Safe & Productive High-Performance Programming with **User-Schedulable Languages**

Jonathan Ragan-Kelley / MIT CSAIL

Safe & Productive High-Performance Programming with **User-Schedulable Languages**

Jonathan Ragan-Kelley / MIT CSAIL

High-performance programming **requires low-level control**

~~Therefore it must be **unsafe & unproductive**~~

This talk

Halide: user scheduling for safe, productive high performance

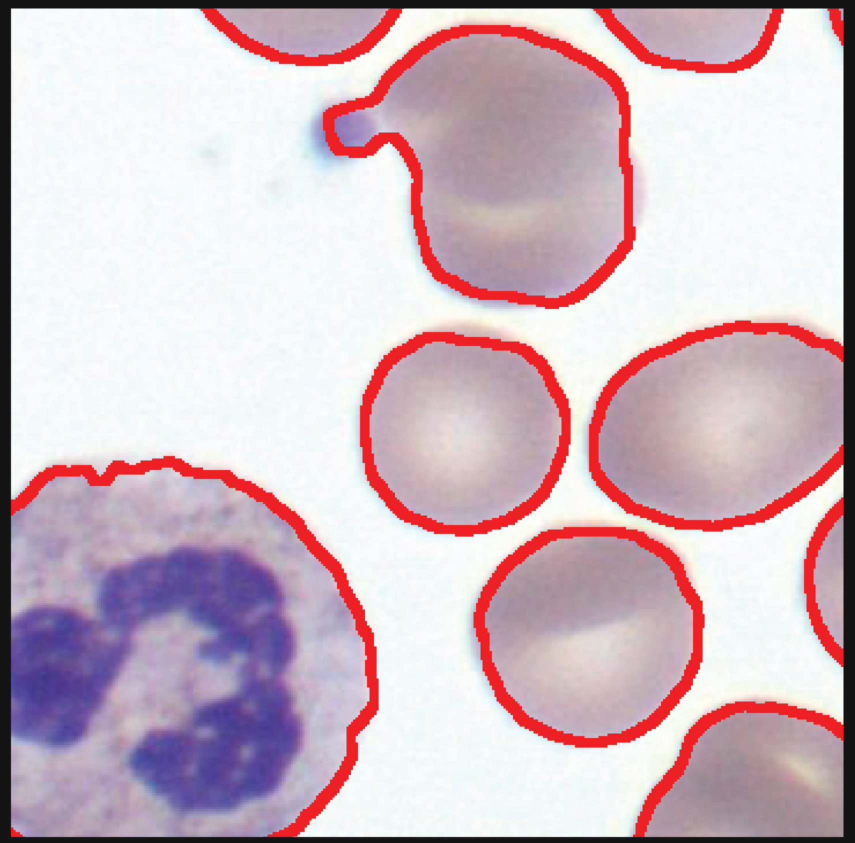
User-schedulable languages **2.0**

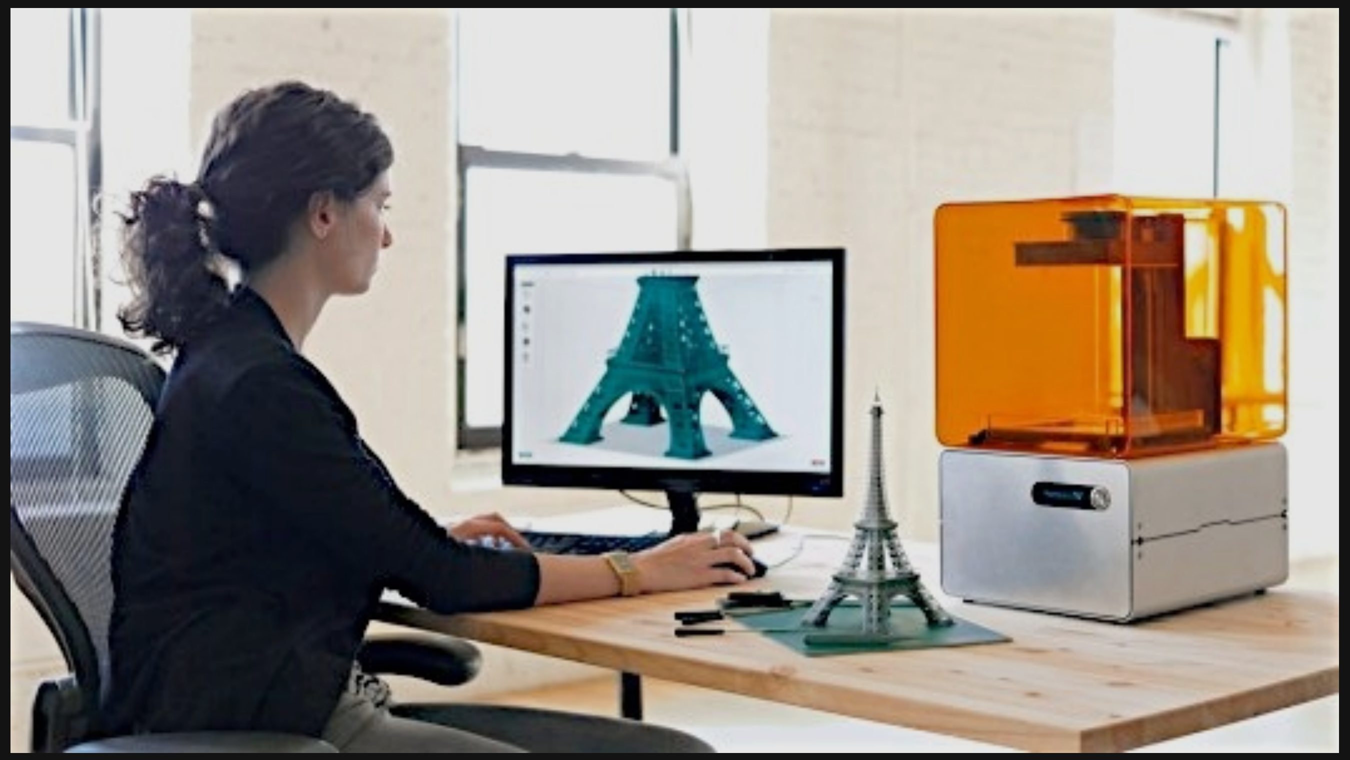
Performance

What is it?

Where does it come from?

Why do we **care?**




Visual computing and spatiotemporal data are **everywhere**




Visual computing demands **orders of magnitude more computation**

Rendering:
insatiable demand for computation

Modern game

2 Mpixels

10 Mpolys

15 ms/frame

15 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 2016]

1.5 Terapixels/sec

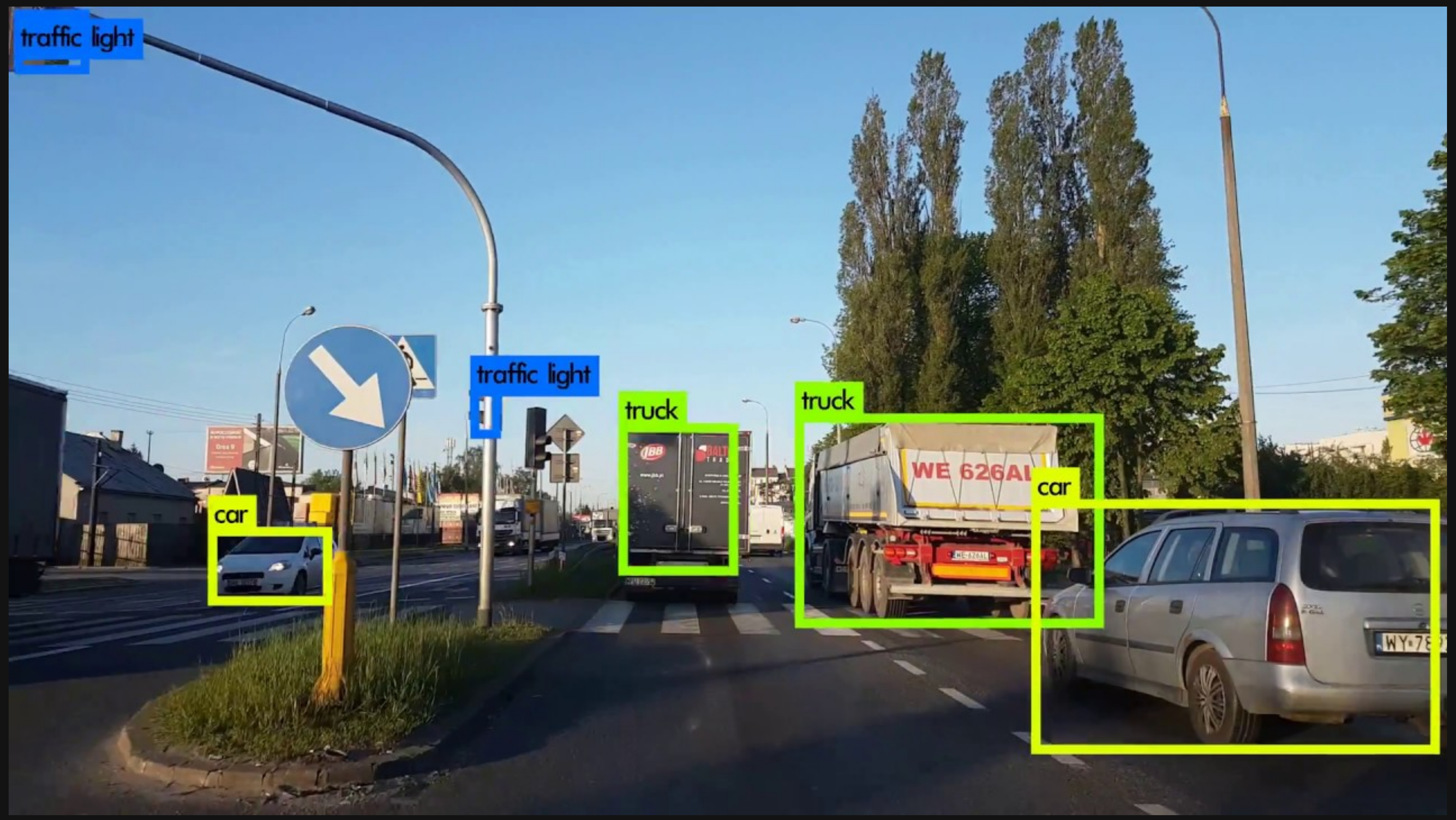
250 M surveillance cameras,
2.5 B cell phone cameras, …




Visual data analysis is expensive

One object detection neural net:

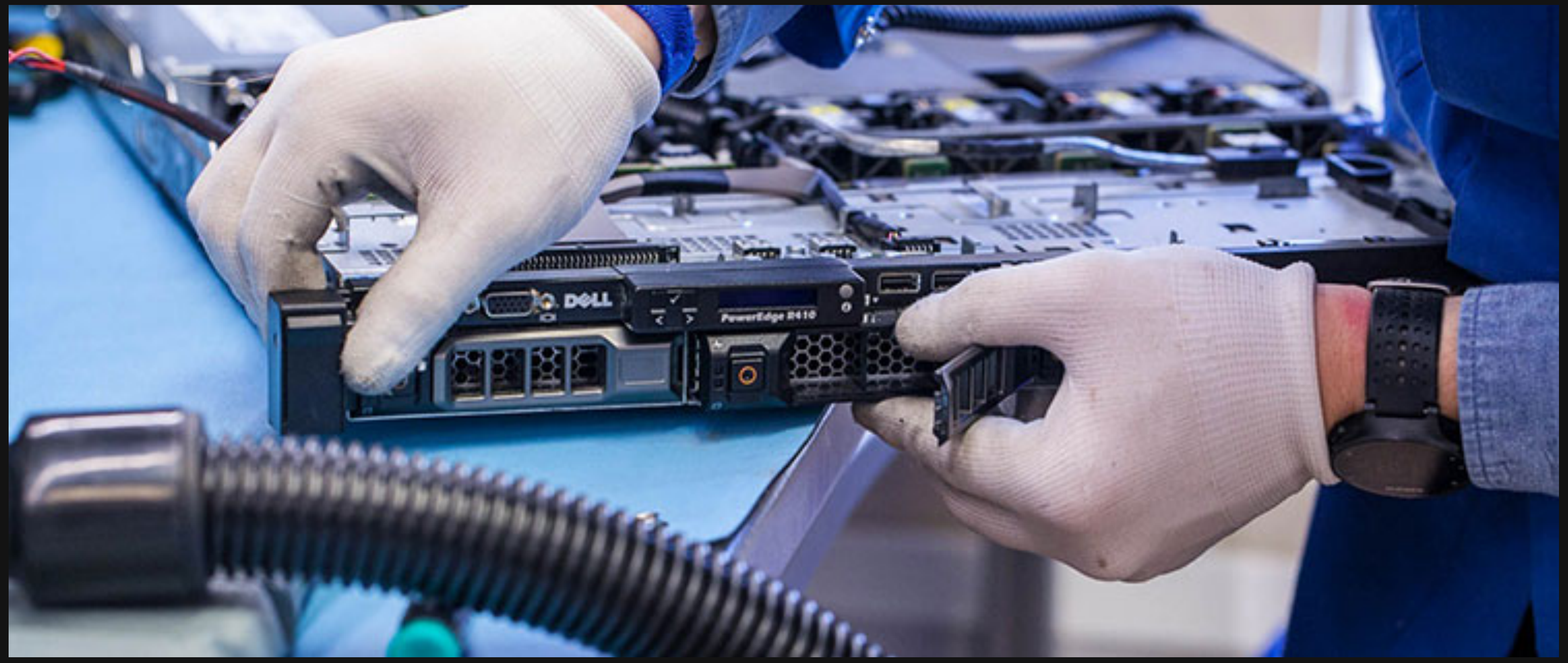
250 Watt GPU → 0.25 megapixels at video rate



(yolo-v3 on Tesla V100)

Programmer productivity has exploded

ORACLE



1990s

python

React

Firebase

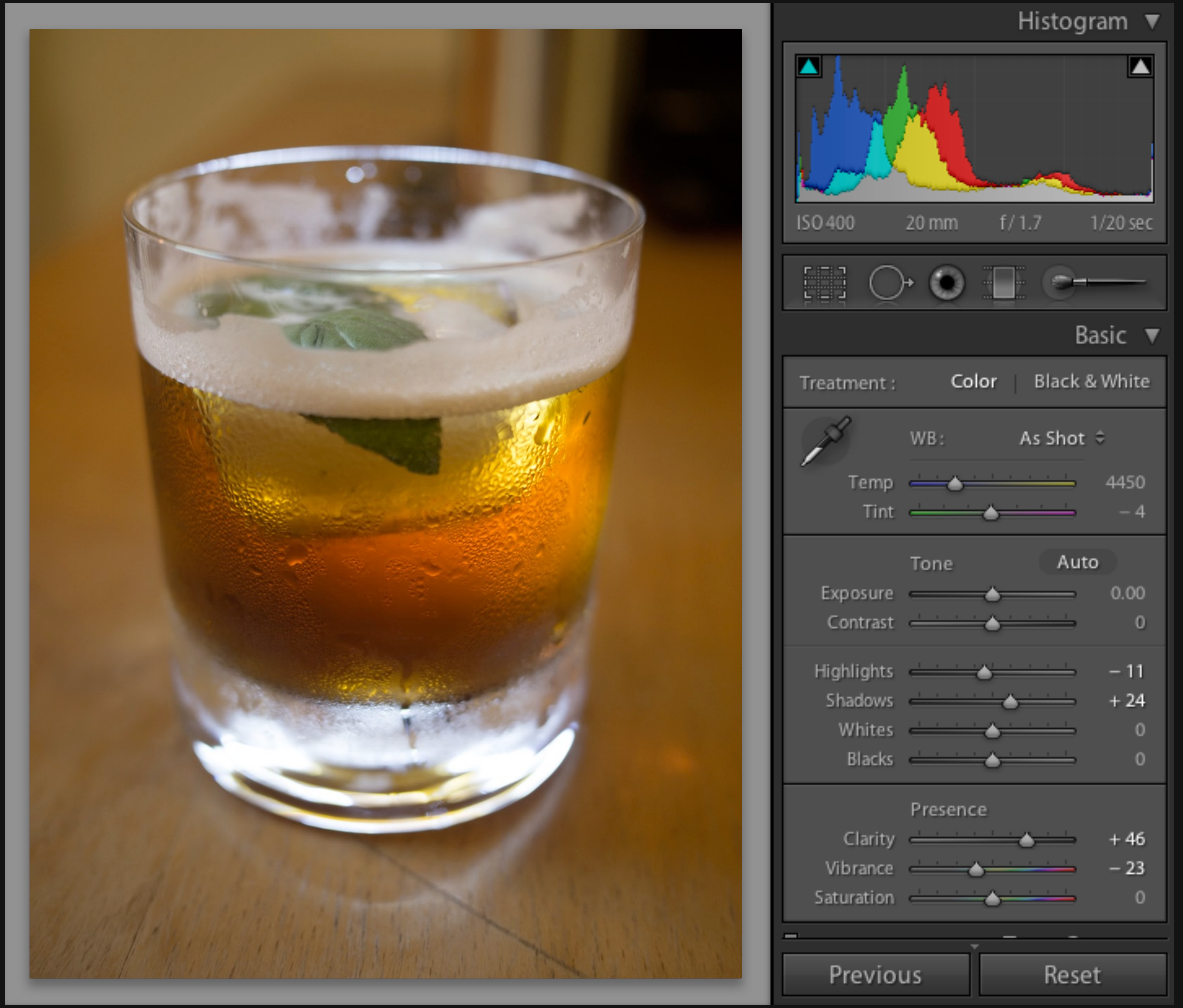
amazon web services

2010s

Building high-performance systems is harder than ever

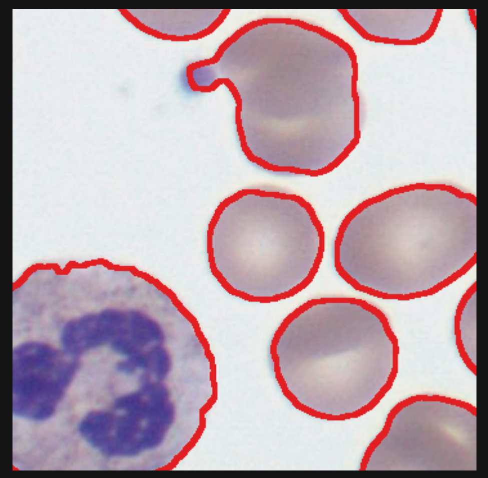
Reference:
200 lines C++

Adobe: 1500 lines
3 months of work
10x faster



My group's research:

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



Reorganize computations & data

Capture & control **dependencies**

Define & exploit **domain-specific structure**

applications

algorithms

data structures

languages

compilers

hardware

How can we increase performance and efficiency?

Parallelism

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

Locality

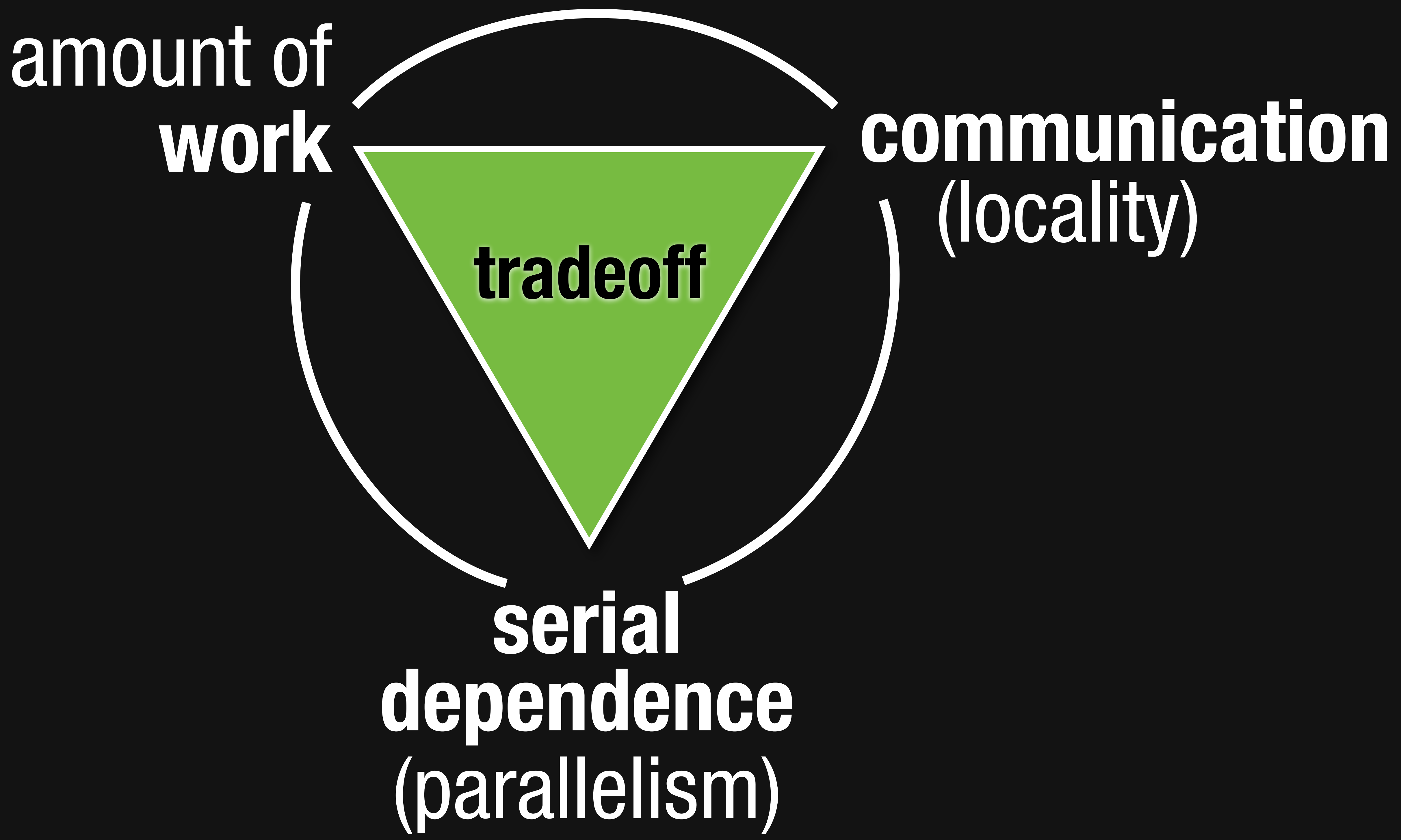
Data should move as little as possible.

Communication dominates computation
in both energy and time

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

data from John Brunhaver, Bill Dally, Mark Horowitz

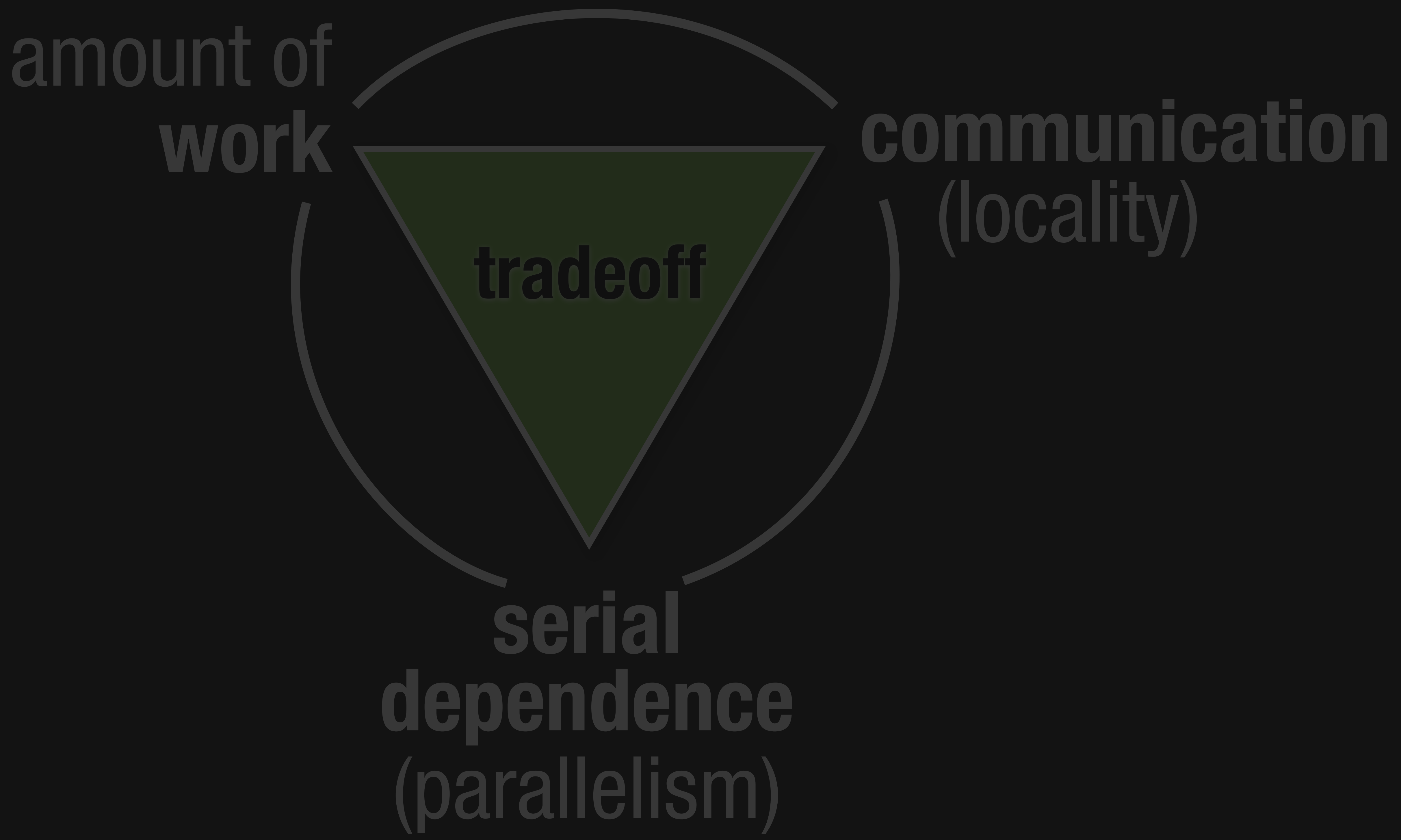
Message #1: Performance requires complex tradeoffs



Where does performance come from?

Program

Hardware



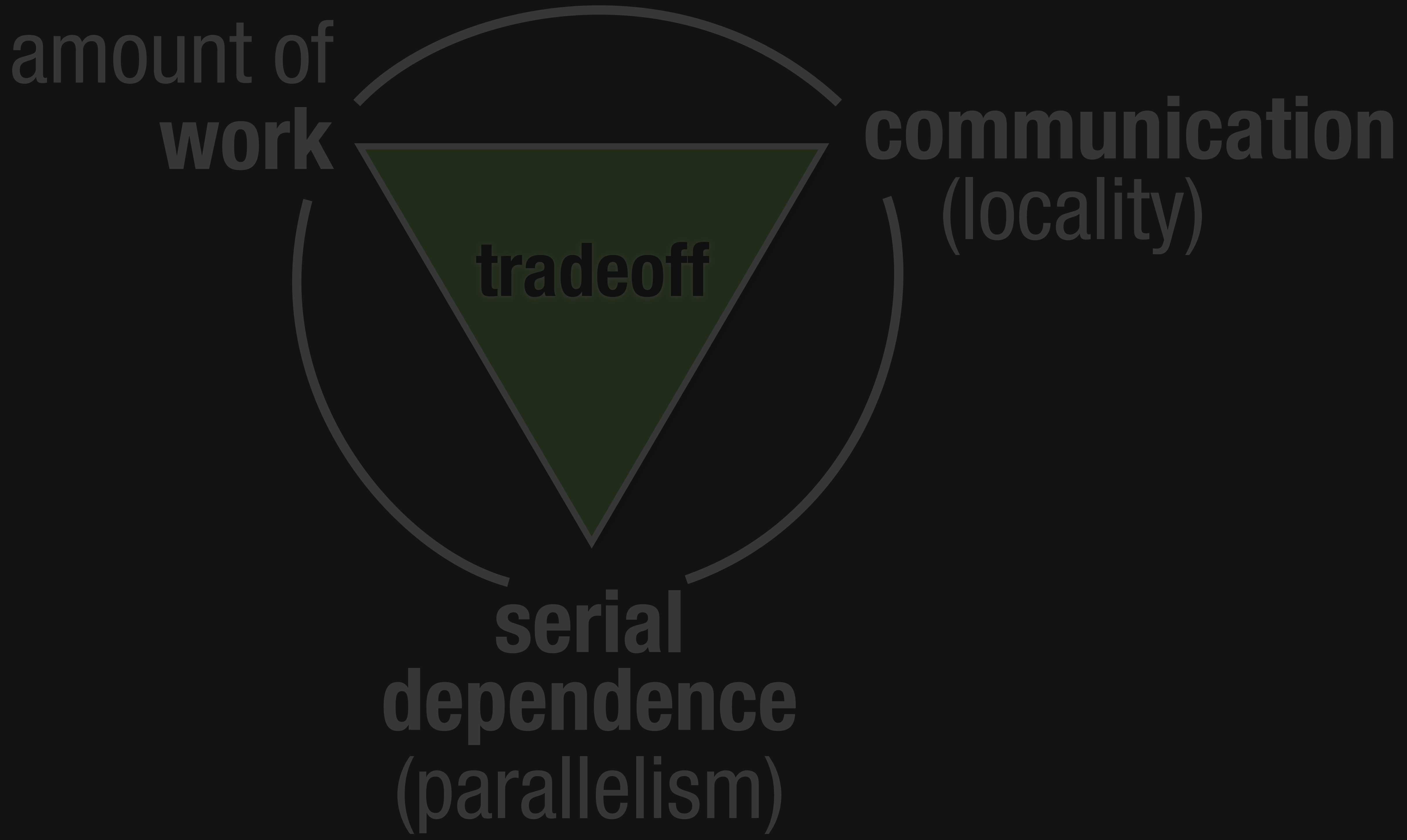
Message #2: organization of computation
is a first-class issue

Program:

Algorithm

Organization of computation

Hardware

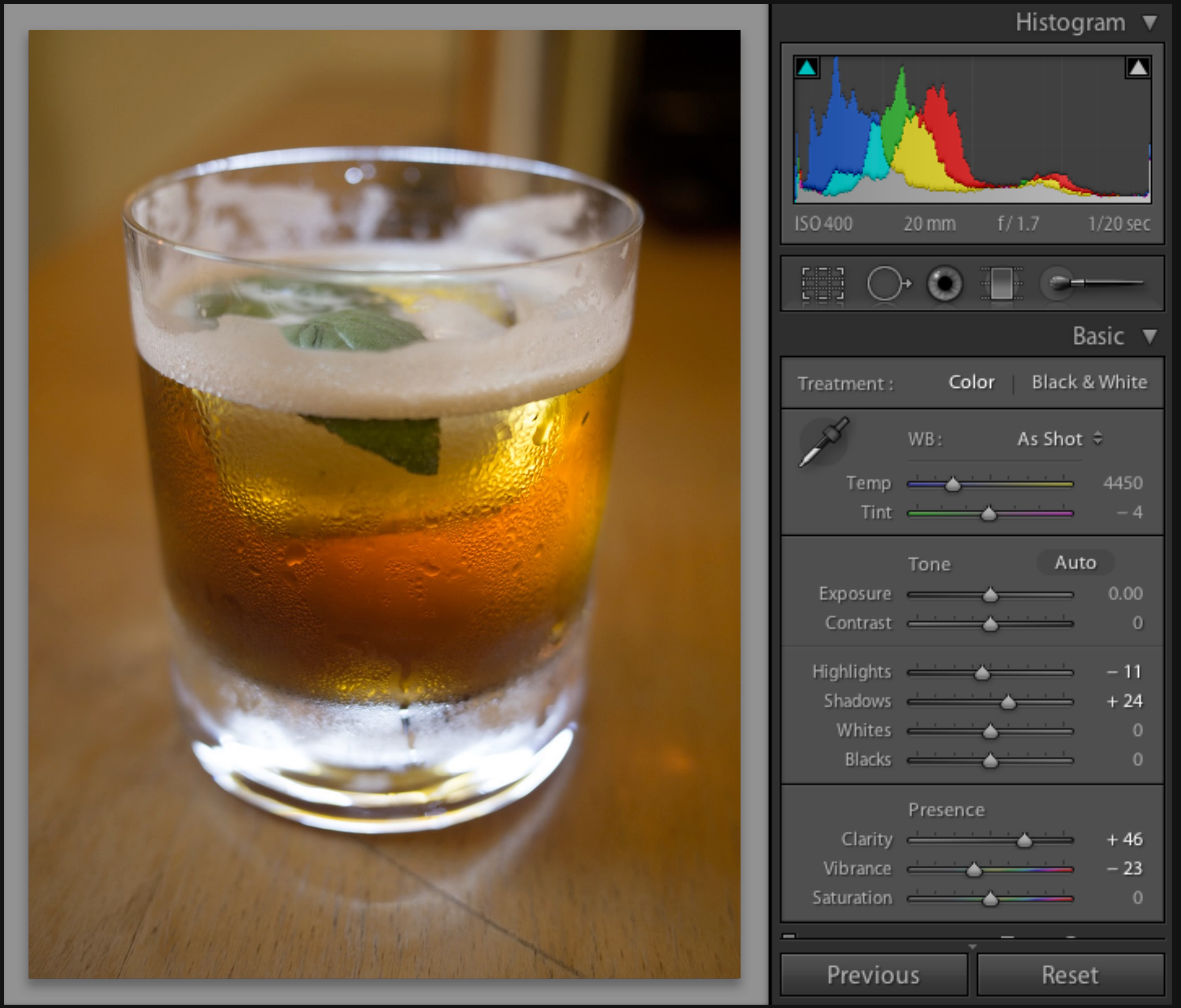


Reorganizing computation is painful

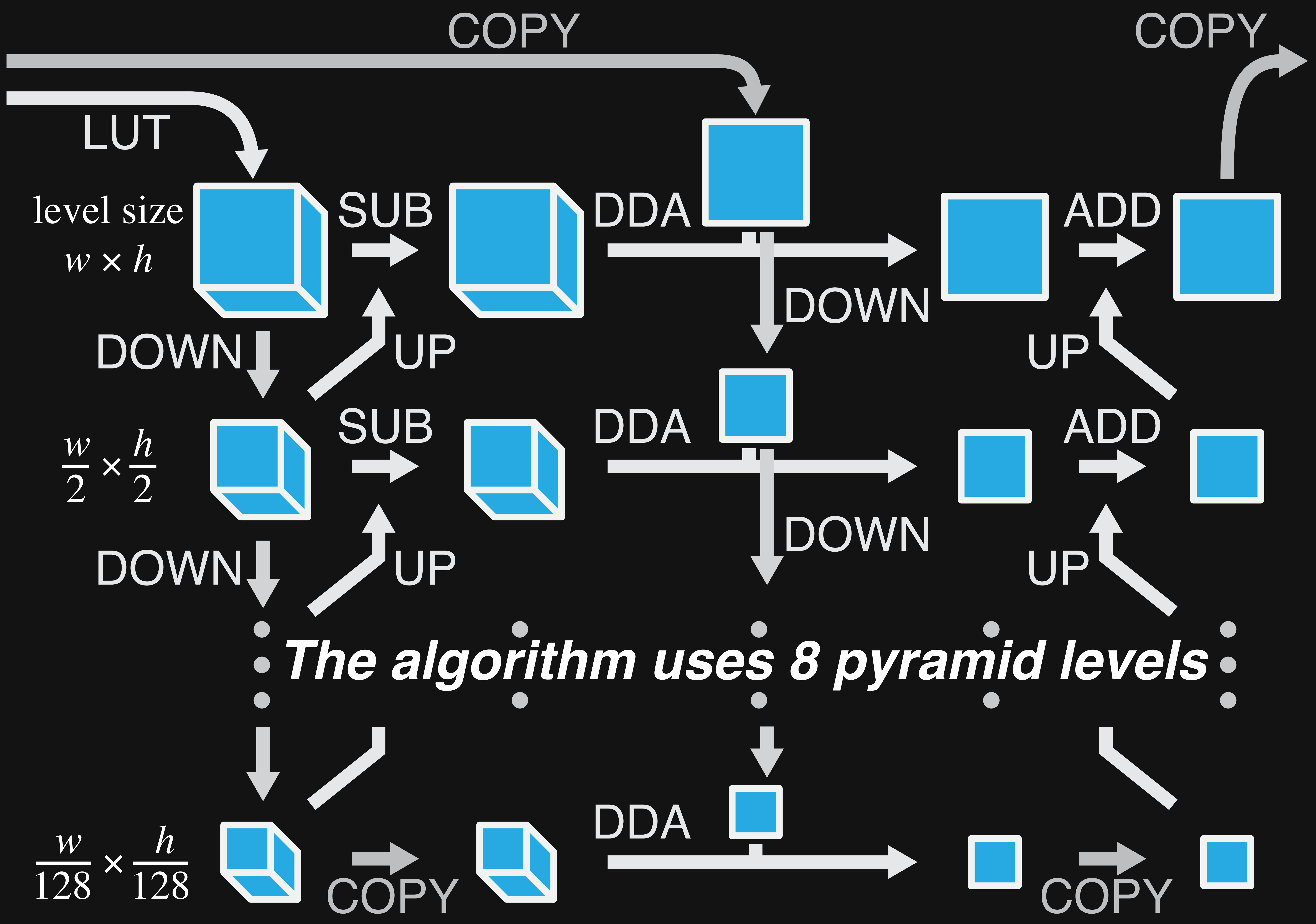
Reference:
300 lines C++

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

**Same algorithm,
Different organization**



Global reorganization breaks modularity



Halide

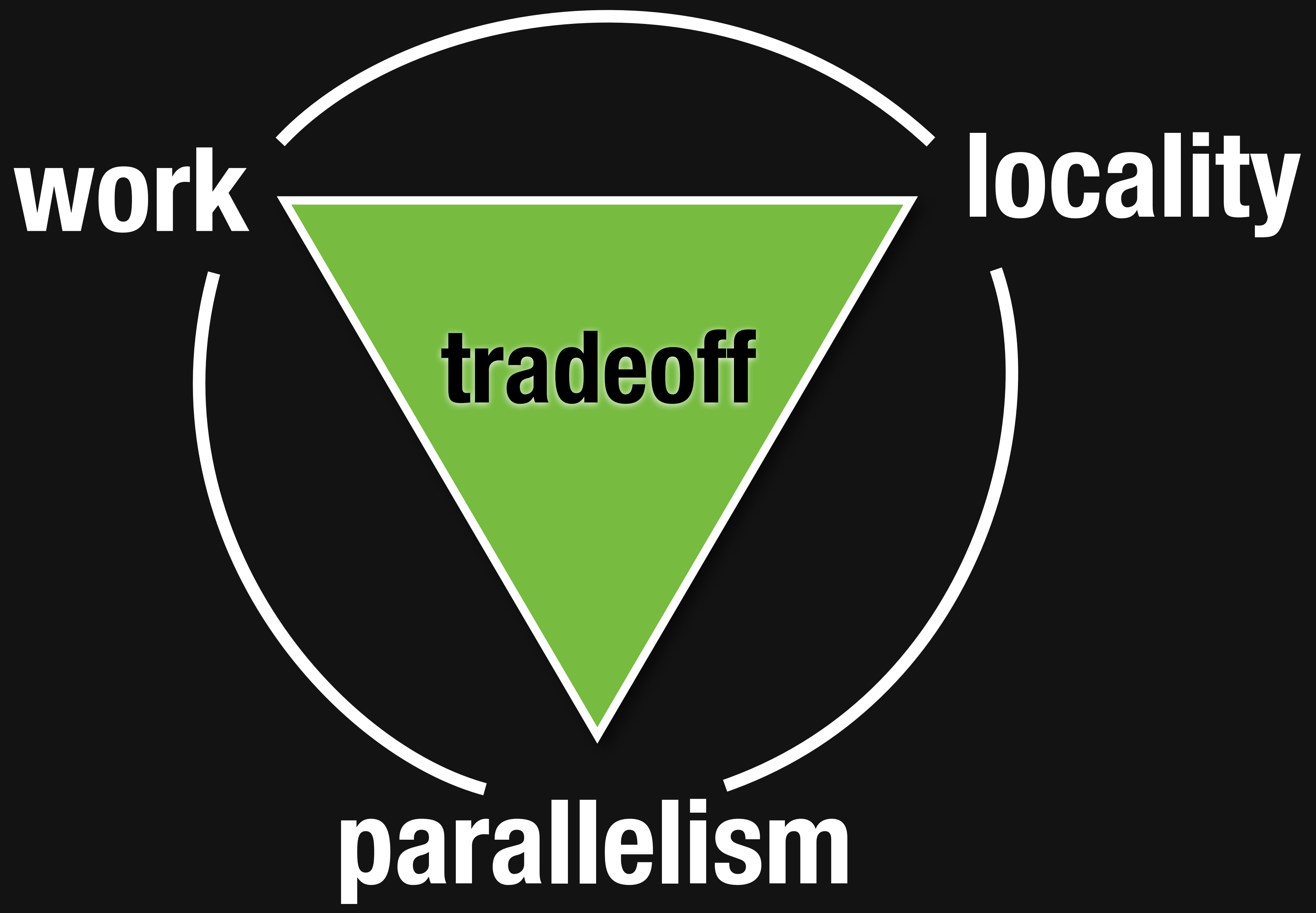
a language and compiler with decoupled organization

