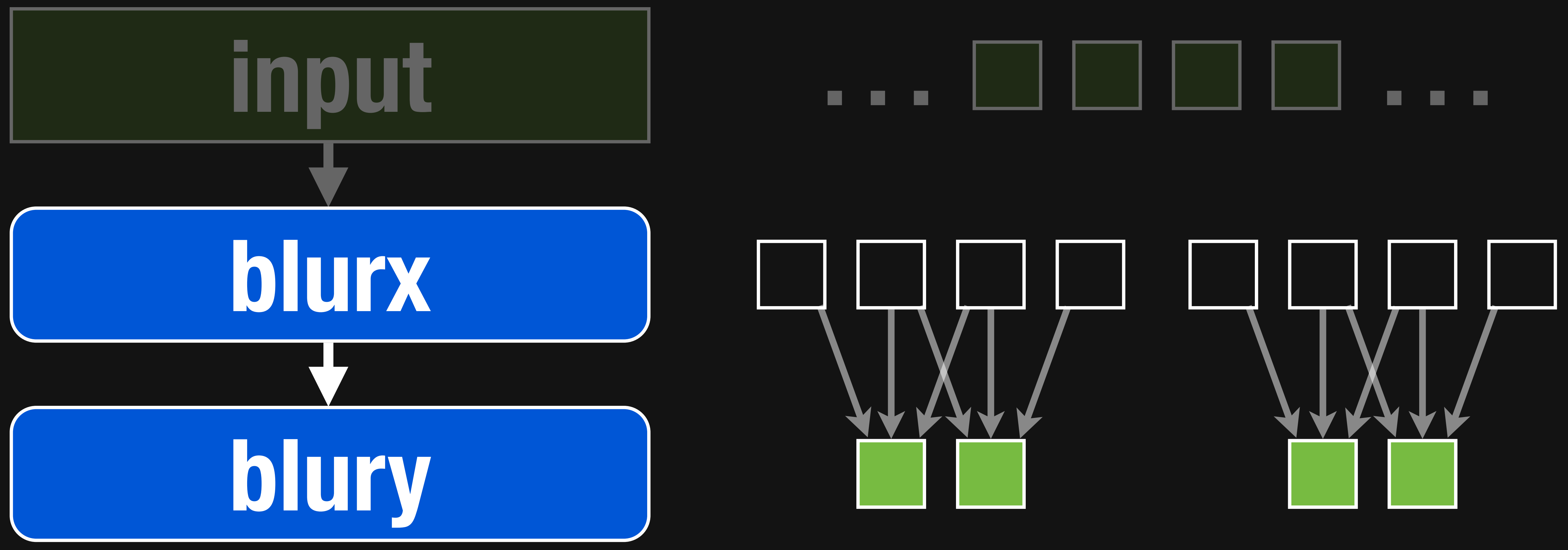
Halide vs. Adobe:
2x faster on same CPU,
9x faster on GPU



Message #1: performance requires tradeoffs

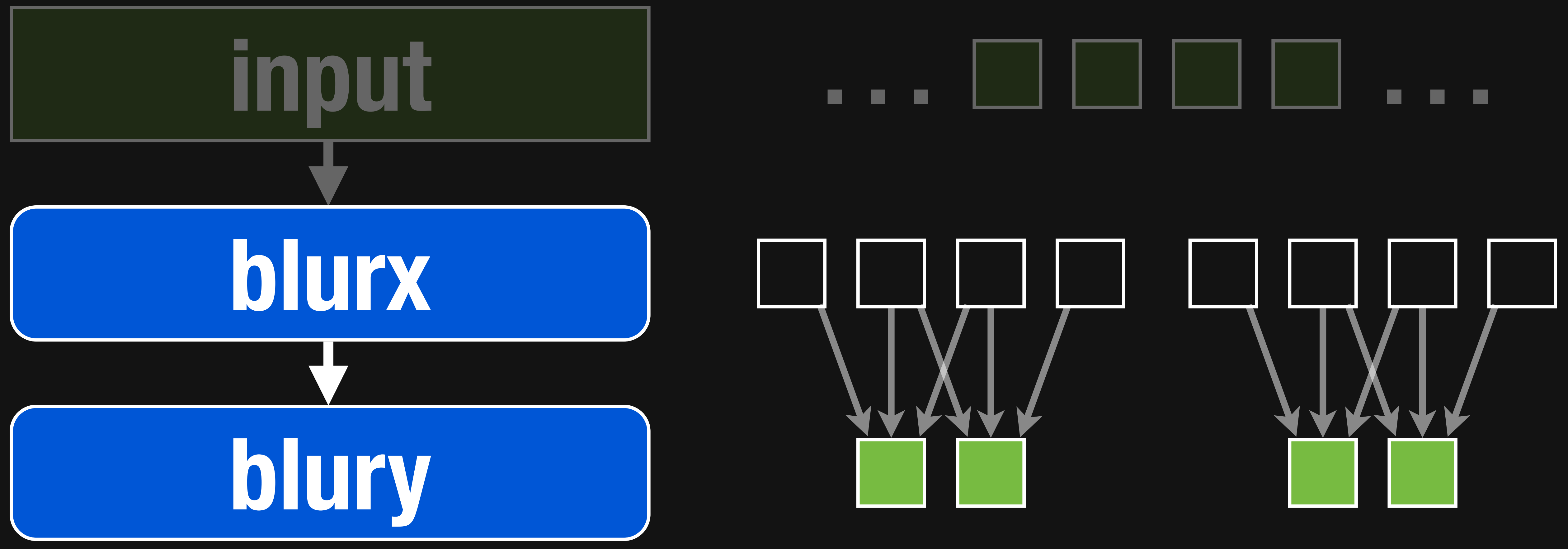


Message #1: performance requires tradeoffs



redundant work ——— locality

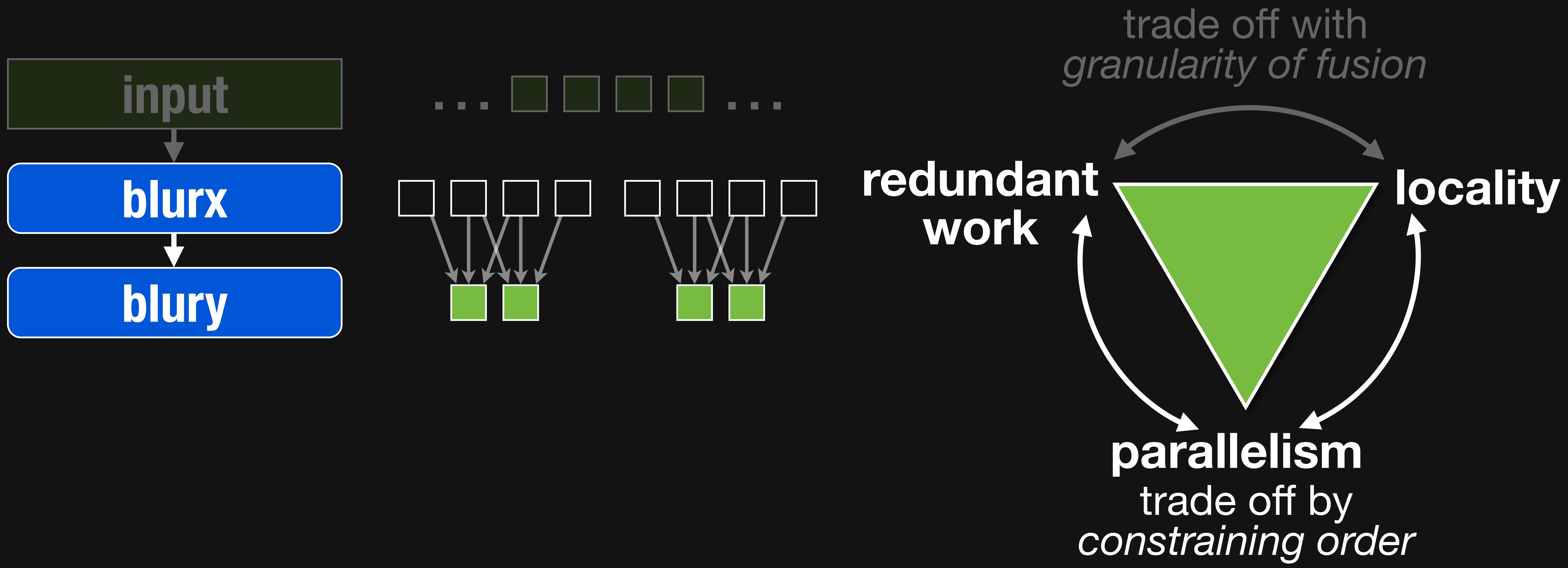
Message #1: performance requires tradeoffs



trade off with
granularity of fusion

redundant work — **locality**

Message #1: performance requires tradeoffs

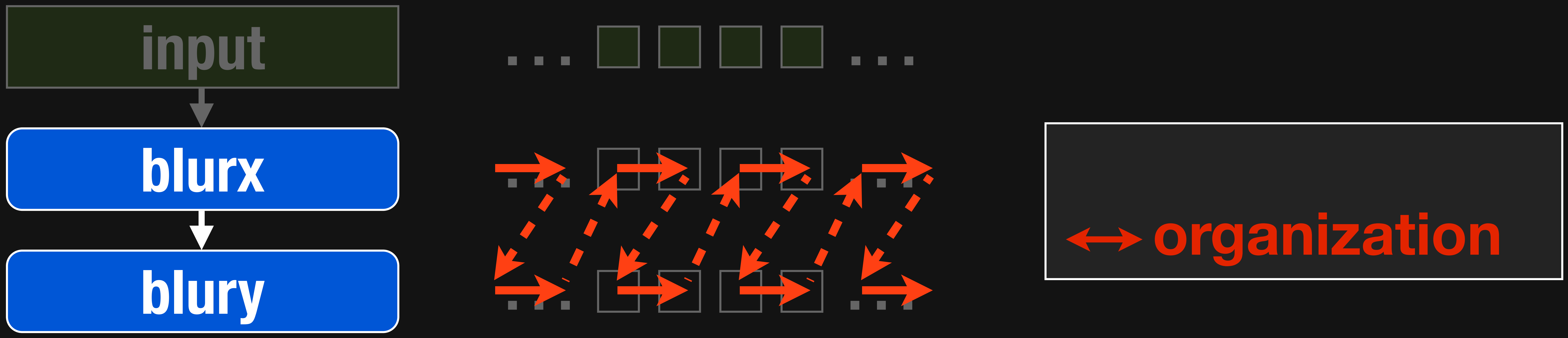


Message #2: algorithm vs. organization

Message #2: algorithm vs. organization

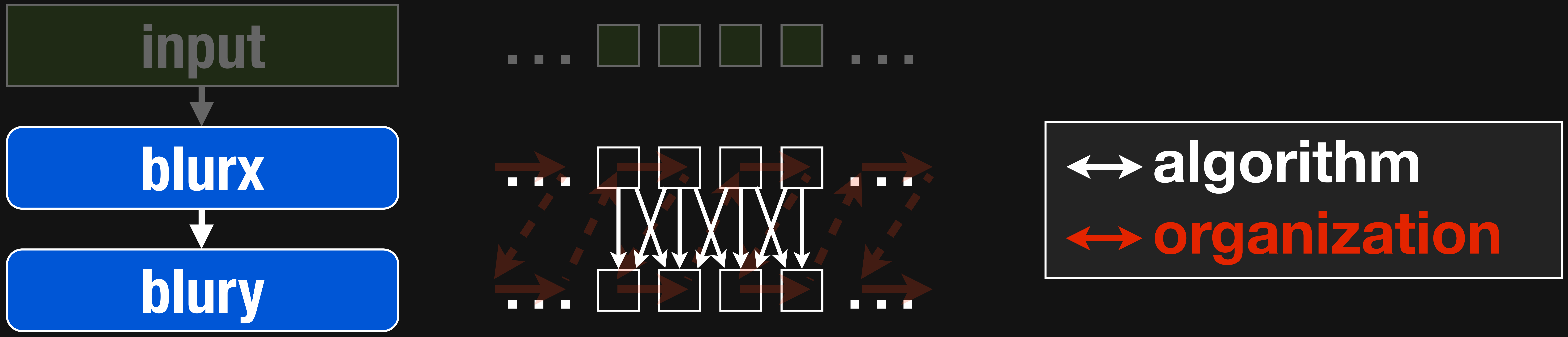
order and interleaving
radically alter performance
of the *same algorithm*

Message #2: algorithm vs. organization



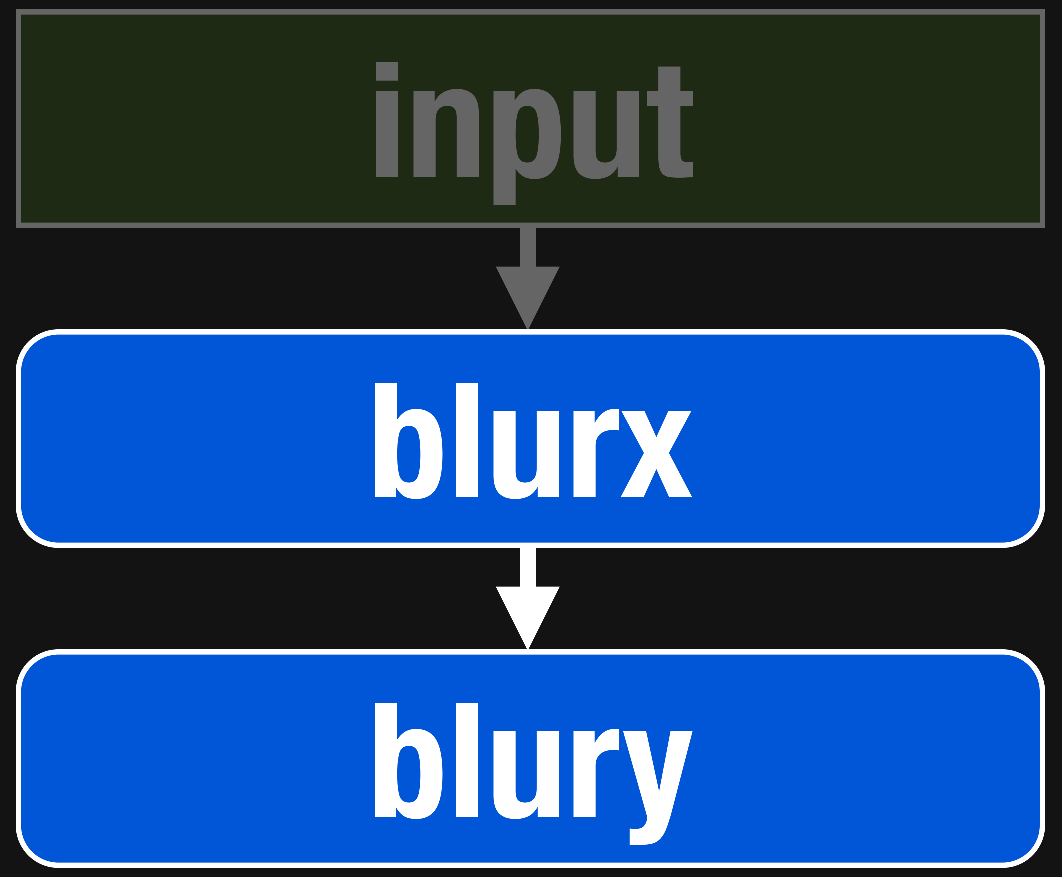
order and interleaving
radically alter performance
of the *same algorithm*

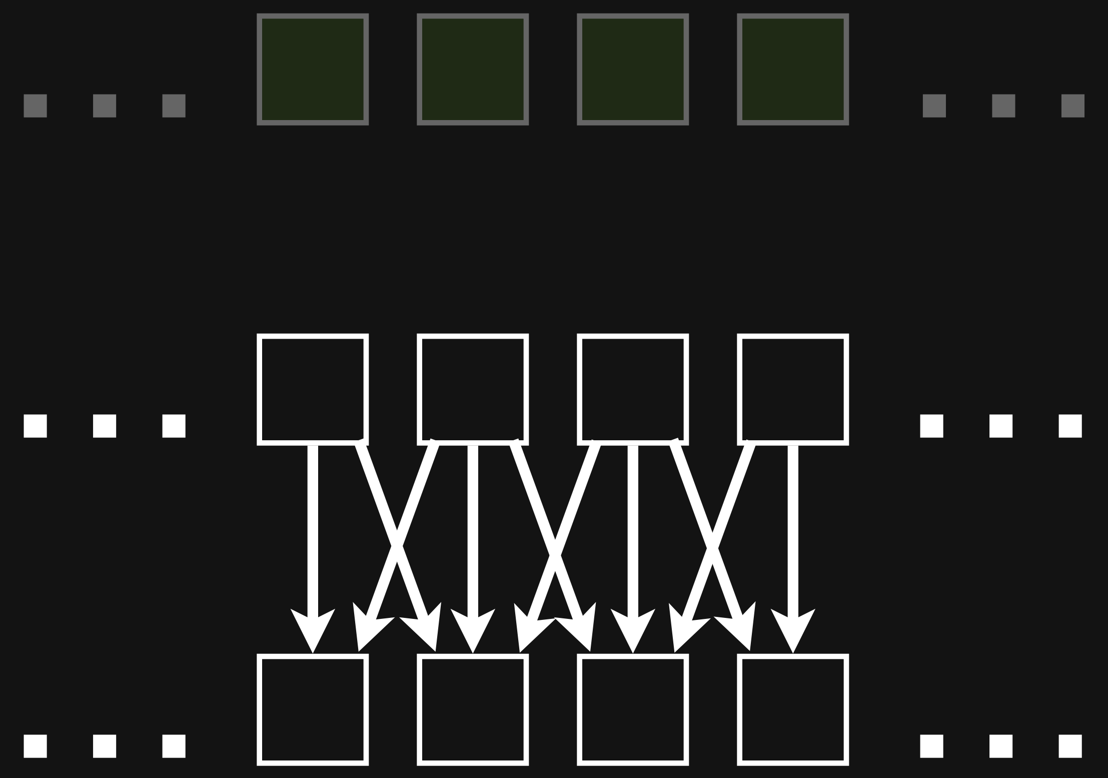
Message #2: algorithm vs. organization



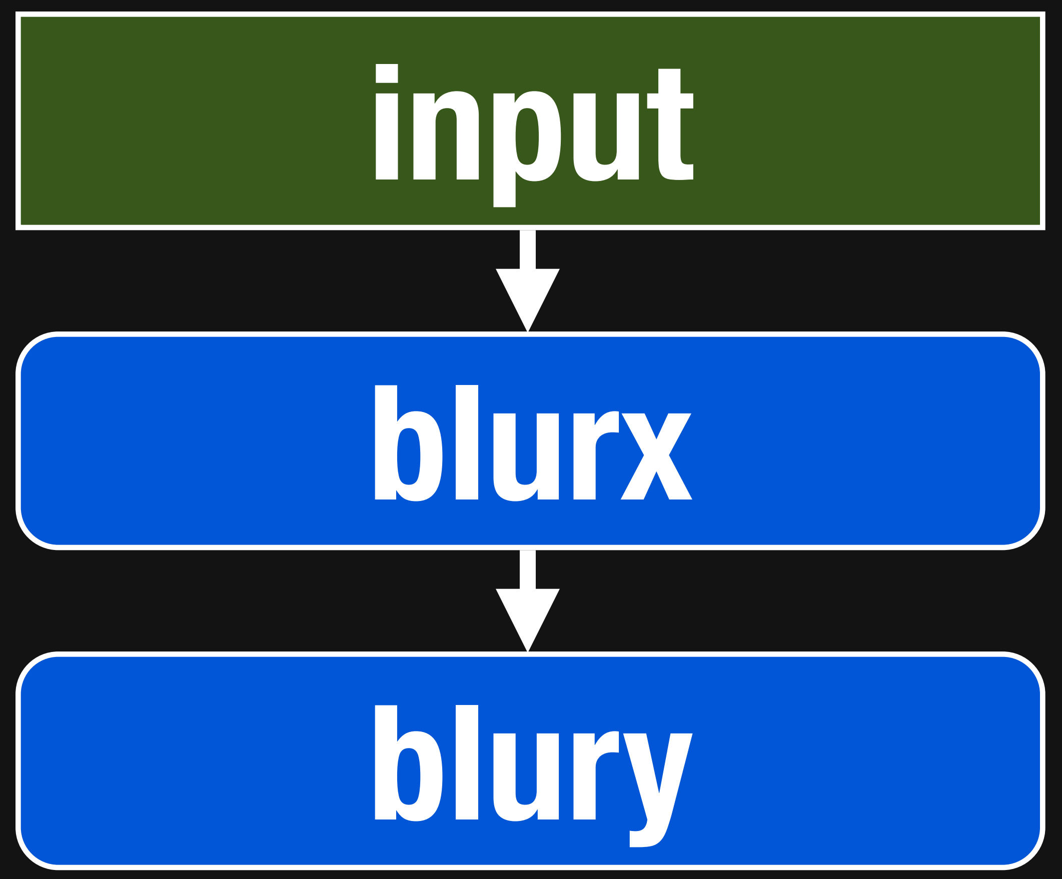
order and interleaving
radically alter performance
of the *same algorithm*

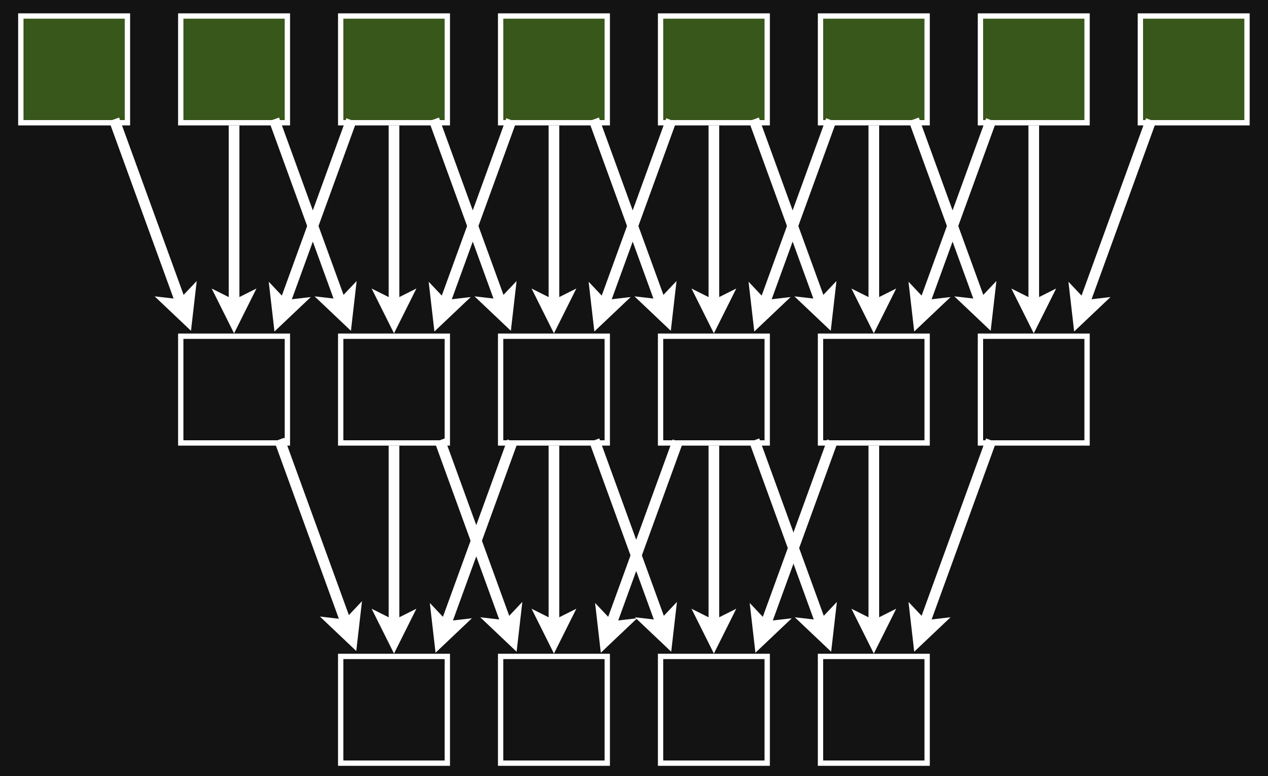
Message #3: dependencies limit choices of organization

