

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

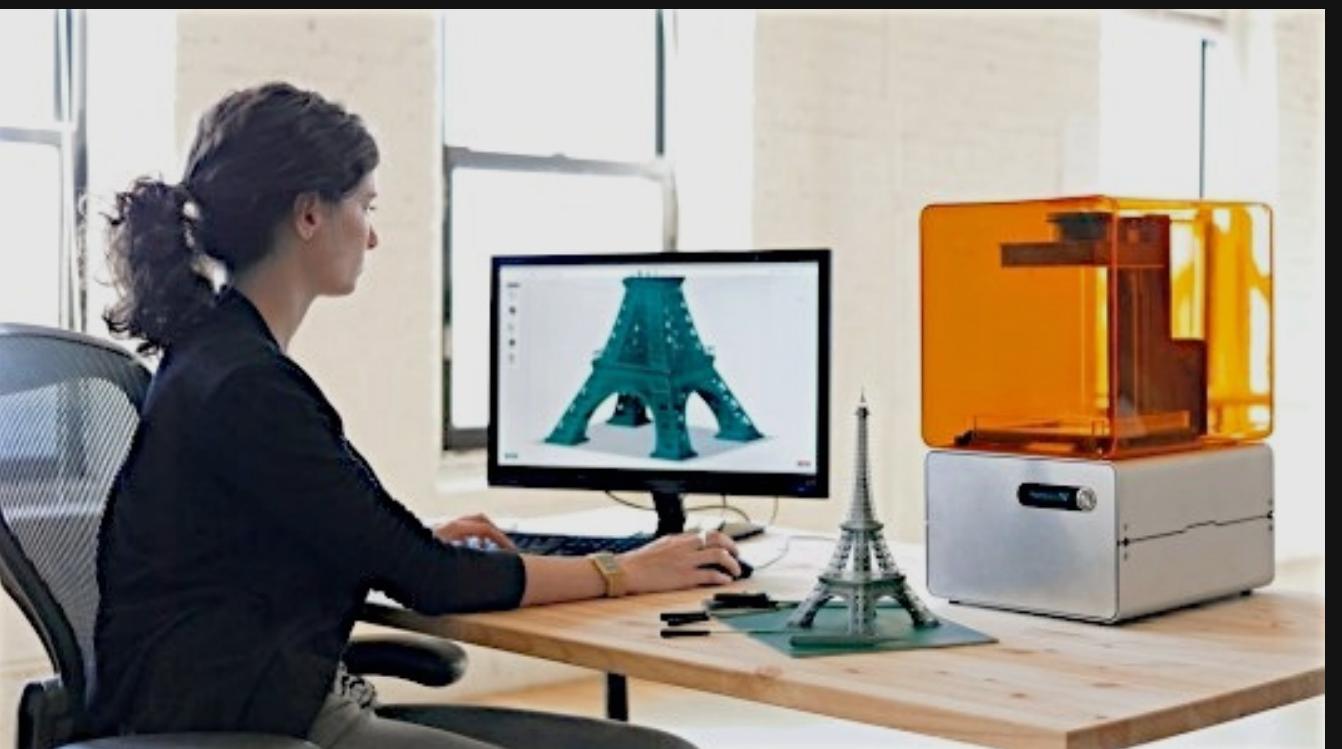
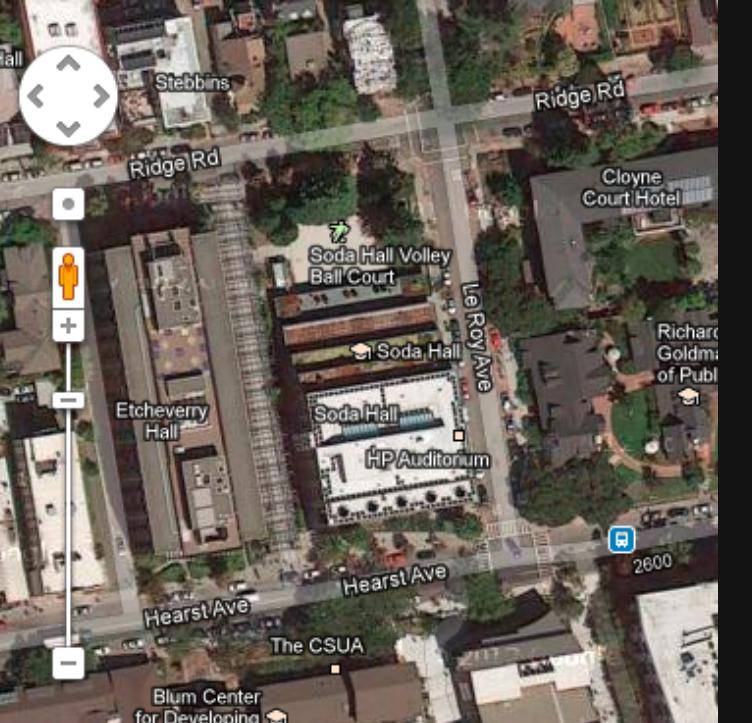
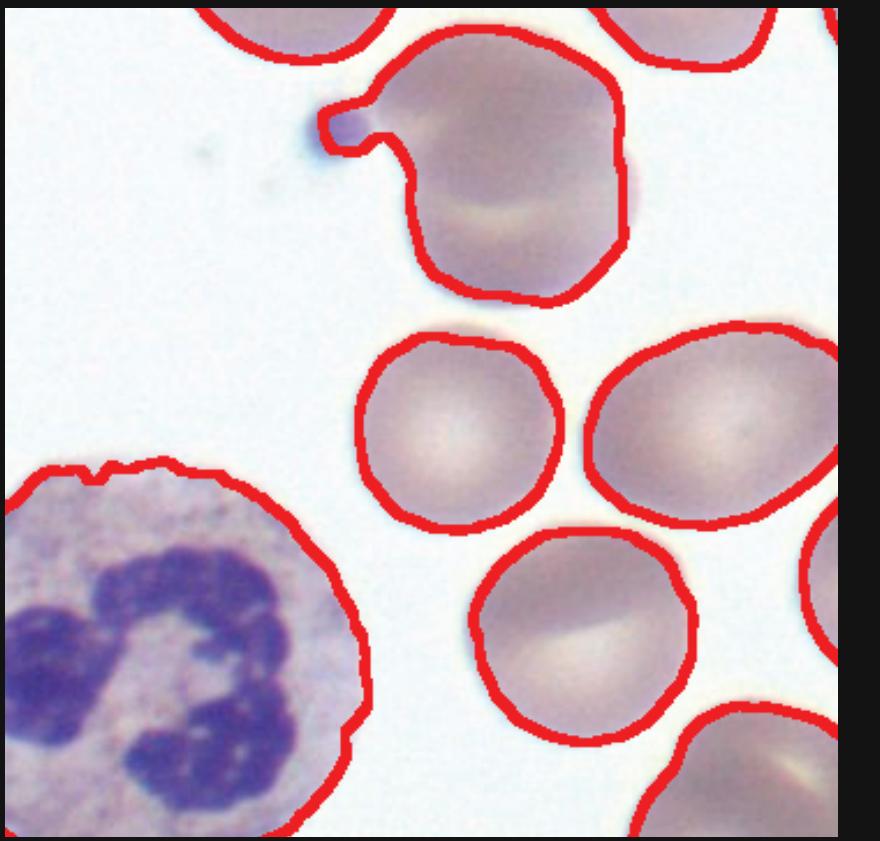
User-schedulable languages 2.0

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



6 orders of magnitude
more computation

images by Valve, Weta

The biggest data is visual

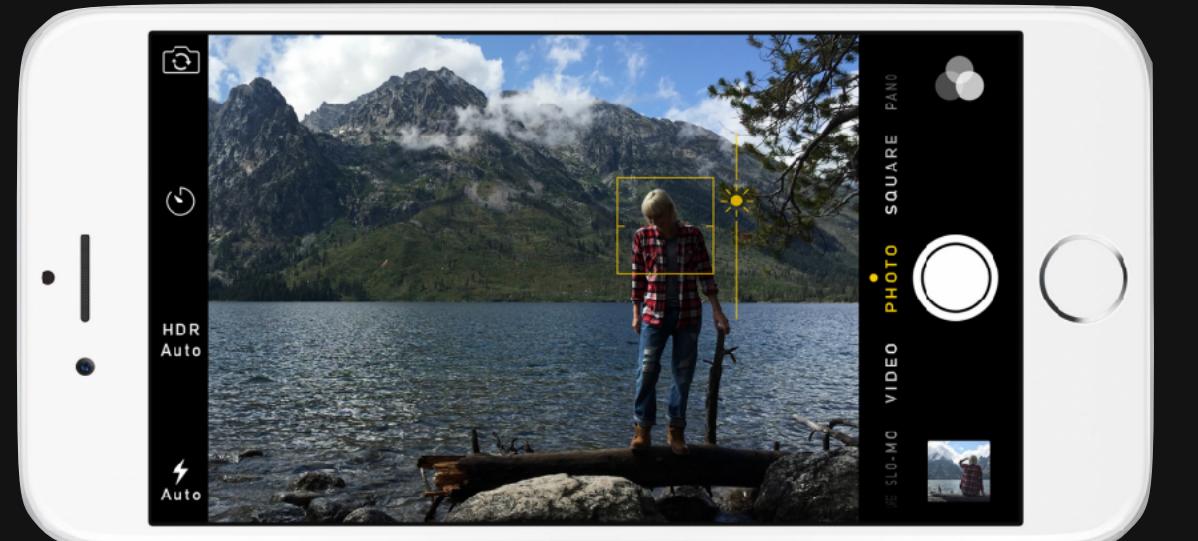
YouTube: 400 hrs uploaded / min

[Brewer 2016]

1.5 Terapixels/sec

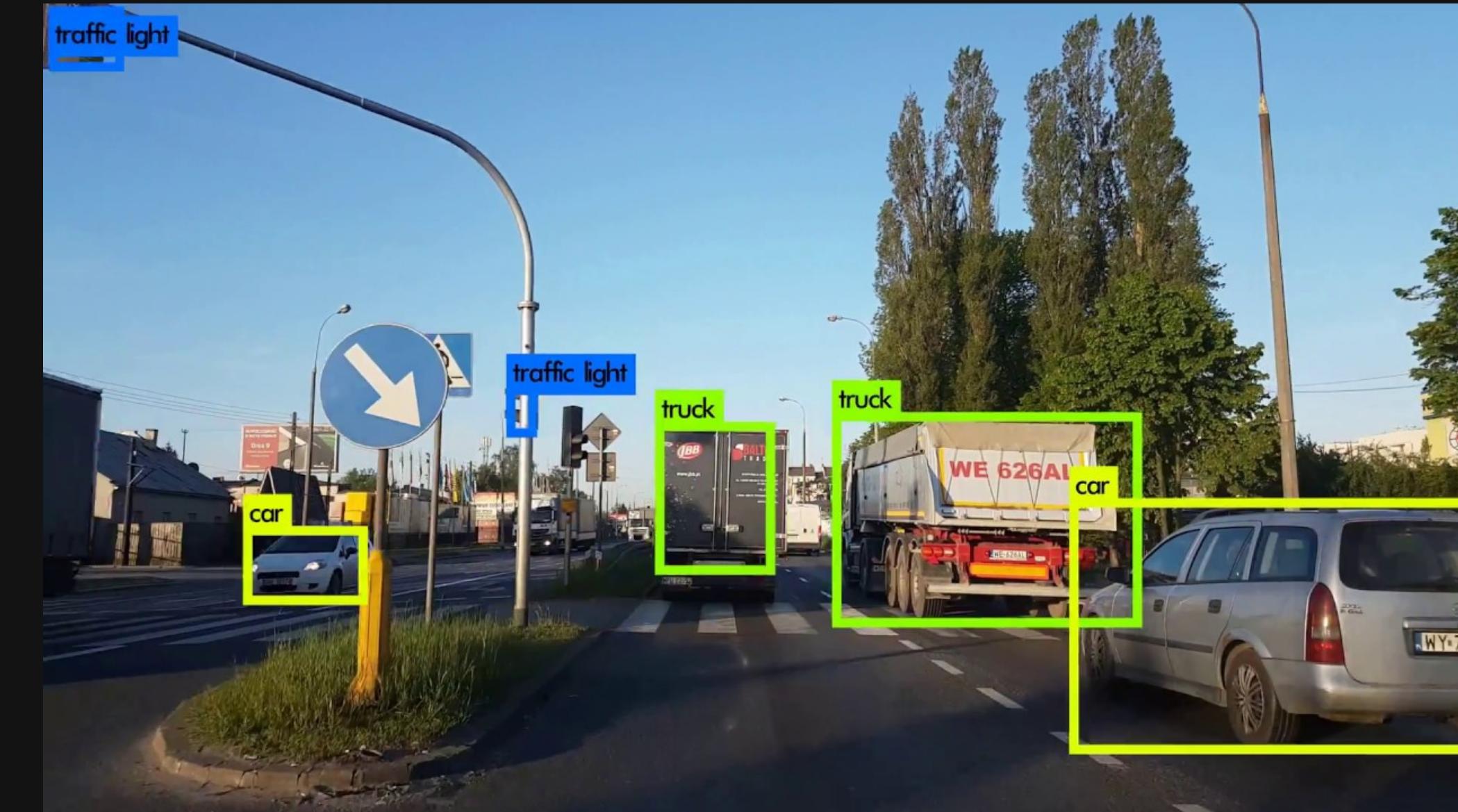


250 M surveillance cameras,
2.5 B cell phone cameras, ...



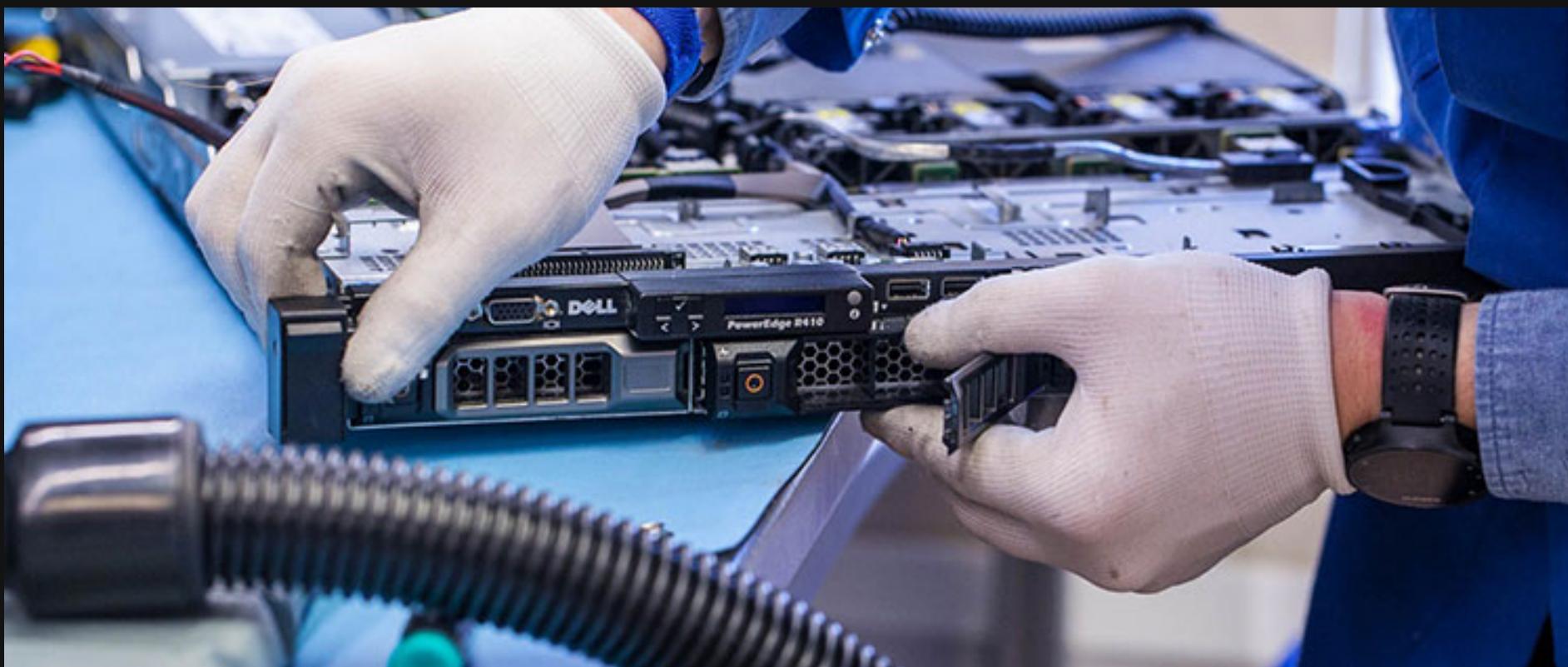
Visual data analysis is expensive

One object detection neural net:
250 Watt GPU → 0.25 megapixels at video rate



(yolo-v3 on Tesla V100)

Programmer productivity has exploded



1990s



ORACLE®



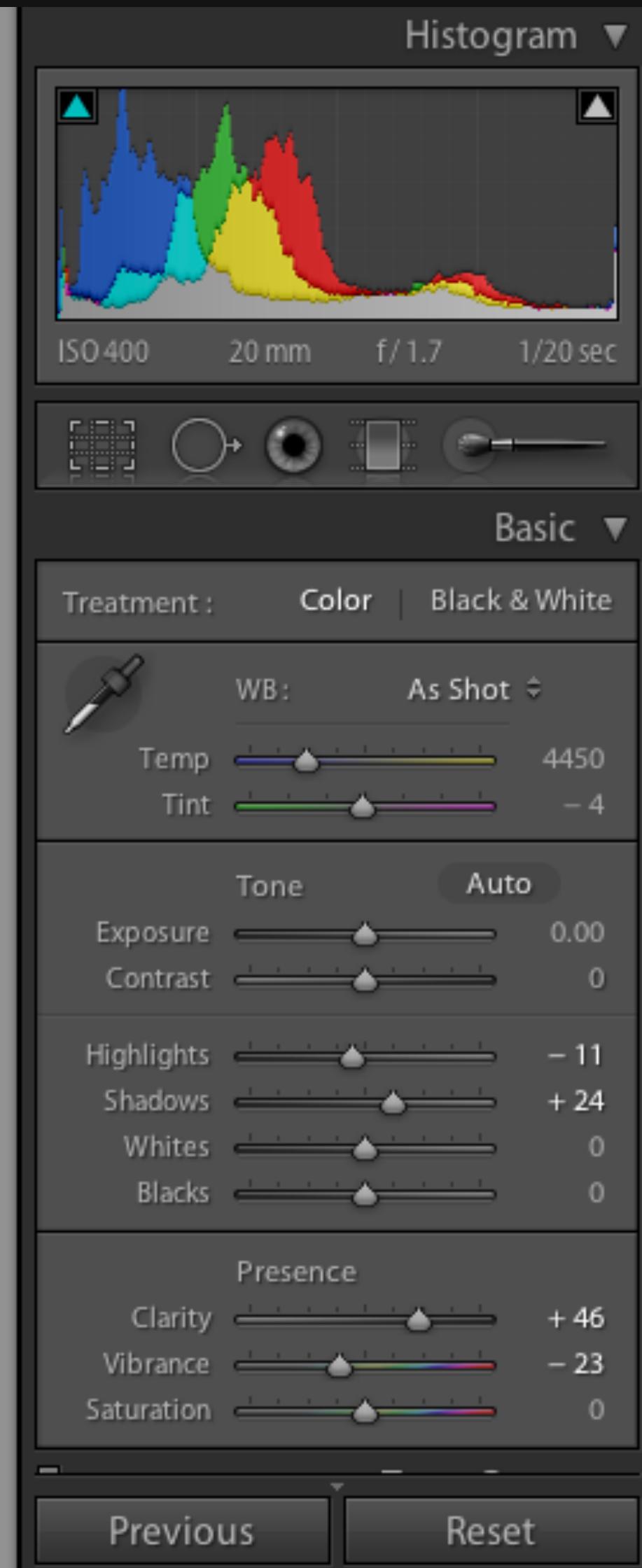
2010s

Building high-performance systems is harder than ever



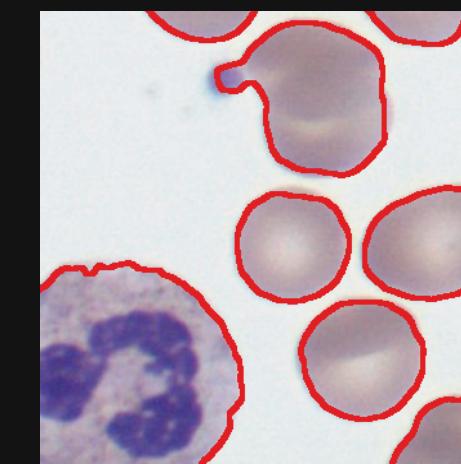
Reference:
200 lines C++

Adobe: 1500 lines
3 months of work
10x faster



My group's research:

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



applications

algorithms

data structures

languages

compilers

hardware

- { Reorganize computations & data
- Capture & control dependencies
- Define & exploit domain-specific structure

How can we increase performance and efficiency?

Parallelism

“Moore’s law” scaling requires exponentially more parallelism.

Locality

Data should move as little as possible.

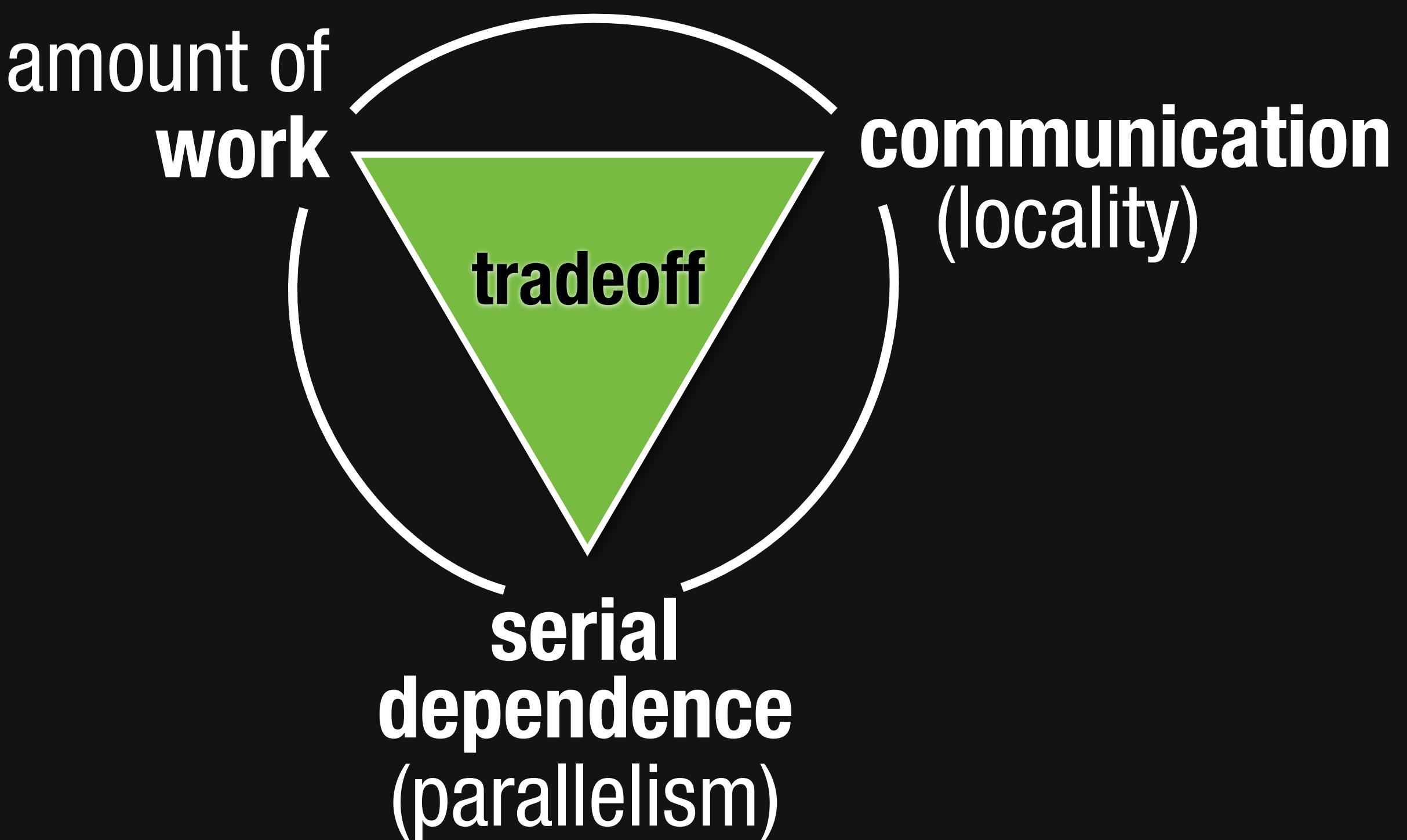
Communication dominates computation in both energy and time

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

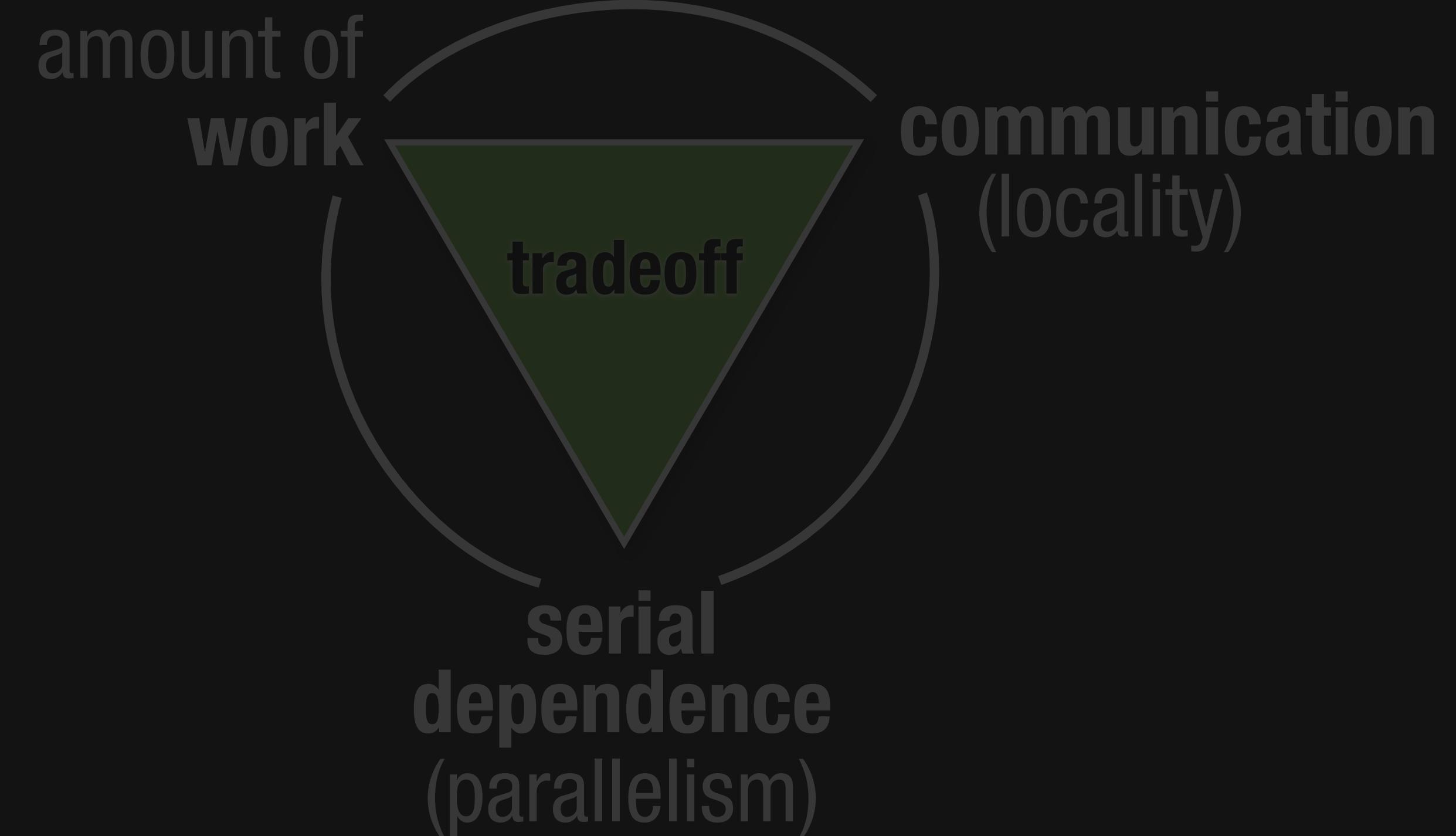
Message #1: Performance requires complex tradeoffs



Where does performance come from?