[SIGGRAPH 2012,
PLDI 2013,
SIGGRAPH 2016,
CACM 2017,
SIGGRAPH 2018,
SIGGRAPH 2019,
OOPSLA 2021…]

Algorithm

Organization of computation

Hardware



languages

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

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;
```

Hand-optimized C++

8 → 0.25 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);
}}}}}
```

30x faster
(dual 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;
}
```

unreadable

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

(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:
cuDNN, BLAS, MKL, OpenCV…
optimized kernels compose into inefficient pipelines (no fusion)

Halide's answer:
decouple algorithm from schedule

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

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;
```

Domain scope
of the programming model

All computation is over **regular grids** (“tensors”)

not Turing complete

Only **feed-forward pipelines**

Iterative computations are a (partial) escape hatch

Iteration must have **bounded depth**

Dependence must be inferable

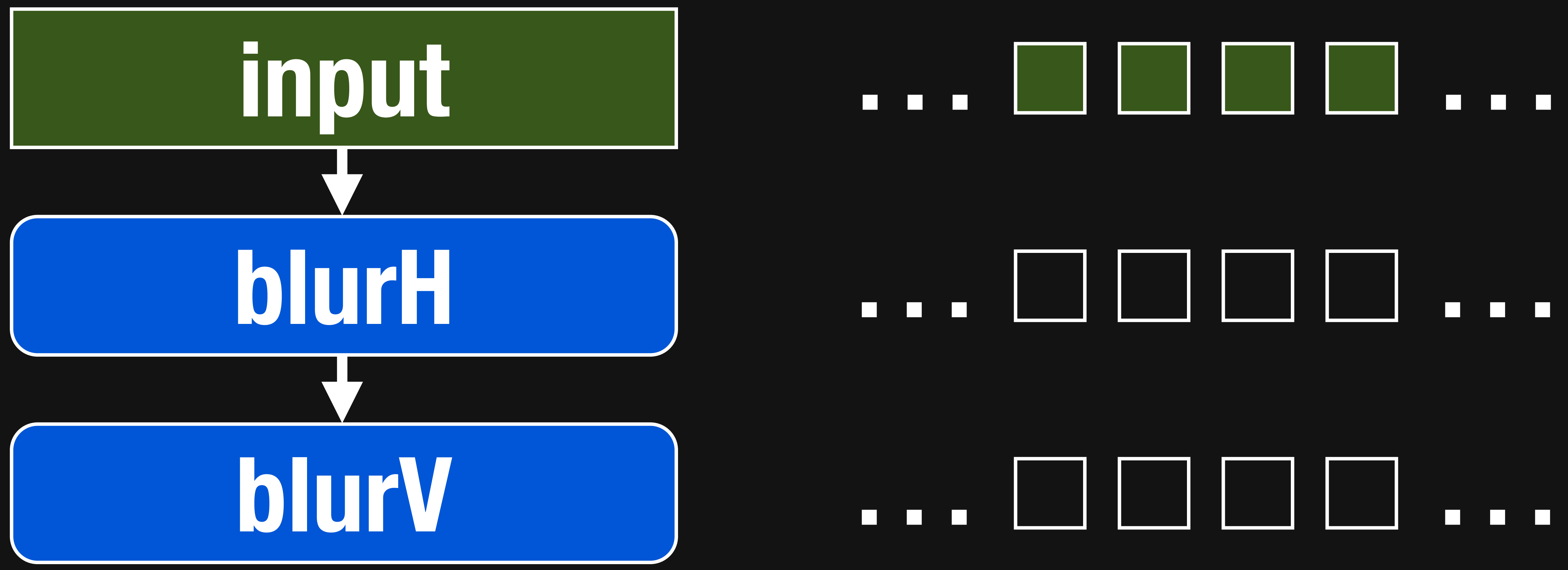
User-defined clamping can impose tight bounds, when needed

Long, heterogeneous pipelines

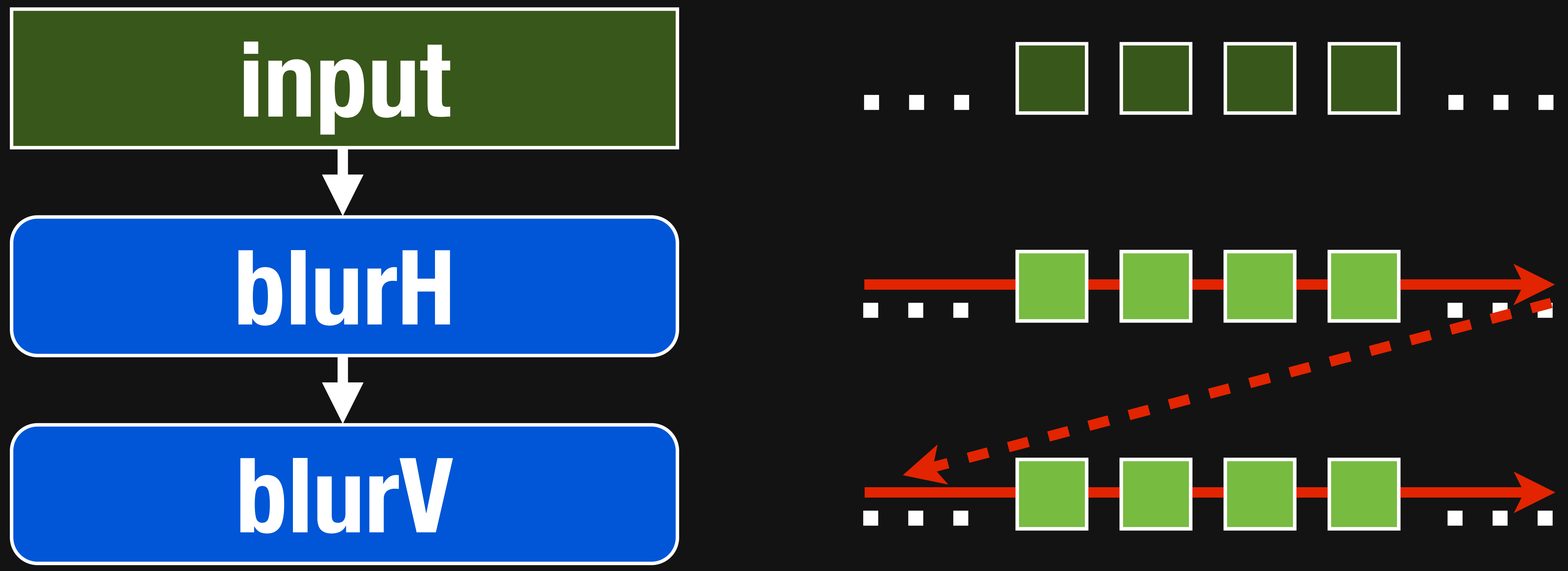
Complex graphs, deeper than traditional stencil computations

How can we **organize** this computation?

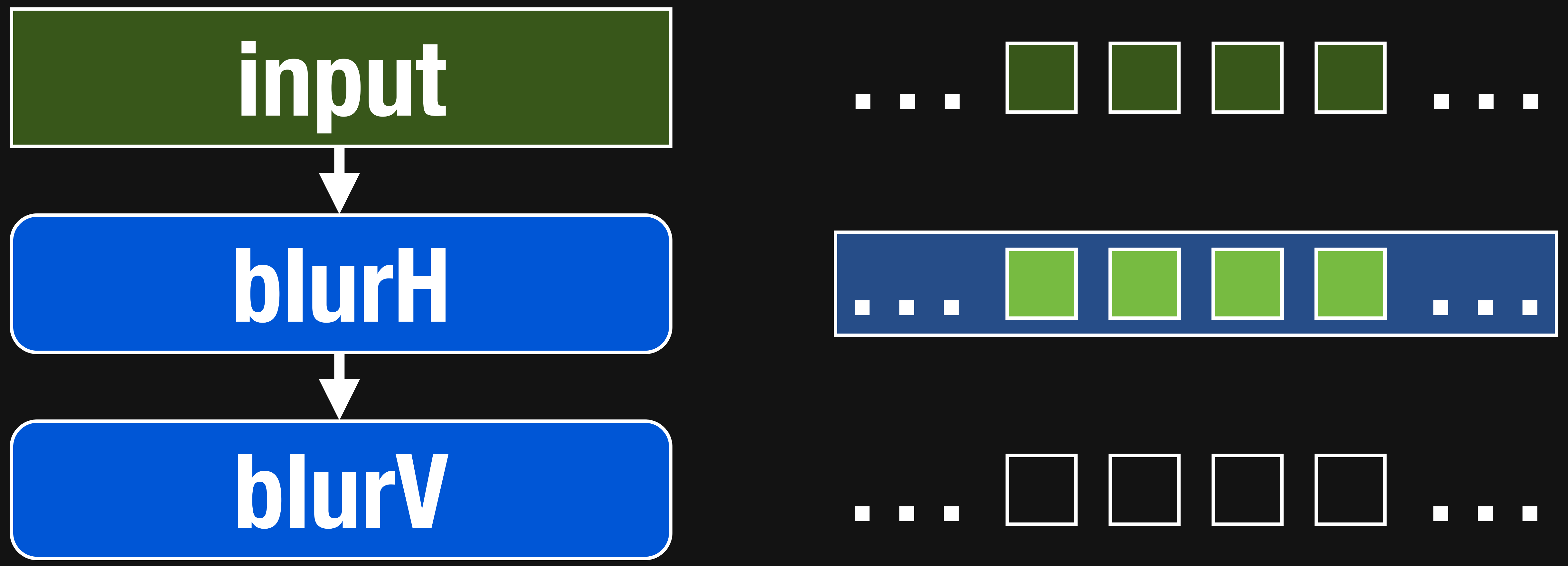
Organizing a data-parallel pipeline



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

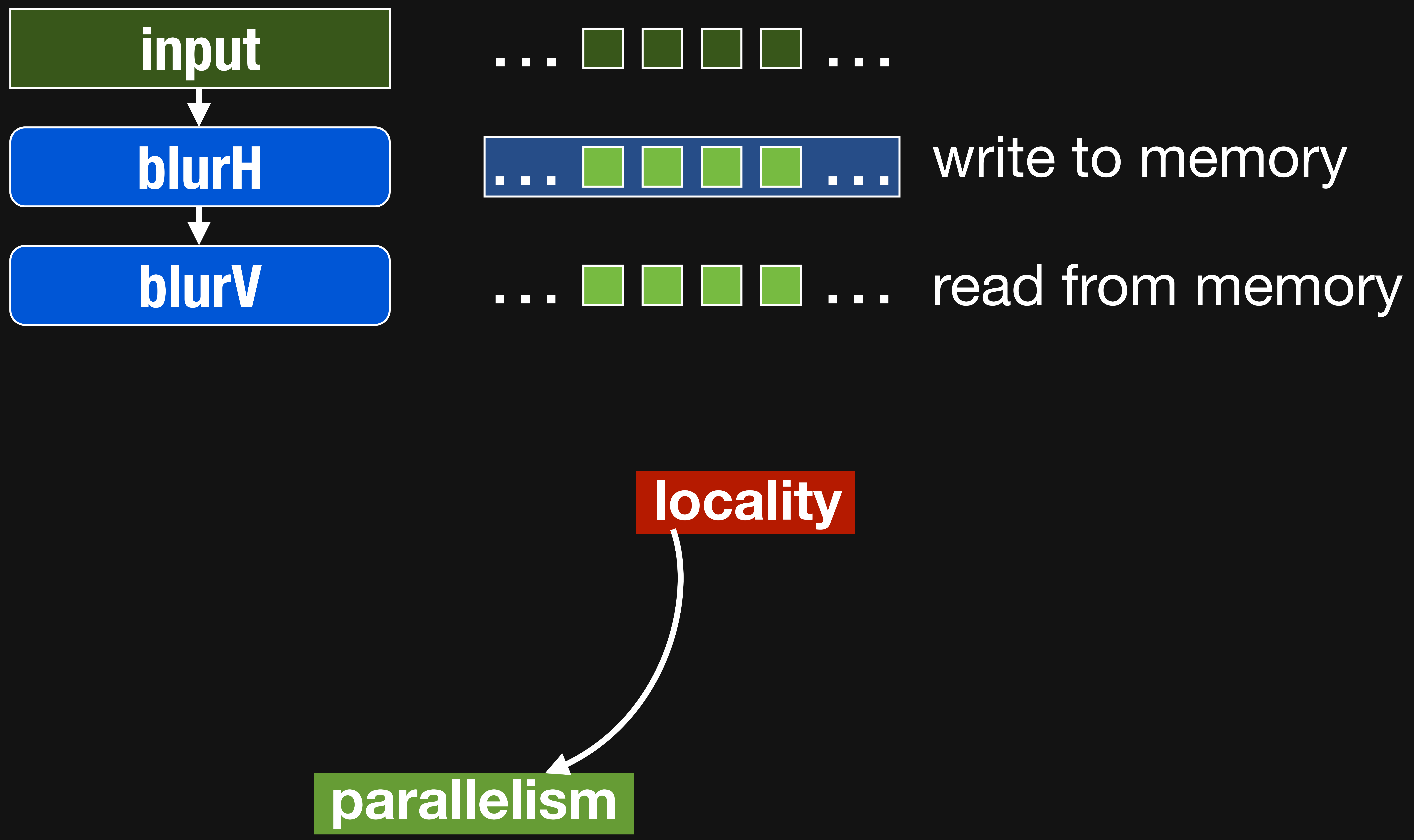


Simple loops execute **breadth-first** across stages

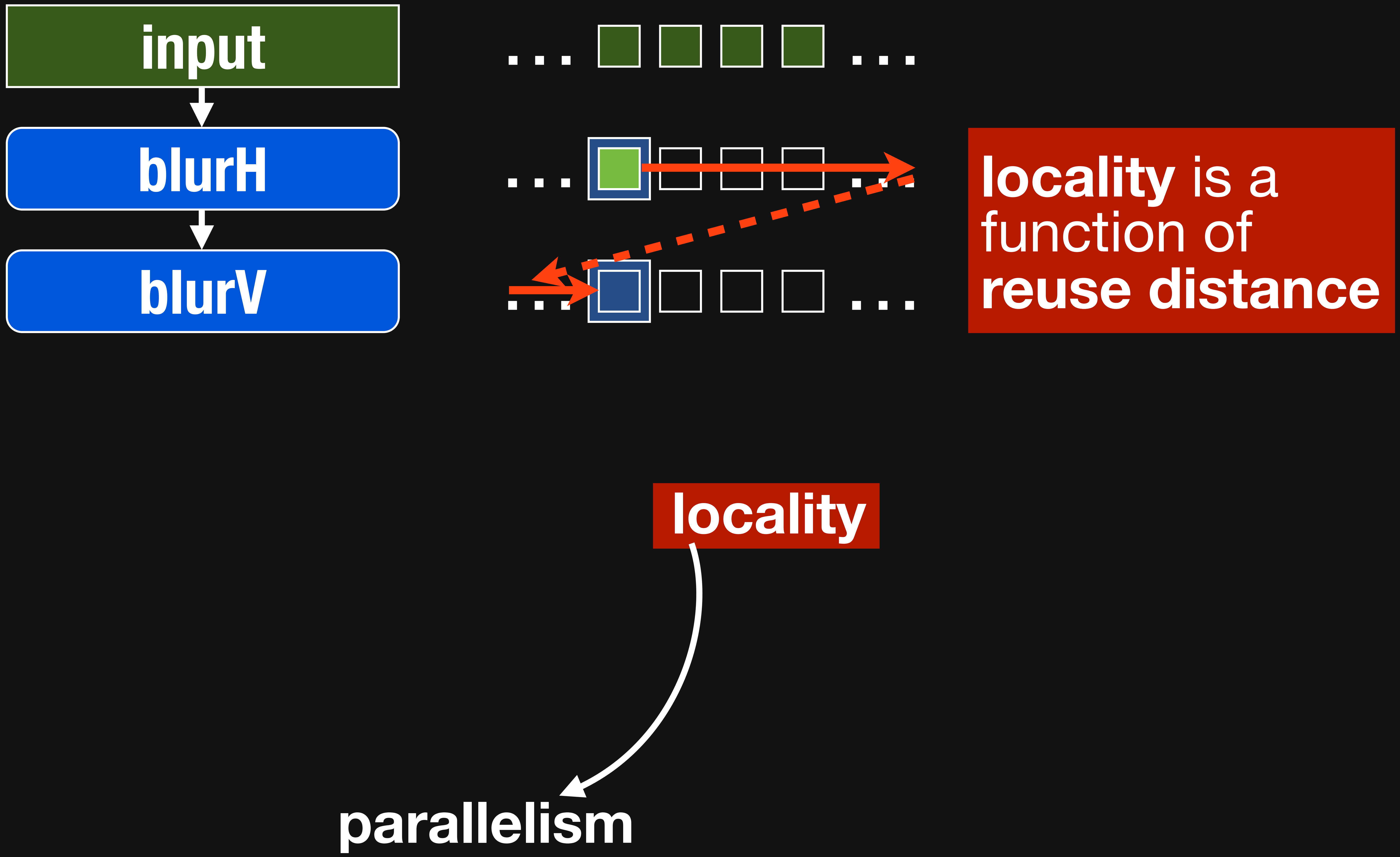


parallelism

Breadth-first execution **sacrifices locality**

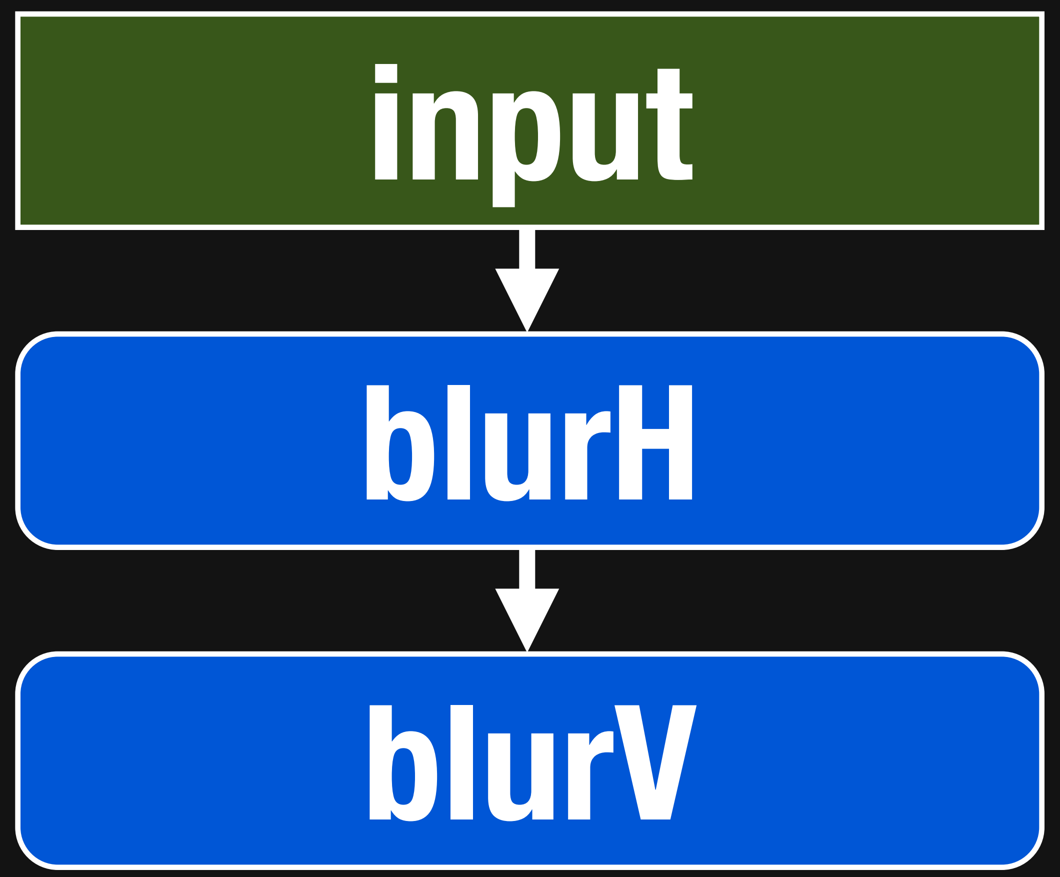


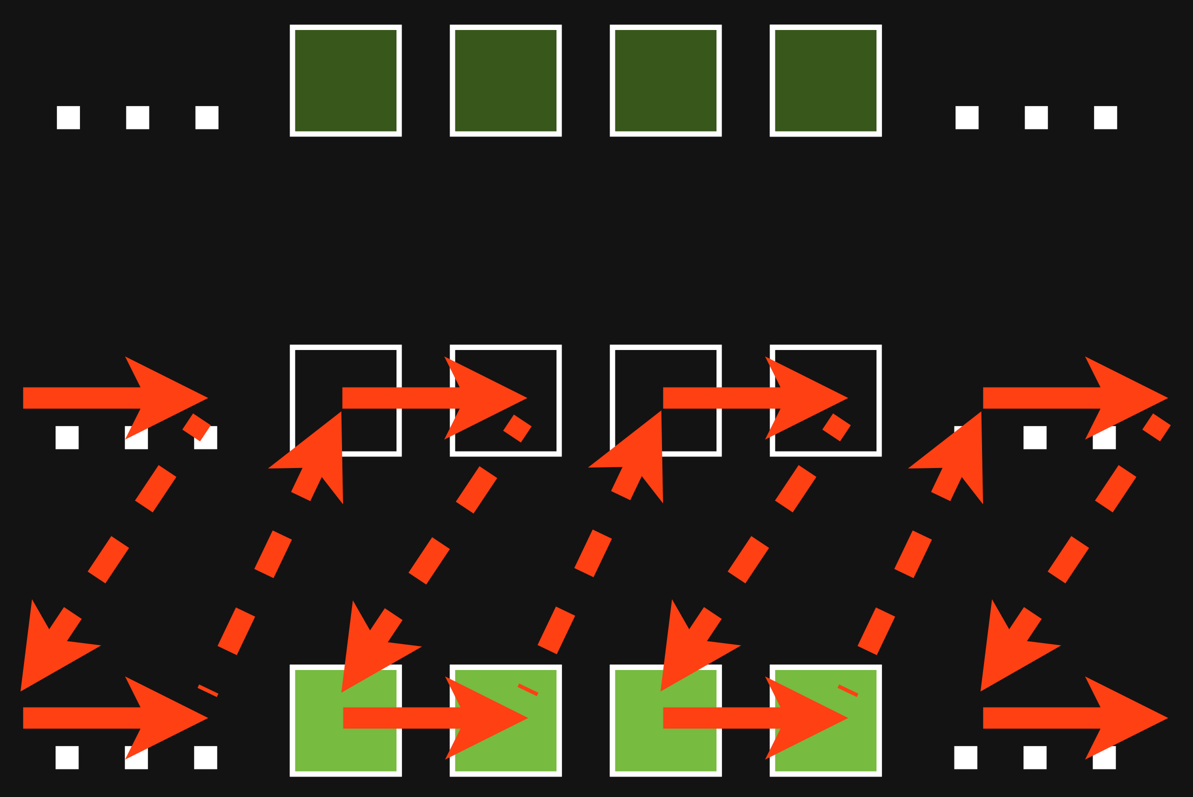
Breadth-first execution **sacrifices locality**



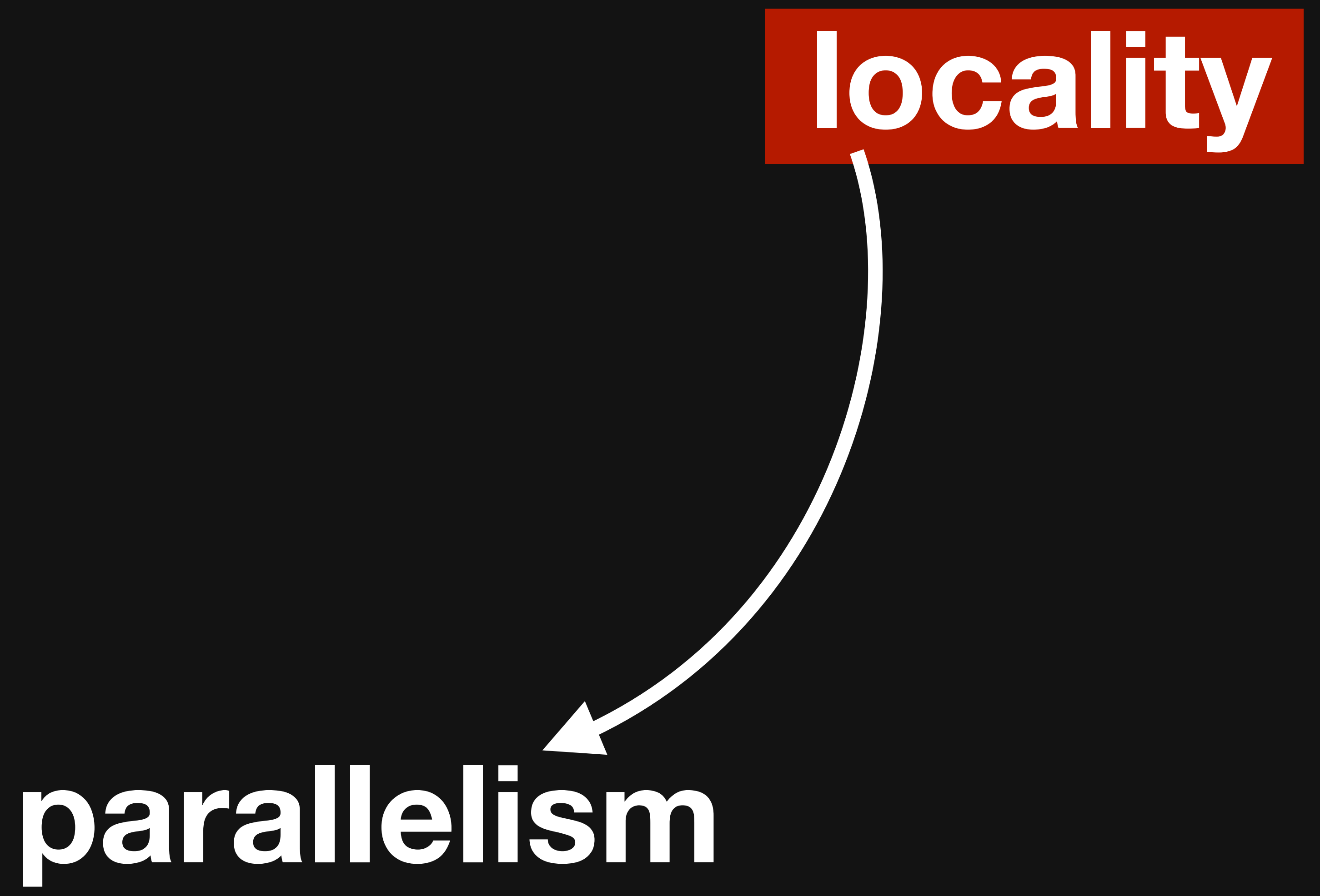
Interleaved execution (fusion) improves locality

*fusion **globally interleaves** computation*



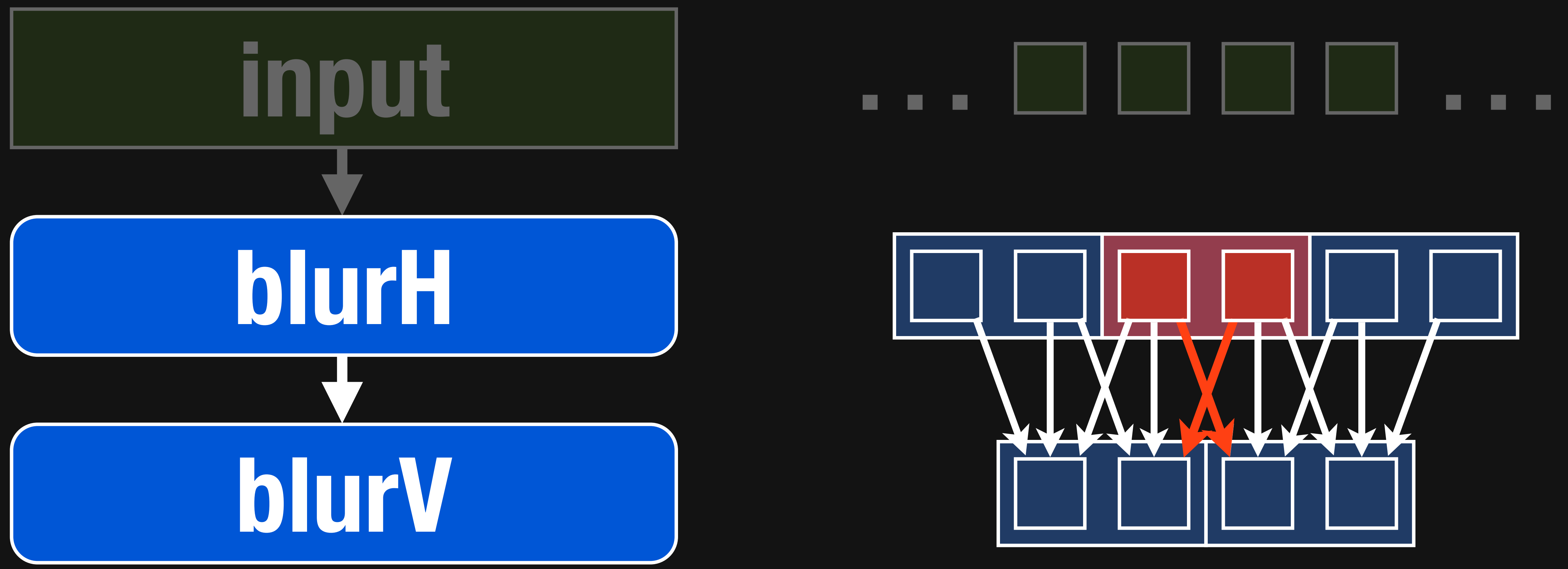


reduce reuse distance from **producer** to **consumer**

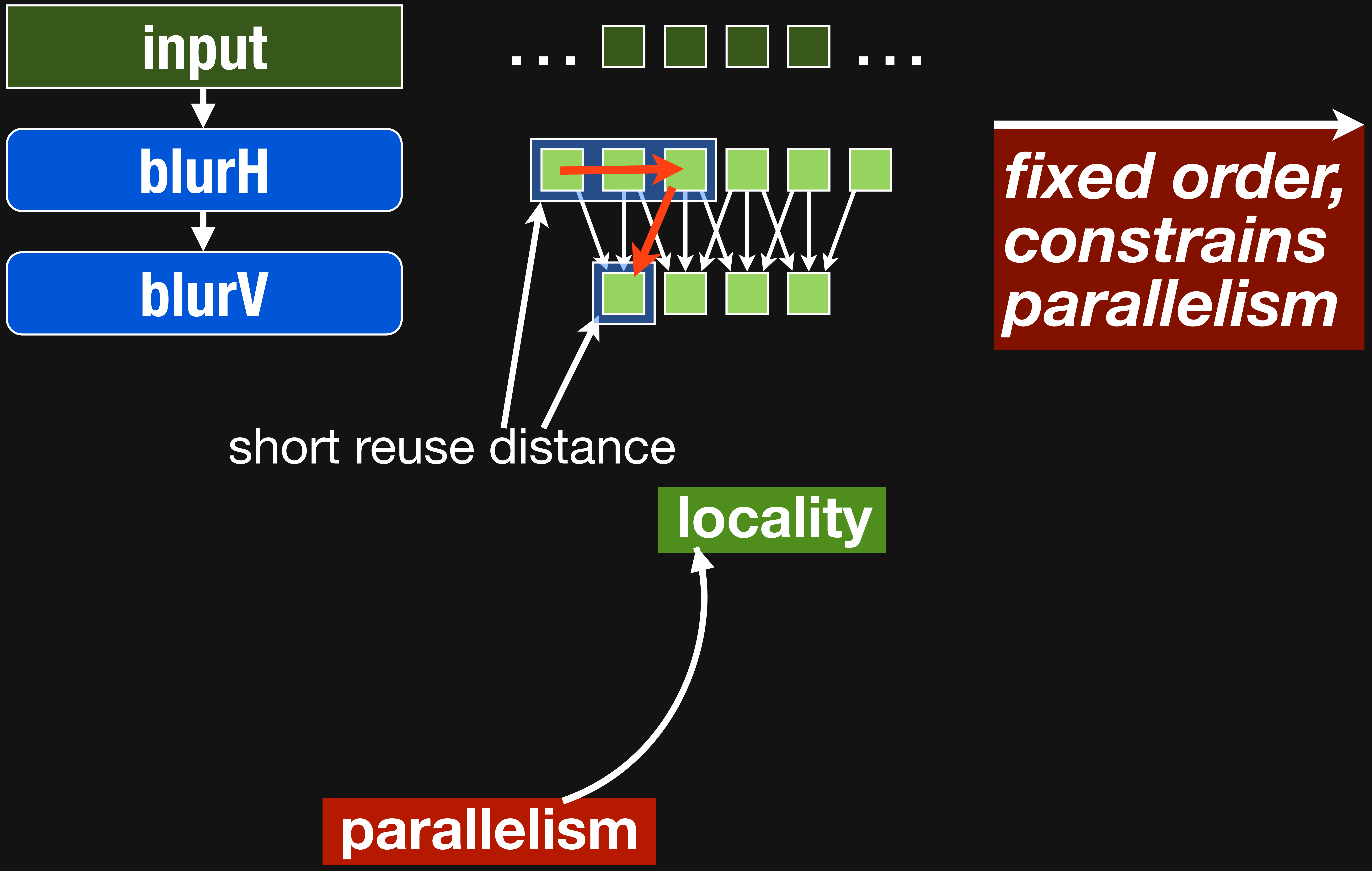


Understanding dependencies

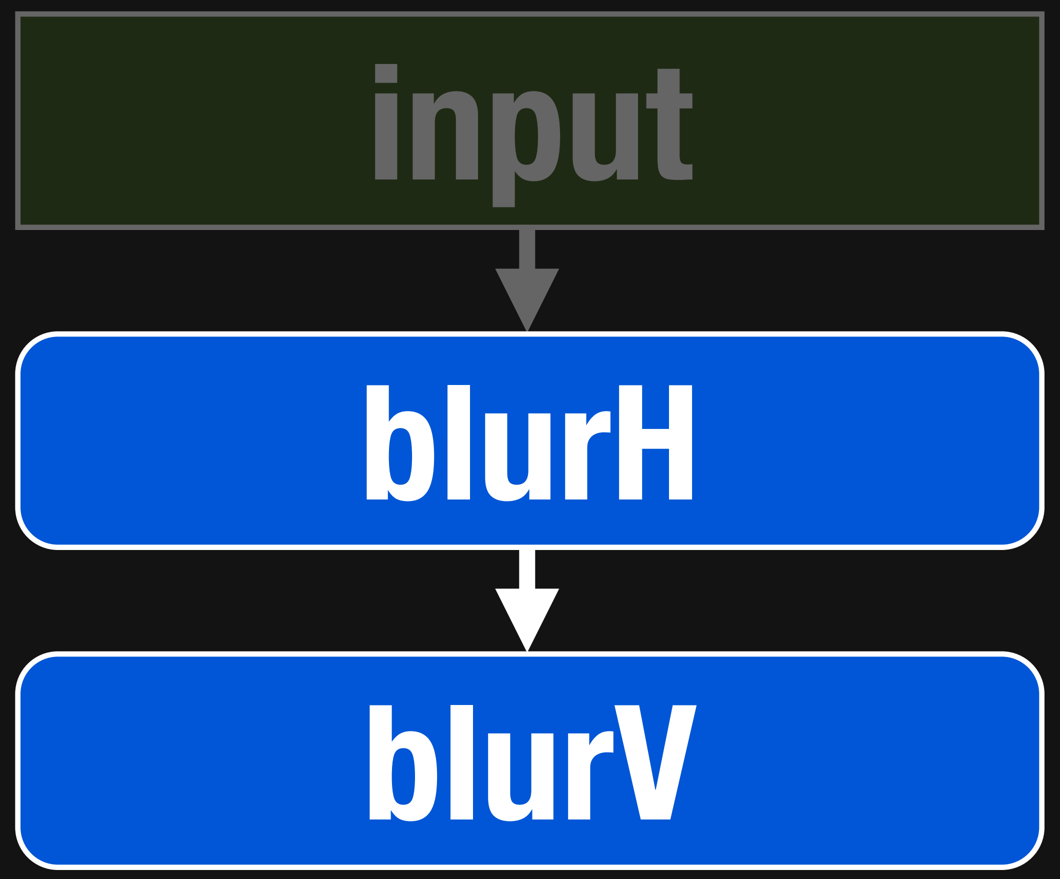
Stencils have overlapping dependencies

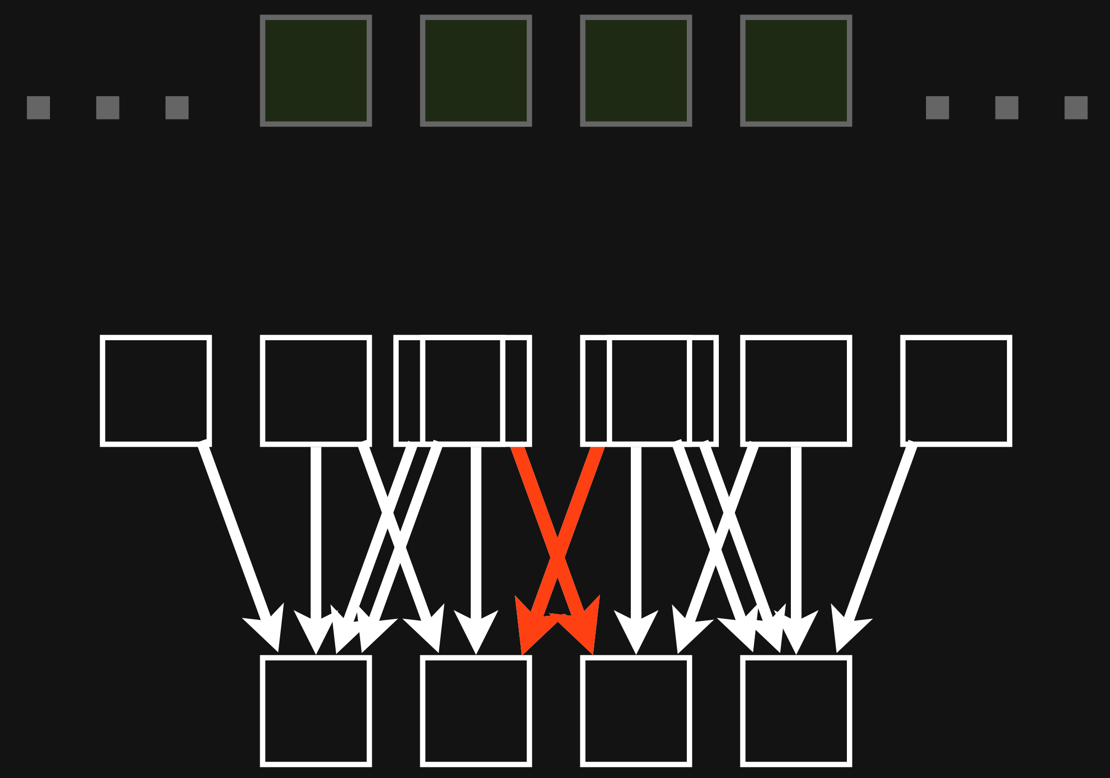


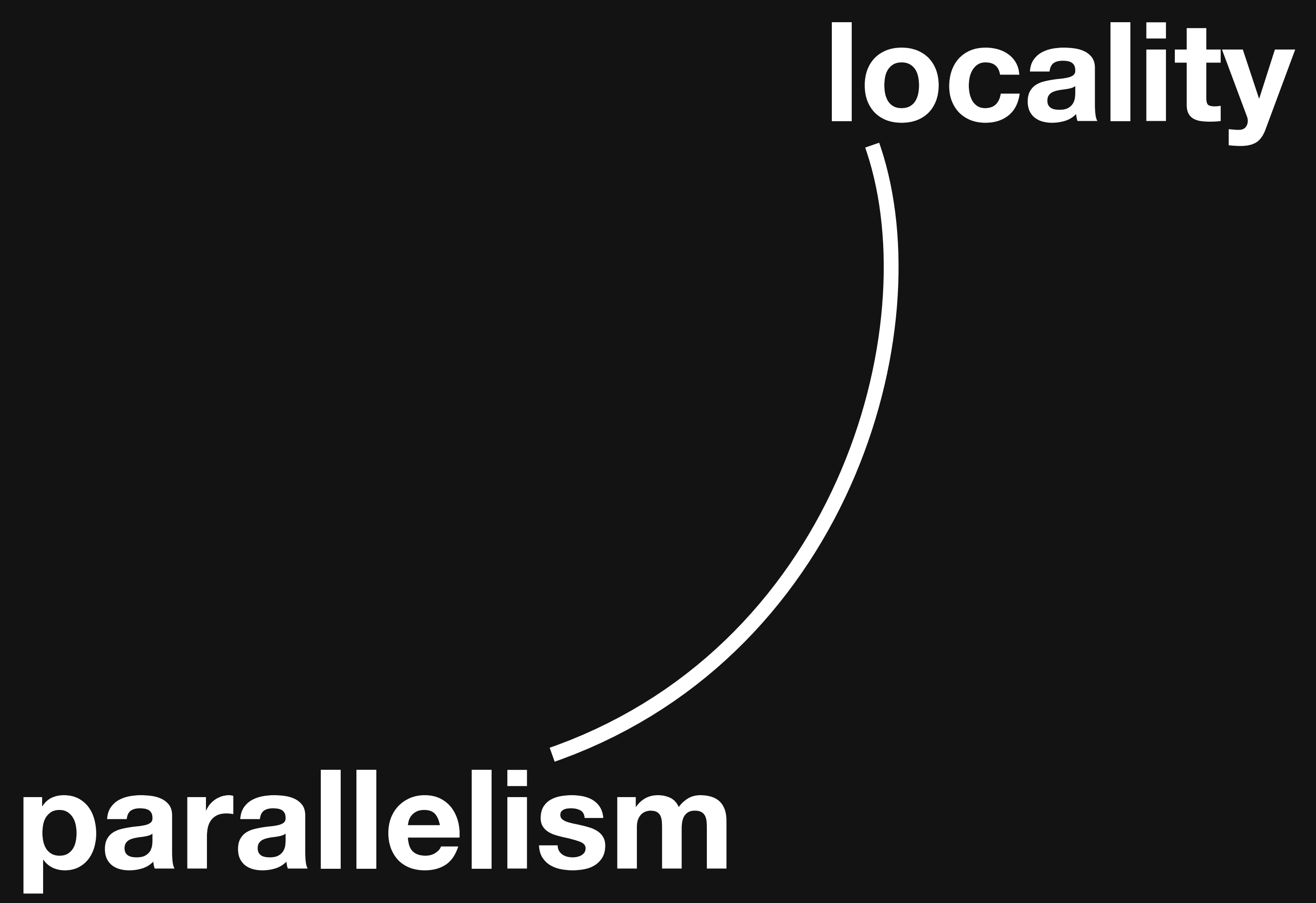
Sliding window execution **sacrifices parallelism**