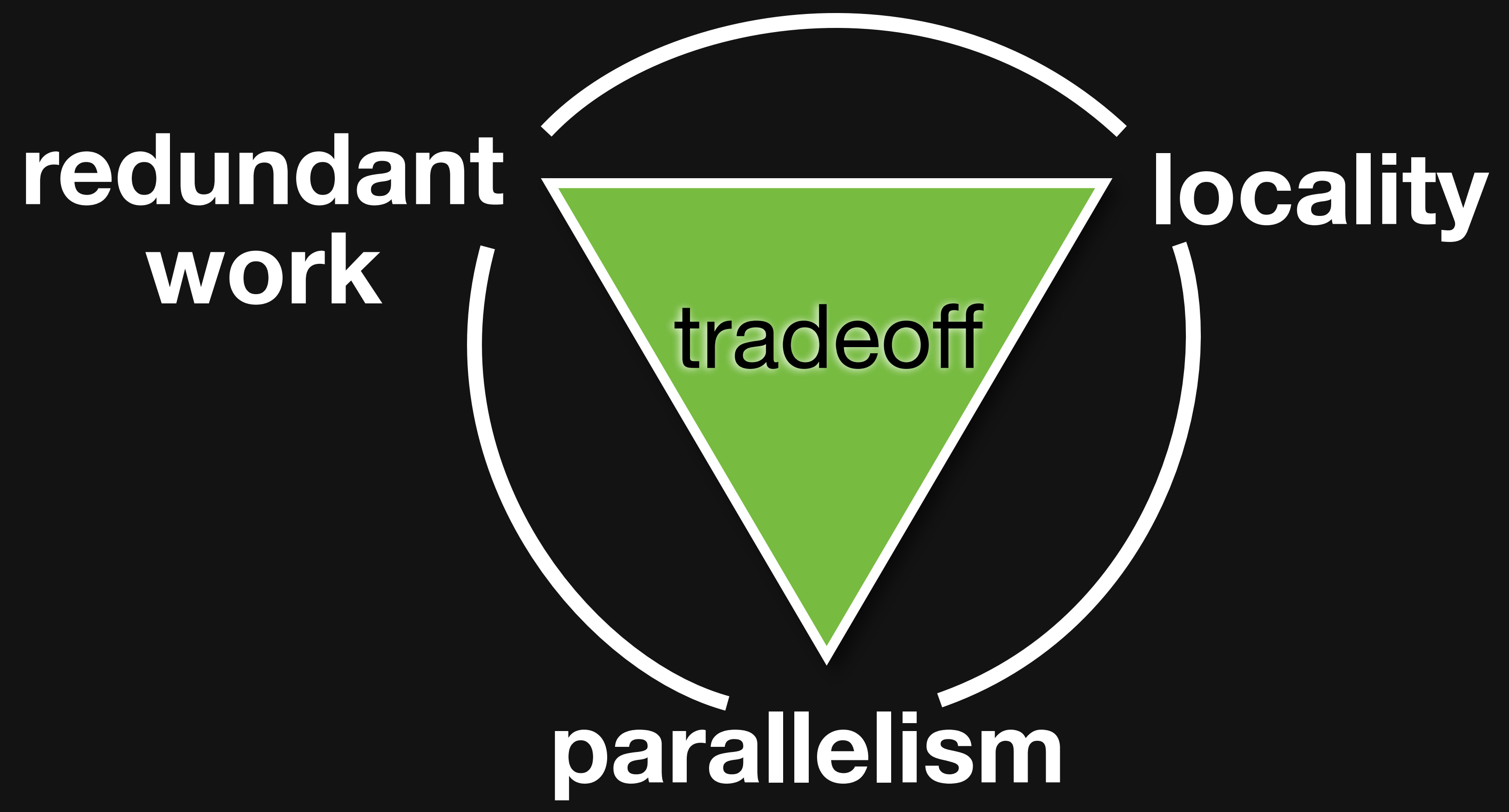




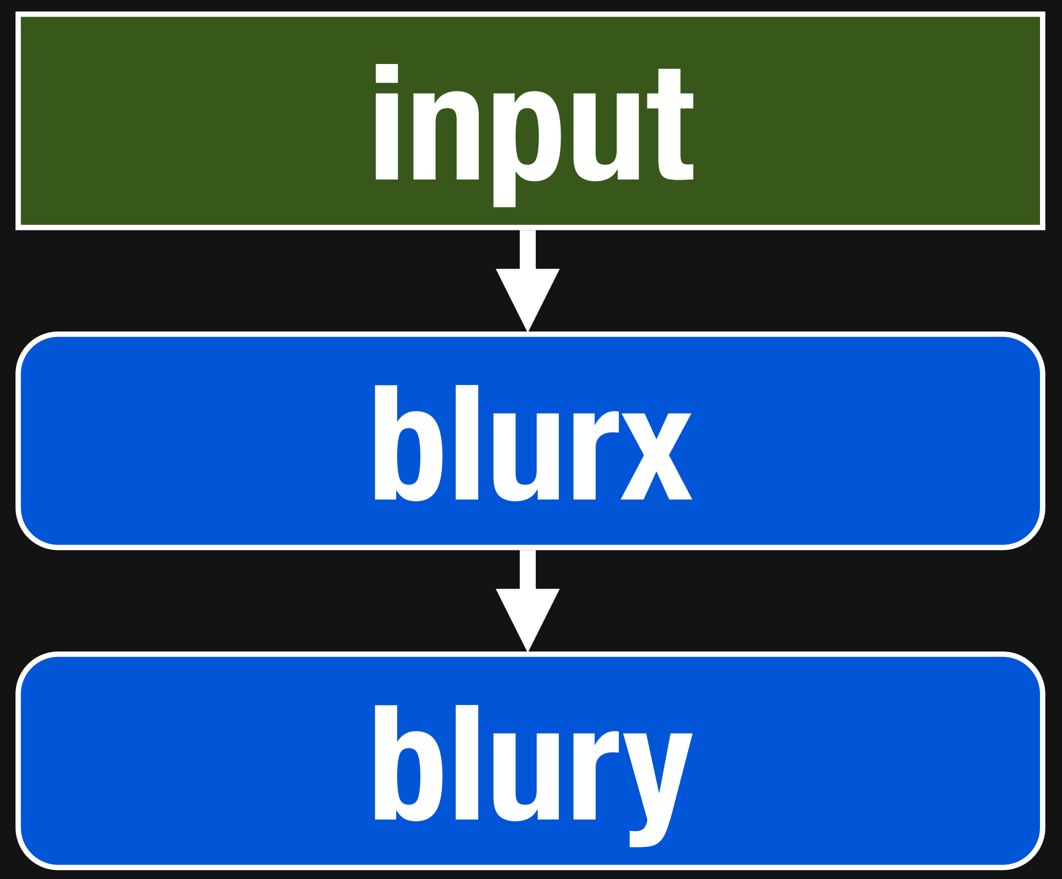
This is a general **task graph**

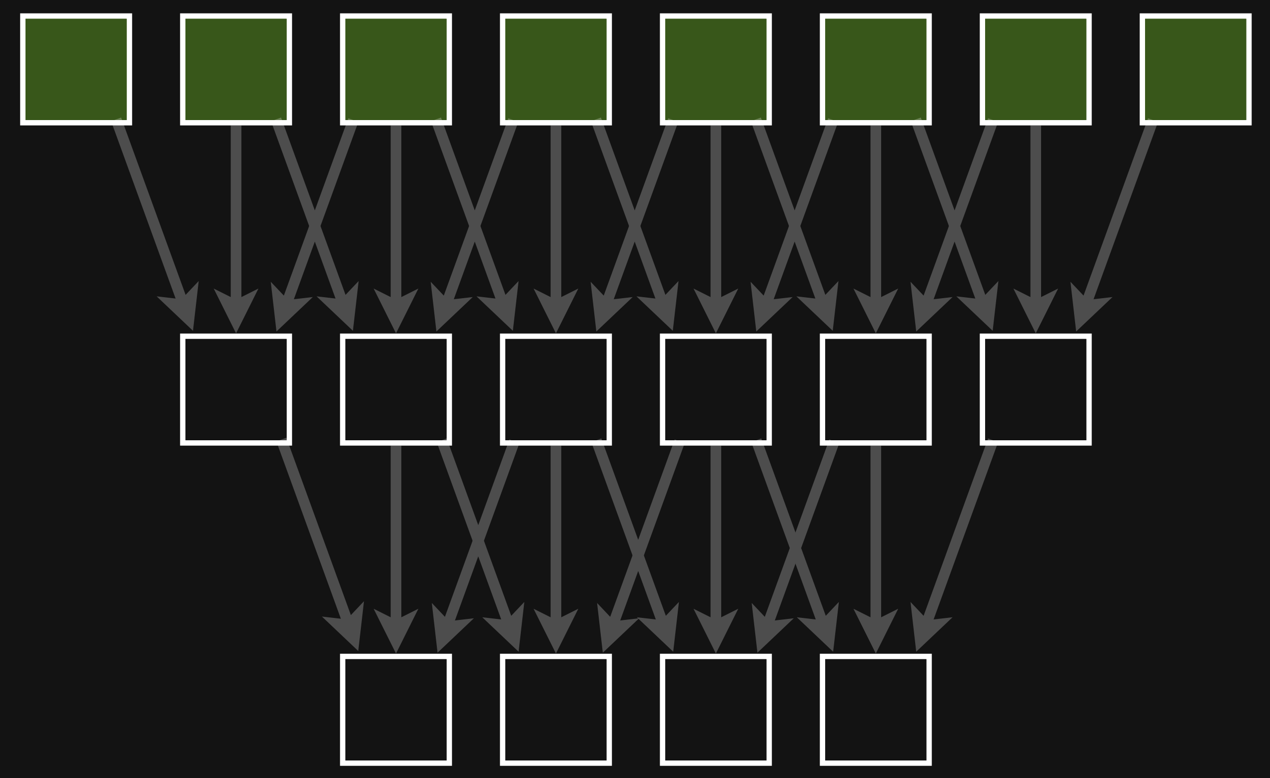


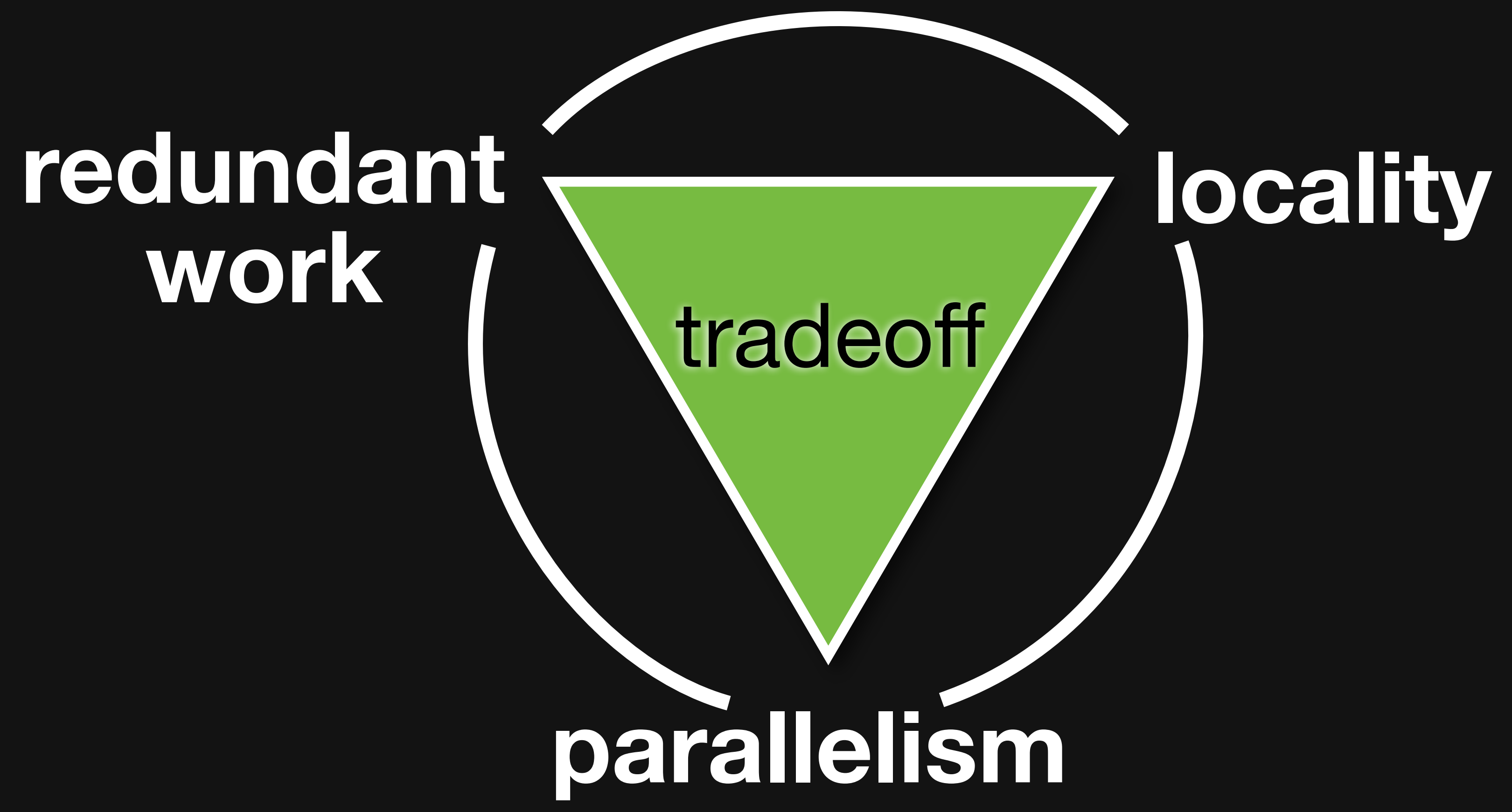




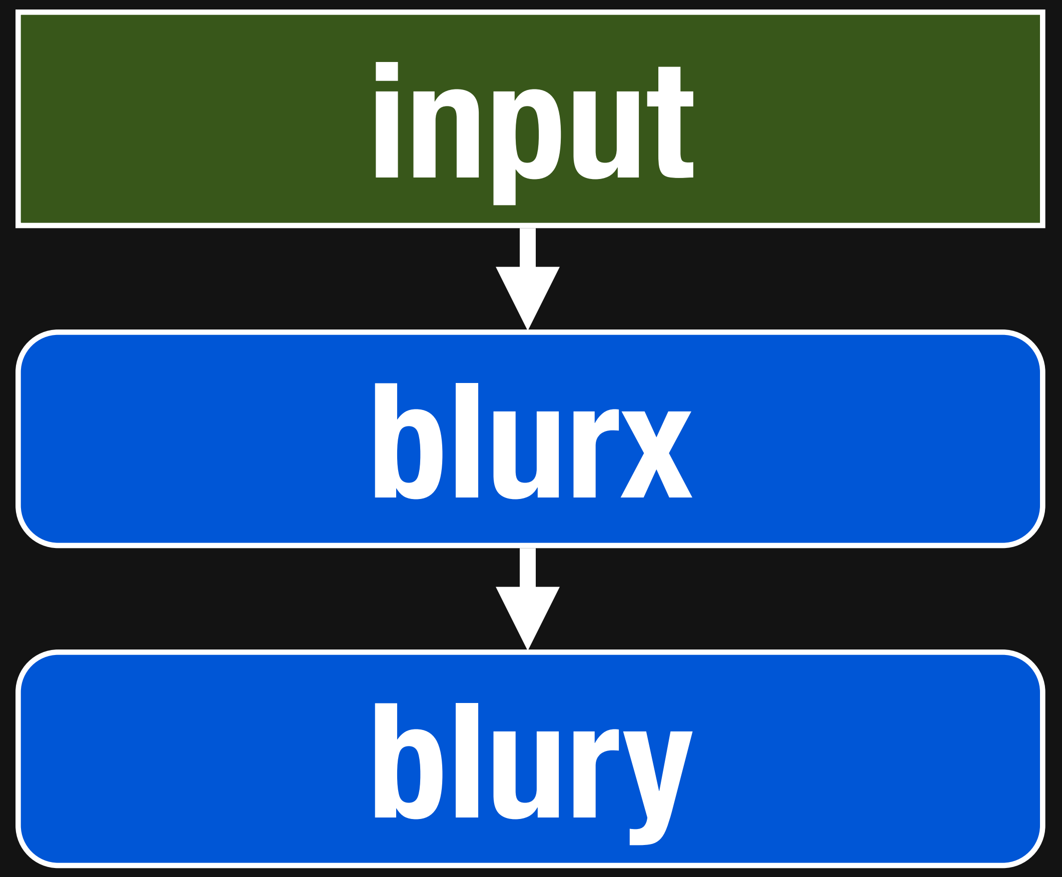
This is a general **task graph**

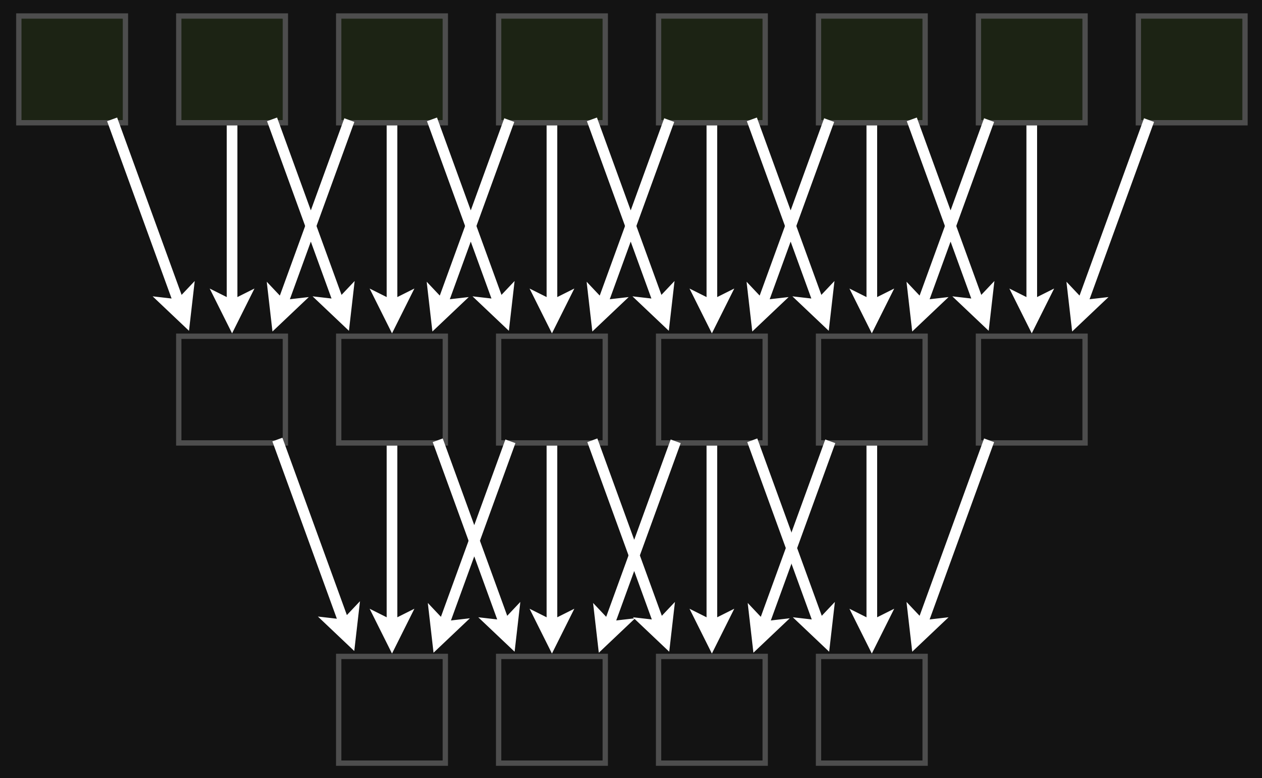




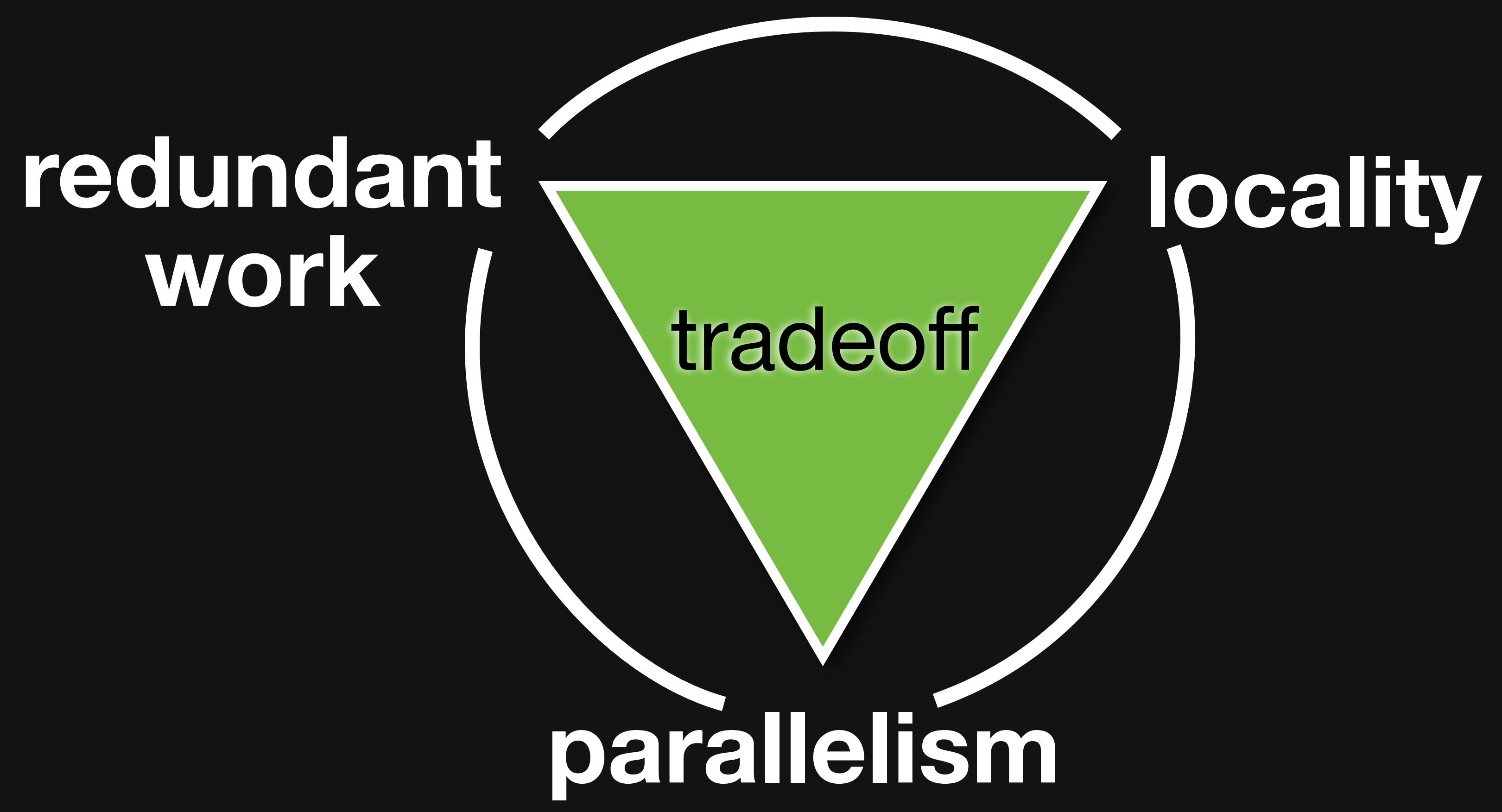


This is a general **task graph**

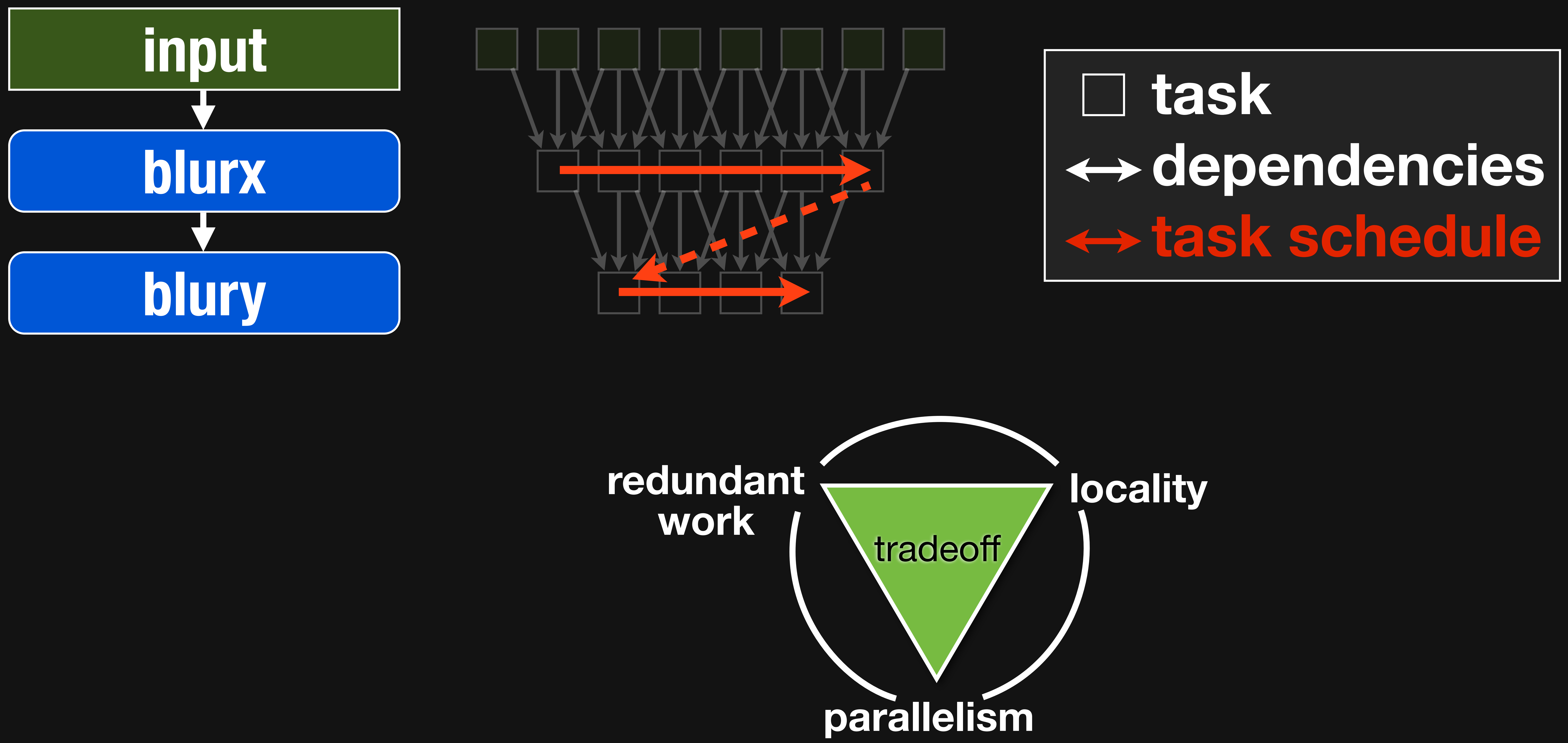




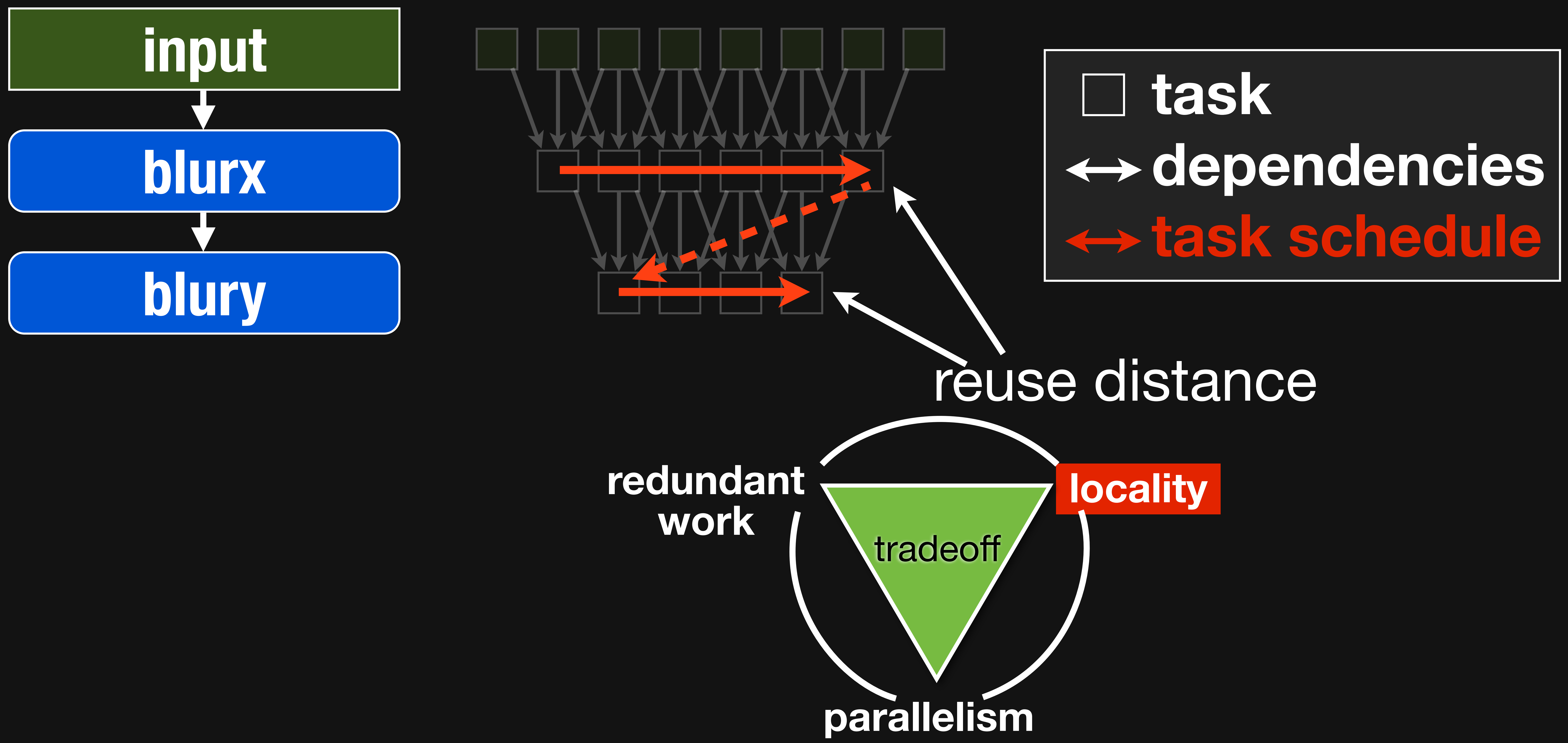
□ task
↔ dependencies
↔ task schedule



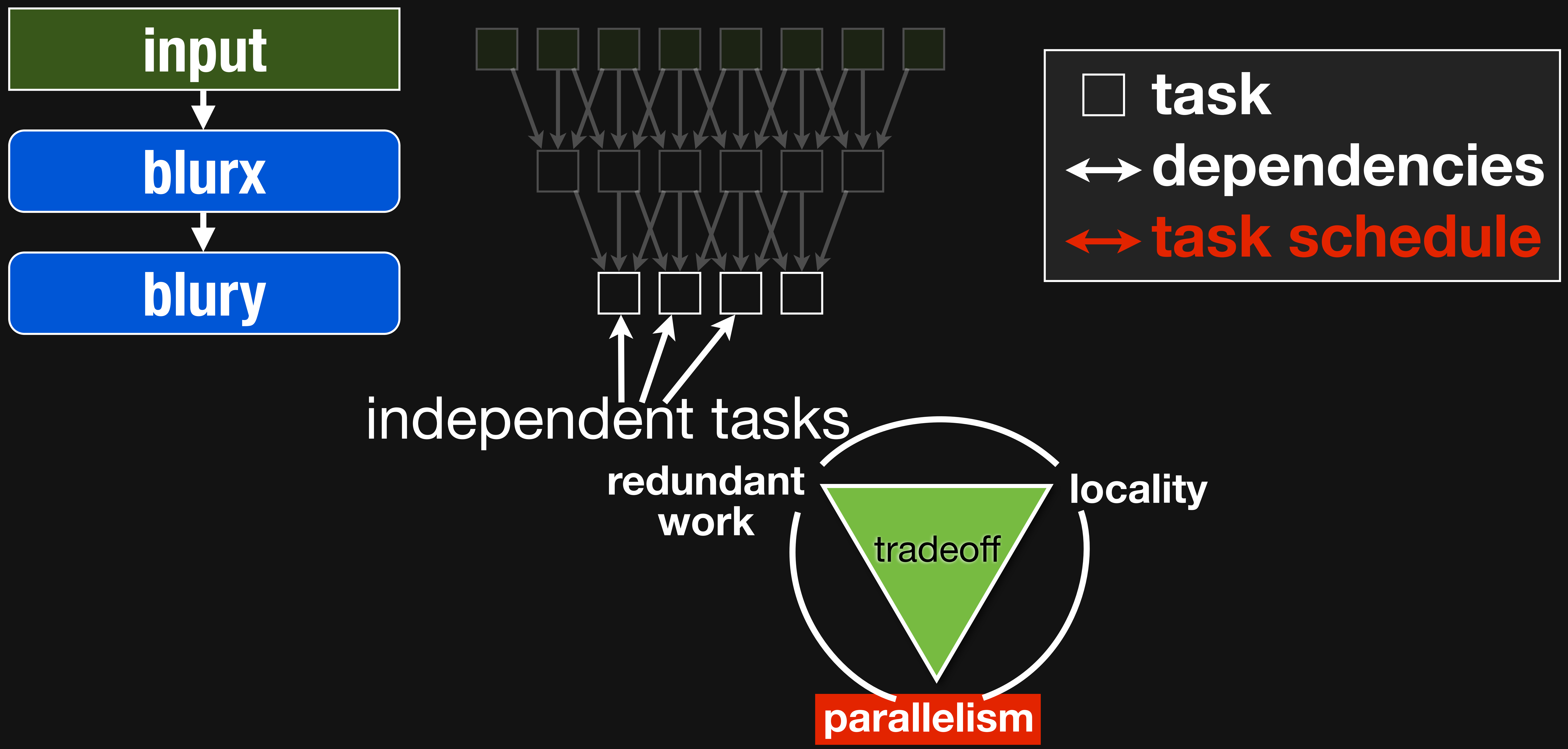
This is a general **task graph**



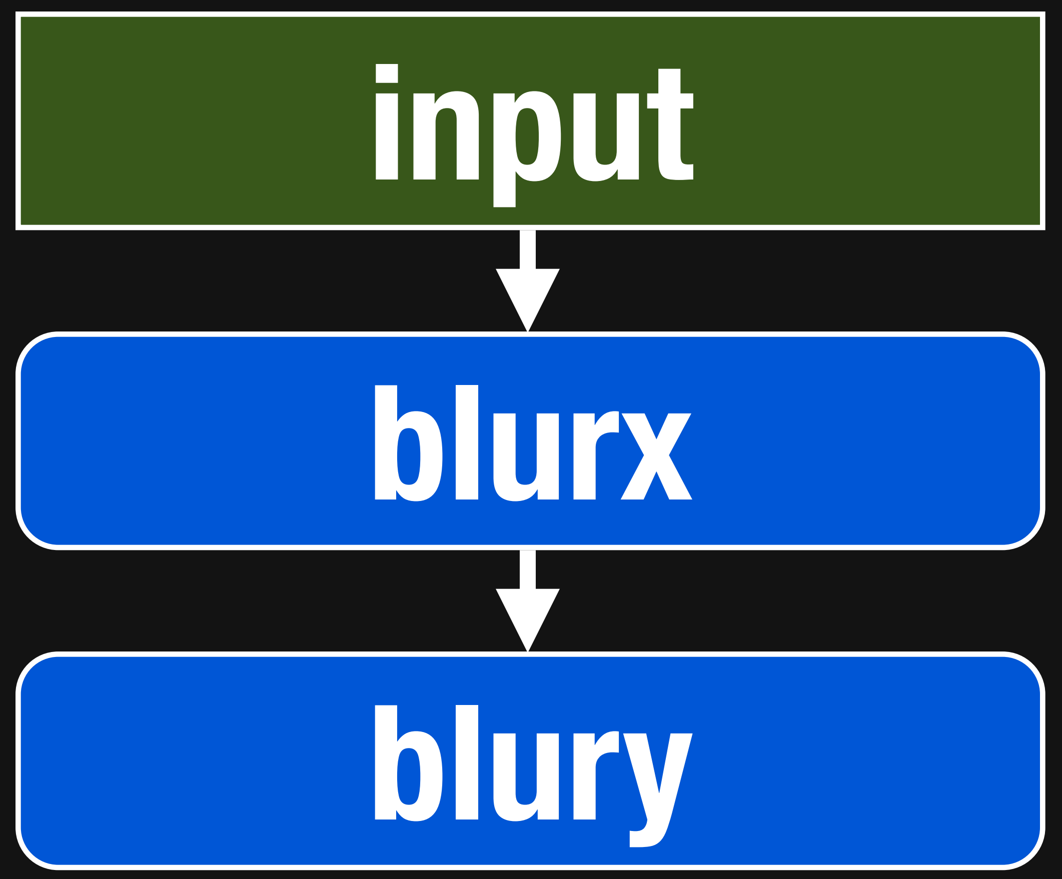
This is a general **task graph**

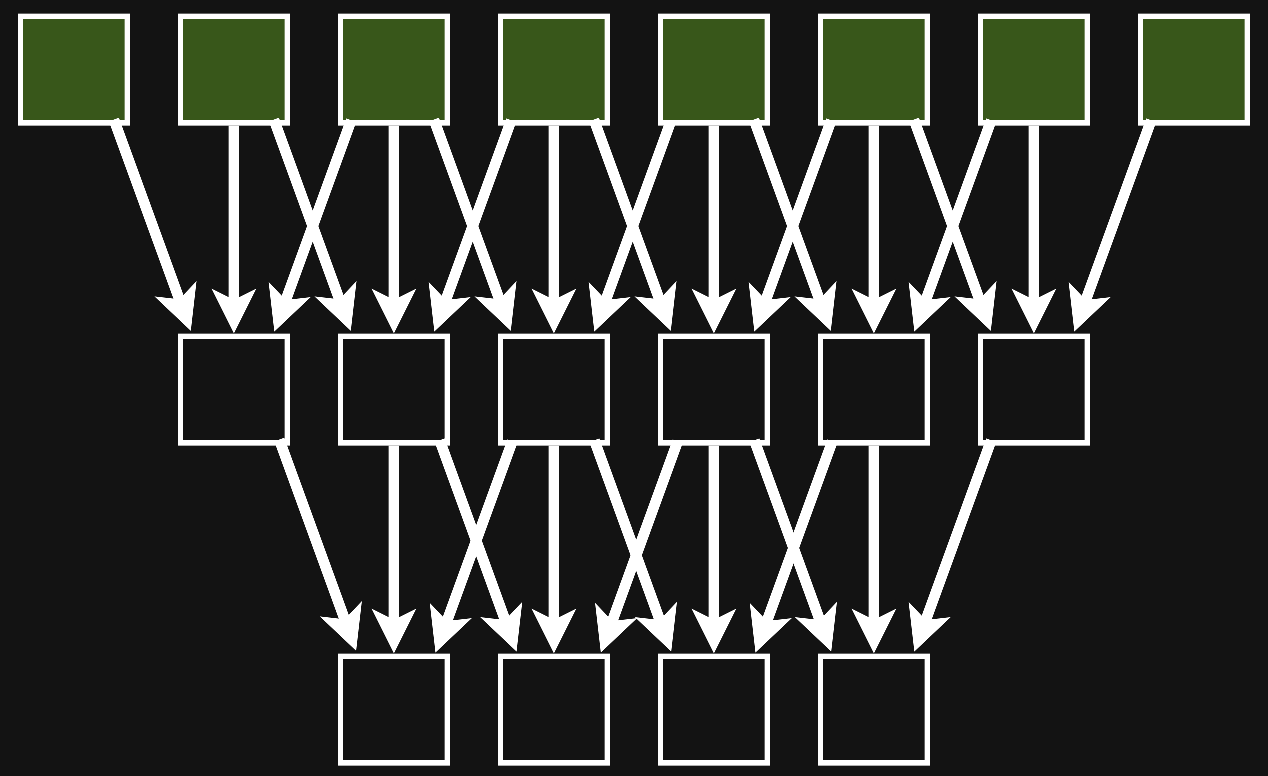


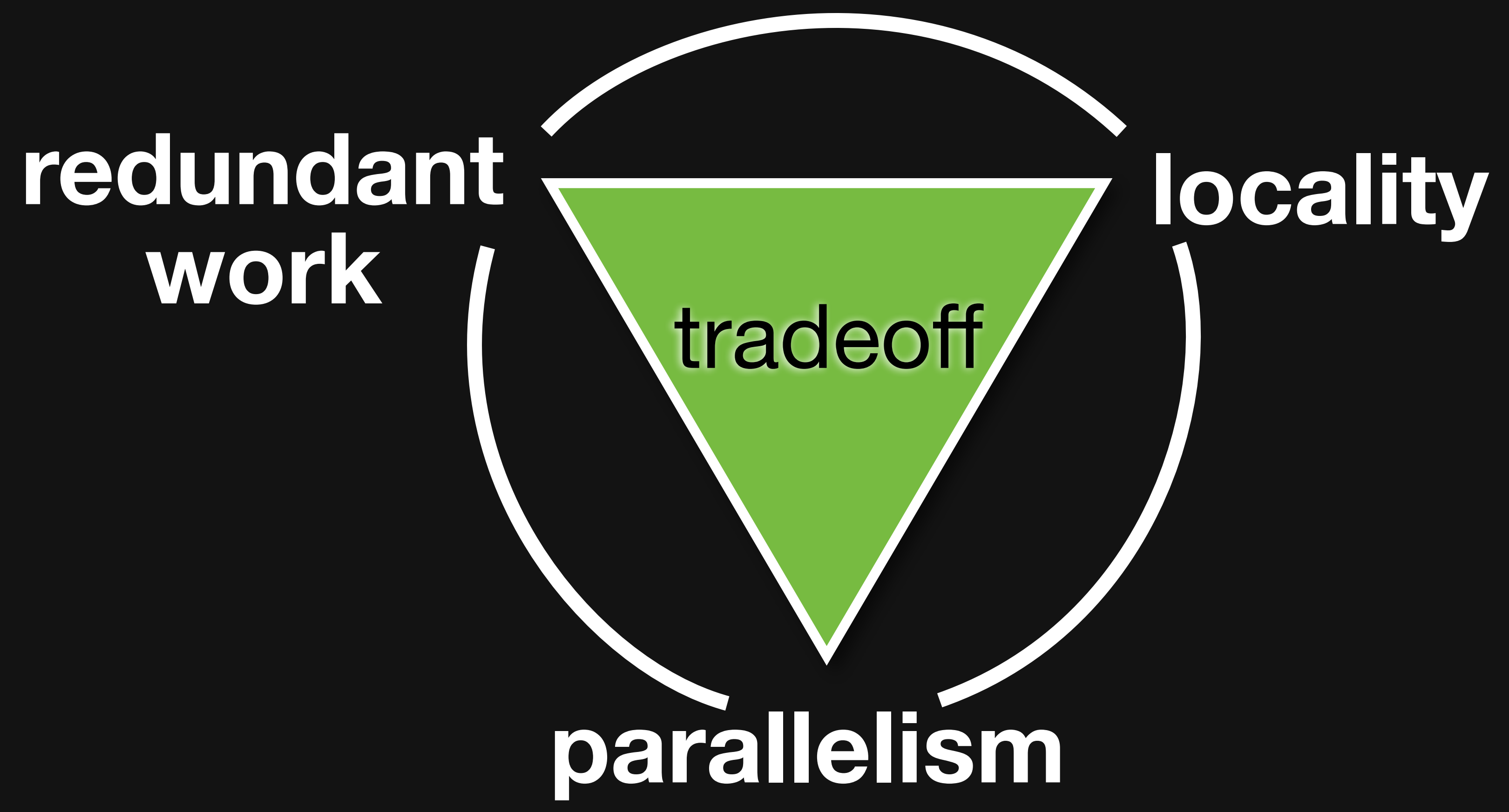
This is a general **task graph**



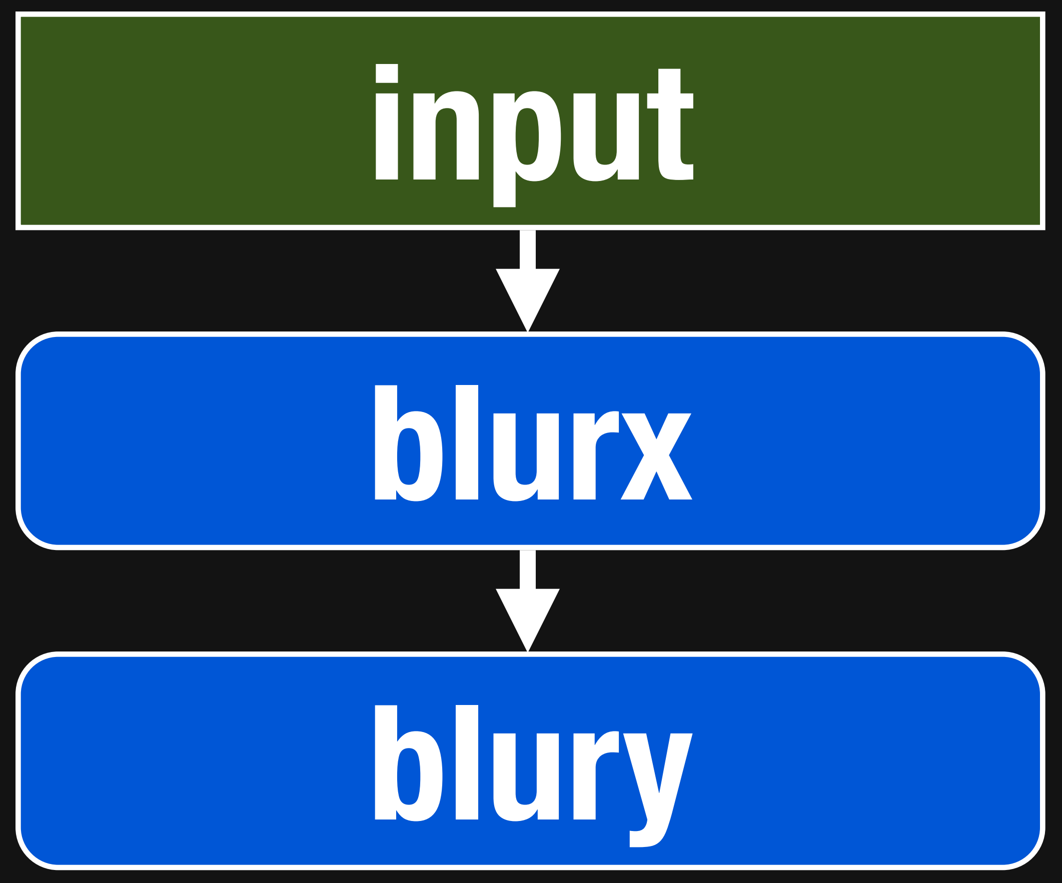
This is a general **task graph**

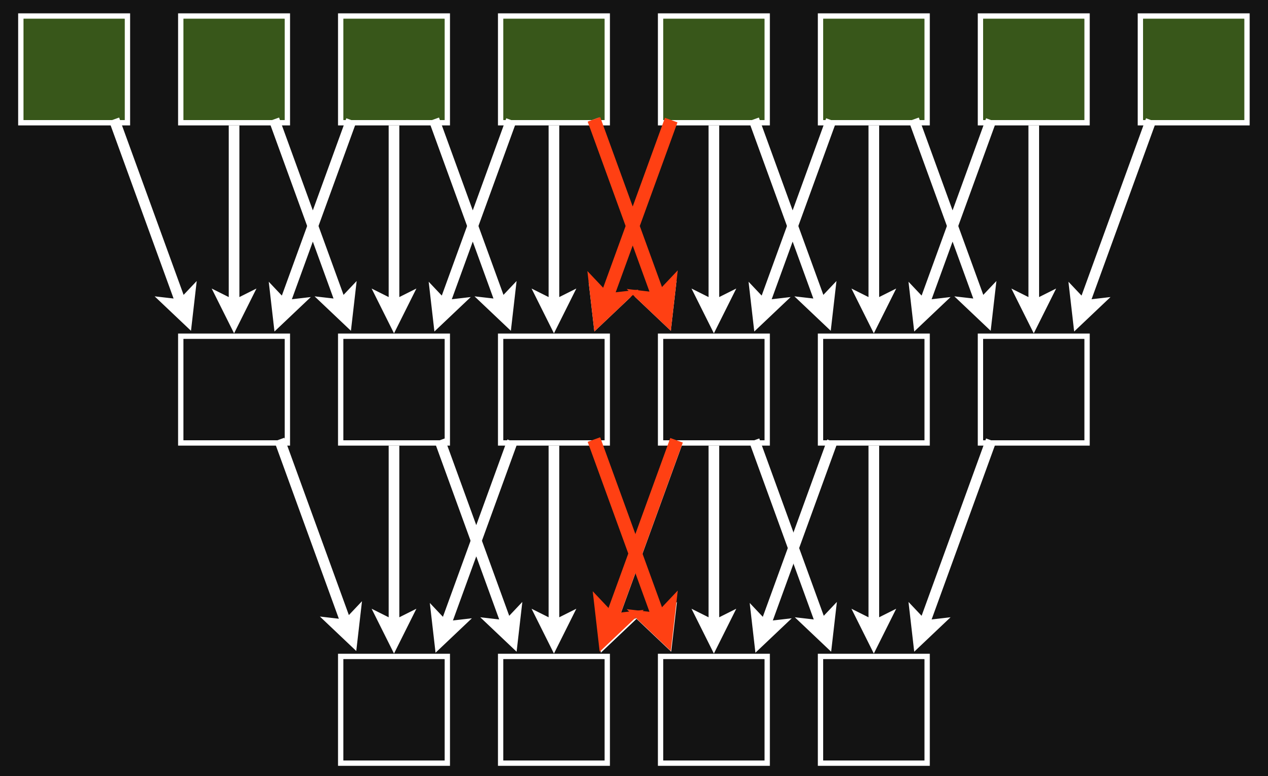


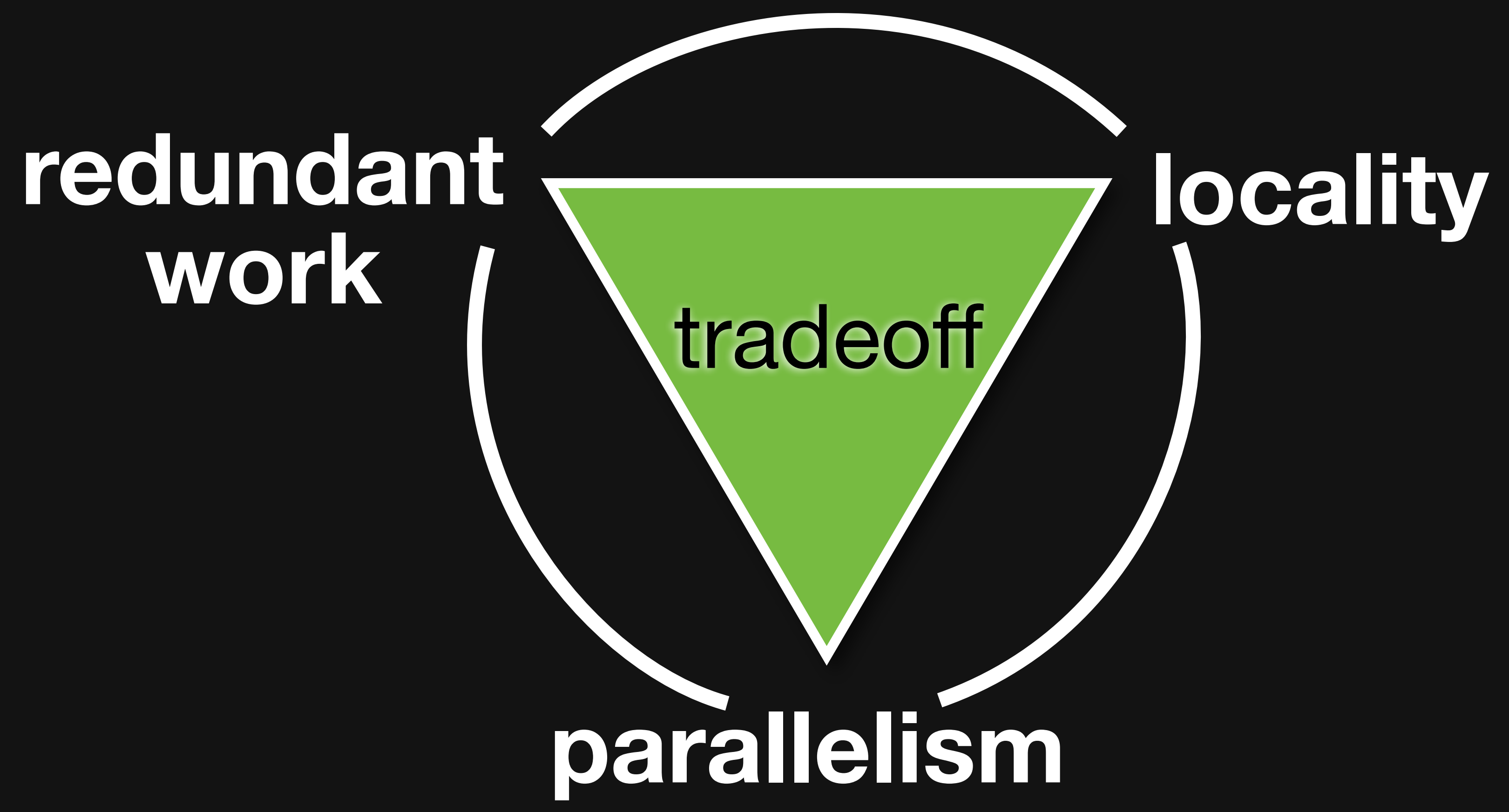




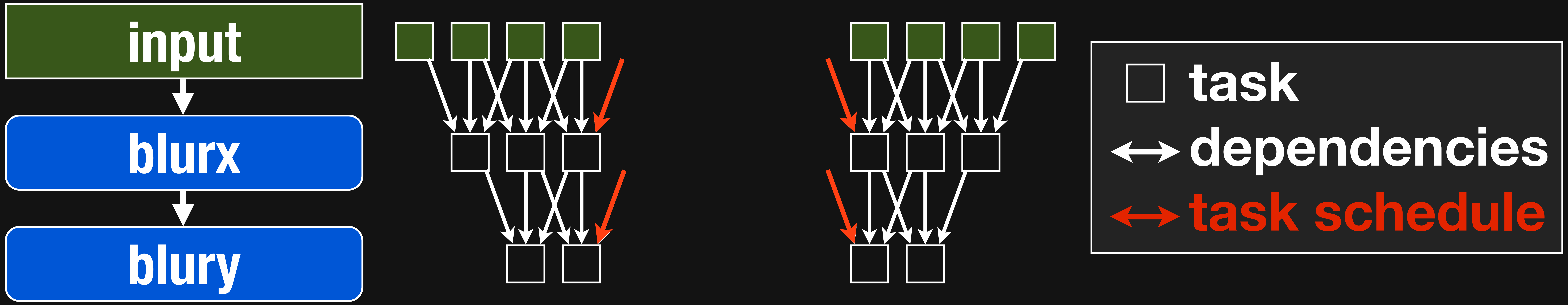
This is a general **task graph**

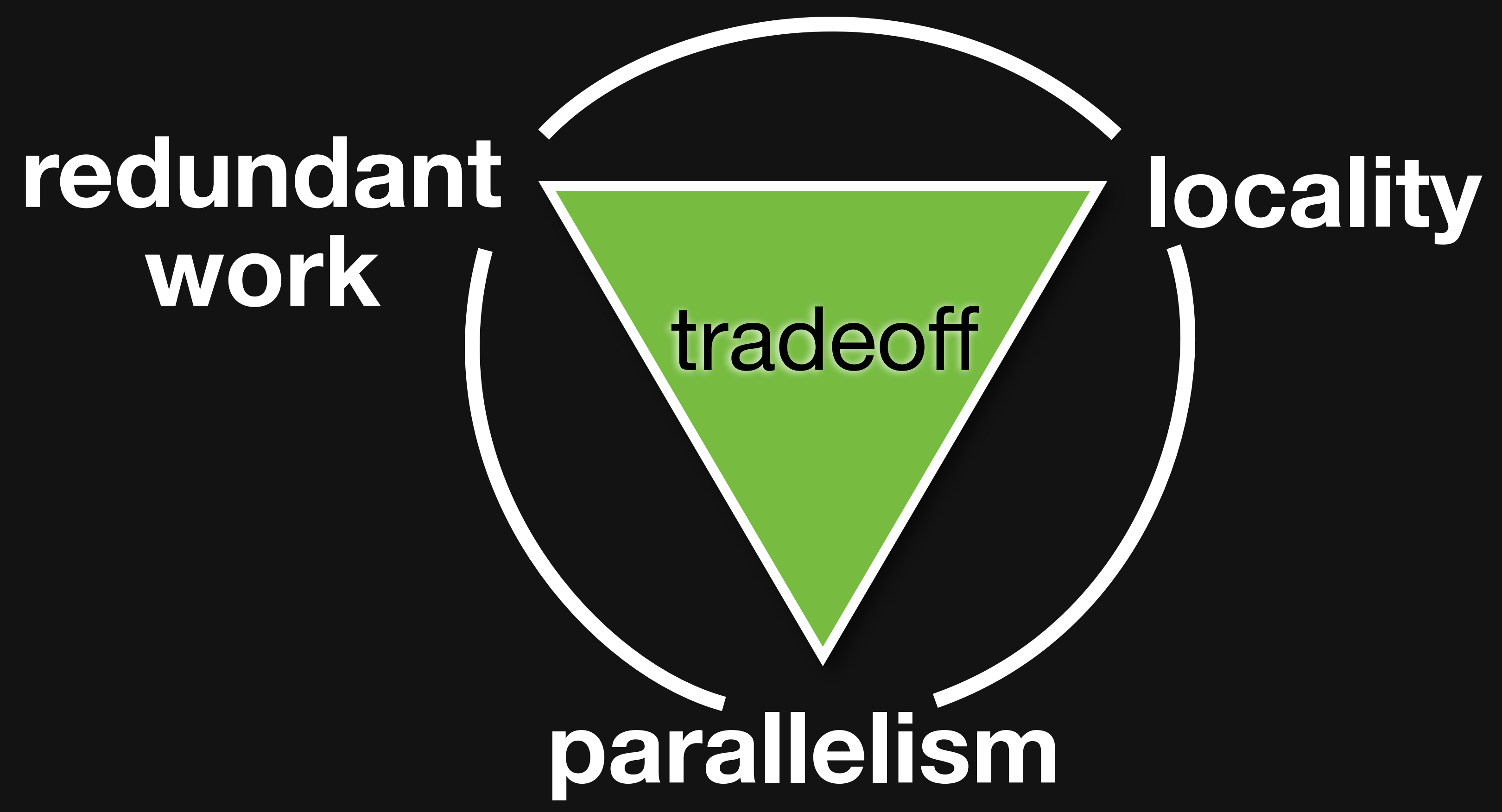




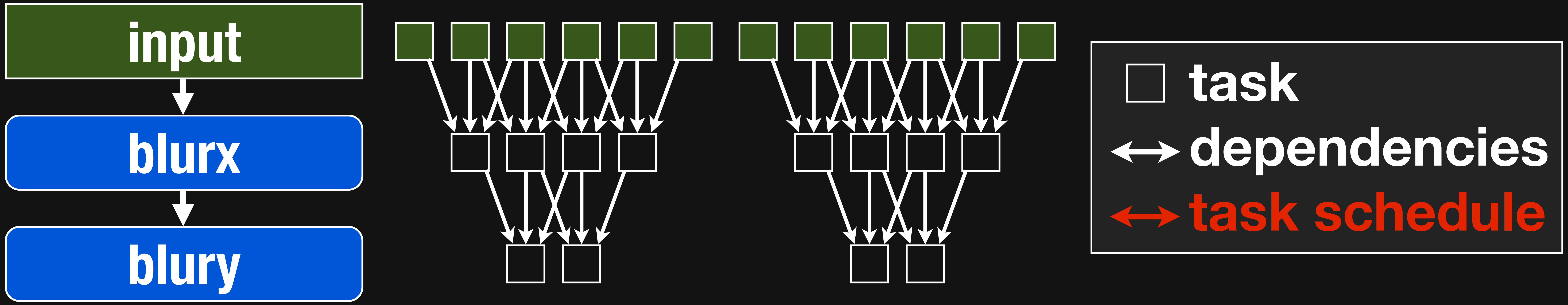


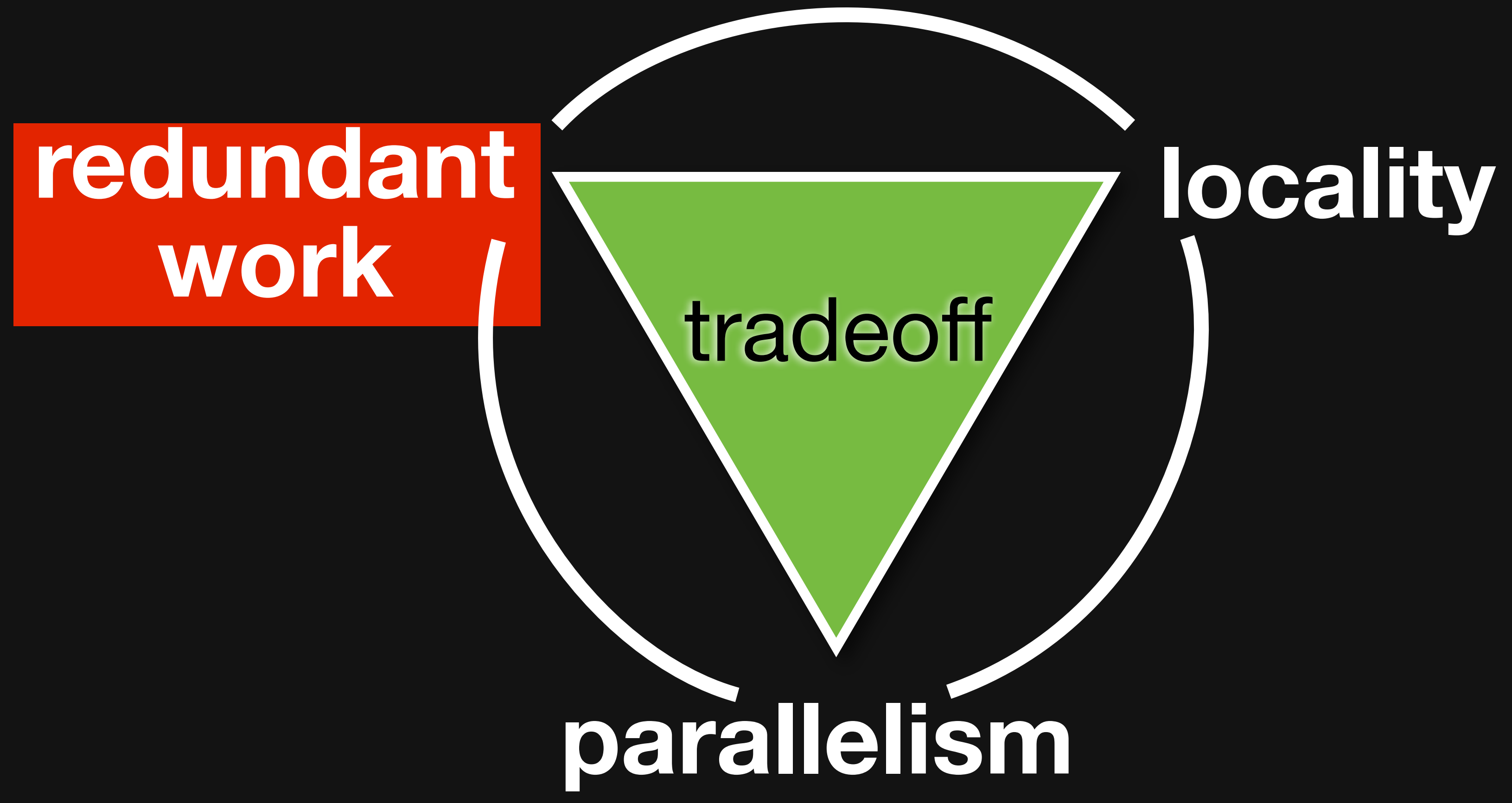
This is a general **task graph**





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

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);
}}}}}
```

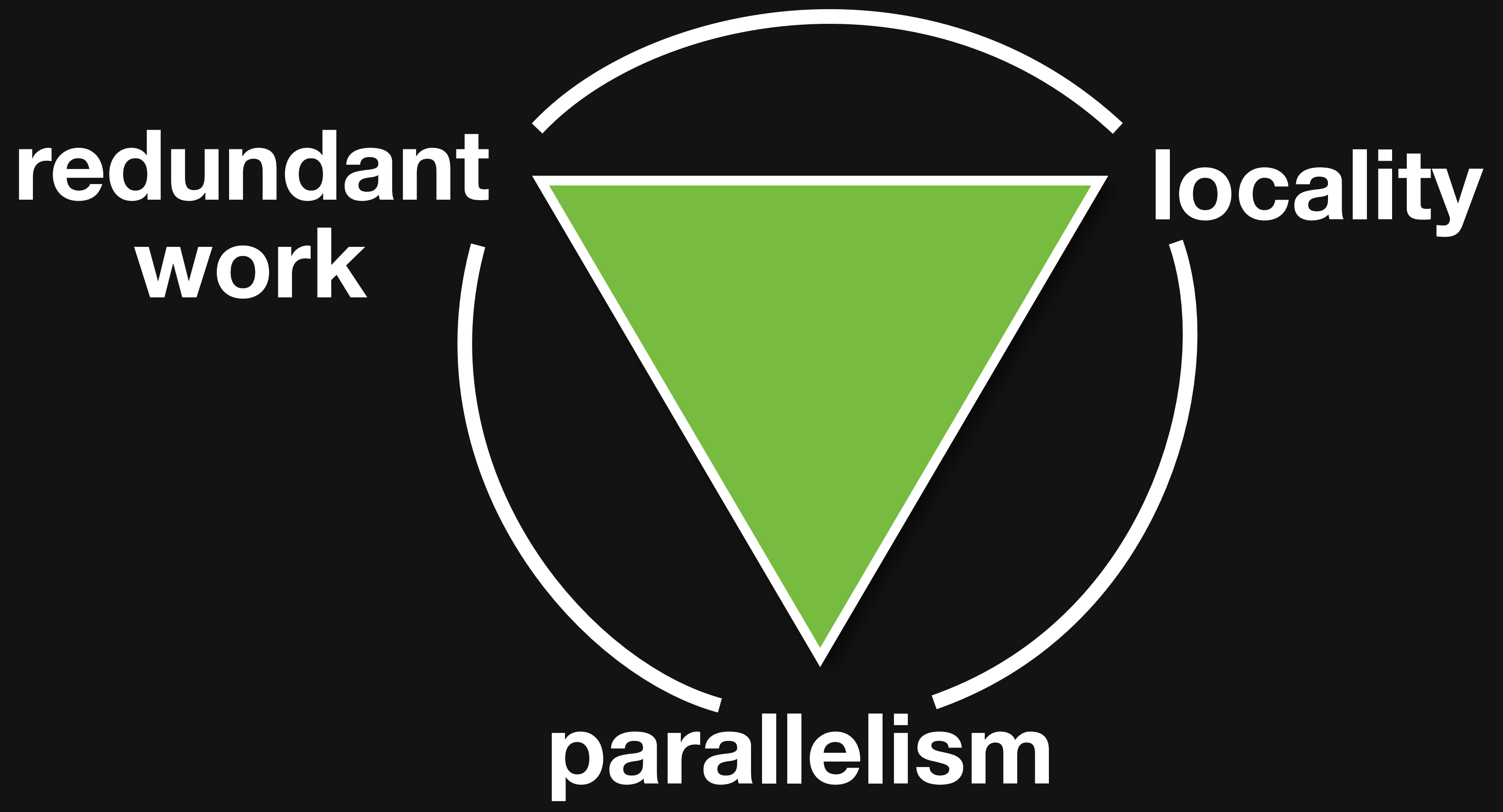
```
void box_filter_3x3(const Image &in, Image &blury) {
  Image blurx(in.width(), in.height());  // allocate blurx array

  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;
}
```

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;
```

Domain scope
of the programming model

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

Only **feed-forward pipelines**

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

Recursion 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.