Program

Hardware

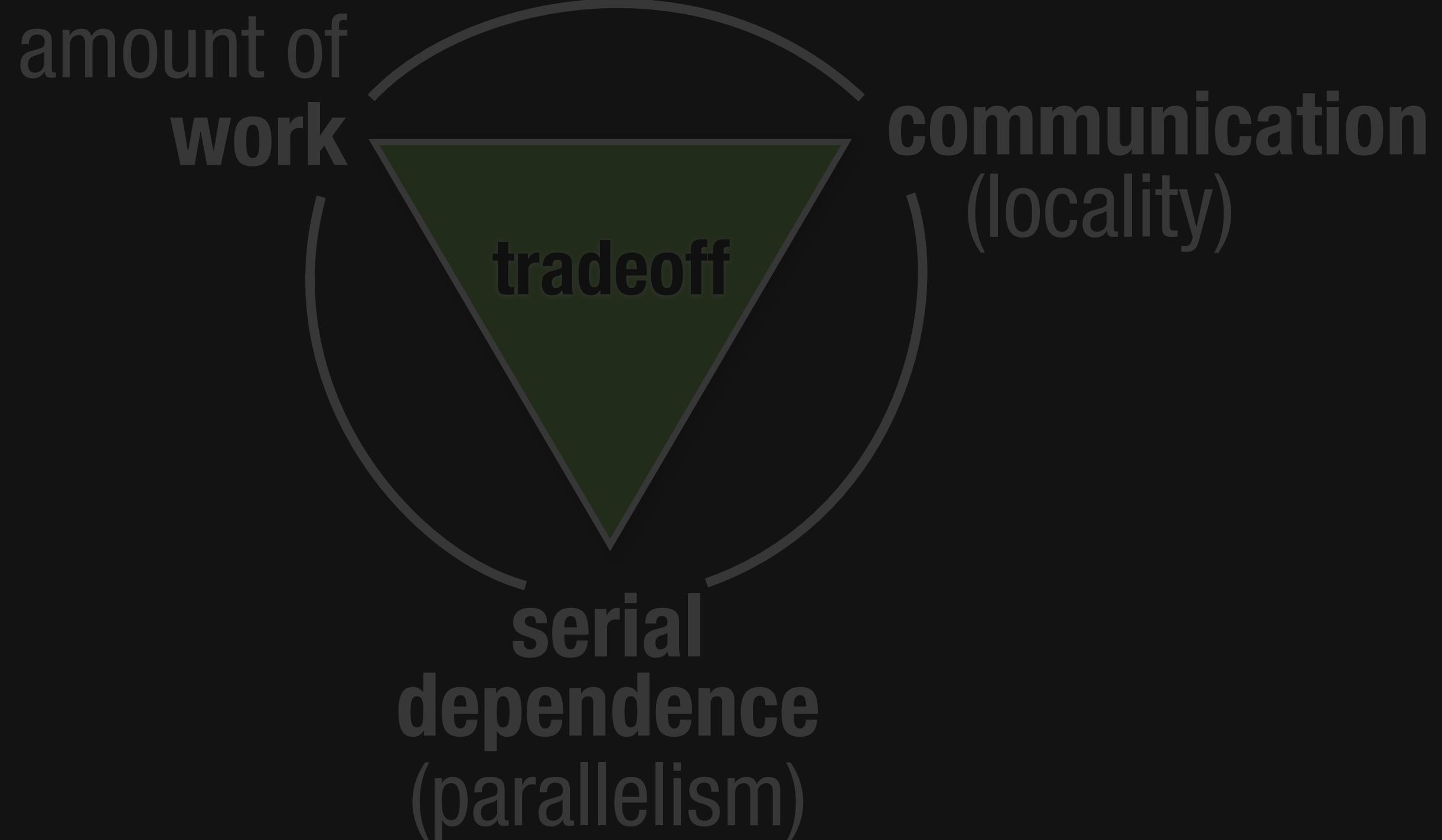


Message #2: organization of computation is a first-class issue

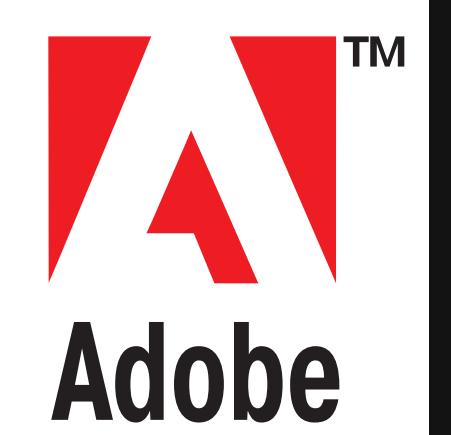
Program:



Hardware



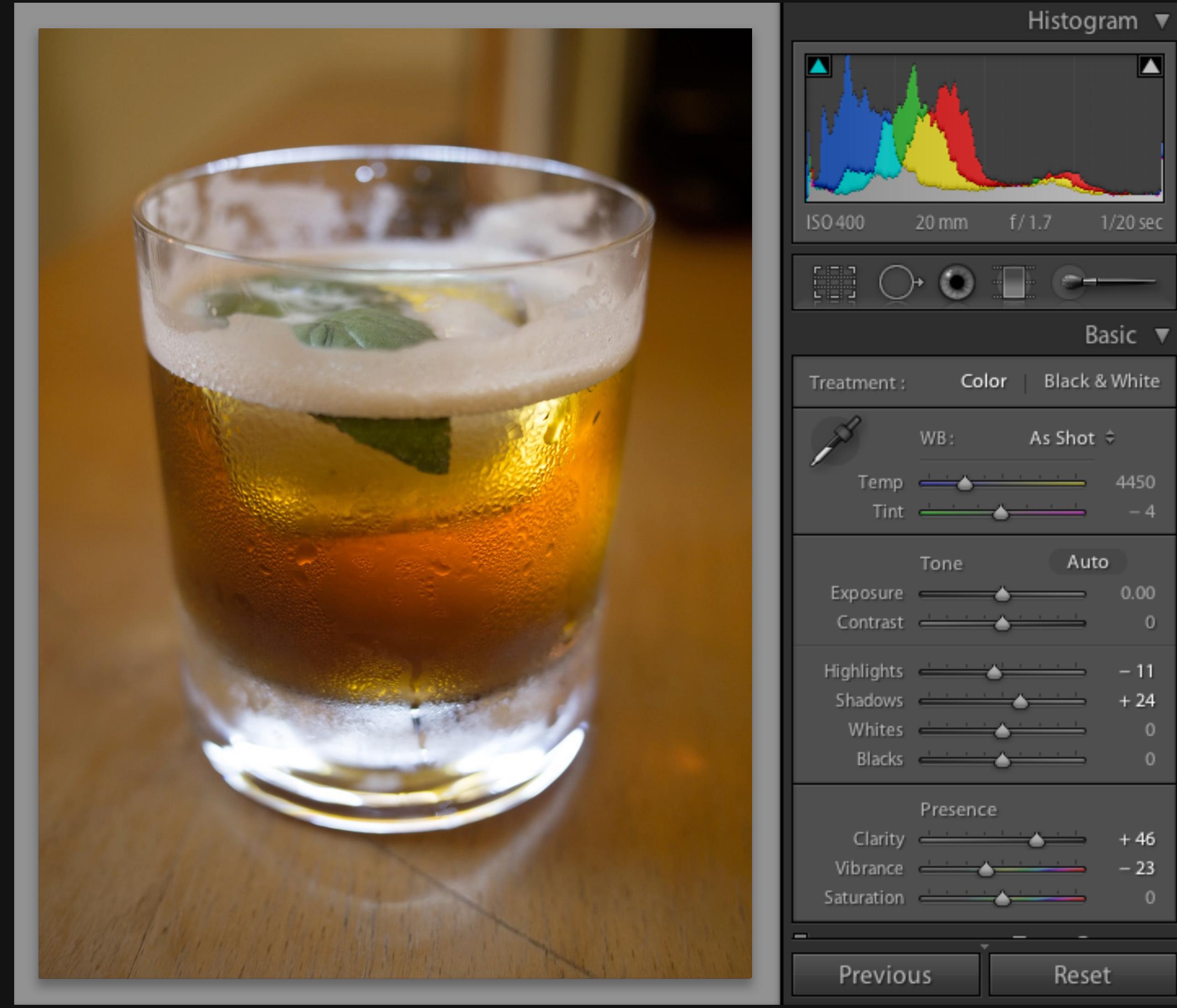
Reorganizing computation is painful



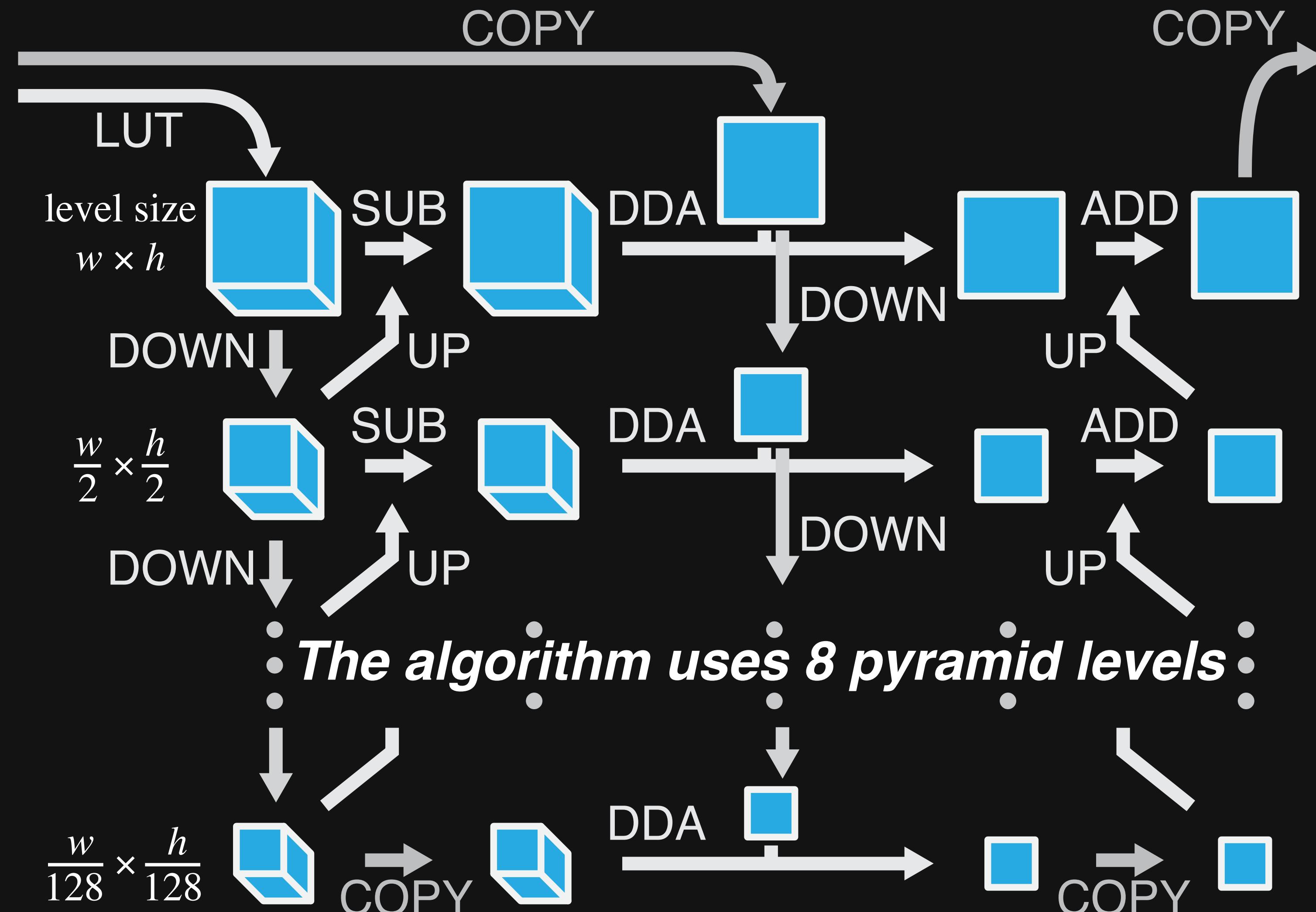
Reference:
300 lines C++

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

Same algorithm,
Different organization



Global reorganization breaks modularity



Halide

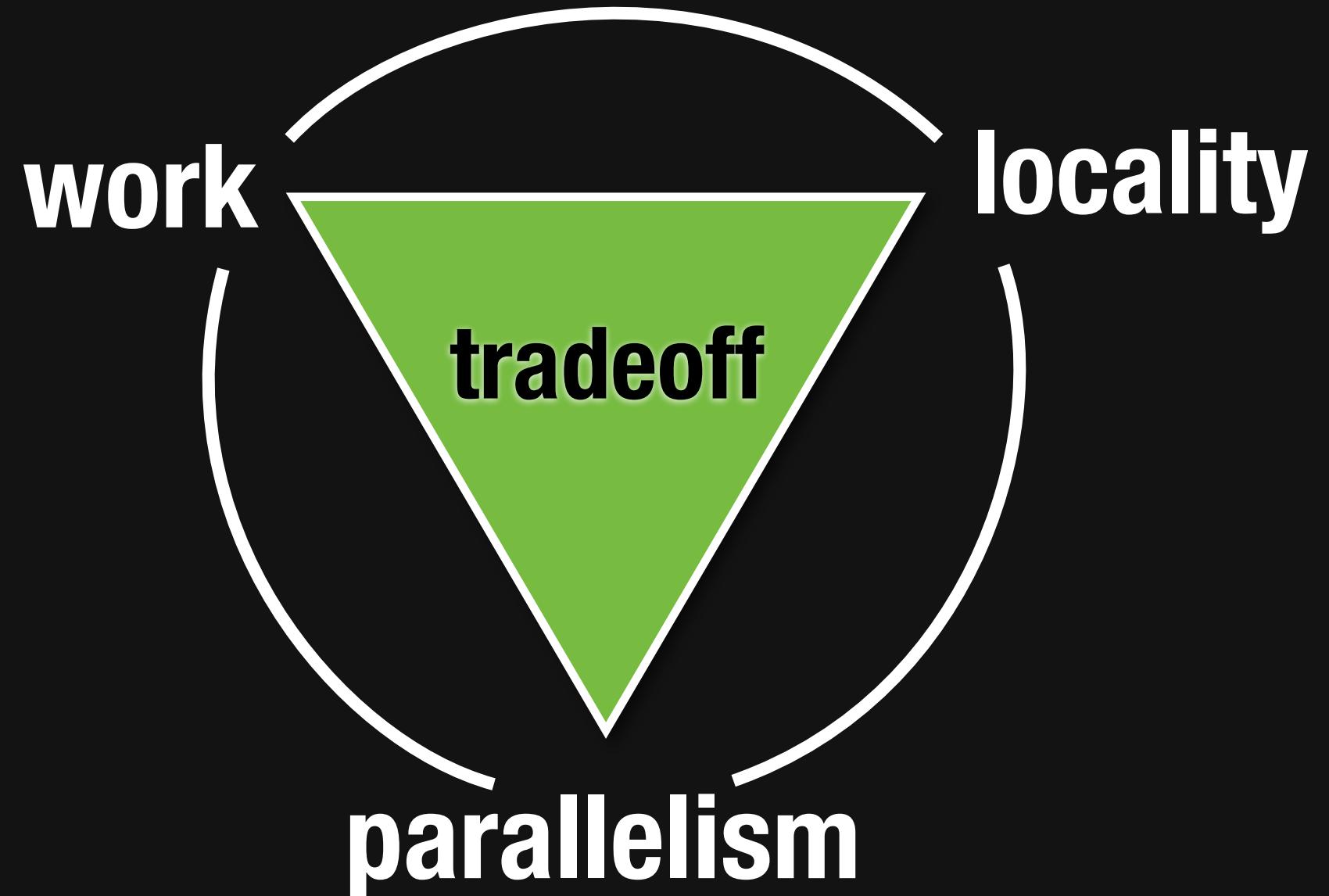
a language and compiler with decoupled organization