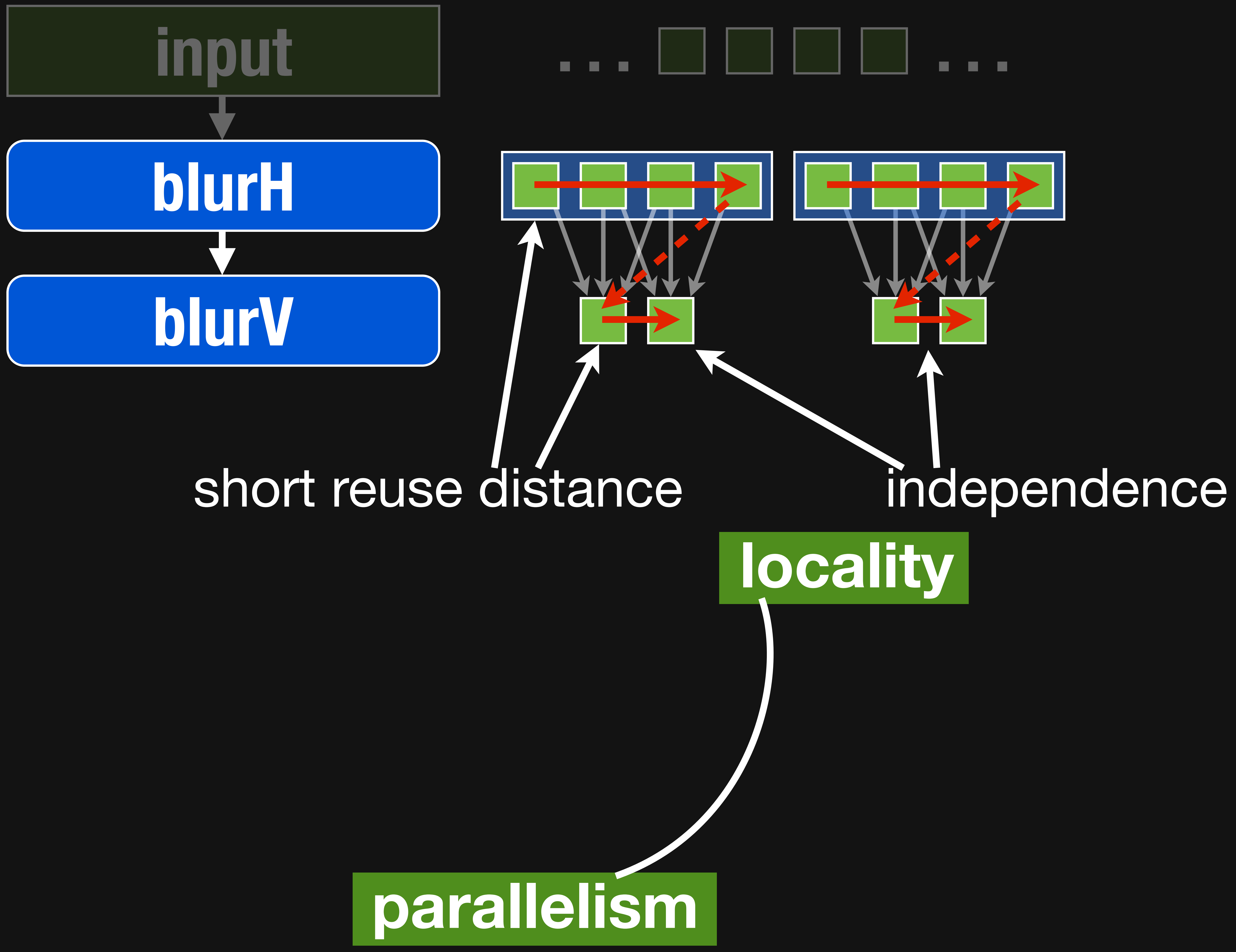
Breaking dependencies with tiling



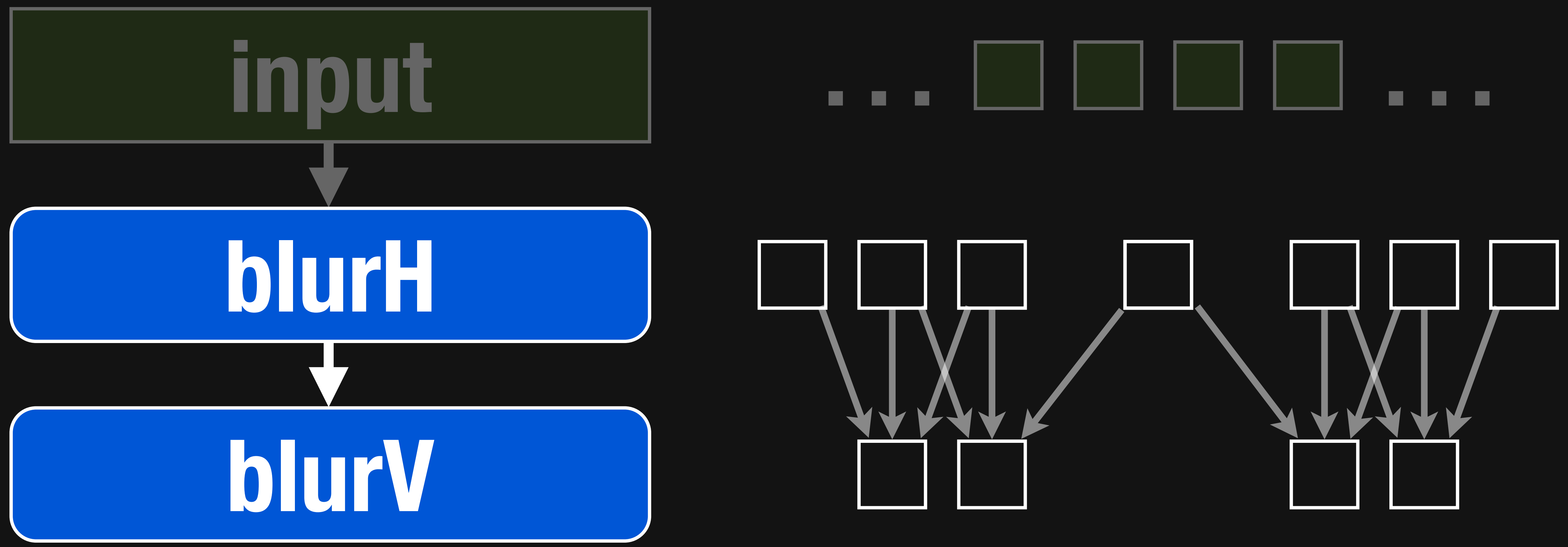


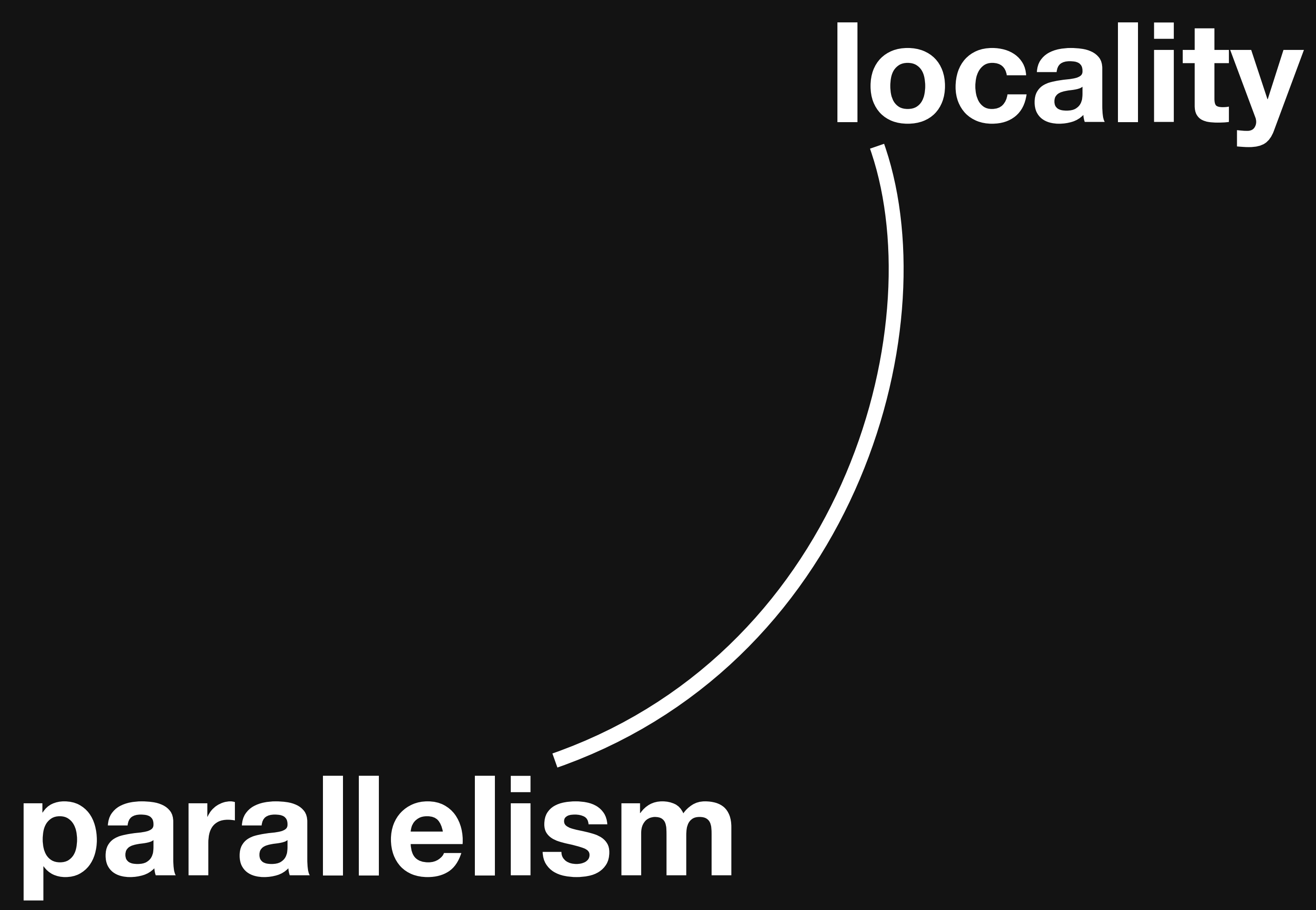


Decoupled tiles optimize **parallelism & locality**

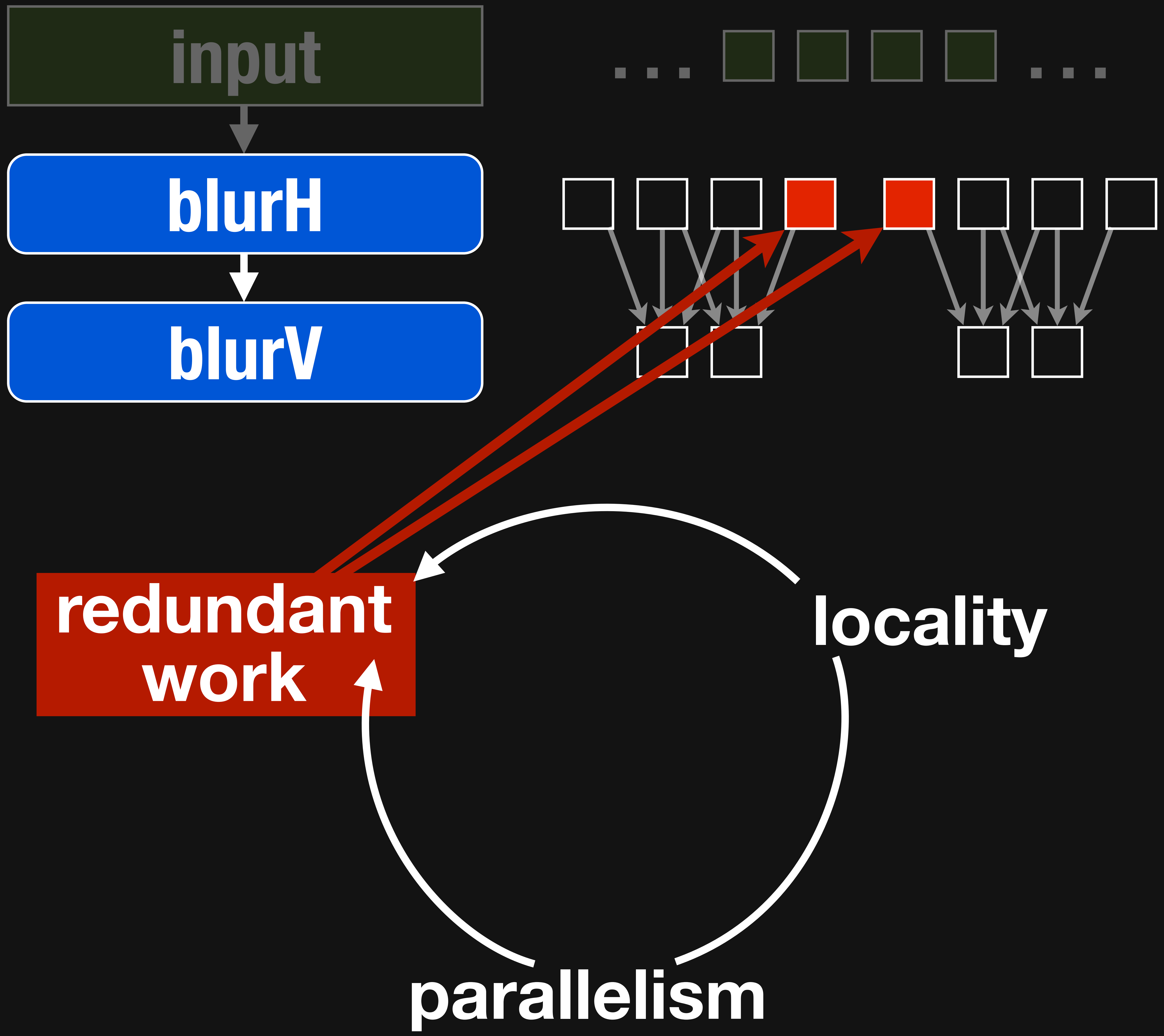


Breaking dependencies introduces **redundant work**

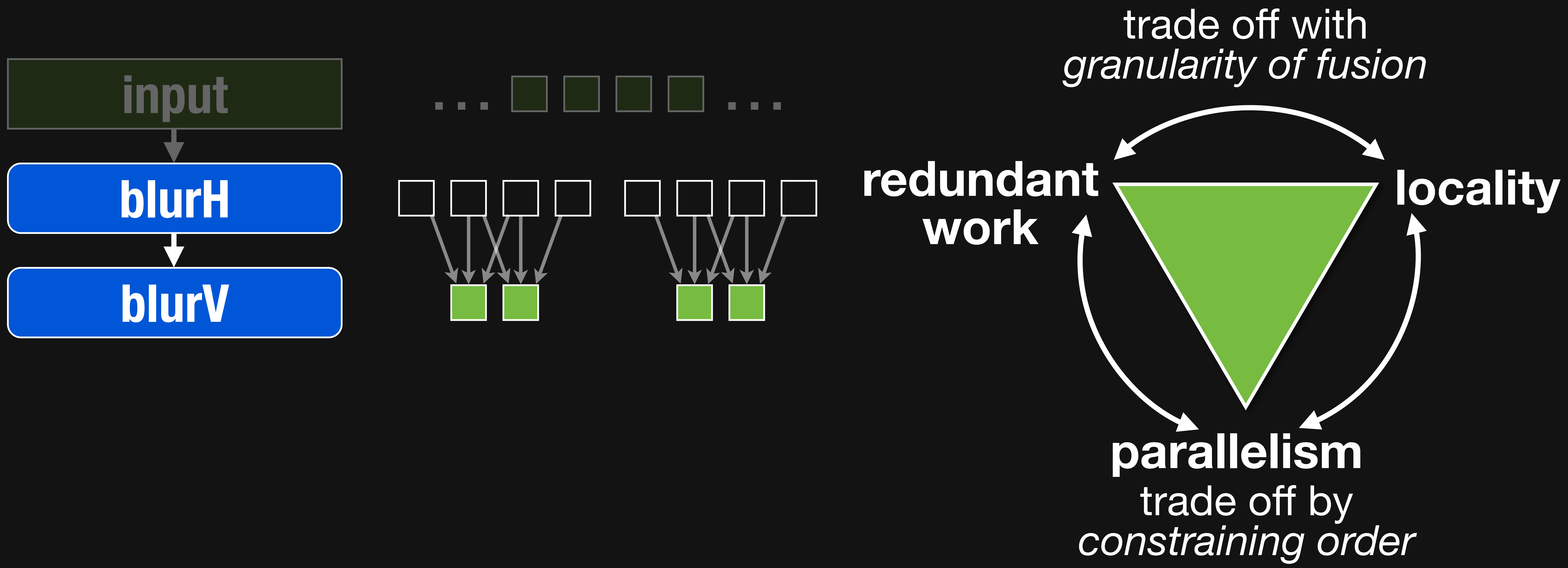




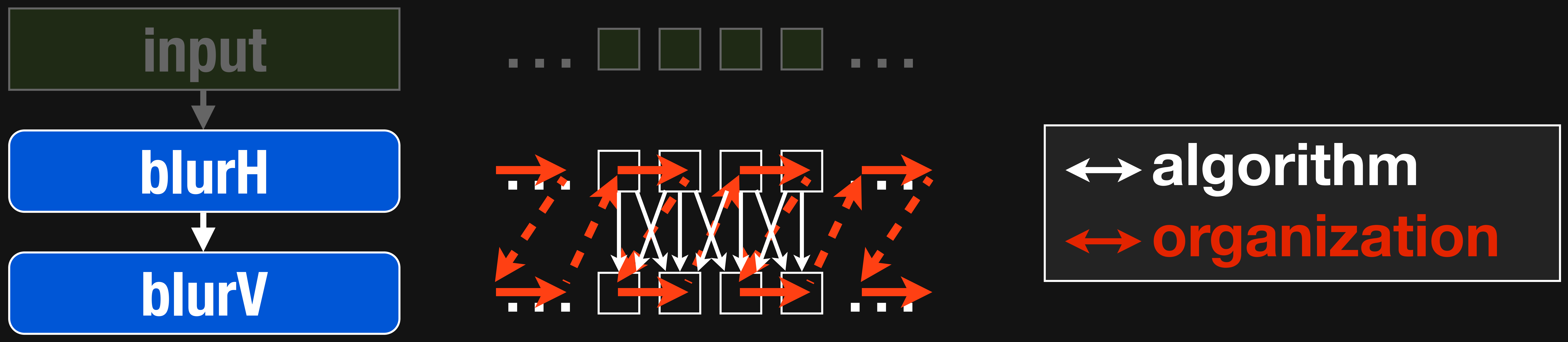
Breaking dependencies introduces **redundant work**



Message #1: performance requires tradeoffs

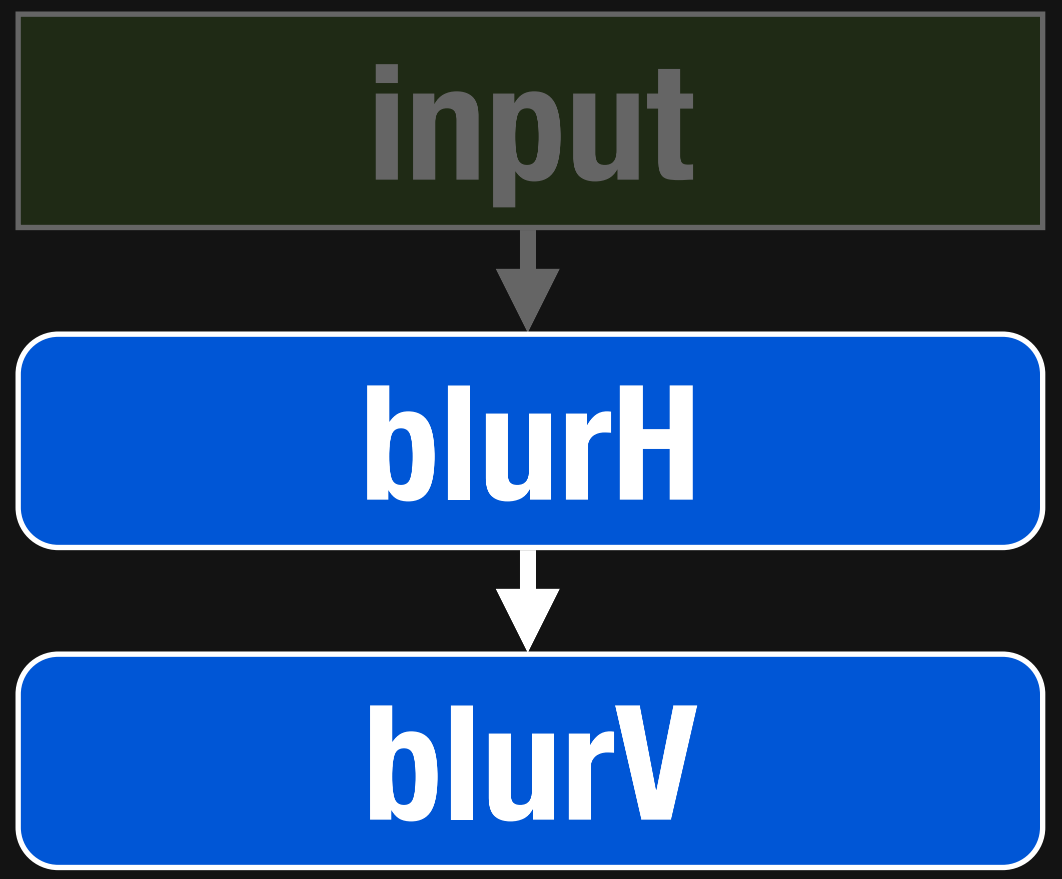


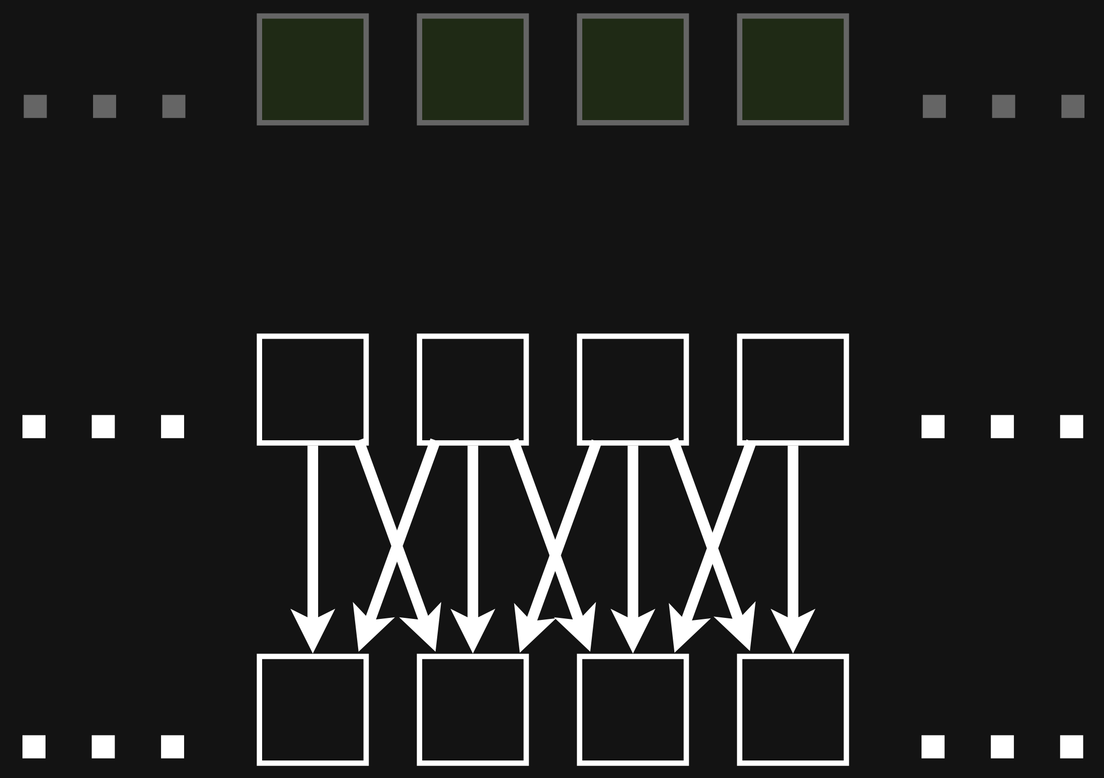
Message #2: algorithm vs. organization



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

Dependencies limit choices of organization





↔ dependencies
↔ organization

A **language** of **schedules**

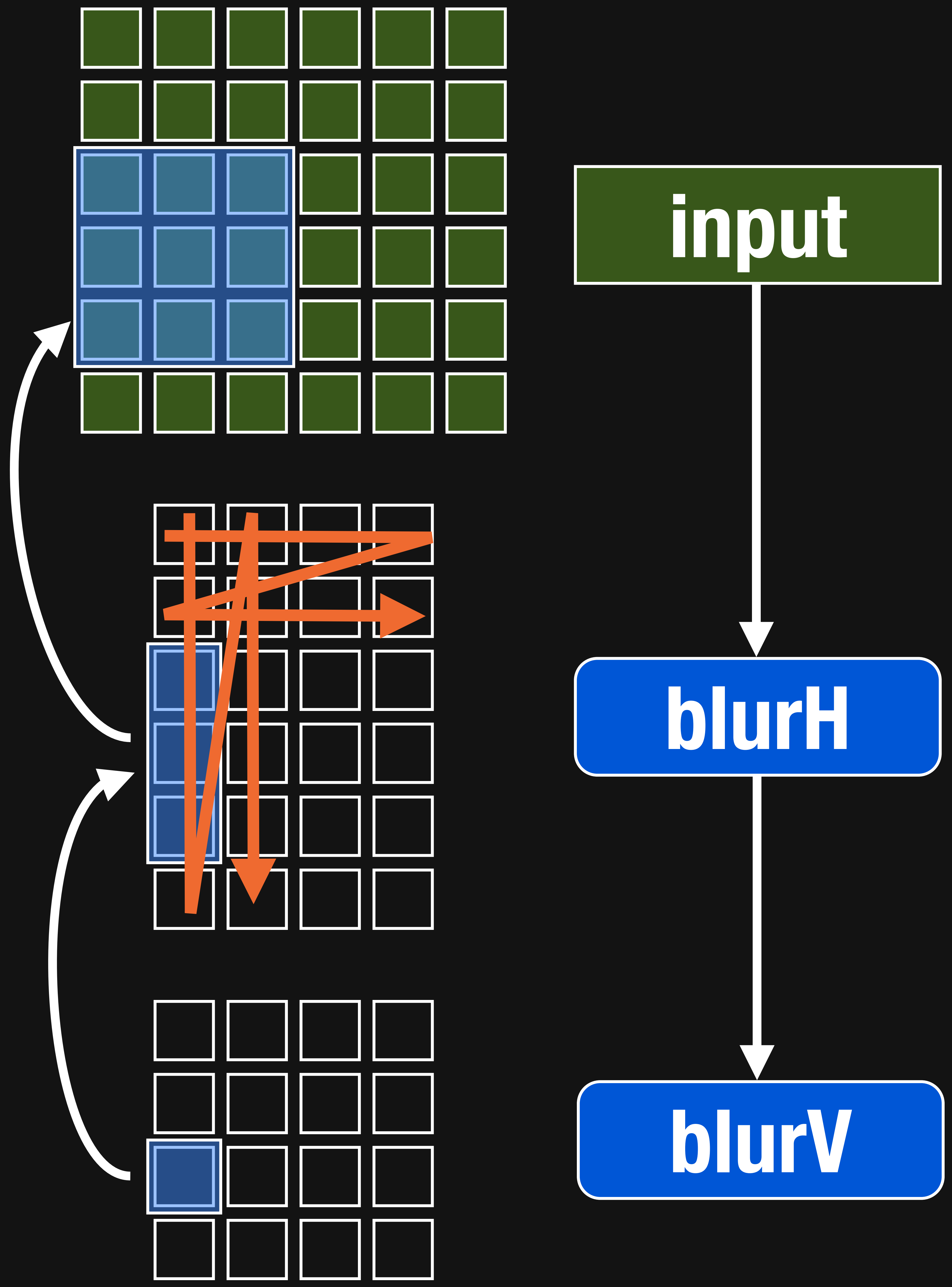
The schedule defines intra-stage order, inter-stage interleaving

For each stage:

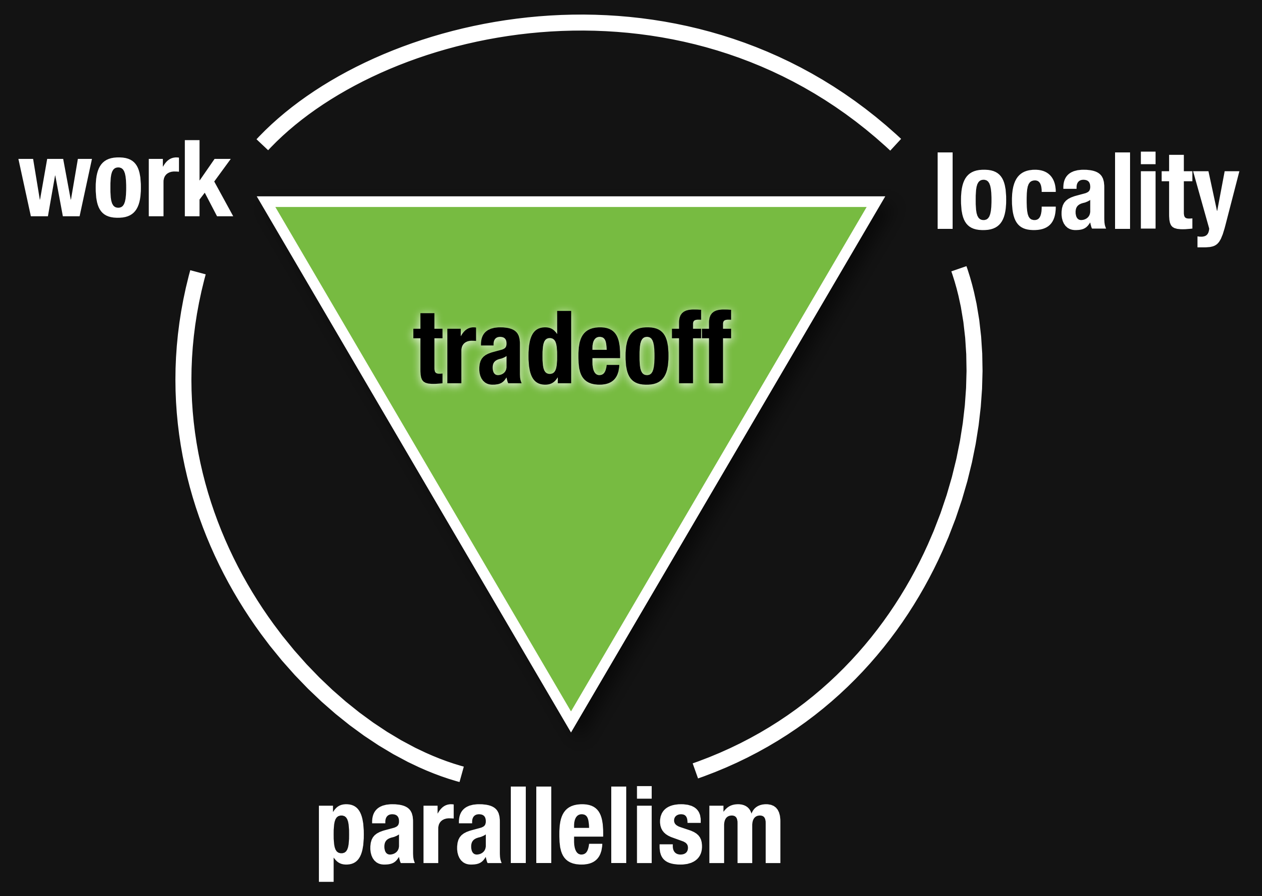
1) In **what order** should we **compute its values?**

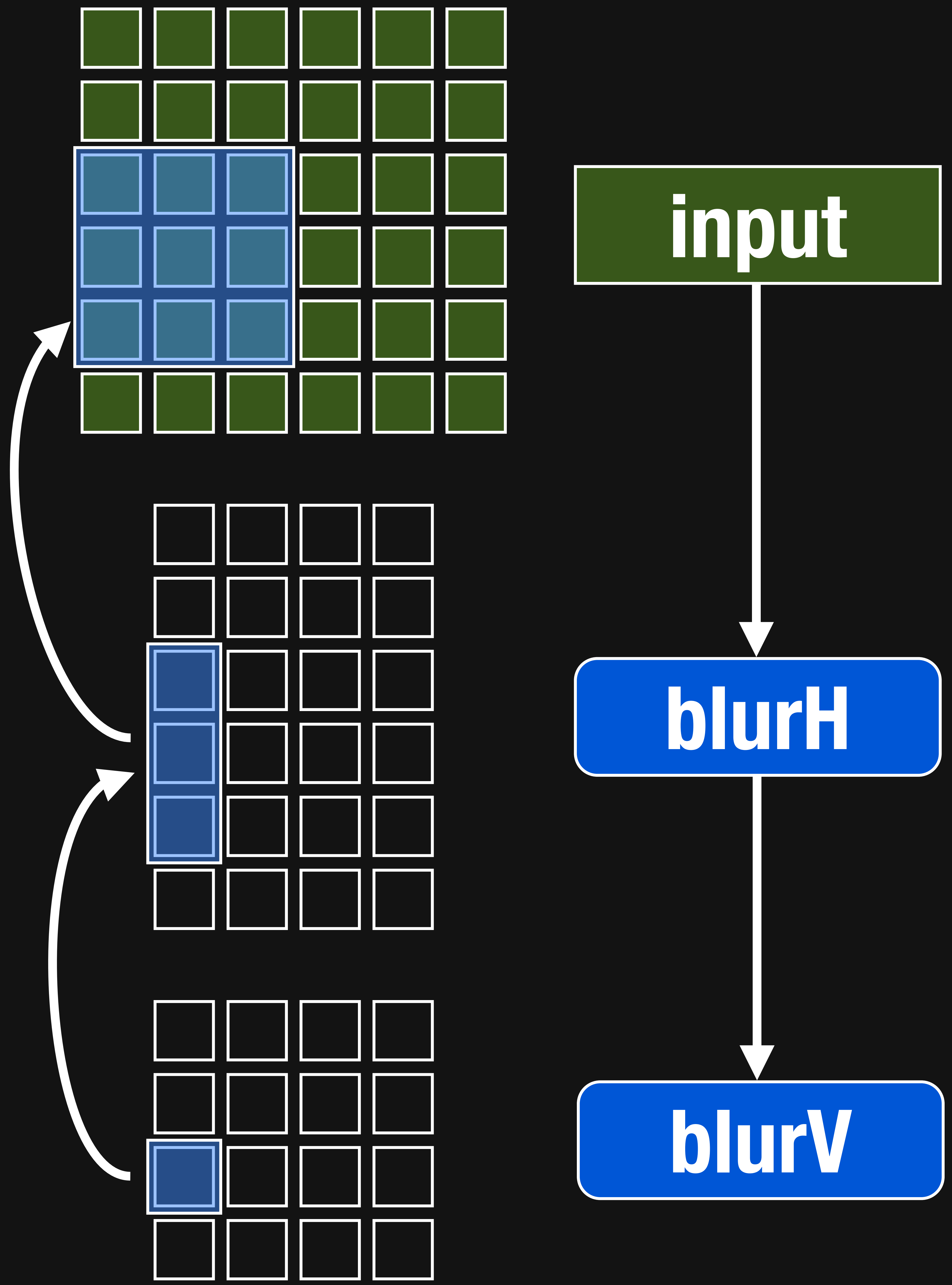
2) **When** should we **compute its inputs?**