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.**

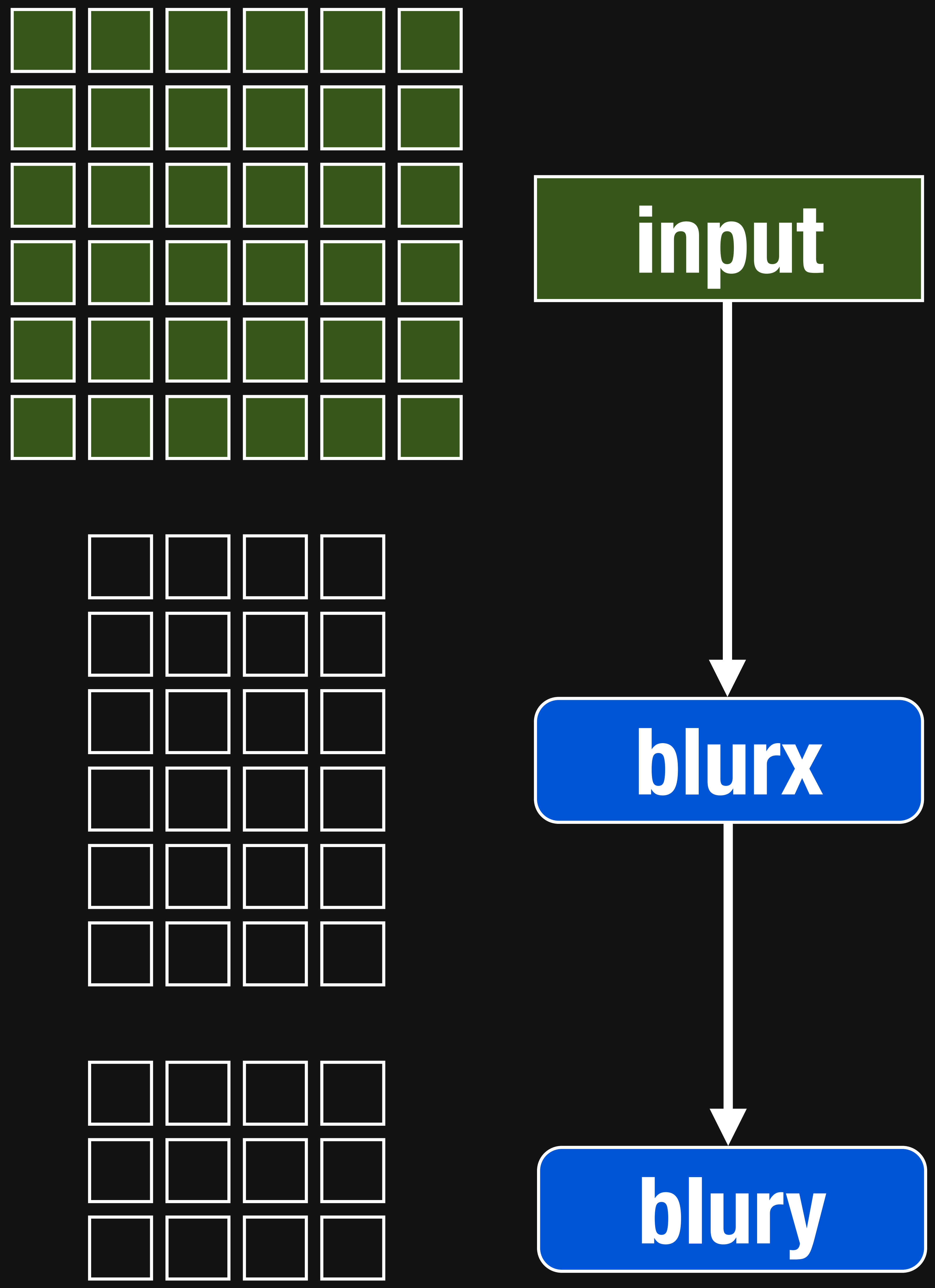
Dependence must be inferable.

User-defined clamping can impose tight bounds, when needed.

Long, heterogeneous pipelines.

Complex graphs, deeper than traditional stencil computations.

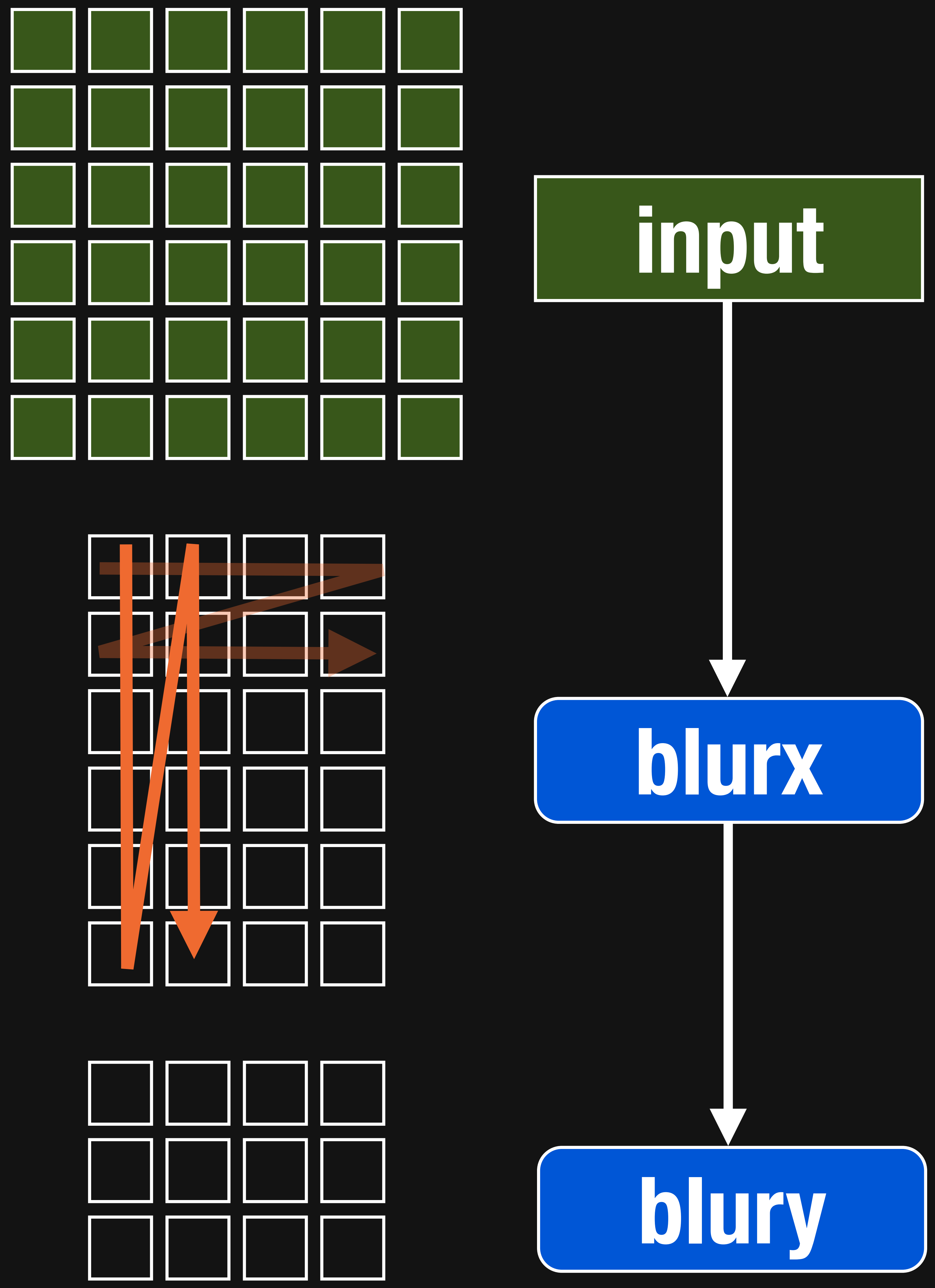
The schedule defines intra-stage order, inter-stage interleaving



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

For each stage:

1) In what order should we compute its values?

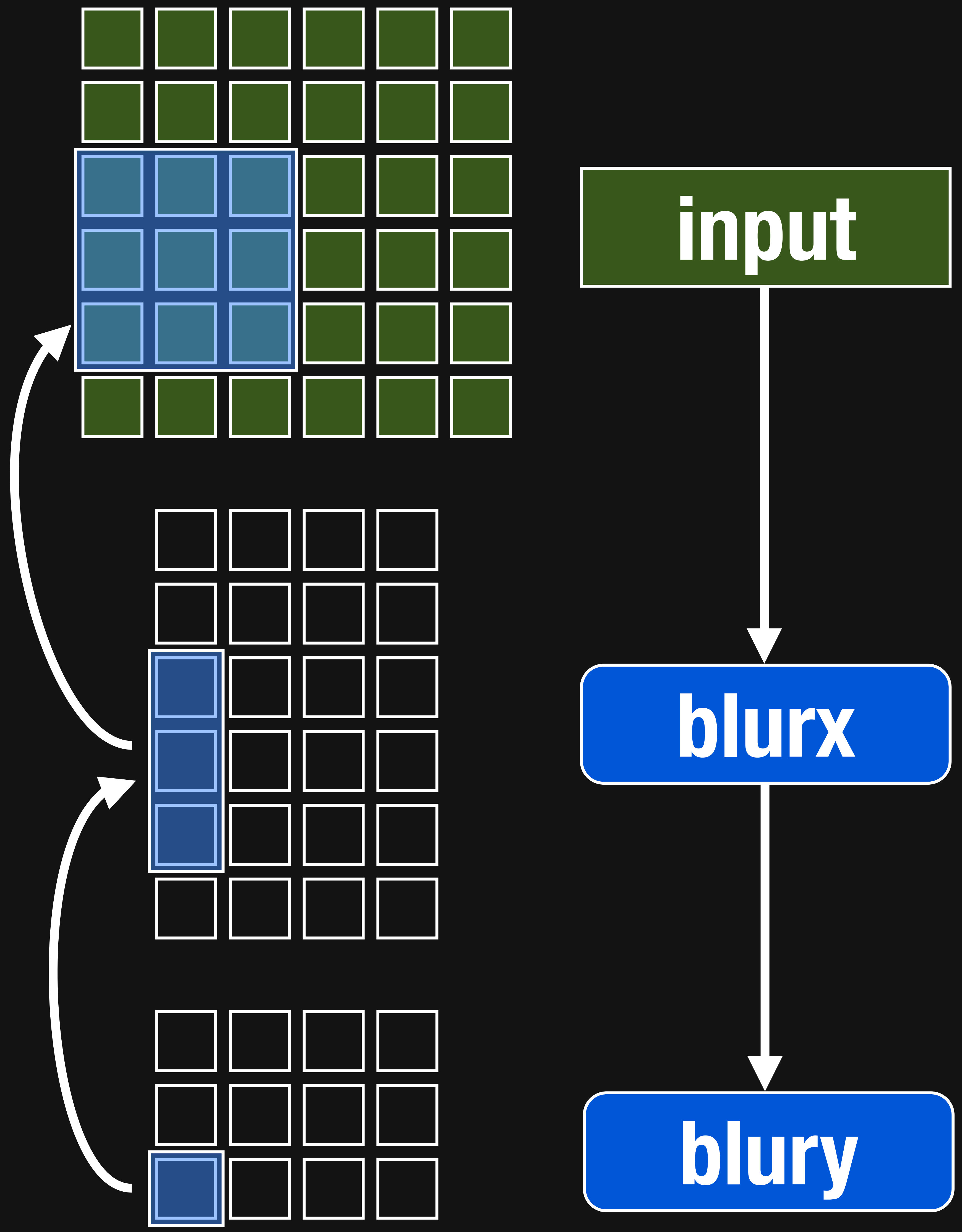


The schedule defines intra-stage order, inter-stage interleaving

For each stage:

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

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



The schedule defines order & parallelism within stages

The schedule defines order & parallelism within stages

Serial y,
Serial x

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

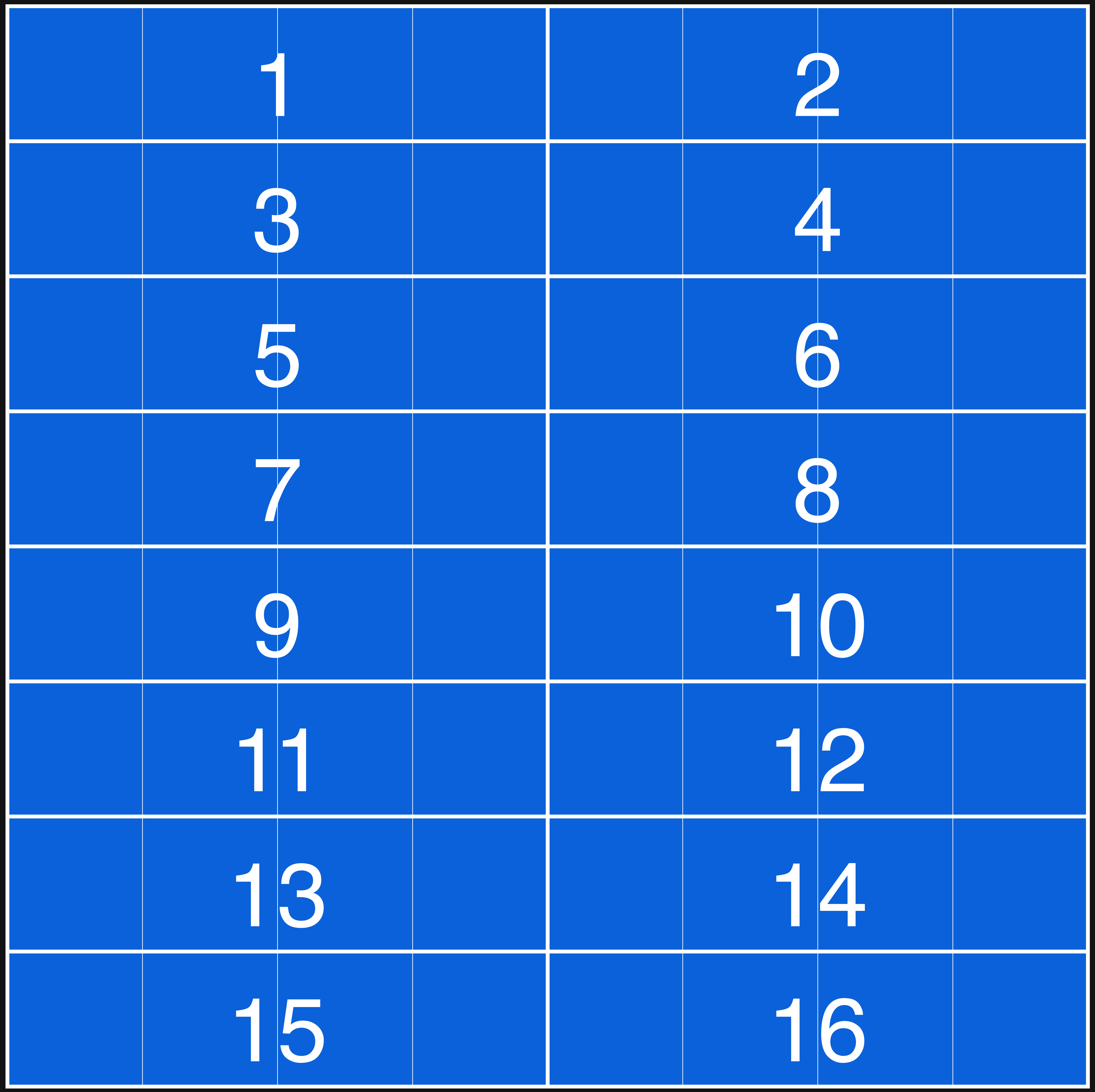
The schedule defines order & parallelism within stages

Serial y,
Serial x

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

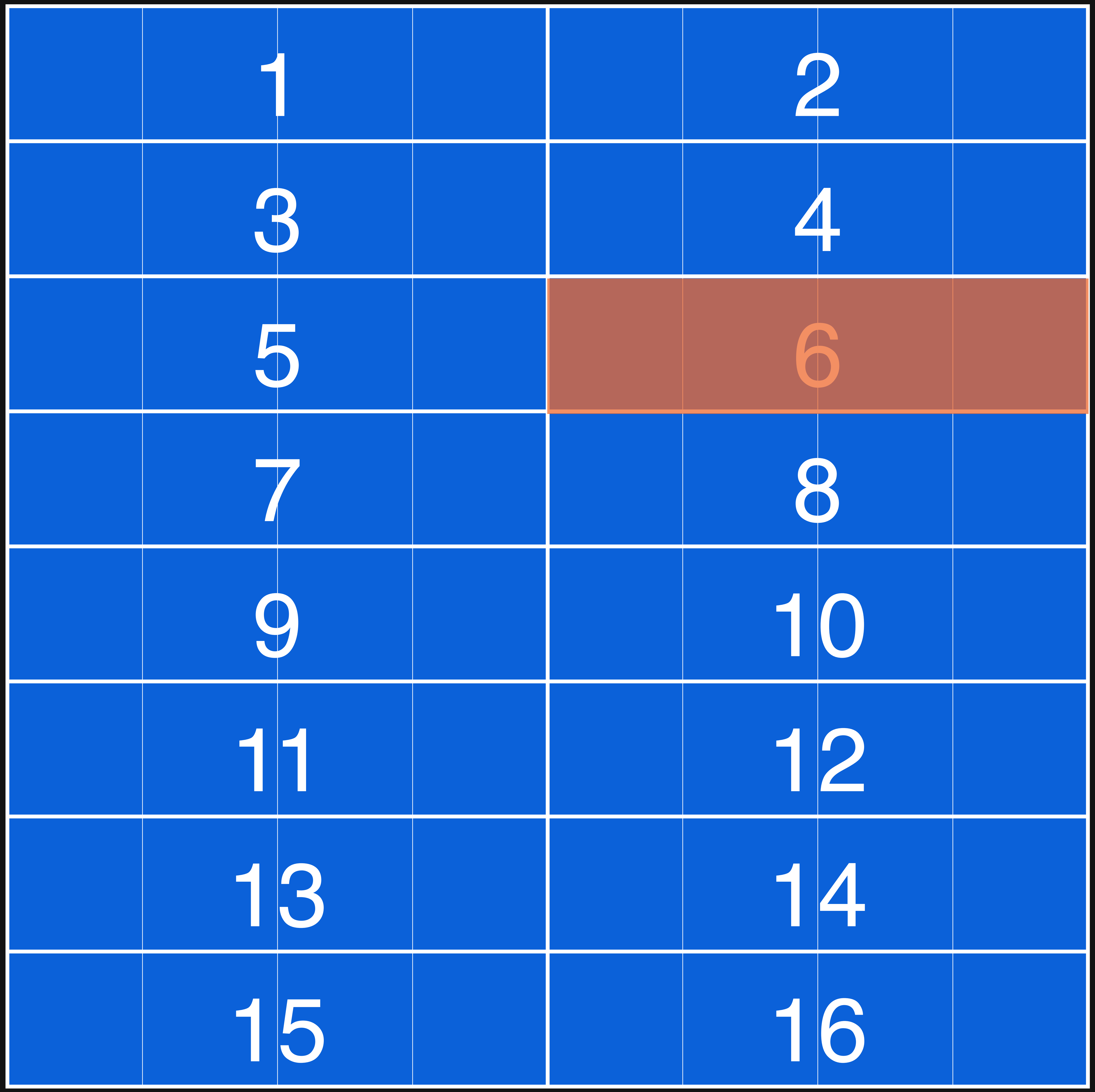
The schedule defines order & parallelism within stages

Serial y,
Vectorize x by 4



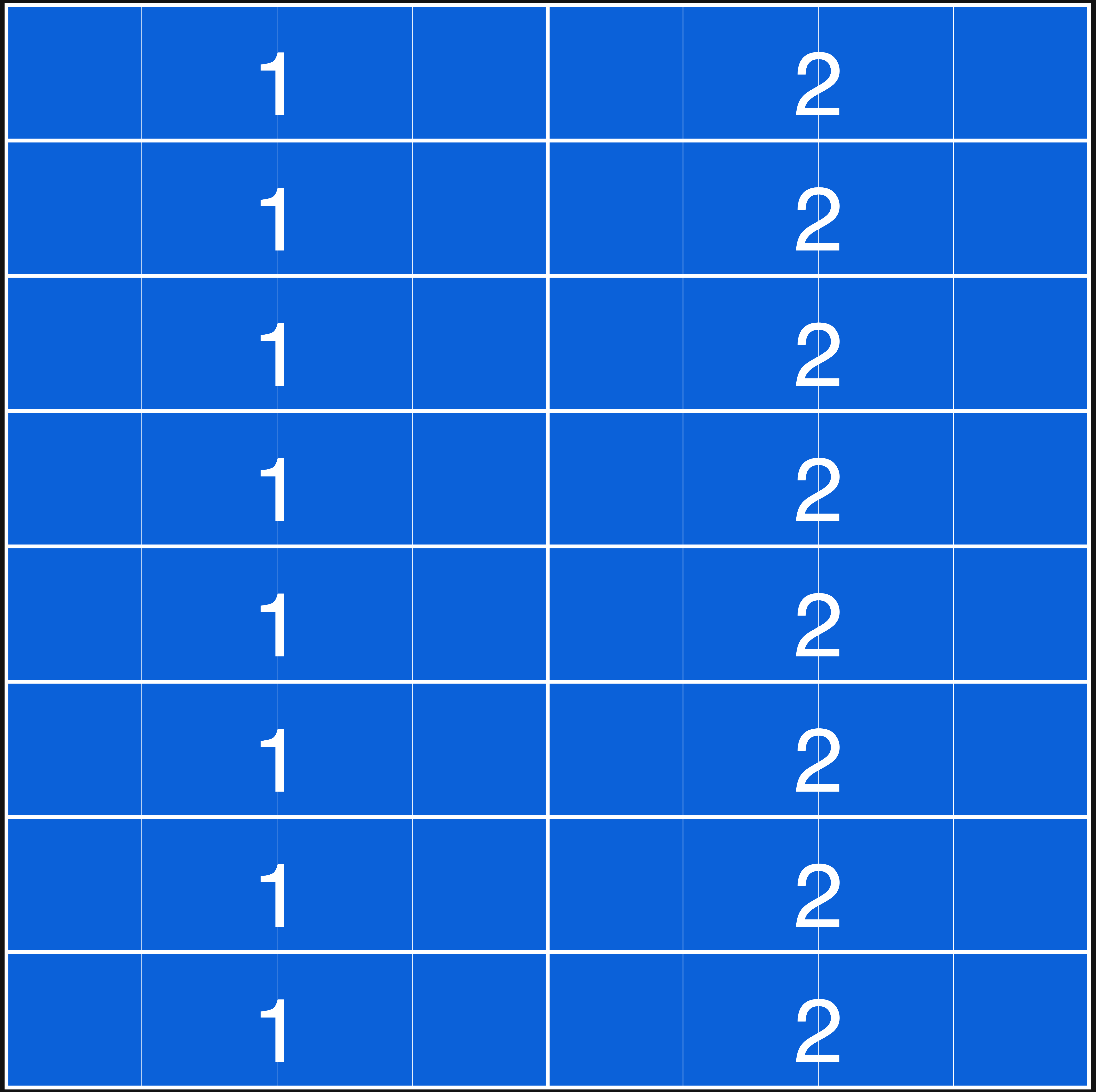
The schedule defines order & parallelism within stages

Serial y,
Vectorize x by 4



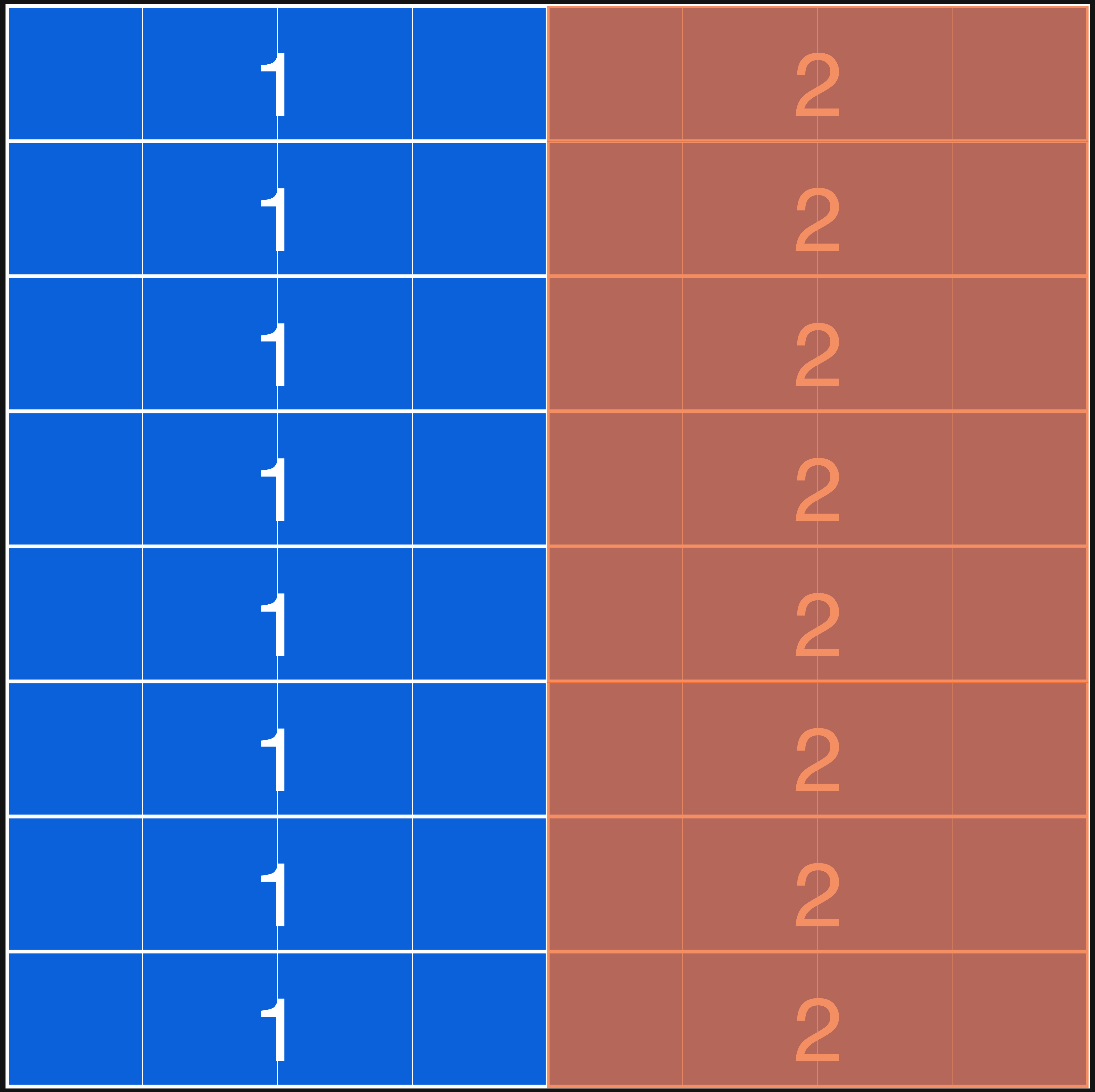
The schedule defines order & parallelism within stages

Parallel y,
Vectorize x by 4



The schedule defines order & parallelism within stages

Parallel y,
Vectorize x by 4



The schedule defines order & parallelism within stages

Split x by 2,
Split y by 2.

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

The schedule defines order & parallelism within stages

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

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

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

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) ]
  }
```

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

Serial y,
Serial x

```
for (y : ymin..ymax)
  for (x : xmin..xmax) {
    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

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

```
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) ]
      }
```

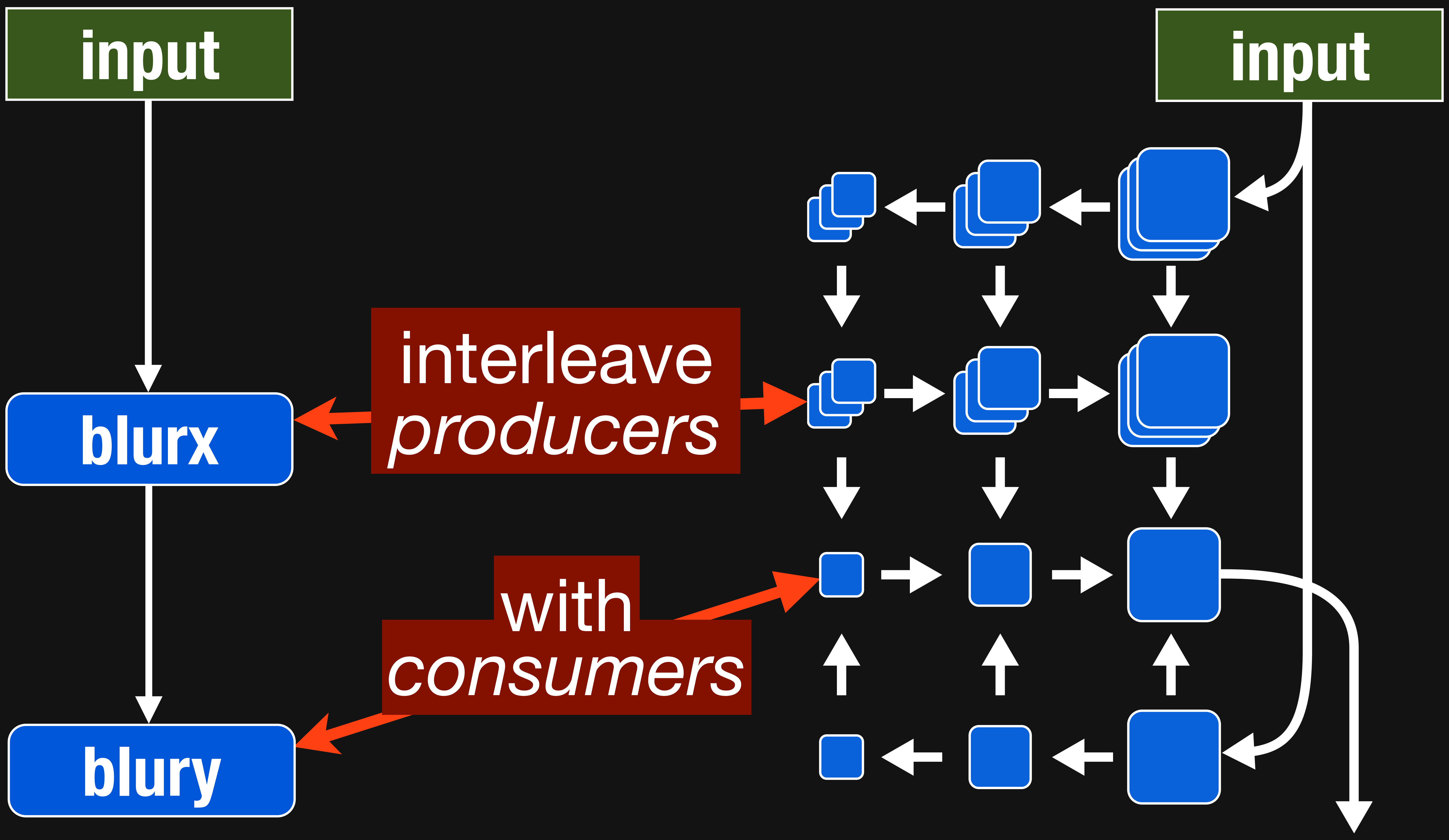
Domain order defines a **loop nest** for each function

Serial y,
Serial x

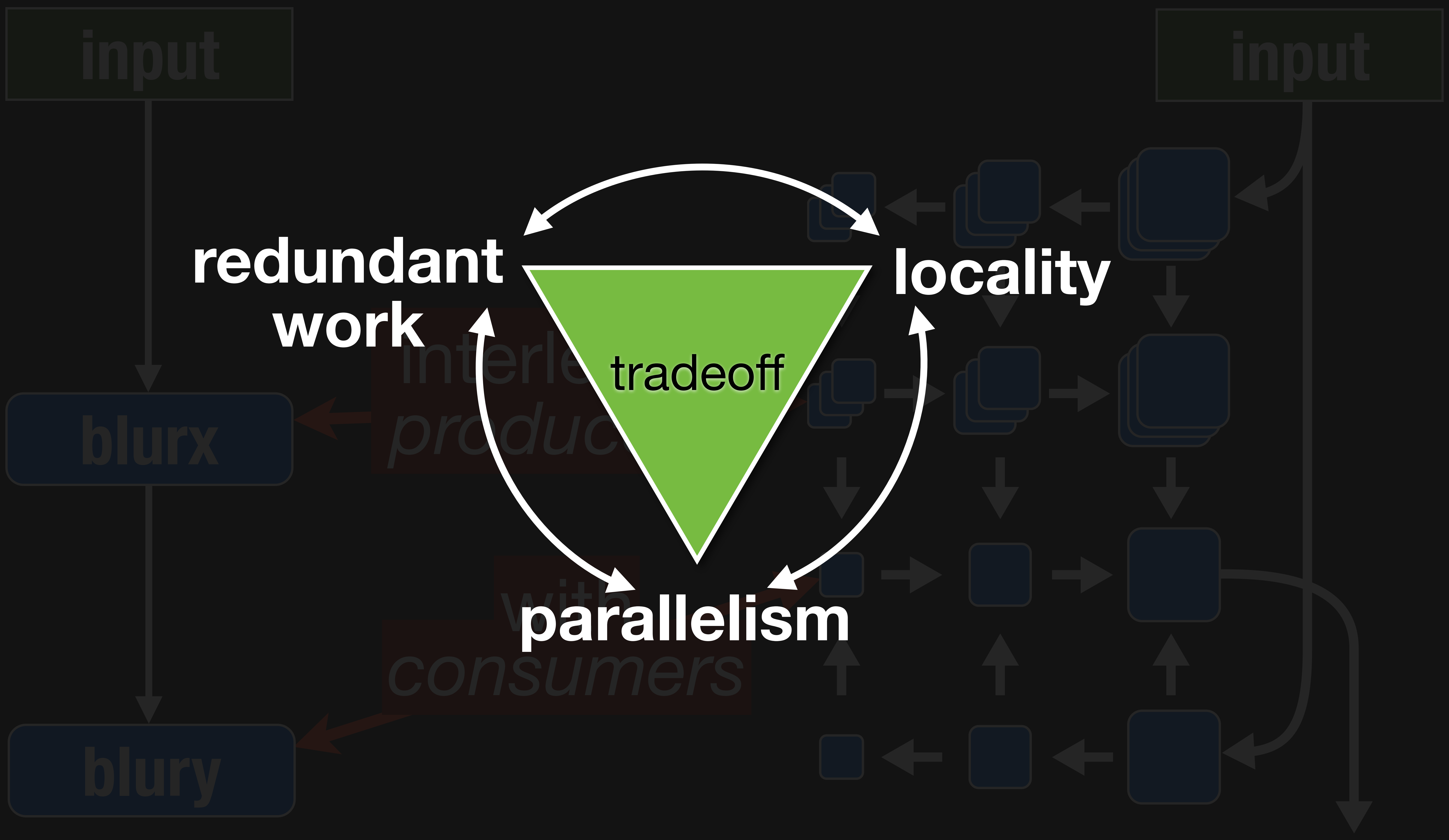
Split x by 4,
Split y by 4.
Parallel y$_o$,
Serial x$_o$,
Serial y$_i$,
Vectorize x$_i$ by 4

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

The schedule defines producer-consumer interleaving

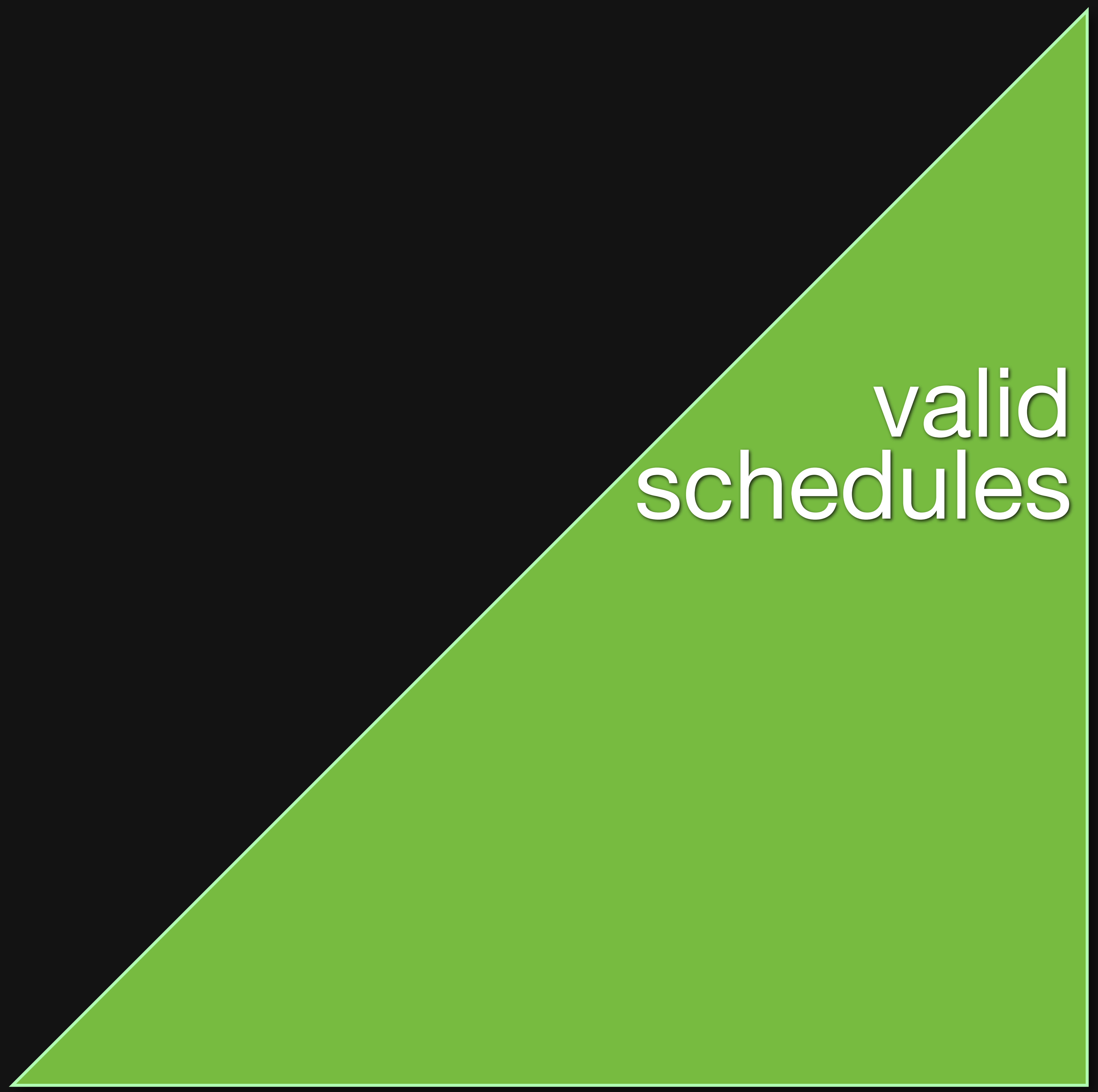


The schedule defines producer-consumer interleaving

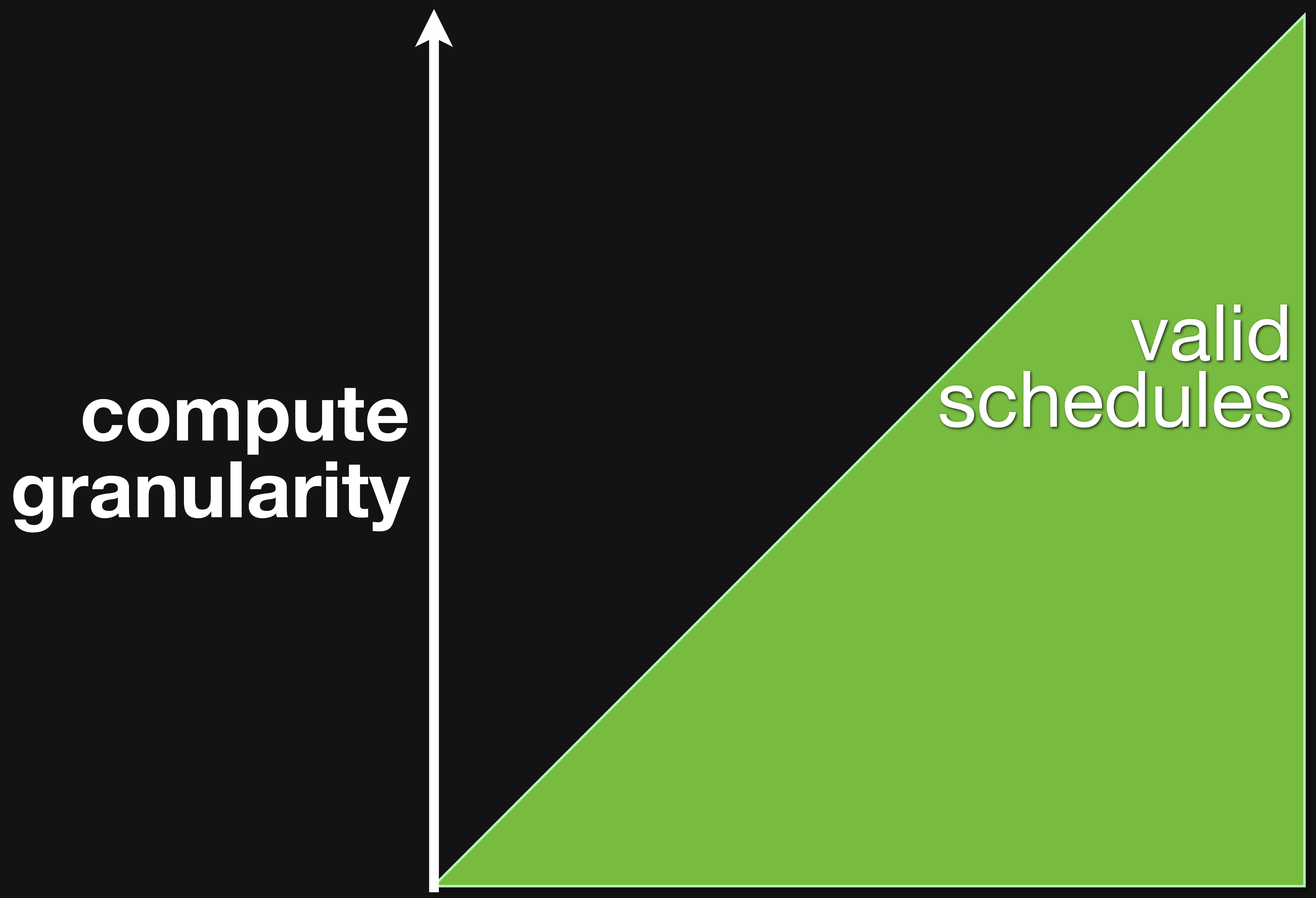


Tradeoff space modeled by **granularity of interleaving**

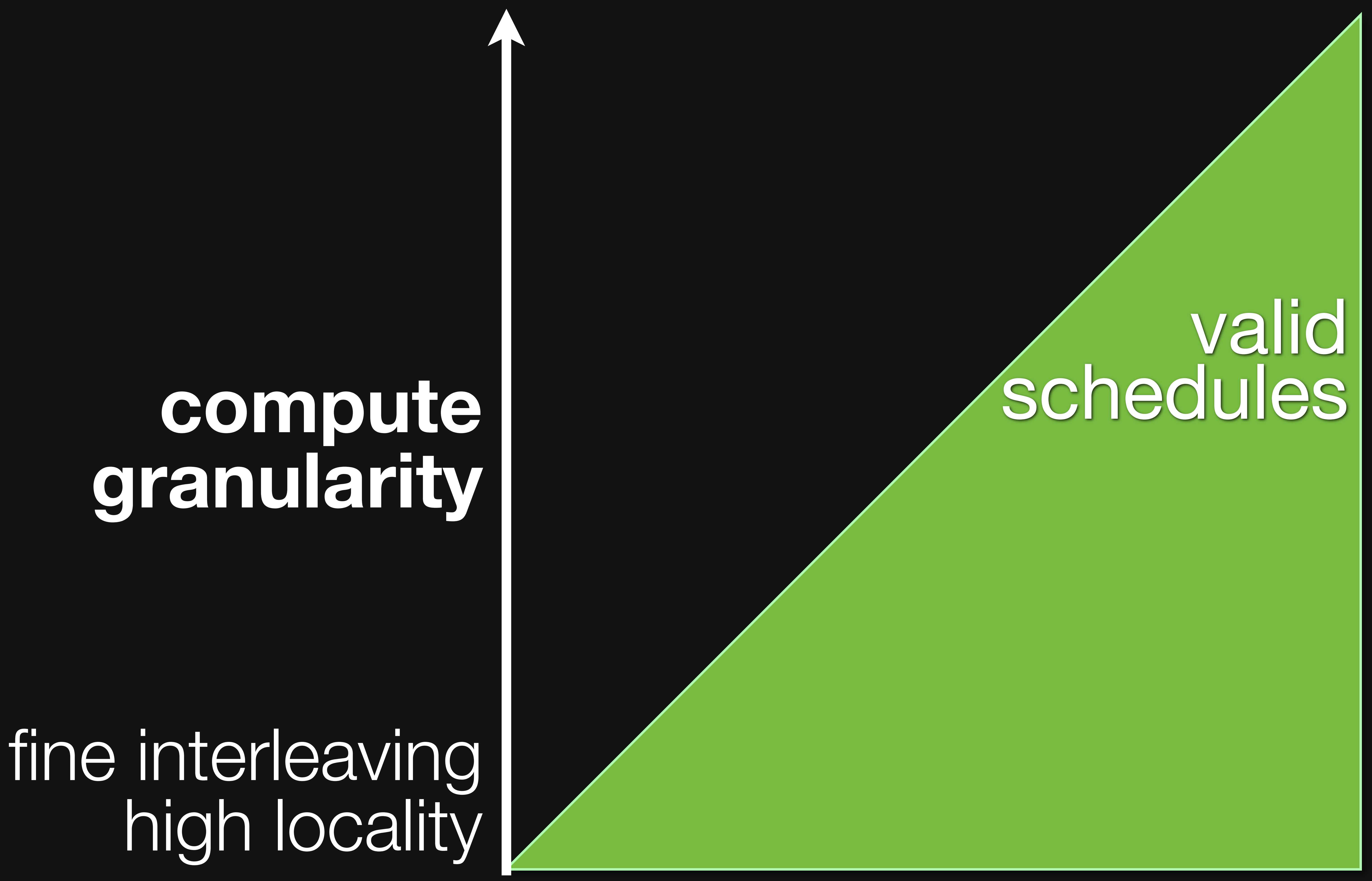
Tradeoff space modeled by **granularity of interleaving**