[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;
```

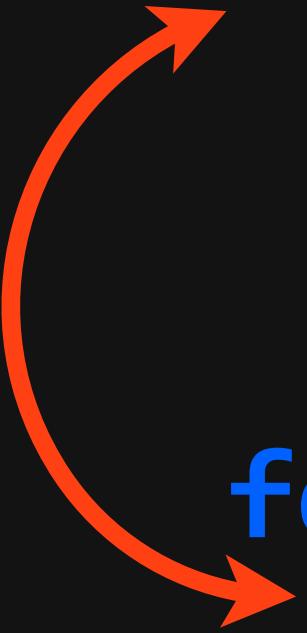
```
→ for (int y = 0; y < input.height(); y++)
→ for (int x = 0; x < input.width(); x++)
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
```

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_load_si128((__m128i*)(inPtr-1));
                    b = _mm_load_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_load_si128((__m128i*)(inPtr-1));
                    b = _mm_load_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_load_si128((__m128i*)(inPtr-1));
                    b = _mm_load_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

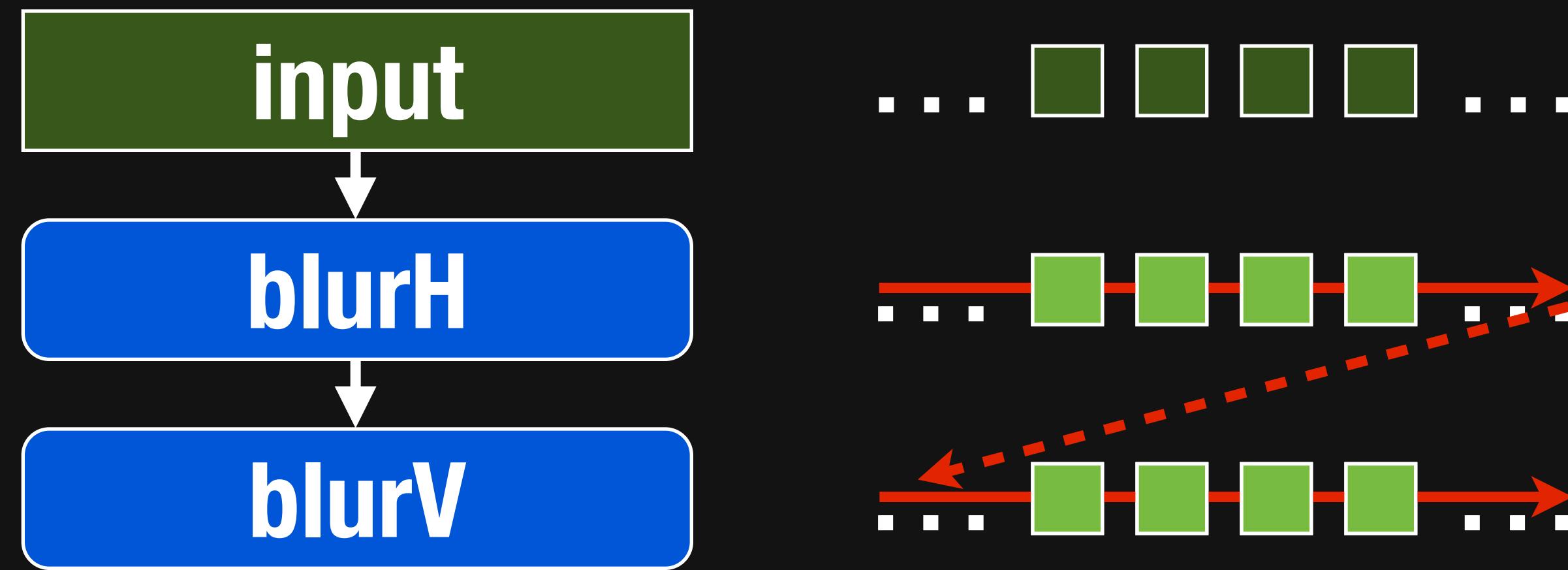
Complex graphs, deeper than traditional stencil computations

How can we organize
this computation?

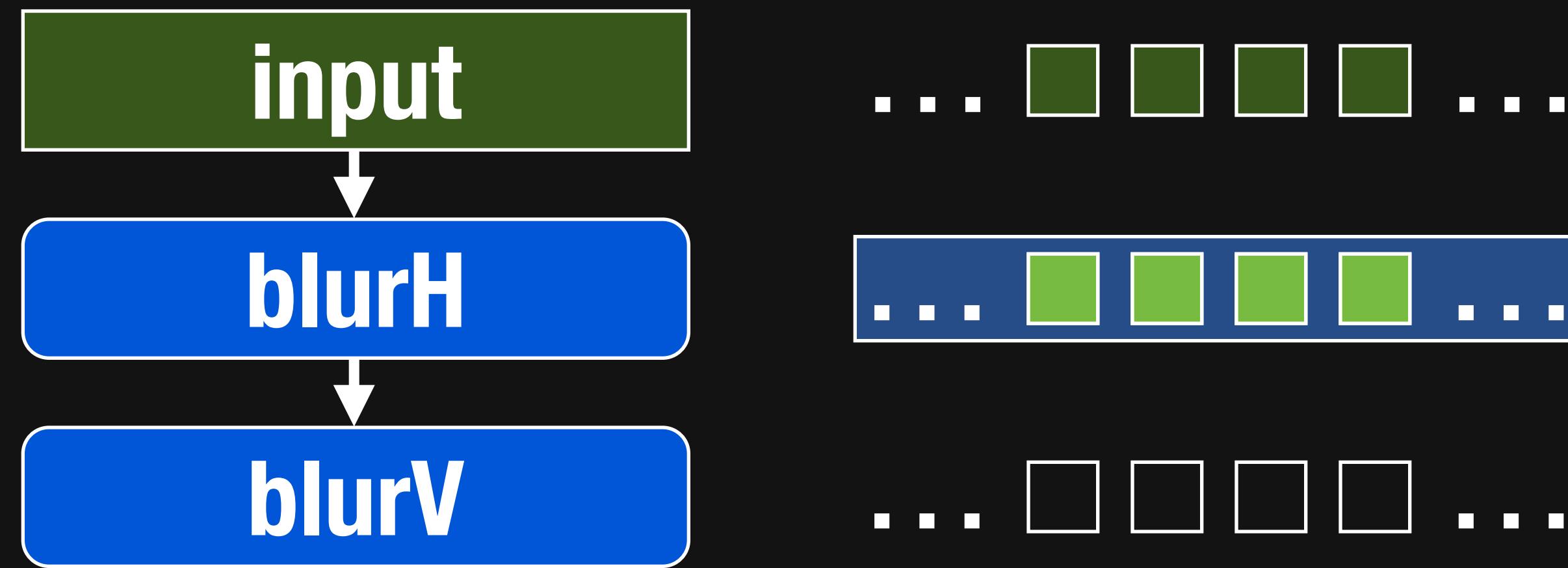
Organizing a data-parallel pipeline



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

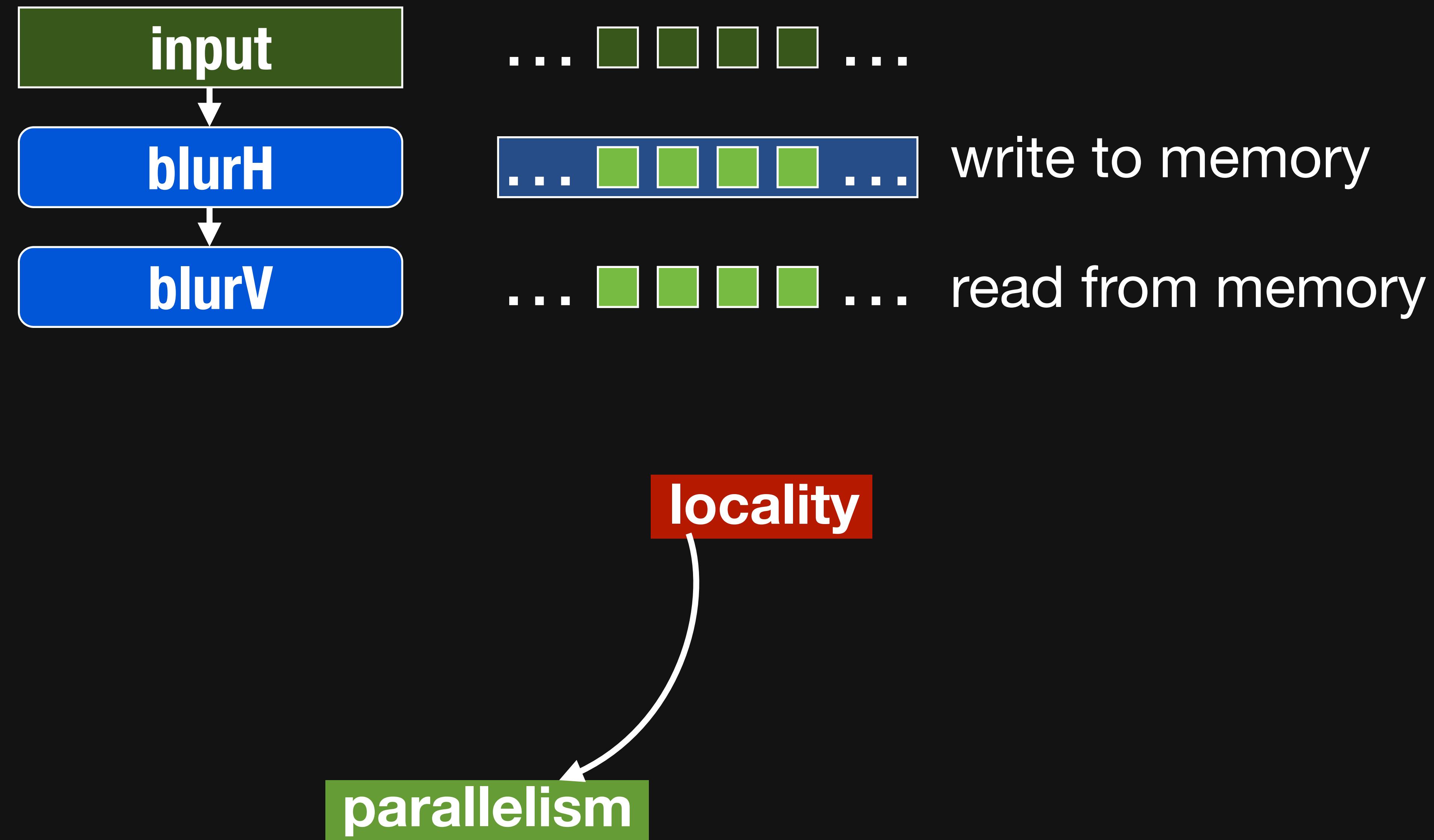


Simple loops execute **breadth-first** across stages

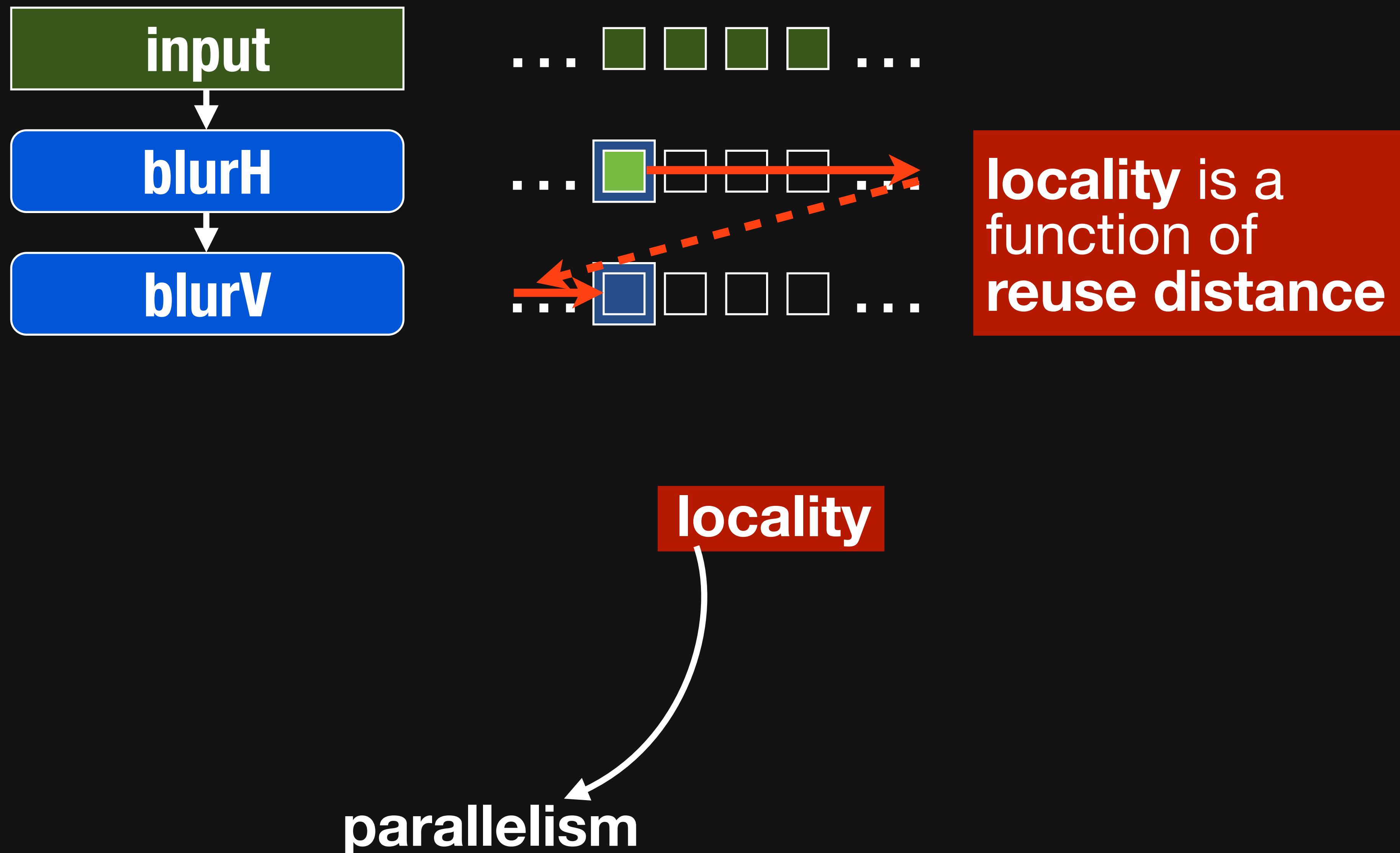


parallelism

Breadth-first execution sacrifices locality

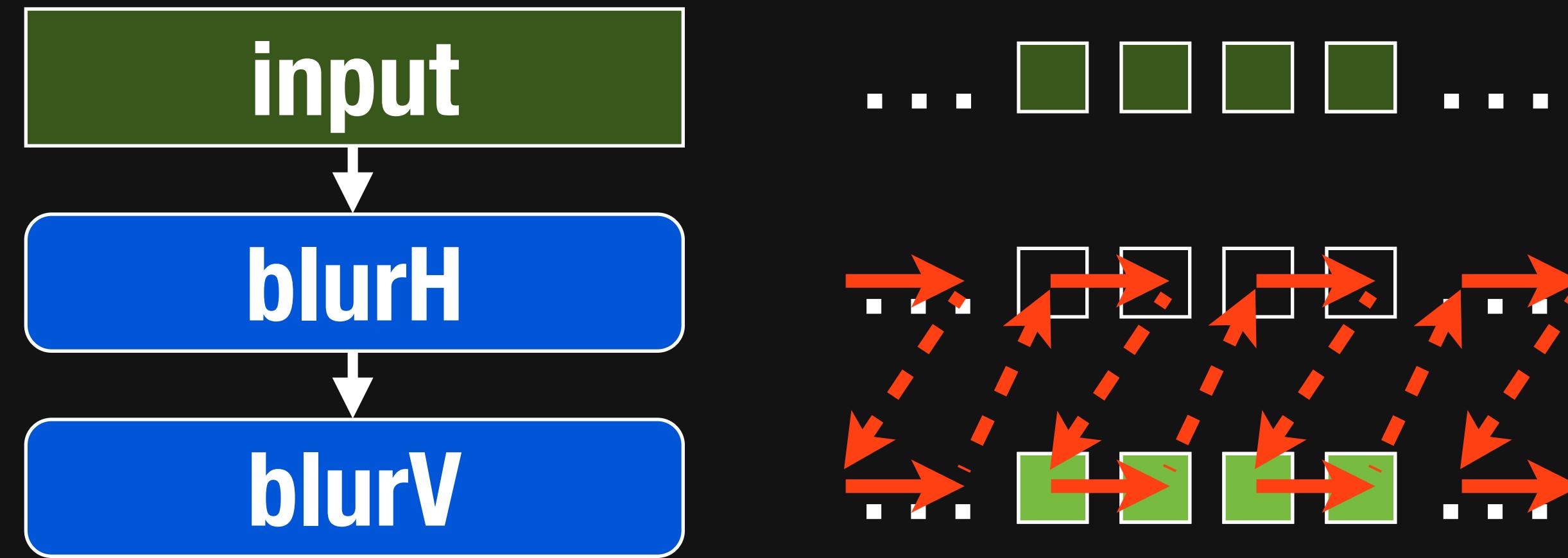


Breadth-first execution sacrifices locality



Interleaved execution (fusion) improves locality

*fusion globally
interleaves
computation*



reduce reuse
distance from
producer
to
consumer

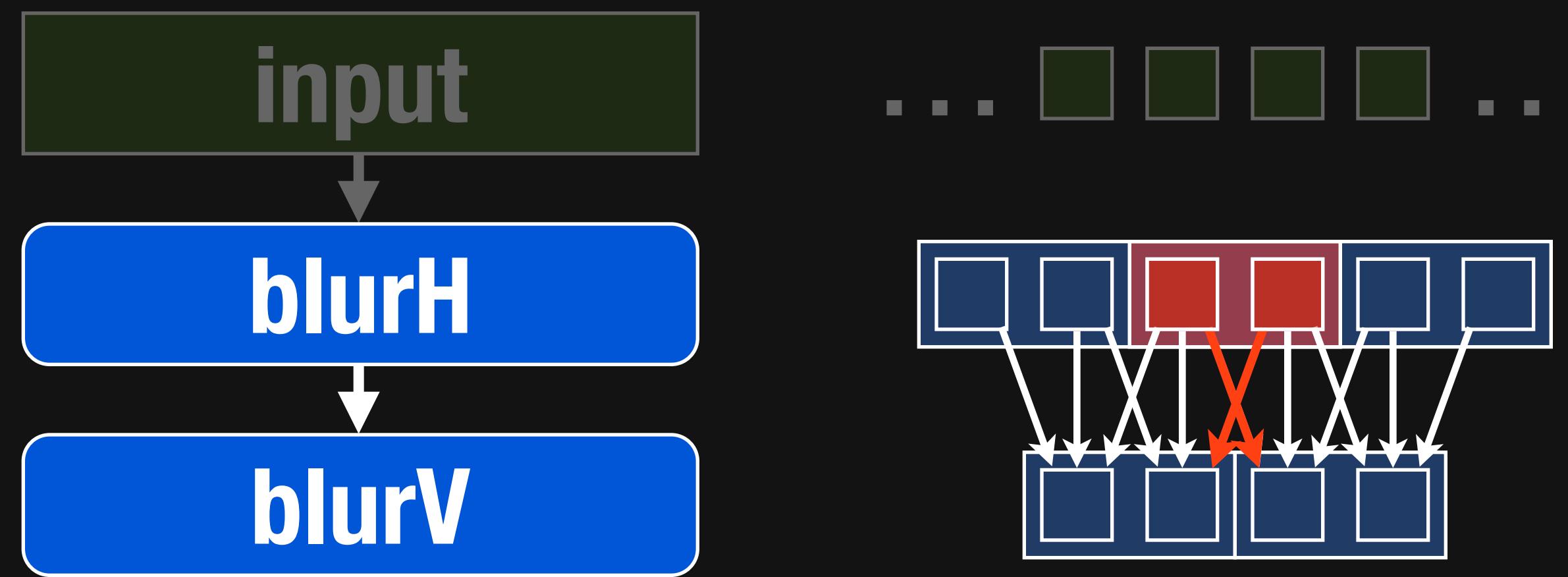
locality

parallelism

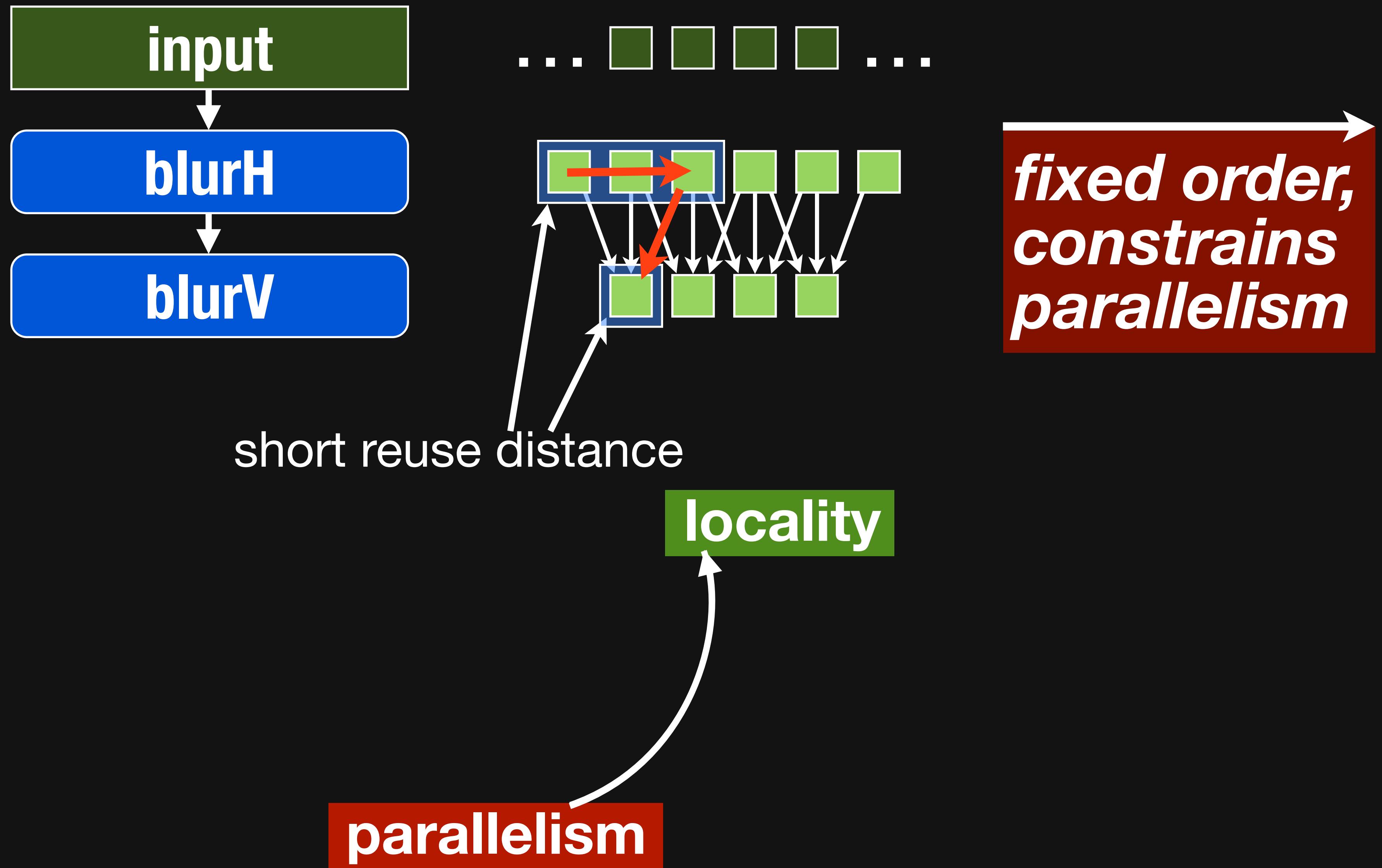
Understanding dependencies



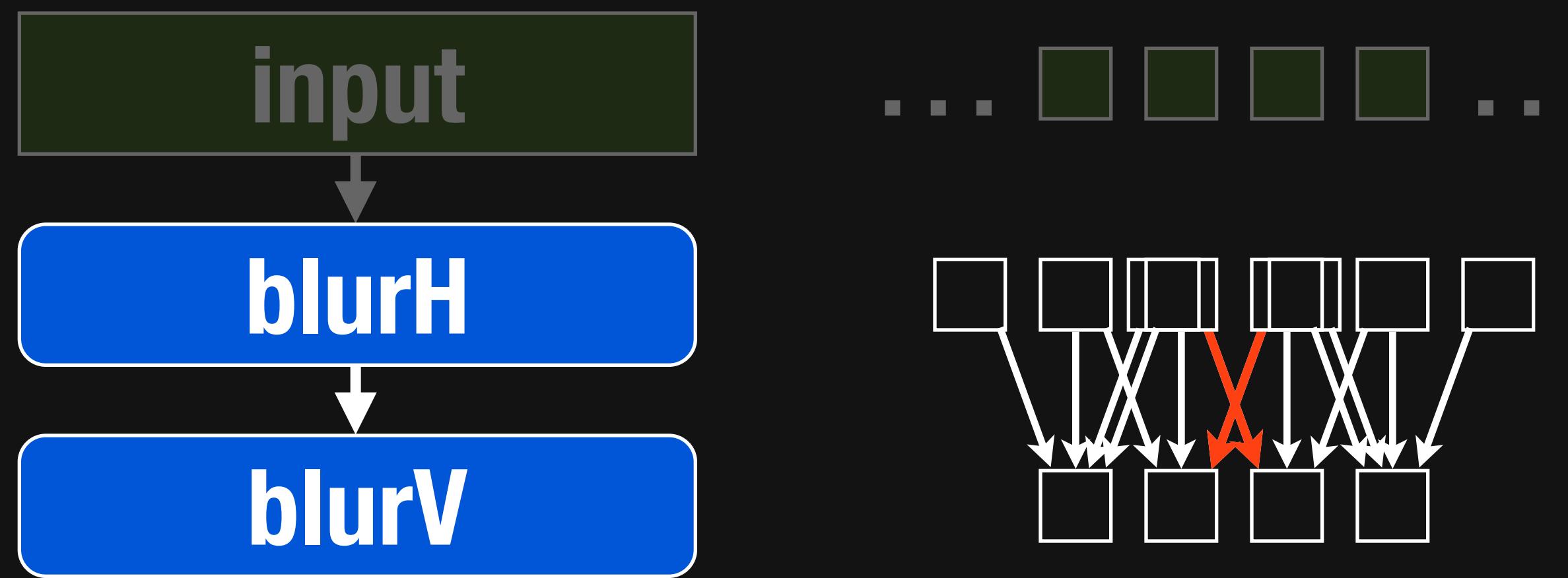
Stencils have overlapping dependencies



Sliding window execution sacrifices parallelism

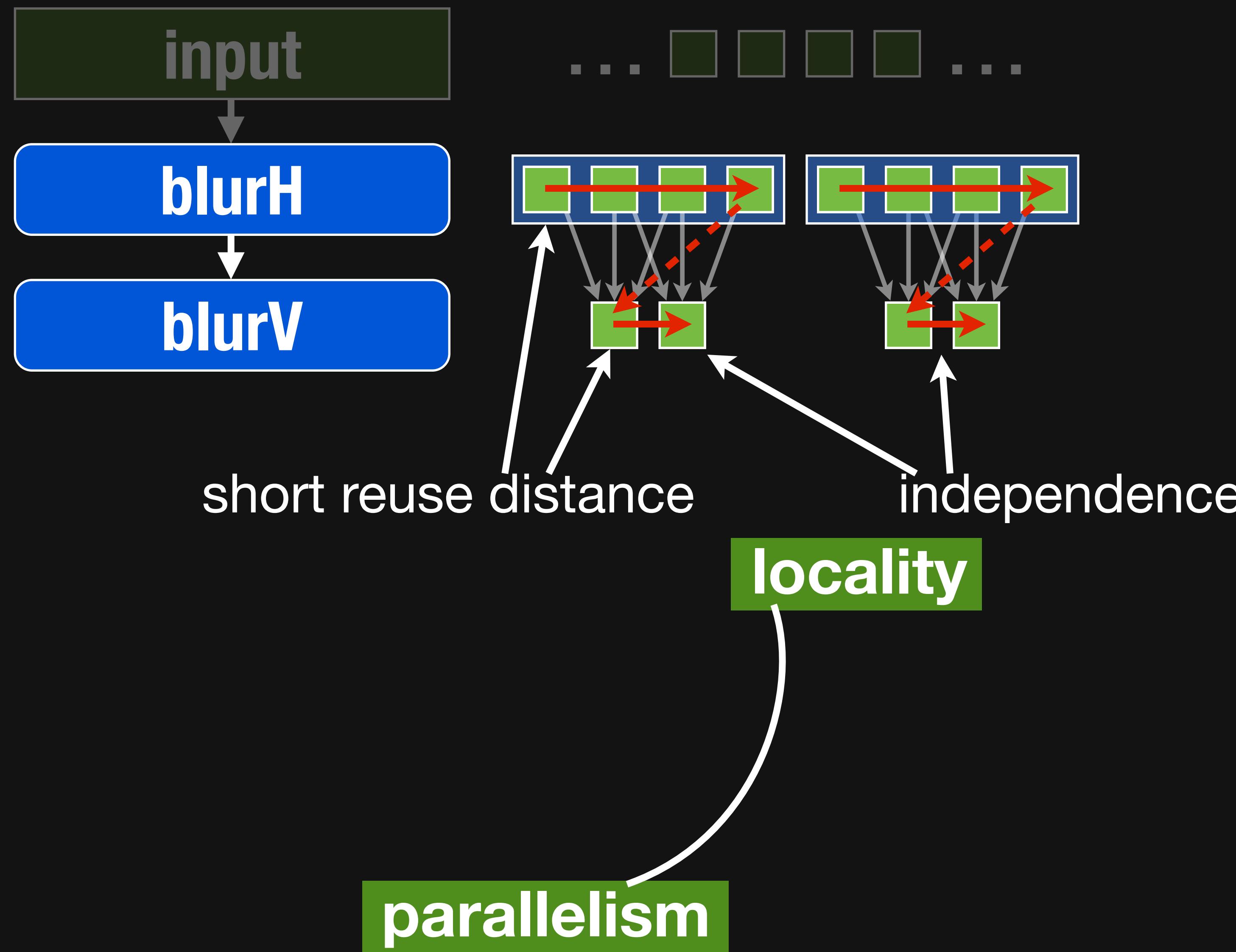


Breaking dependencies with tiling

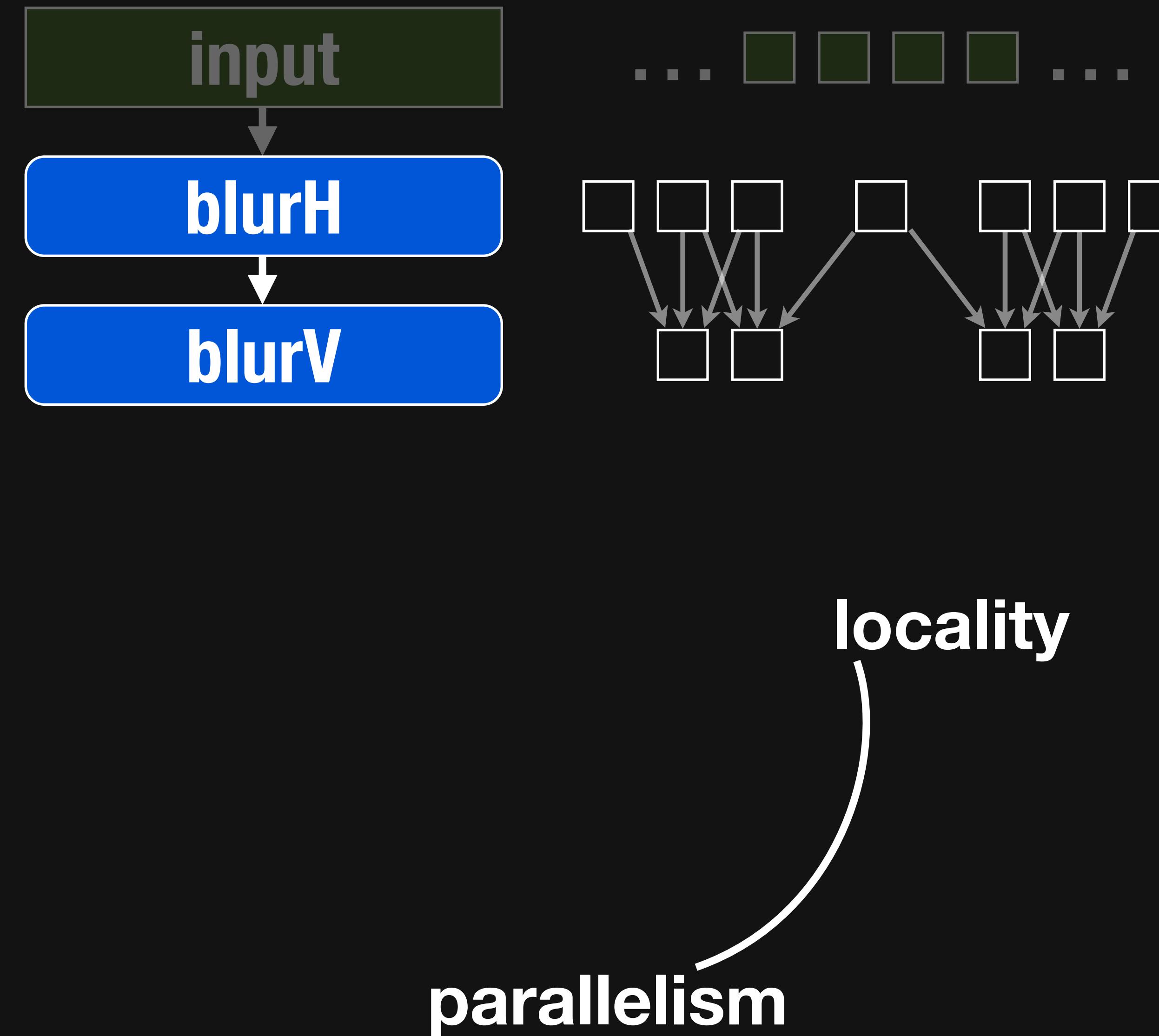


locality
parallelism

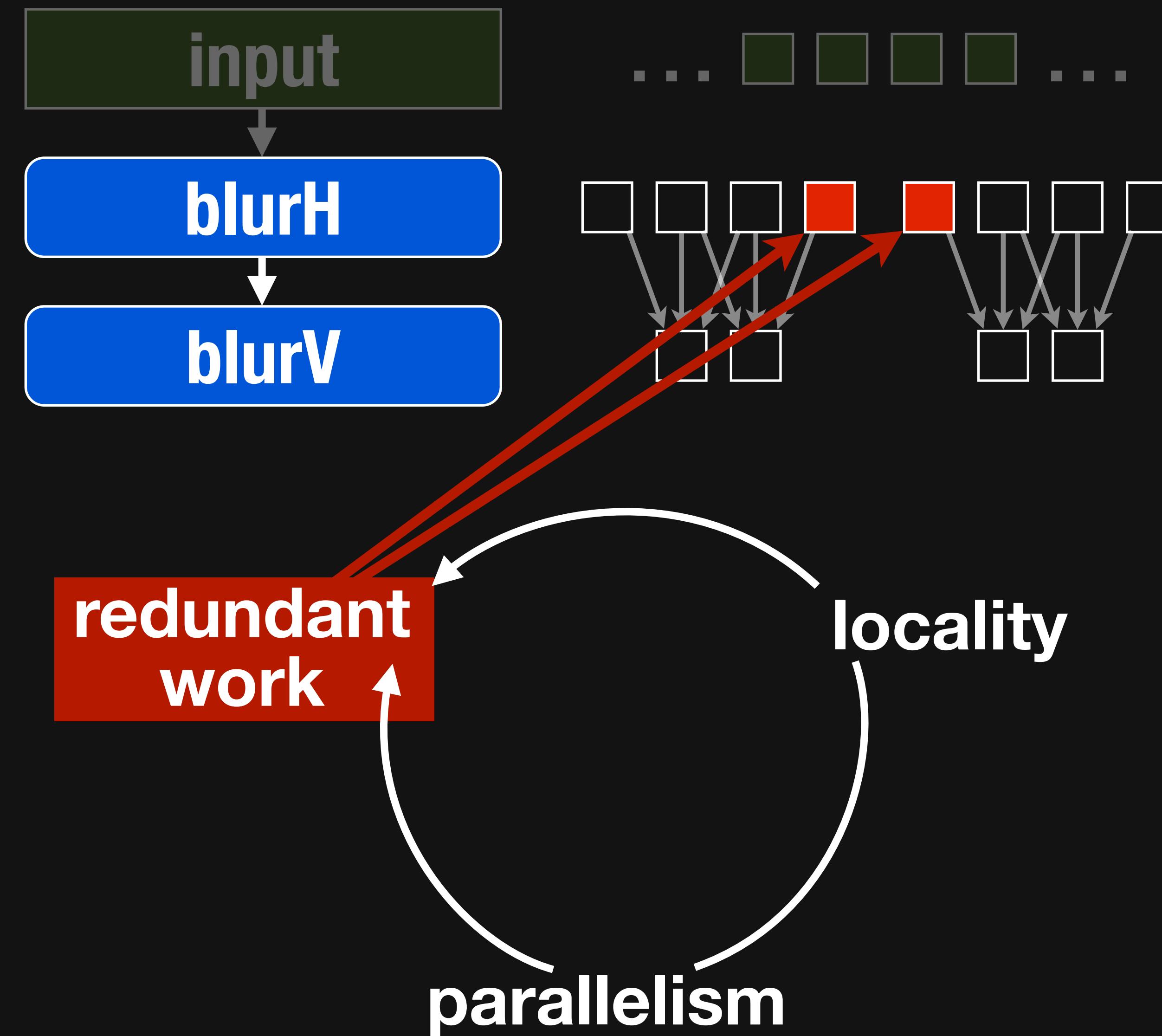
Decoupled tiles optimize parallelism & locality



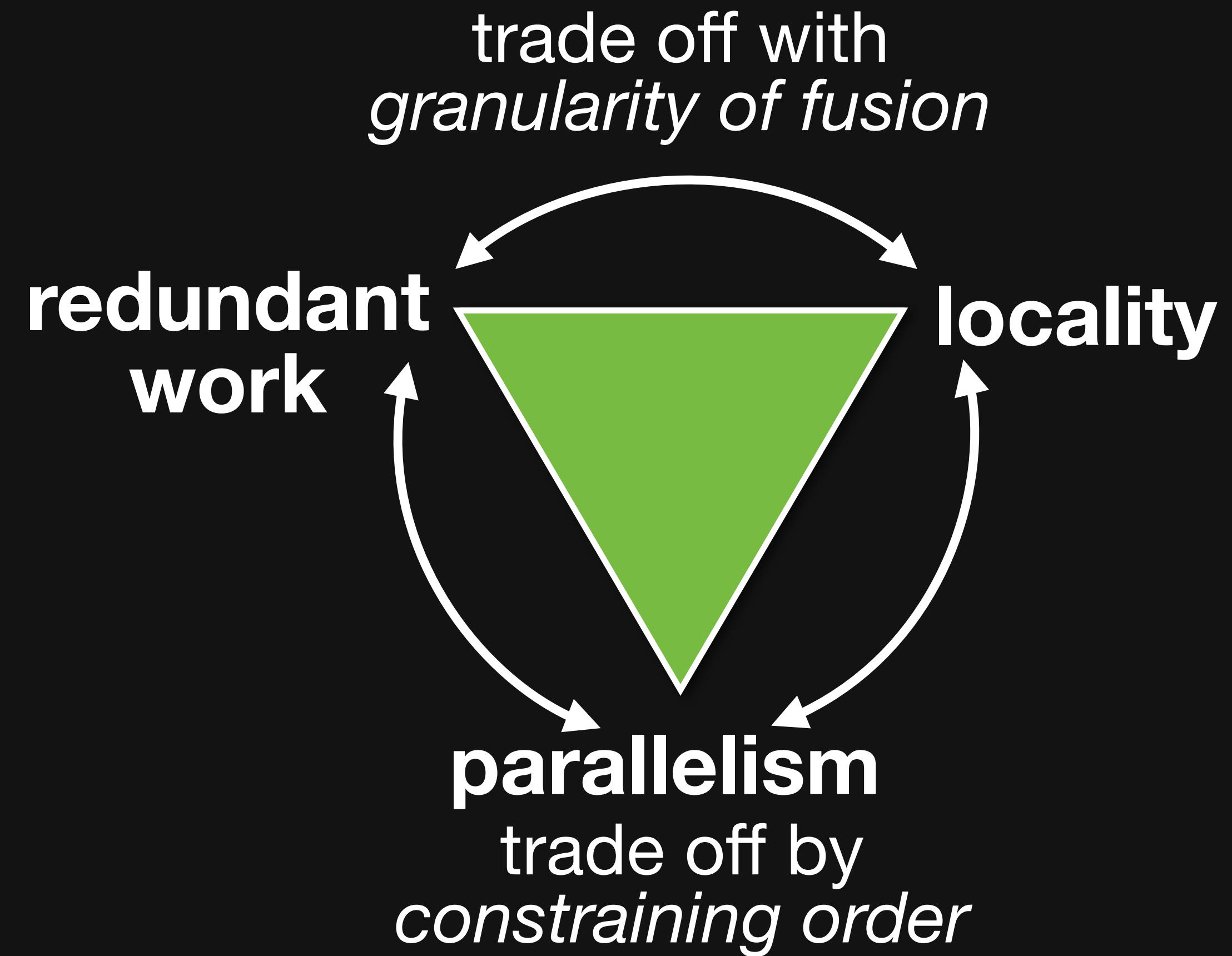
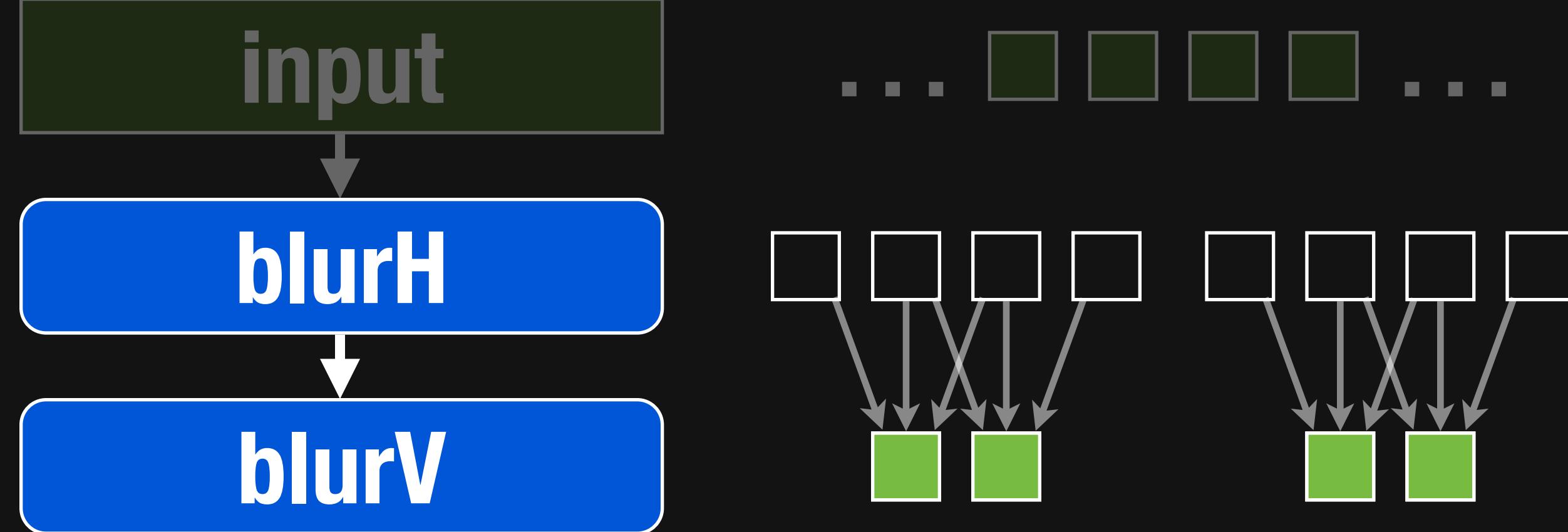
Breaking dependencies introduces redundant work



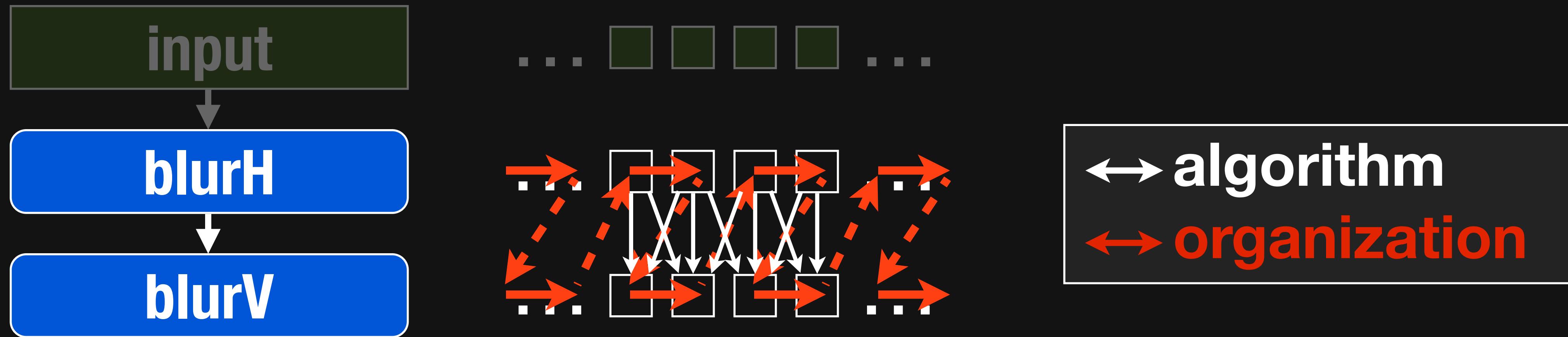
Breaking dependencies introduces redundant work



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



A language of schedules

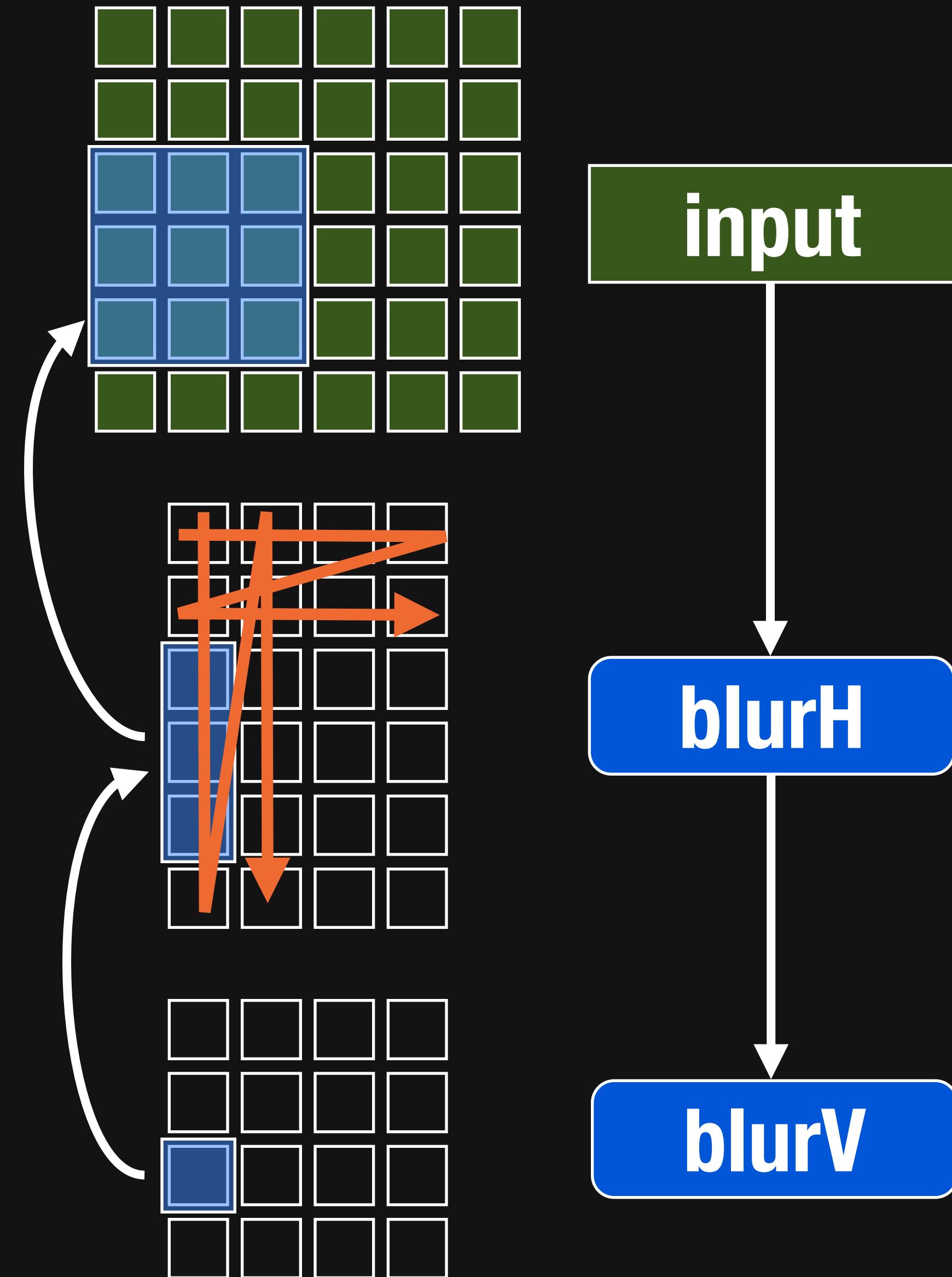
The schedule defines intra-stage order, inter-stage interleaving

For each stage:

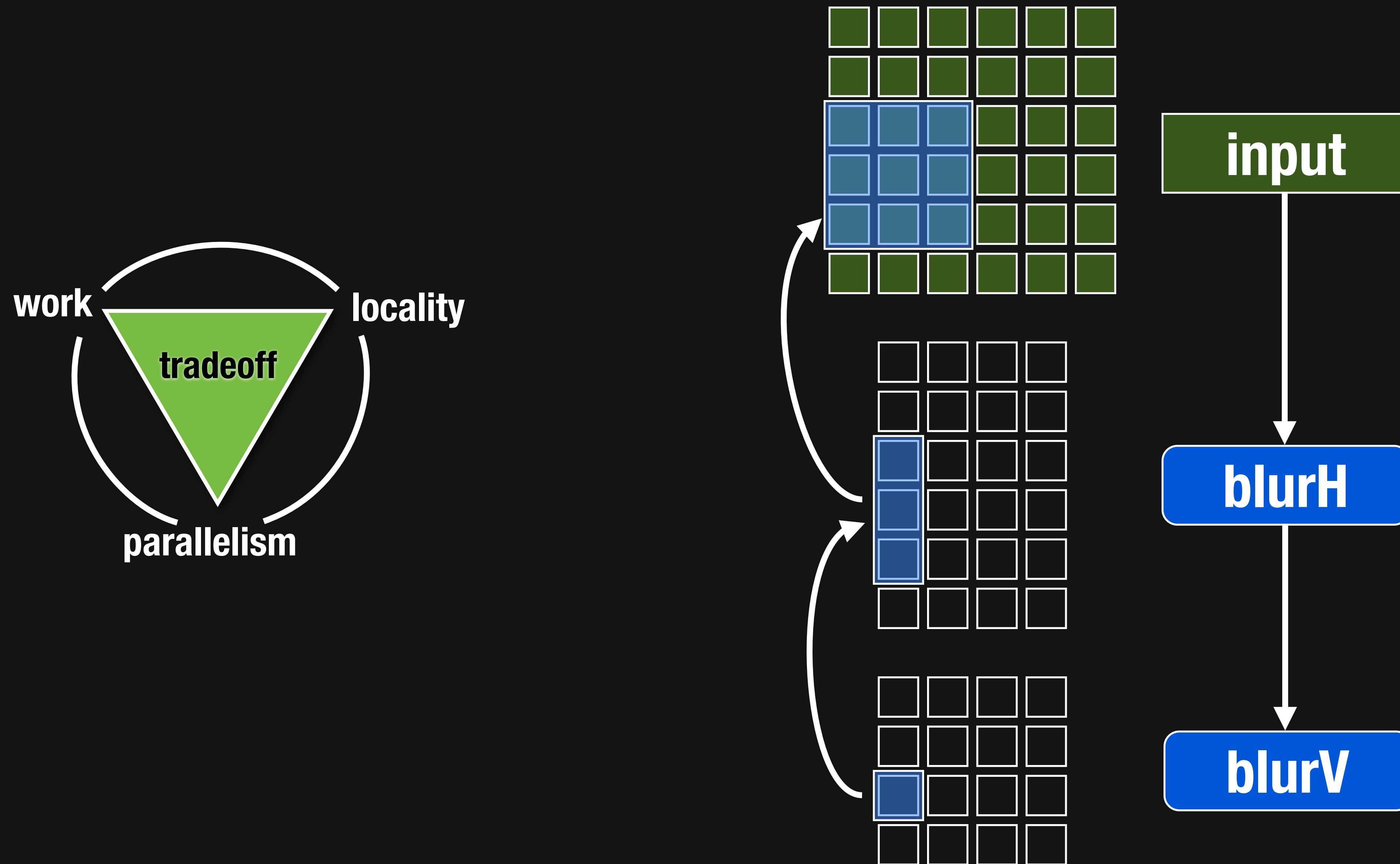
1) In what order should we compute its values?

2) When should we compute its inputs?

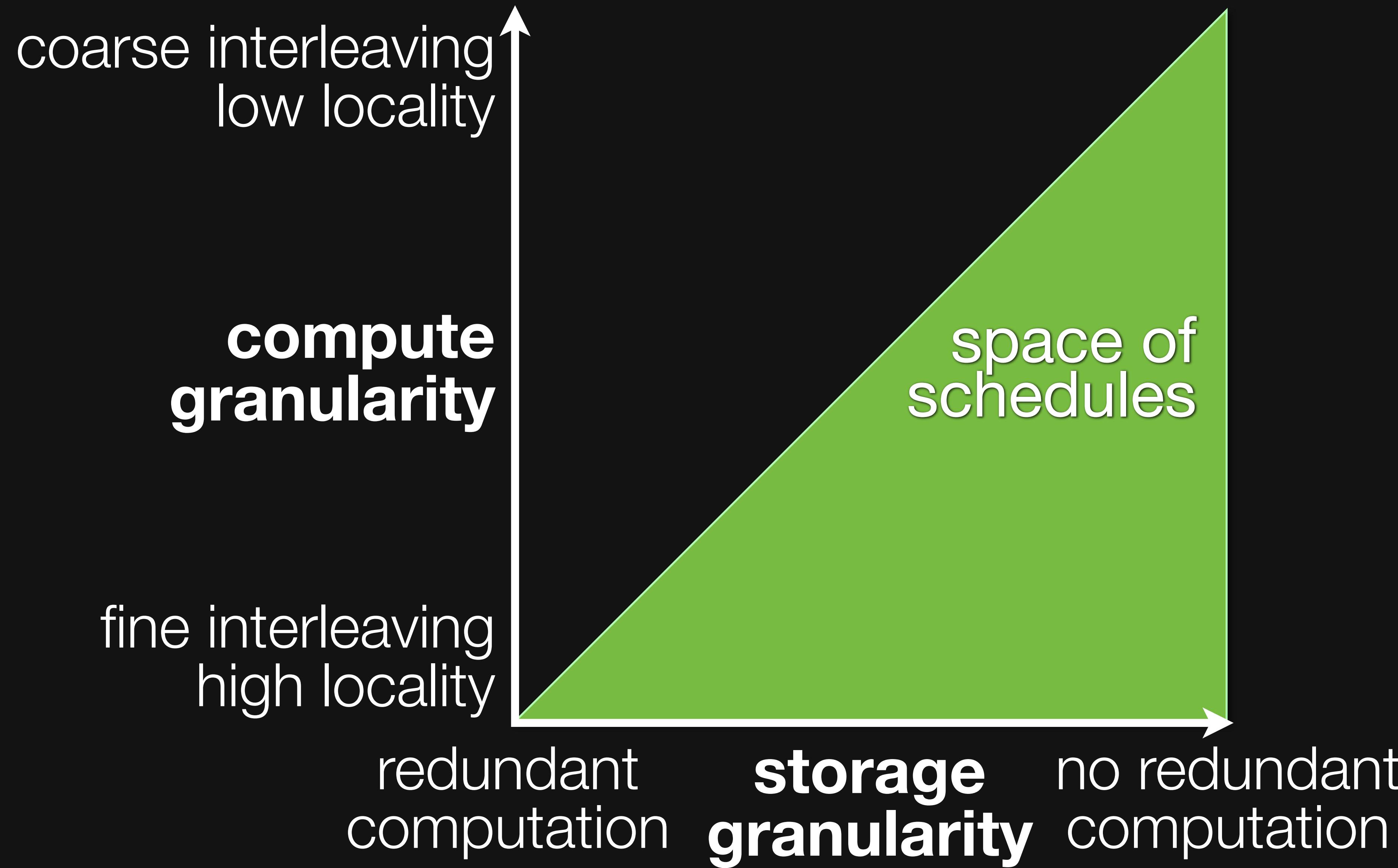
This is a language for scheduling choices.



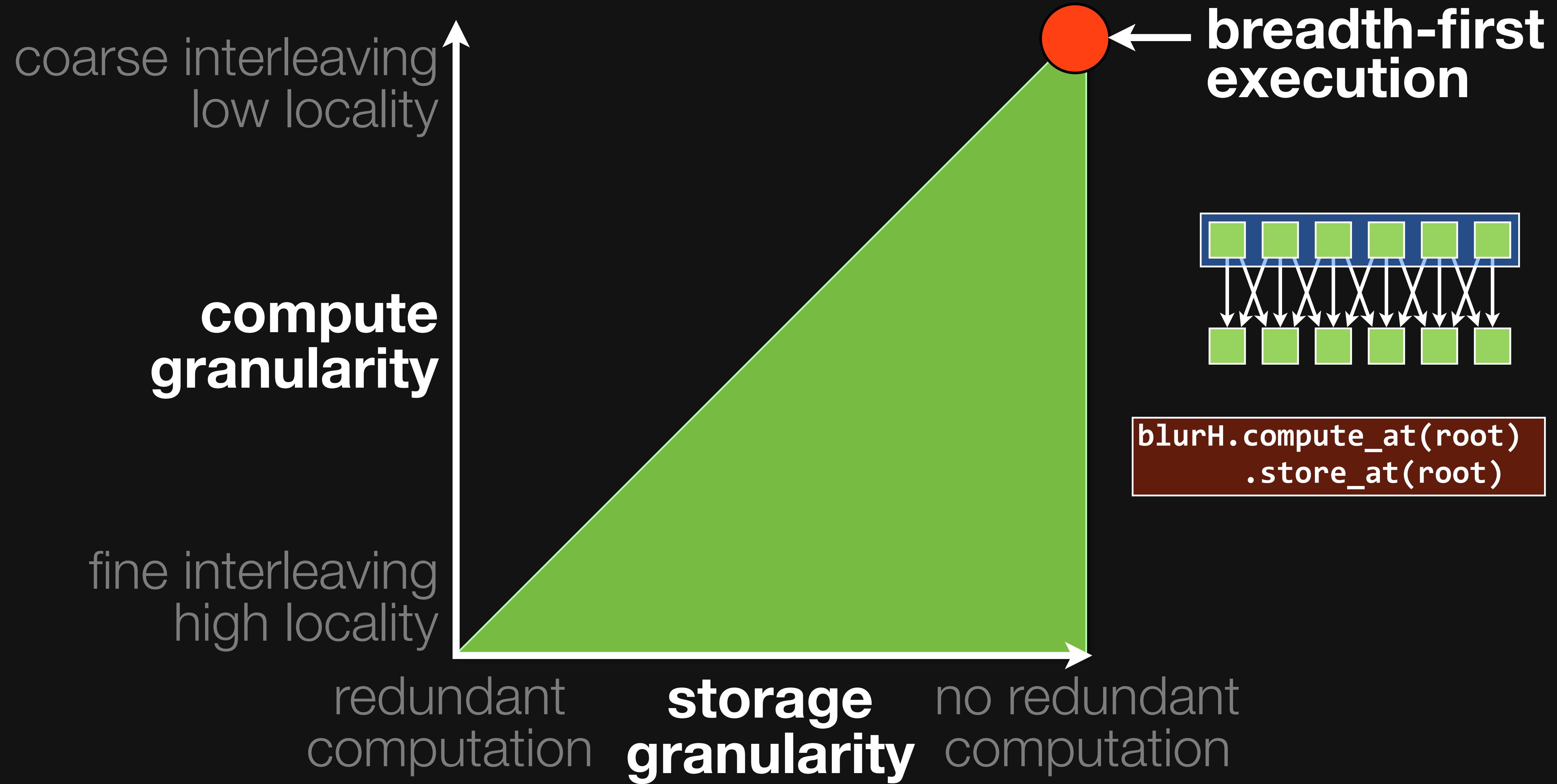
The schedule defines intra-stage order, inter-stage interleaving



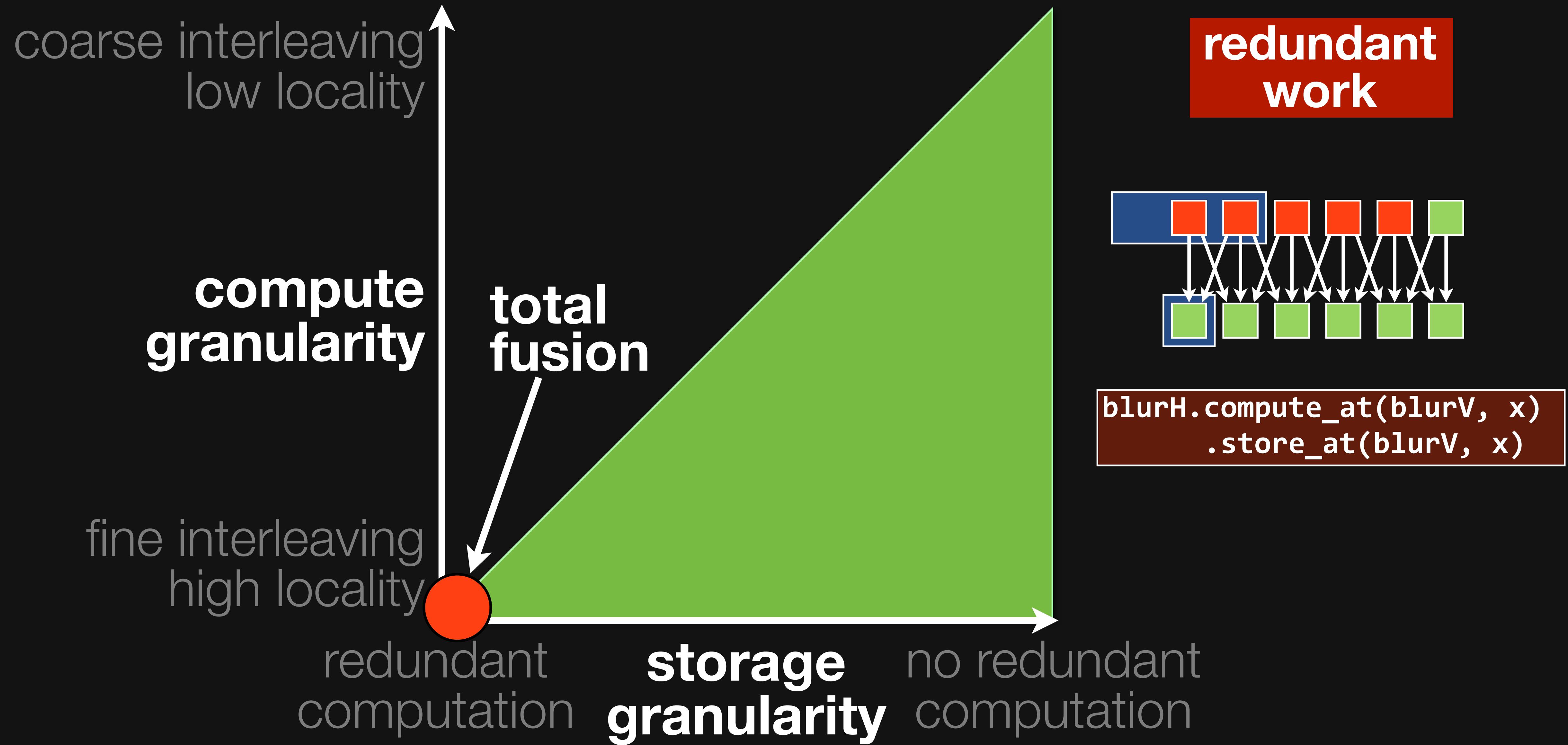
Tradeoff space modeled by granularity of interleaving



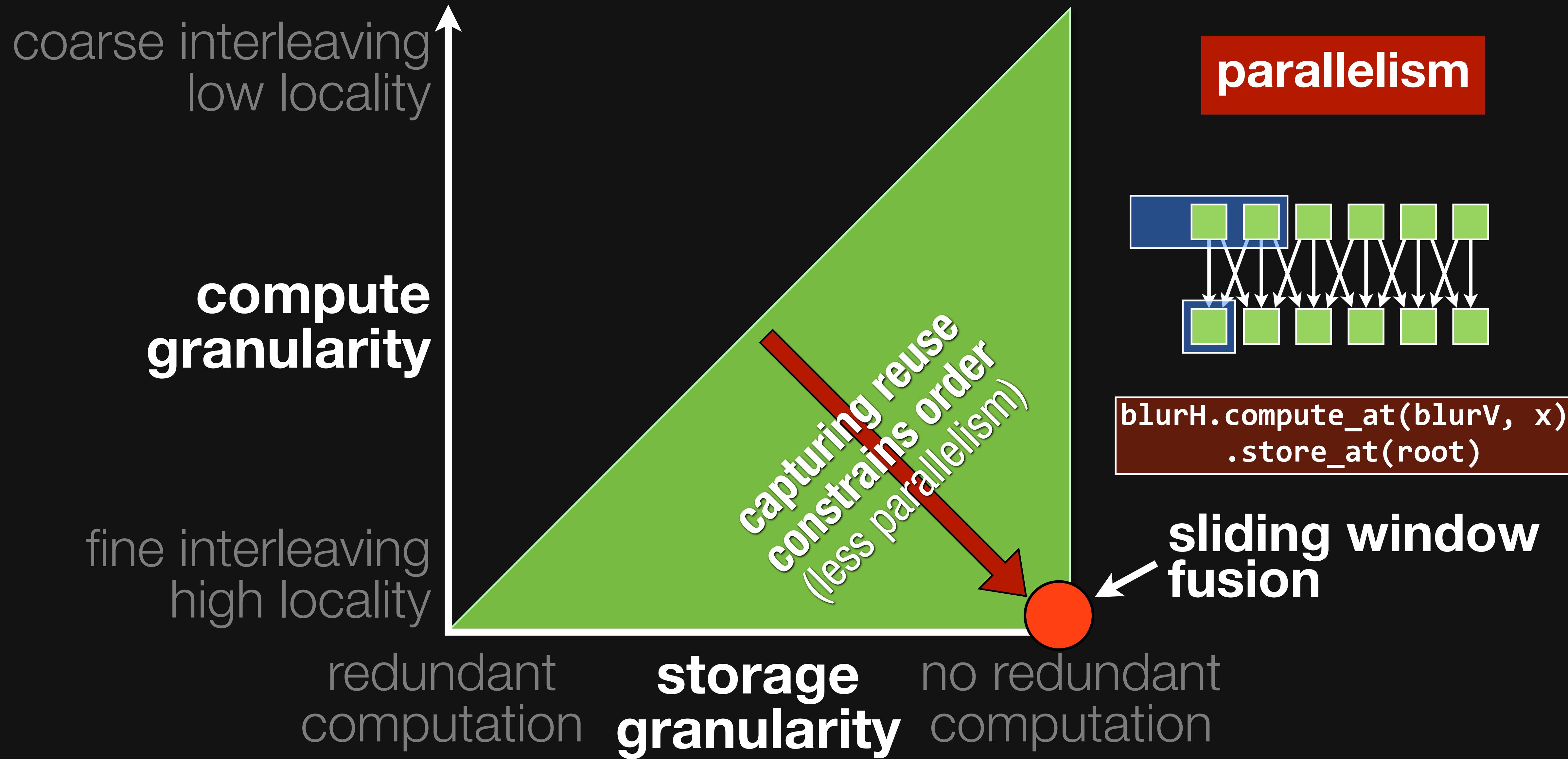
Tradeoff space modeled by granularity of interleaving



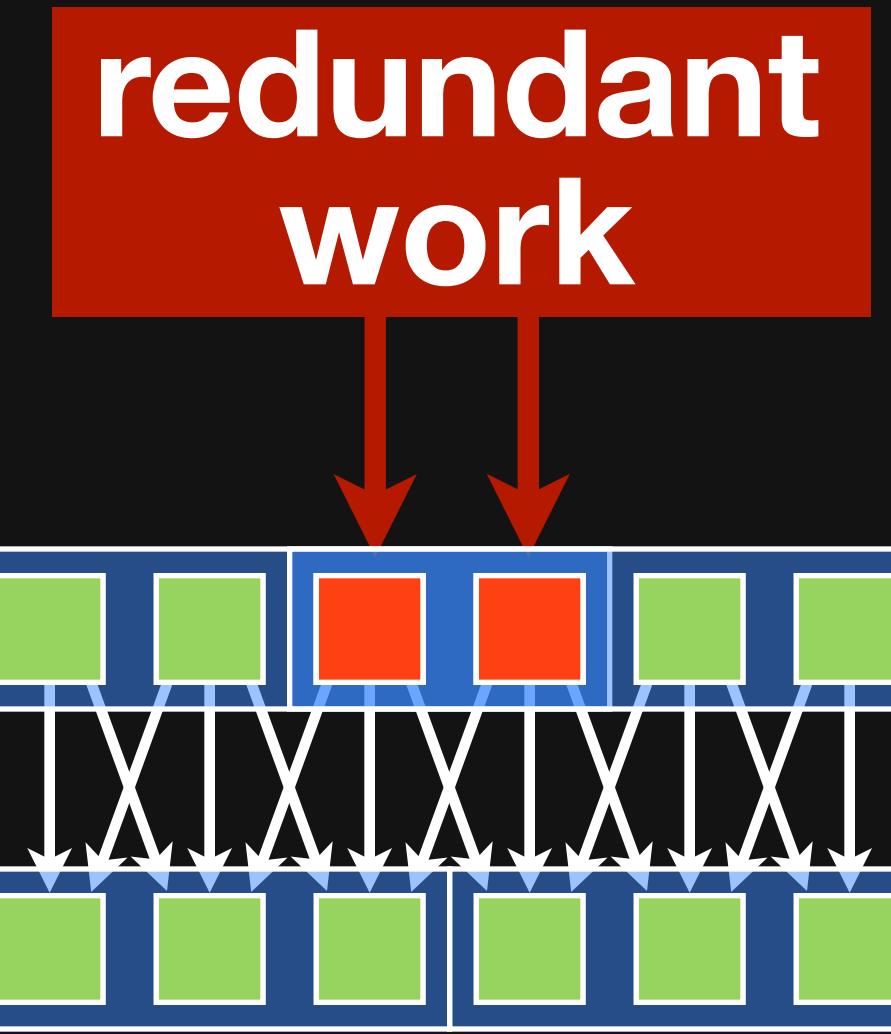
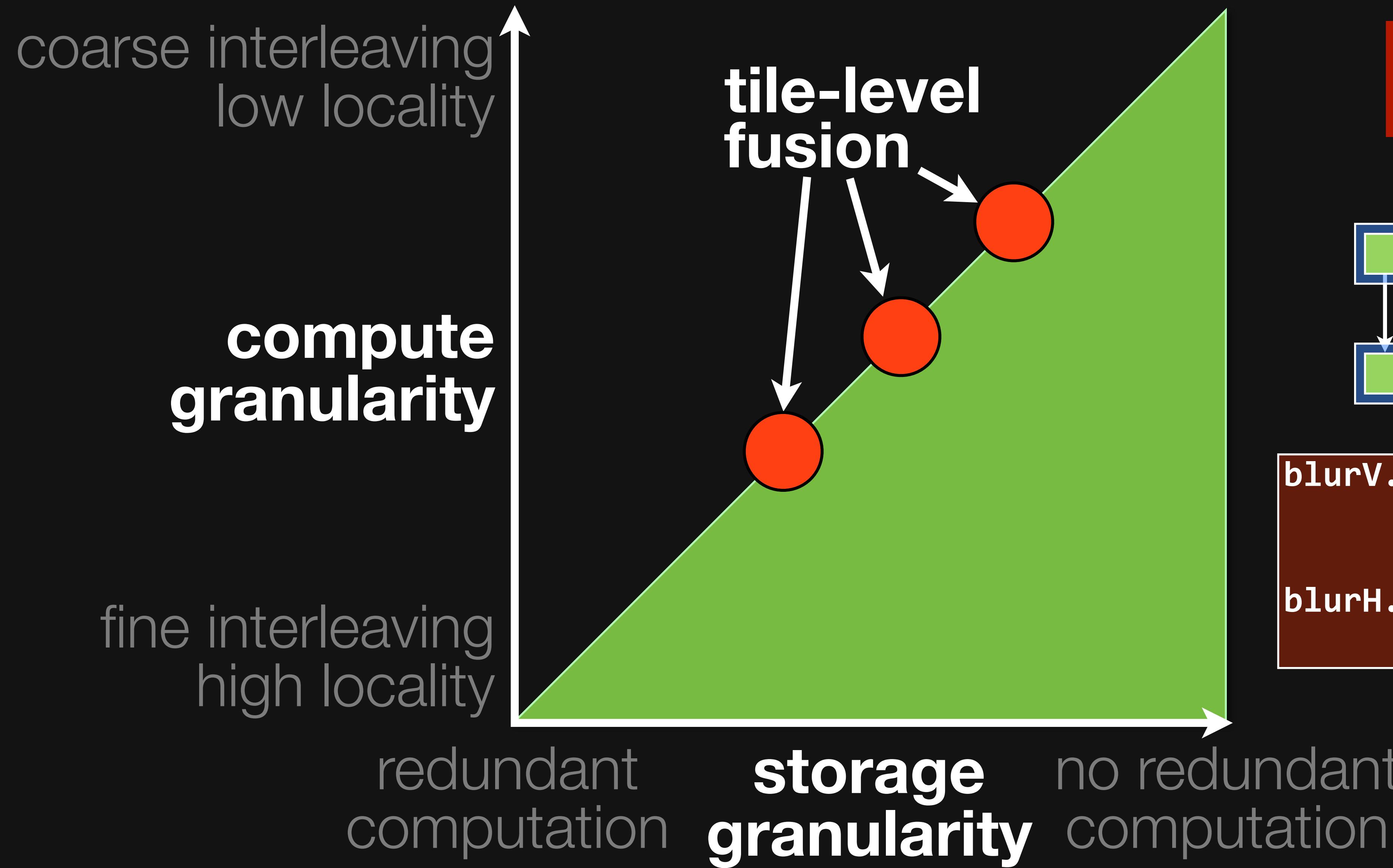
Tradeoff space modeled by granularity of interleaving



Tradeoff space modeled by granularity of interleaving

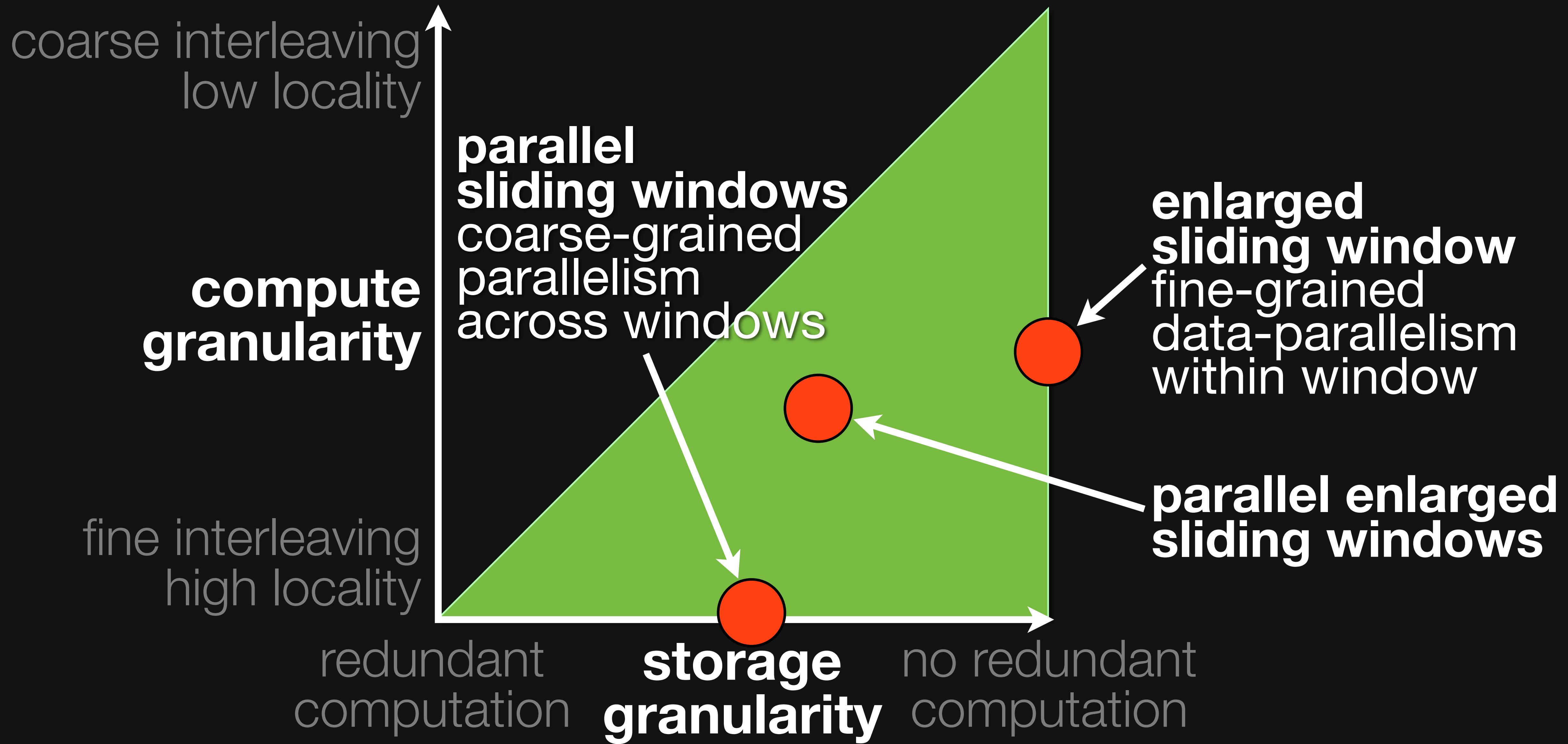


Tradeoff space modeled by granularity of interleaving

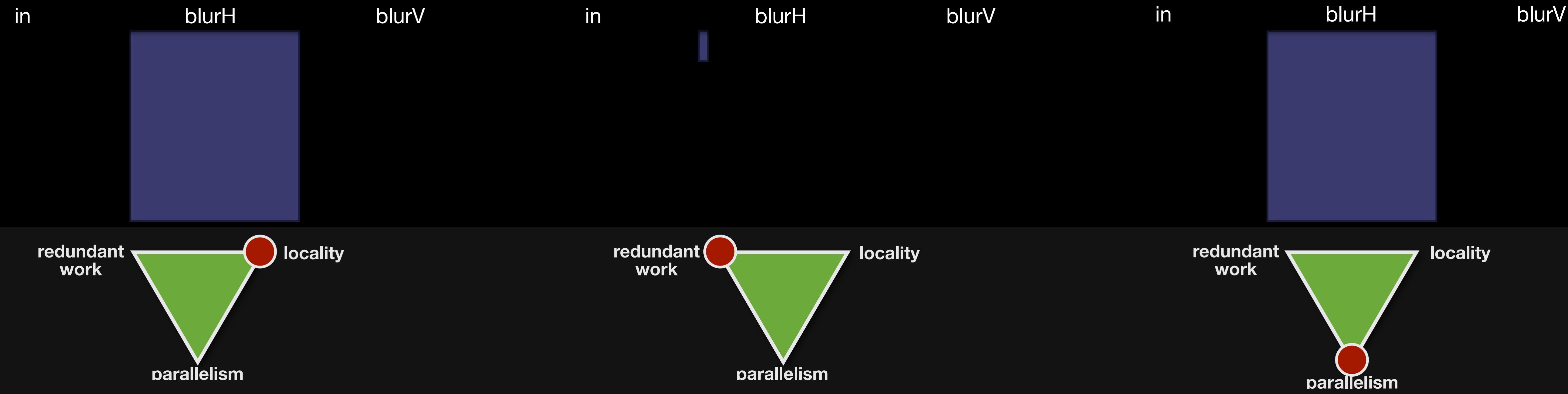