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

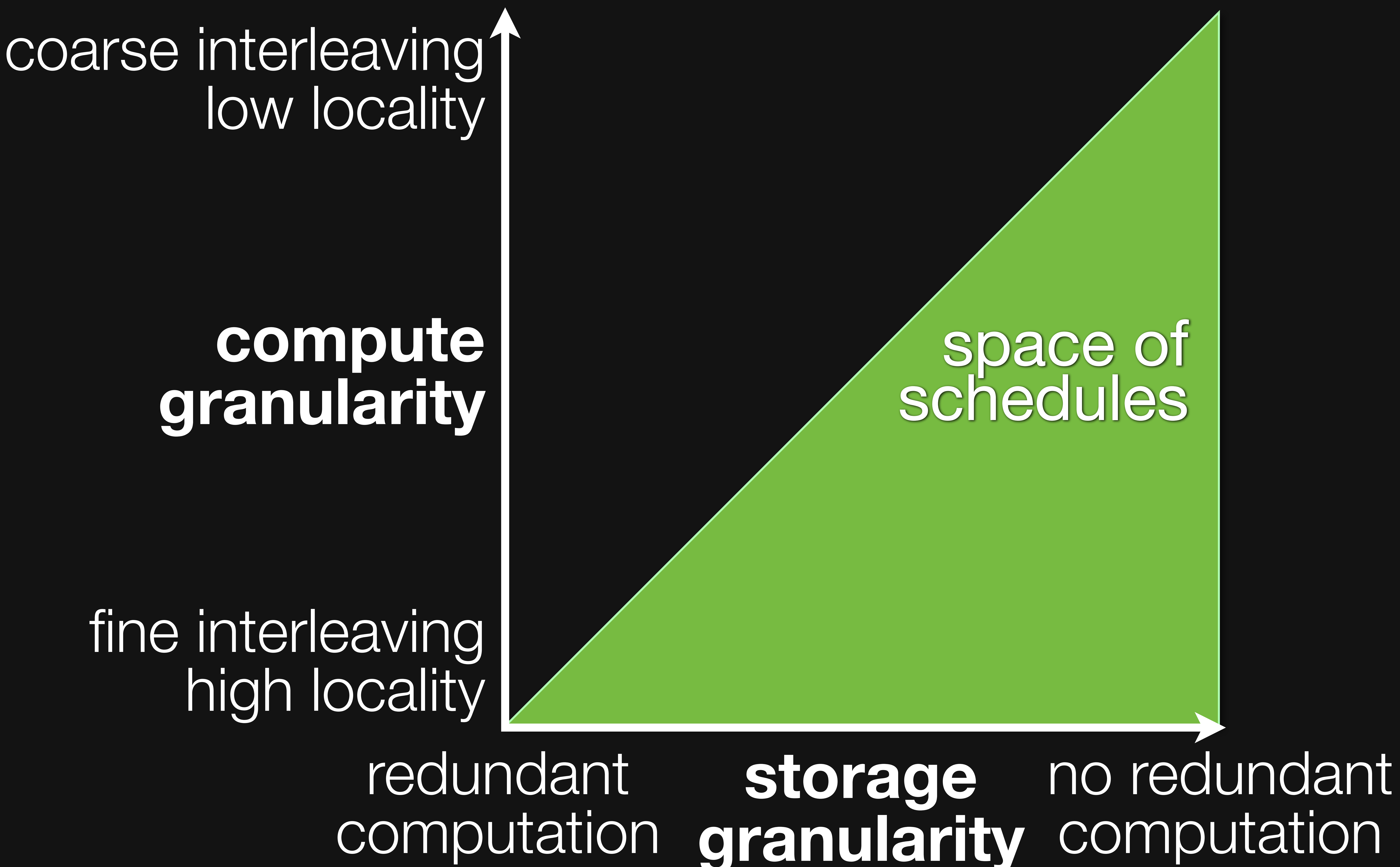


The schedule defines intra-stage order, inter-stage interleaving

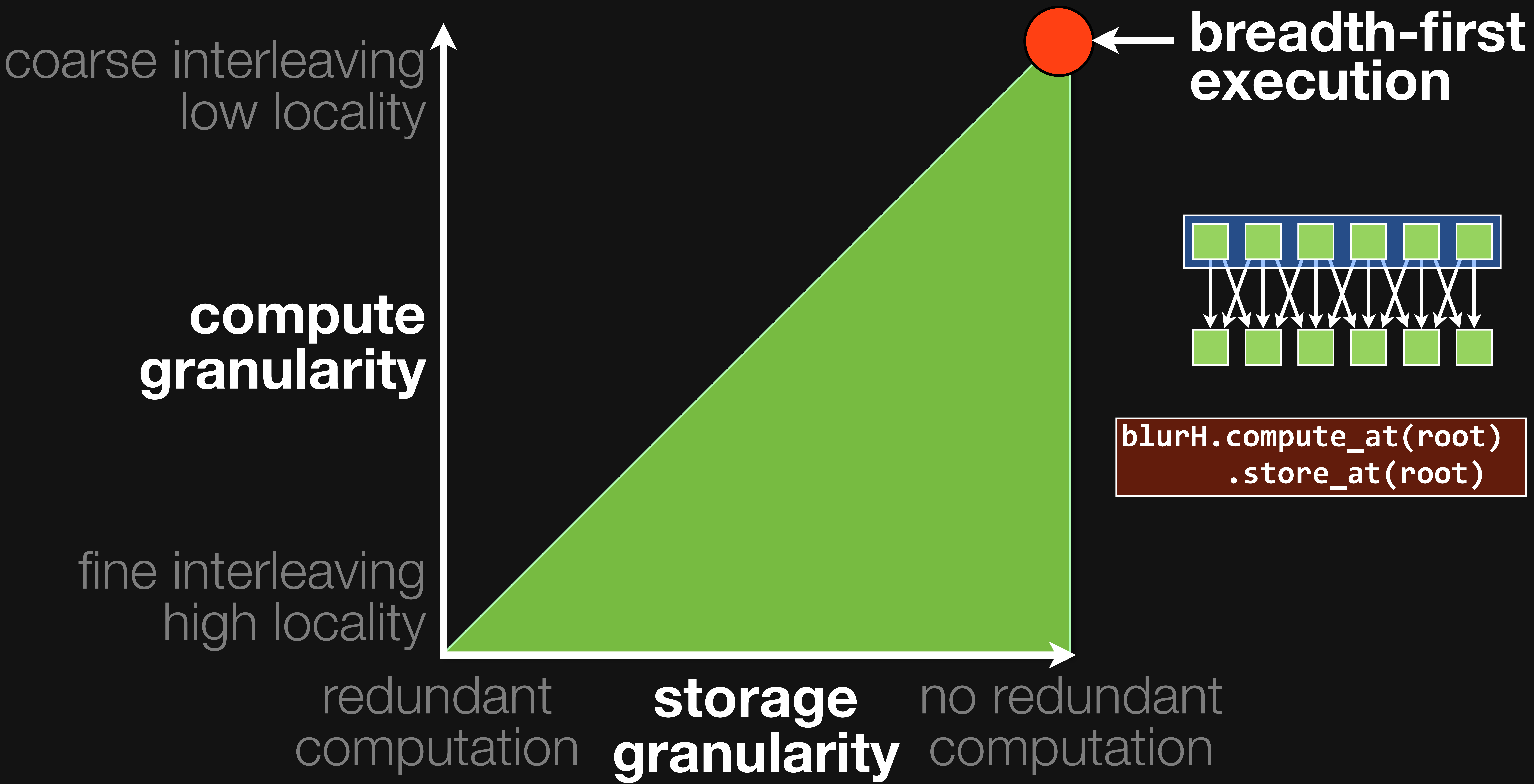




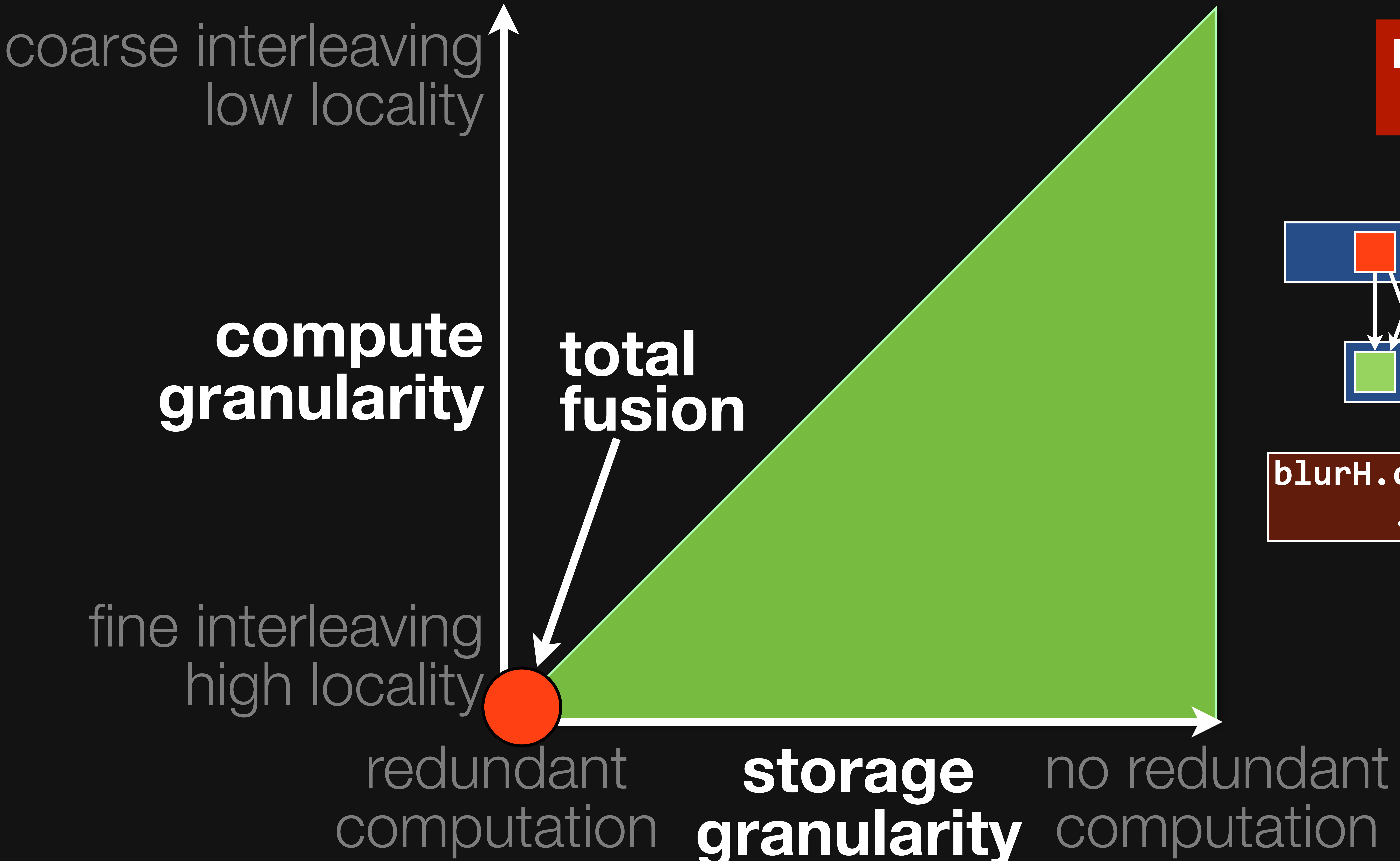
Tradeoff space modeled by **granularity of interleaving**



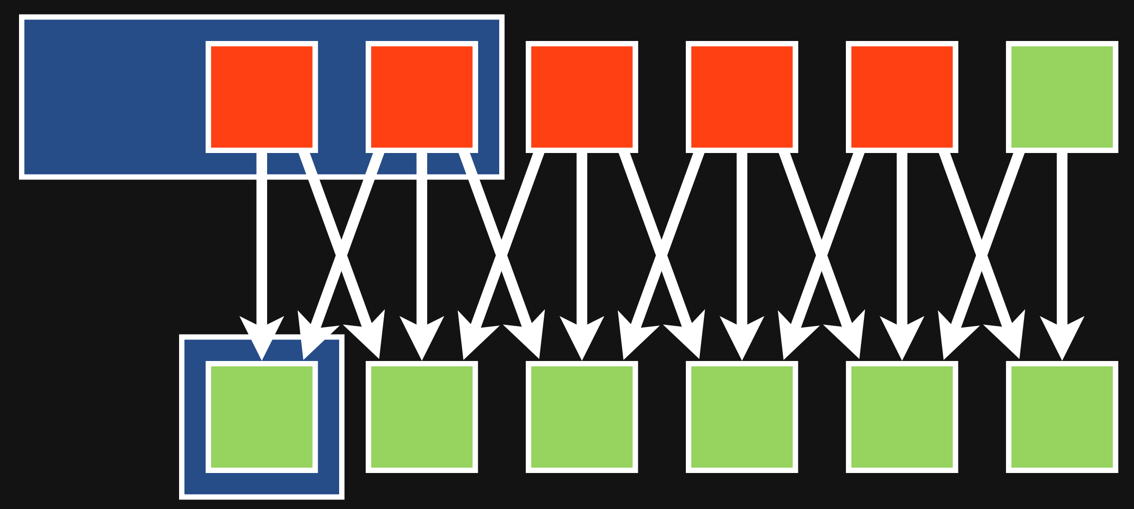
Tradeoff space modeled by **granularity of interleaving**



Tradeoff space modeled by **granularity of interleaving**

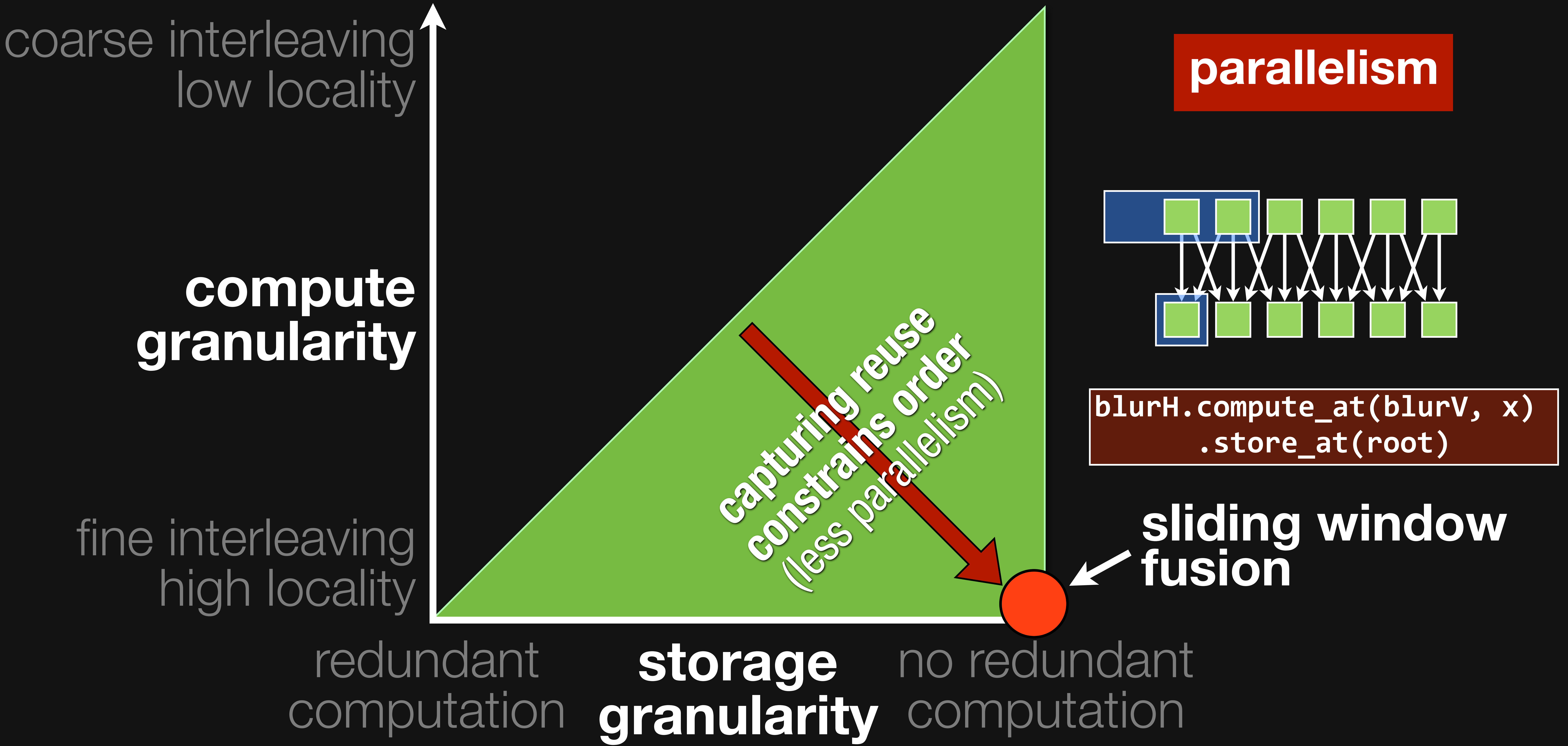


redundant work

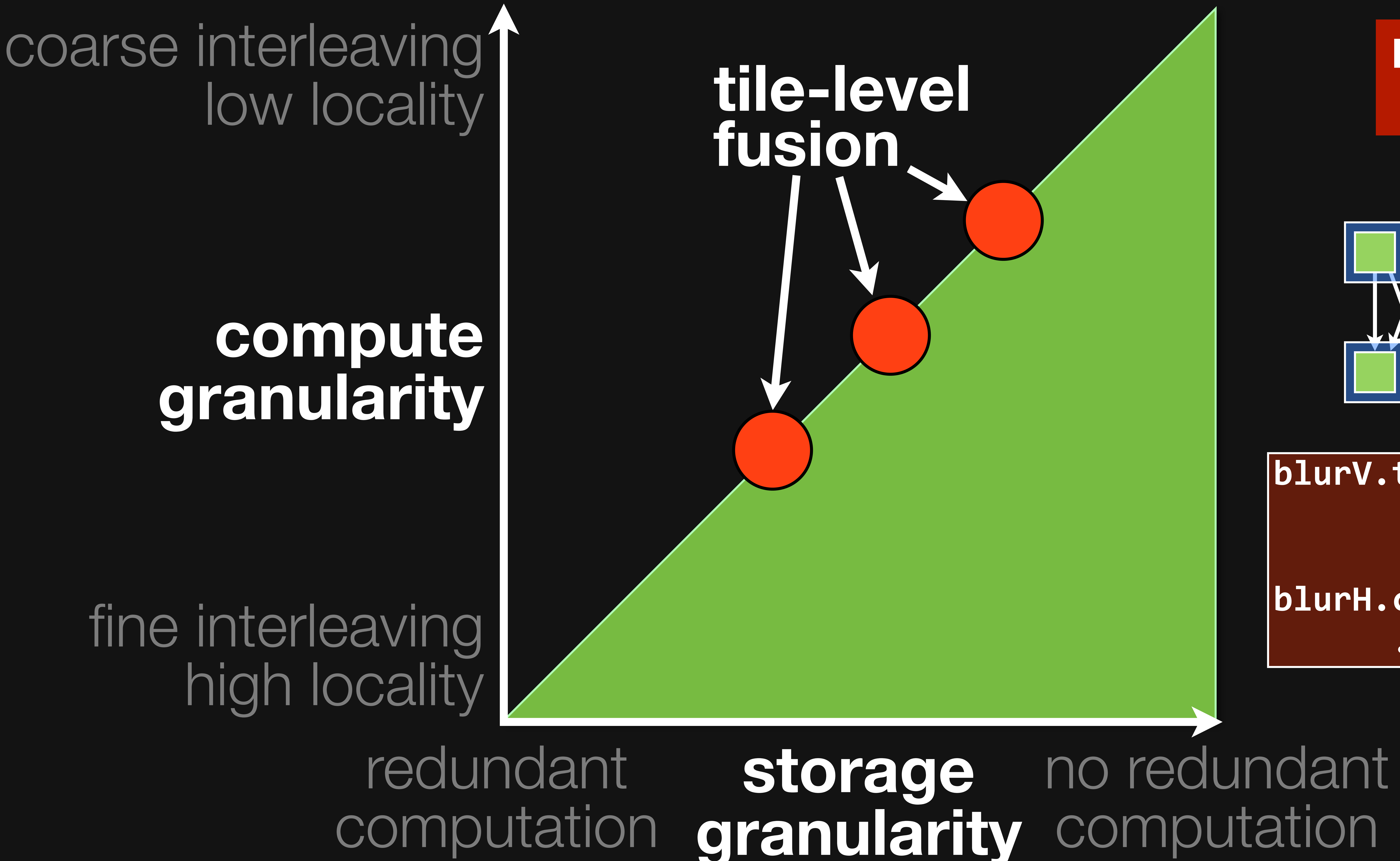


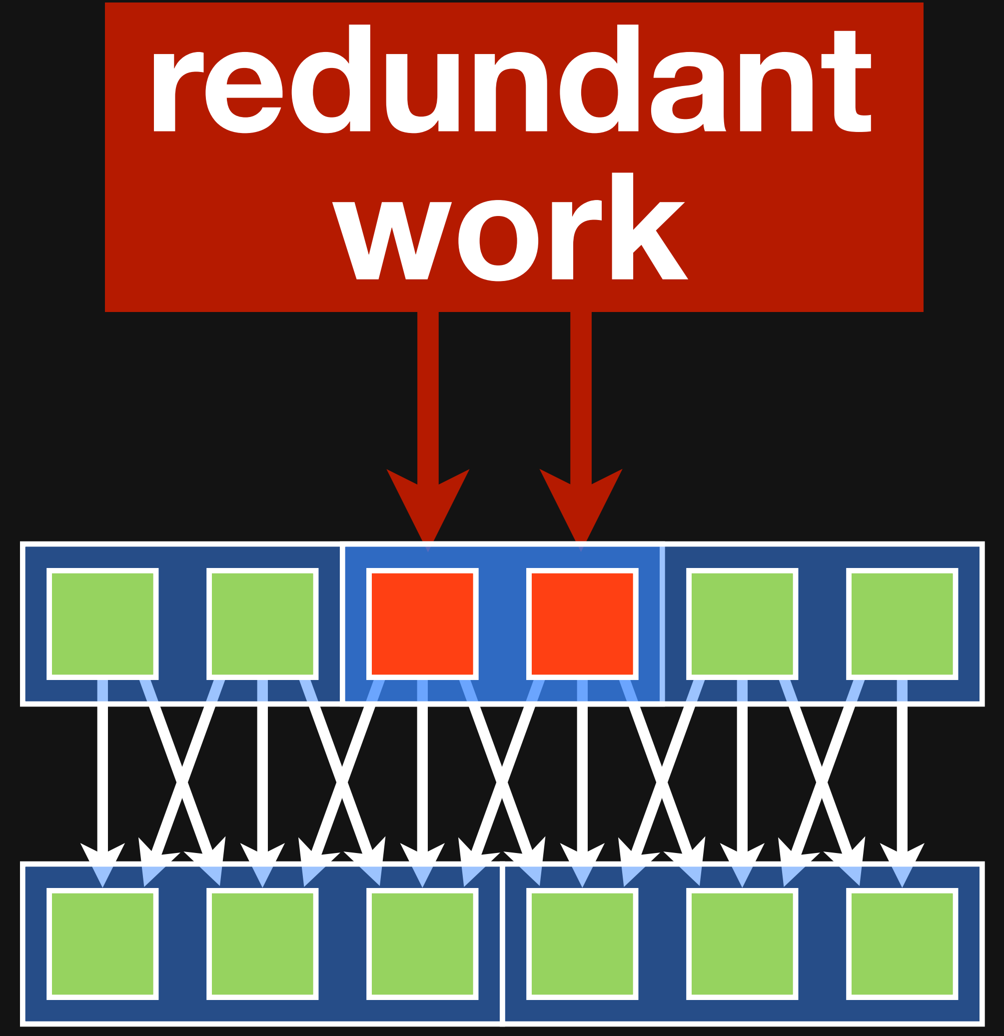
```
blurH.compute_at(blurV, x)
     .store_at(blurV, x)
```

Tradeoff space modeled by **granularity of interleaving**



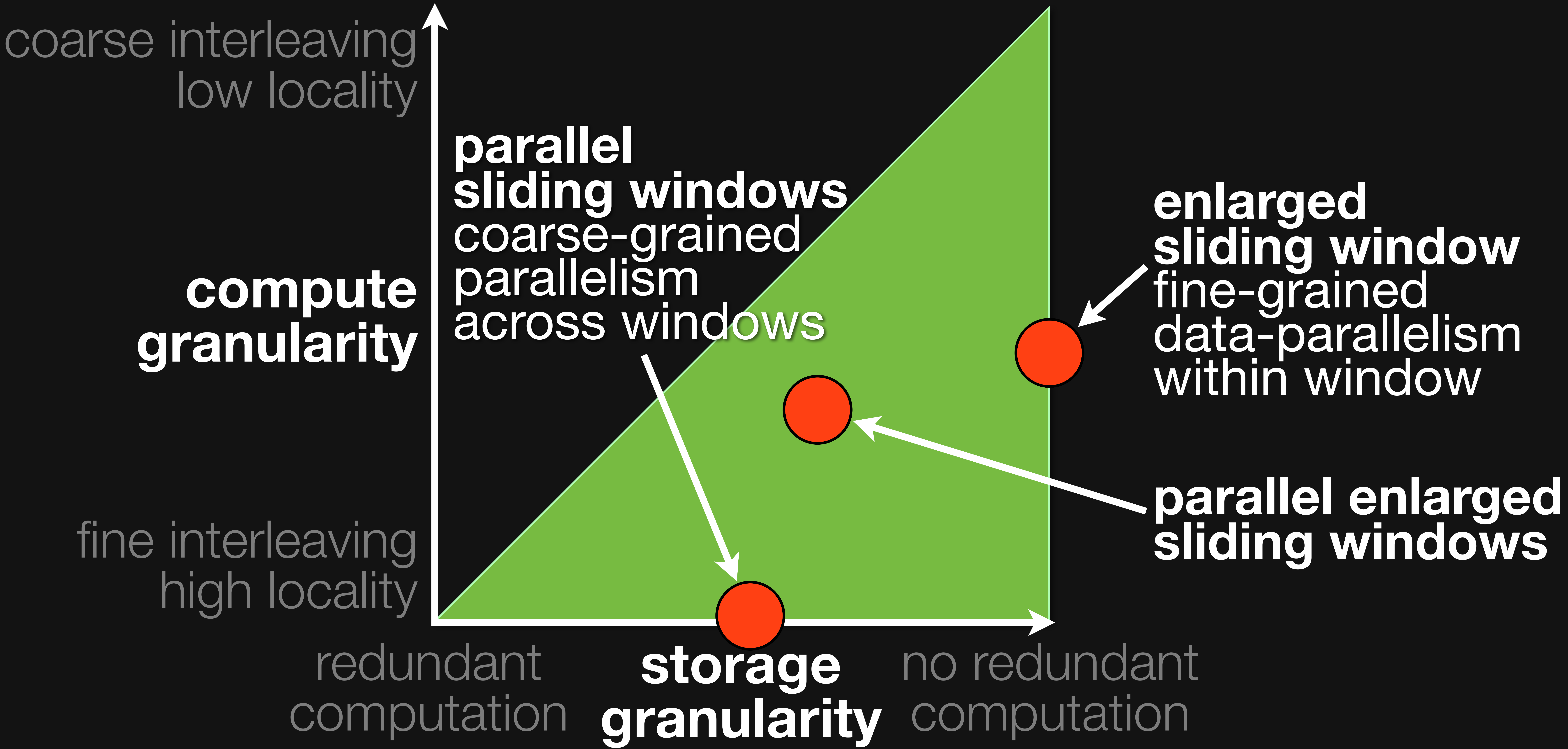
Tradeoff space modeled by **granularity of interleaving**



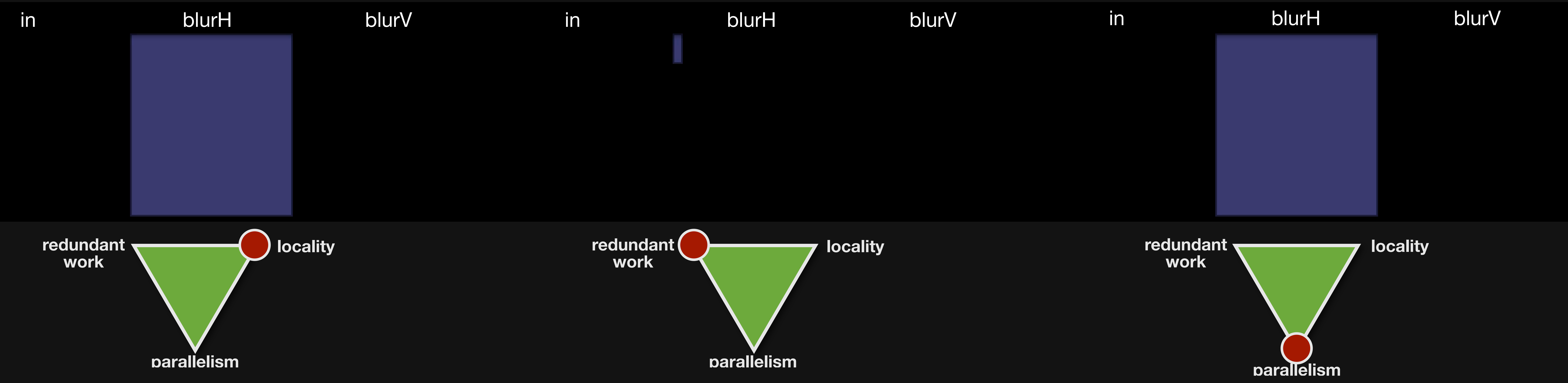



```
blurV.tile(xo, yo,
           xi, yi,
           W,  H)
blurH.compute_at(blurV, xo)
     .store_at(blurV, xo)
```

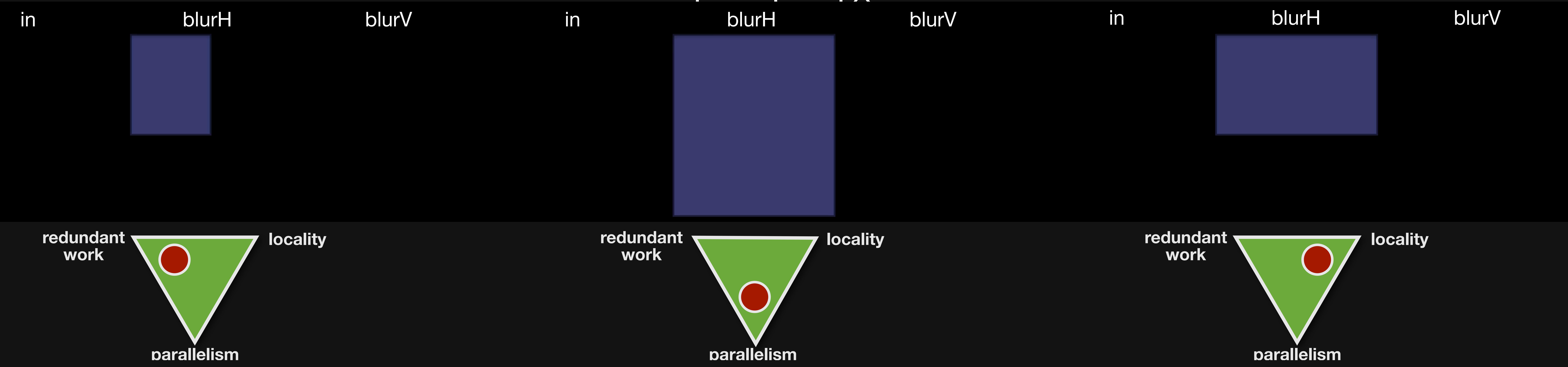
Tradeoff space modeled by **granularity of interleaving**



Schedule primitives **compose** to create many organizations

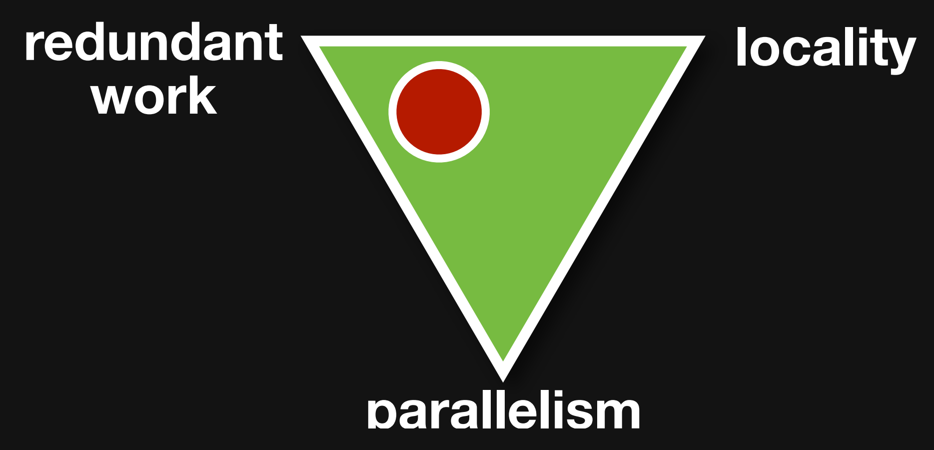



```
blurH.compute_at(blurV, y)
```



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);
}}}}}
```

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

```
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;
```

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

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

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

```
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;
```

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

compute *here*

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

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

```
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;
```

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

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

compute *here*

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

```
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;
```

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

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

Halide

0.25 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(x, y, xi, yi, 256, 32)
       .vectorize(xi, 8).parallel(y);
  blurH.compute_at(blurV, x).store_at(blurV, x).vectorize(x, 8);

  return blurV;
}
```

Halide

0.25 ms/megapixel

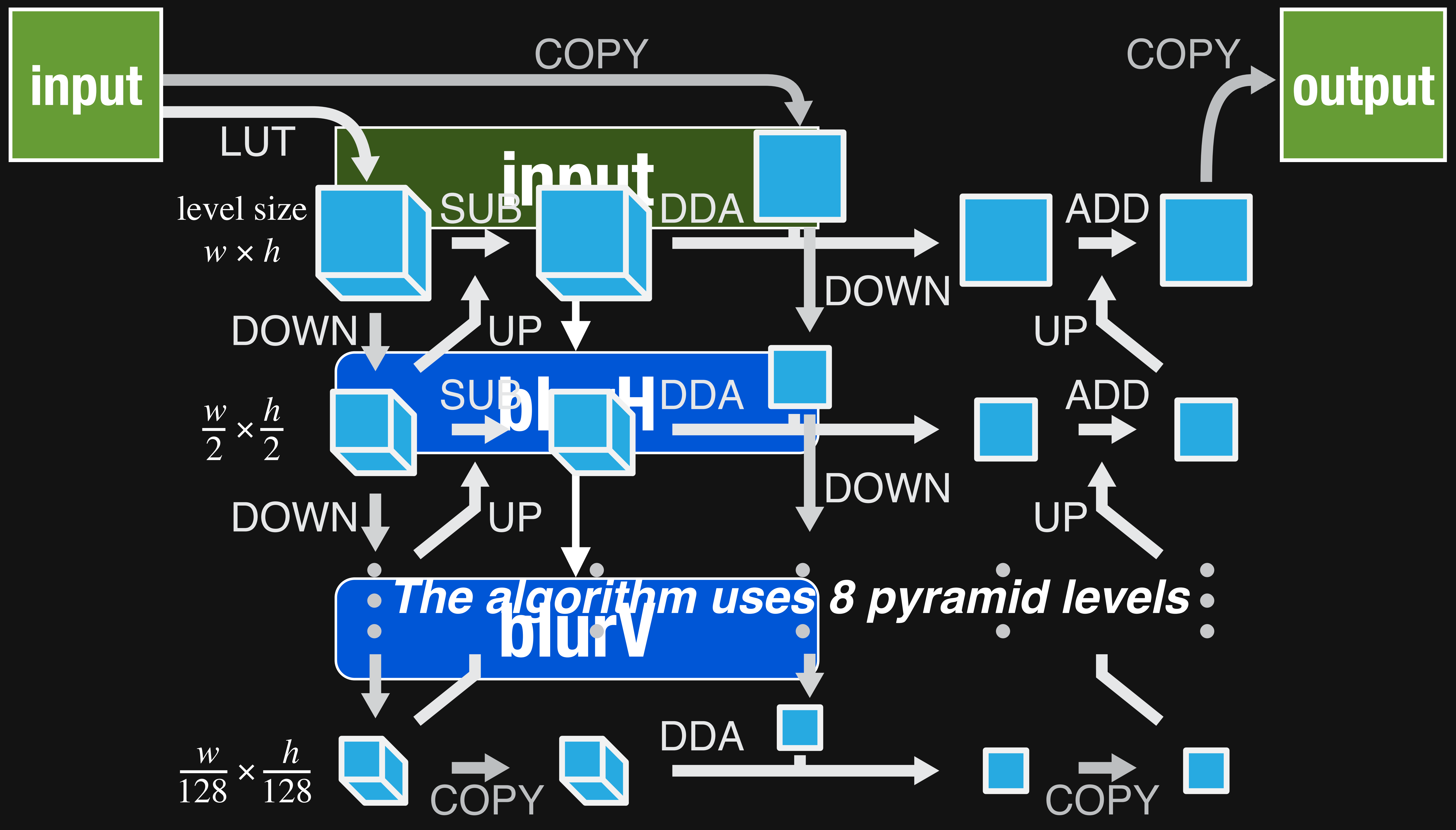
```
Func box_filter_3x3(Func in) {
  Func blurH, blurV;
  Var x, y, xi, yi;

  // 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(x, y, xi, yi, 256, 32)
       .vectorize(xi, 8).parallel(y);
  blurH.compute_at(blurV, x).store_at(blurV, x).vectorize(x, 8);

  return blurV;
}
```