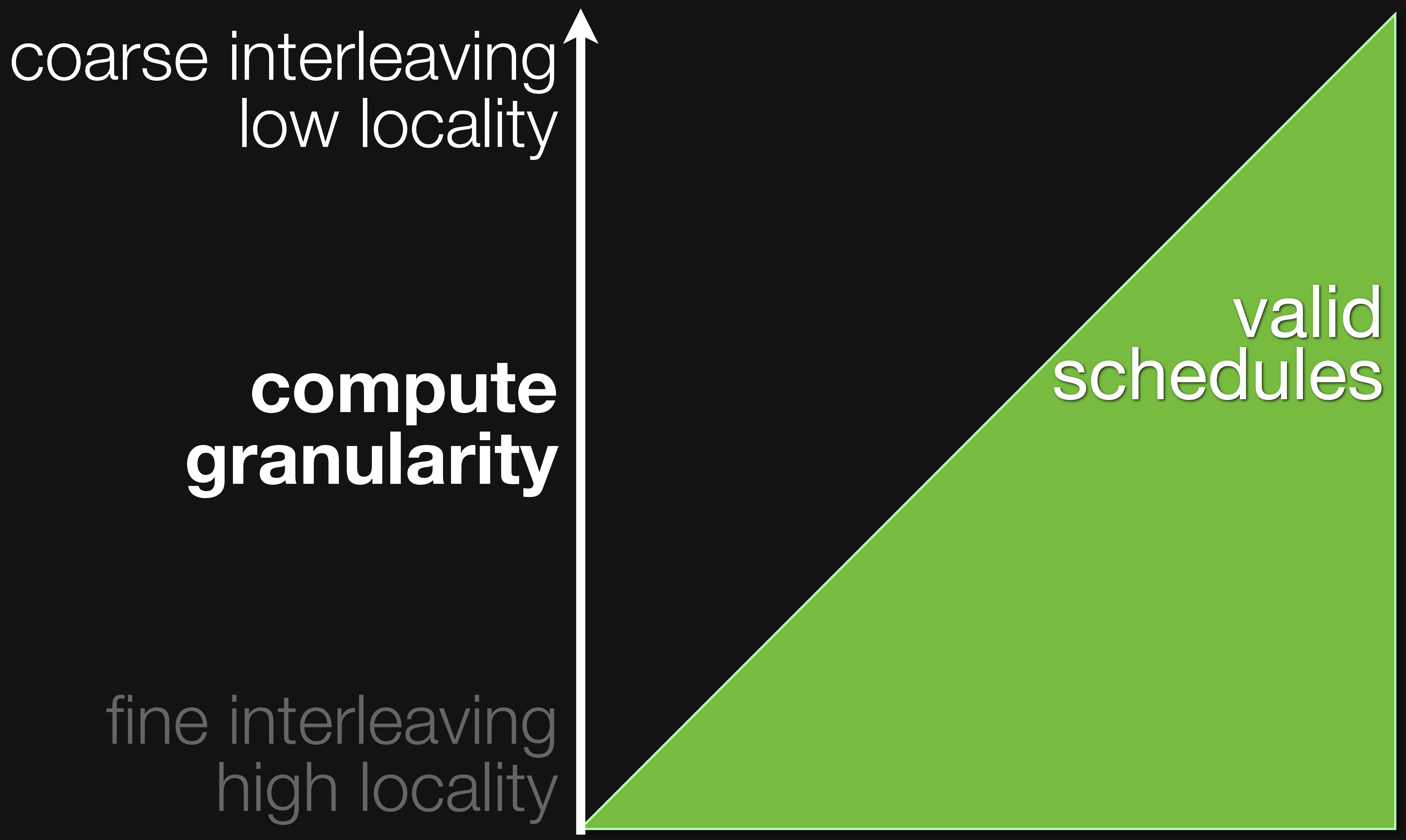
Tradeoff space modeled by **granularity of interleaving**



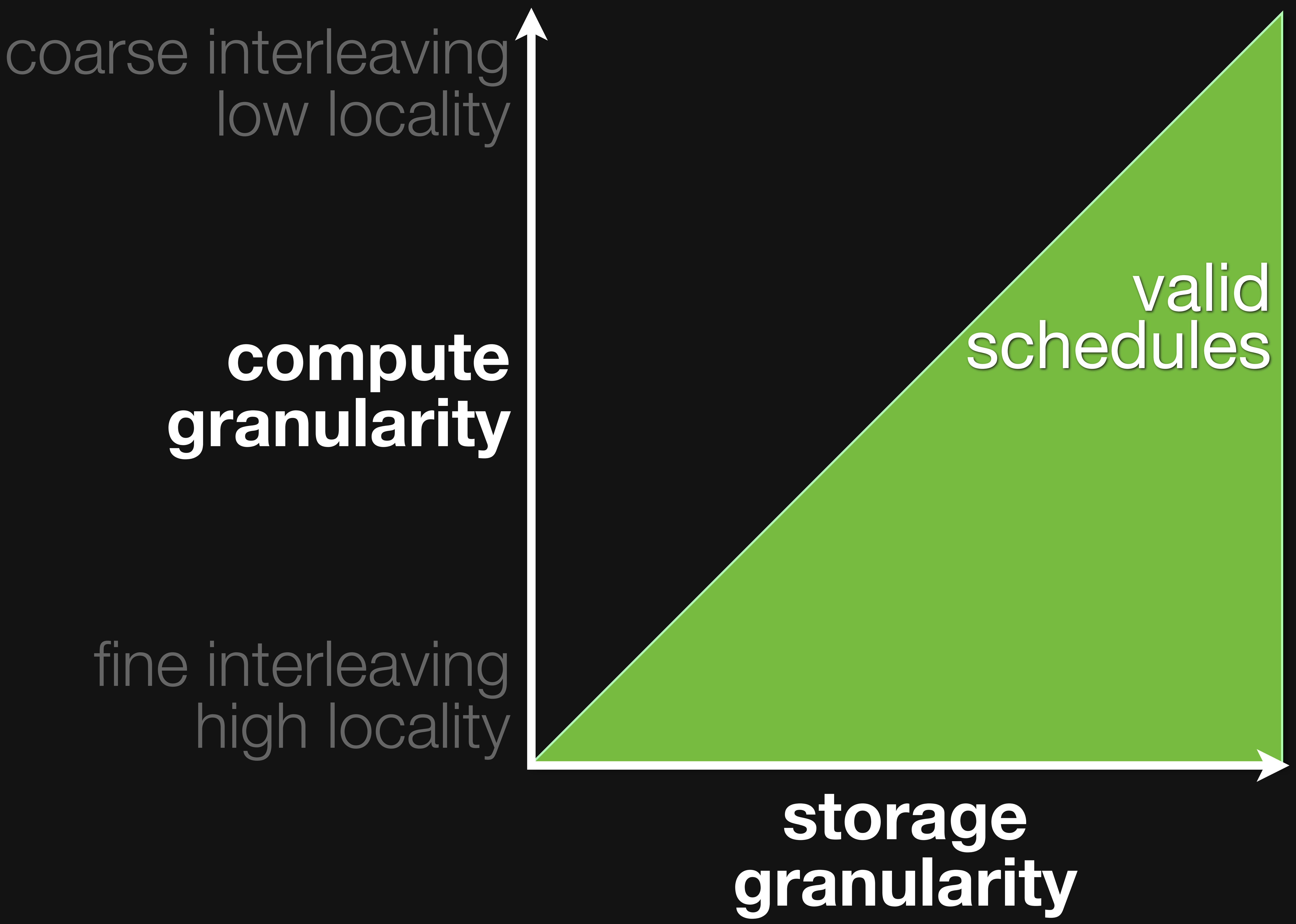
Tradeoff space modeled by **granularity of interleaving**



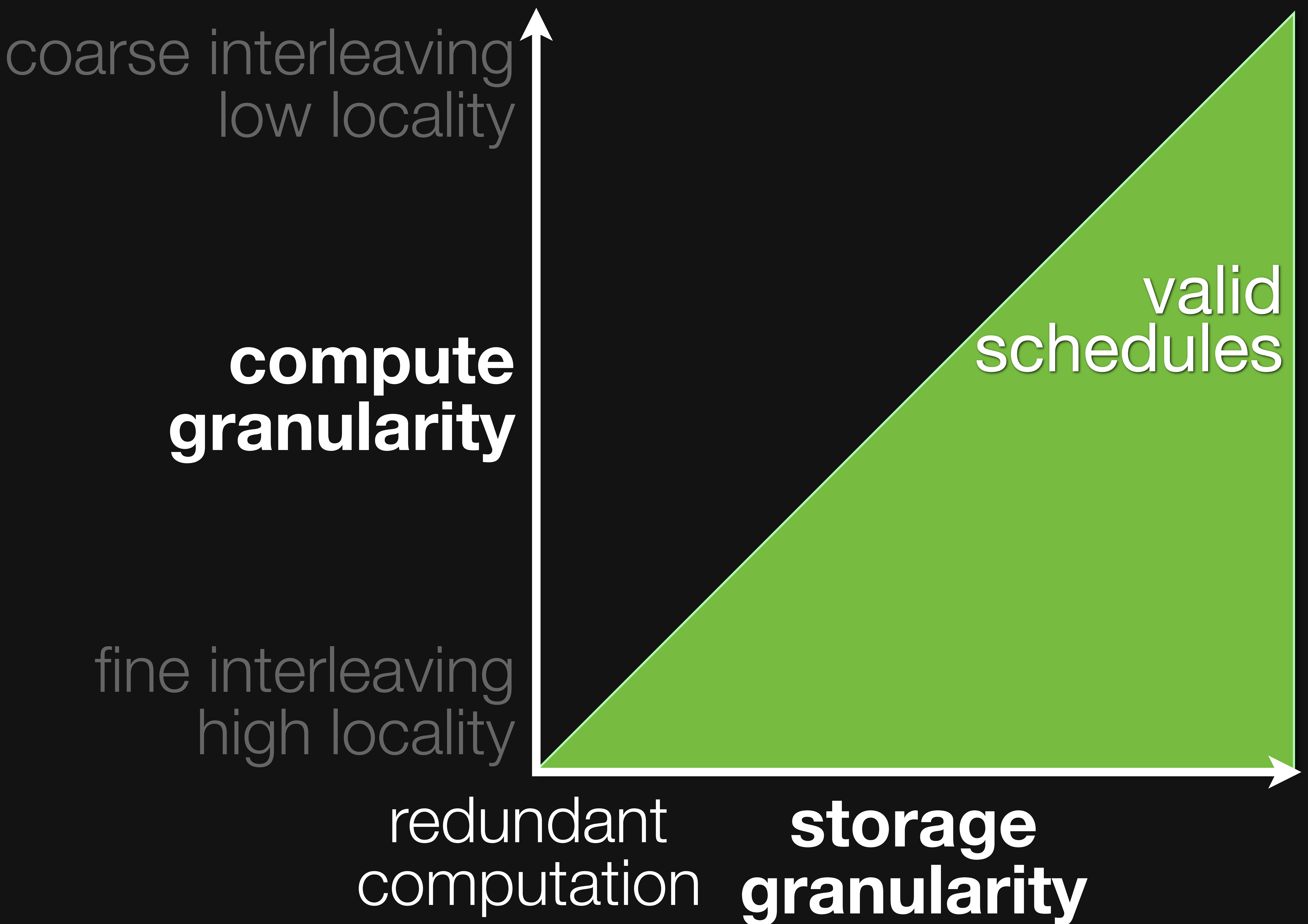
Tradeoff space modeled by **granularity of interleaving**



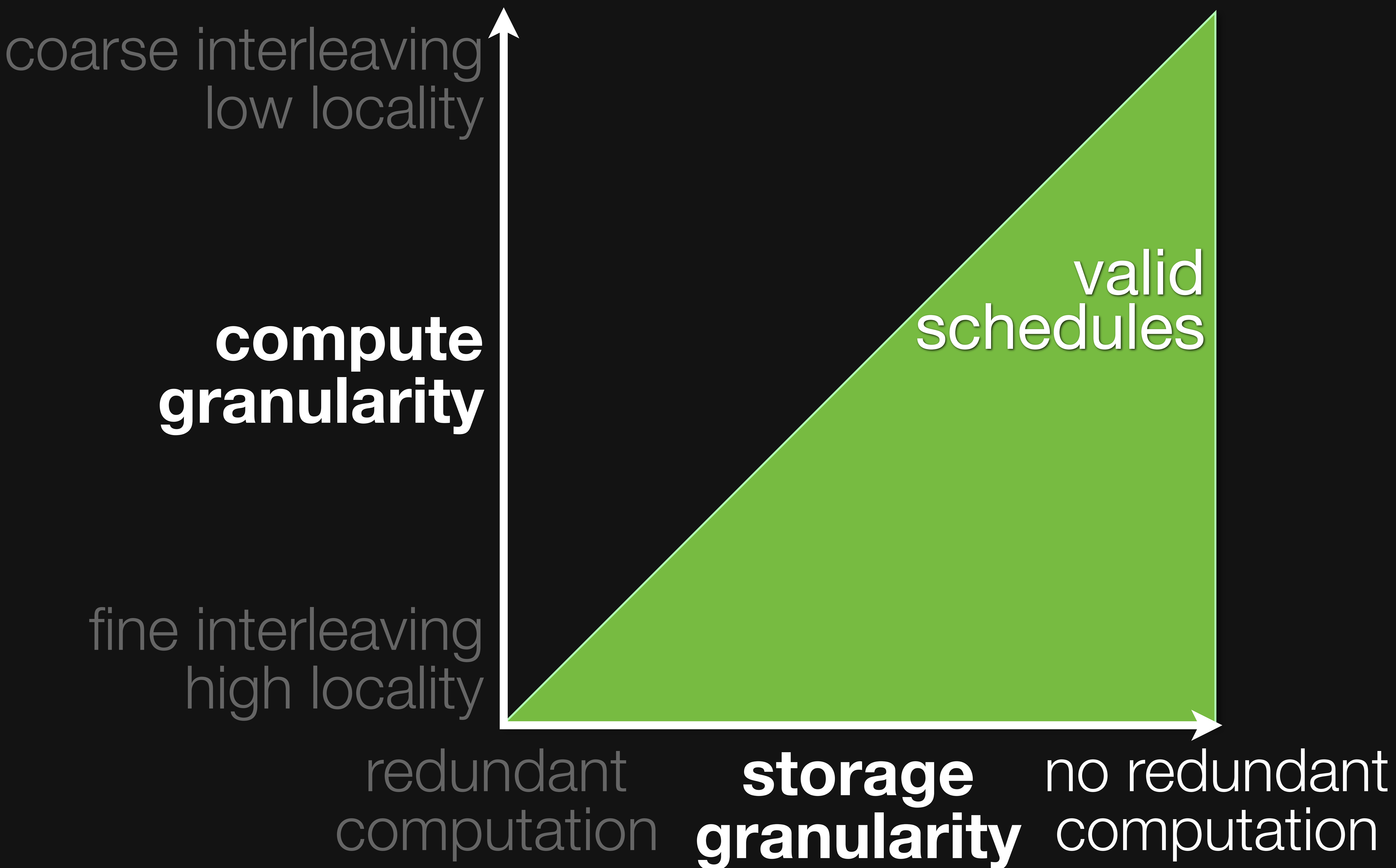
Tradeoff space modeled by **granularity of interleaving**



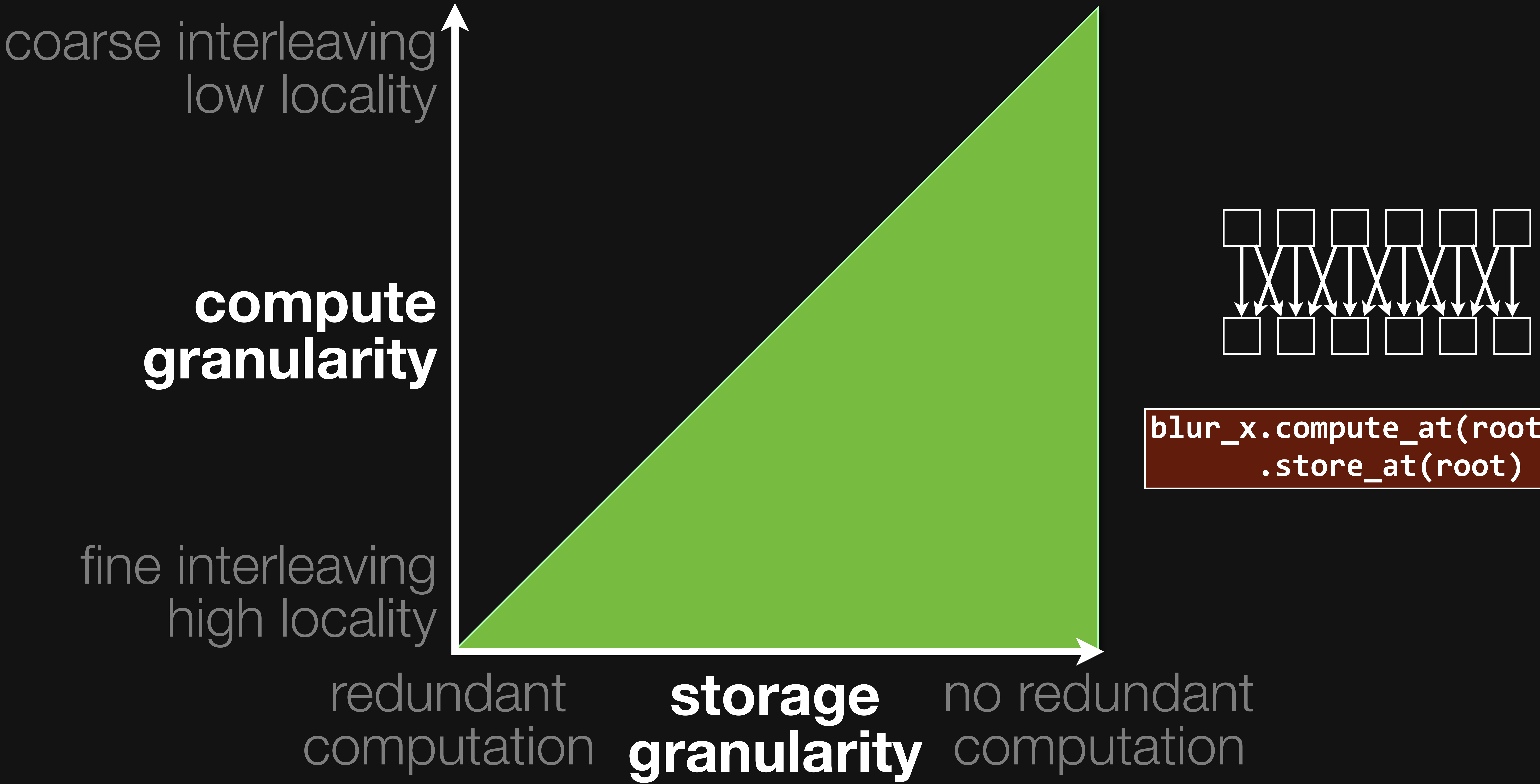
Tradeoff space modeled by **granularity of interleaving**



Tradeoff space modeled by **granularity of interleaving**

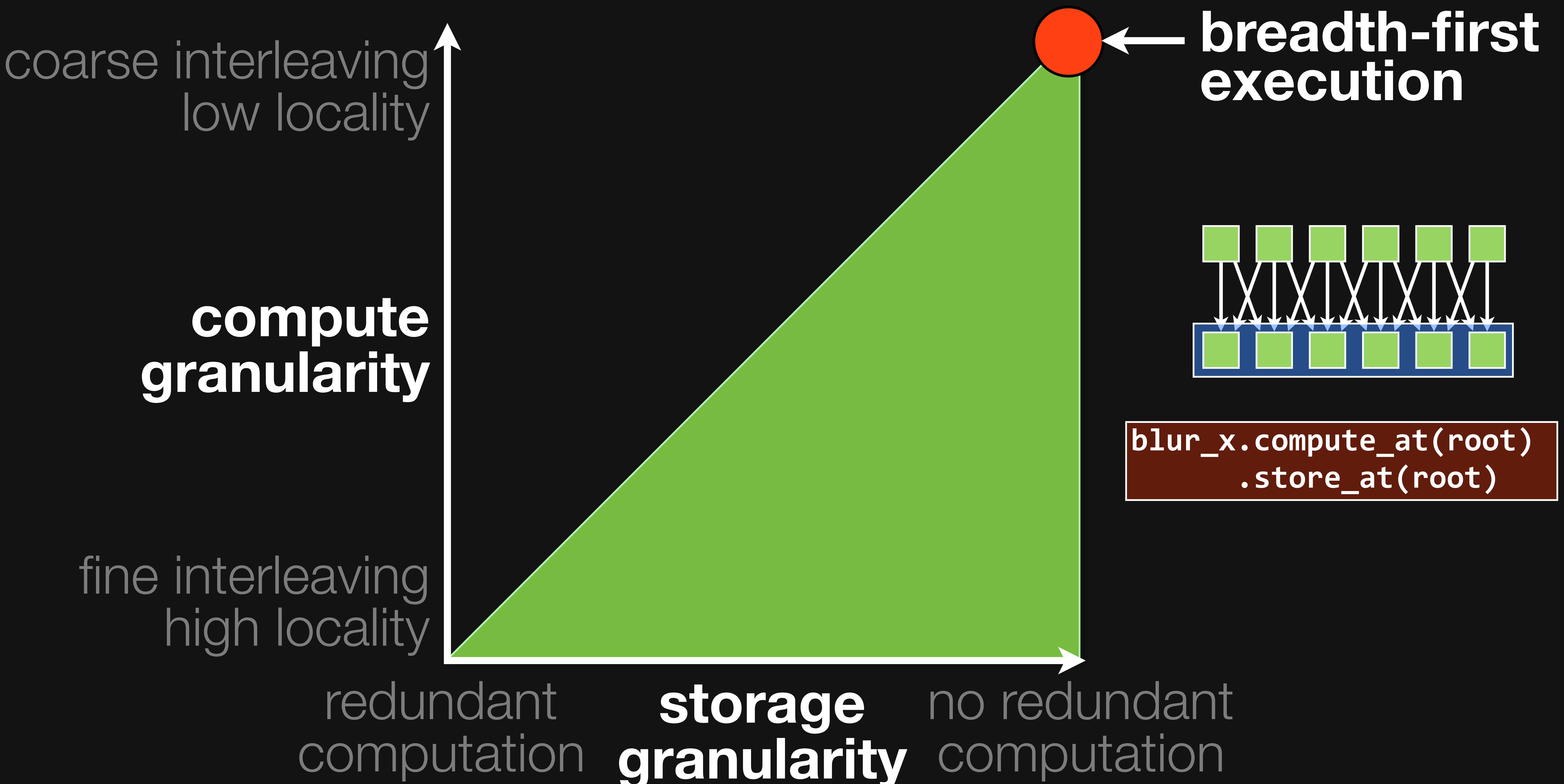


Tradeoff space modeled by **granularity of interleaving**

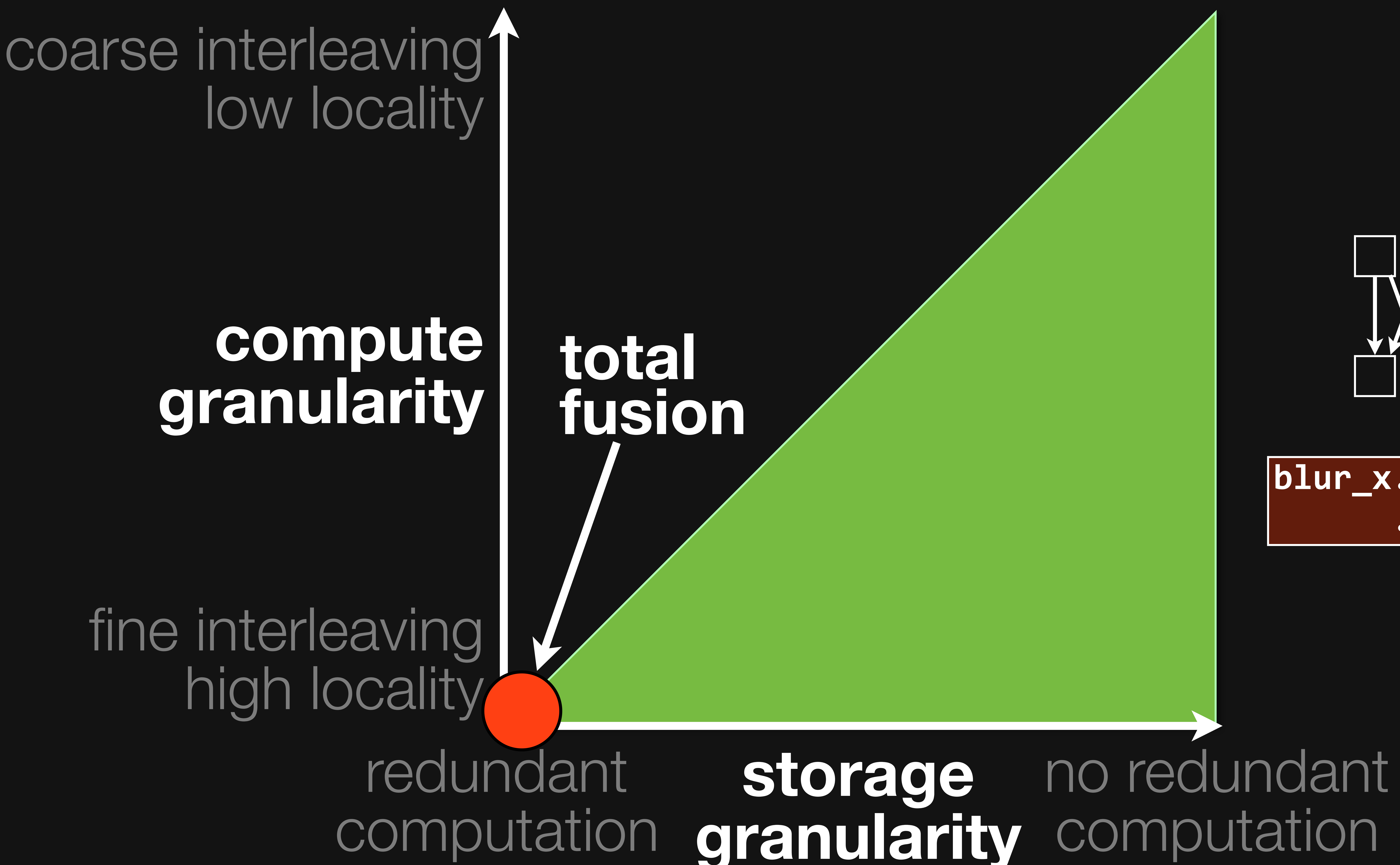


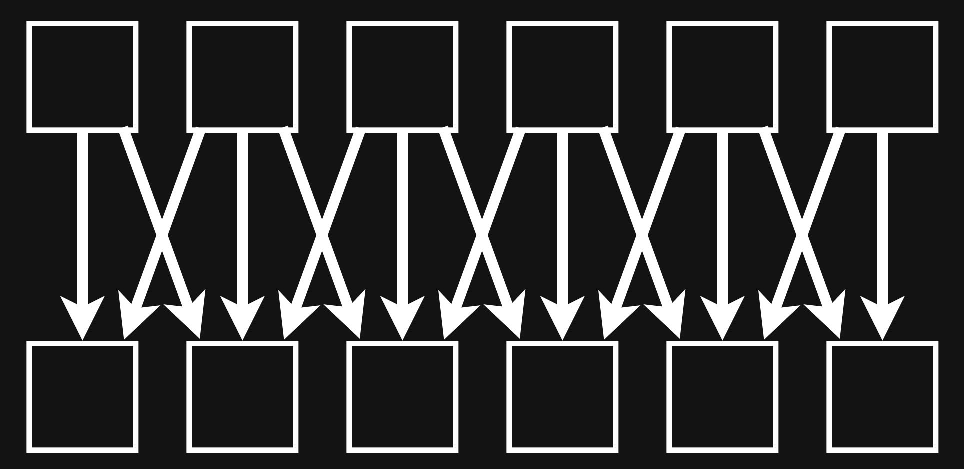

```
blur_x.compute_at(root)
      .store_at(root)
```

Tradeoff space modeled by **granularity of interleaving**



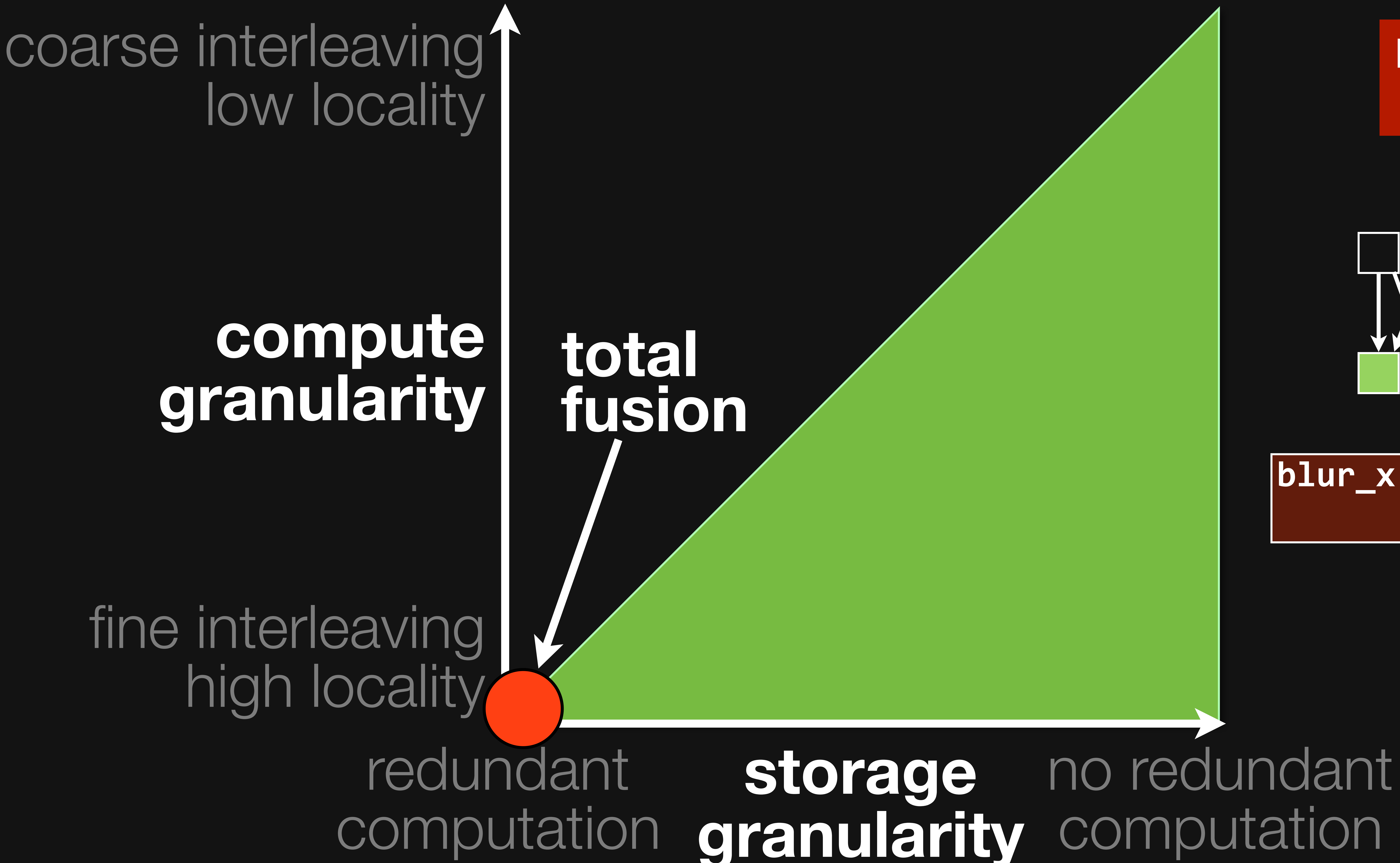
Tradeoff space modeled by **granularity of interleaving**



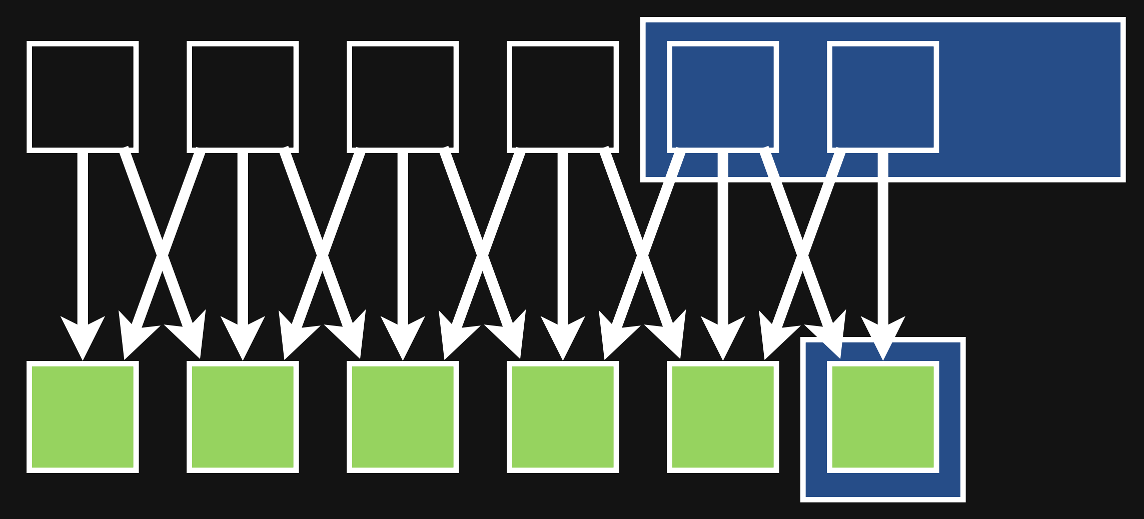


```
blur_x.compute_at(blury, x)
      .store_at(blury, x)
```

Tradeoff space modeled by **granularity of interleaving**

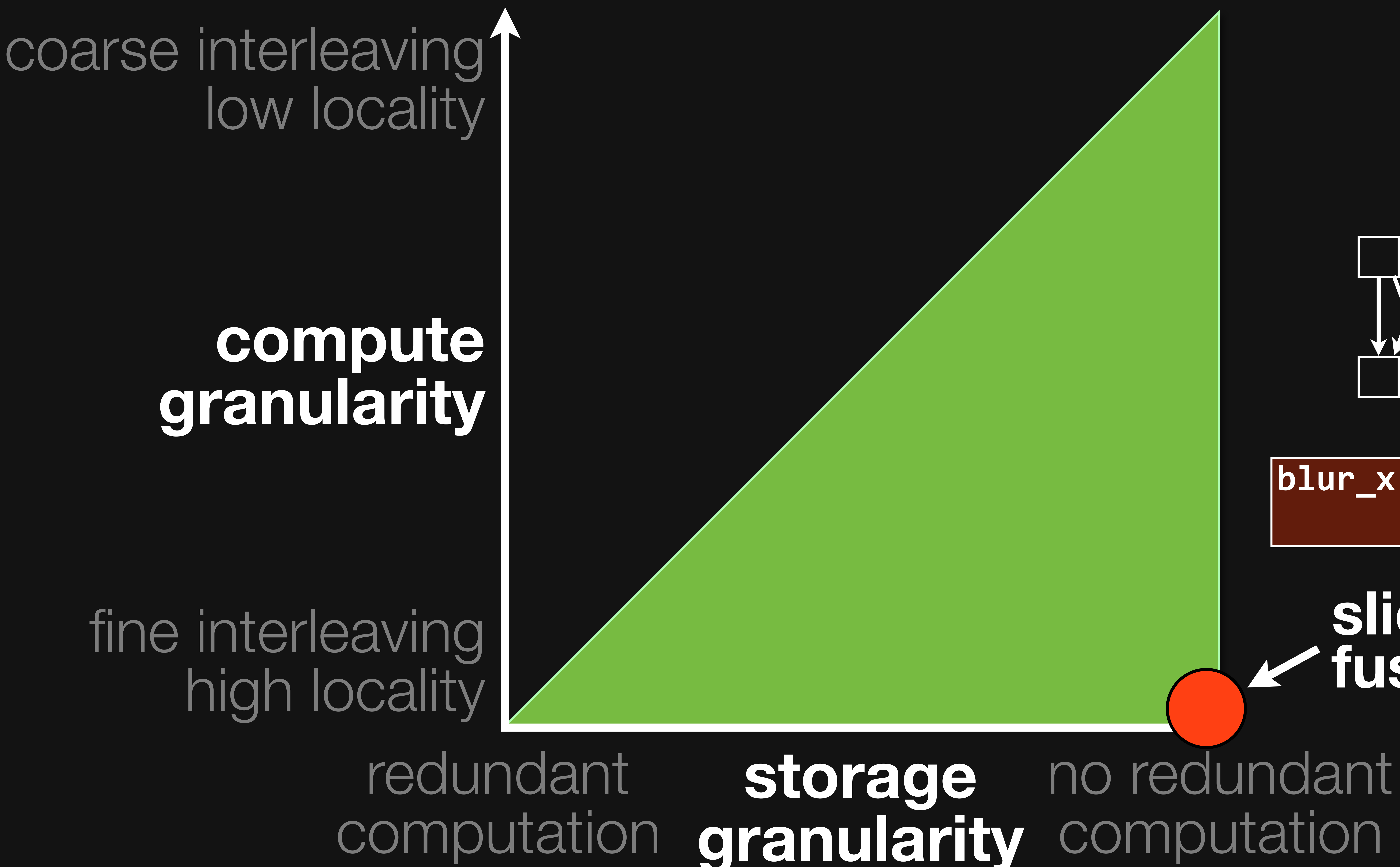


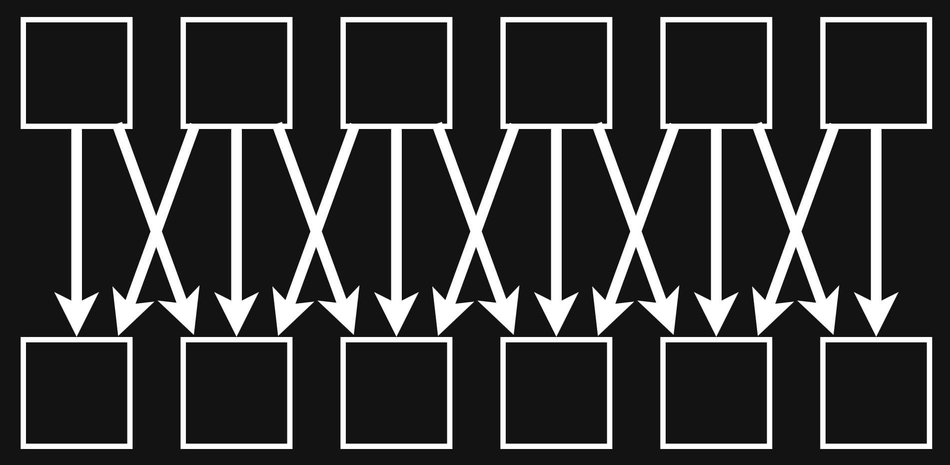
redundant work



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

Tradeoff space modeled by **granularity of interleaving**

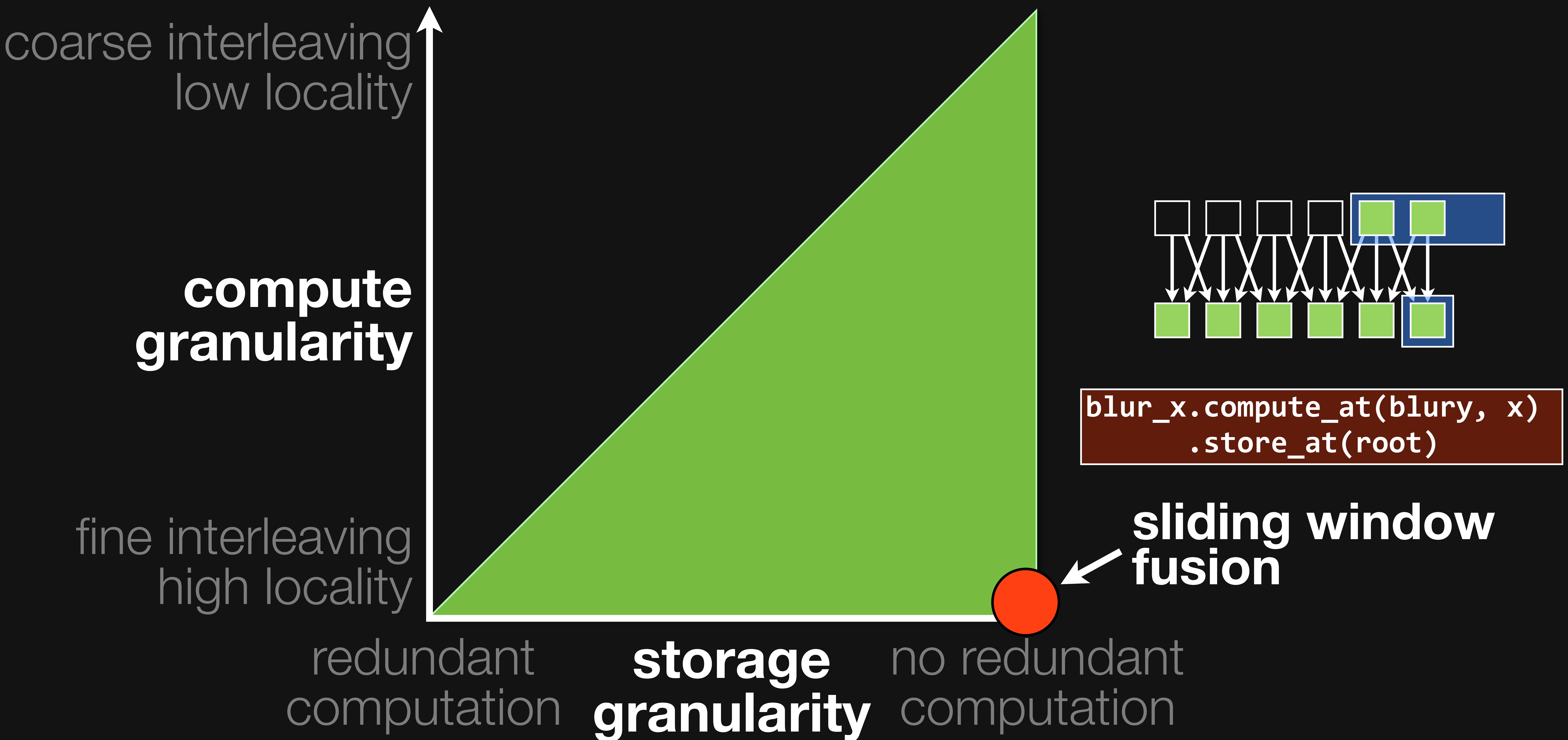




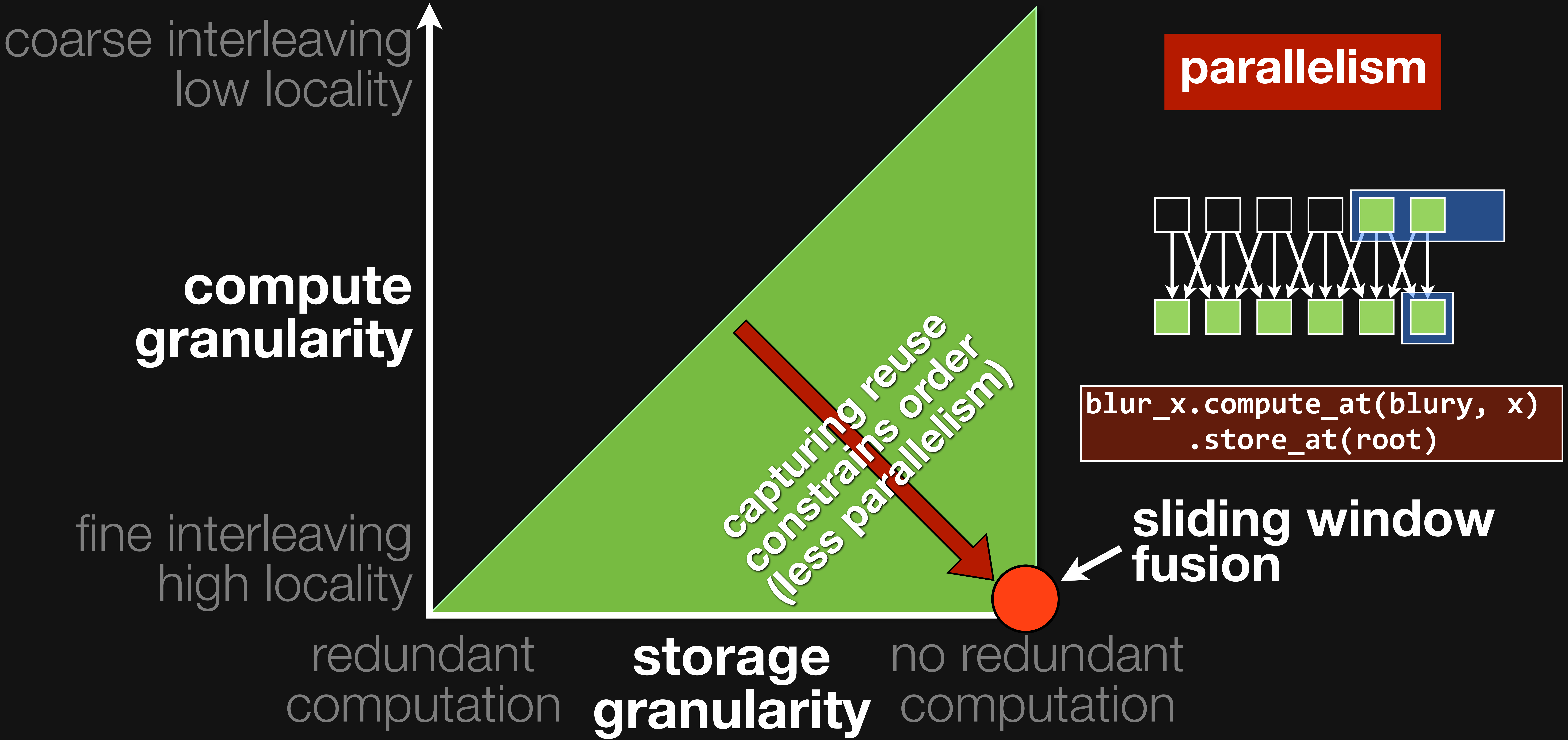
```
blur_x.compute_at(blury, x)
      .store_at(root)
```