```
blurV.tile(xo, yo,  
xi, yi,  
w, H)  
blurH.compute_at(blurV, xo)  
.store_at(blurV, xo)
```

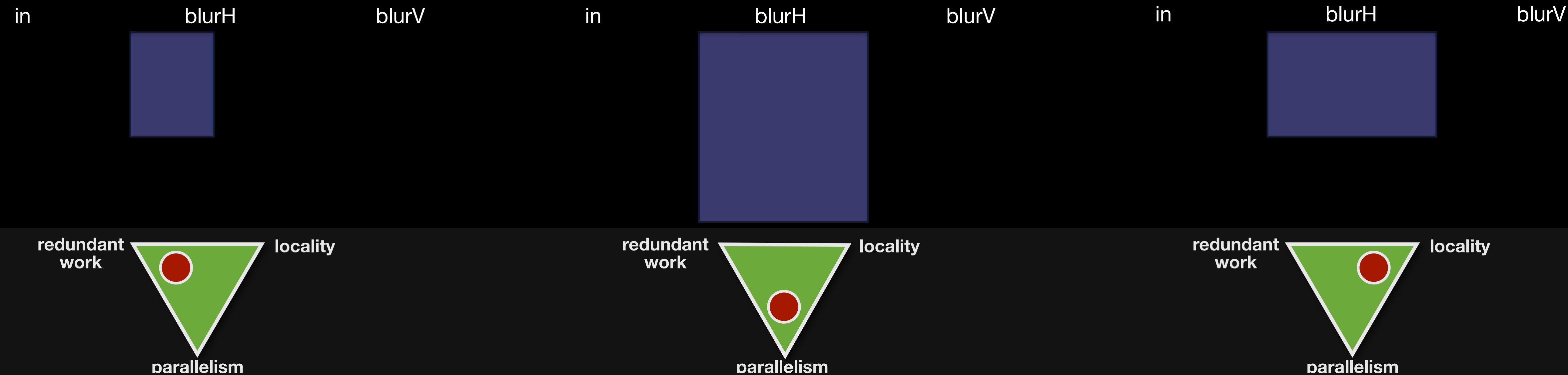
Tradeoff space modeled by granularity of interleaving



Schedule primitives **compose** to create many organizations



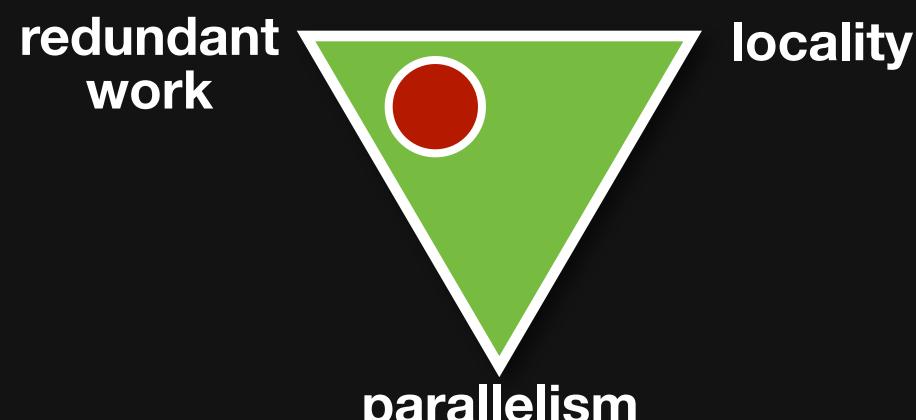
blurH.compute_at(blurV, y)



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
```