C++

0.25 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
$O(x,y,k) \leftarrow \mathrm{lut}(I(x,y) - k\sigma)$

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]

3x3 box filter

Local Laplacian Filters

in Adobe Photoshop

Adobe: 1500 lines

expert-tuned C++

multi-threaded, SSE

3 months of work

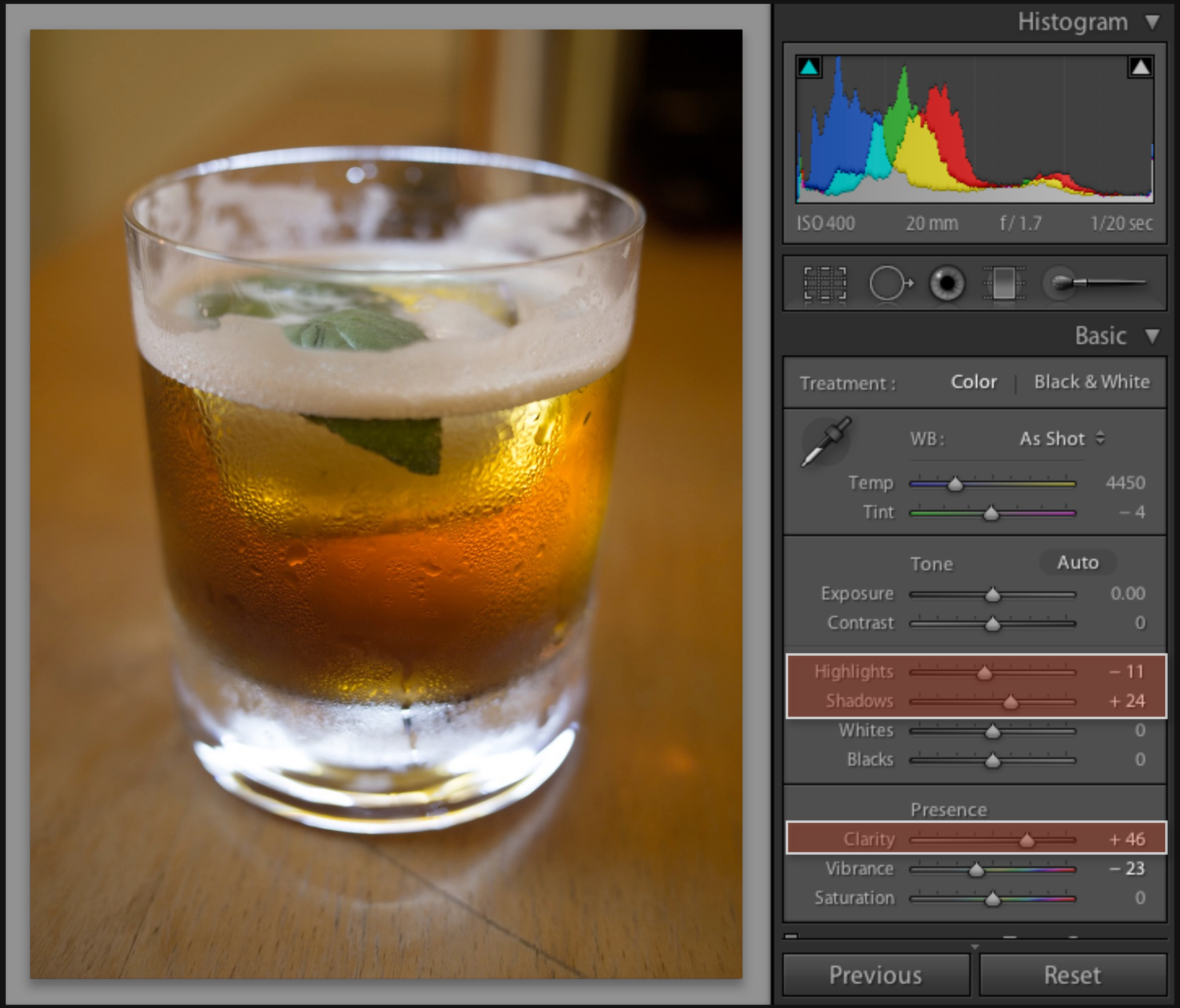
10x faster than original C++

Halide: 60 lines

1 day

Halide vs. Adobe:

2x faster on same **CPU**



Adobe: 1500 lines

expert-tuned C++

multi-threaded, SSE

3 months of work

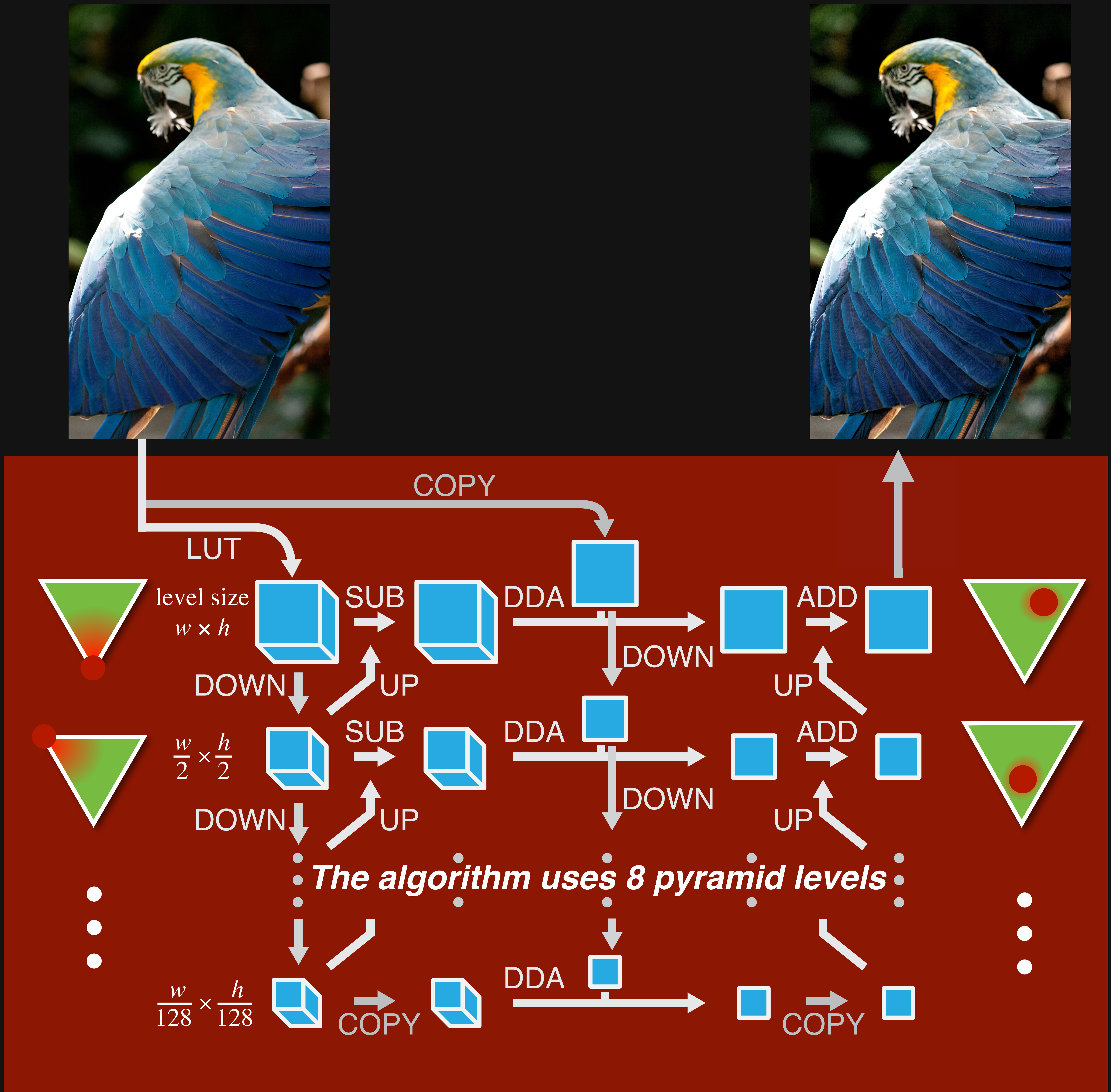
10x faster than original C++

Halide: 60 lines

1 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++

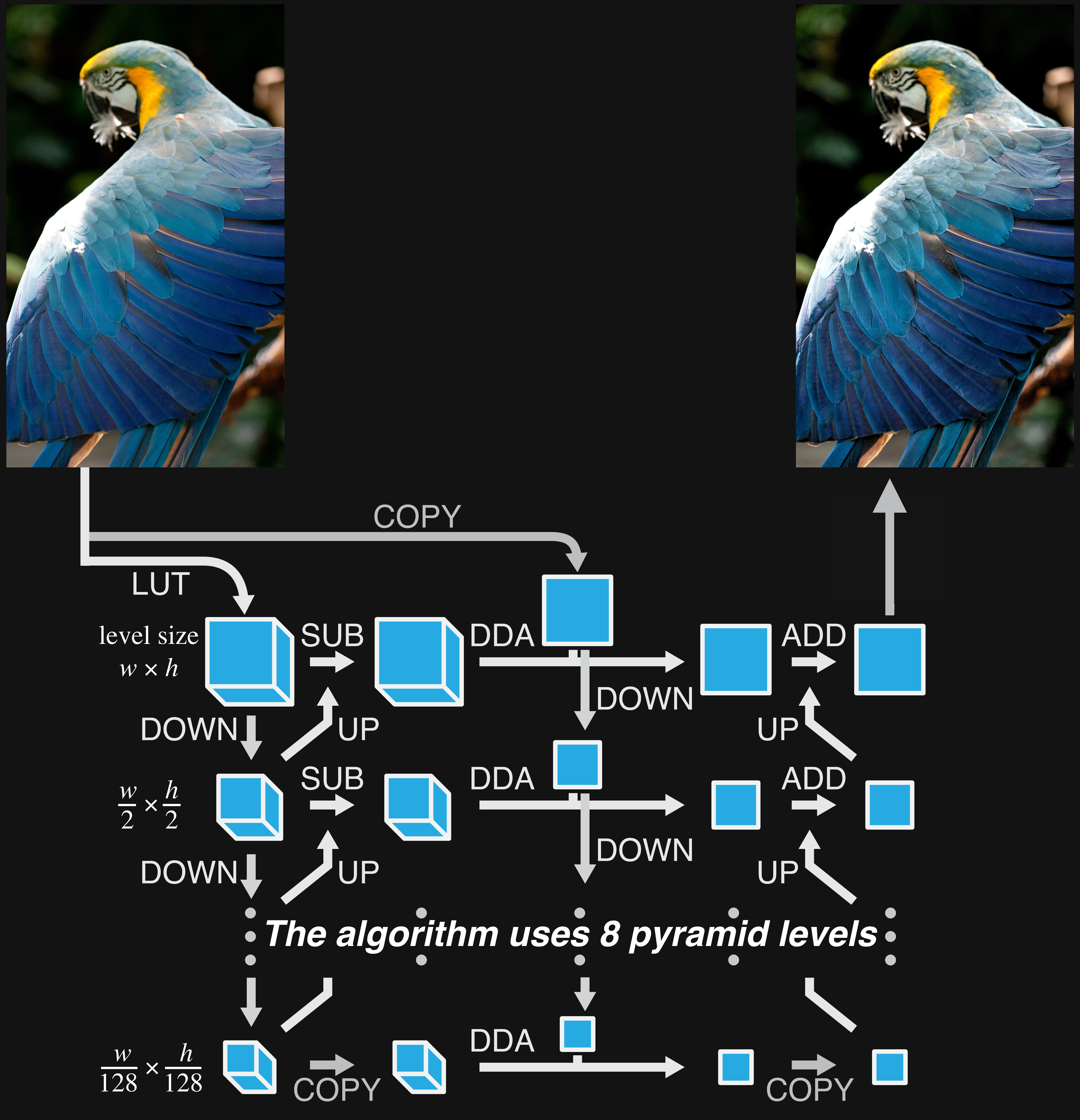
Halide: 60 lines

1 day

Halide vs. Adobe:

2x faster on same **CPU**

10x faster** on **GPU



Real-world adoption

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

Adobe

Insta

Google

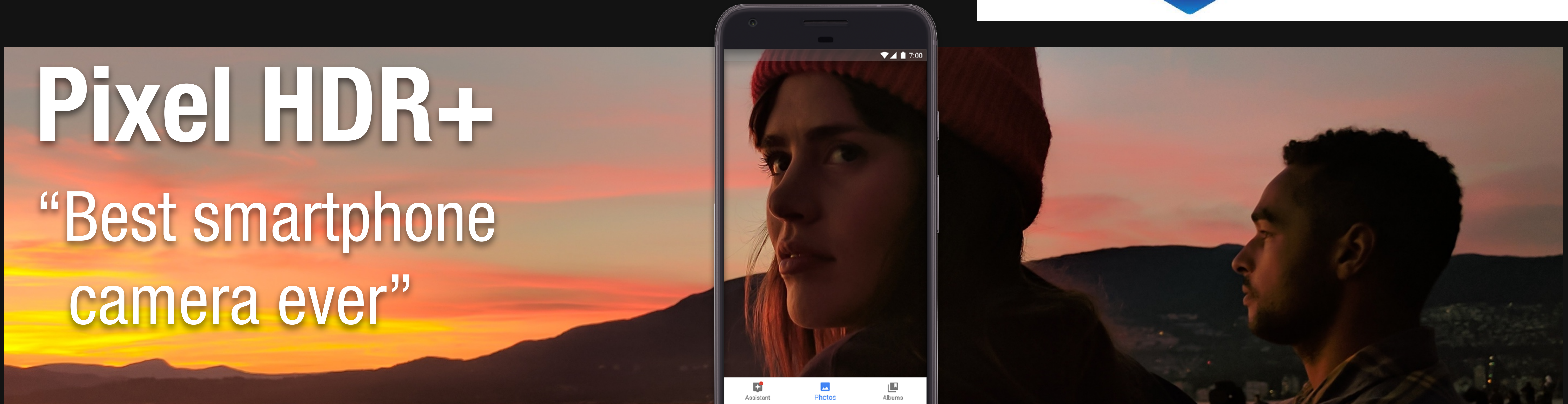
> 2000 pipelines
10s of kLOC in production

YouTube

Tensor Comprehensions

Qualcomm snapdragon

intel



User-schedulable languages
are spreading

HALIDE

tvm

TACO

GraphIt

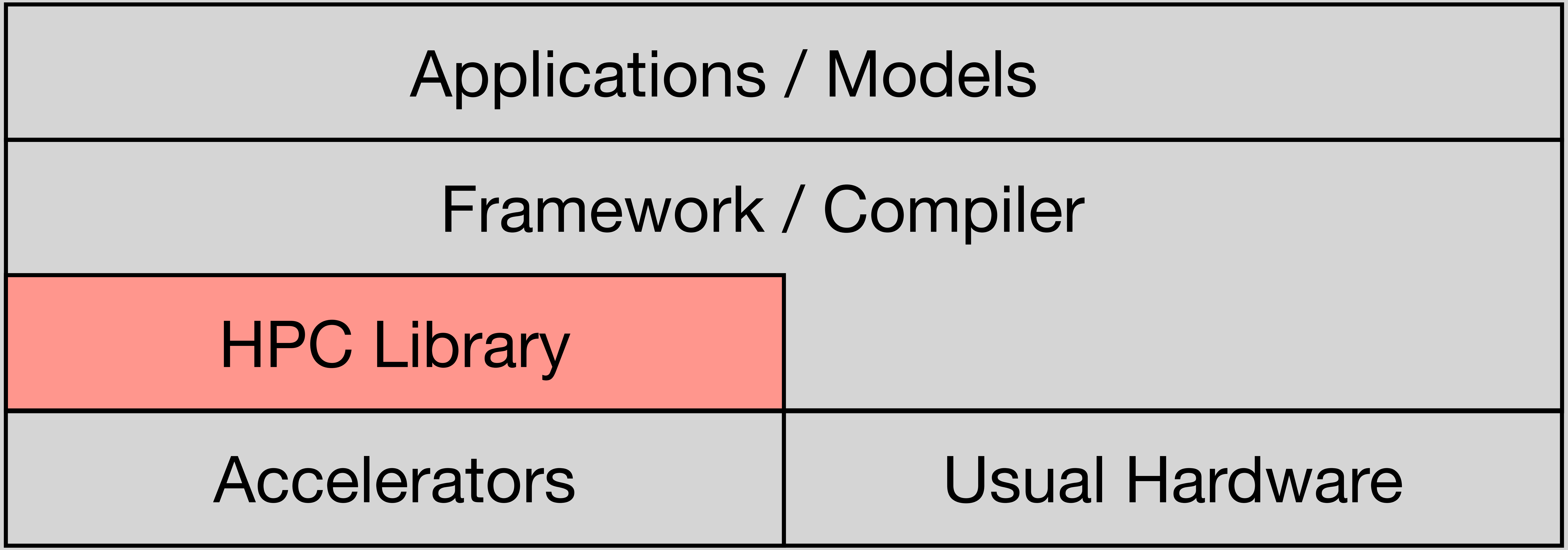
User-Schedulable Languages 2.0

Programming New HW & Verifying Optimizations

[PLDI 2022 + POPL 2022]


Intel MKL
CoreML
BLAS
cuDNN
Neural Engine
GraphCore IPU
NVIDIA GPU
Google TPU
Machine Learning Programmer
Framework/ Compiler Developer
Performance Engineers
Computer Architects

The ML Stack

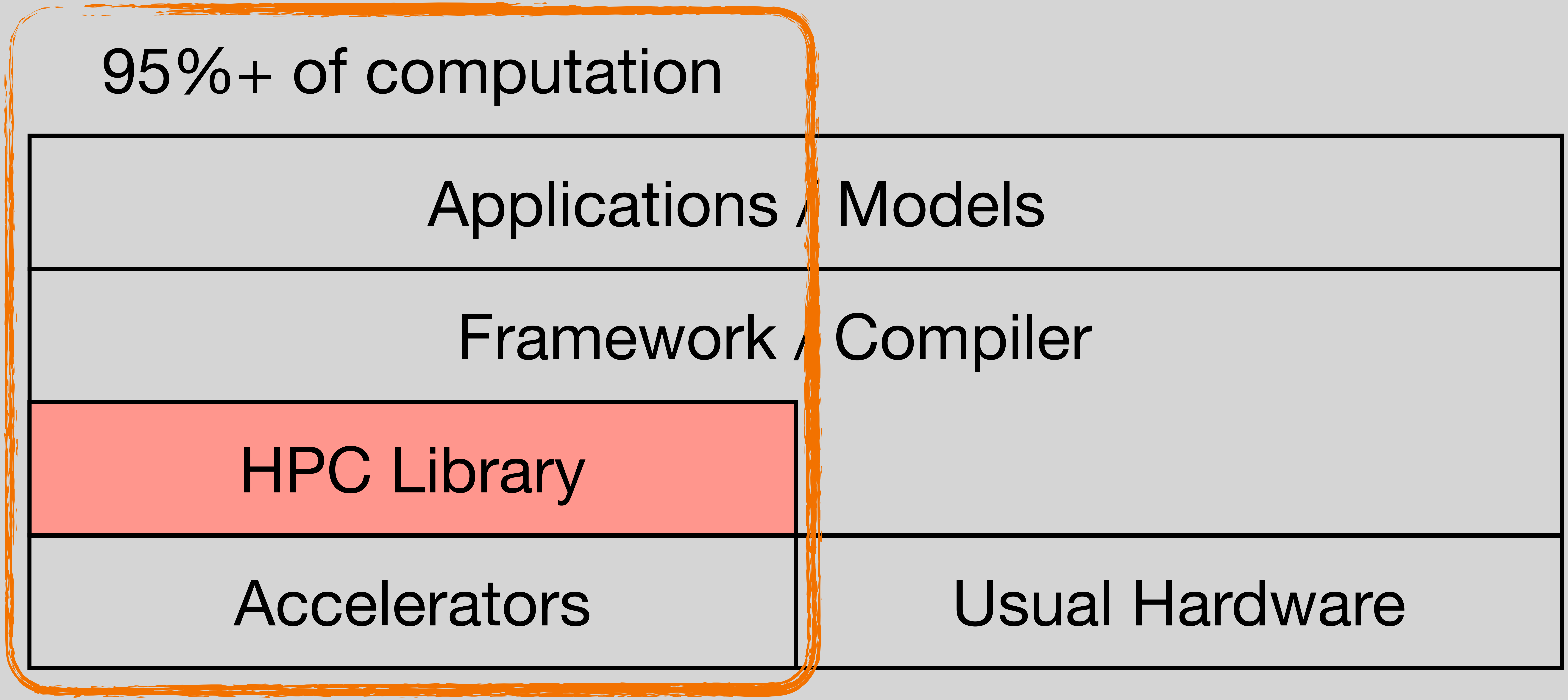

Applications / Models
Framework / Compiler
HPC Library
Accelerators
Usual Hardware

The ML ~~Stack~~ Stovepipe

95%+ of computation

Applications / Models	
Framework / Compiler	
HPC Library	
Accelerators	Usual Hardware

The ML ~~Stack~~ Stovepipe



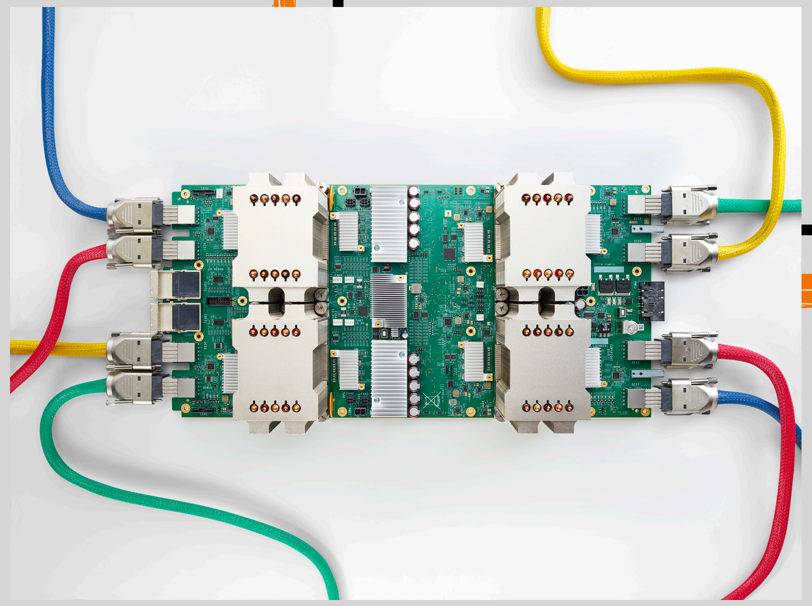
The ML ~~Stack~~ Stovepipe

95%+ of computation

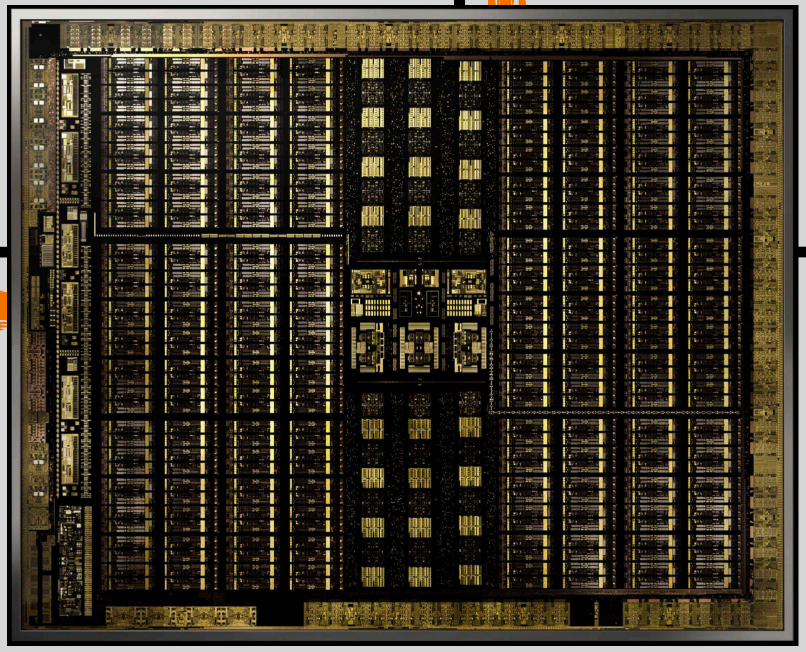
Applications / Models

Framework / Compiler

HPC Library



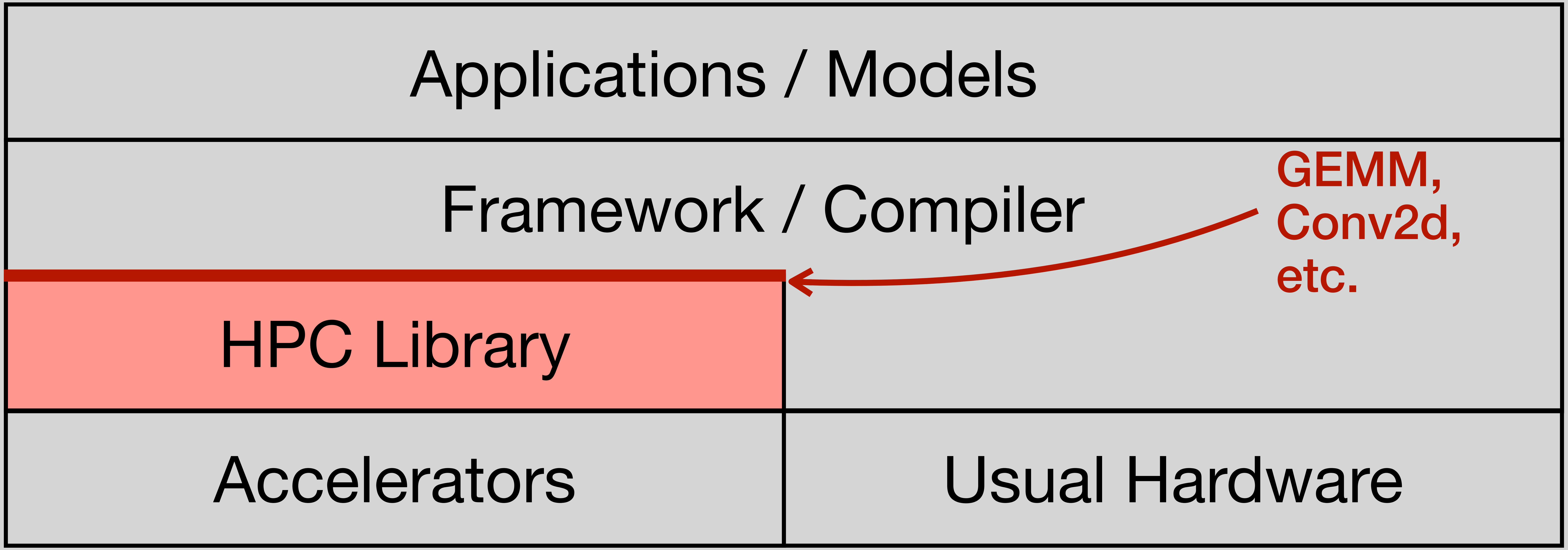
Accelerators



Usual Hardware

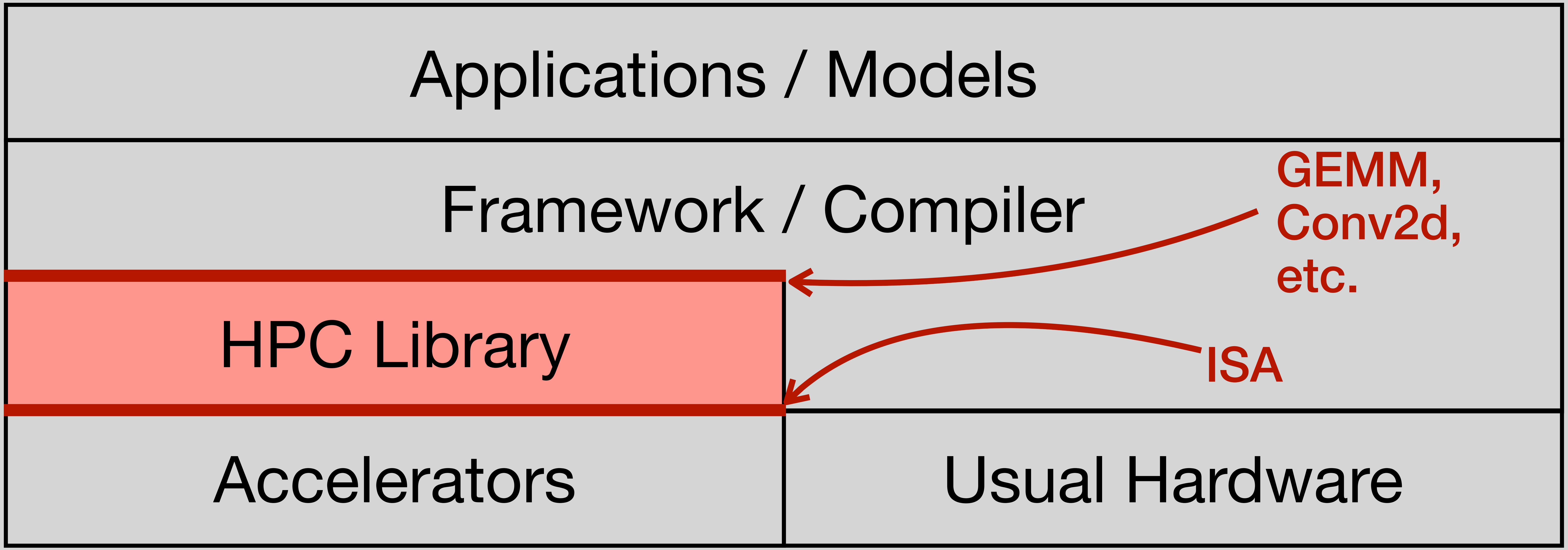
The ML ~~Stack~~ Stovepipe

Consequence 1: Portability Interface



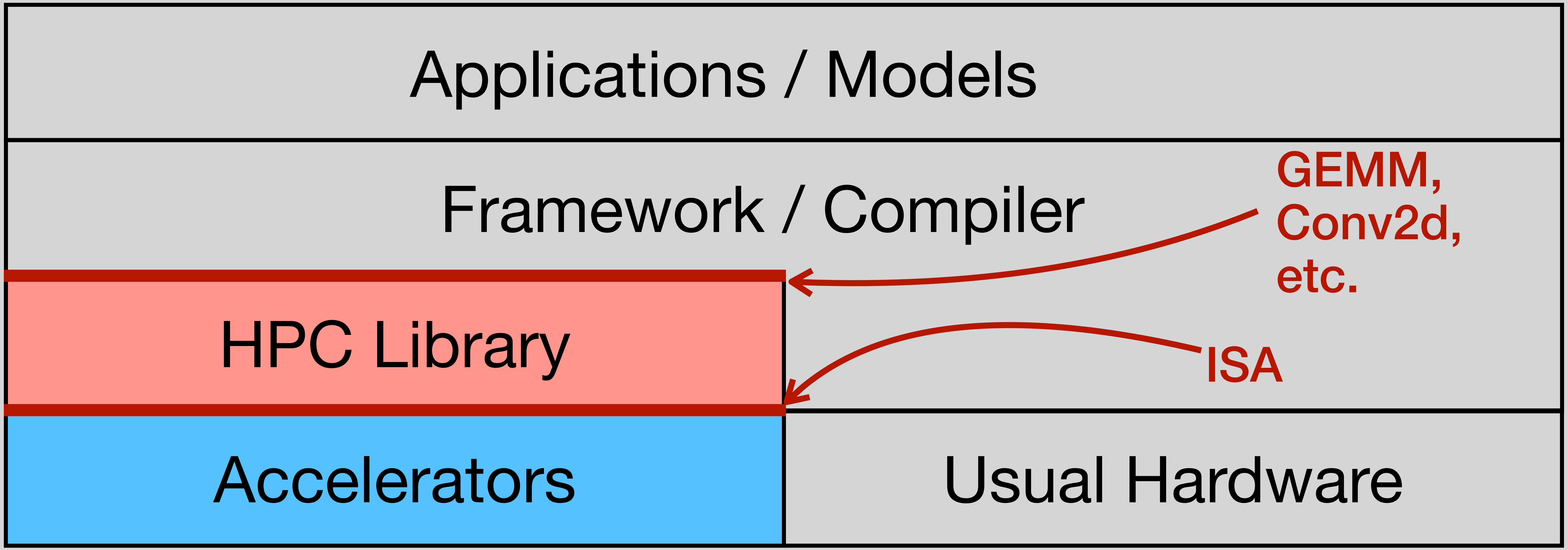
The ML ~~Stack~~ Stovepipe

Consequence 1: Portability Interface

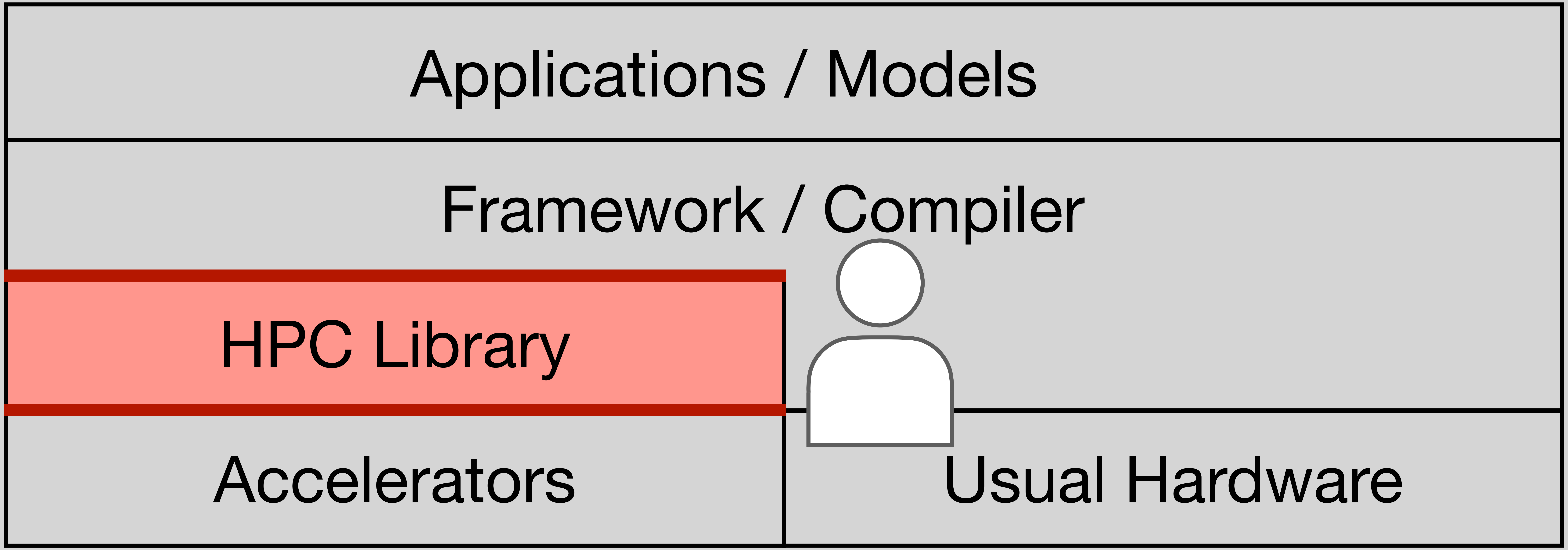


The ML ~~Stack~~ Stovepipe

Consequence 2: HPC Libs are on “Critical Path”



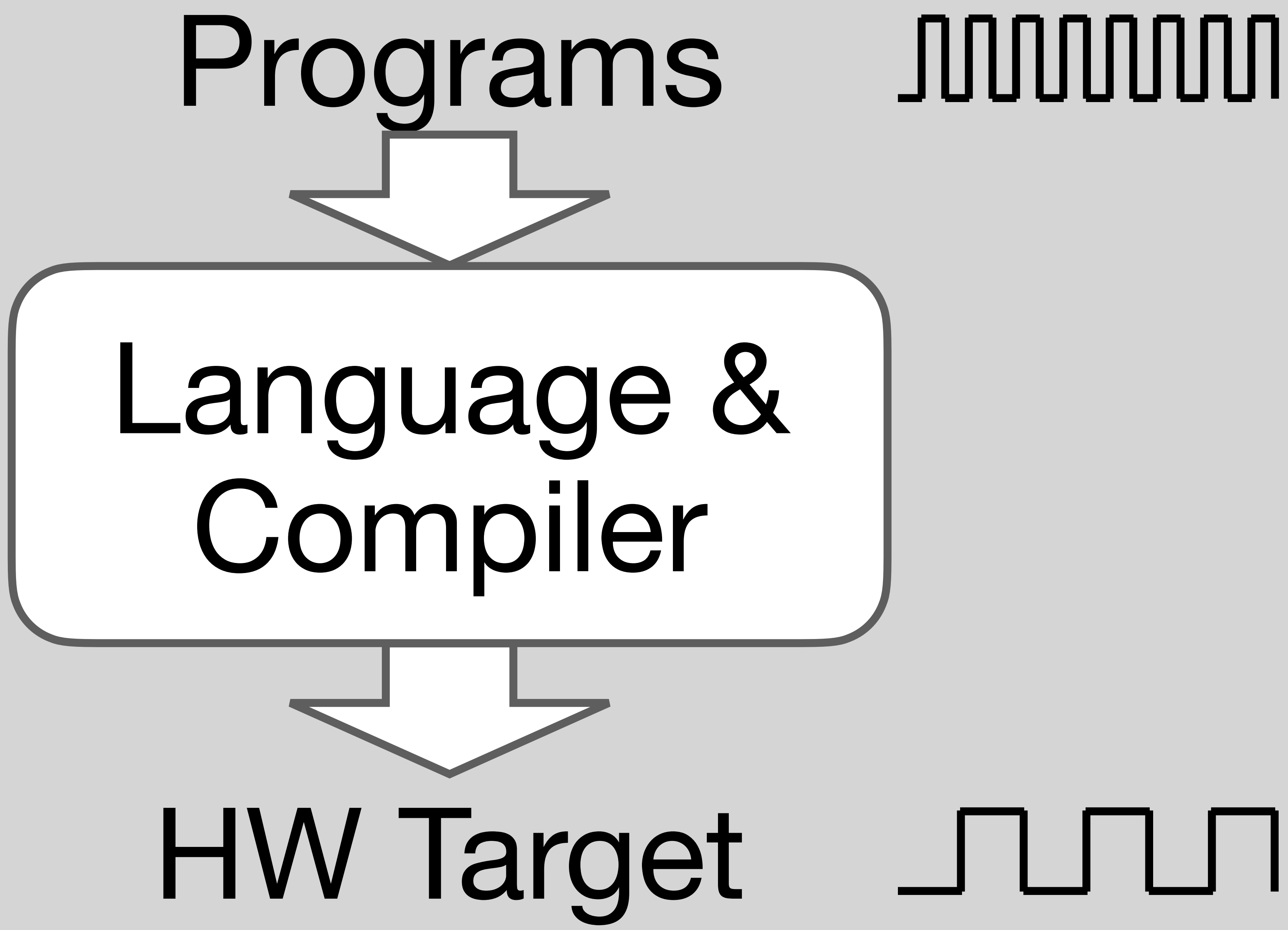
The ML ~~Stack~~ Stovepipe



What’s Special About Writing HPC Libraries?