sliding window fusion

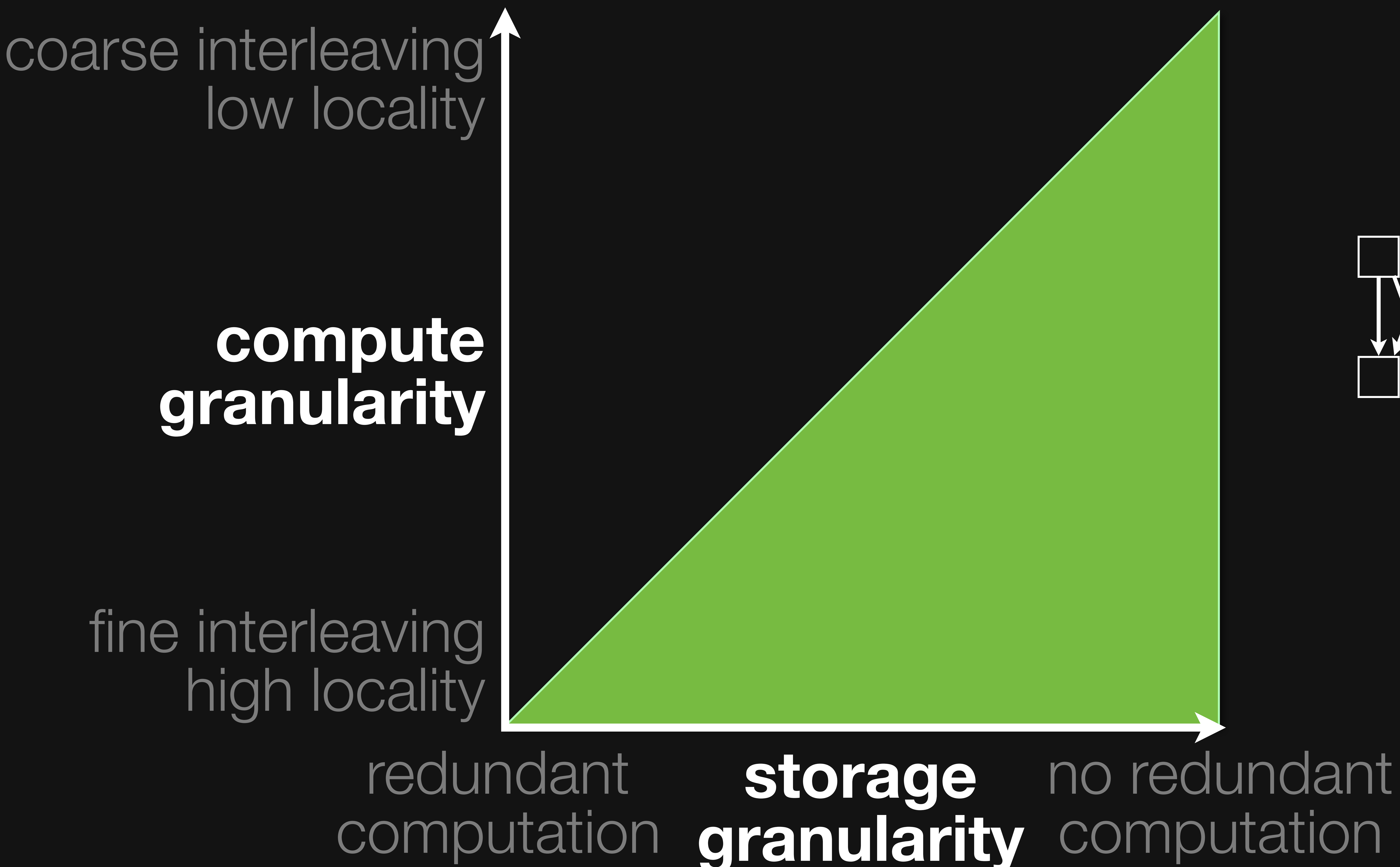
Tradeoff space modeled by **granularity of interleaving**

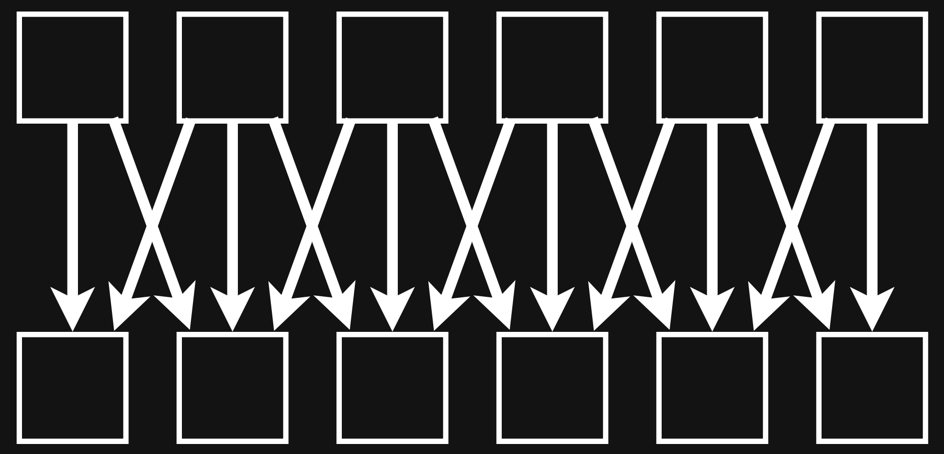


Tradeoff space modeled by granularity of interleaving

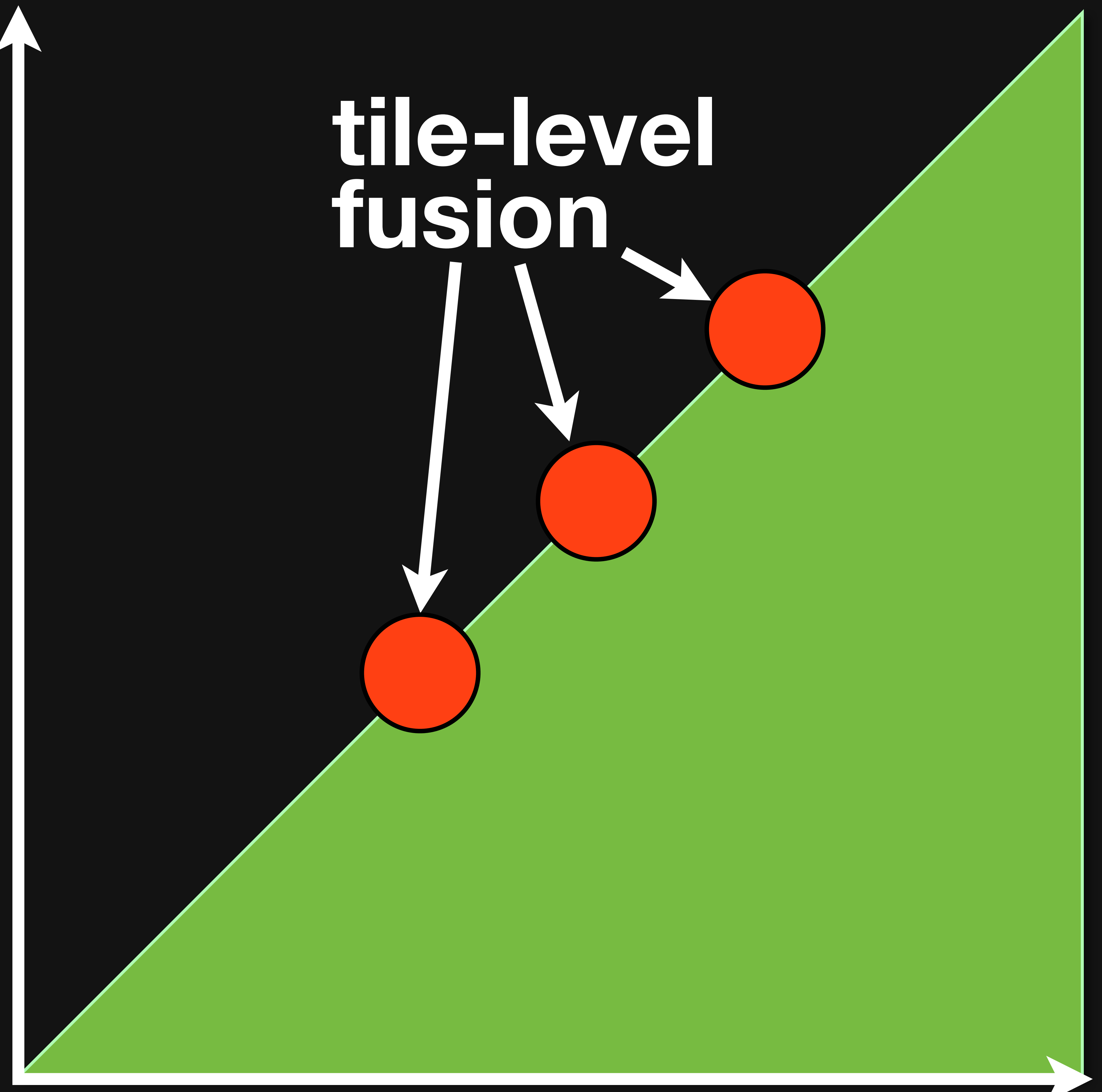


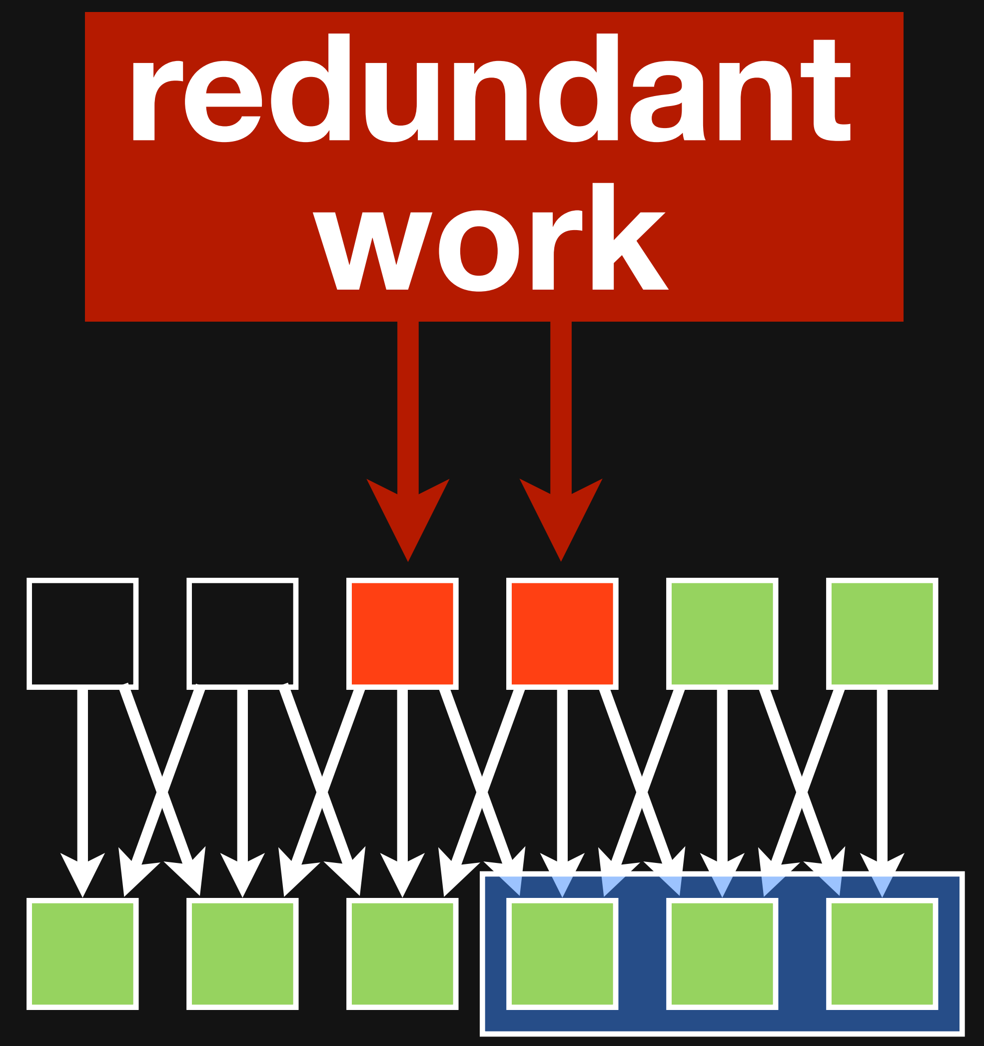
Tradeoff space modeled by **granularity of interleaving**





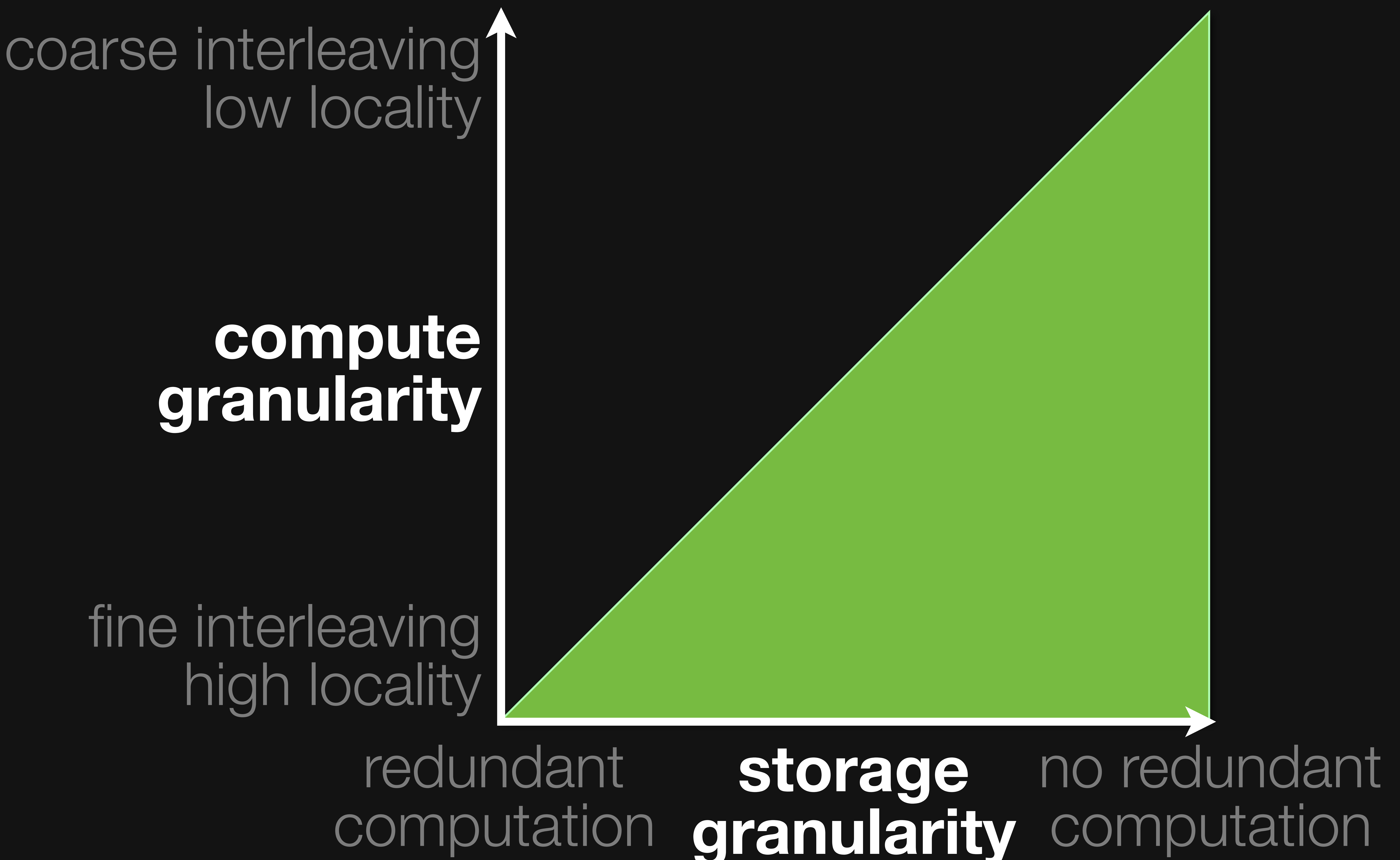
Tradeoff space modeled by **granularity of interleaving**



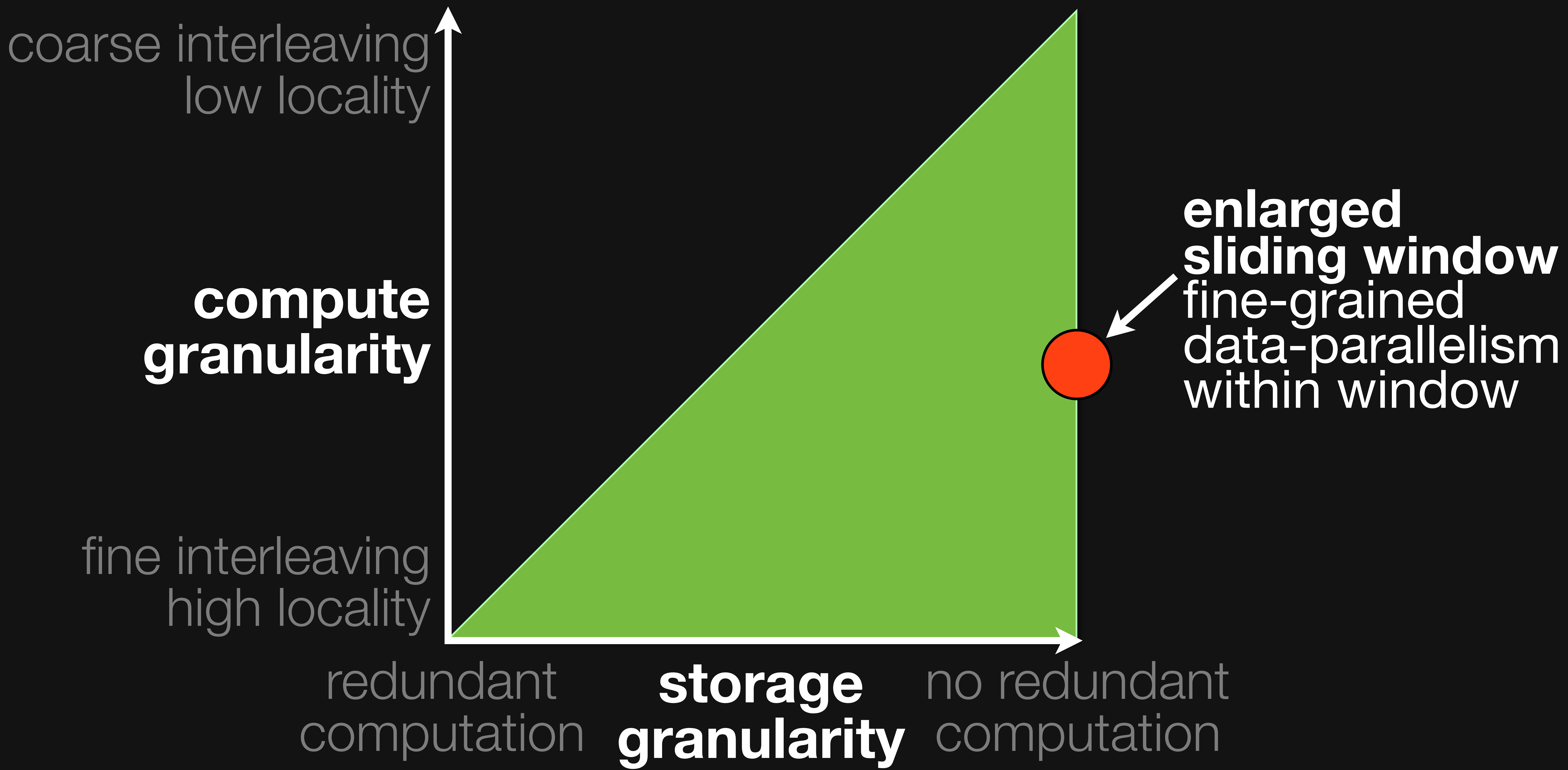



```
blur_y.tile(xo, yo,
            xi, yi,
            W,  H)
blur_x.compute_at(blury, xo)
      .store_at(blury, xo)
```

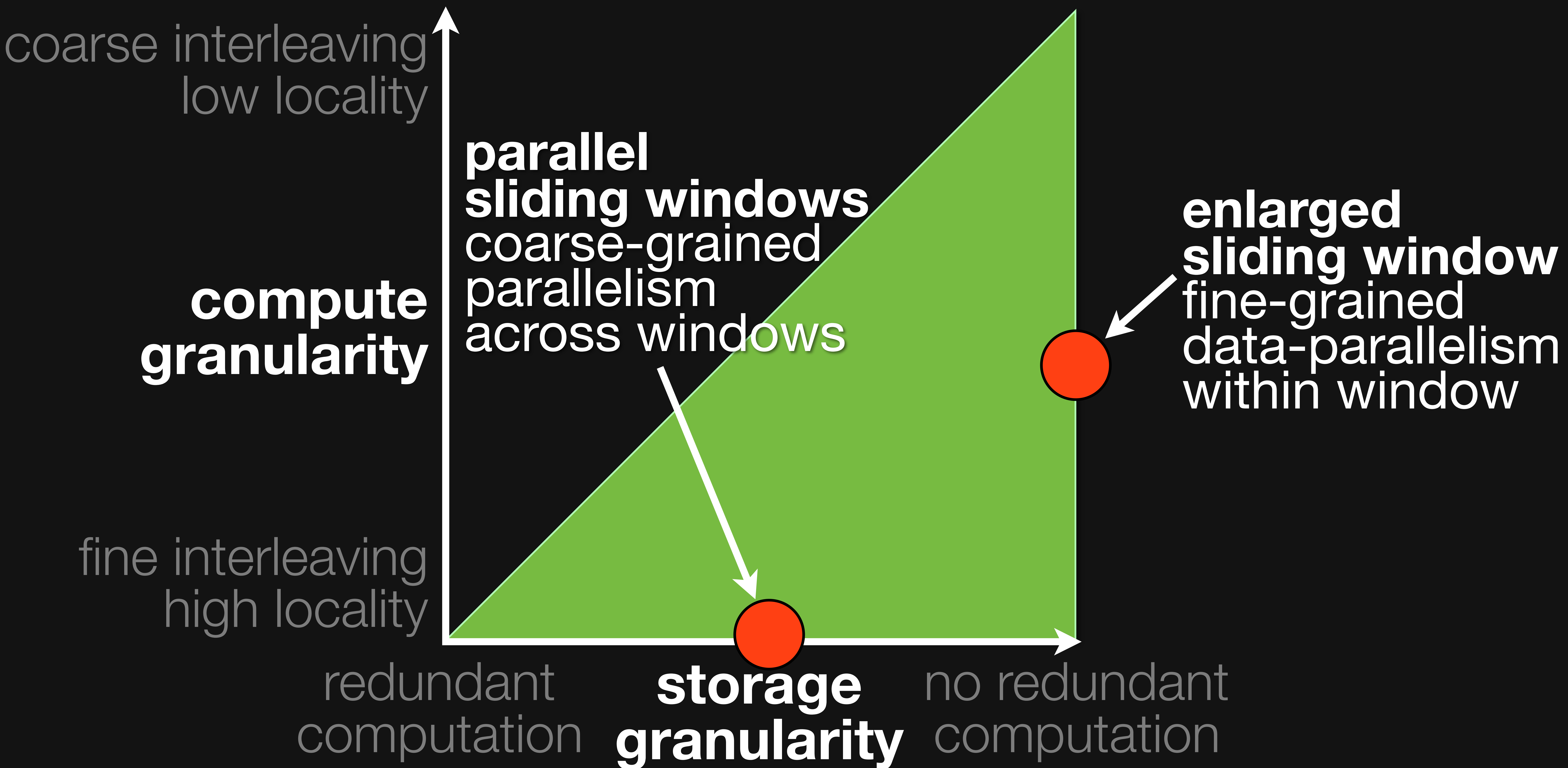
Tradeoff space modeled by **granularity of interleaving**



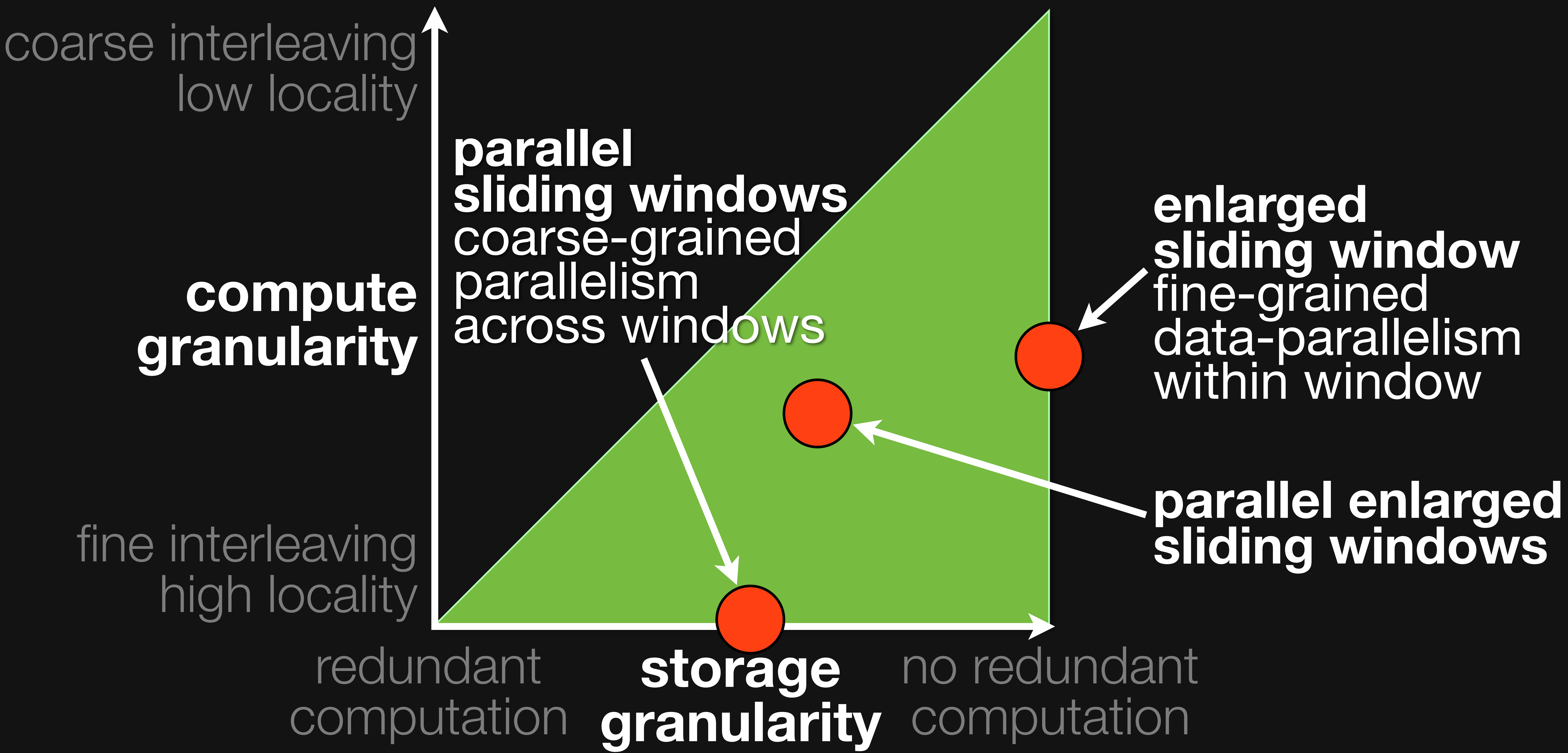
Tradeoff space modeled by **granularity of interleaving**



Tradeoff space modeled by granularity of interleaving

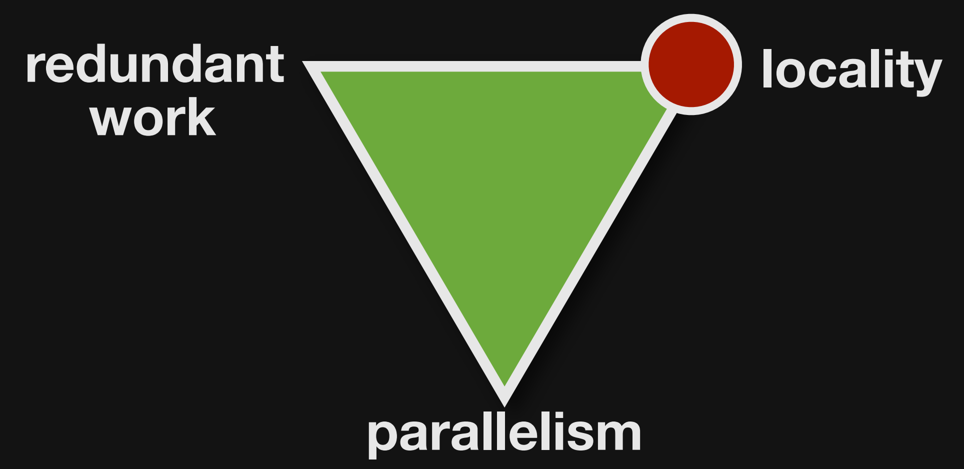


Tradeoff space modeled by **granularity of interleaving**

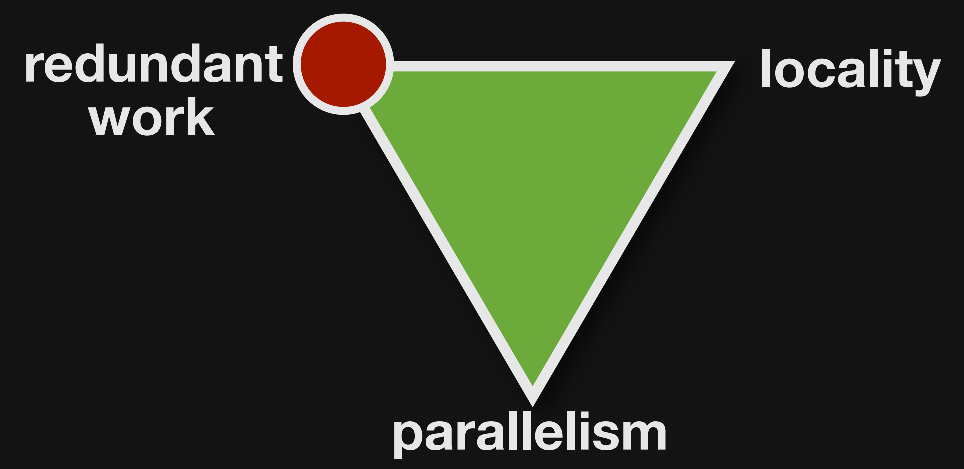


Schedule primitives **compose** to create many organizations

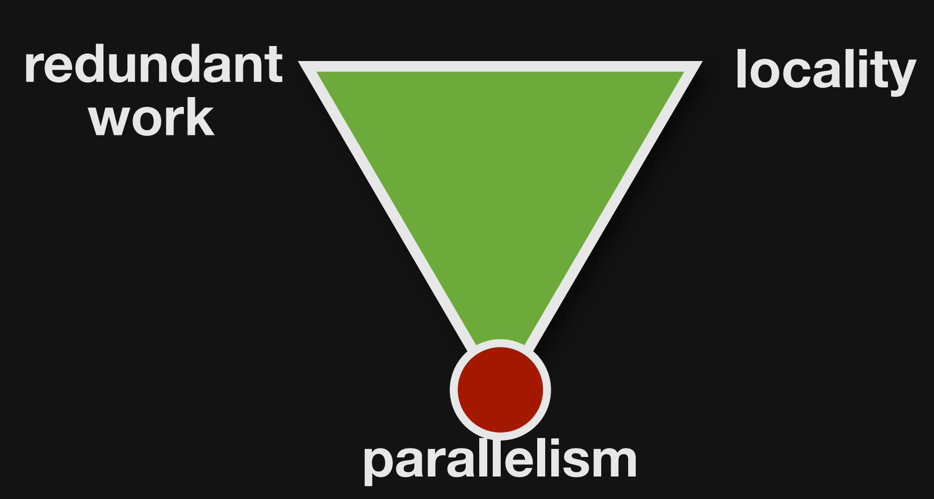
```
blur_x.compute_at_root()
```



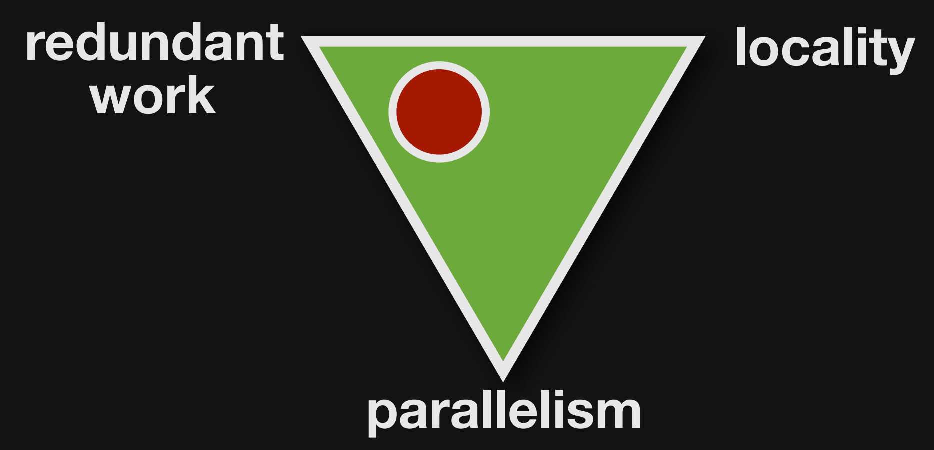

```
blur_x.compute_at(blury, x)
```



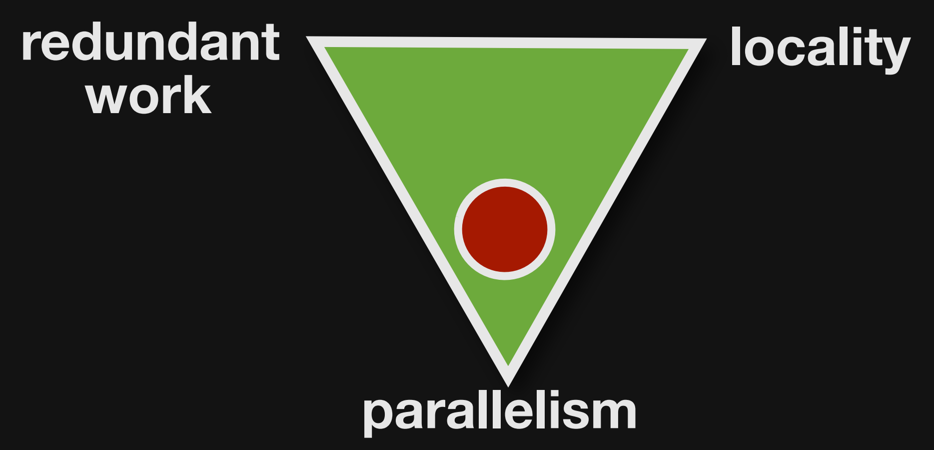

```
blur_x.compute_at(blury, x)
      .store_at_root()
```



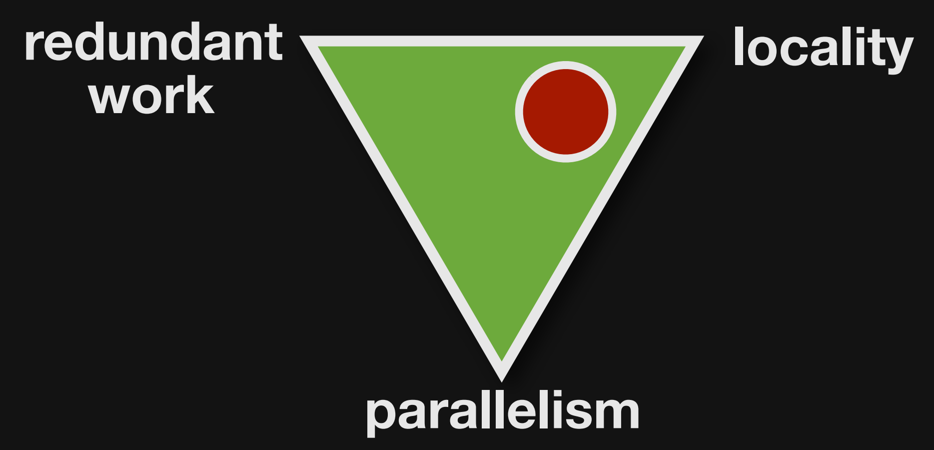

```
blur_x.compute_at(blury, x)
      .vectorize(x, 4)
blur_y.tile(x, y, xi, yi, 8, 8)
      .parallel(y)
      .vectorize(xi, 4)
```




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



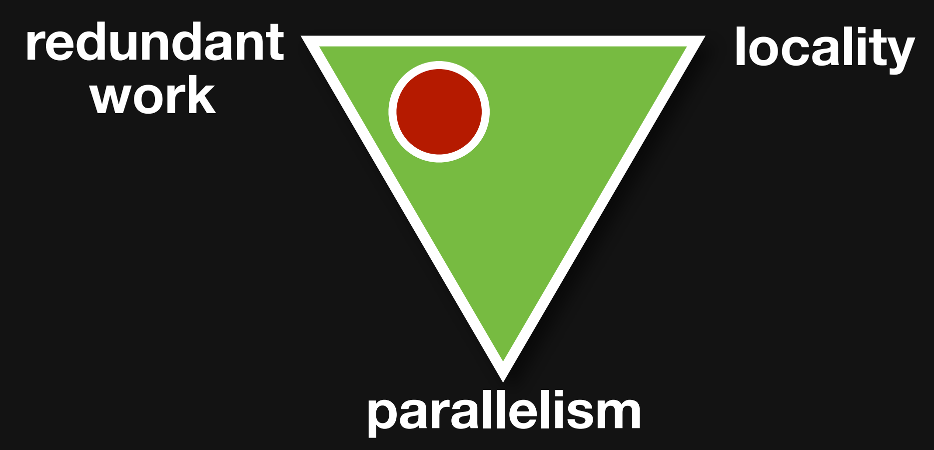

```
blur_x.compute_at(blury, y)
      .store_at(blury, yi)
      .vectorize(x, 4)
blur_y.split(y, y, yi, 8)
      .parallel(y)
      .vectorize(x, 4)
```



Schedule primitives **compose** to create many organizations

Schedule primitives **compose** to create many organizations

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

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




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

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;
}
```

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)
```

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)

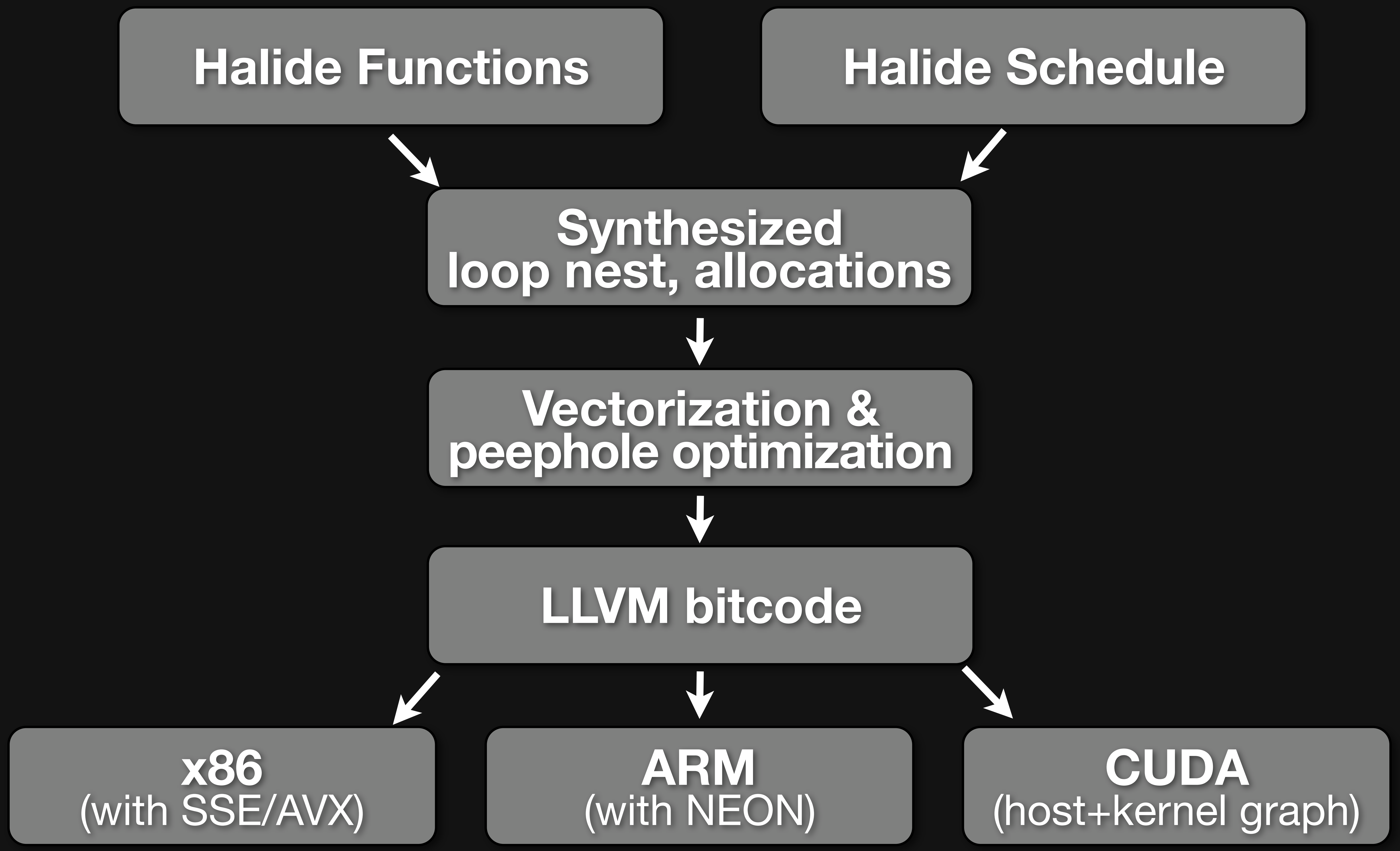
```
f(g(floor(2.3*in(x)))) = in(x)
```

Recursive functions (IIR convolution, scan)

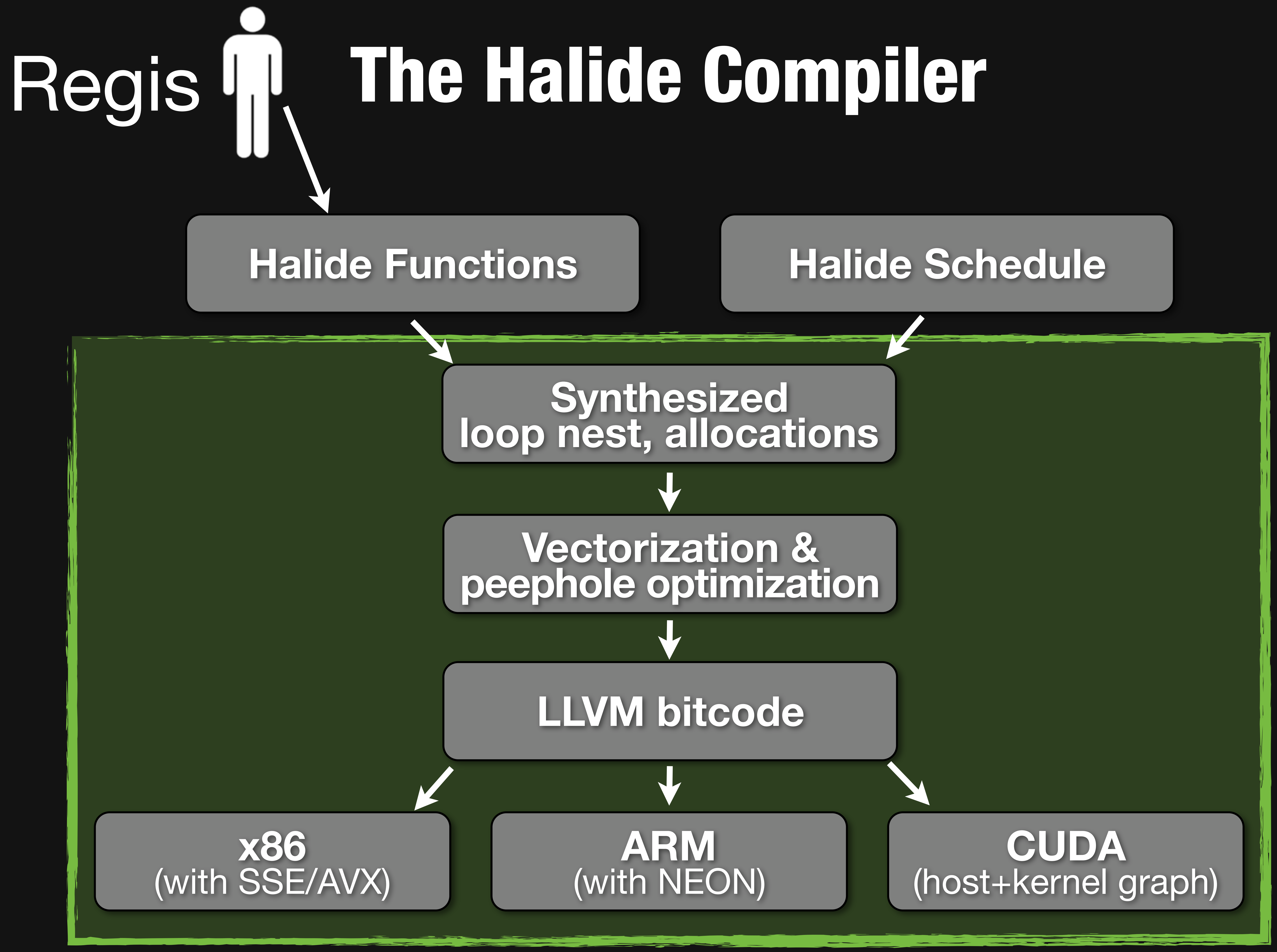
```
cdf(i) = cdf(i-1) + pdf(i)
```

Reductions
histogram, etc.