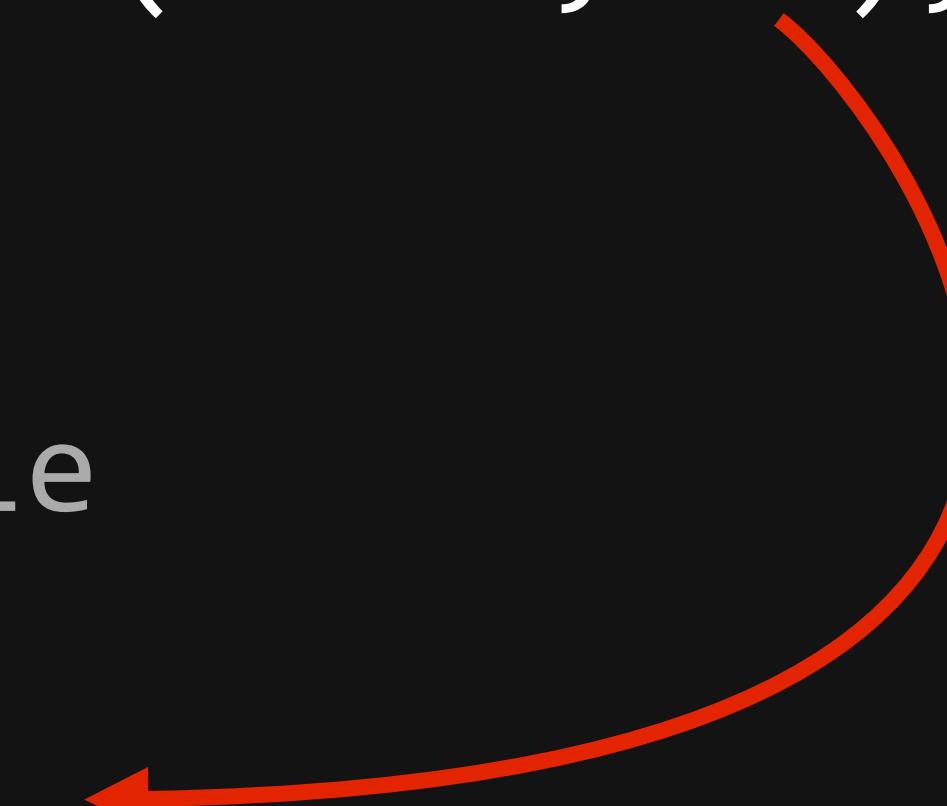
The Schedule defines a **loop nest** to compute the pipeline

```
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);
```

```
// for each tile  
for blurV.yo:  
    for blurV.xo:  
        // 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);  
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

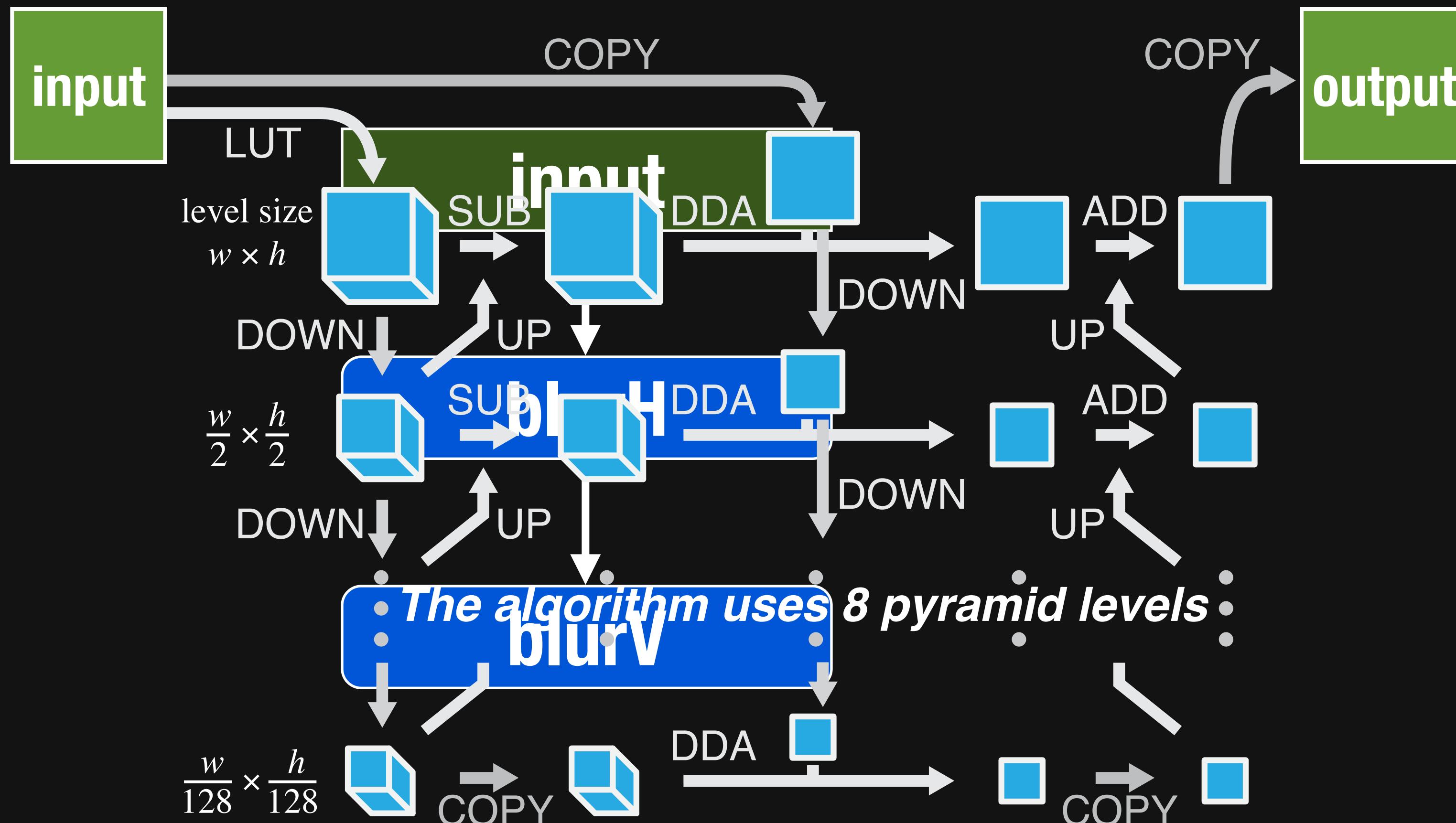
```
Func box_filter_3x3(Func in) {
    Func blurH, blurV;
    Var x, y, xi, yi;
    // The algorithm - no storage, order
    blurH(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurV(x, y) = (blurH(x, y-1) + blurH(x, y) + blurH(x, y+1))/3;
    // The schedule - defines order, locality; implies 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	$O(x,y,k) \leftarrow \text{lut}(I(x,y) - k\sigma)$	UP: upsample	$T_1(2x,2y) \leftarrow I(x,y)$
			$T_2 \leftarrow T_1 \otimes_x [1 \ 3 \ 3 \ 1]$
			$O \leftarrow T_2 \otimes_y [1 \ 3 \ 3 \ 1]$
ADD: addition	$O(x,y) \leftarrow I_1(x,y) + I_2(x,y)$	DOWN: downsample	$T_1 \leftarrow I \otimes_x [1 \ 3 \ 3 \ 1]$
			$T_2 \leftarrow T_1 \otimes_y [1 \ 3 \ 3 \ 1]$
SUB: subtraction	$O(x,y) \leftarrow I_1(x,y) - I_2(x,y)$		$O(x,y) \leftarrow T_2(2x,2y)$
DDA: data-dependent access	$k \leftarrow \text{floor}(I_1(x,y) / \sigma)$		
	$\alpha \leftarrow (I_1(x,y) / \sigma) - k$		
	$O(x,y) \leftarrow (1-\alpha) I_2(x,y,k) + \alpha I_2(x,y,k+1)$		