What’s Special About Writing HPC Libraries?

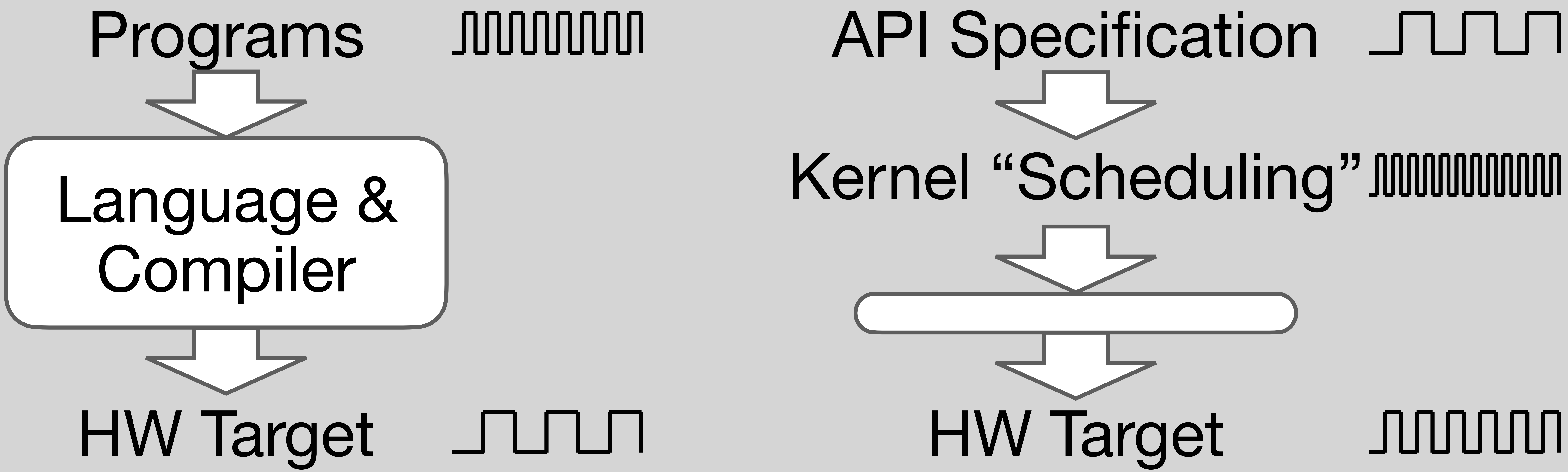
Usual Programs



What’s Special About Writing HPC Libraries?

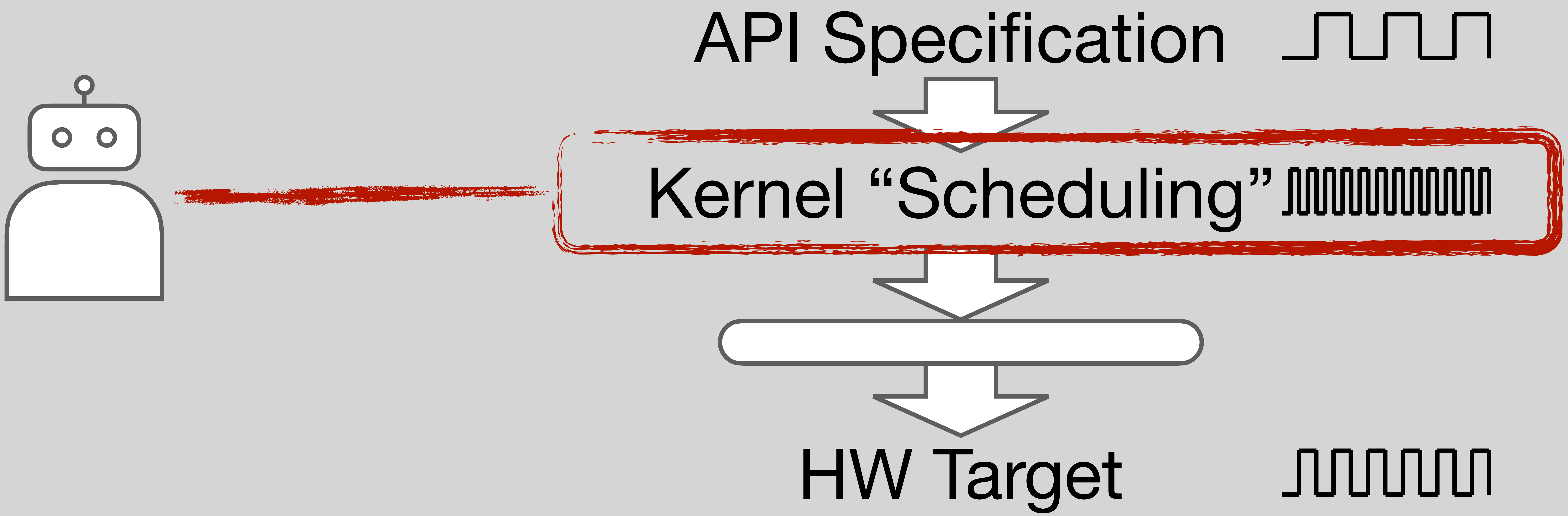
Usual Programs

HPC Libraries

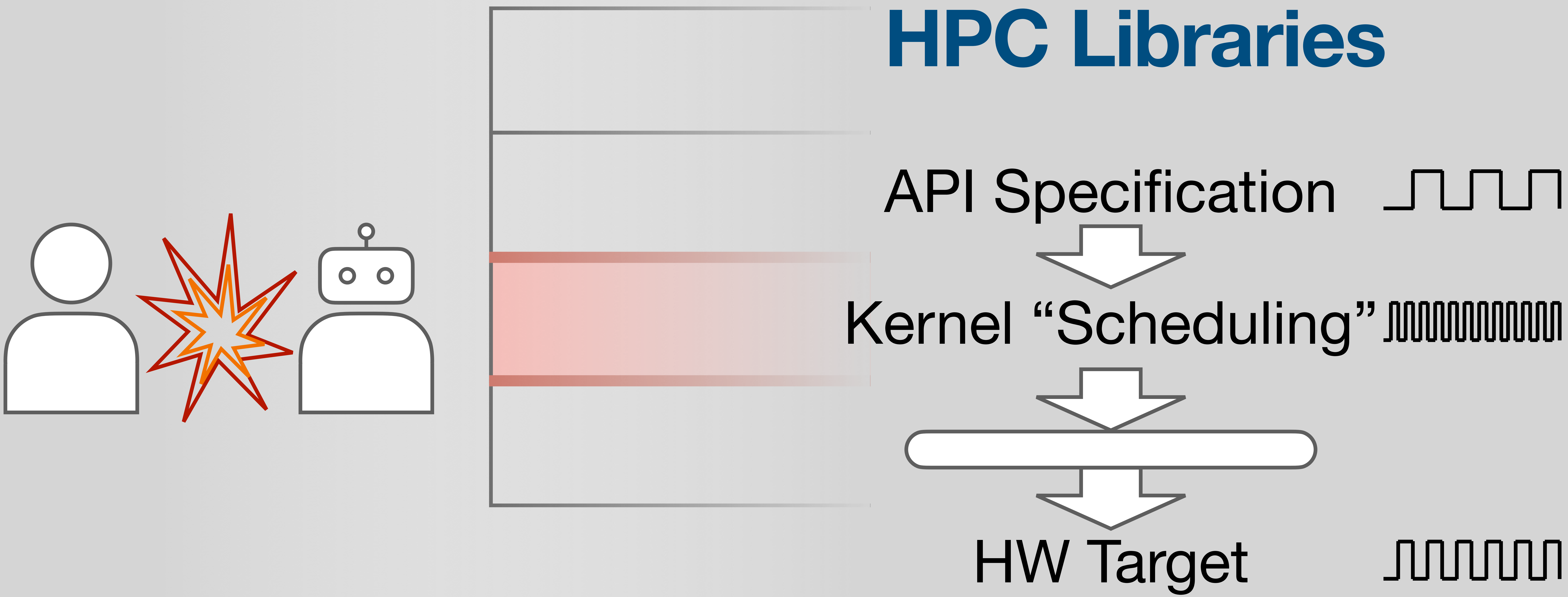


The Trouble With Automation

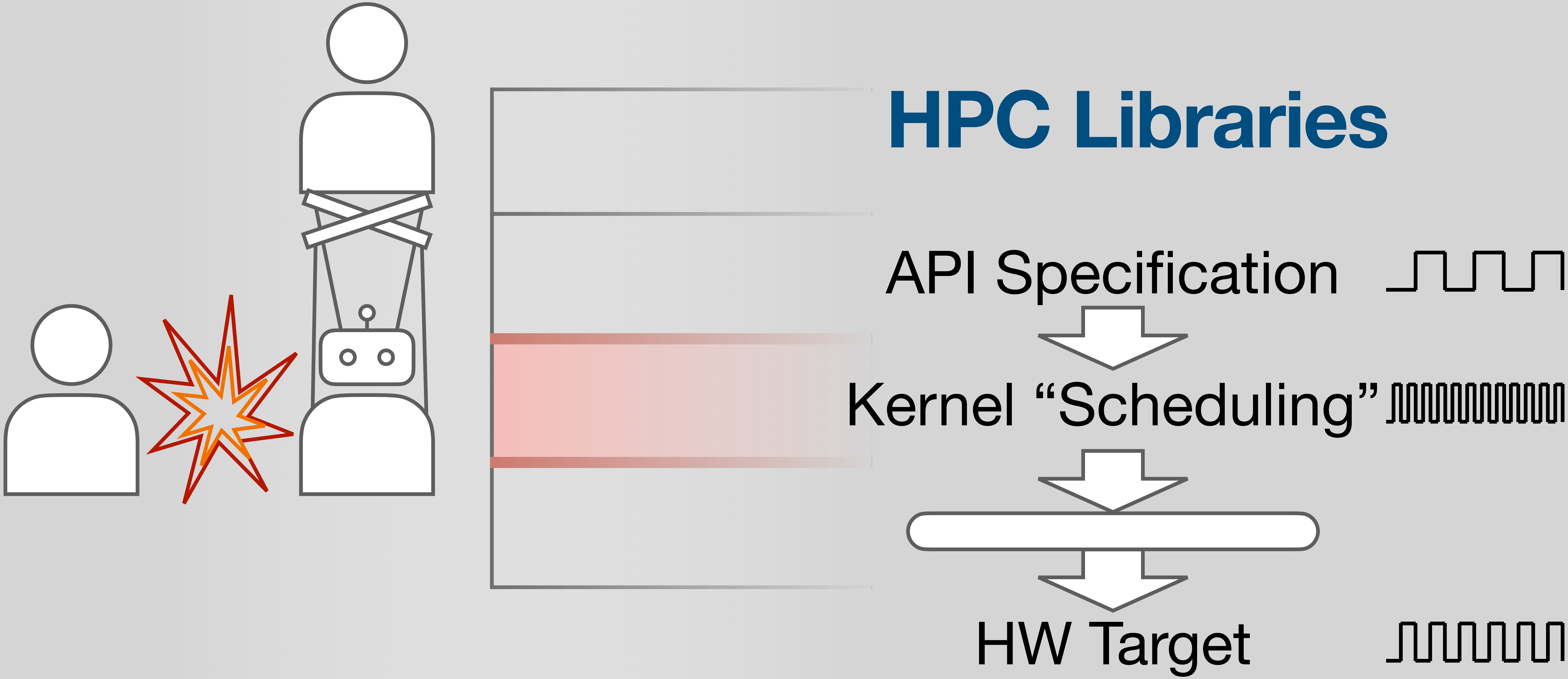
HPC Libraries



The Trouble With Automation



The Trouble With Automation



Exocompilation

a compiler/language design that *externalizes* parts of the compiler in order to give programmers more control

Exocompilation

a compiler/language design that *externalizes* parts of the compiler in order to give programmers more control

HW Targets as Libraries

User Scheduling

Exo

Exo code

```
def gemm(N : size, M : size, K : size,
         A : f32[N,K], B : f32[K,M], C : f32[N,M]
):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]
```

“basically C”

Performance Gains from Scheduling

```
def gemm(N : size, M : size, K : size,
         A : f32[N,K], B : f32[K,M], C : f32[N,M]
):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]
```

0.94% of peak throughput

Naive Matrix Multiply

88.1% of peak throughput!

Matrix Multiply
Optimized for AVX-512

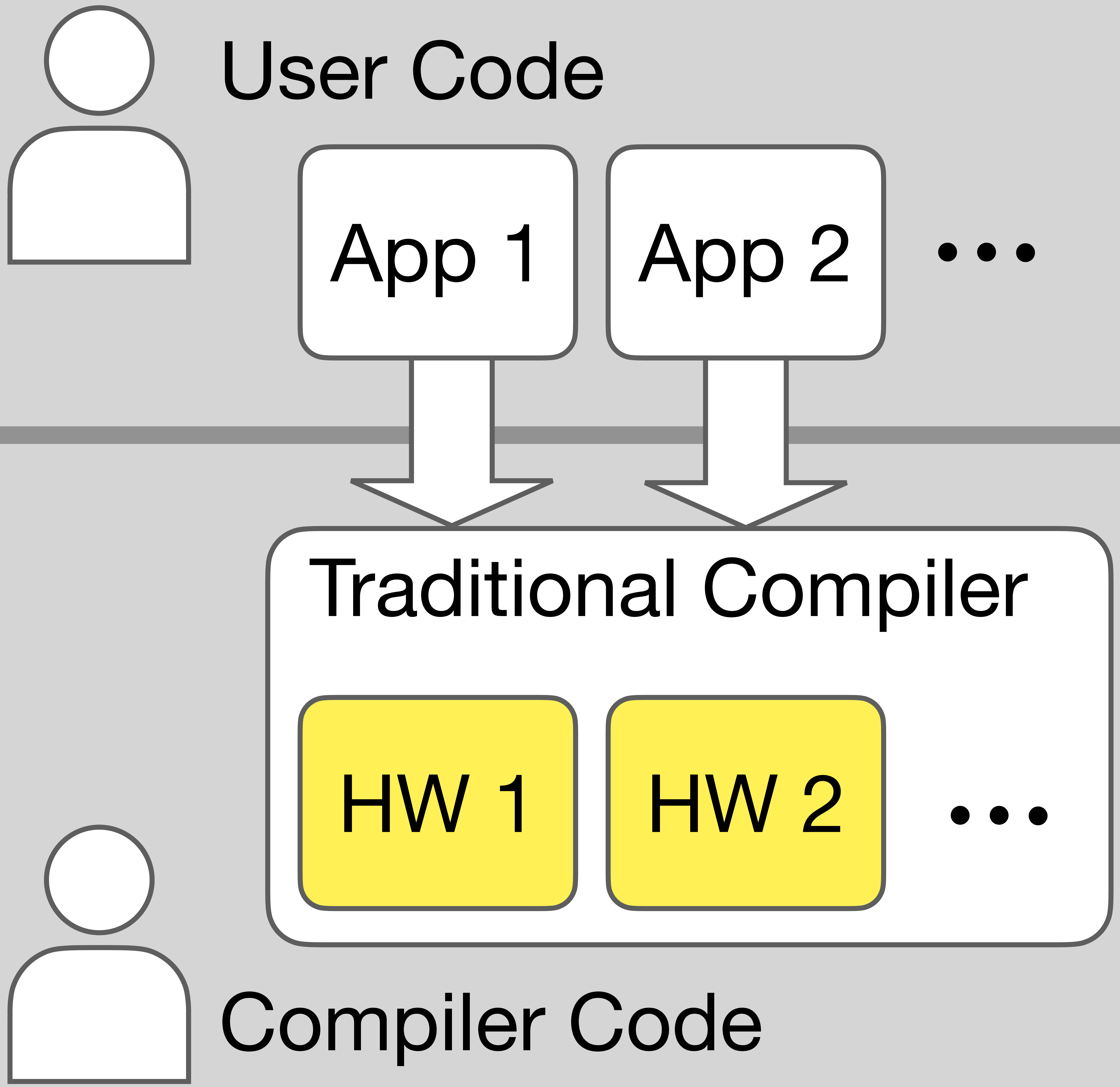
Exocompilation

a compiler/language design that *externalizes* parts of the compiler in order to give programmers more control

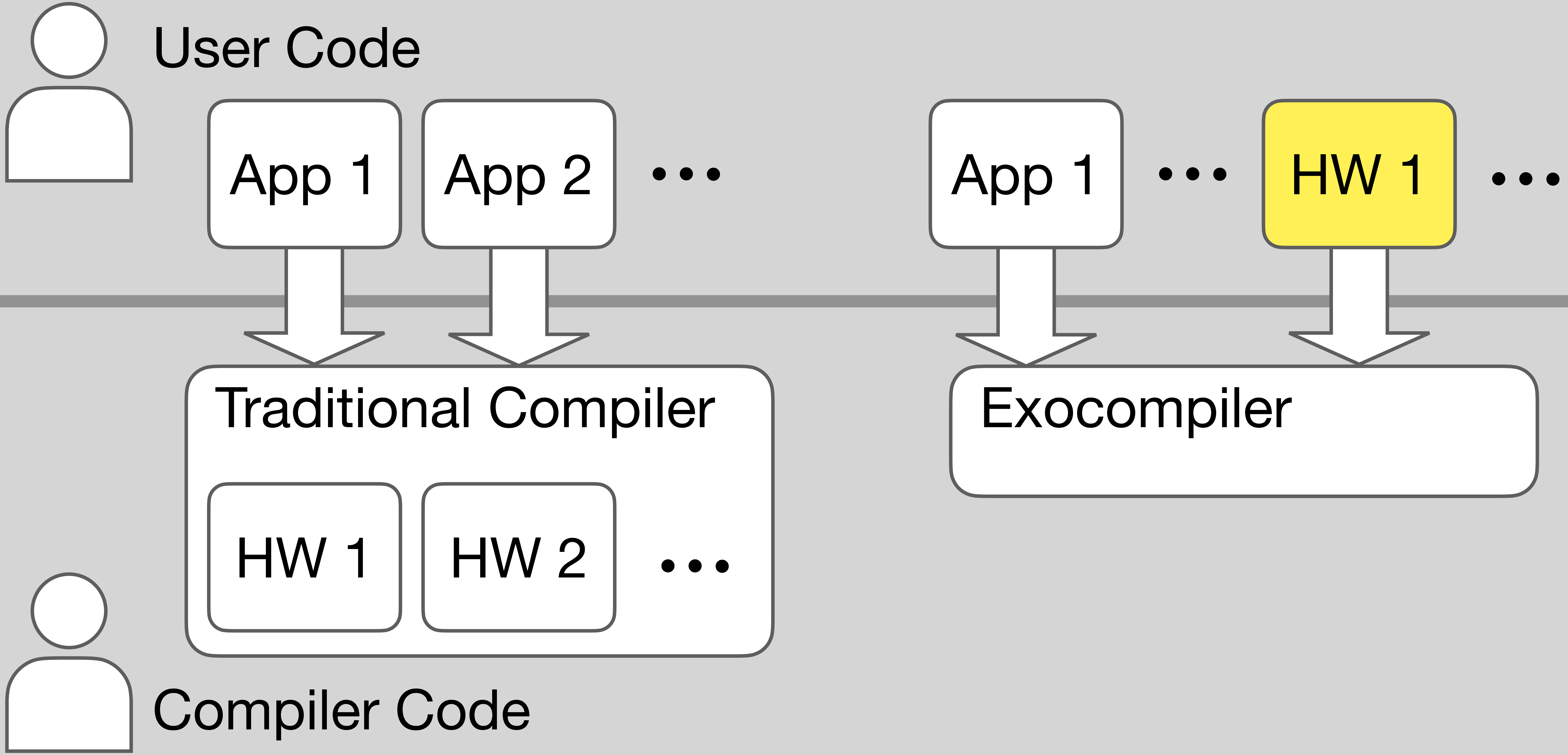
HW Targets as Libraries

User Scheduling

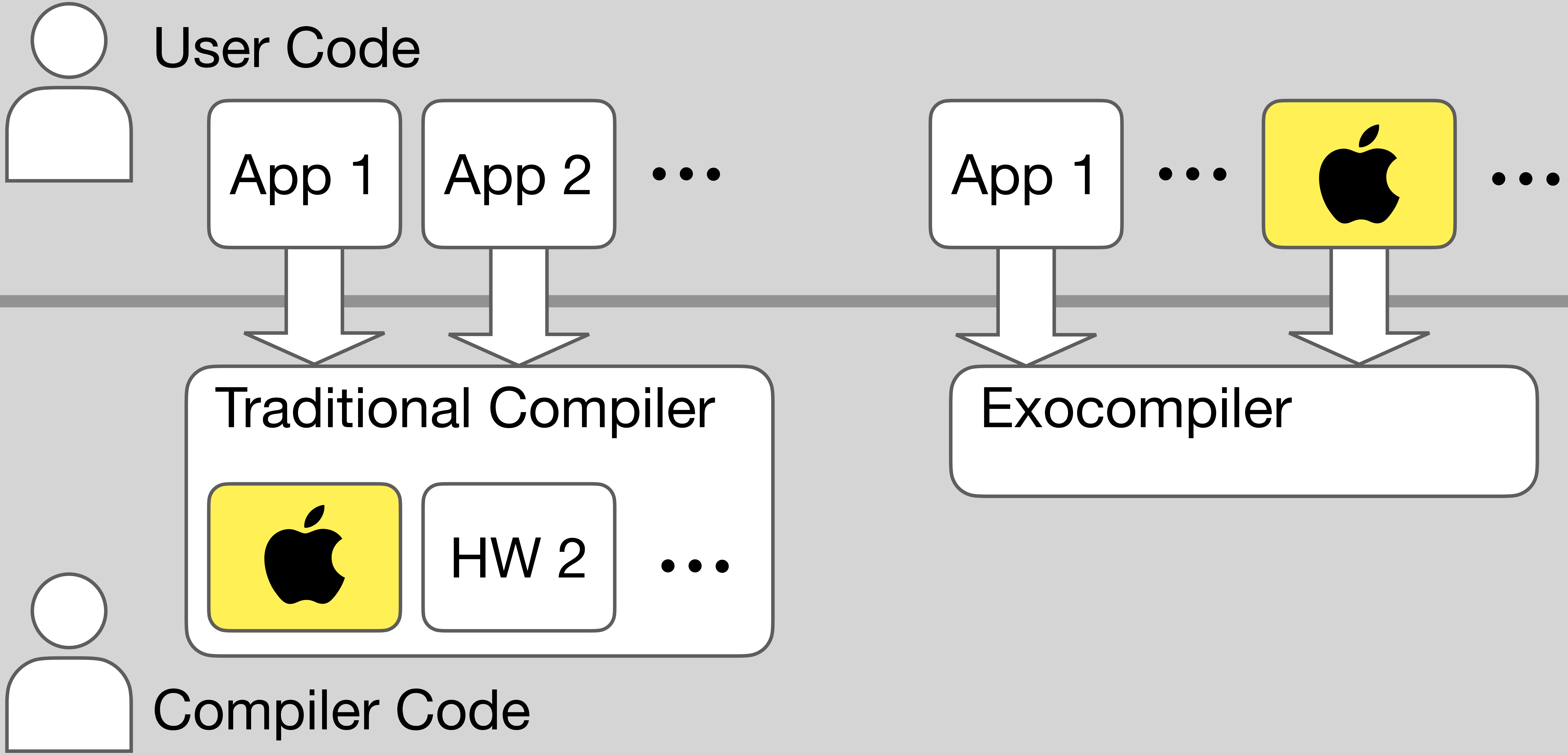
HW Backends as User-Level Libraries



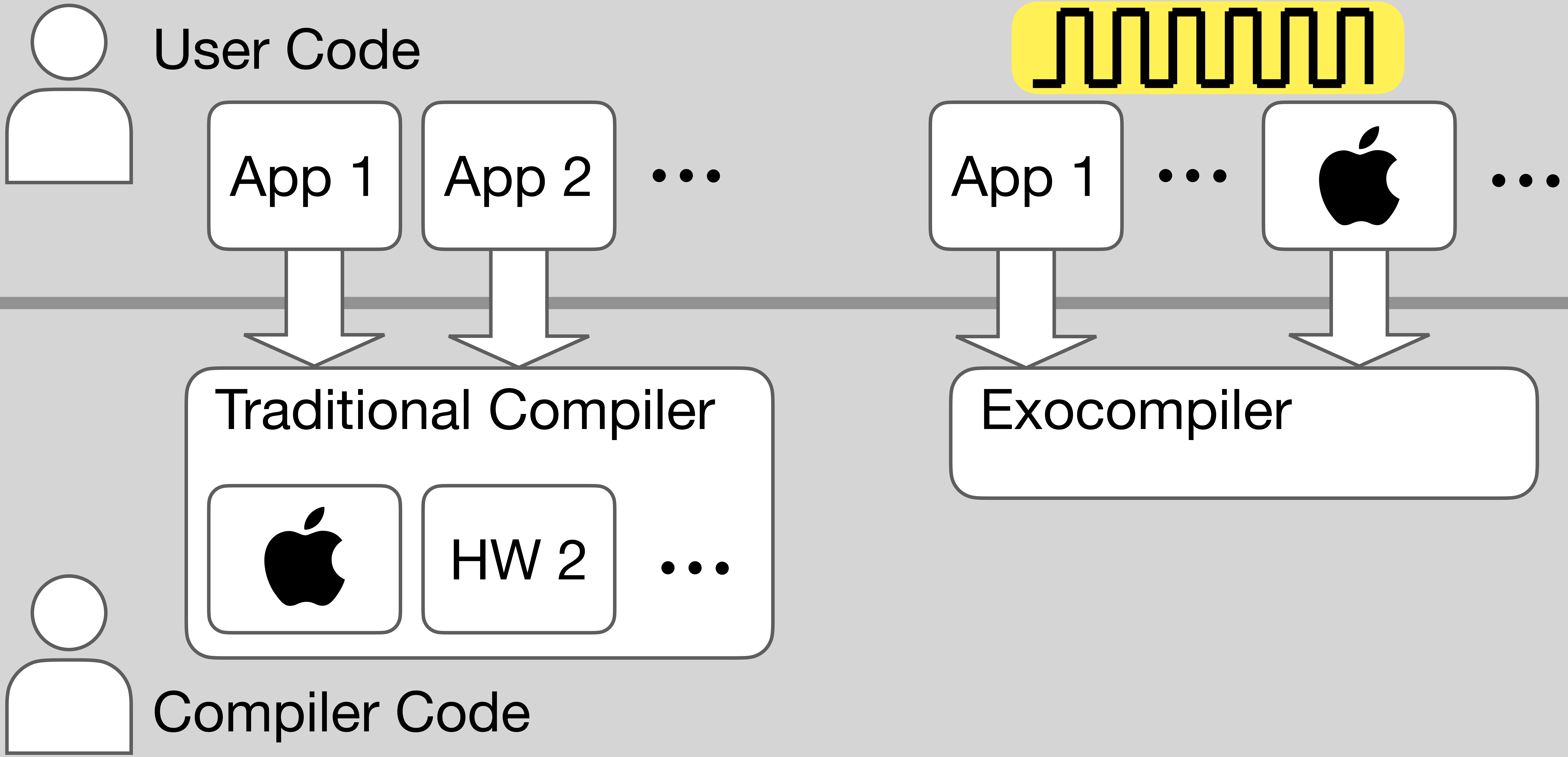
HW Backends as User-Level Libraries



HW Backends as User-Level Libraries



HW Backends as User-Level Libraries

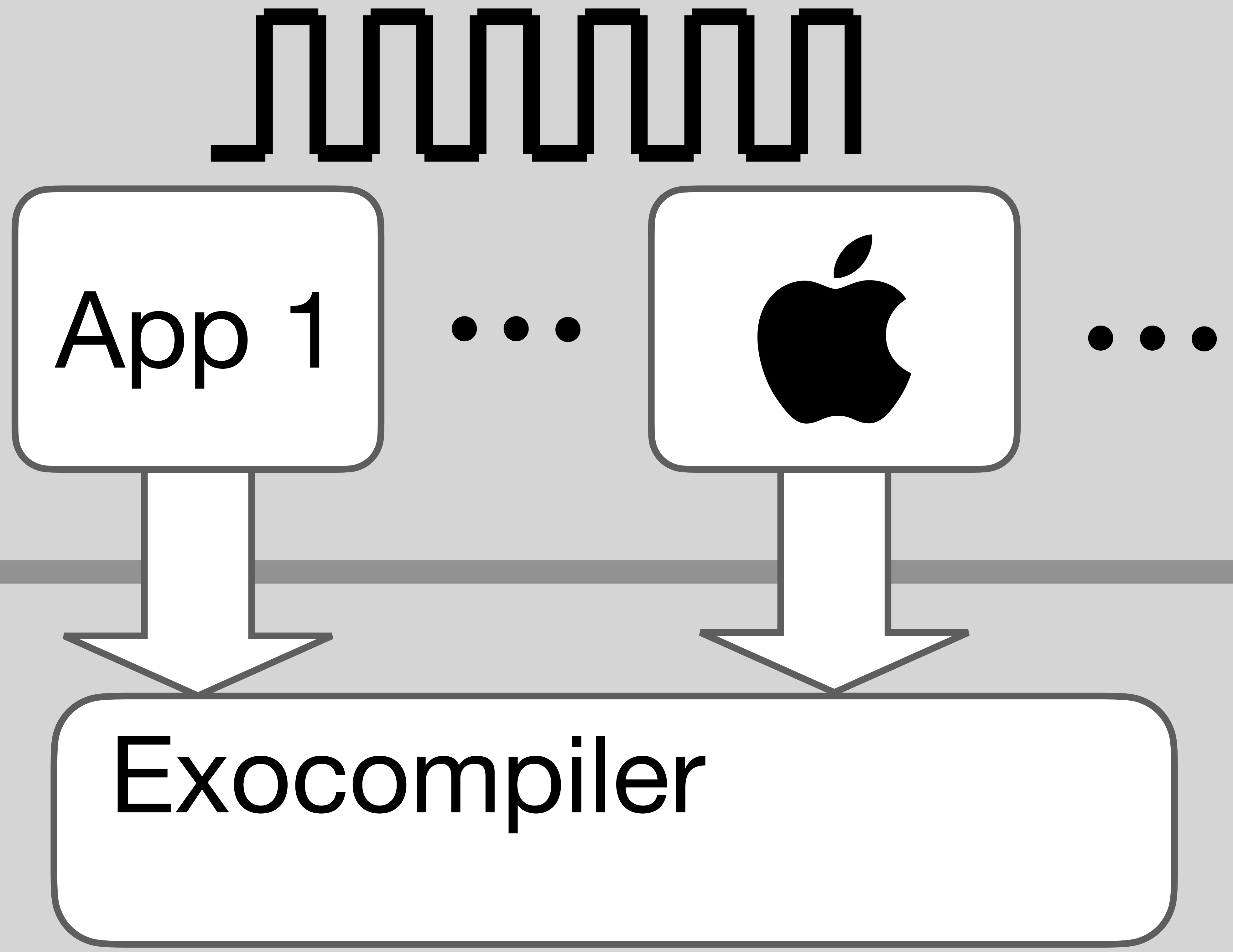


HW Backends as User-Level Libraries

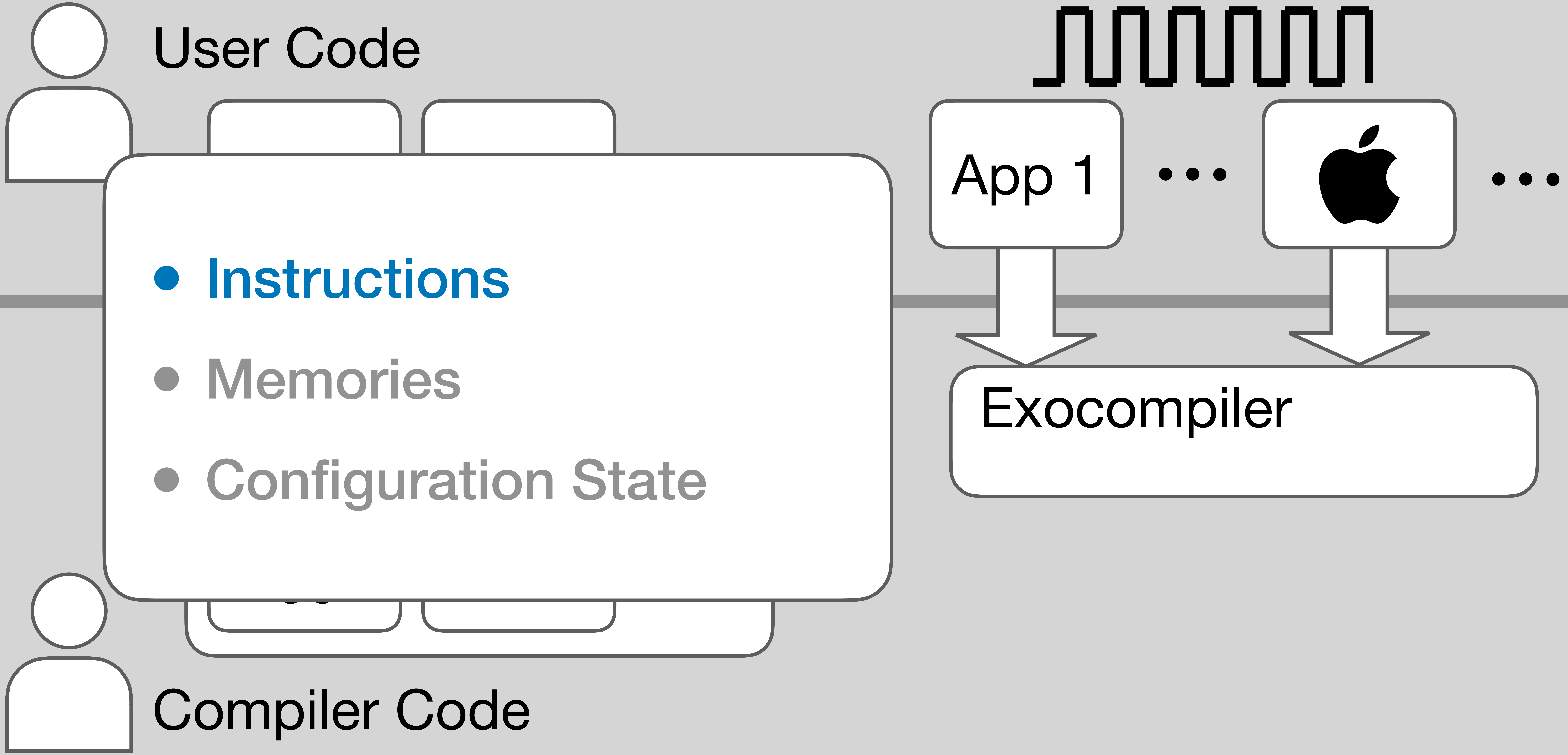
User Code

- Instructions
- Memories
- Configuration State

Compiler Code



HW Backends as User-Level Libraries



Instructions are Procedures

- Instructions
- Memories
- Configuration State

```
def main(…):
    …
    instr_A(…)
    …
```

Calls

```
@instr("ABCD")
def instr_A(…):
    …
```

Instructions are Procedures

- Instructions
- Memories
- Configuration State

```
def main(…):
    …
    instr_A(…)
    …
```

```
@instr(“ABCD”)
def instr_A(…):
    …
```

Generate Code

```
void main(…) {
    …
    ABCD
    …
}
```

Instructions are Procedures

```
def main(…):
    …
    instr_A(…)
    …
```

- Instructions
- Memories
- Configuration State

```
@instr("ABCD")
def instr_A(…):
    …
```

Annotation

Procedure

Exocompilation

a compiler/language design that *externalizes* parts of the compiler in order to give programmers more control

HW Targets as Libraries

User Scheduling

User Scheduling

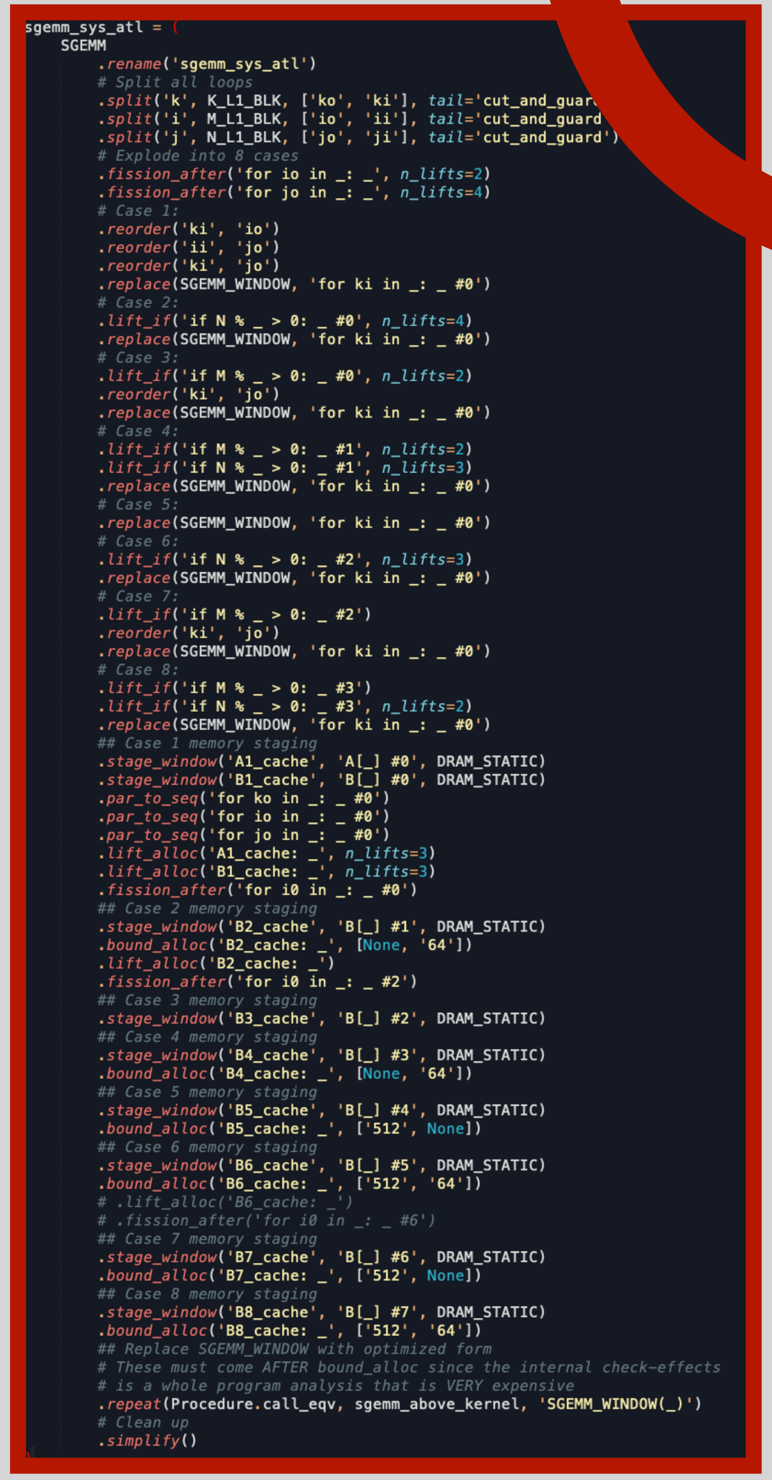
```
def gemm(N : size, M : size, K : size,
         A : f32[N,K], B : f32[K,M], C : f32[N,M]
):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]
```

User Scheduling

```
def gemm(N : size, M : size, K : size,
         A : f32[N,K], B : f32[K,M], C : f32[N,M]
):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]
```