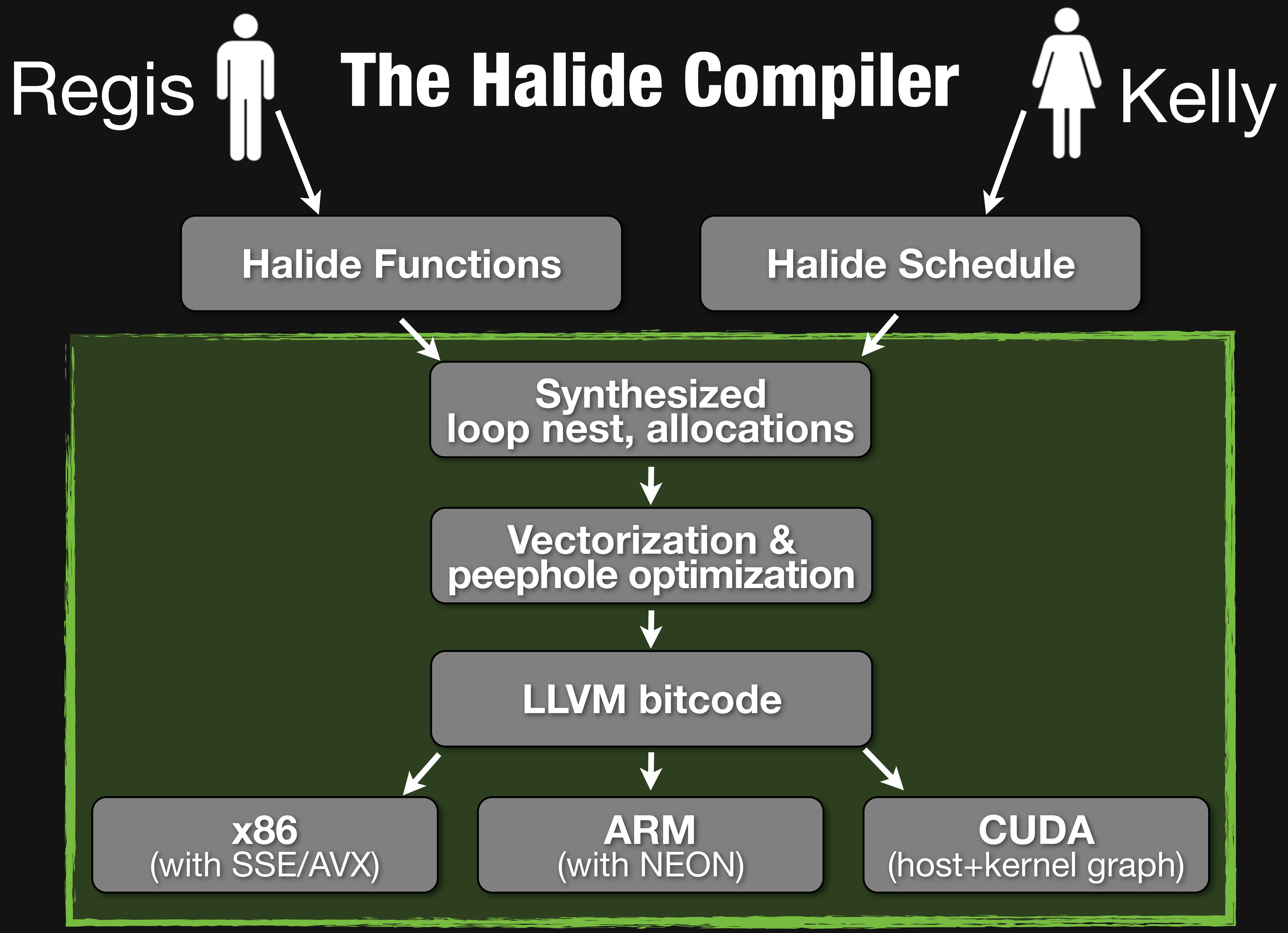
The Halide Compiler



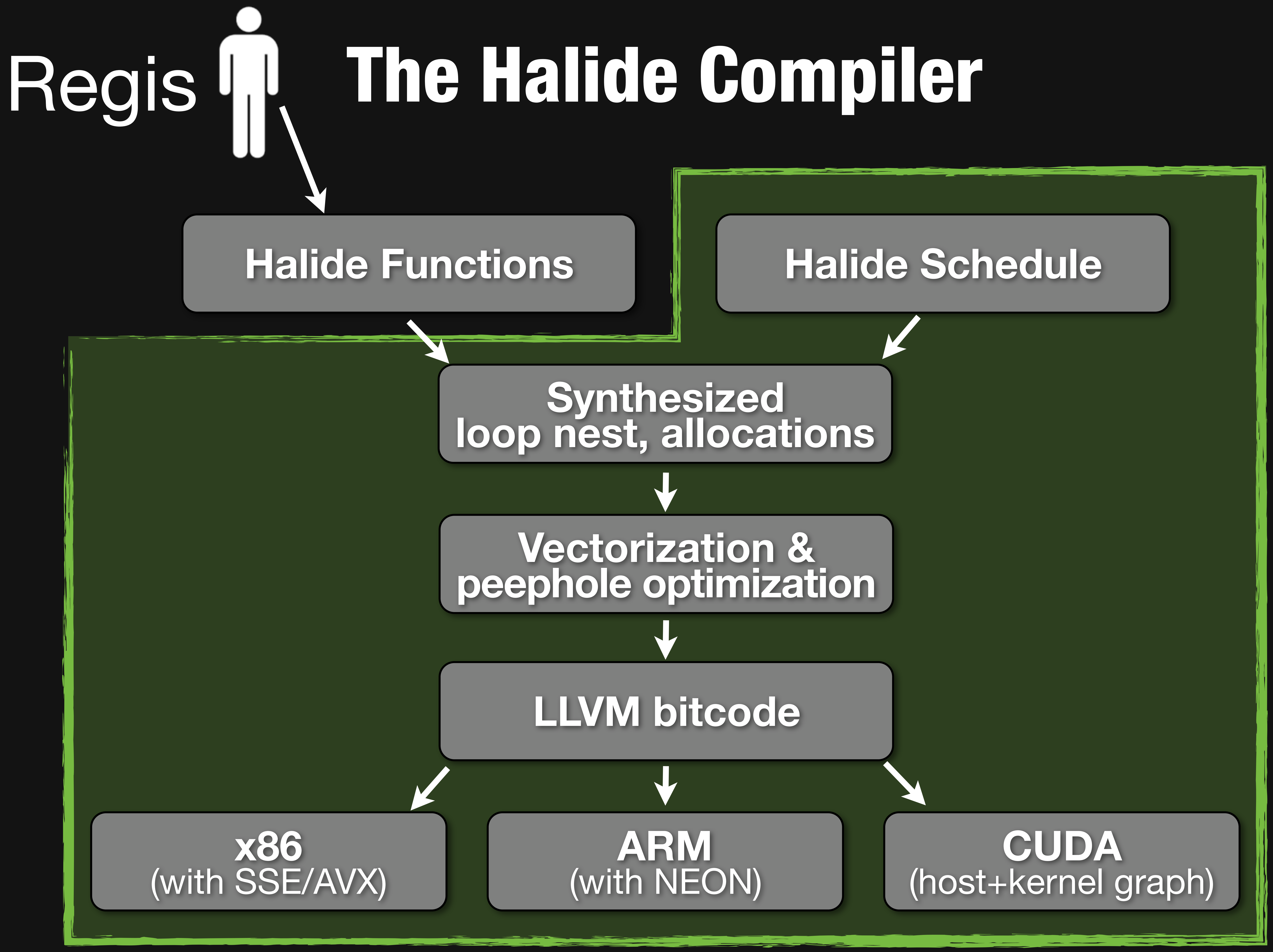
The Halide Compiler



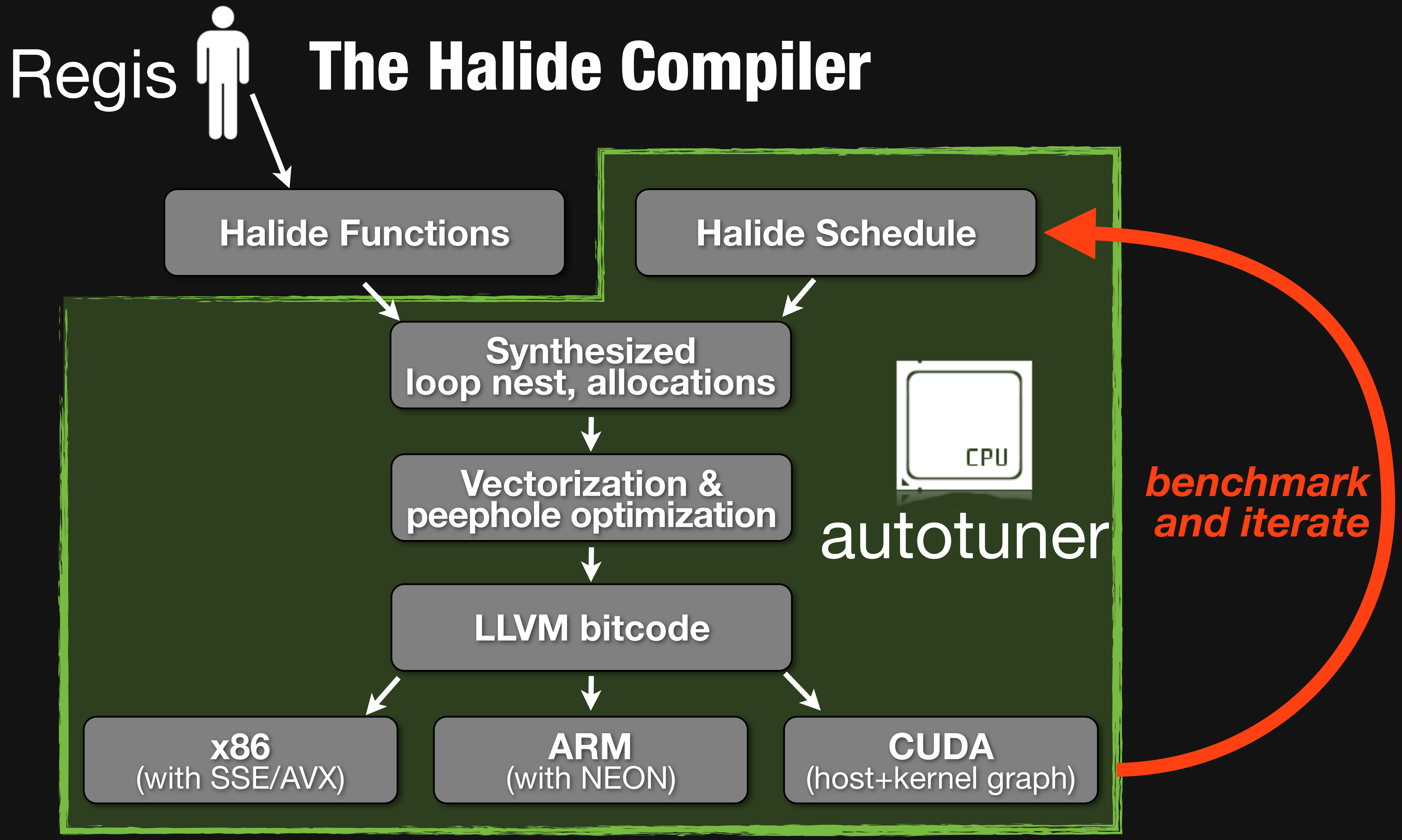
The Halide Compiler



The Halide Compiler



The Halide Compiler



Results

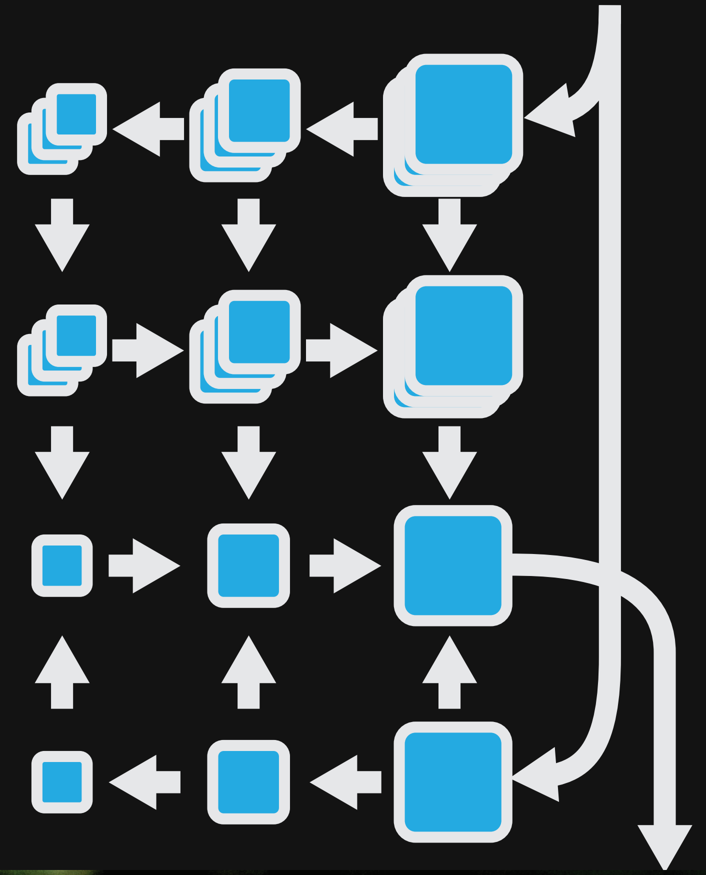
Local Laplacian Filters

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

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,
9x faster on GPU



Local Laplacian Filters

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

Adobe: 1500 lines of expert-optimized C++

multi-threaded, SSE

3 months of work

10x faster than original C++

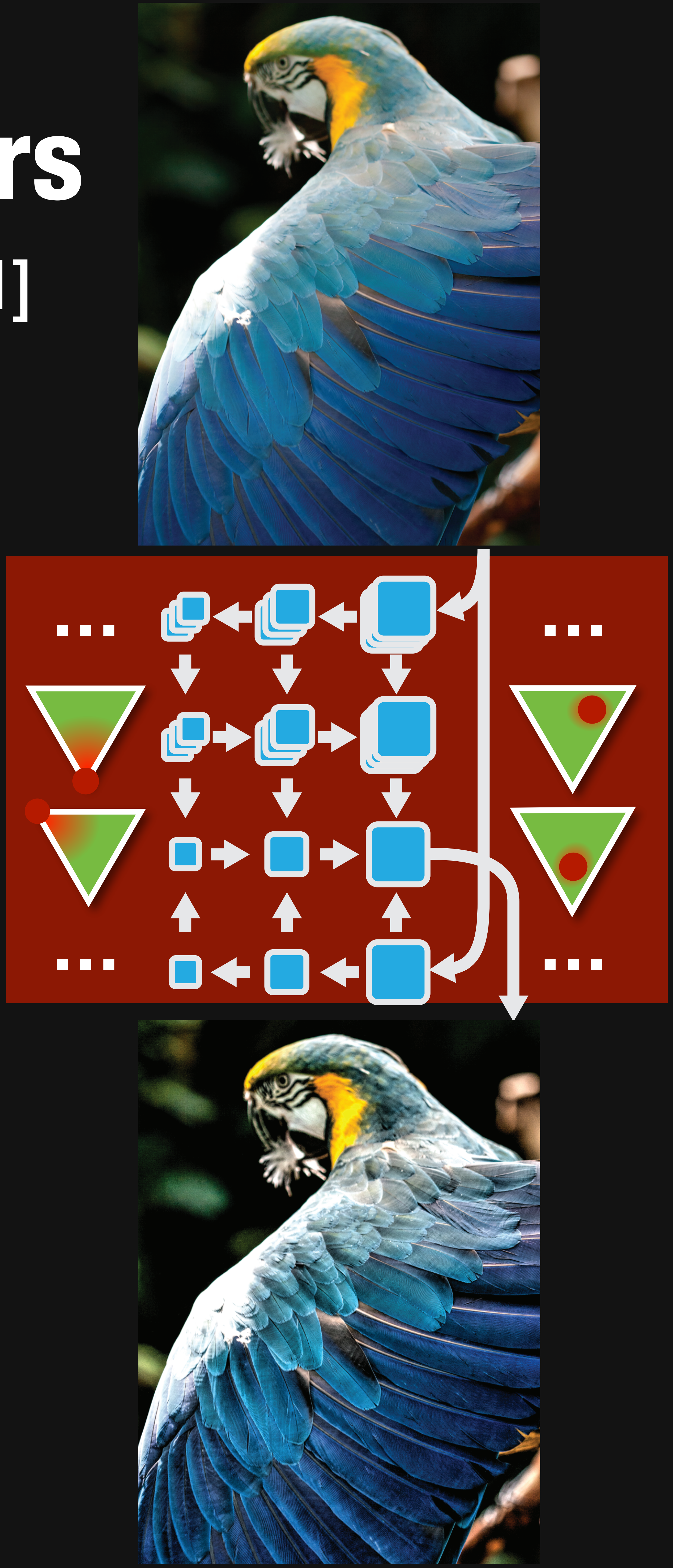
Halide: 60 lines

1 intern-day

Halide vs. Adobe:

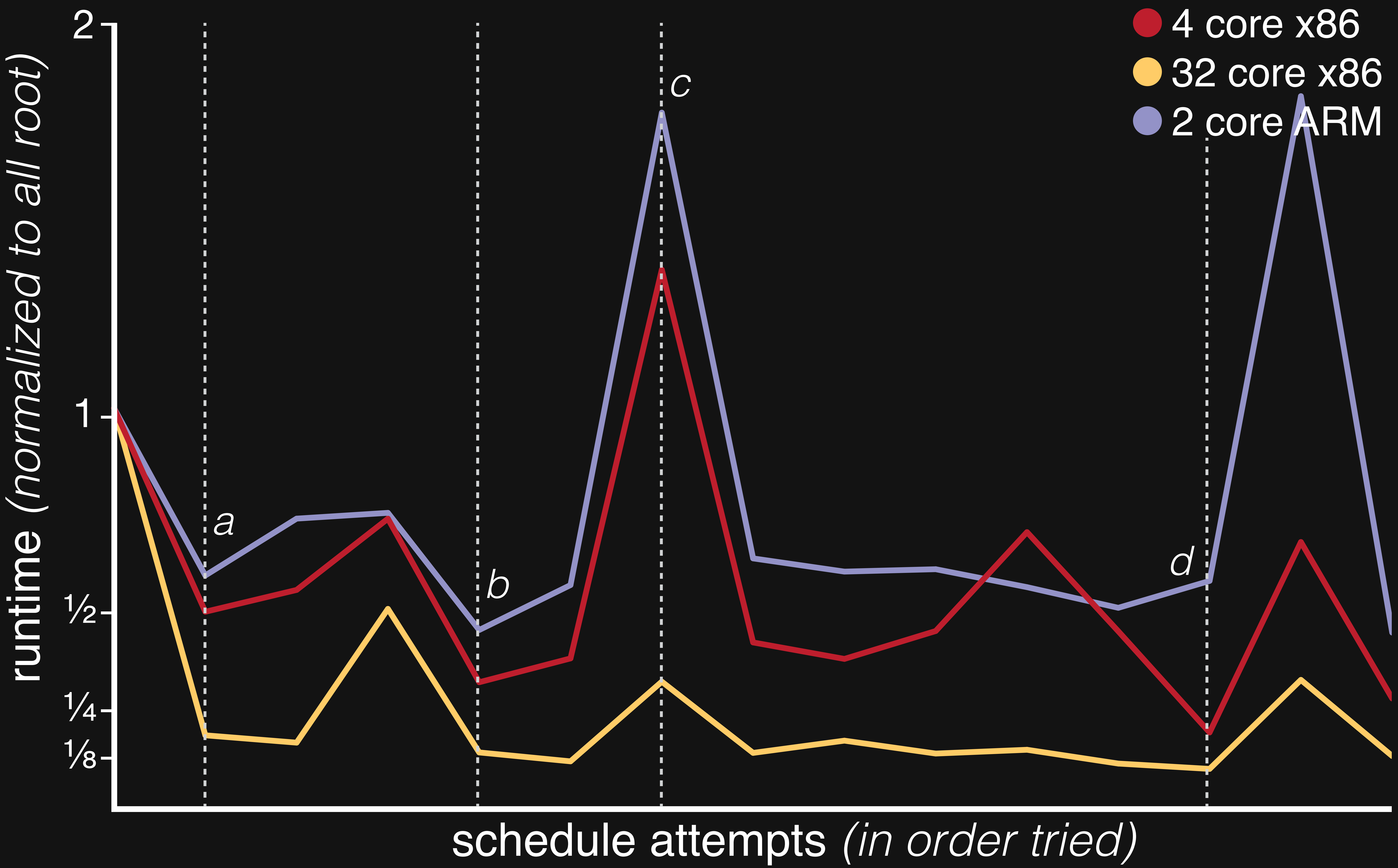
2x faster on same CPU,

9x faster on GPU



Local Laplacian Filters

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

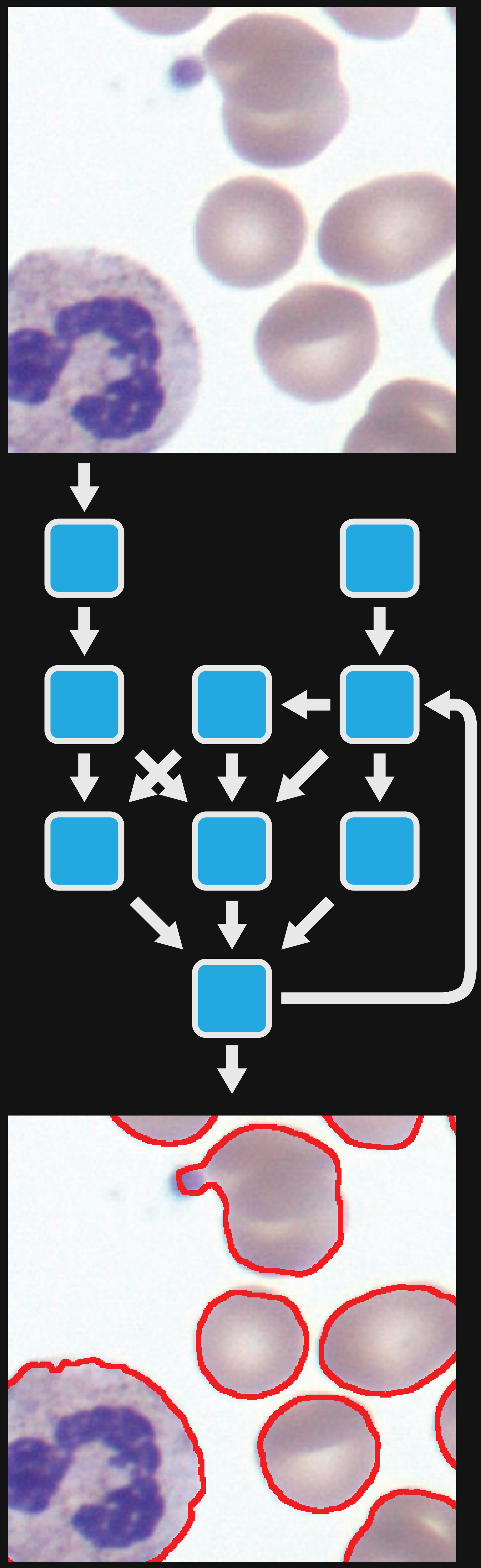


“Snake” Image Segmentation

[Li et al. 2010]

Segments objects in an image using level-sets

Original: 67 lines of matlab



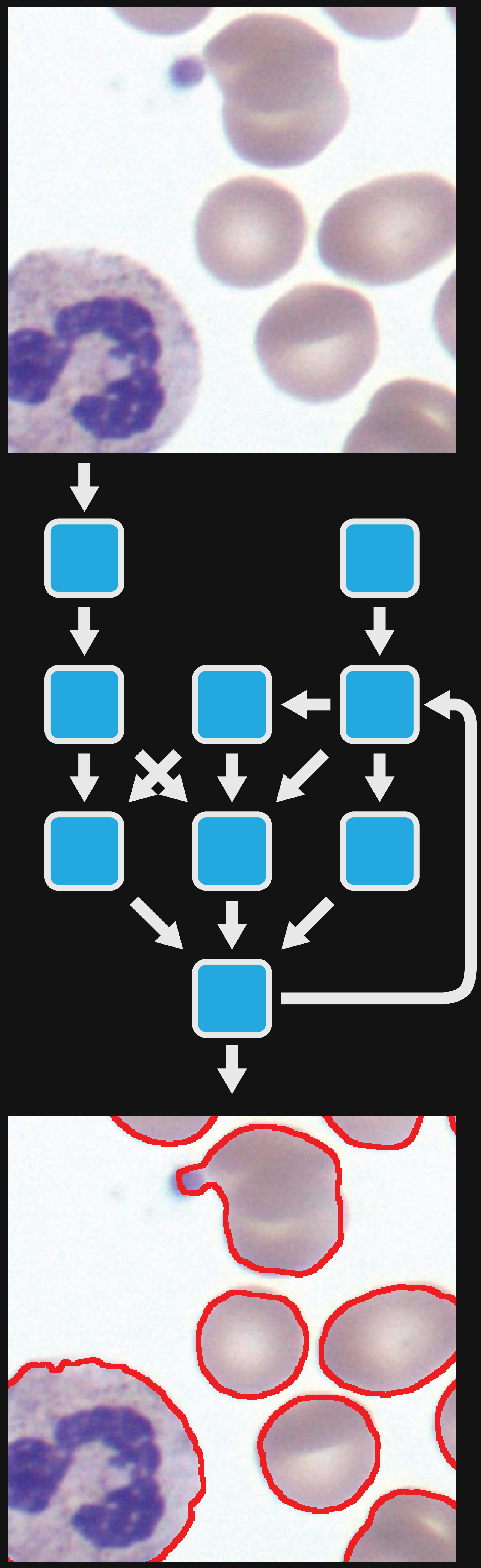
“Snake” Image Segmentation

[Li et al. 2010]

Segments objects in an image using level-sets

Original: 67 lines of matlab

Halide: 148 lines of algorithm, 7 lines of schedule



“Snake” Image Segmentation

[Li et al. 2010]

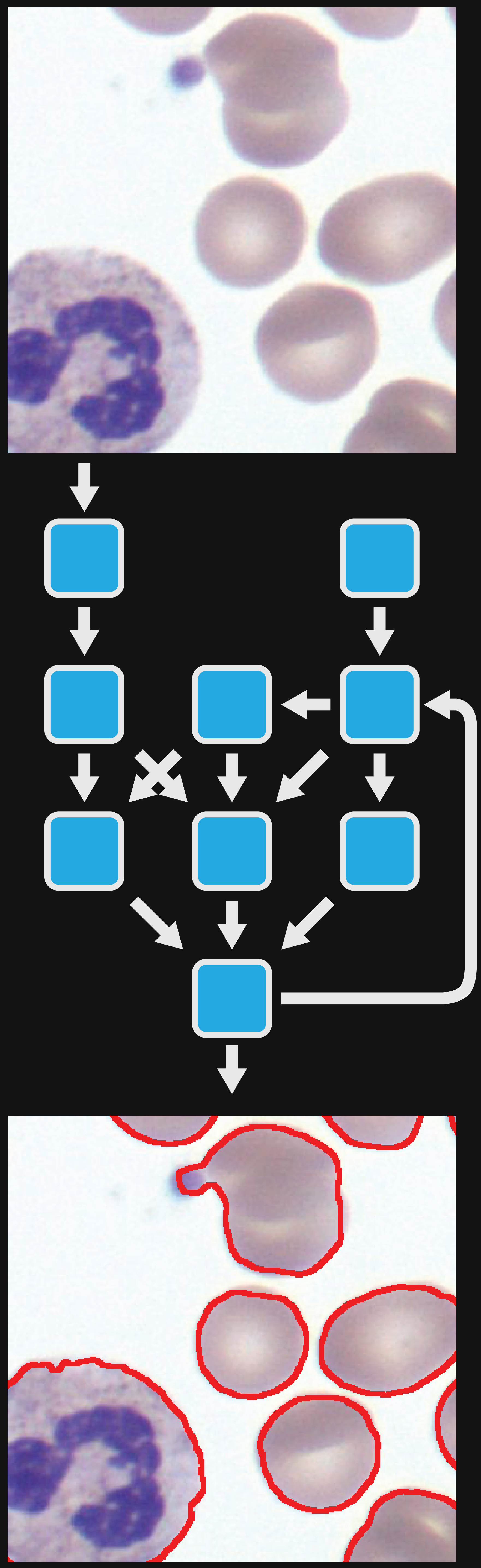
Segments objects in an image using level-sets

Original: 67 lines of matlab

Halide: 148 lines of algorithm, 7 lines of schedule

On the CPU, 70x faster

MATLAB is memory-bandwidth limited



"Snake" Image Segmentation

[Li et al. 2010]

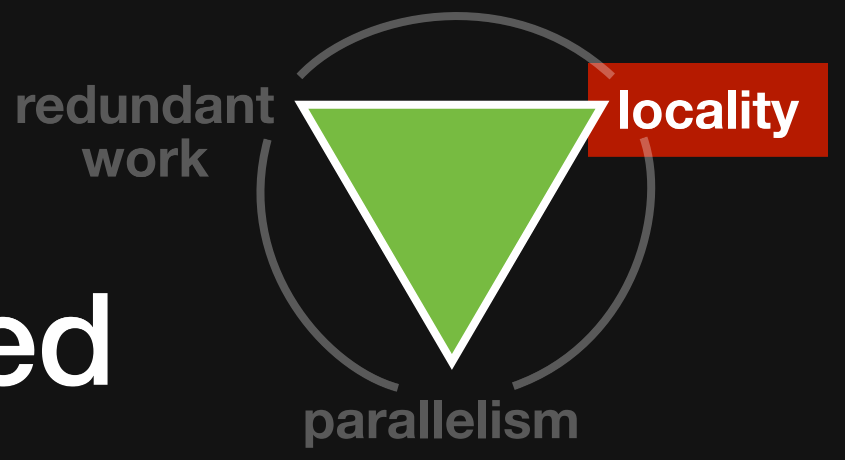
Segments objects in an image using level-sets

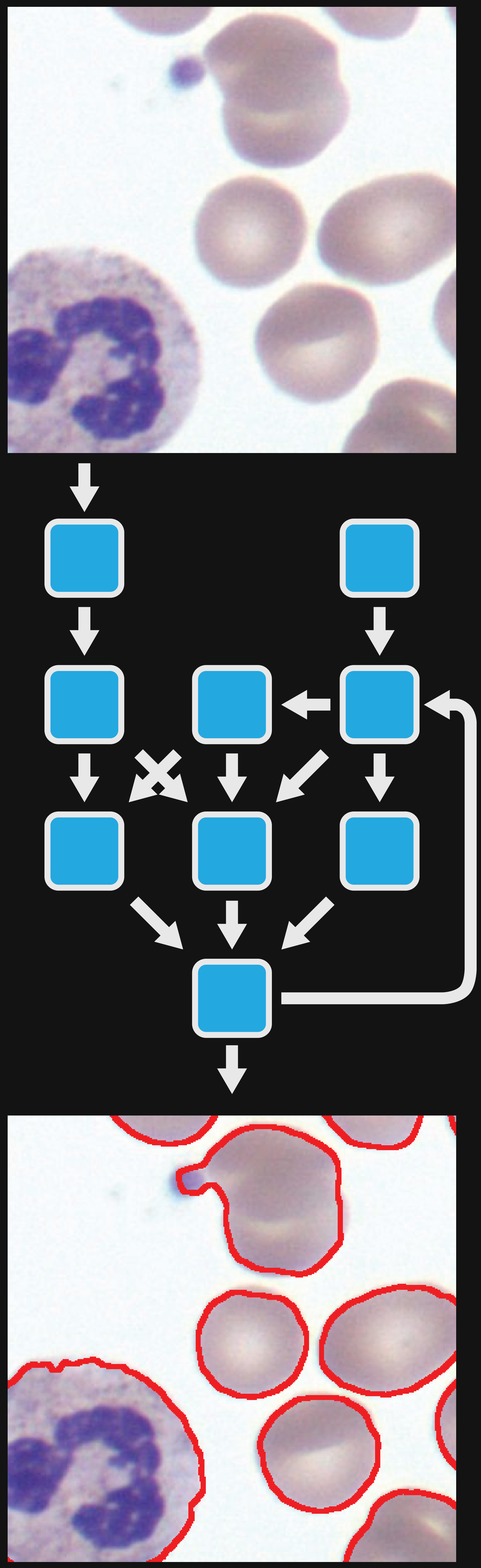
Original: 67 lines of matlab

Halide: 148 lines of algorithm, 7 lines of schedule

On the CPU, 70x faster

MATLAB is memory-bandwidth limited





"Snake" Image Segmentation

[Li et al. 2010]

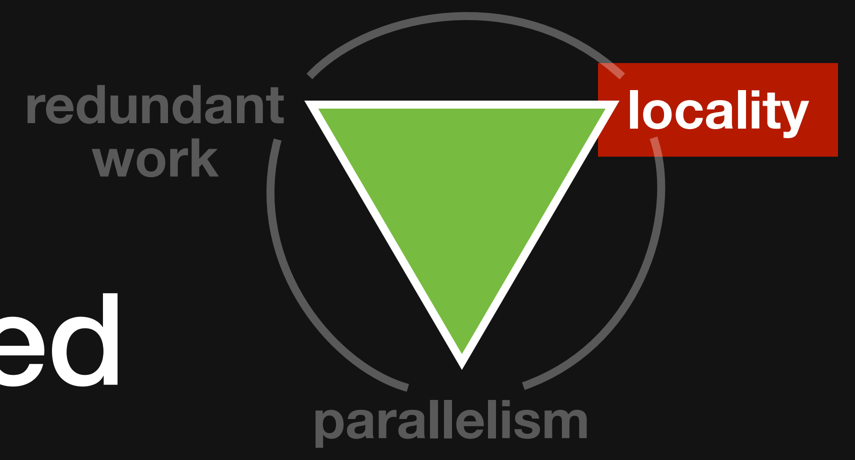
Segments objects in an image using level-sets

Original: 67 lines of matlab

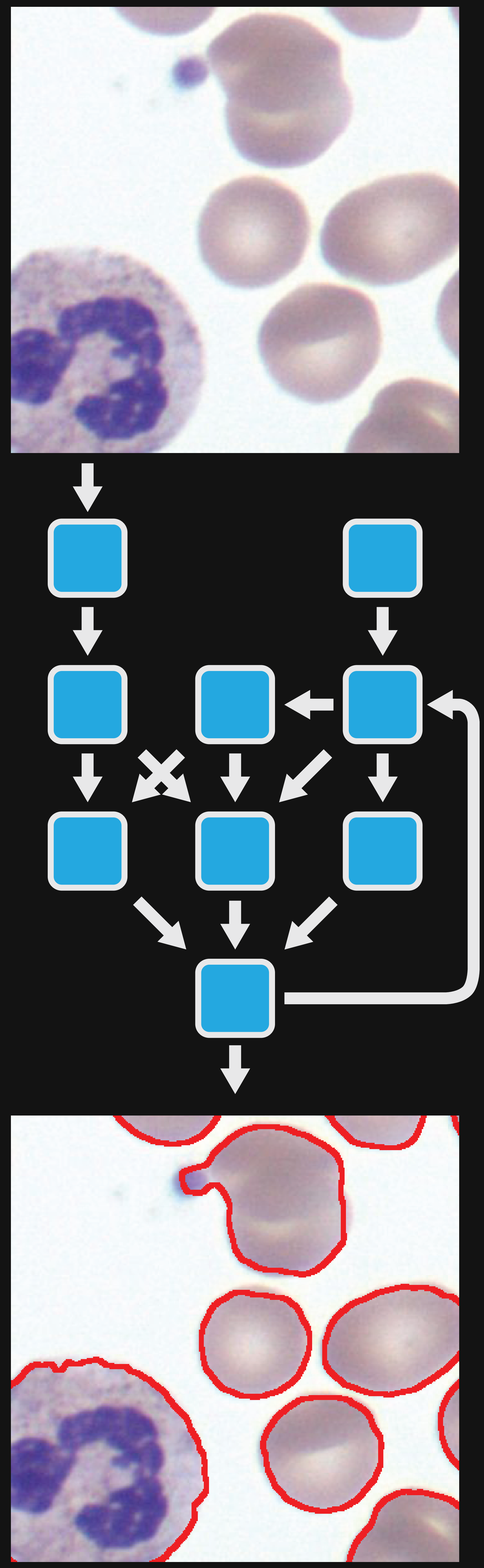
Halide: 148 lines of algorithm, 7 lines of schedule

On the CPU, 70x faster

MATLAB is memory-bandwidth limited



On the GPU, 1250x faster

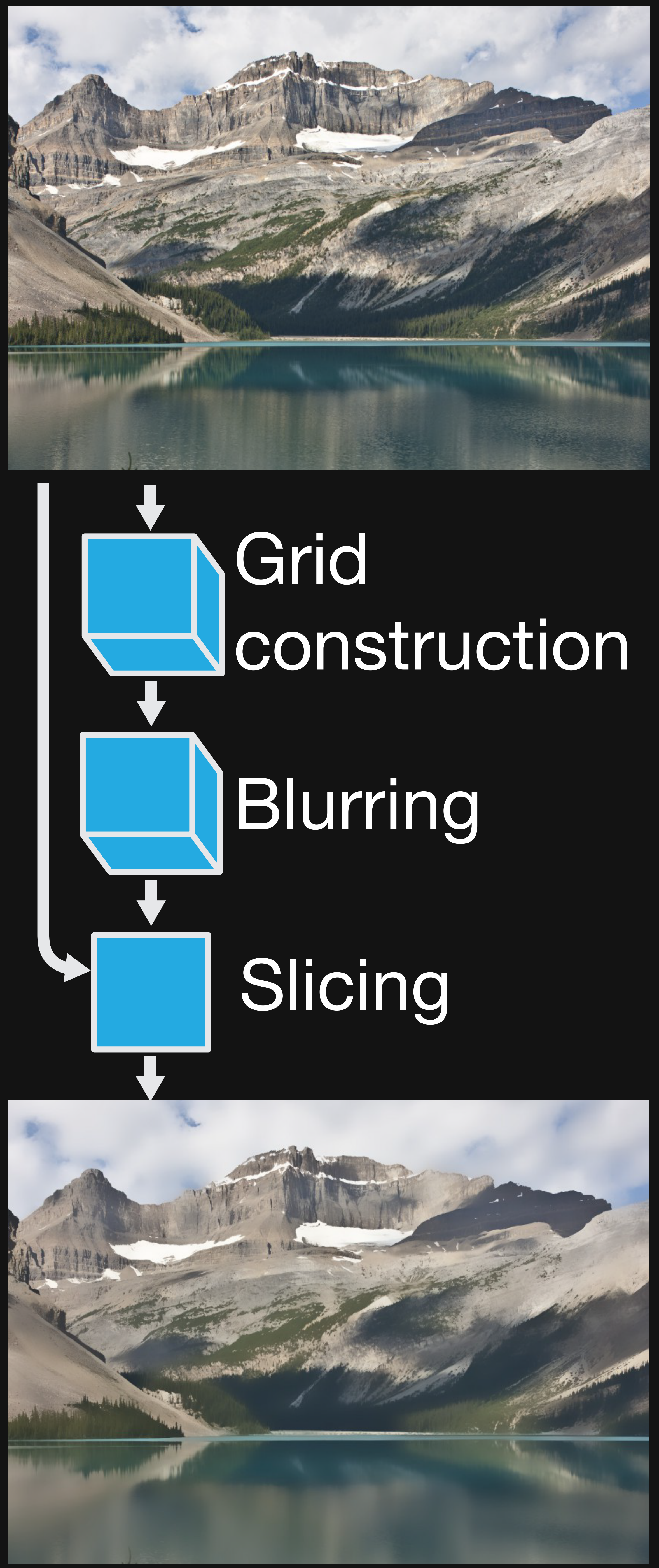


The Bilateral Grid

[Chen et al. 2007]