local Laplacian filters
[Paeth et al., 2011]

Adobe: 1500 lines
expert-tuned C++
multi-threaded, SSE

3 months of work

10x faster than original C++

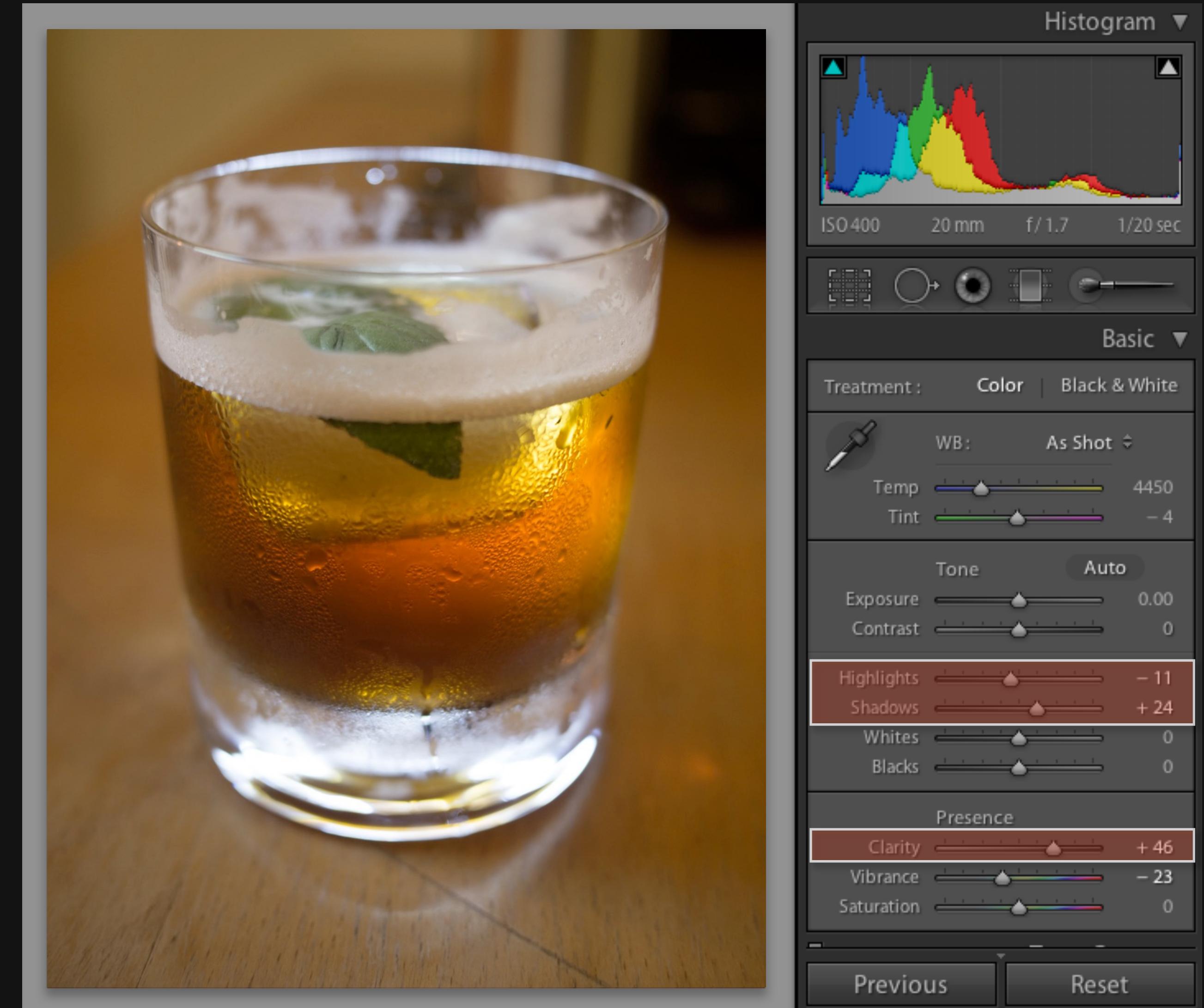
Halide: 60 lines

1 day

Halide vs. Adobe:
2x faster on same CPU

Local Laplacian Filters

in Adobe Photoshop

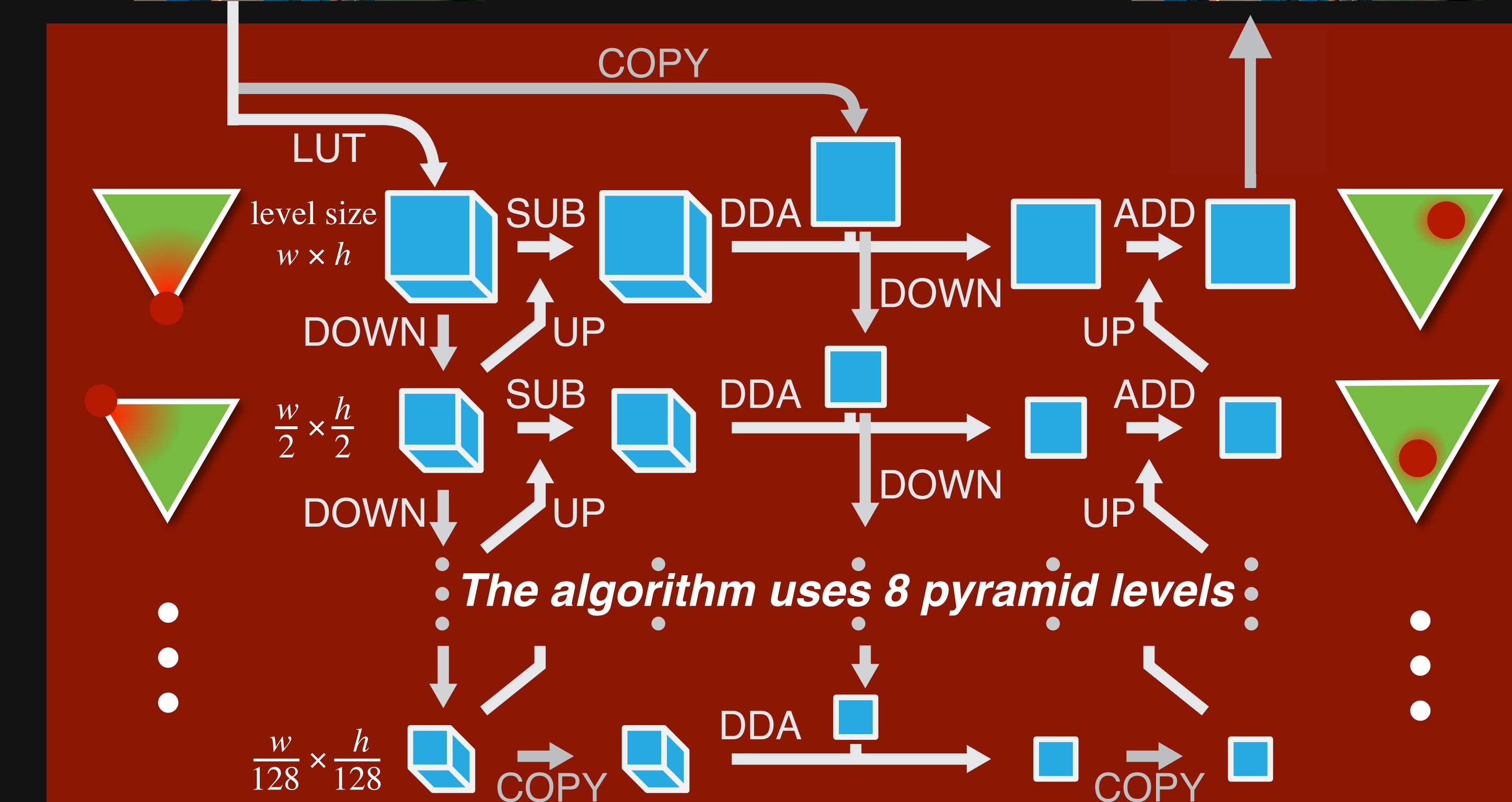


Adobe: 1500 lines
expert-tuned C++
multi-threaded, SSE
3 months of work
10x faster than original C++



Halide: 60 lines
1 day

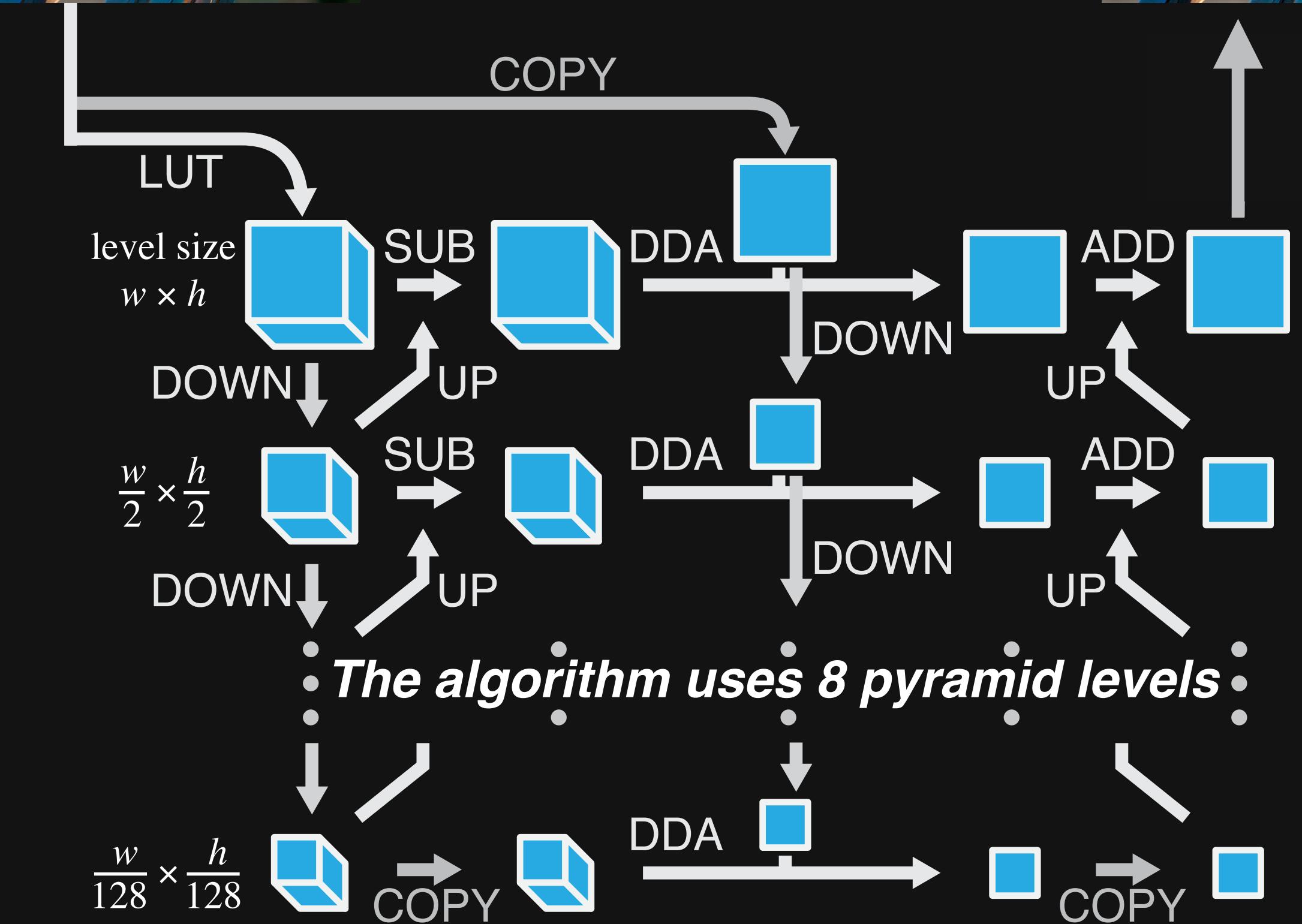
Halide vs. Adobe:
2x faster on same CPU



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



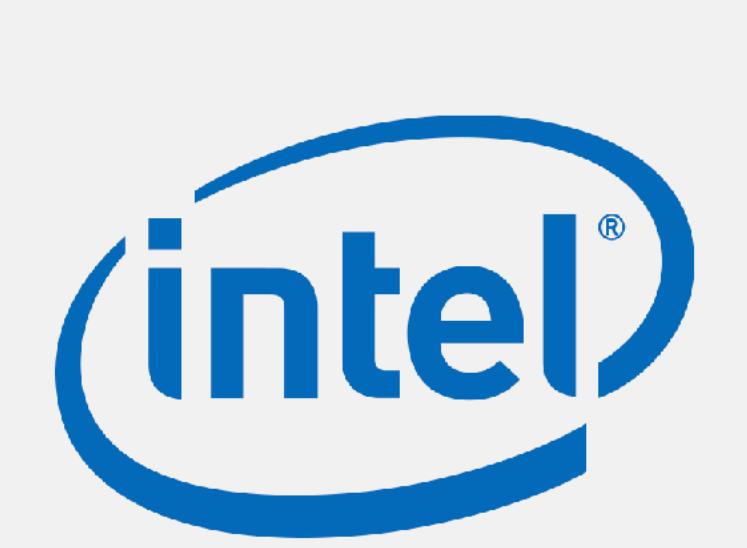
Halide: 60 lines
1 day
Halide vs. Adobe:
2x faster on same CPU
10x faster on GPU





Real-world adoption

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



Google
> 2000 pipelines
10s of kLOC in production

YouTube

Tensor Comprehensions



User-schedulable languages
are spreading

Malmo stvm

TACO GraphIt

User-Schedulable Languages 2.0

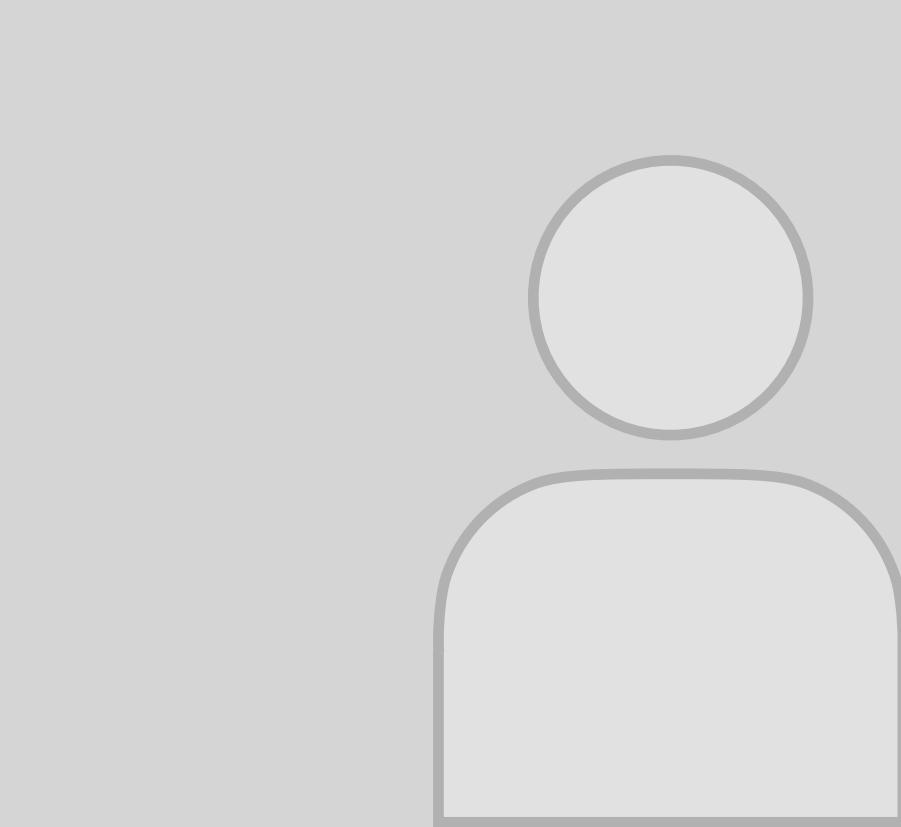
**Programming New HW
& Verifying Optimizations**

[PLDI 2022 + POPL 2022]