User Scheduling

```
def gemm(N : size, M : size, K : size,
         A : f32[N,K], B : f32[K,M], C : f32[N,M]
):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]
```



User Scheduling

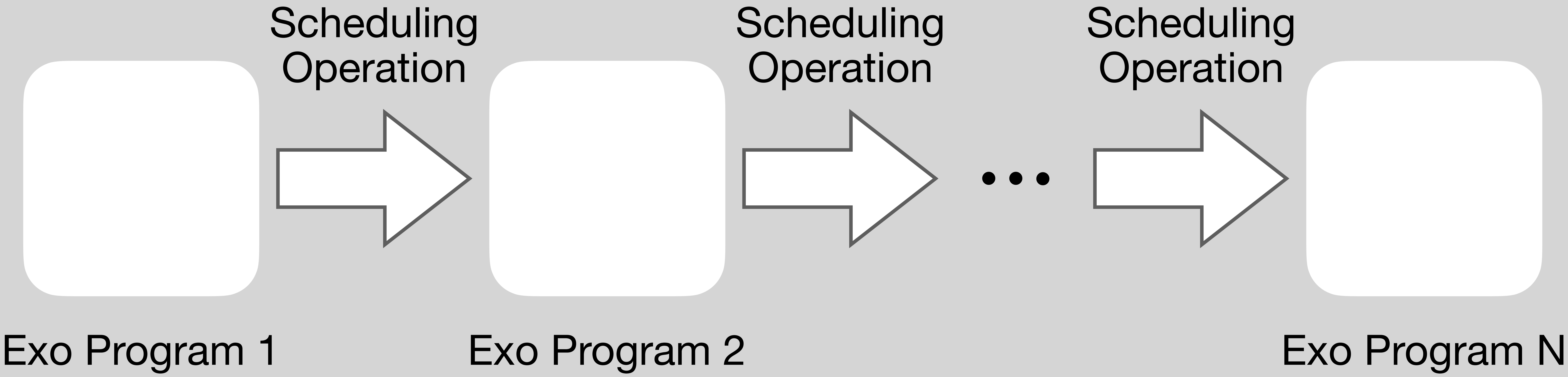
A new approach

Scheduling as Rewriting, not Annotation

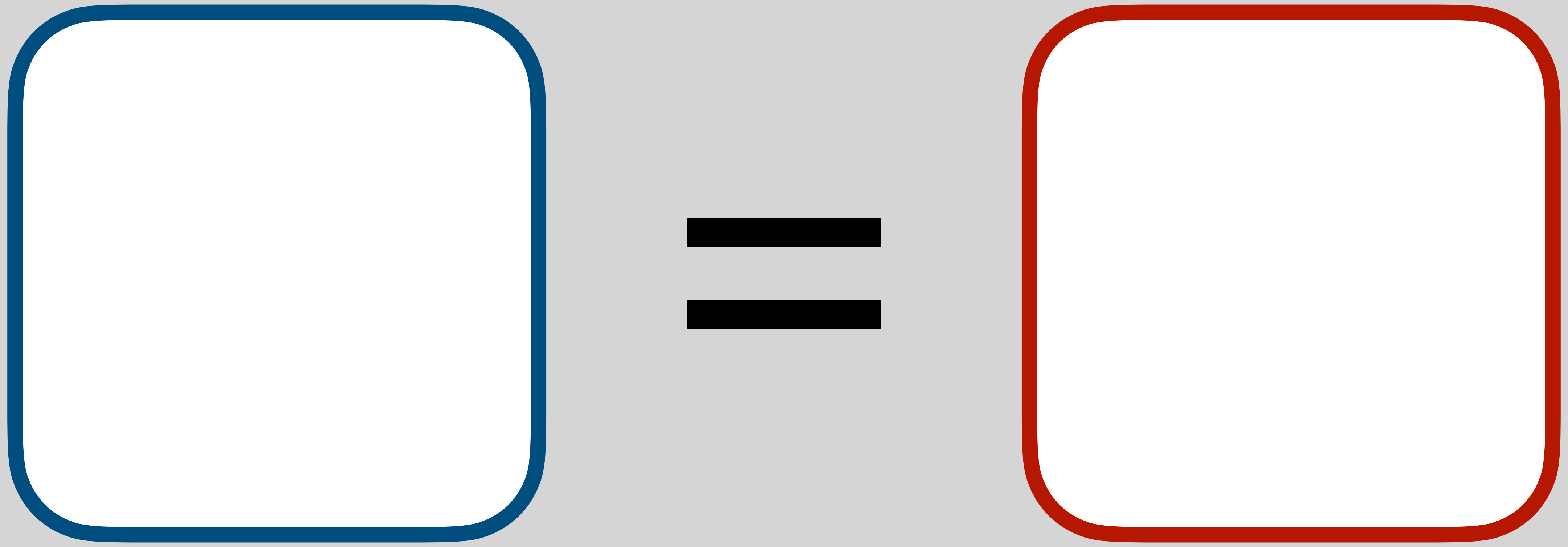
Scheduling of Sub-procedures

```
def gemm(N : size, M : size, K : size,
         A : f32[N,K], B : f32[K,M], C : f32[N,M]
):
    for i in seq(0,N):
        for j in seq(0,M):
            for k in seq(0,K):
                C[i,j] += A[i,k] * B[k,j]
```

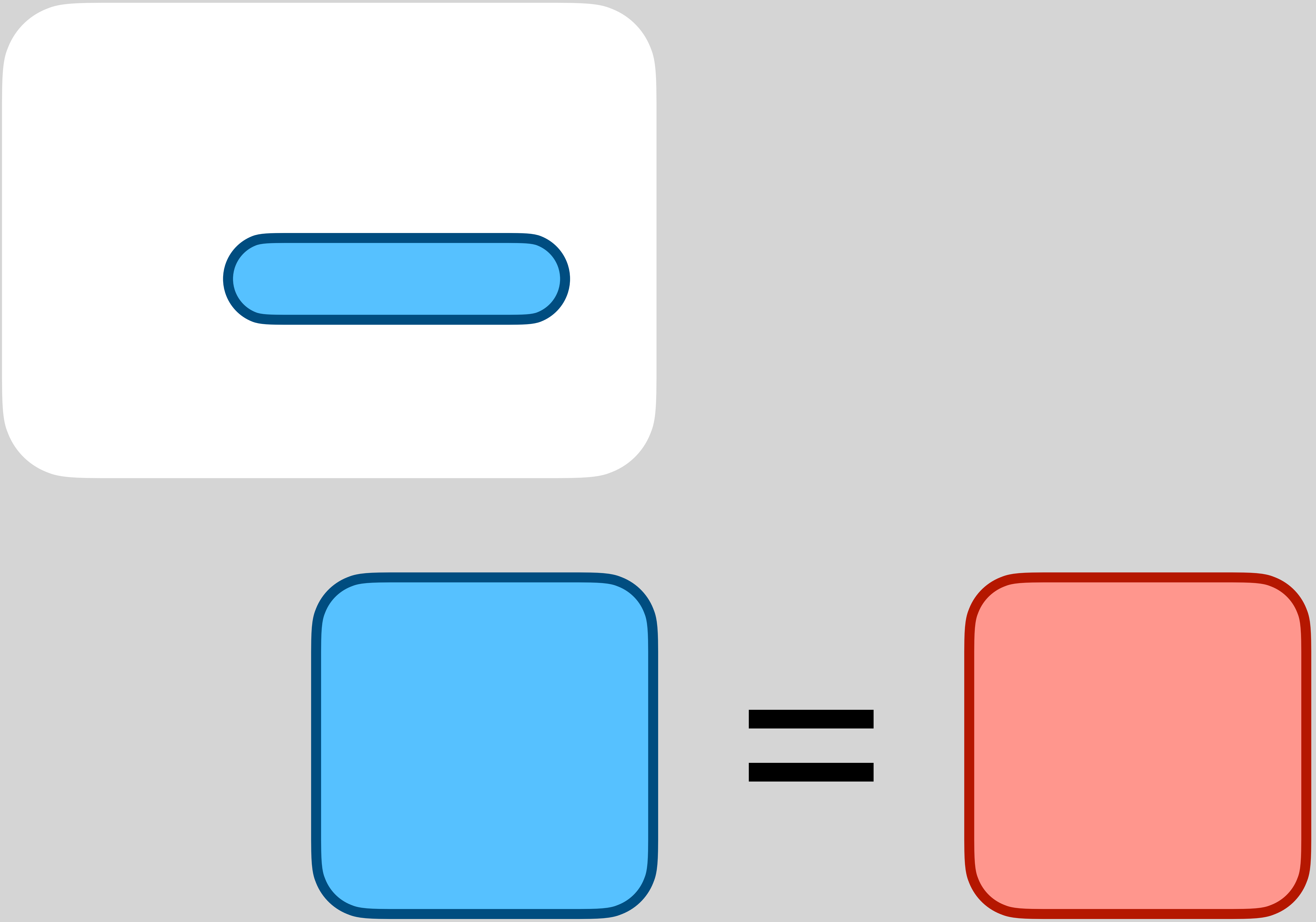
Scheduling as Program Rewriting



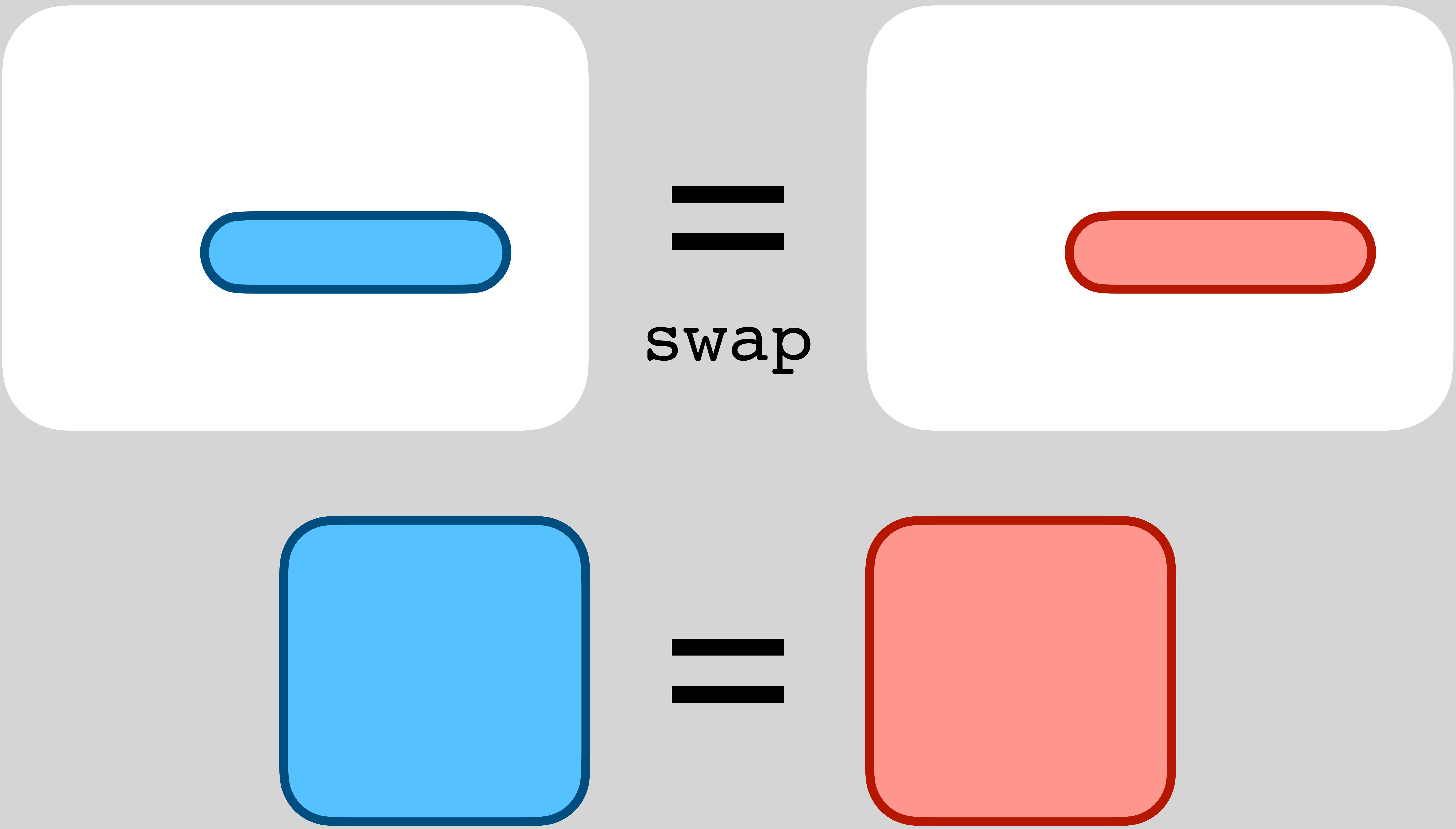
Scheduling as Program Rewriting



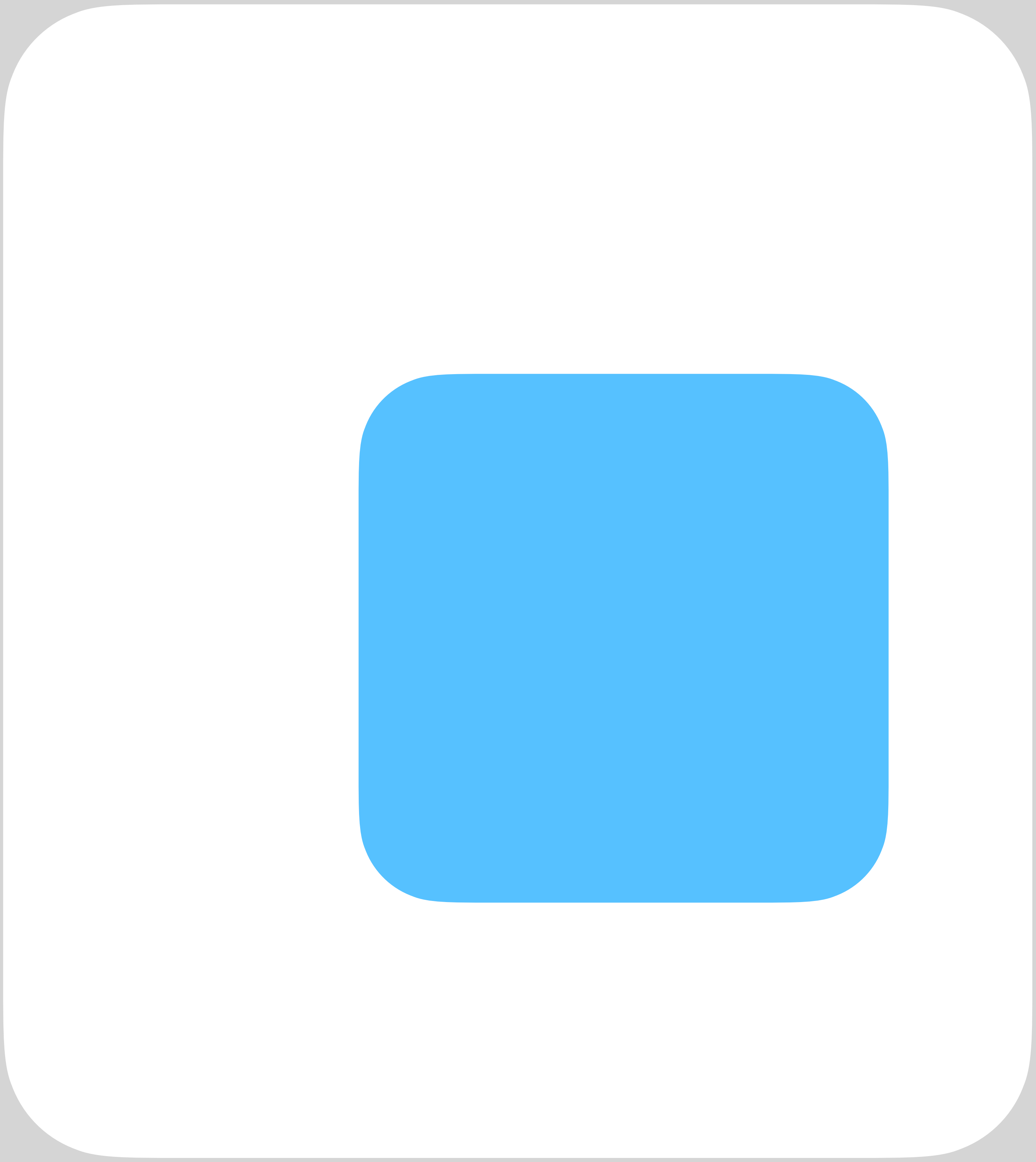
Scheduling as Program Rewriting



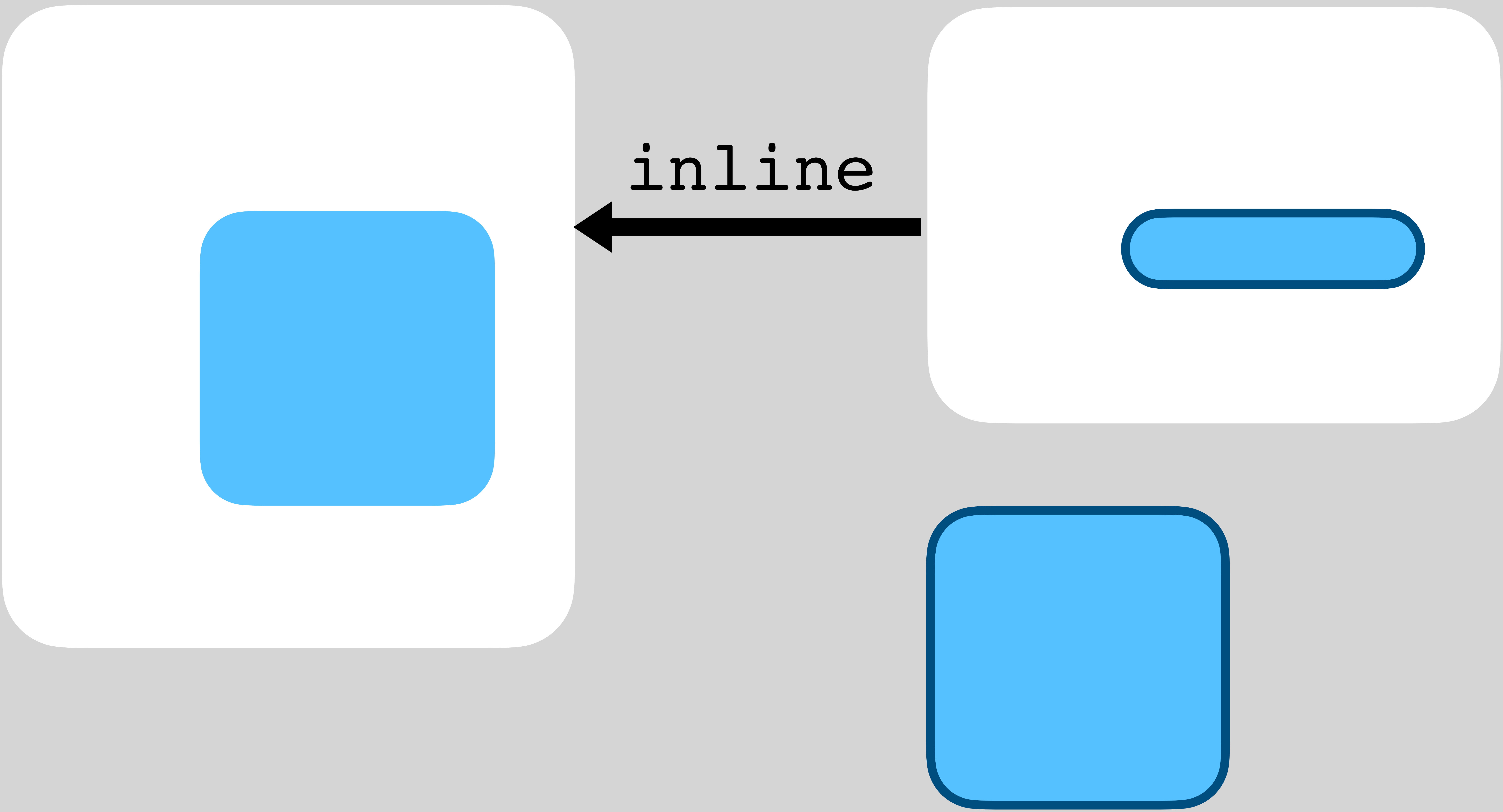
Scheduling as Program Rewriting



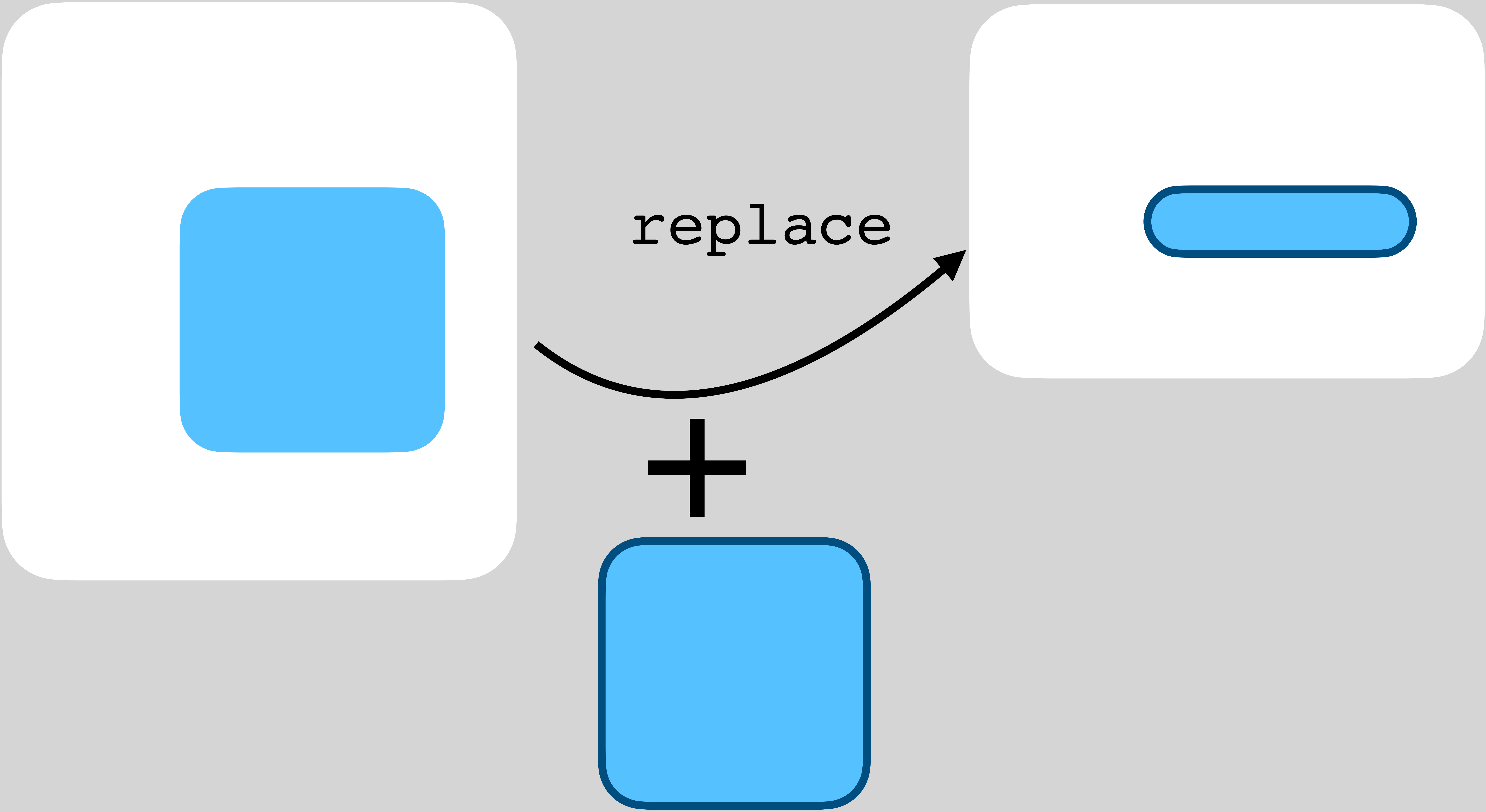
Scheduling Sub-procedures



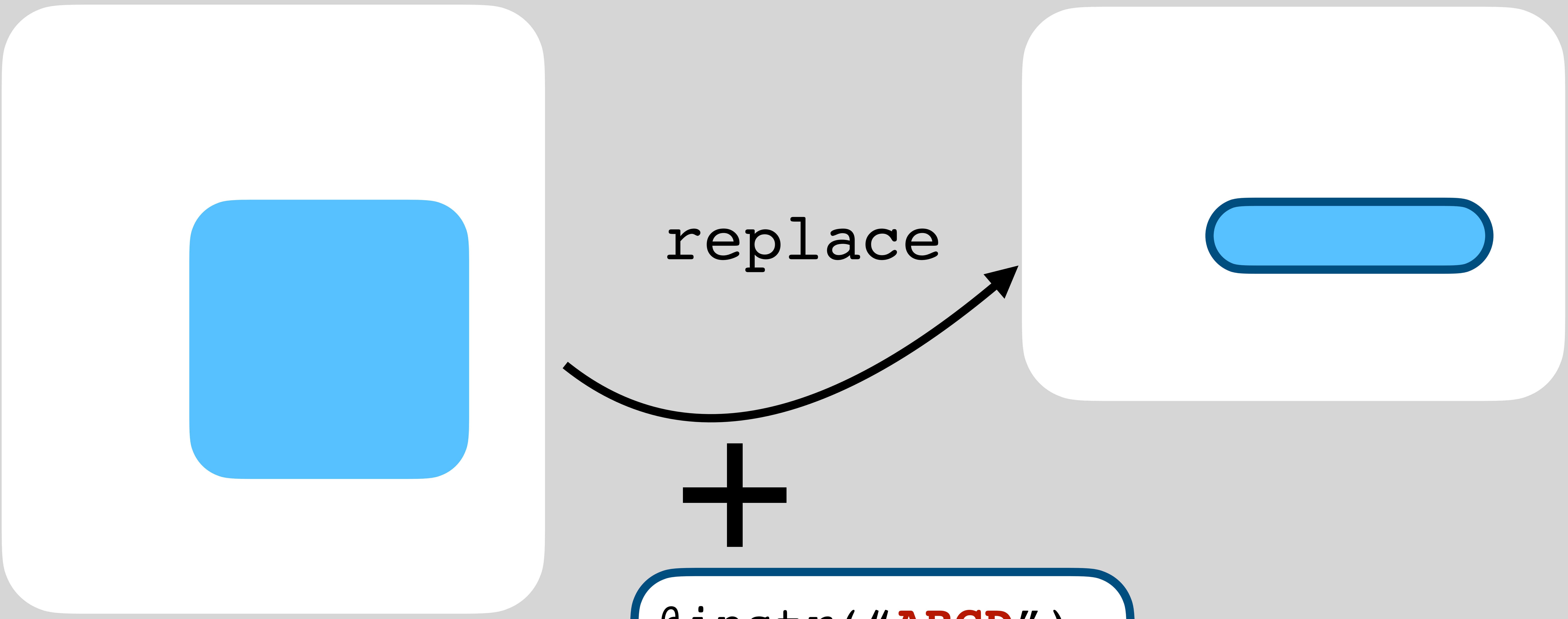
Scheduling Sub-procedures



Scheduling Sub-procedures



Scheduling Sub-procedures




```
@instr("ABCD")
def instr_A(…):
    …
```

Scheduling Operations in Exo

```
swap
inline
replace
```

```
reorder
split
unroll
set_memory
set_precision
bind_expr
stage_mem
bind_config
reorder_dim
expand_dim
add_guard
```

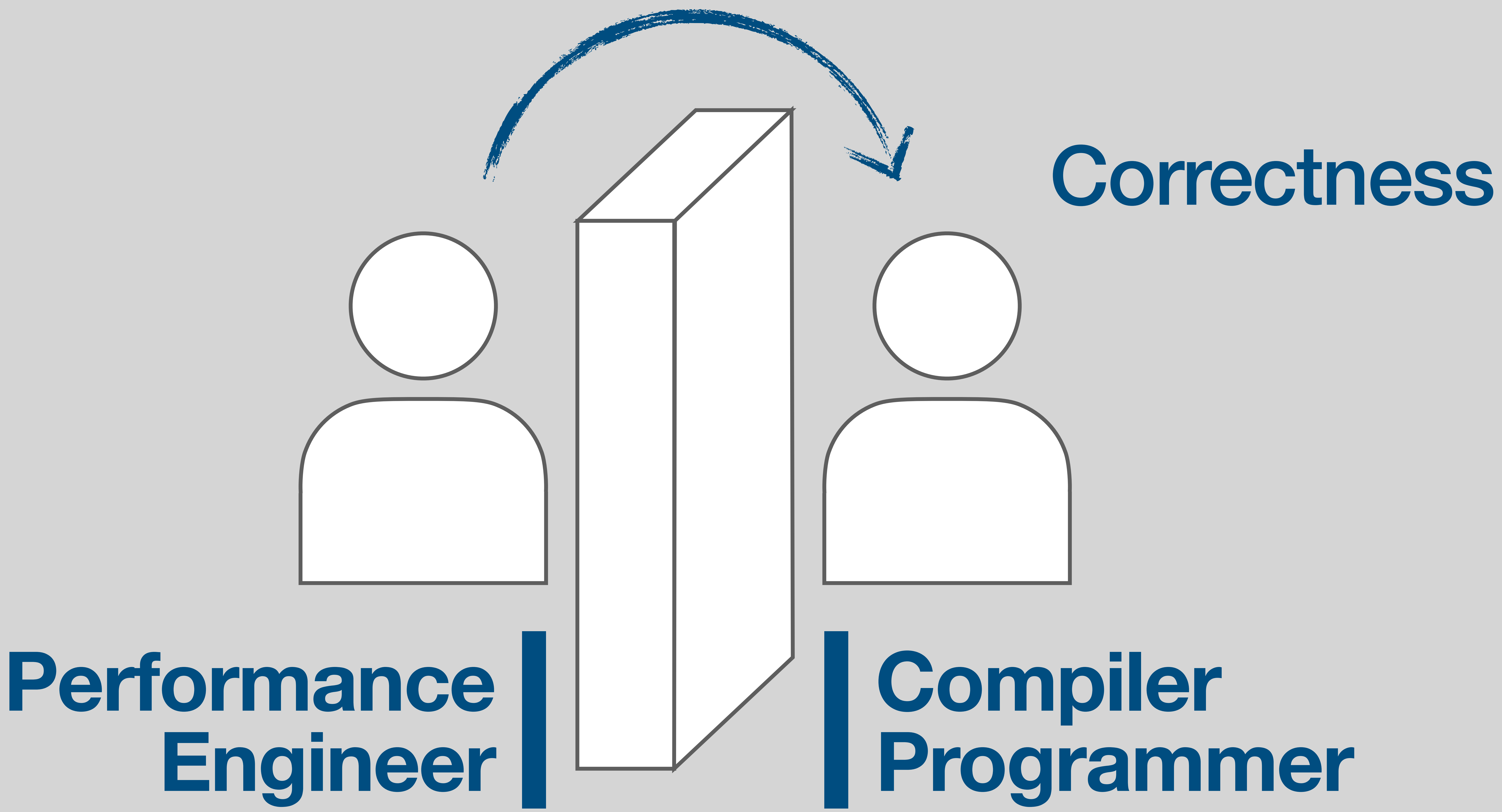
```
lift_alloc
fission_after
reorder_stmts
config_write
fuse_loop
lift_if
partition
remove_loop
...
```

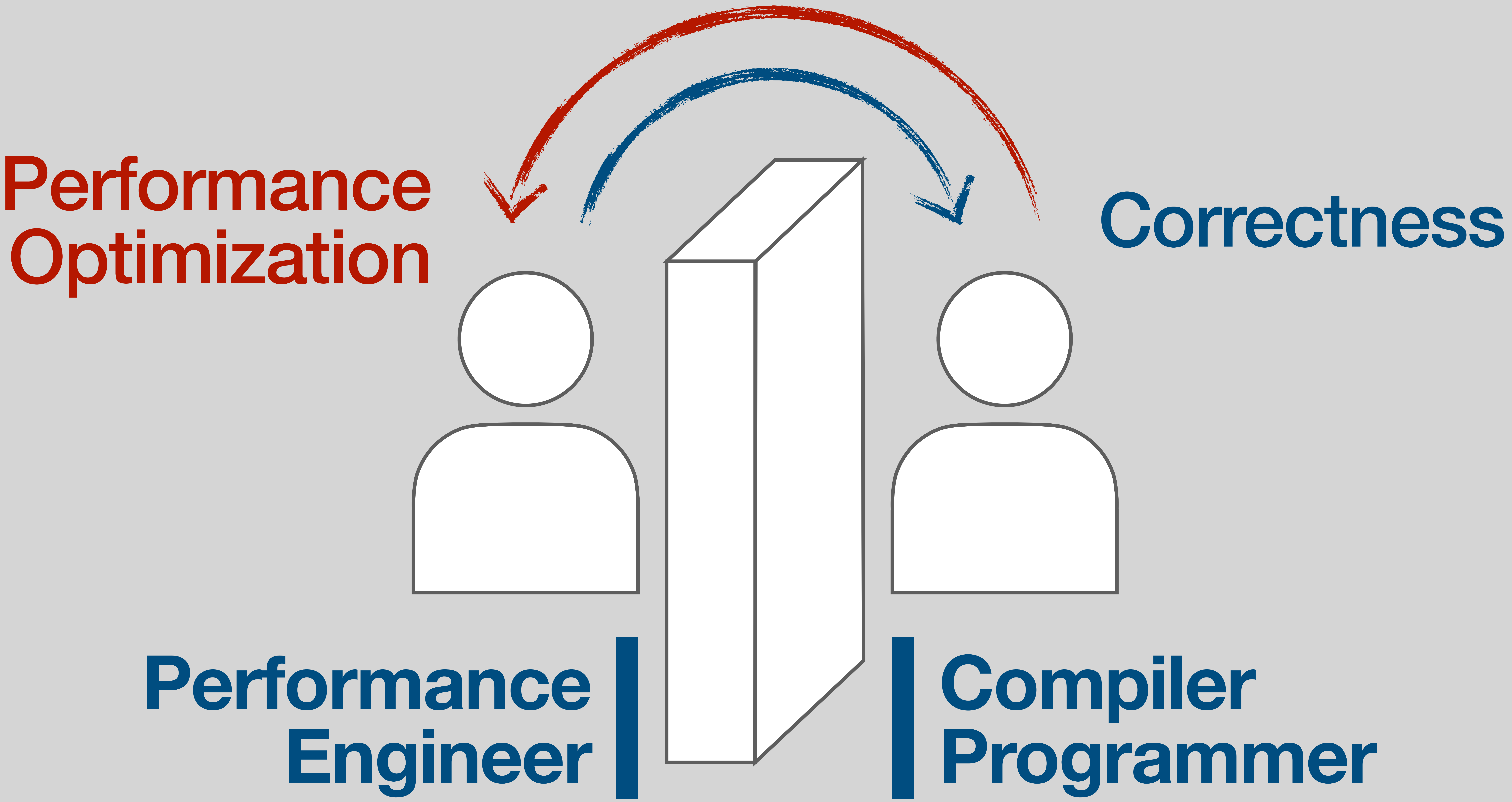
Scheduling Operations in Exo

```
swap          reorder        lift_alloc
```

Safety checked using SMT verification

```
              stage_mem      partition
              bind_config    remove_loop
              reorder_dim    ...
              expand_dim
              add_guard
```