An accelerated bilateral filter

Original: 122 lines of (clean) C++



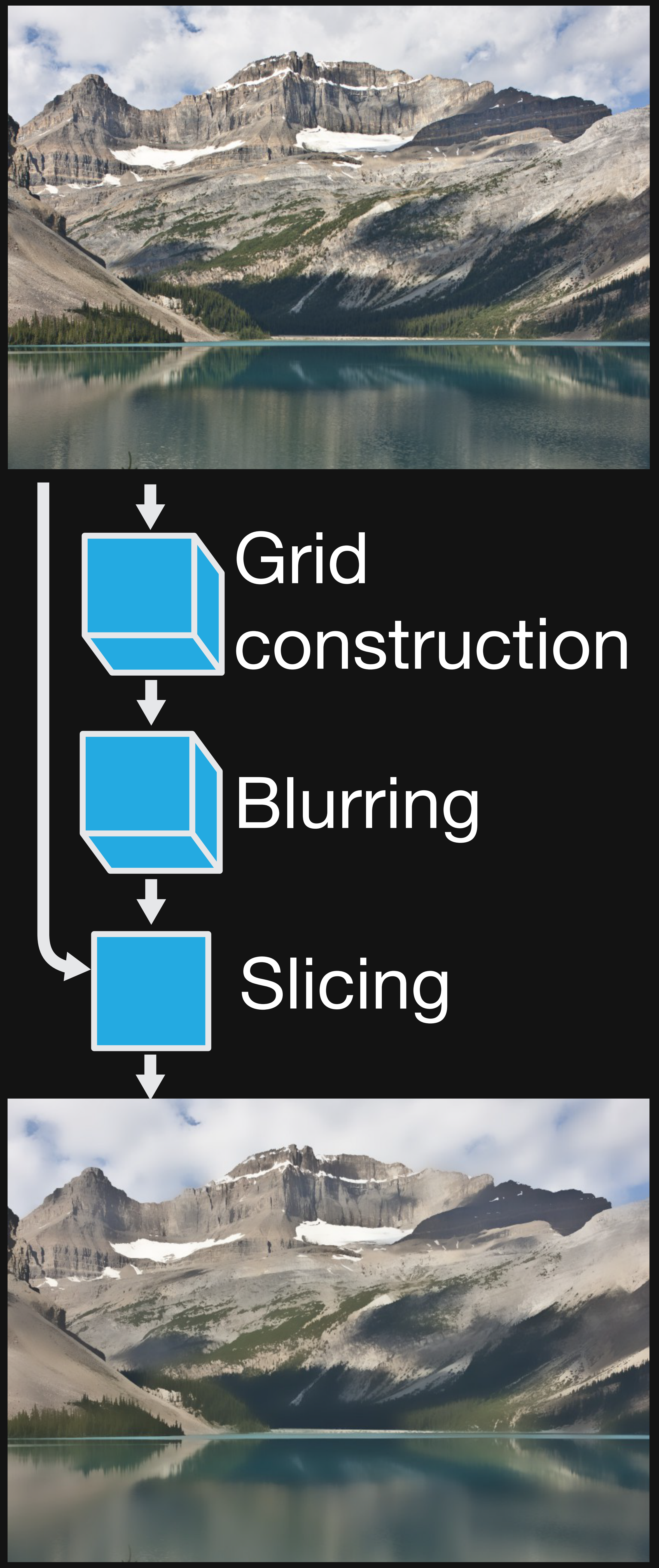
The Bilateral Grid

[Chen et al. 2007]

An accelerated bilateral filter

Original: 122 lines of (clean) C++

Halide: 34 lines of algorithm, 6 lines of schedule



The Bilateral Grid

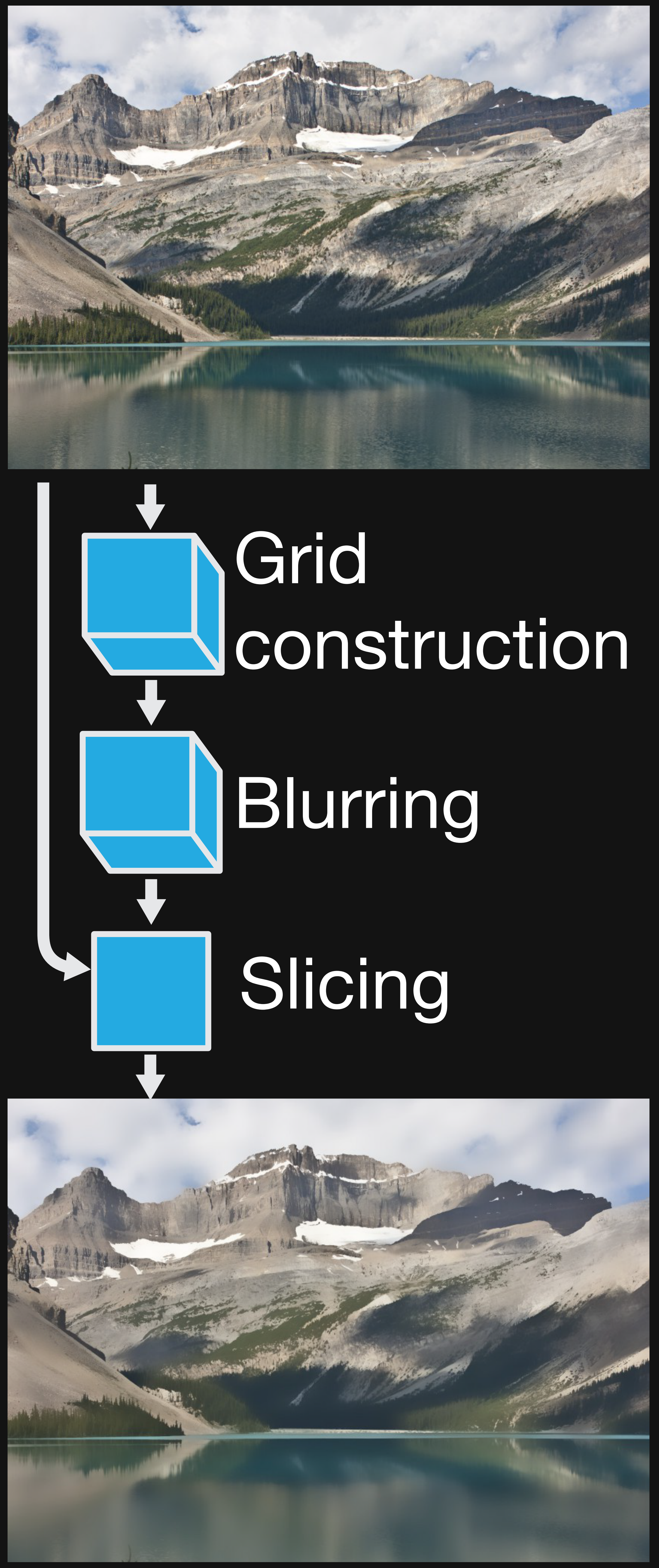
[Chen et al. 2007]

An accelerated bilateral filter

Original: 122 lines of (clean) C++

Halide: 34 lines of algorithm, 6 lines of schedule

On the CPU, 5.9x faster



The Bilateral Grid

[Chen et al. 2007]

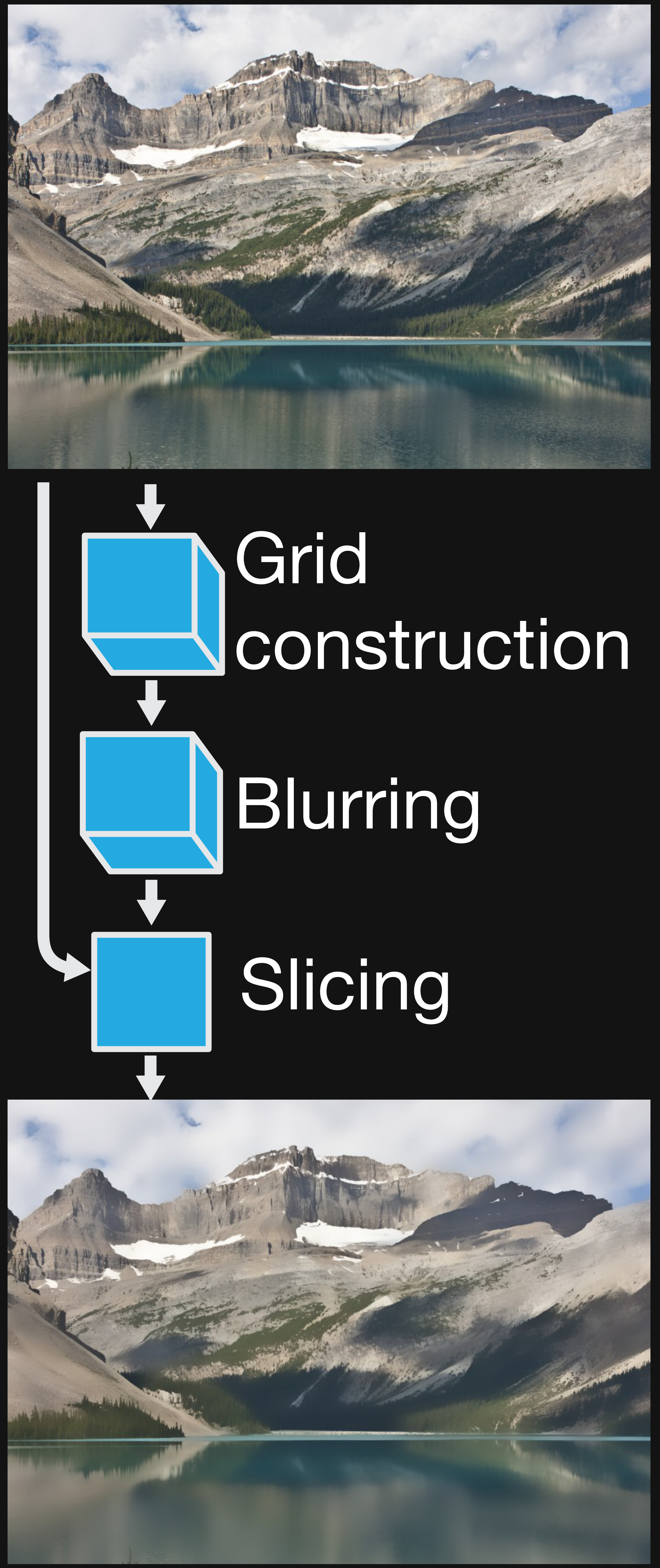
An accelerated bilateral filter

Original: 122 lines of (clean) C++

Halide: 34 lines of algorithm, 6 lines of schedule

On the CPU, 5.9x faster

On the GPU, 2x faster than Chen's hand-written CUDA version (*and equivalent Halide schedule*)



The Bilateral Grid

[Chen et al. 2007]

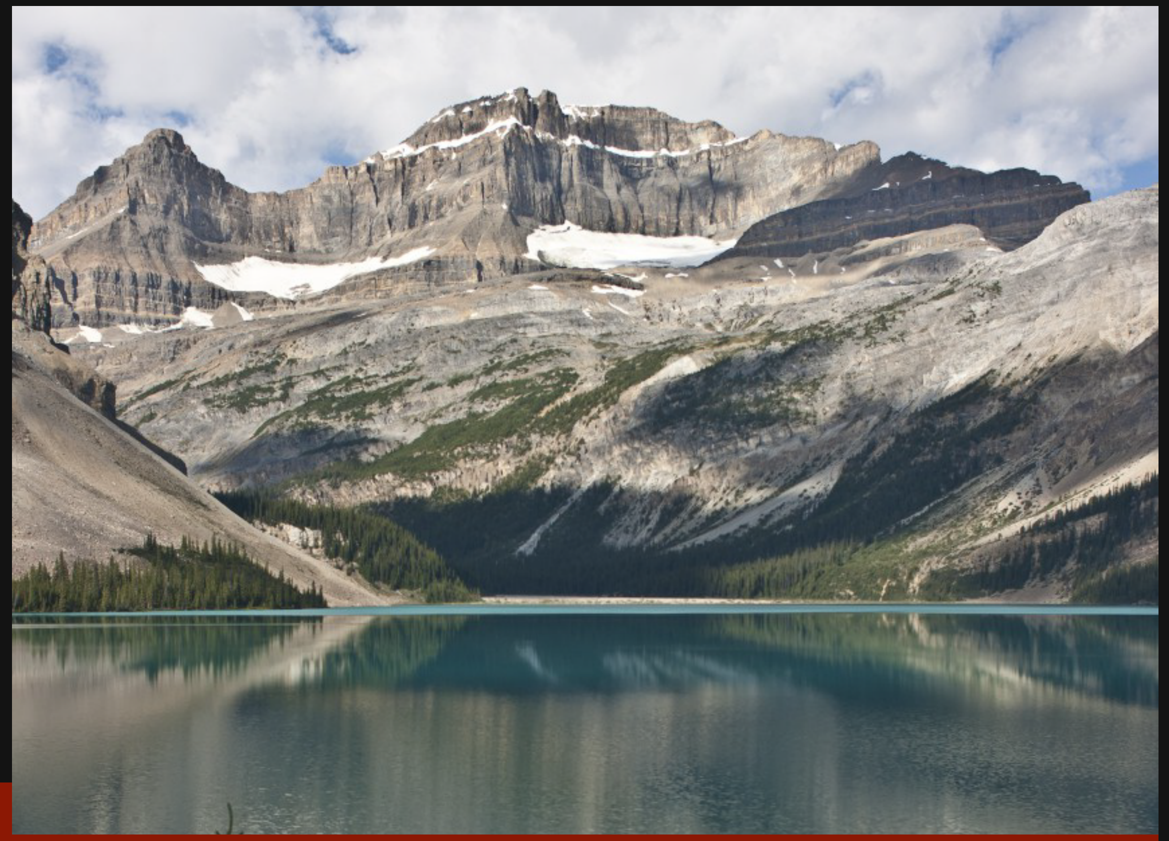
An accelerated bilateral filter

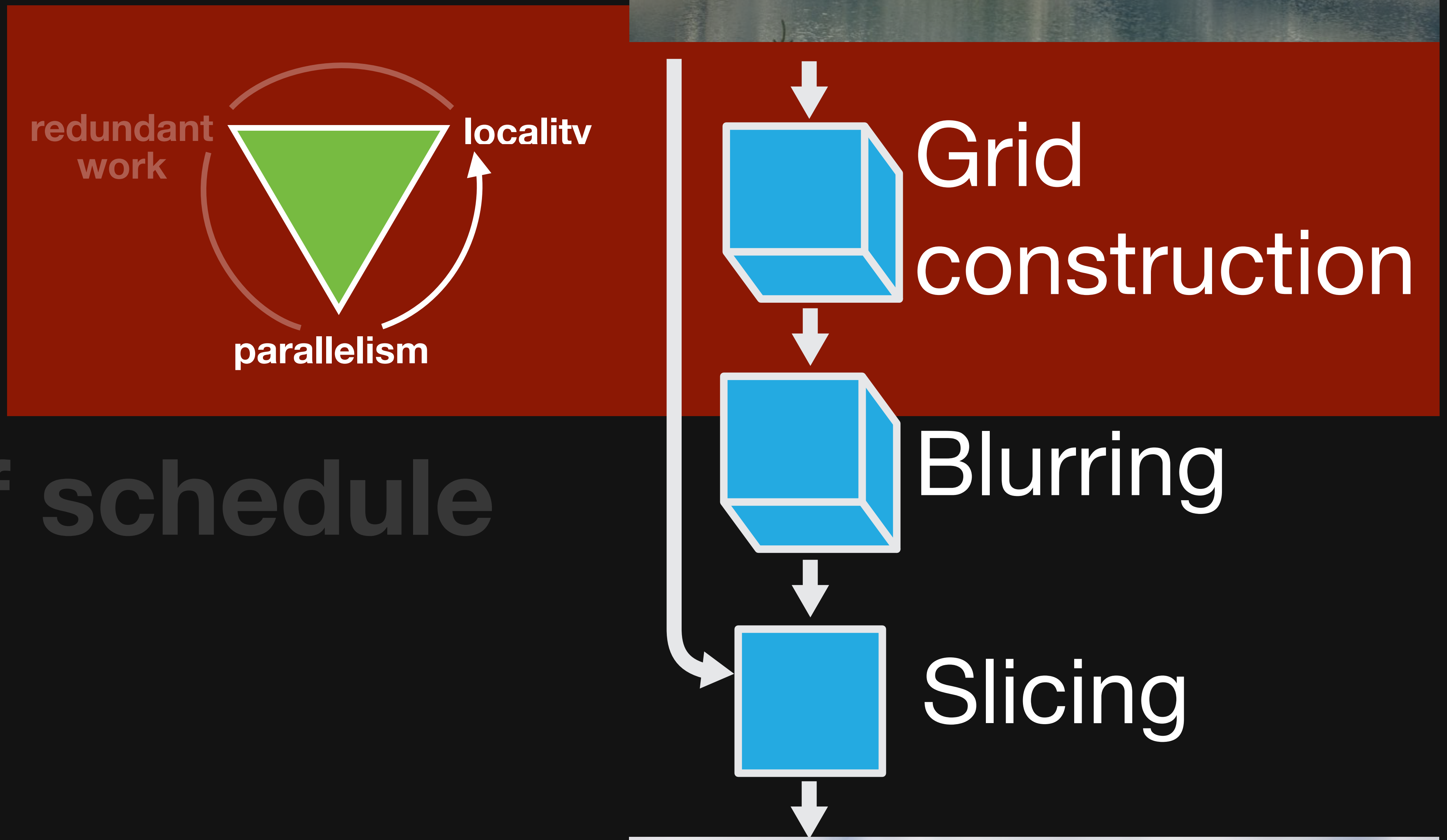
Original: 122 lines of (clean) C++

Halide: 34 lines of algorithm, 6 lines of schedule

On the CPU, 5.9x faster

On the GPU, 2x faster than Chen's hand-written CUDA version (*and equivalent Halide schedule*)



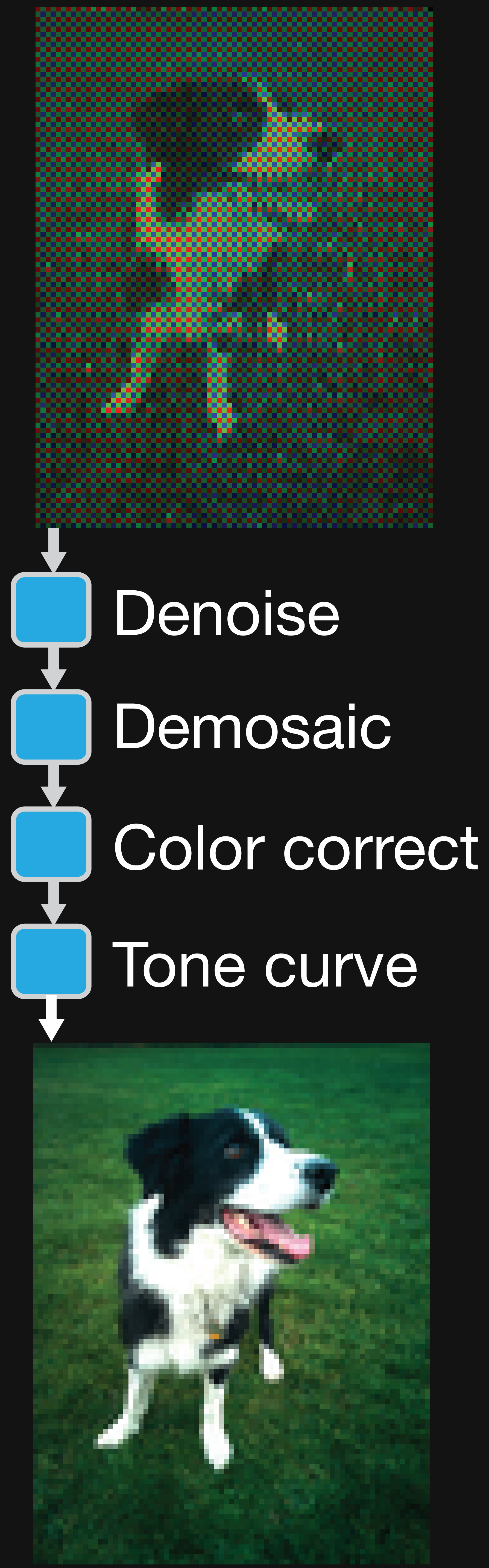


The Frankencamera Raw Pipeline

[Adams et al. 2010]

Converts raw image sensor data into an image

Original: 463 lines of ARM assembly and intrinsics in one big function



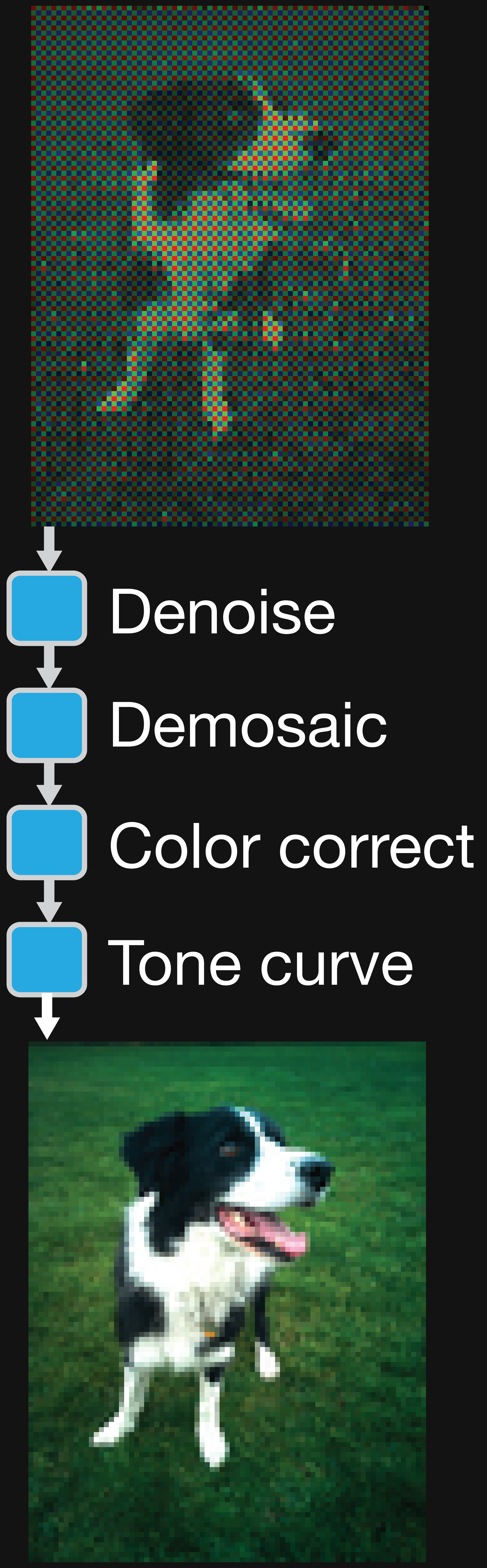
The Frankencamera Raw Pipeline

[Adams et al. 2010]

Converts raw image sensor data into an image

Original: 463 lines of ARM assembly and intrinsics in one big function

Rewritten in Halide, it is 2.75x less code, and runs 5% faster



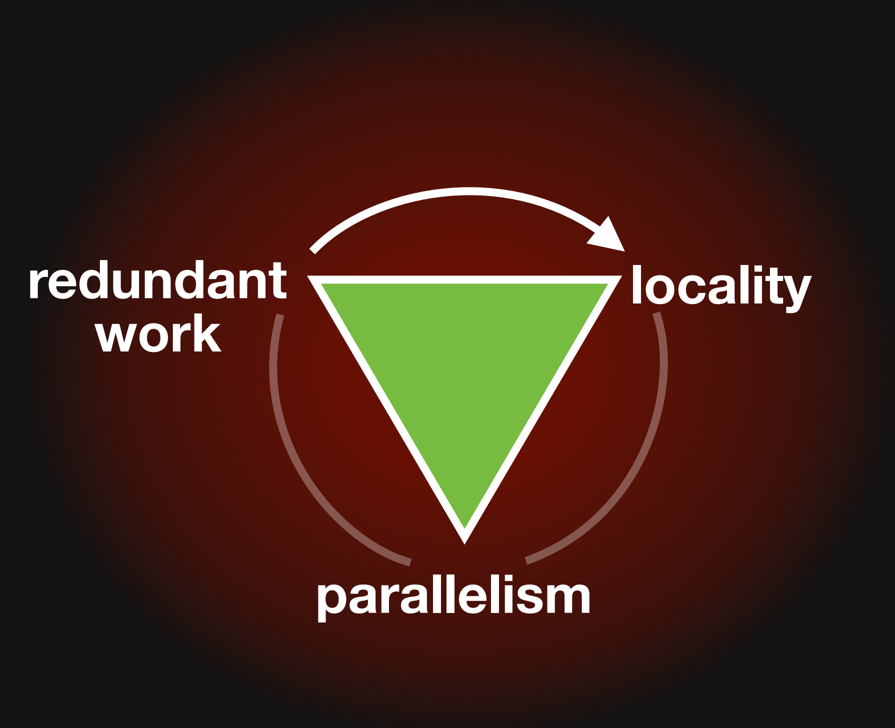
The Frankencamera Raw Pipeline

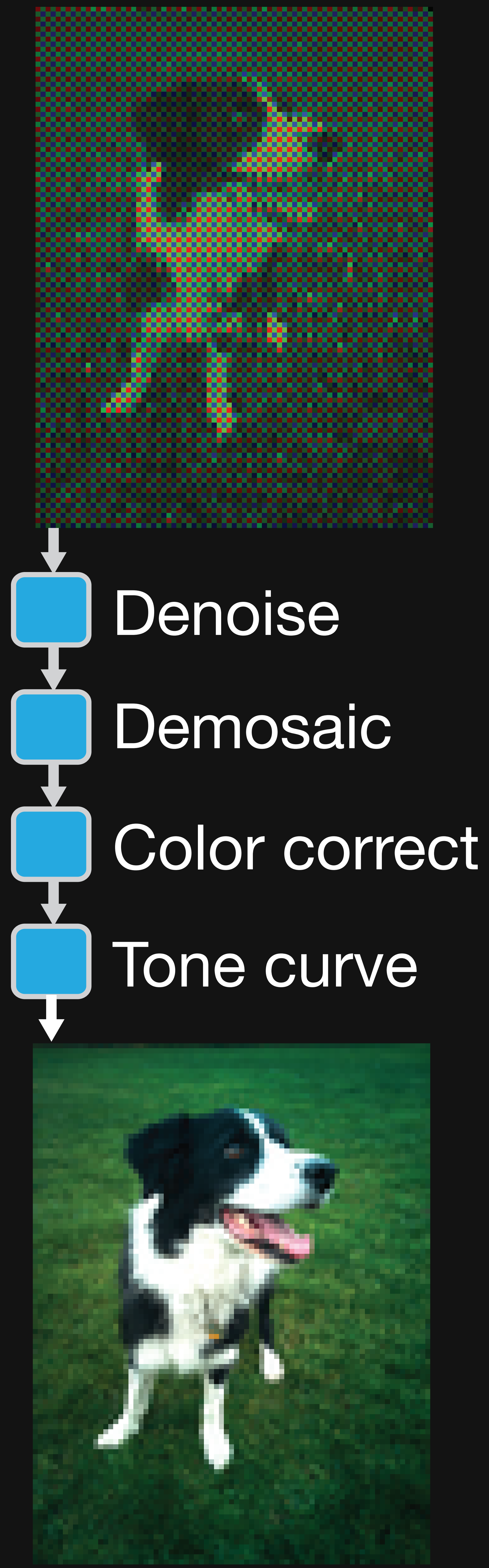
[Adams et al. 2010]

Converts raw image sensor data into an image

Original: 463 lines of ARM assembly and intrinsics in one big function

Rewritten in Halide, it is 2.75x less code, and runs 5% faster





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

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

ARM	Speedup	Factor shorter
Camera pipeline	1.1 ×	3 ×

x86	Speedup	Factor shorter
Blur	1.2 ×	18 ×
Bilateral Grid	4.4 ×	4 ×
Camera pipeline	3.4 ×	2 ×
"Healing brush"	1.7 ×	7 ×
Local Laplacian	1.7 ×	5 ×
Gaussian Blur	1.5 ×	5 ×
FFT (vs. FFTW)	1.5 ×	*10s*
BLAS (vs. Eigen)	1 ×	*100s*

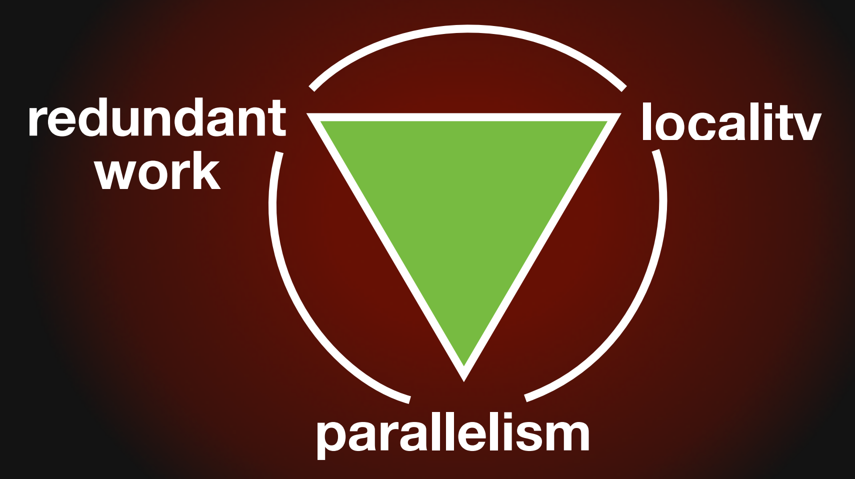
GPU	Speedup	Factor shorter
Bilateral Grid	2.3 ×	11 ×
"Healing brush"	*5.9** ×	*7** ×
Local Laplacian	*9** ×	*7** ×

ARM	Speedup	Factor shorter
Camera pipeline	1.1 ×	3 ×

Summary

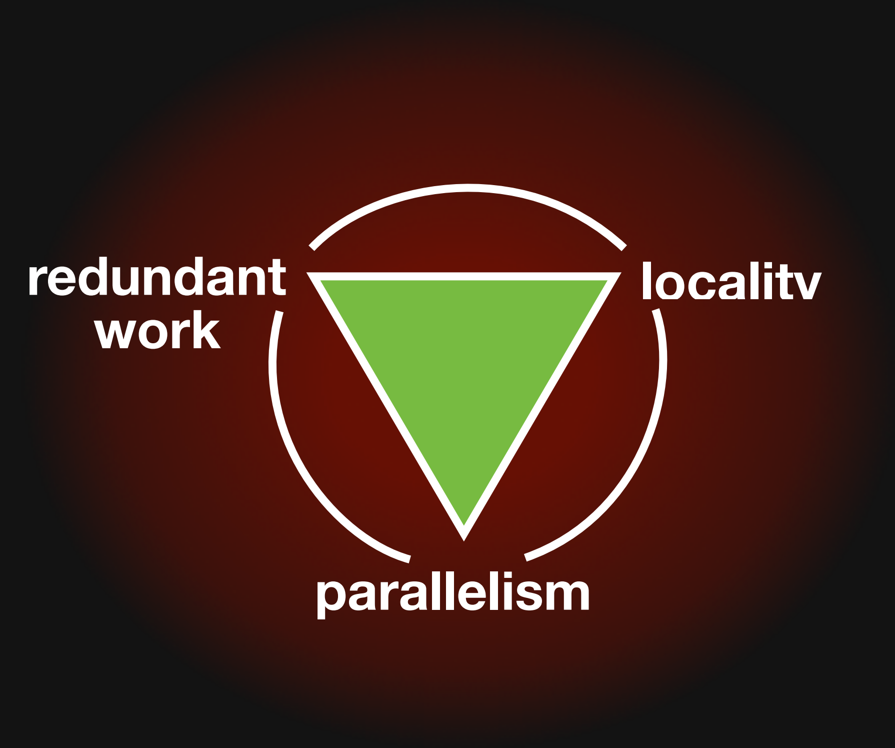
Summary

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



Summary

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



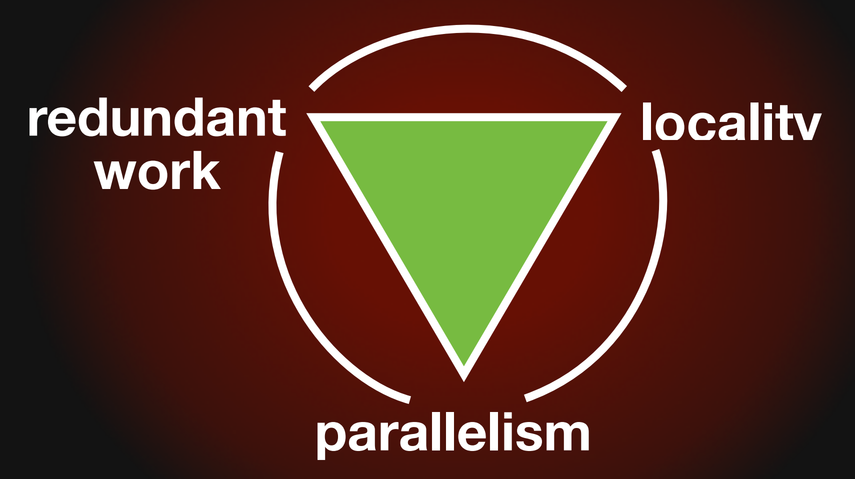
Schedule:

Order *within* stage
Interleaving *between* stages

→ Loop nest & storage allocations

Summary

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



Schedule:

Order *within* stage
Interleaving *between* stages

→ Loop nest & storage allocations

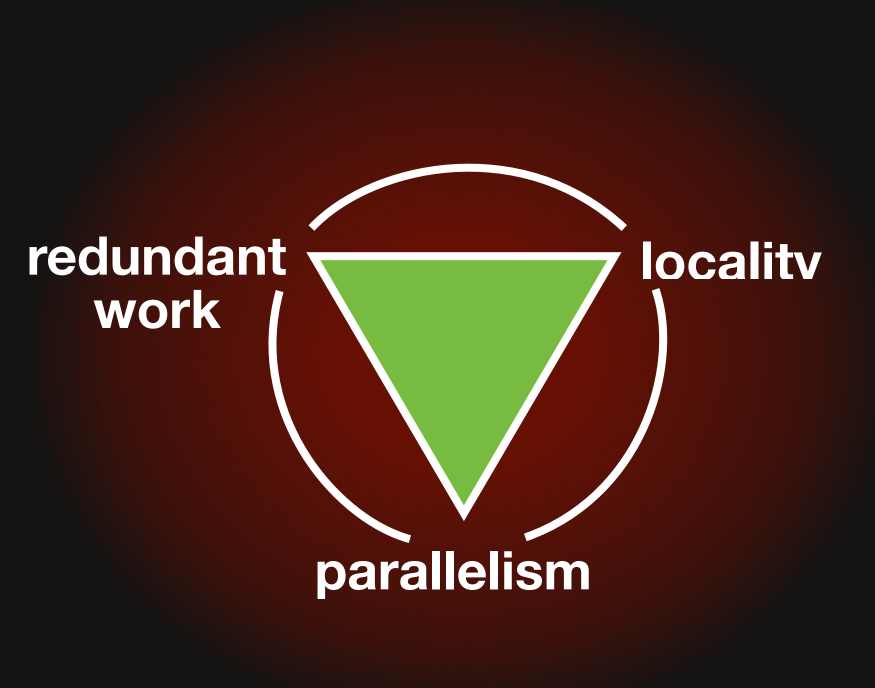
Simpler programs

Faster than hand-tuned code

Scalable across architectures

Summary

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



Schedule:

Order *within* stage

Interleaving *between* stages

→ Loop nest & storage allocations

Simpler programs

Faster than hand-tuned code

Scalable across architectures

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

Discussion

Why *reductions*, rather than loops?

What is *chunking*?

Why the four caller-callee relationships?

what do these *not* represent?

Discussion

How did we choose our example apps?

Why are schedules user-controlled rather than compiler-controlled?

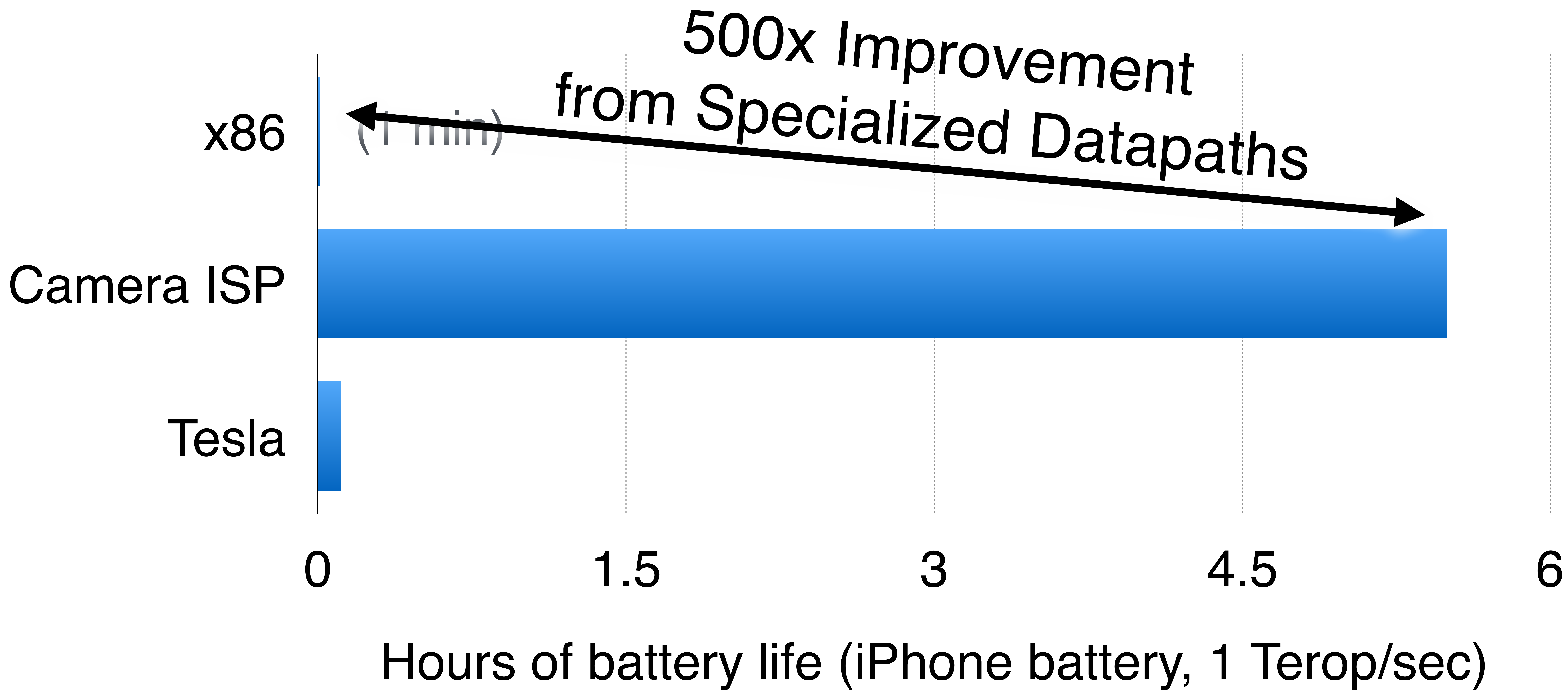
Are there other Halide-like systems, in other domains?

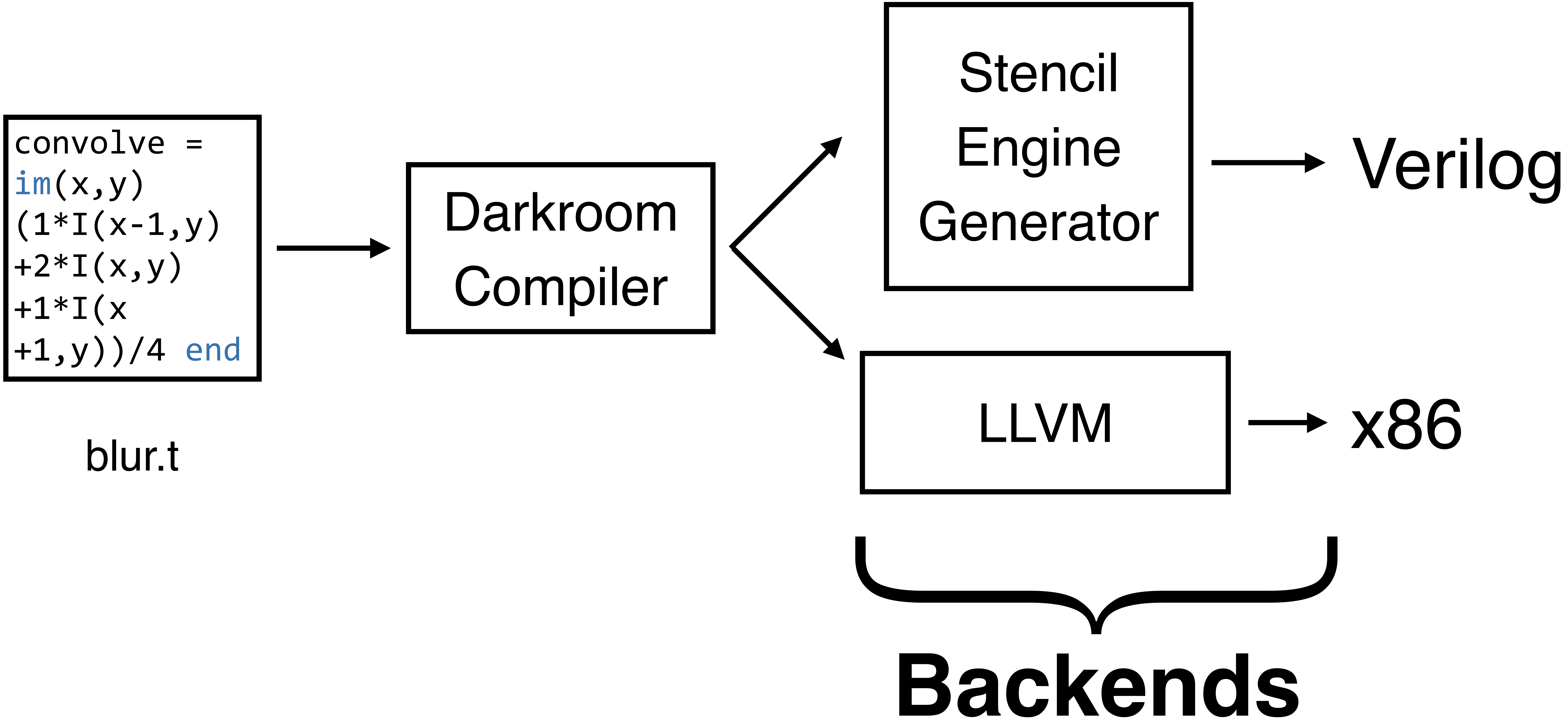
DARKROOM

Compiling High-Level Image Processing Code into Hardware Pipelines

James Hegarty
Jonathan Ragan-Kelley
Artem Vasilyev
John Brunhaver
Noy Cohen
Mark Horowitz
Zachary DeVito
Steven Bell
Pat Hanrahan

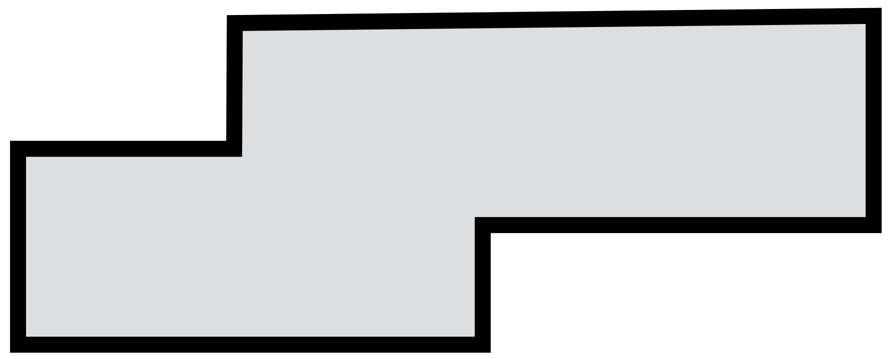
Programmable processors are too inefficient for real-time image processing!