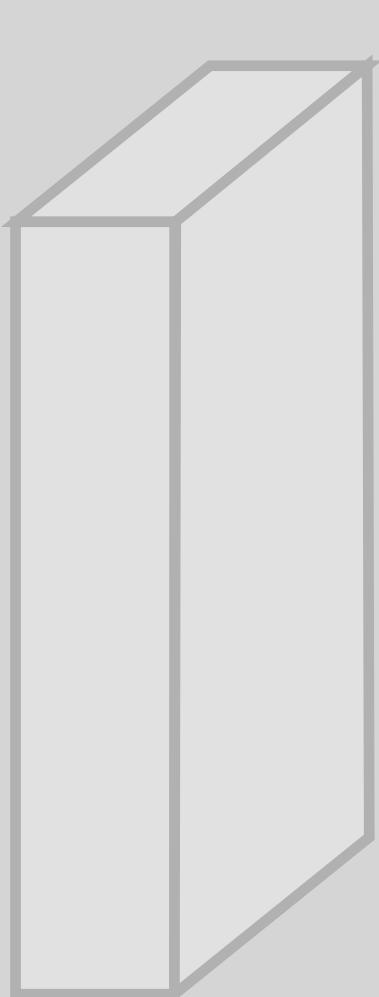
Intel MKL
CoreML
BLAS
cuDNN

↔

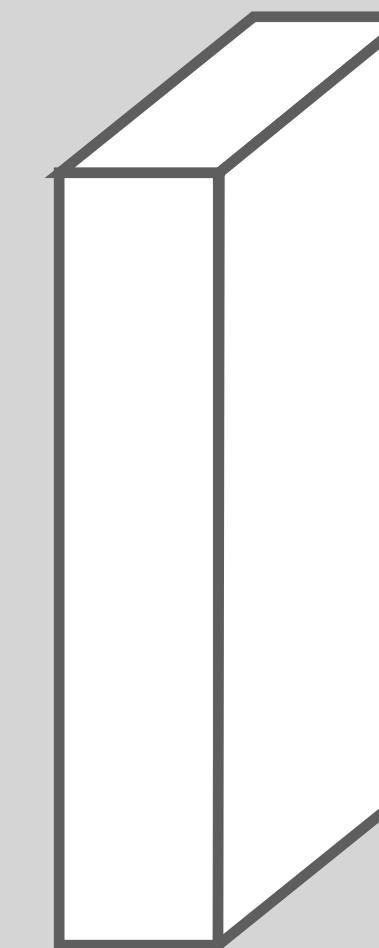
Neural Engine
GraphCore IPU
NVIDIA GPU
Google TPU



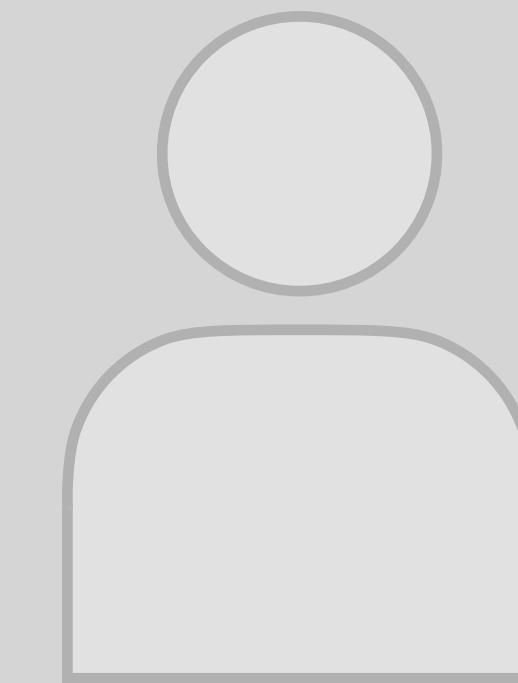
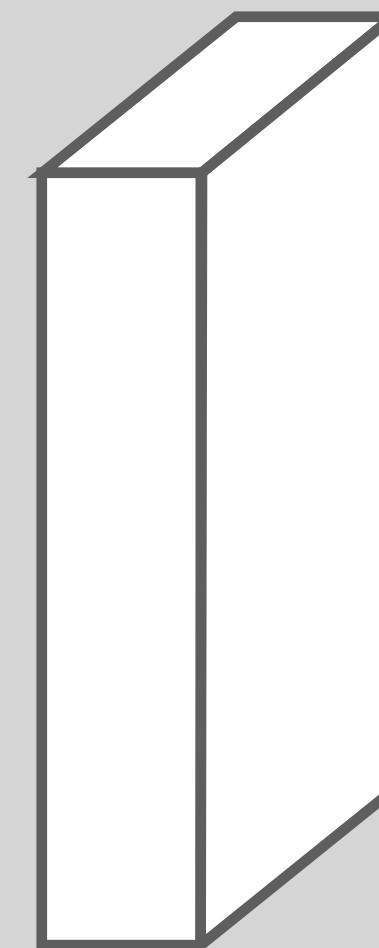
**Machine
Learning
Programmer**



**Framework/
Compiler
Developer**

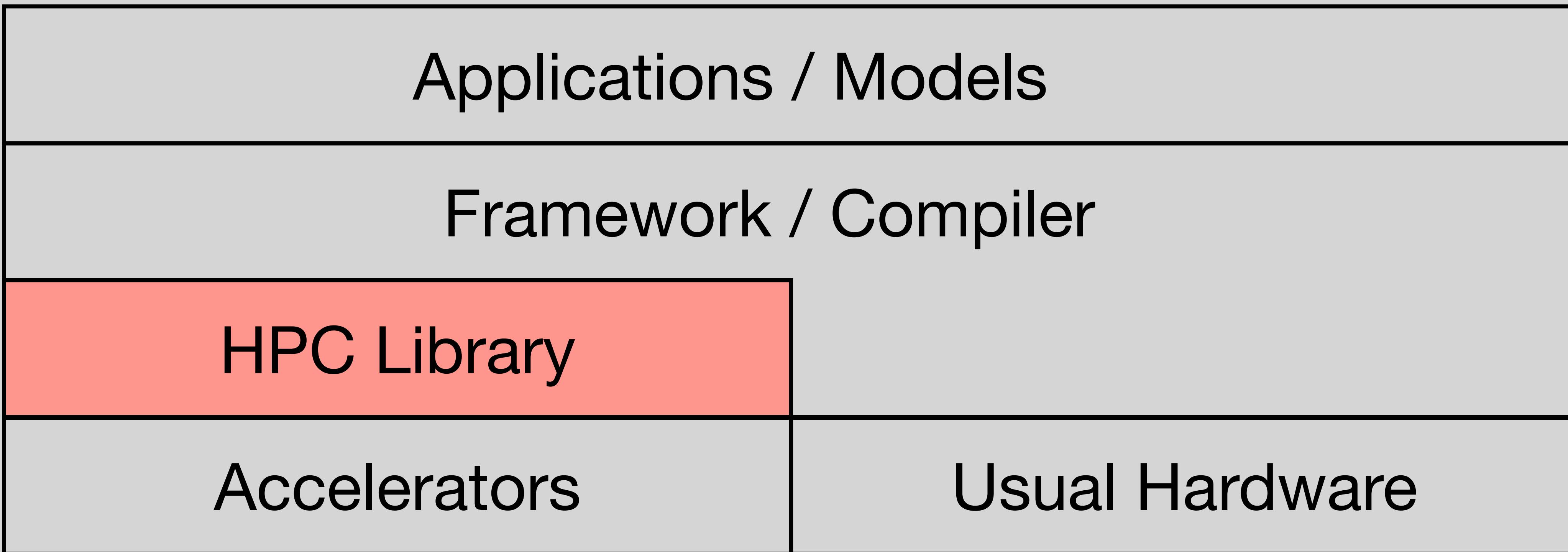


**Performance
Engineers**



**Computer
Architects**

The ML Stack



The ML-Stack Stovepipe

95%+ of computation

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

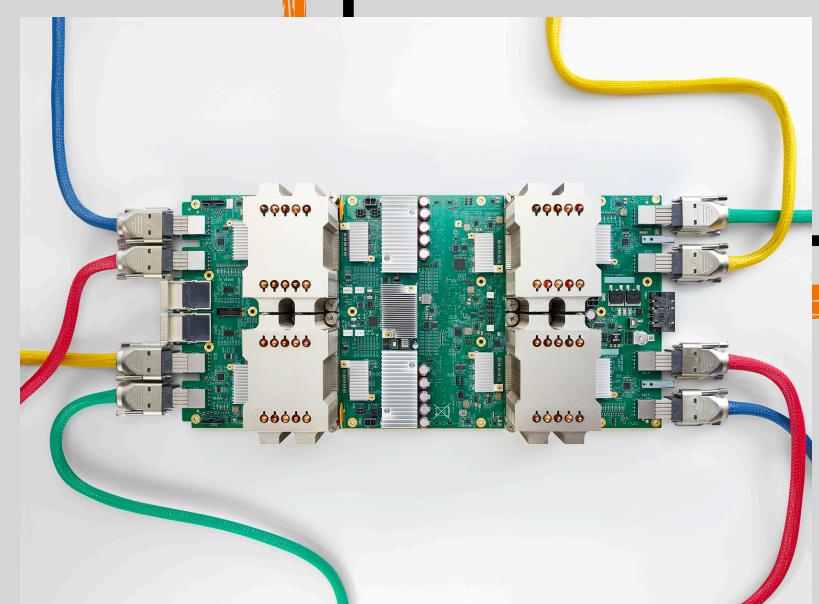
95%+ of computation

Applications / Models

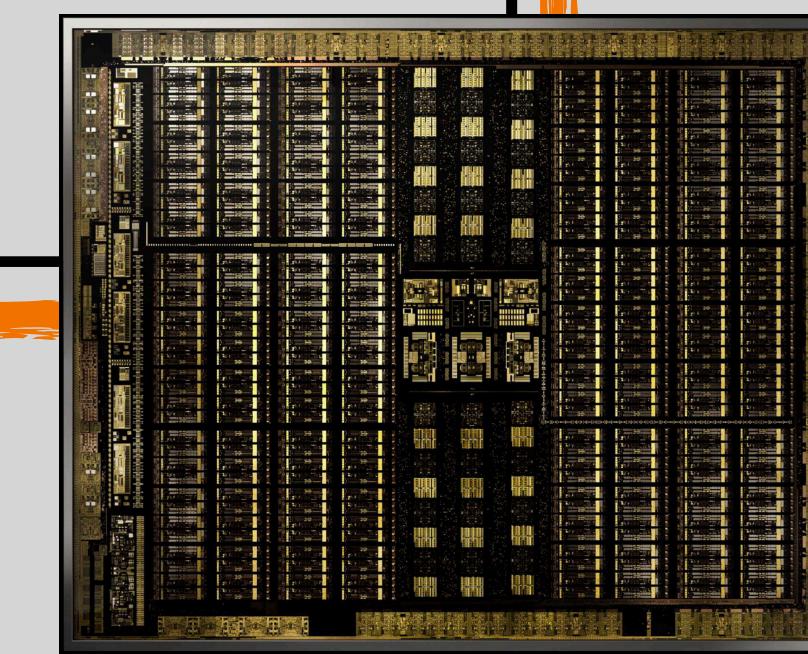
Framework / Compiler

HPC Library

Accelerators

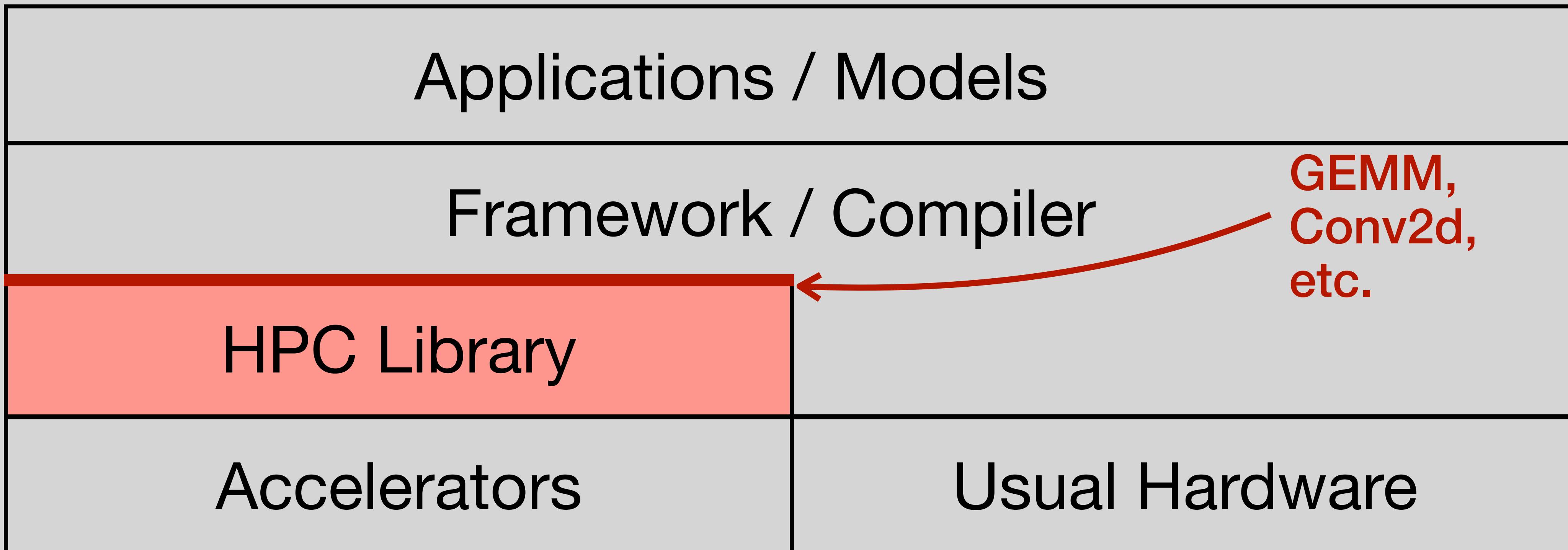


Usual Hardware



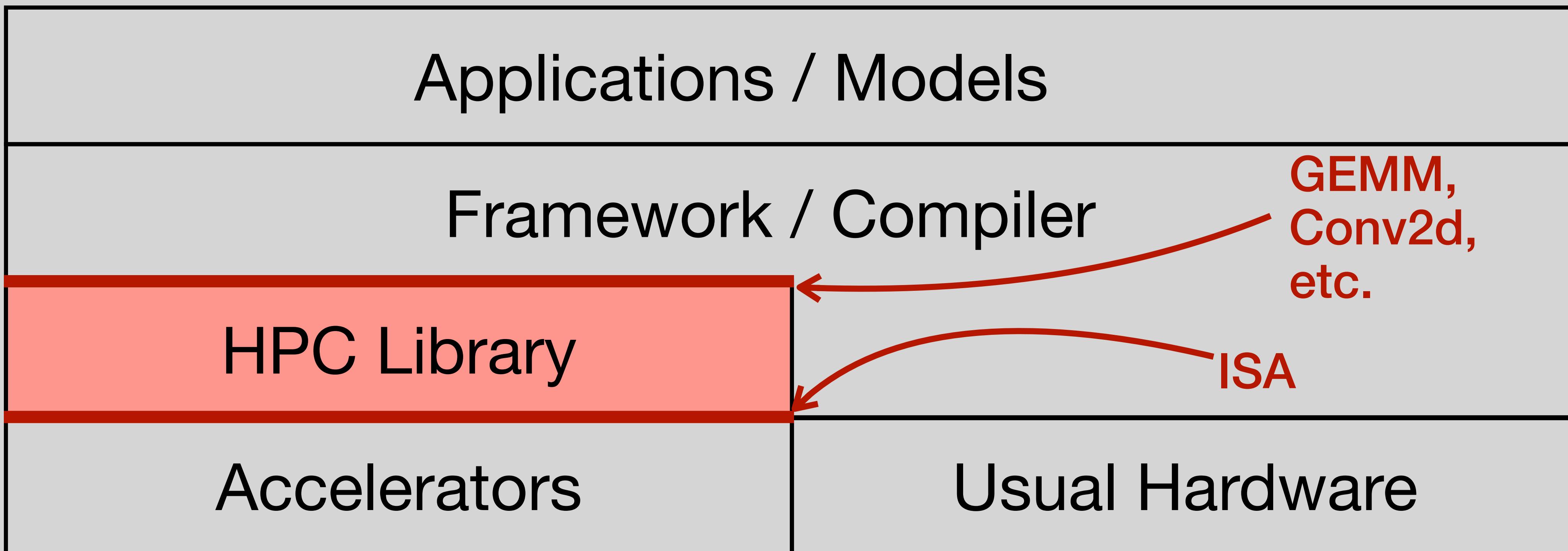
The ML-Stack Stovepipe

Consequence 1: Portability Interface



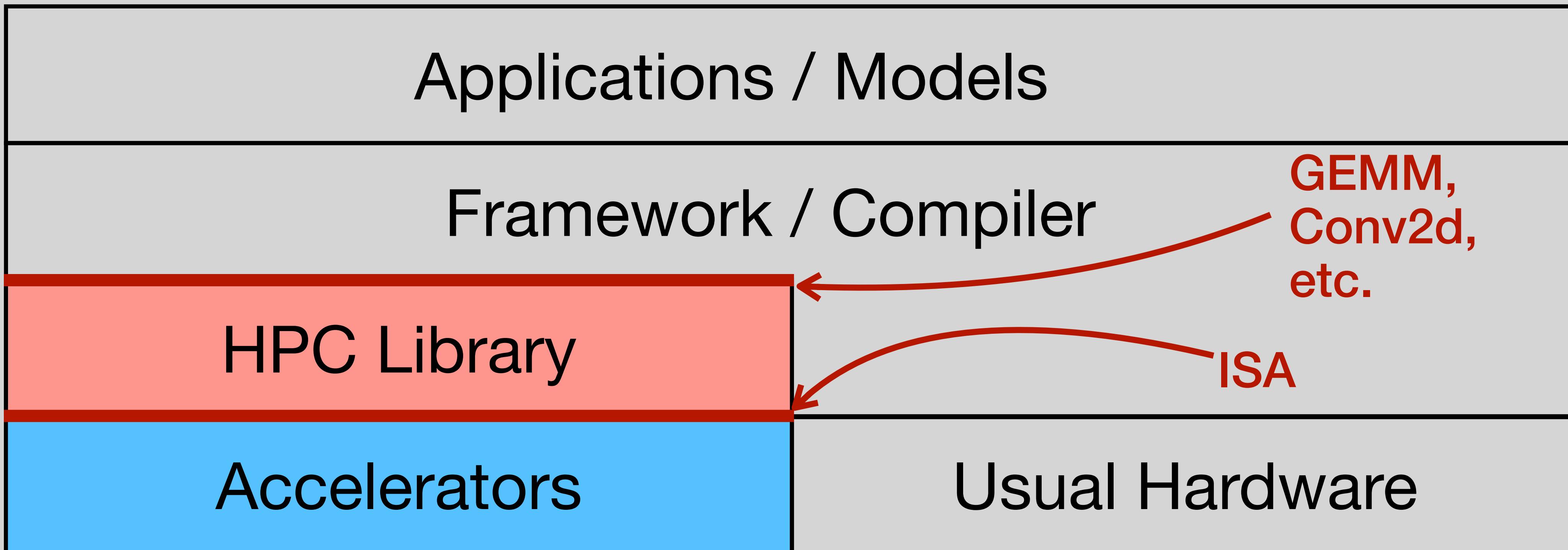
The ML-Stack Stovepipe

Consequence 1: Portability Interface

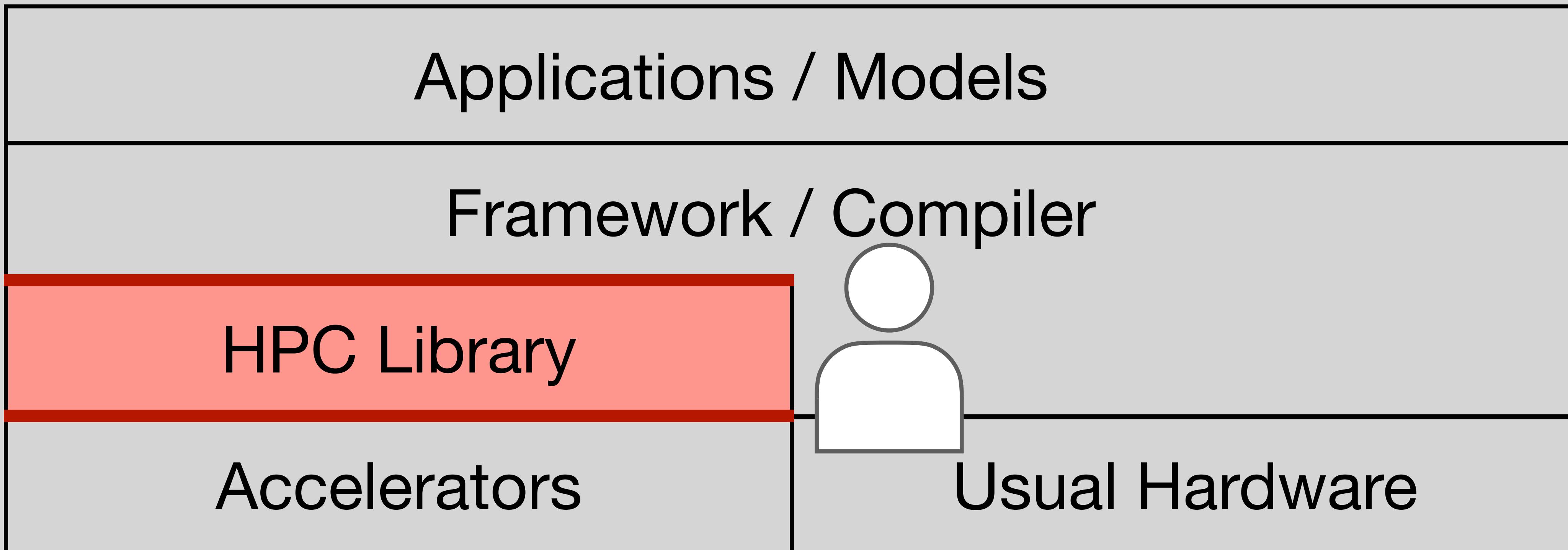


The ML-Stack Stovepipe

Consequence 2: HPC Libs are on “Critical Path”



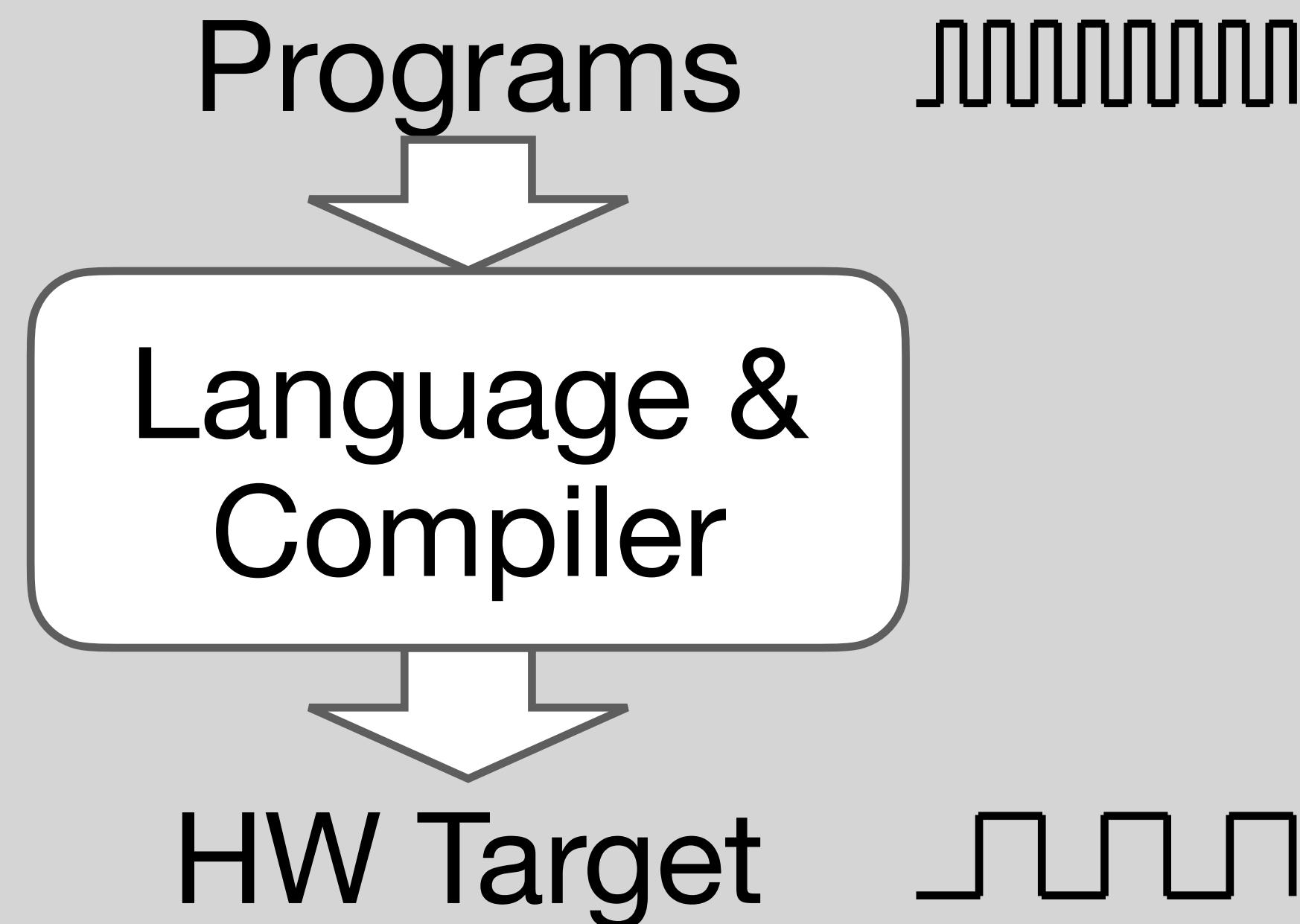
The ML-Stack Stovepipe



What's Special About Writing HPC Libraries?