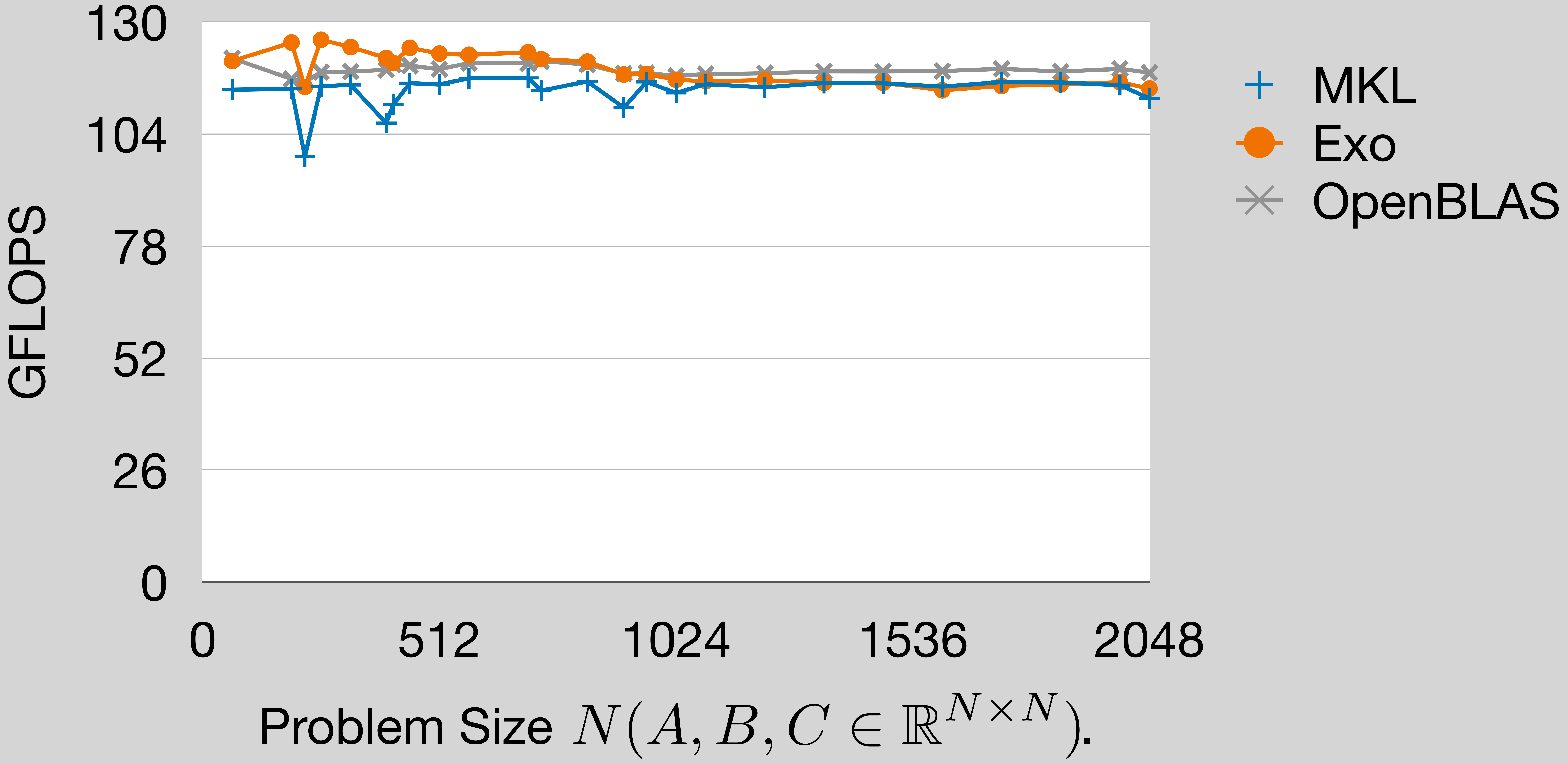

Correctness
Performance Engineer
Compiler Programmer


Performance
Optimization
Correctness
Performance
Engineer
Compiler
Programmer

Exo Performance Results

AVX-512 MatMul

Matches best possible performance

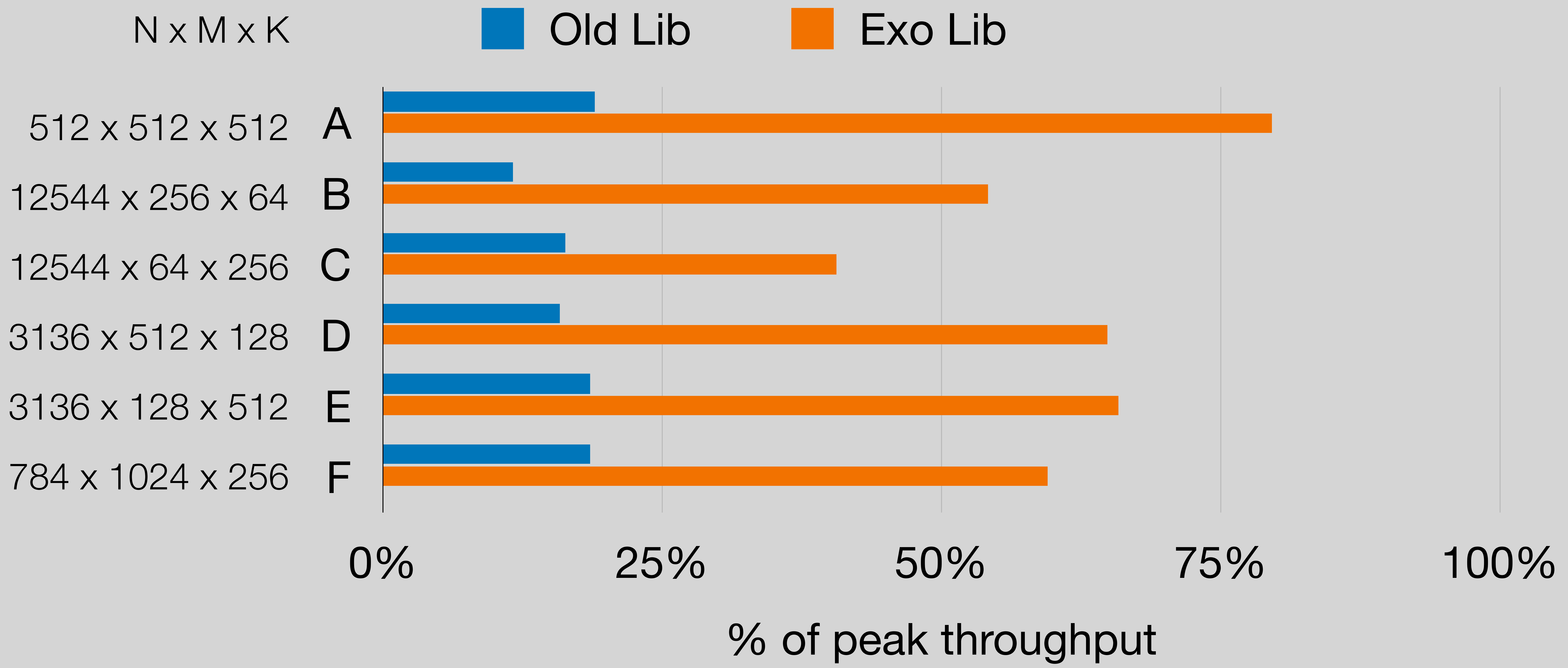


UC Berkeley

GEMMINI

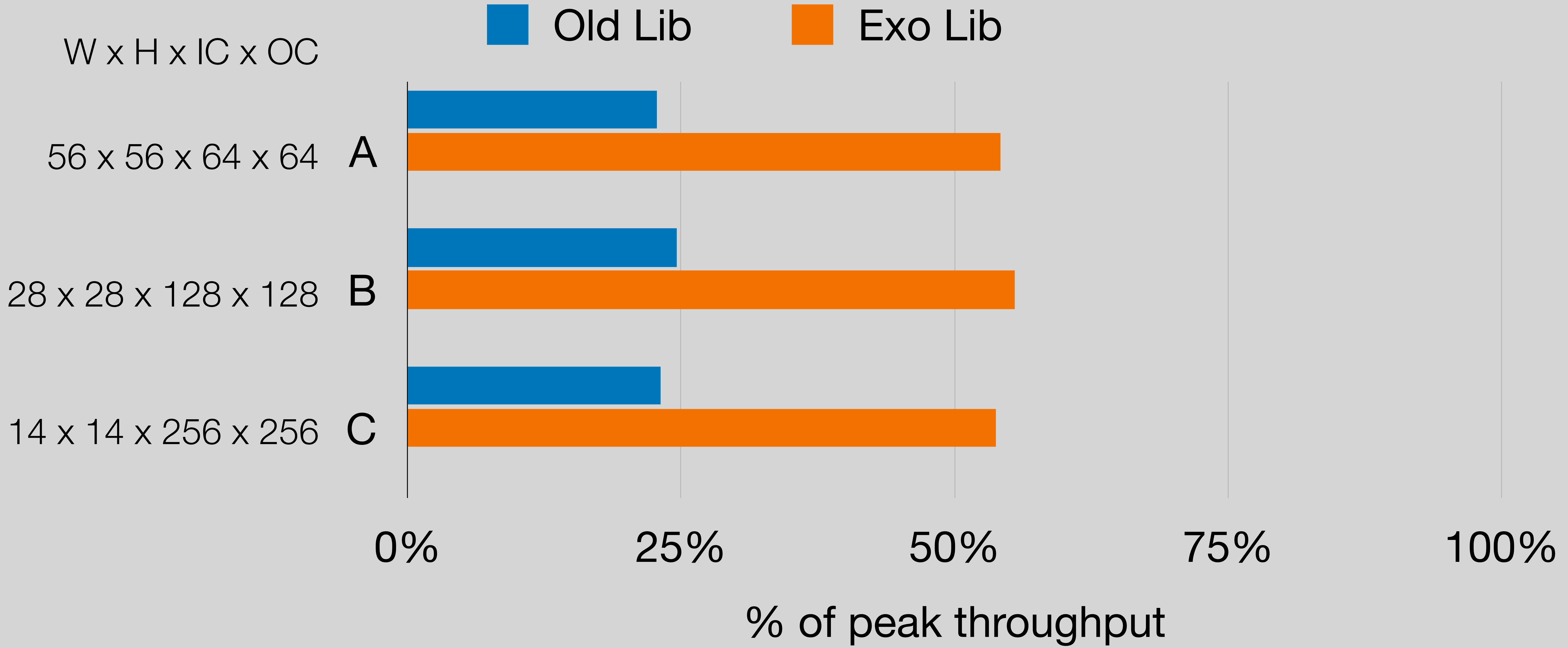
GEMMINI MatMul

2.3x - 3.7x faster than original HPC Library

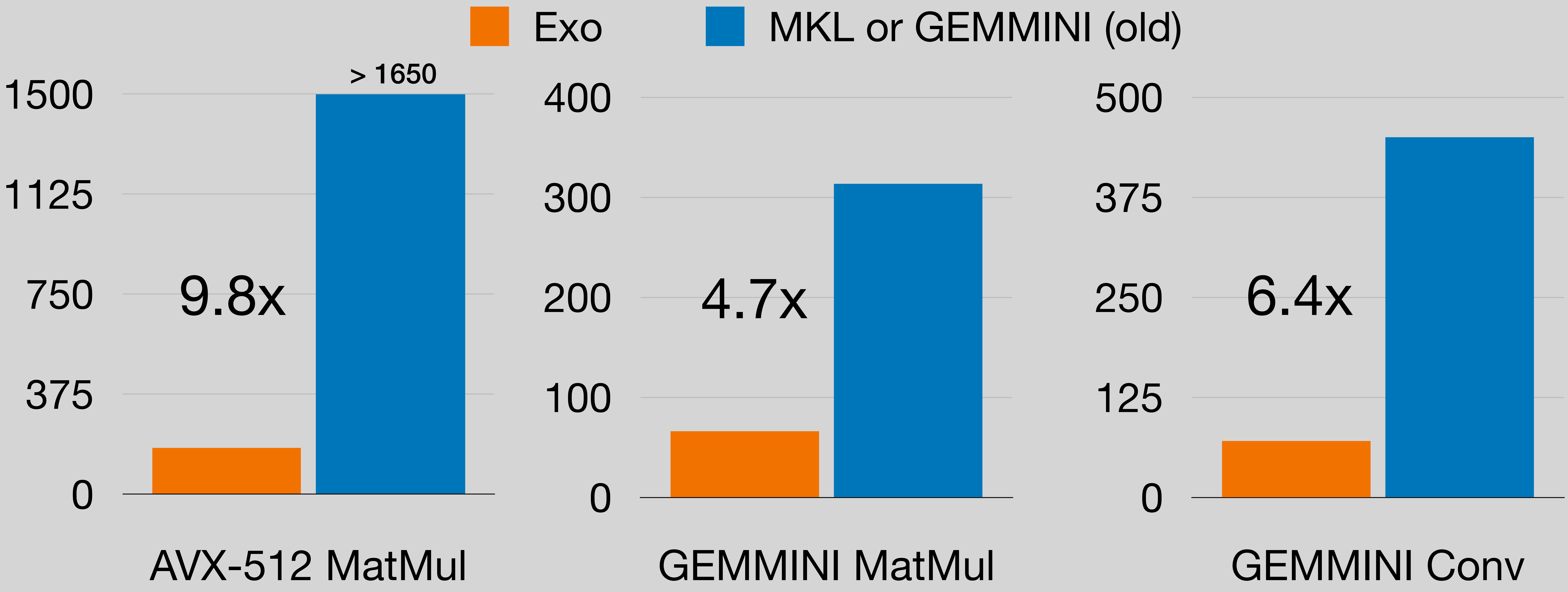


GEMMINI Conv

Similar Story for Convolution Layers



Exo vs. Existing - Lines of Code



HW / SW Co-Design

Benefits to Software Maintenance

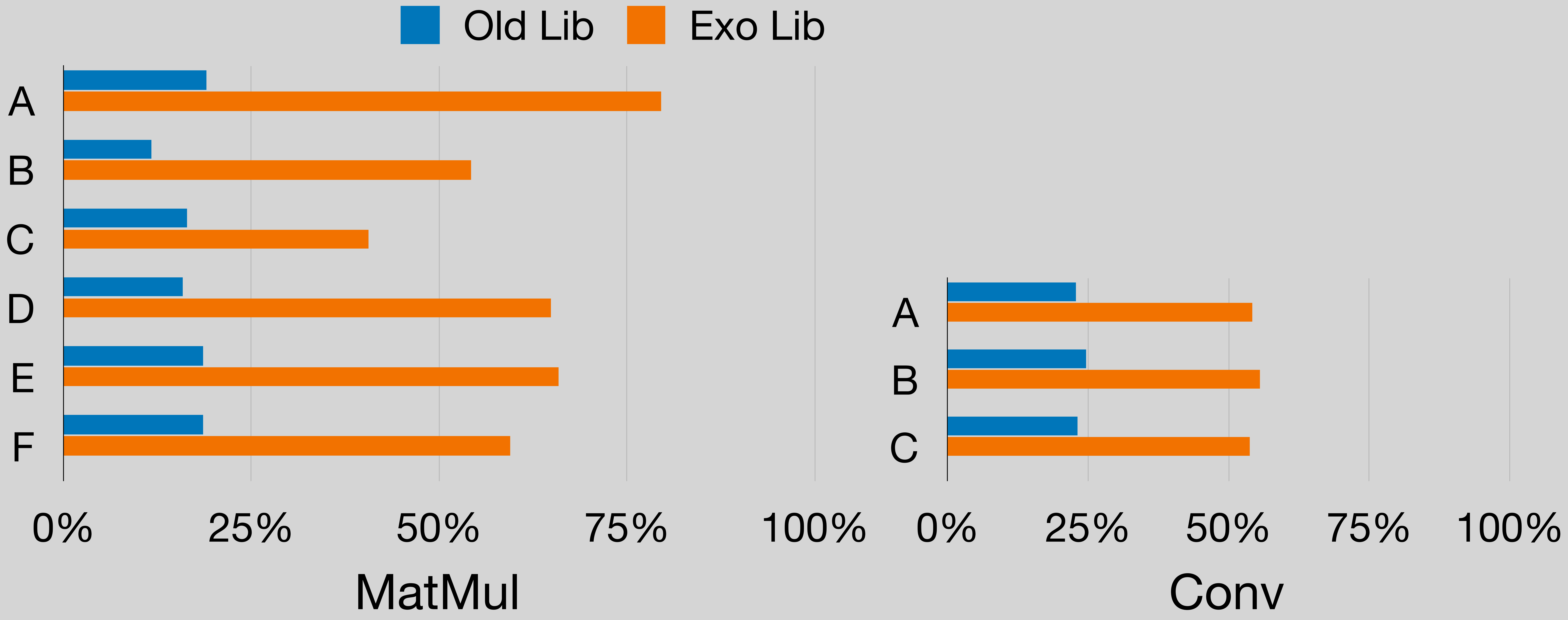
ISA Change

5 lines of Exo code changed

46 lines of old handwritten GEMMINI library changed

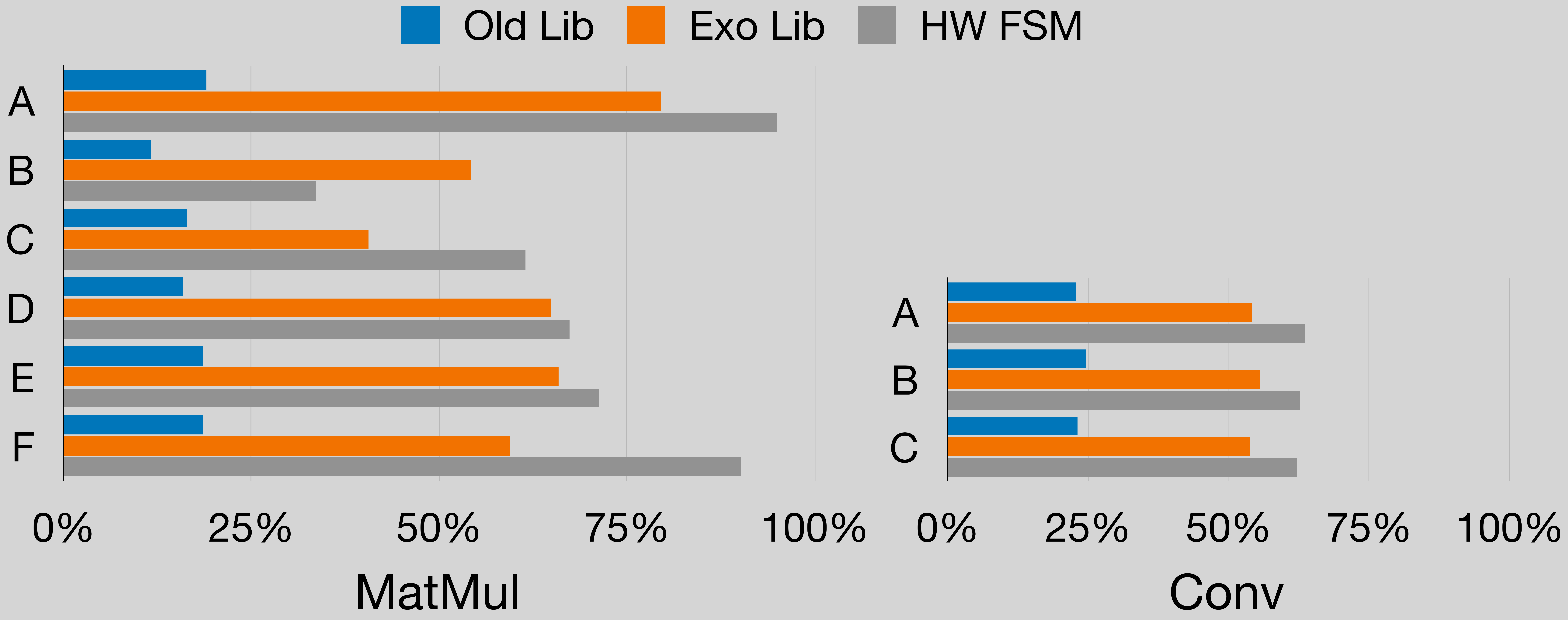
GEMMINI MatMul + Conv

HW / SW Co-Design



GEMMINI MatMul + Conv

HW / SW Co-Design



Exo Collaborations

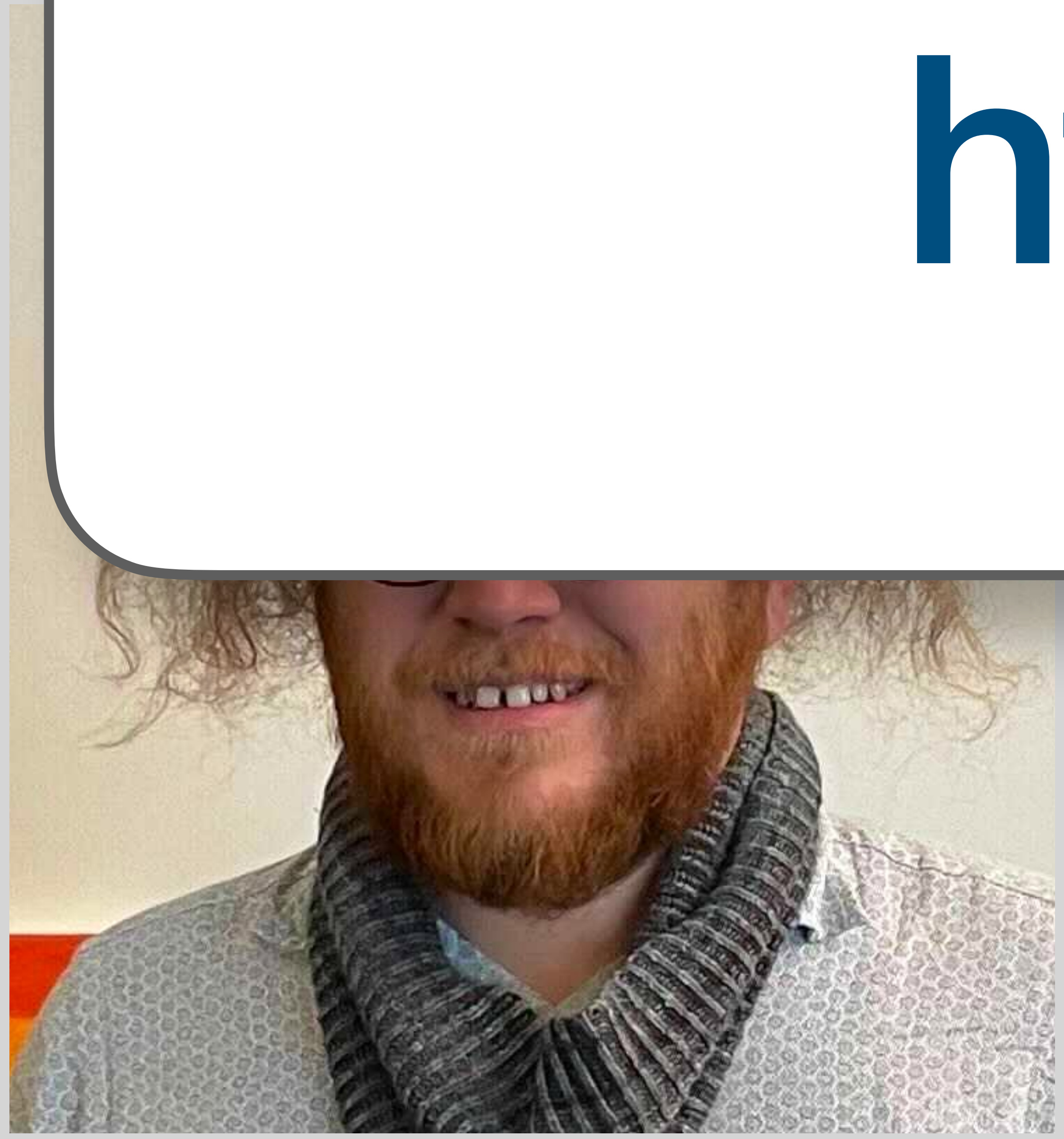
UC Berkeley - GEMMINI

Intel Labs - VTA

Apple - CoreML & Accelerate

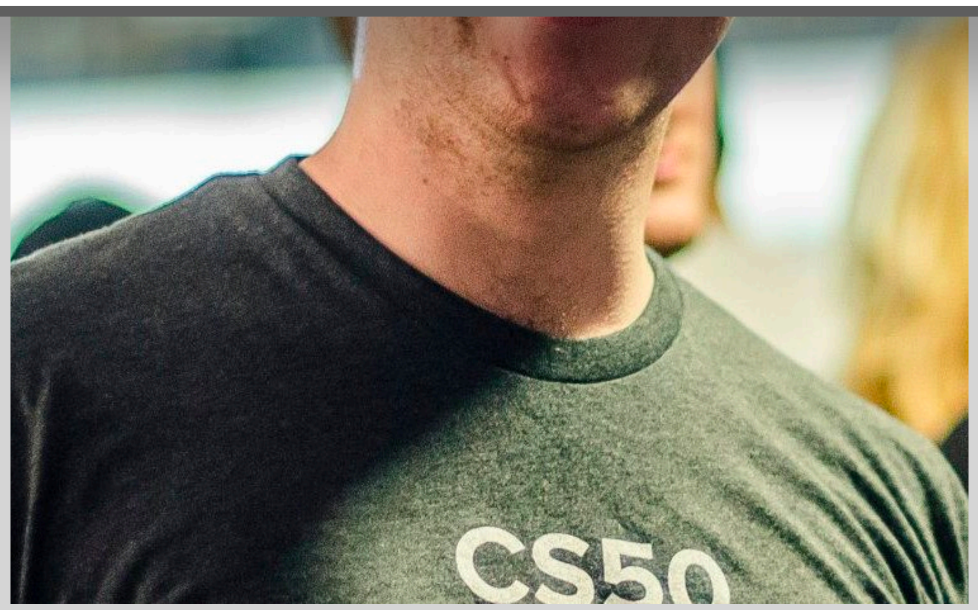
Exocompilation for Productive Programming of Hardware Accelerators

https://exo-lang.dev

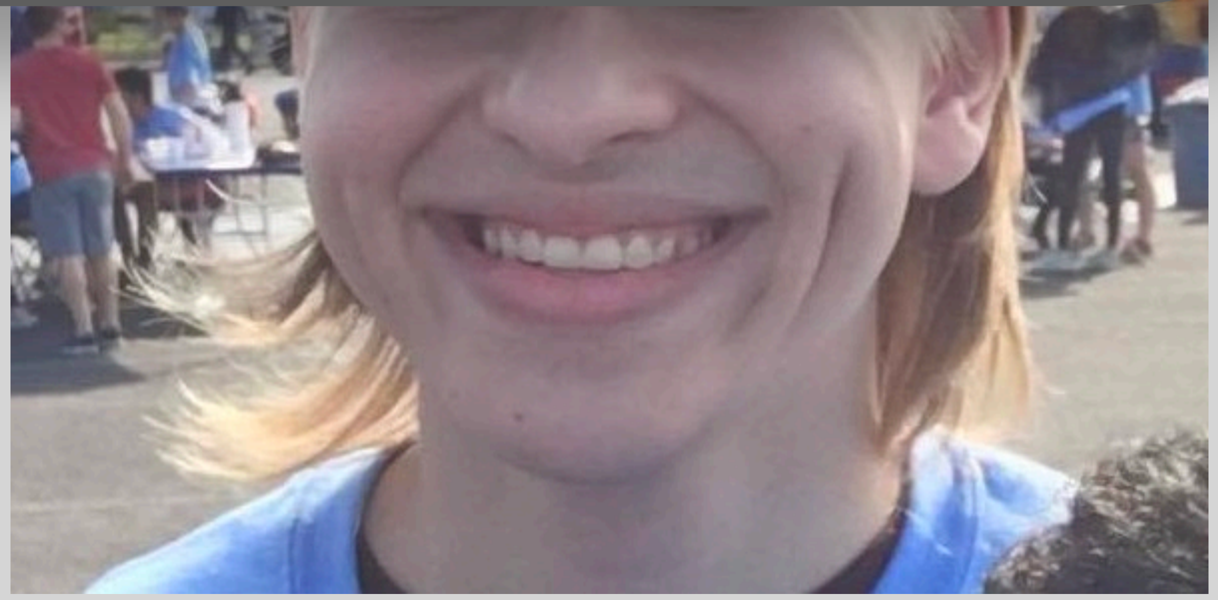


Gilbert Bernstein

Yuka Ikarashi



Alex Reinking



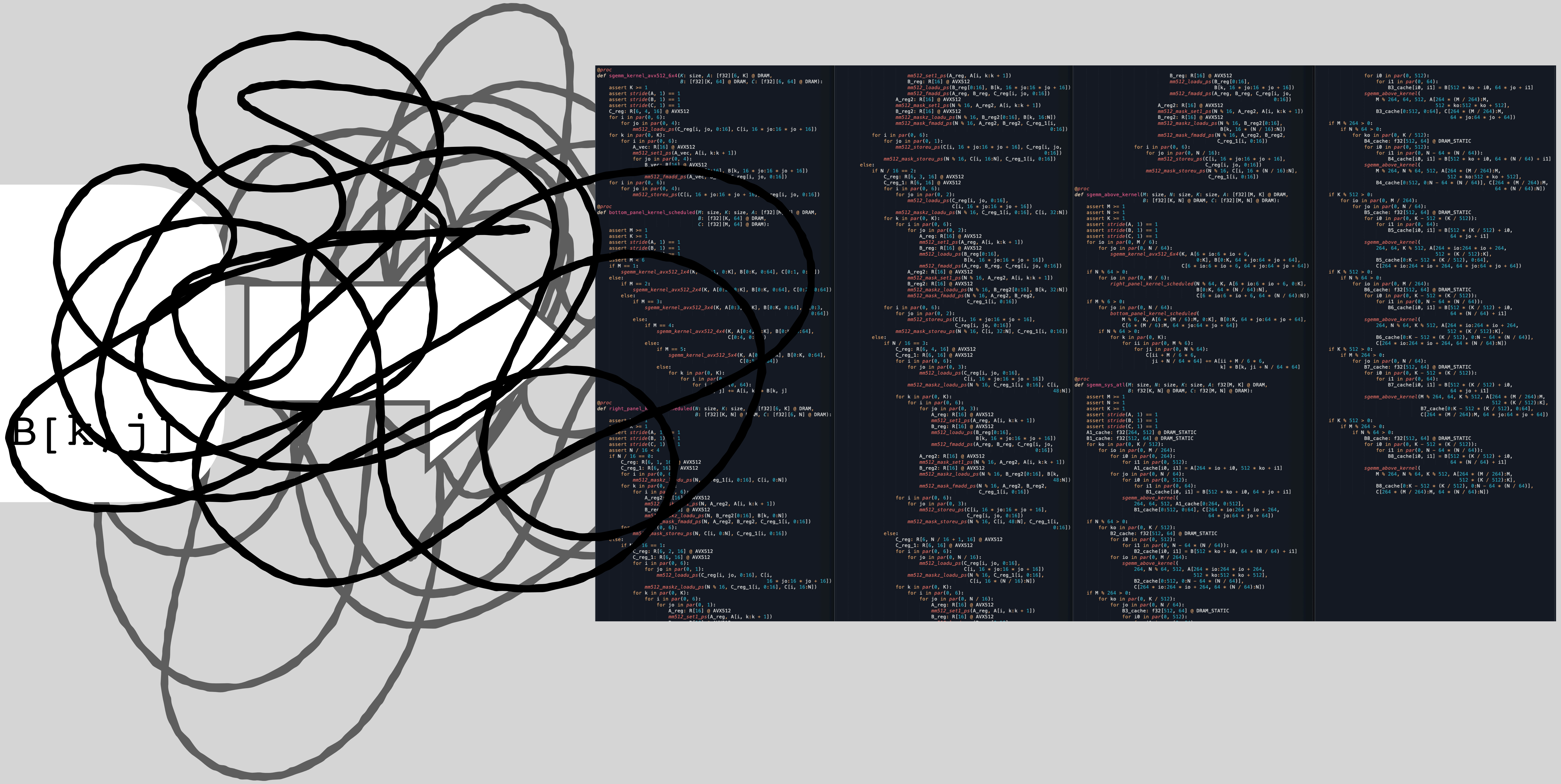
Hasan Genc

Observations From Exo

```
for i in seq(0,N):
  for j in seq(0,M):
    for k in seq(0,K):
      C[i,j] += A[i,k] * B[k,j]
```

Observations From Exo

```
for i in seq(0,N):
  for j in seq(0,M):
    for k in seq(0,K):
      C[i,j] += A[i,k] * B[k,j]
```



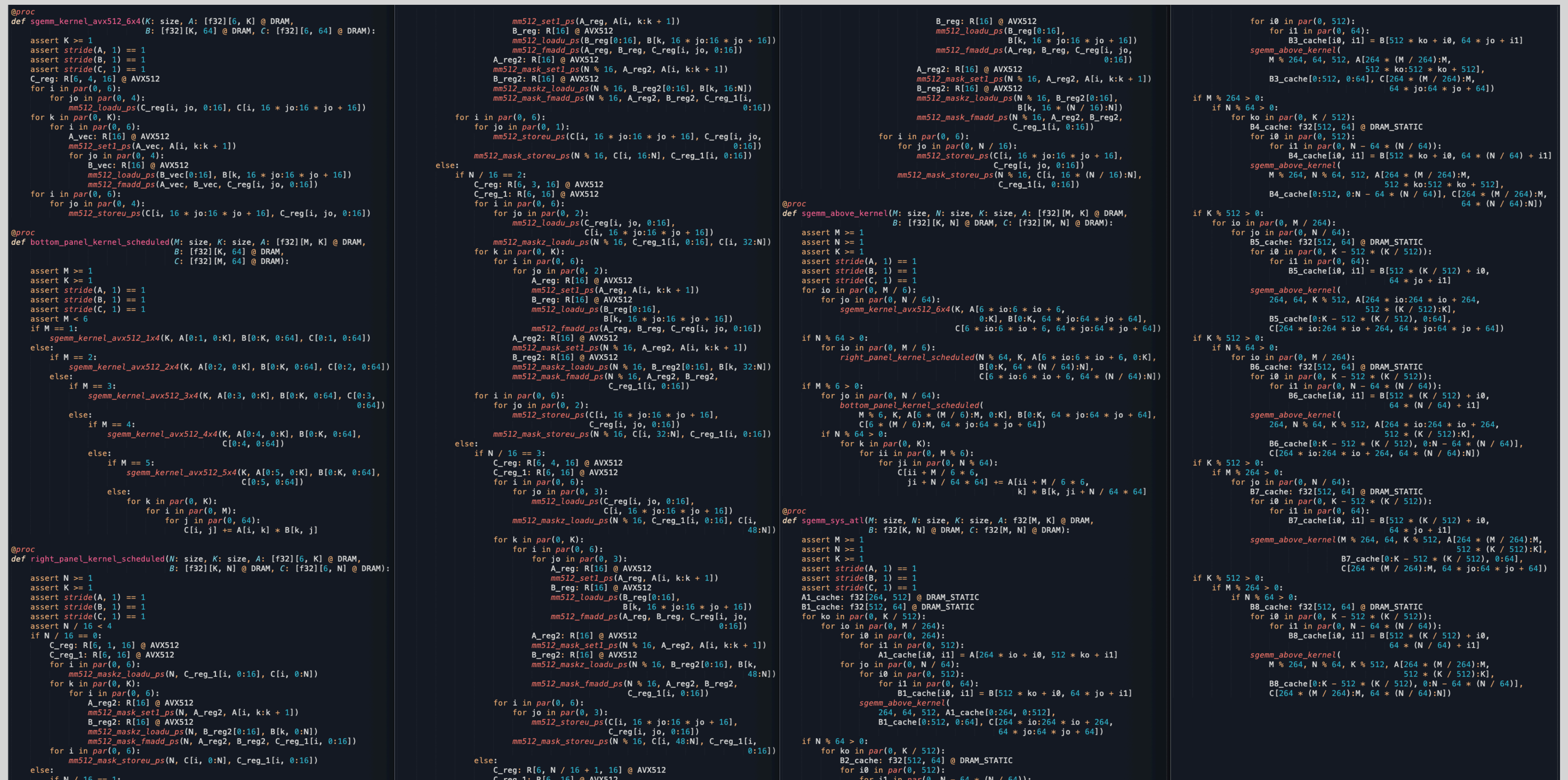
Functional Scheduling

Coq

$$\boxplus_{i=0}^{N} \boxplus_{j=0}^{M} \sum_{k=0}^{K} A[i,k] \cdot B[k,j]$$

$$\boxplus_{i=0}^{N/16} \boxplus_{j=0}^{M/16} \sum_{k=0}^{K/16} \left(\begin{array}{l} \text{let } A' := \boxplus_{ii=0}^{16} \ldots \\ \text{let } B' := \boxplus_{kk=0}^{16} \ldots \\ \text{let } C' := \boxplus_{ii=0}^{16} \ldots \\ \quad \vdots \end{array} \right.$$

```
for i in seq(0,N):
  for j in seq(0,M):
    for k in seq(0,K):
      C[i,j] += A[i,k] * B[k,j]
```



Functional Scheduling



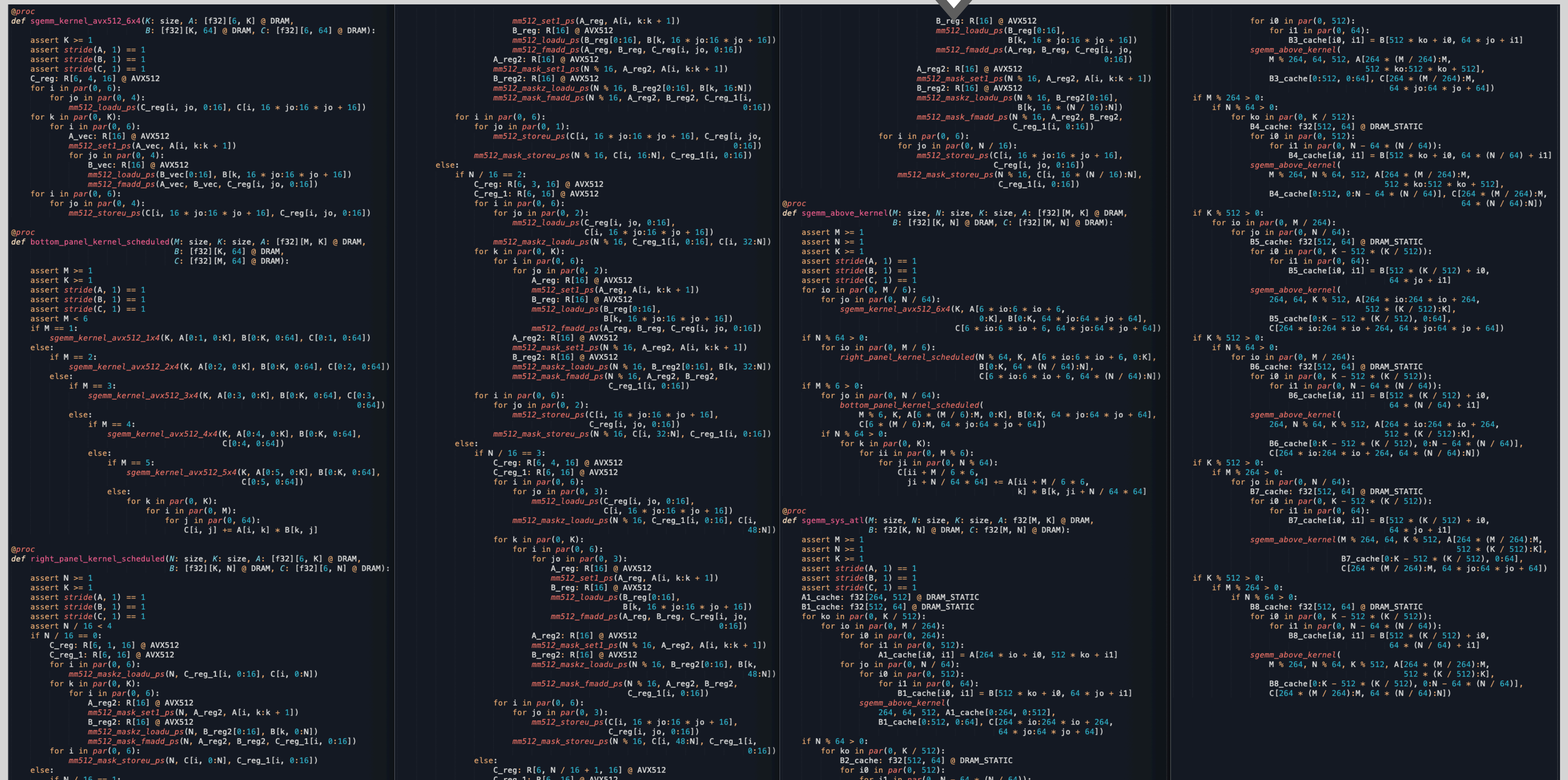
Coq

$$\boxplus_{i=0}^{N} \boxplus_{j=0}^{M} \sum_{k=0}^{K} A[i,k] \cdot B[k,j]$$

$$\boxplus_{i=0}^{N/16} \boxplus_{j=0}^{M/16} \sum_{k=0}^{K/16}$$

$$\left(\begin{array}{l} \text{let } A' := \boxplus_{ii=0}^{16} \boxplus_{kk} \\ \text{let } B' := \boxplus_{kk=0}^{16} \boxplus_{j} \\ \text{let } C' := \boxplus_{ii=0}^{16} \boxplus_{jj} \\ \vdots \end{array} \right.$$

```
for i in seq(0,N):
  for j in seq(0,M):
    for k in seq(0,K):
      C[i,j] += A[i,k] * B[k,j]
```



First Formally Verified User-Schedulable Language

[POPL 2022]

Coq

$$\underset{i=0}{\overset{N}{\boxplus}}\ \underset{j=0}{\overset{M}{\boxplus}}\ \sum_{k=0}^{K} A[i,k] \cdot B[k,j]$$

Lemmas ⟺ Scheduling Rewrites

Interactive Proving ⟺ Interactive Scheduling

User-Schedulable Languages 2.0

Programming New HW & Verifying Optimizations

[PLDI 2022 + POPL 2022]

Acknowledgements

Andrew Adams
Luke Anderson
Saman Amarasinghe
Connelly Barnes
Gilbert Bernstein
Adam Chlipala
Frédo Durand
Kayvon Fatahalian
Hasan Genc
Michael Gharbi
Yuka Ikarashi
Steven Johnson
Shoaib Kamil
Marc Levoy
Tzu-Mao Li
Amanda Liu
Karima Ma
Ravi Mullapudi
Sylvain Paris
Alex Reinking
Dillon Sharlet
Zalman Stern

Thank you!

Acknowledgements

Andrew Adams
Luke Anderson
Saman Amarasinghe
Connelly Barnes
Gilbert Bernstein
Adam Chlipala
Frédo Durand
Kayvon Fatahalian
Hasan Genc
Michael Gharbi
Yuka Ikarashi
Steven Johnson
Shoaib Kamil
Marc Levoy
Tzu-Mao Li
Amanda Liu
Karima Ma
Ravi Mullapudi
Sylvain Paris
Alex Reinking
Dillon Sharlet
Zalman Stern

My group's research:

Capture & control **dependencies**

Reorganize computations & data

Define & exploit **structure**

Image Processing
3D Graphics
Machine Learning

- applications
- algorithms
- data structures
- languages
- compilers
- hardware