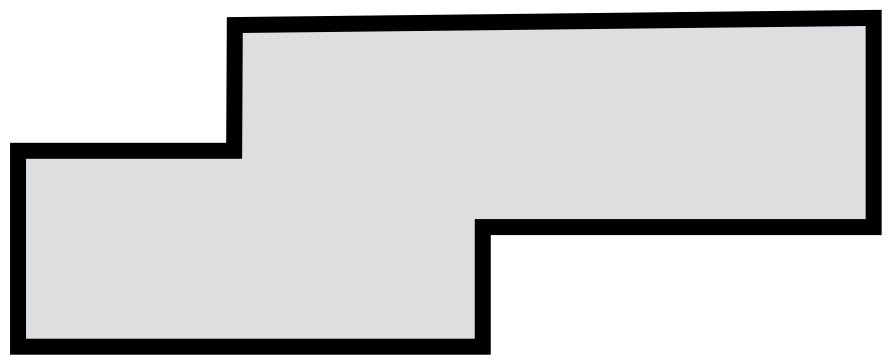
convolve = im(x,y) (1*I(x-1,y) +2*I(x,y) +1*I(x +1,y))/4 end
blur.t
Darkroom Compiler
Stencil Engine Generator
Verilog
LLVM
x86
Backends

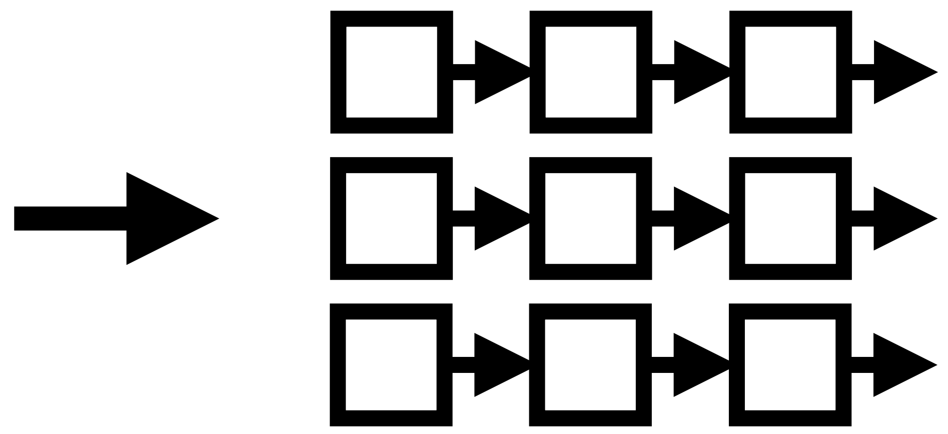
Stencil Engine Generator



1. Parameterized Linebuffer

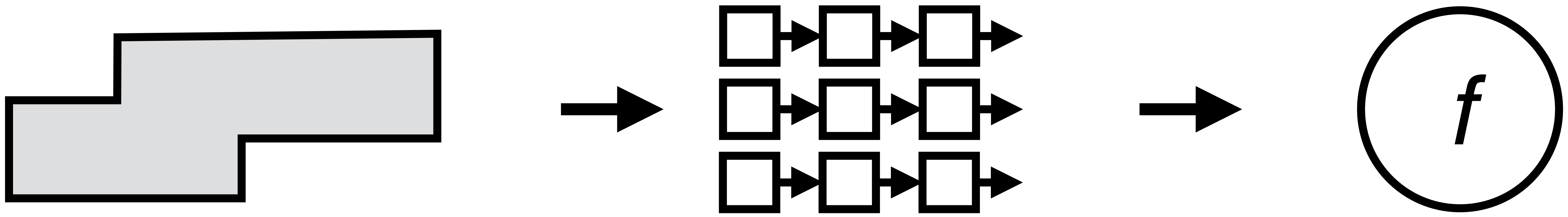
Stencil Engine Generator





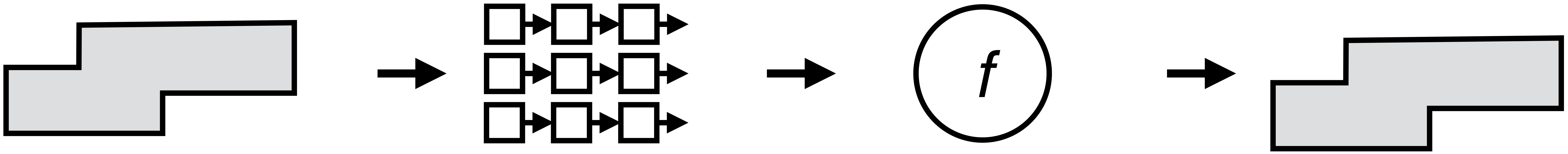
1. Parameterized Linebuffer
2. Stencil Shift Register

Stencil Engine Generator



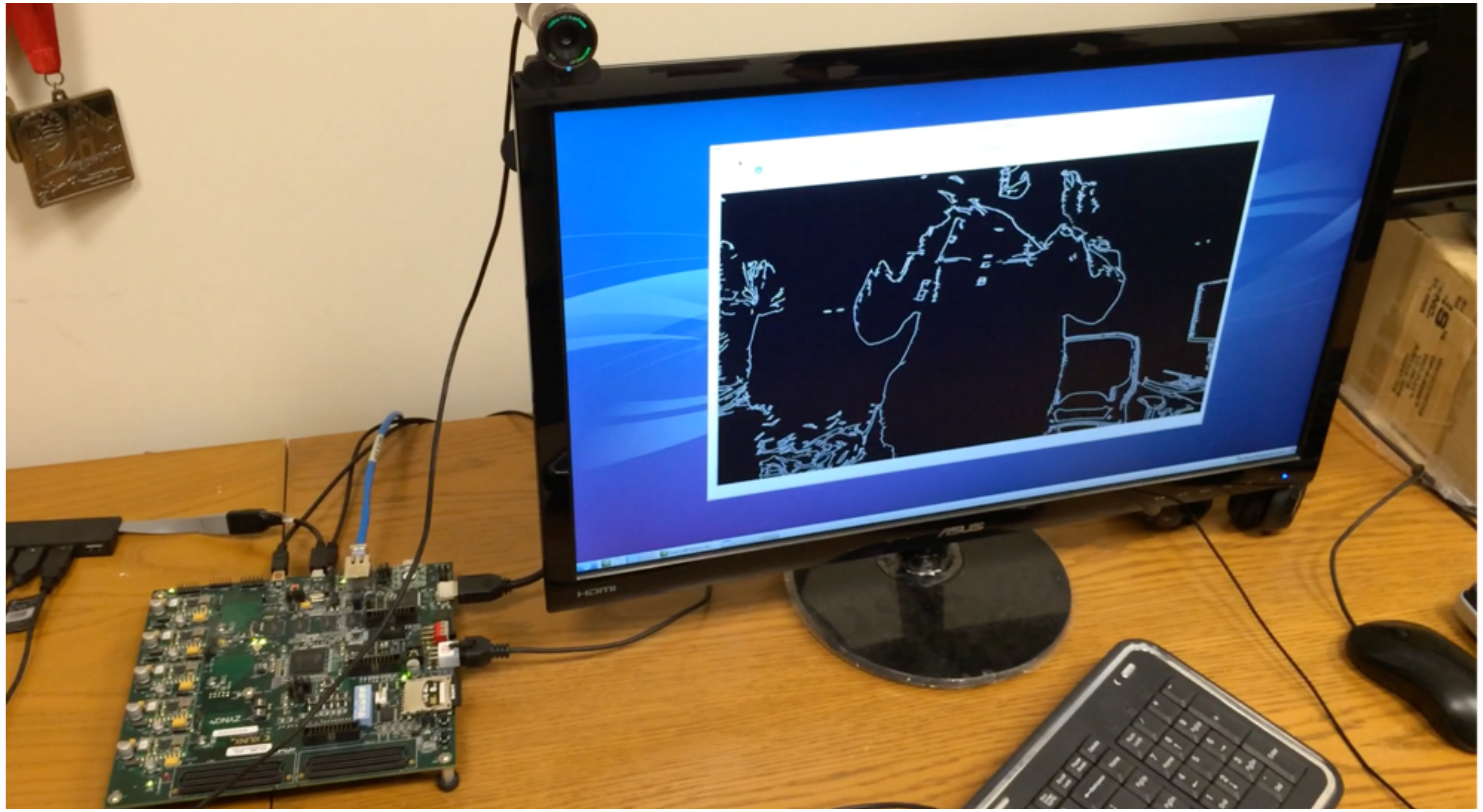
1. Parameterized Linebuffer
2. Stencil Shift Register
3. Synopsys Designware

Stencil Engine Generator



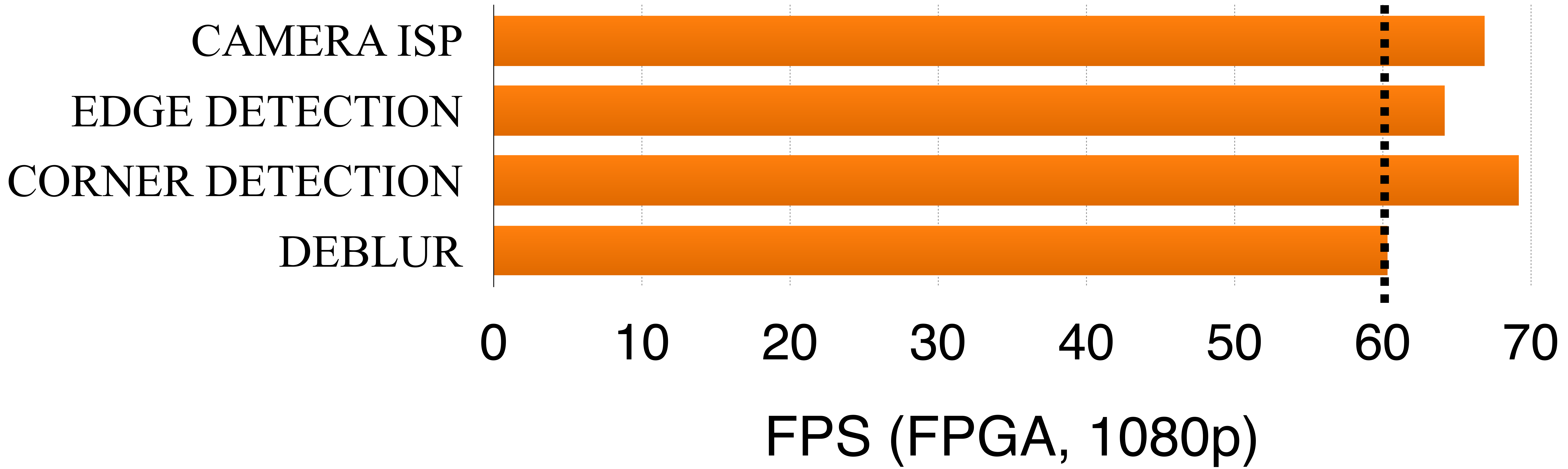
1. Parameterized Linebuffer
2. Stencil Shift Register
3. Synopsys Designware
4. Pipeline Generator

FPGA

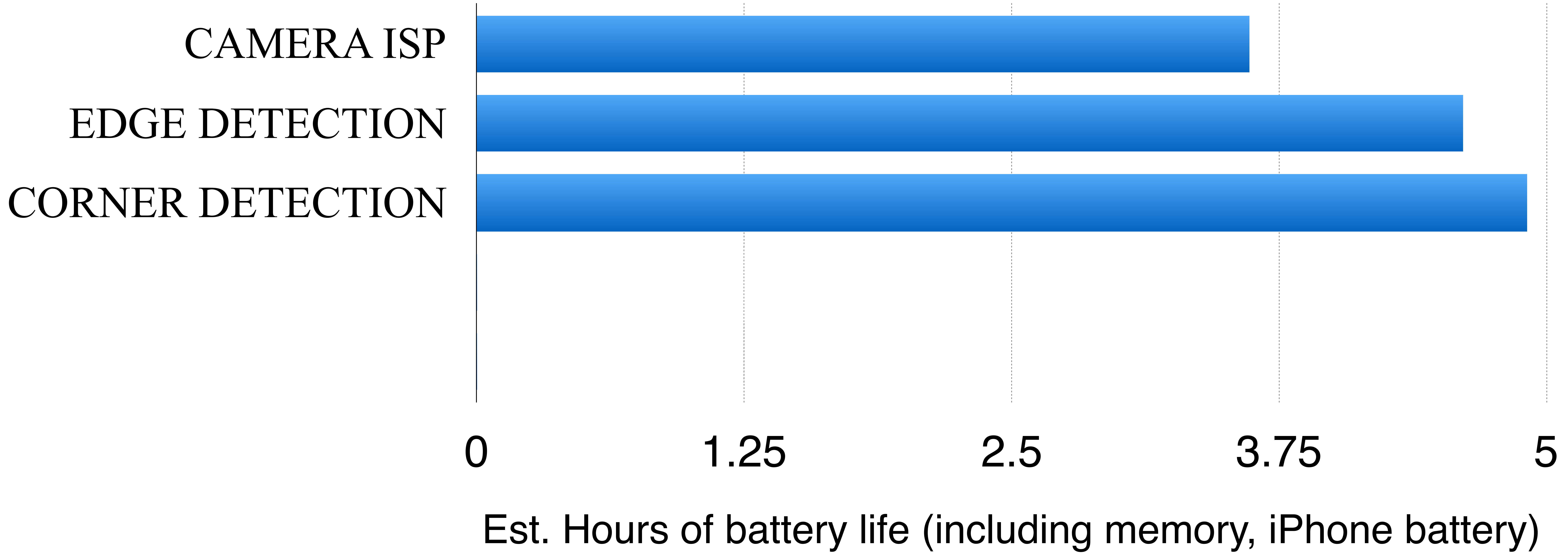


Xilinx Zynq XC7Z045

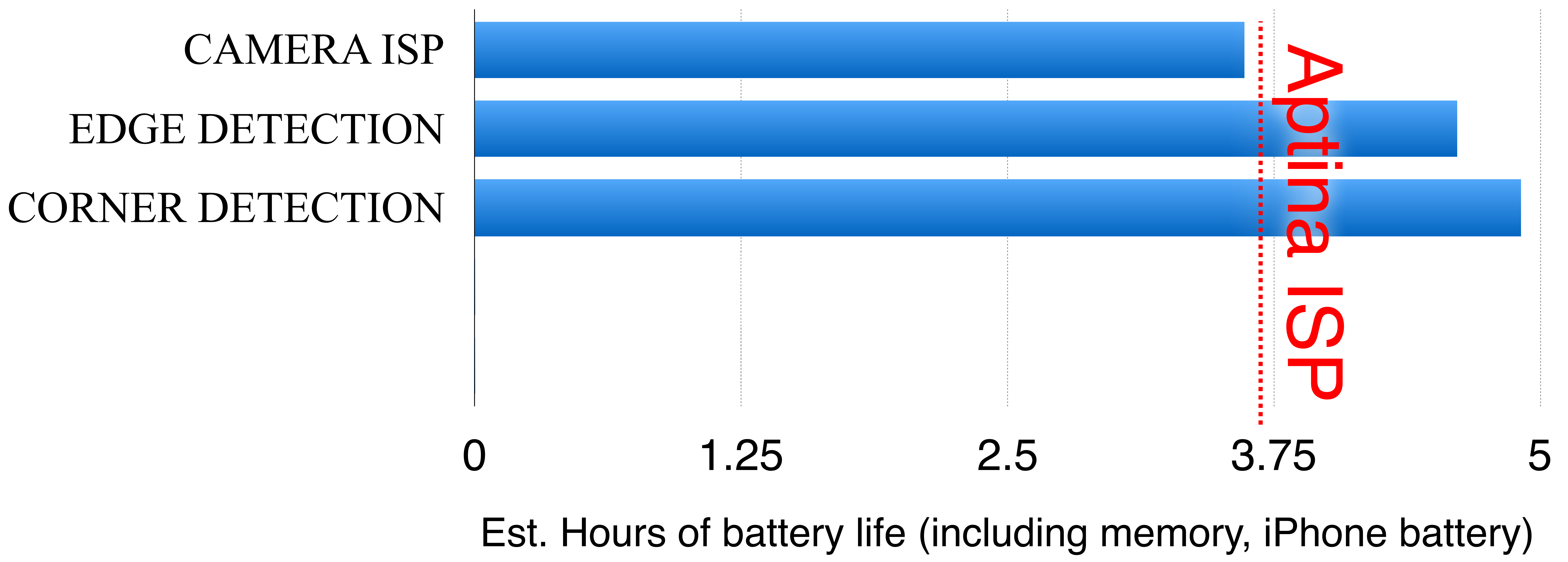
Darkroom pipelines deliver real-time performance



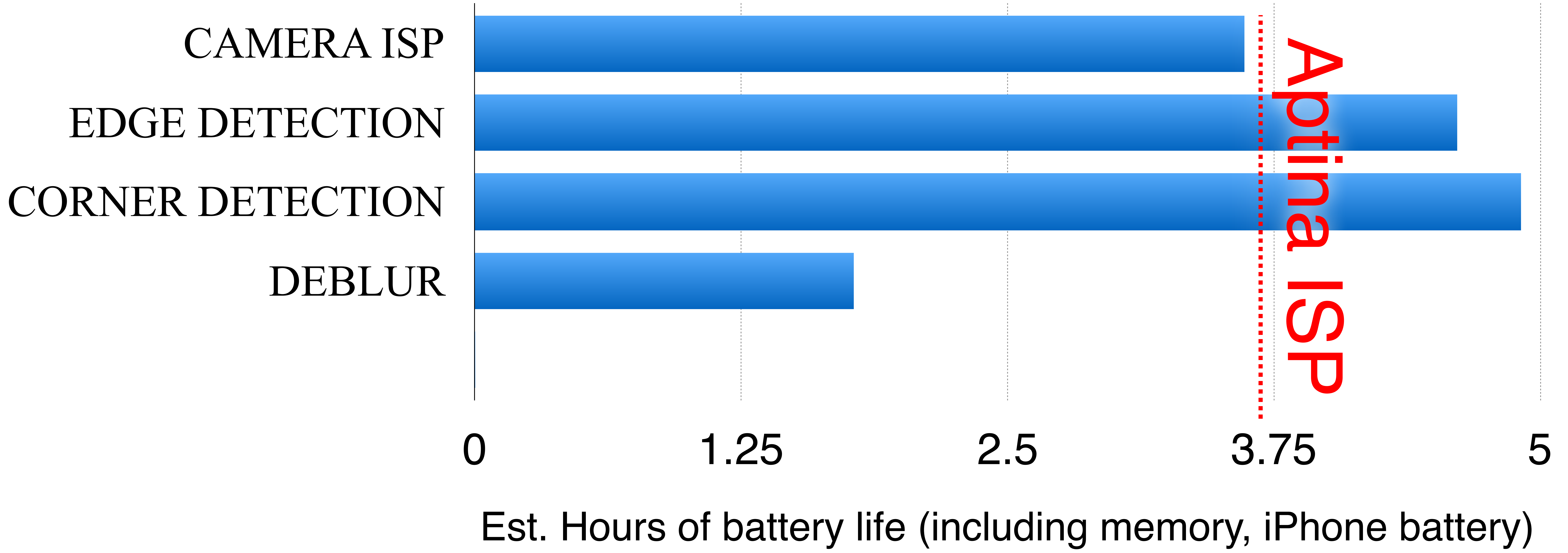
Darkroom pipelines are energy efficient



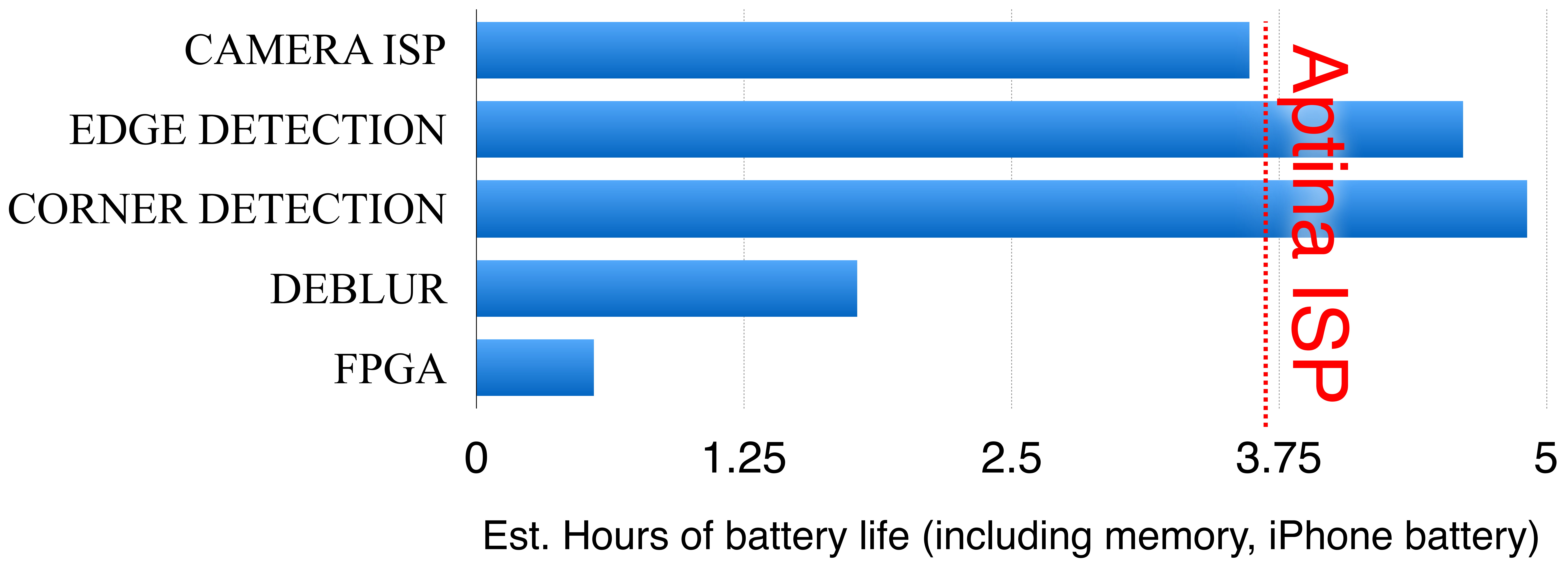
Darkroom pipelines are energy efficient



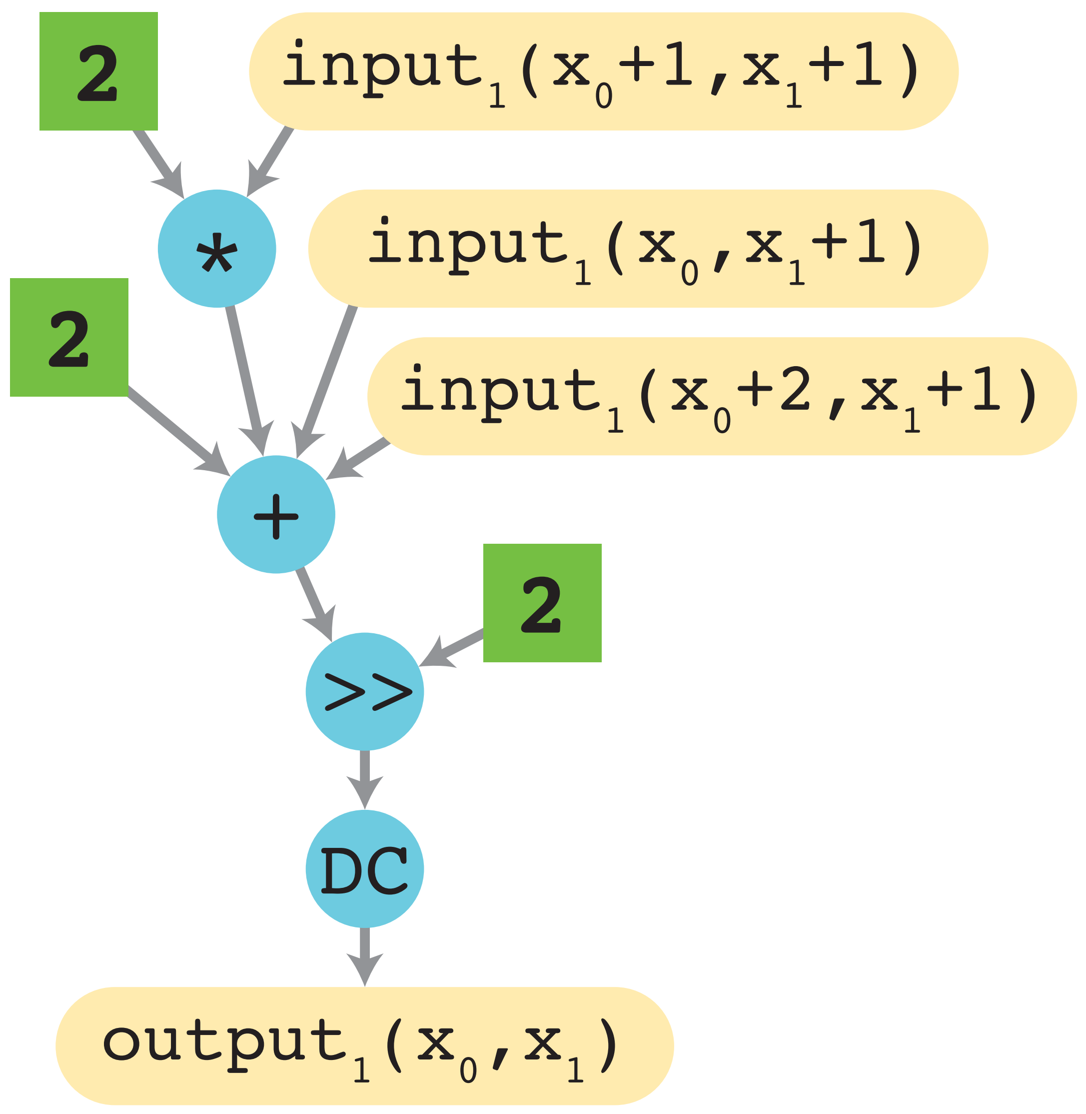
Darkroom pipelines are energy efficient



Darkroom pipelines are energy efficient



Helium: decompiling x86 binaries

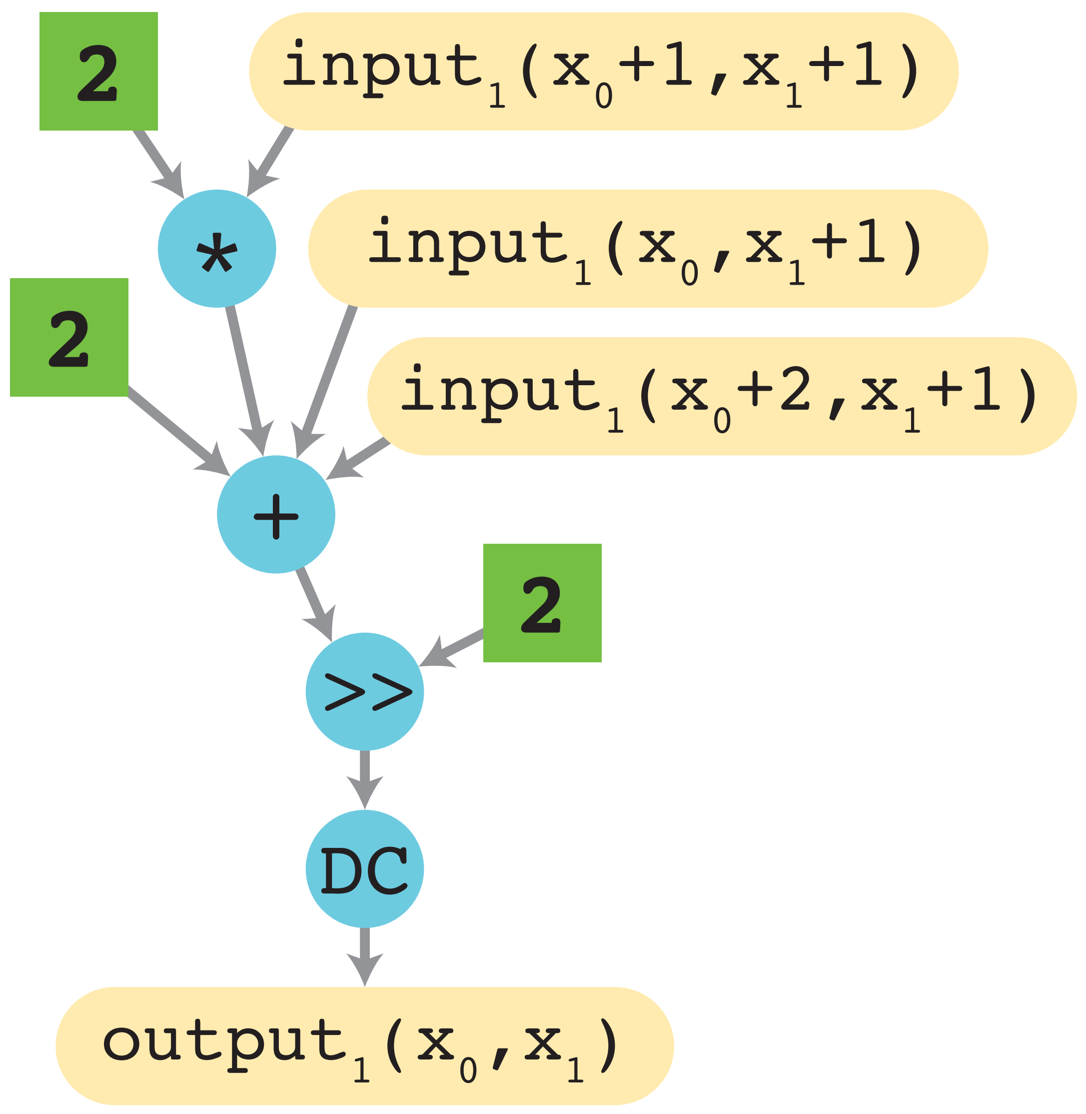


Symbolic tree

```
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;

int main(){
  Var x_0;
  Var x_1;
  ImageParam input_1(UInt(8),2);
  Func output_1;
  output_1(x_0,x_1) =
    cast<uint8_t>(((((2+
      (2*cast<uint32_t>(input_1(x_0+1,x_1+1))) +
         cast<uint32_t>(input_1(x_0,  x_1+1)) +
         cast<uint32_t>(input_1(x_0+2,x_1+1)))
     >> cast<uint32_t>(2))) & 255));
  vector<Argument> args;
  args.push_back(input_1);
  output_1.compile_to_file("halide_out_0",args);
  return 0;
}
```

Generated Halide DSL code

Helium: decompiling x86 binaries



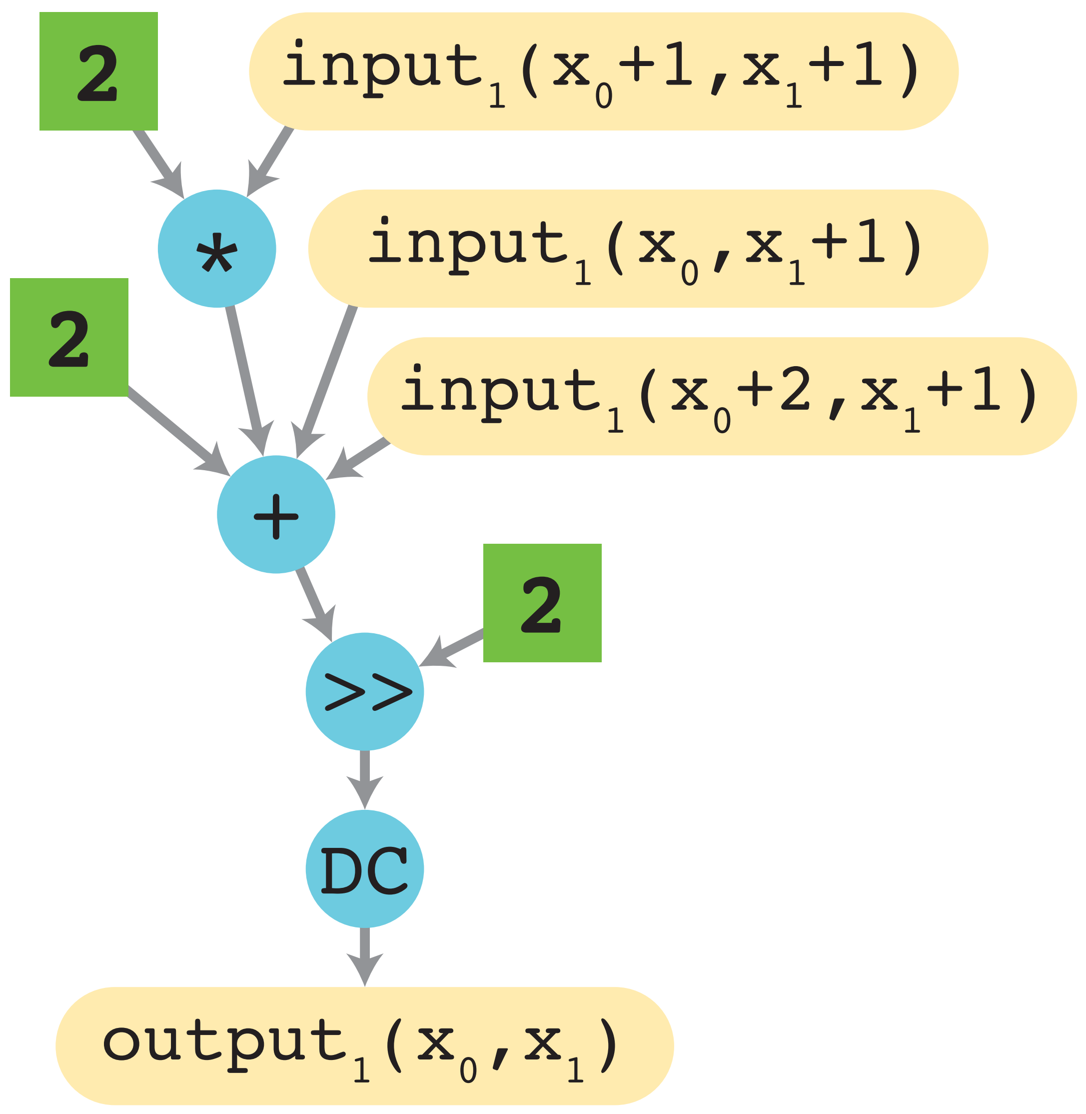
Symbolic tree

```
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;

int main(){
  Var x_0;
  Var x_1;
  ImageParam input_1(UInt(8),2);
  Func output_1;
  output_1(x_0,x_1) =
    cast<uint8_t>(((((2+
      (2*cast<uint32_t>(input_1(x_0+1,x_1+1))) +
        cast<uint32_t>(input_1(x_0,  x_1+1)) +
        cast<uint32_t>(input_1(x_0+2,x_1+1)))
     >> cast<uint32_t>(2))) & 255));
  vector<Argument> args;
  args.push_back(input_1);
  output_1.compile_to_file("halide_out_0",args);
  return 0;
}
```

Generated Halide DSL code

Helium: decompiling x86 binaries

x86
Executable

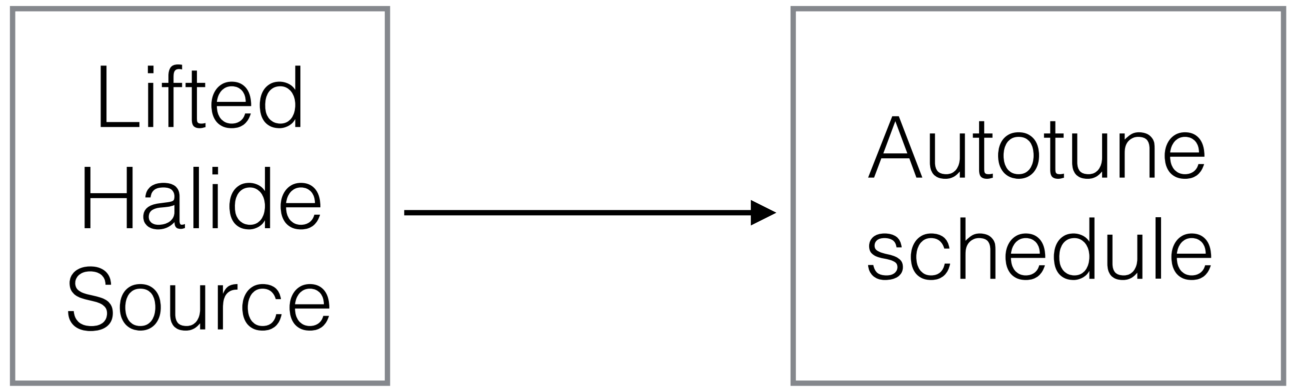


Symbolic tree

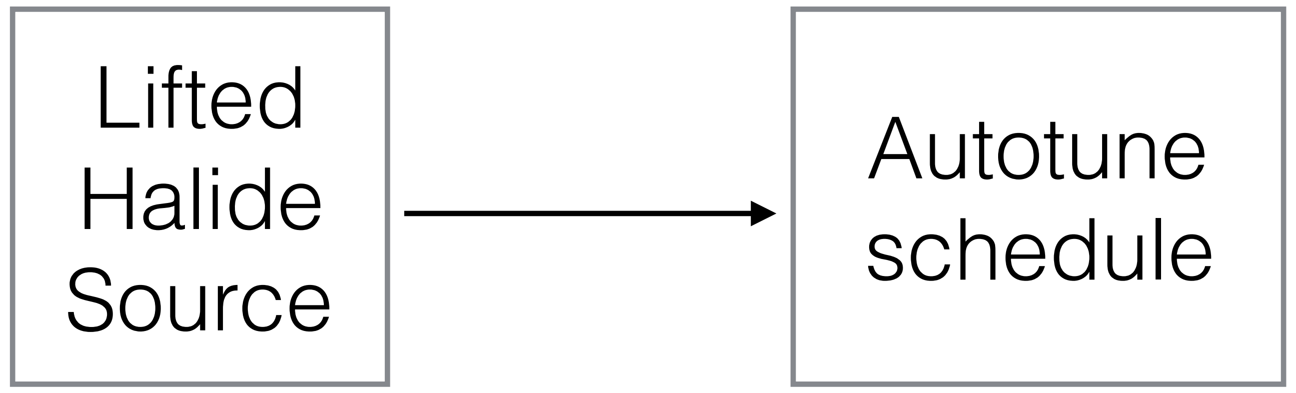
```
#include <Halide.h>
#include <vector>
using namespace std;
using namespace Halide;

int main(){
  Var x_0;
  Var x_1;
  ImageParam input_1(UInt(8),2);
  Func output_1;
  output_1(x_0,x_1) =
    cast<uint8_t>(((((2+
      (2*cast<uint32_t>(input_1(x_0+1,x_1+1))) +
         cast<uint32_t>(input_1(x_0,  x_1+1)) +
         cast<uint32_t>(input_1(x_0+2,x_1+1)))
     >> cast<uint32_t>(2))) & 255));
  vector<Argument> args;
  args.push_back(input_1);
  output_1.compile_to_file("halide_out_0",args);
  return 0;
}
```

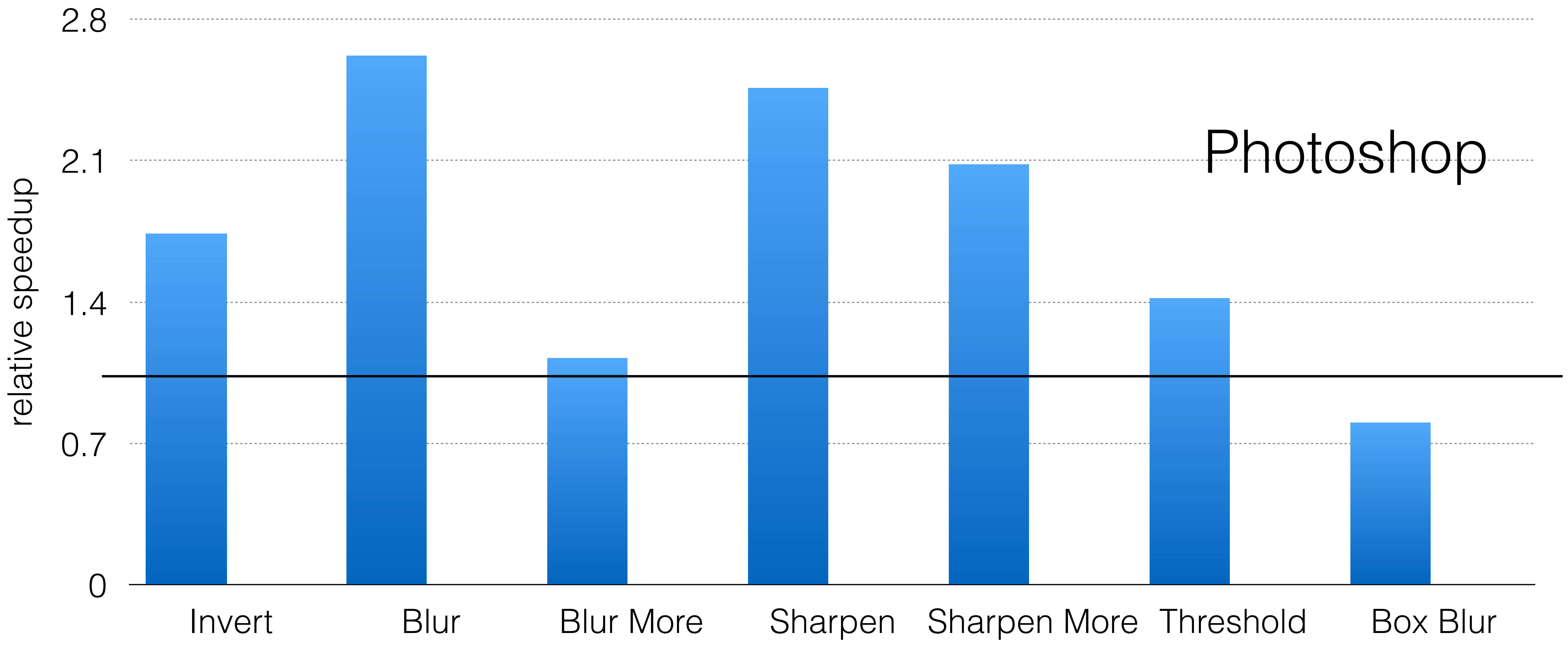
Generated Halide DSL code

Performance Results



Performance Results





Next time: final projects

Come to class with a potential project idea

1. Build a new DSL
2. Extend an existing DSL
3. Build or extend DSL-creation infrastructure

EXTRA

x86

	Halide autotuned	Expert tuned	Speedup	Halide source	Expert source	Factor shorter
Blur	11 ms	13 ms	1.2 ×	2 lines	35 lines	18 ×
Bilateral Grid	36 ms	158 ms	4.4 ×	34 lines	122 lines	4 ×
Camera pipeline	14 ms	39 ms	3.4 ×	123 lines	306 lines	2 ×
“Healing brush”	32 ms	54 ms	1.7 ×	21 lines	152 lines	7 ×

GPU

	Halide autotuned	Expert tuned	Speedup	Halide source	Expert source	Factor shorter
Bilateral Grid	8.1 ms	18 ms	2.3 ×	34 lines	370 lines	11 ×
“Healing brush”	9.1 ms	*54* ms*	*5.9** ×	21 lines	*152* lines*	*7** ×

x86

	Halide autotuned	Expert tuned	Speedup	Halide source	Expert source	Factor shorter
Blur	11 ms	13 ms	1.2 ×	2 lines	35 lines	18 ×
Bilateral Grid	36 ms	158 ms	4.4 ×	34 lines	122 lines	4 ×
Camera pipeline	14 ms	39 ms	3.4 ×	123 lines	306 lines	2 ×
“Healing brush”	32 ms	54 ms	1.7 ×	21 lines	152 lines	7 ×

GPU

	Halide autotuned	Expert tuned	Speedup	Halide source	Expert source	Factor shorter
Bilateral Grid	8.1 ms	18 ms	2.3 ×	34 lines	370 lines	11 ×
“Healing brush”	9.1 ms	*54* ms*	*5.9** ×	21 lines	*152* lines*	*7** ×

x86

	Halide autotuned	Expert tuned	Speedup	Halide source	Expert source	Factor shorter
Blur	11 ms	13 ms	1.2 ×	2 lines	35 lines	18 ×
Bilateral Grid	36 ms	158 ms	4.4 ×	34 lines	122 lines	4 ×
Camera pipeline	14 ms	39 ms	3.4 ×	123 lines	306 lines	2 ×
“Healing brush”	32 ms	54 ms	1.7 ×	21 lines	152 lines	7 ×

GPU

	Halide autotuned	Expert tuned	Speedup	Halide source	Expert source	Factor shorter
Bilateral Grid	8.1 ms	18 ms	2.3 ×	34 lines	370 lines	11 ×
“Healing brush”	9.1 ms	*54* ms*	*5.9** ×	21 lines	*152* lines*	*7** ×

Autotuning time: **2 hrs to 2 days**

(single node) ***85% within < 24 hrs***

High-efficiency image processing

Automatic inference of good *schedules*
beyond brute force autotuning

Synthesis of specialized hardware

“OpenGL” and “programmable GPU” for image processing pipelines
current ISP pipelines are hard-wired, inflexible

High-performance computer vision
programming model, language, compiler support

Detection & retrieval

e.g. HOG + LDA, Viola-Jones

Markov Random Fields

Photosynth

Automatic panoramas as a starting proxy

Computational fabrication software systems

Scalable synthesis pipeline for huge data size

architecture

programming model

compiler

New representations for authoring, output

Authoring tools

beyond CAD for classical mass production

Compilers & DSLs

Optimization as data-driven search

Composition of DSLs

Domain-specific programming tools

Richer authoring, debugging, and tuning

Ongoing work

Synthesizing hardware imaging pipelines from a Halide-like description

with James Hegarty, John Brunhaver, Pat Hanrahan, Mark Horowitz

Halide schedule visualization, debugging

with Jovana Knezevic

Producer-consumer parallelism, schedule-controlled memory layouts

with Nick Chornay

Ongoing work

Static schedule inference based on the task dependence graph

Irregular data in Halide

Level-of-detail for imaging pipelines

Scaling the OpenFab 3D printing pipeline

Stable Fluids [Stam 1999]

~200 stages

Complex dependence

Iterative linear solvers

Multi-phase computation

Programmable Graphics Pipelines

Visual sensor networks