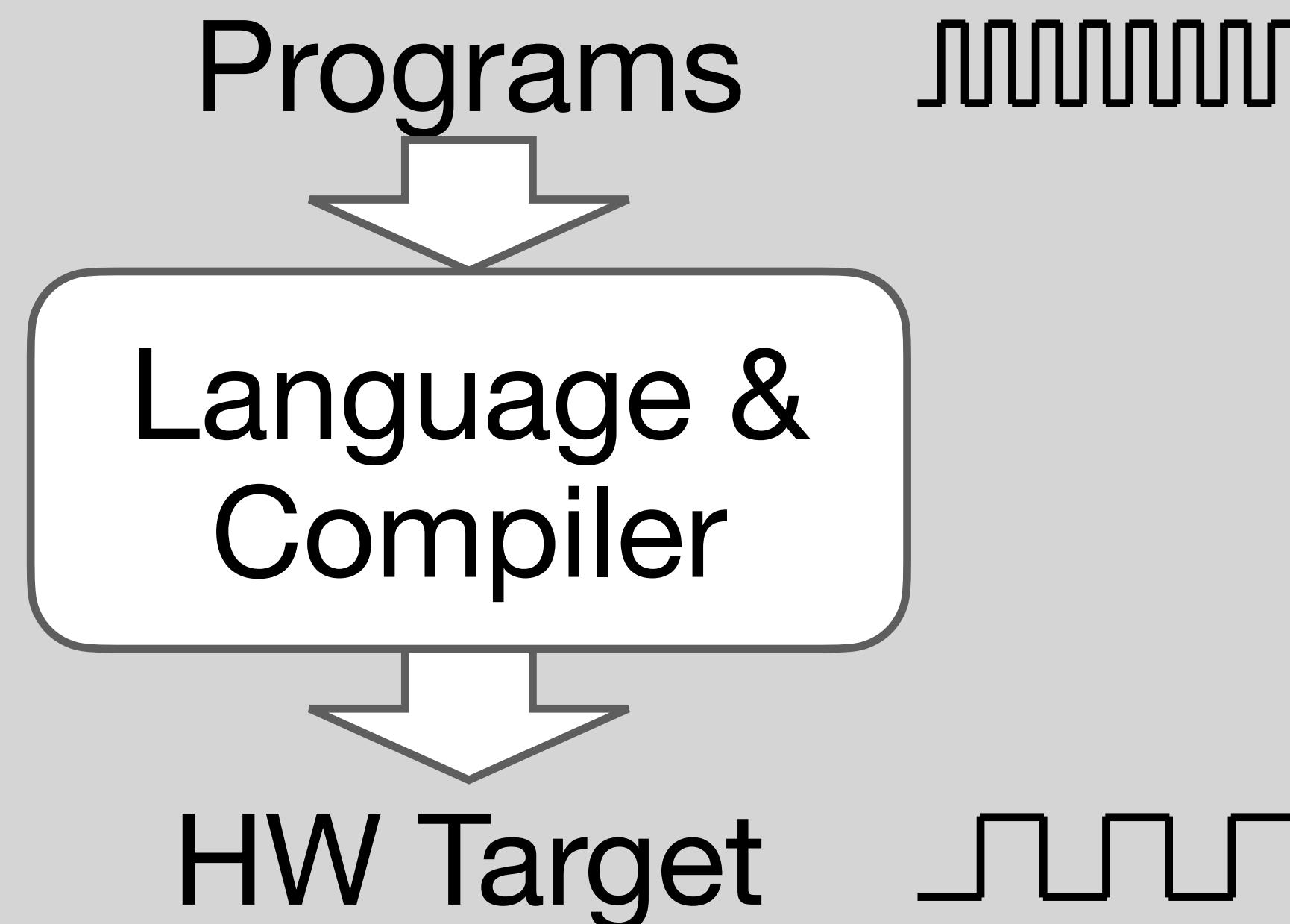
What's Special About Writing HPC Libraries?

Usual Programs

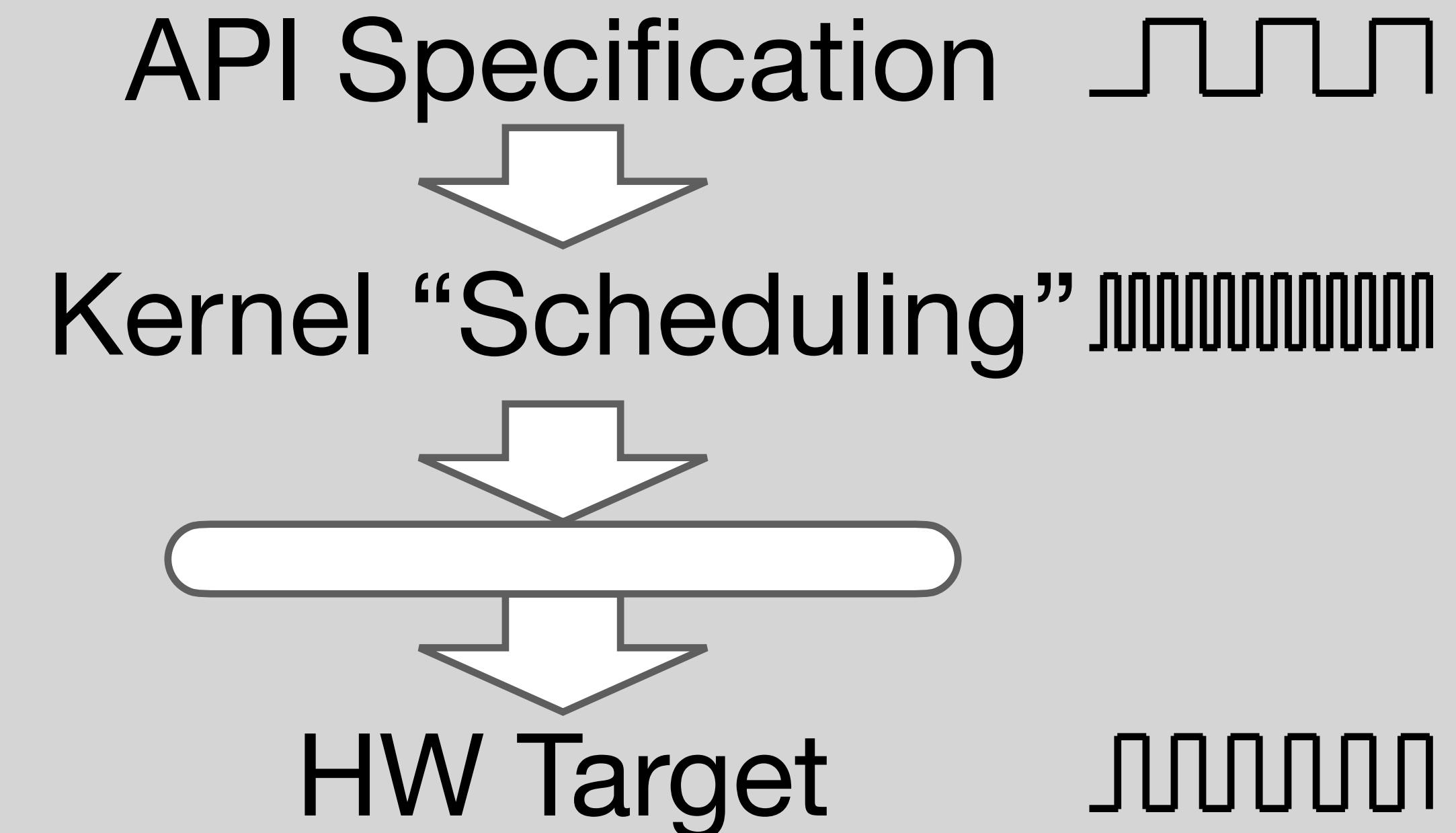


What's Special About Writing HPC Libraries?

Usual Programs

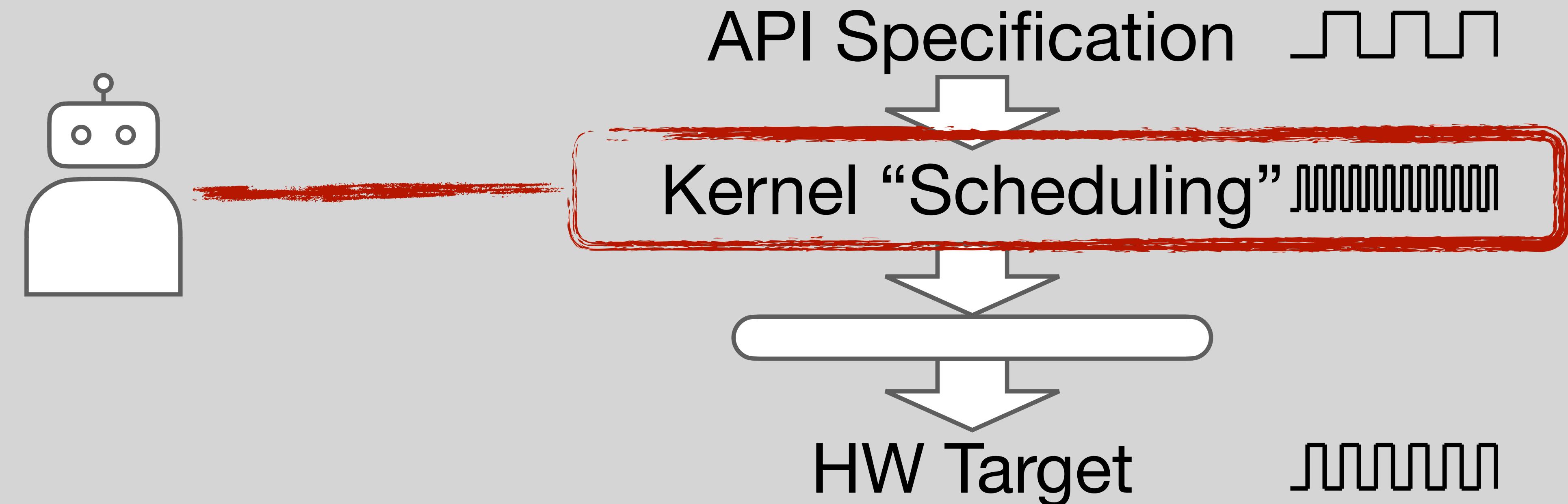


HPC Libraries

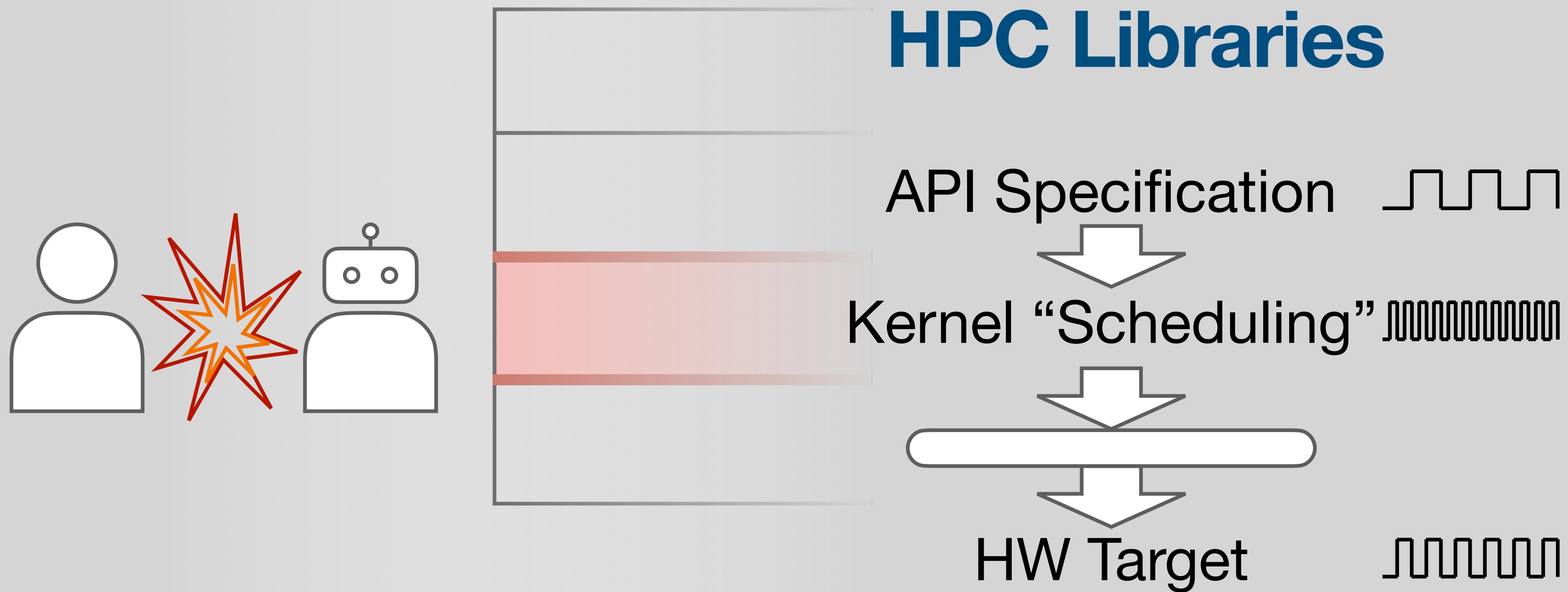


The Trouble With Automation

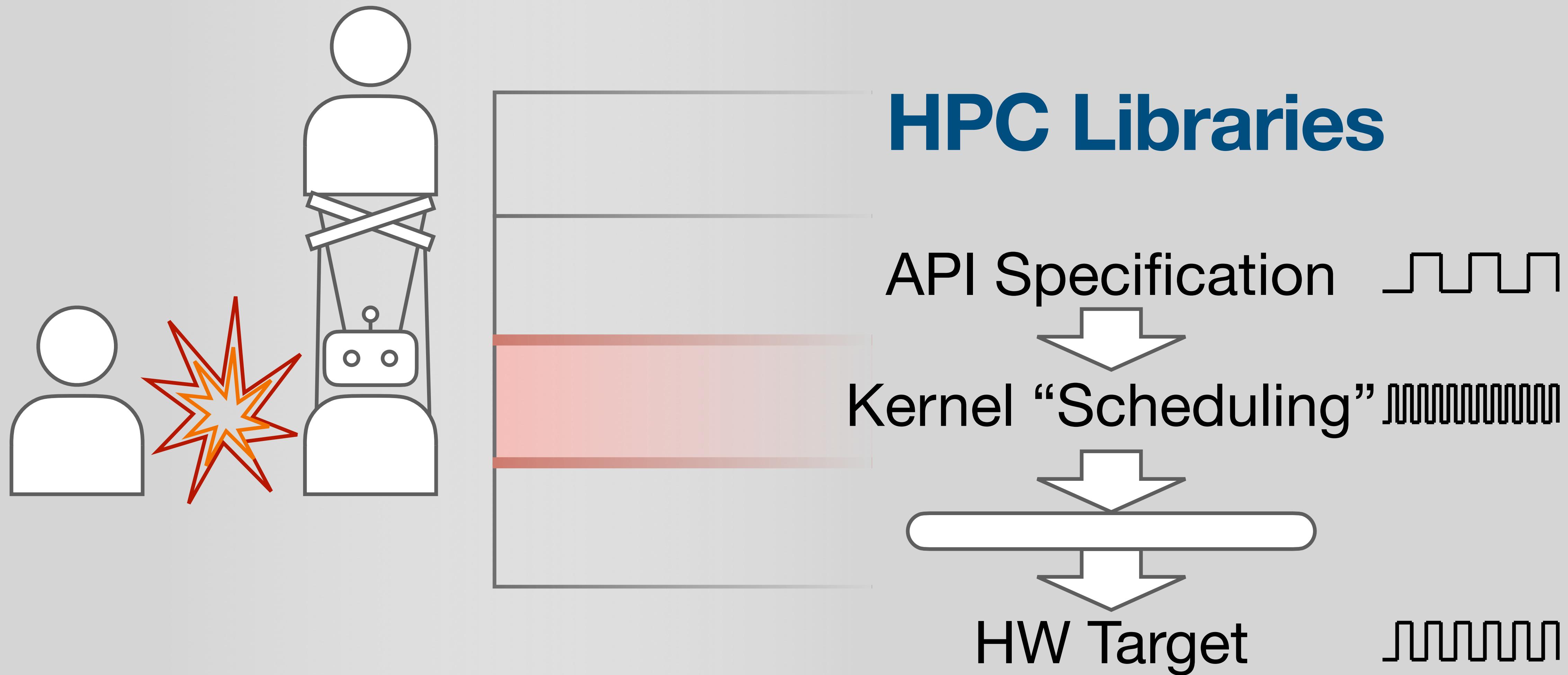
HPC Libraries



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]
```

Naive Matrix Multiply

88.1%
of peak
throughput!

Matrix Multiply

Optimized for AVX-512

**0.94% of peak
throughput**

```

@proc
def sgemm_kernel_avx512_6x4(K: size, A: [f32][6, K] @ DRAM,
                             B: [f32][K, 64] @ DRAM, C: [f32][6, 64] @ DRAM):
    assert K >= 1
    assert stride(A, 1) == 1
    assert stride(B, 1) == 1
    assert stride(C, 1) == 1
    C_reg: R[6, 4, 16] @ AVX512
    for i in par(0, 6):
        for jo in par(0, 4):
            mm512_loadu_ps(C_reg[i, jo, 0:16], C[i, 16 * jo:16 * jo + 16])
    for k in par(0, K):
        for i in par(0, 6):
            A_vec: R[16] @ AVX512
            mm512_set1_ps(A_vec, A[i, k:k + 1])
        for jo in par(0, 4):
            B_vec: R[16] @ AVX512
            mm512_loadu_ps(B_vec[0:16], B[k, 16 * jo:16 * jo + 16])
            mm512_fmadd_ps(A_vec, B_vec, C_reg[i, jo, 0:16])
    for i in par(0, 6):
        for jo in par(0, 4):
            mm512_storeu_ps(C[i, 16 * jo:16 * jo + 16], C_reg[i, jo, 0:16])
@proc
def bottom_panel_kernel_scheduled(M: size, K: size, A: [f32][M, K] @ DRAM,
                                   B: [f32][K, 64] @ DRAM,
                                   C: [f32][M, 64] @ DRAM):
    assert M >= 1
    assert K >= 1
    assert stride(A, 1) == 1
    assert stride(B, 1) == 1
    assert stride(C, 1) == 1
    assert M < 6
    if M == 1:
        sgemm_kernel_avx512_1x4(K, A[0:1, 0:K], B[0:K, 0:64], C[0:1, 0:64])
    else:
        if M == 2:
            sgemm_kernel_avx512_2x4(K, A[0:2, 0:K], B[0:K, 0:64], C[0:2, 0:64])
        else:
            if M == 3:
                sgemm_kernel_avx512_3x4(K, A[0:3, 0:K], B[0:K, 0:64], C[0:3, 0:64])
            else:
                if M == 4:
                    sgemm_kernel_avx512_4x4(K, A[0:4, 0:K], B[0:K, 0:64], C[0:4, 0:64])
                else:
                    if M == 5:
                        sgemm_kernel_avx512_5x4(K, A[0:5, 0:K], B[0:K, 0:64], C[0:5, 0:64])
                    else:
                        for k in par(0, K):
                            for i in par(0, M):
                                for j in par(0, 64):
                                    C[i, j] += A[i, k] * B[k, j]
@proc
def right_panel_kernel_scheduled(N: size, K: size, A: [f32][6, K] @ DRAM,
                                   B: [f32][K, N] @ DRAM, C: [f32][6, N] @ DRAM):
    assert N >= 1
    assert K >= 1
    assert stride(A, 1) == 1
    assert stride(B, 1) == 1
    assert stride(C, 1) == 1
    assert N < 16 < 4
    if N / 16 == 0:
        C_reg: R[6, 1, 16] @ AVX512
        C_reg_1: R[6, 16] @ AVX512
        for i in par(0, 6):
            mm512_maskz_loadu_ps(N, C_reg_1[i, 0:16], C[i, 0:N])
        for k in par(0, K):
            for i in par(0, 6):
                A_reg: R[16] @ AVX512
                mm512_set1_ps(A_reg, A[i, k:k + 1])
                B_reg: R[16] @ AVX512
                mm512_loadu_ps(B_reg[0:16], B[k, 0:N])
                mm512_mask_z_fmadd_ps(N, A_reg, B_reg, C_reg_1[i, 0:16])
        for i in par(0, 6):
            mm512_mask_storeu_ps(N, C[i, 0:N], C_reg_1[i, 0:16])
    else:
        if N / 16 == 1:
            C_reg: R[6, 2, 16] @ AVX512
            C_reg_1: R[6, 16] @ AVX512
            for i in par(0, 6):
                for jo in par(0, 1):
                    mm512_loadu_ps(C_reg[i, jo, 0:16], C[i, 16 * jo:16 * jo + 16])
                    mm512_maskz_loadu_ps(N, C_reg_1[i, 0:16], C[i, 16:N])
            for k in par(0, K):
                for i in par(0, 6):
                    for jo in par(0, 1):
                        A_reg: R[16] @ AVX512
                        mm512_set1_ps(A_reg, A[i, k:k + 1])
                        B_reg: R[16] @ AVX512
                        mm512_loadu_ps(B_reg[0:16], B[k, 0:N])
                        mm512_mask_z_fmadd_ps(N, A_reg, B_reg, C_reg_1[i, 0:16])
            for i in par(0, 6):
                mm512_mask_storeu_ps(N, C[i, 0:N], C_reg_1[i, 0:16])
        else:
            C_reg: R[6, N / 16 + 1, 16] @ AVX512
            C_reg_1: R[6, 16] @ AVX512
            for i in par(0, 6):
                for jo in par(0, N / 16):
                    mm512_loadu_ps(C_reg[i, jo, 0:16], C[i, 16 * jo:16 * jo + 16])
                    mm512_maskz_loadu_ps(N, C_reg_1[i, 0:16], C[i, 16 * (N / 16):N])
            for k in par(0, K):
                for i in par(0, 6):
                    for jo in par(0, N / 16):
                        A_reg: R[16] @ AVX512
                        mm512_set1_ps(A_reg, A[i, k:k + 1])
                        B_reg: R[16] @ AVX512
                        mm512_loadu_ps(B_reg[0:16], B[k, 0:N])
                        mm512_mask_z_fmadd_ps(N, A_reg, B_reg, C_reg_1[i, 0:16])
            for i in par(0, 6):
                mm512_mask_storeu_ps(N, C[i, 0:N], C_reg_1[i, 0:16])
@proc
def sgemm_above_kernel(M: size, N: size, K: size, A: [f32][M, K] @ DRAM,
                       B: [f32][K, N] @ DRAM, C: [f32][M, N] @ DRAM):
    assert M >= 1
    assert N >= 1
    assert K >= 1
    assert stride(A, 1) == 1
    assert stride(B, 1) == 1
    assert stride(C, 1) == 1
    for io in par(0, M / 64):
        for i1 in par(0, N / 64):
            B_reg: R[16] @ AVX512
            mm512_loadu_ps(B_reg[0:16], B[k, 16 * jo:16 * jo + 16])
            mm512_fmadd_ps(A[i, k:k + 1], B[k, 16 * jo:16 * jo + 16], C[i, 16:N])
            mm512_loadu_ps(C[i, 16 * jo:16 * jo + 16], C_reg[i, jo, 0:16])
    for i in par(0, 6):
        for jo in par(0, 1):
            mm512_storesu_ps(C[i, 16 * jo:16 * jo + 16], C_reg[i, jo, 0:16])
    else:
        if N / 16 == 2:
            C_reg: R[6, 3, 16] @ AVX512
            C_reg_1: R[6, 16] @ AVX512
            for i in par(0, 6):
                for jo in par(0, 2):
                    mm512_loadu_ps(C_reg[i, jo, 0:16], C[i, 16 * jo:16 * jo + 16])
                    mm512_fmadd_ps(A[i, k:k + 1], B[i, 16 * jo:16 * jo + 16], C[i, 16:N])
                    mm512_loadu_ps(C[i, 16 * jo:16 * jo + 16], C_reg_1[i, 0:16])
            for i in par(0, 6):
                for jo in par(0, N / 16):
                    mm512_storesu_ps(C[i, 16 * jo:16 * jo + 16], C_reg[i, jo, 0:16])
        else:
            if N / 16 == 3:
                C_reg: R[6, 4, 16] @ AVX512
                C_reg_1: R[6, 16] @ AVX512
                for i in par(0, 6):
                    for jo in par(0, 3):
                        mm512_loadu_ps(C_reg[i, jo, 0:16], C[i, 16 * jo:16 * jo + 16])
                        mm512_fmadd_ps(A[i, k:k + 1], B[i, 16 * jo:16 * jo + 16], C[i, 16:N])
                        mm512_loadu_ps(C[i, 16 * jo:16 * jo + 16], C_reg_1[i, 0:16])
                for i in par(0, 6):
                    mm512_storesu_ps(C[i, 16 * jo:16 * jo + 16], C_reg[i, jo, 0:16])
            else:
                if N / 16 == 4:
                    C_reg: R[6, 5, 16] @ AVX512
                    C_reg_1: R[6, 16] @ AVX512
                    for i in par(0, 6):
                        for jo in par(0, 4):
                            mm512_loadu_ps(C_reg[i, jo, 0:16], C[i, 16 * jo:16 * jo + 16])
                            mm512_fmadd_ps(A[i, k:k + 1], B[i, 16 * jo:16 * jo + 16], C[i, 16:N])
                            mm512_loadu_ps(C[i, 16 * jo:16 * jo + 16], C_reg_1[i, 0:16])
                    for i in par(0, 6):
                        mm512_storesu_ps(C[i, 16 * jo:16 * jo + 16], C_reg[i, jo, 0:16])
                else:
                    if N / 16 == 5:
                        C_reg: R[6, 6, 16] @ AVX512
                        C_reg_1: R[6, 16] @ AVX512
                        for i in par(0, 6):
                            for jo in par(0, 5):
                                mm512_loadu_ps(C_reg[i, jo, 0:16], C[i, 16 * jo:16 * jo + 16])
                                mm512_fmadd_ps(A[i, k:k + 1], B[i, 16 * jo:16 * jo + 16], C[i, 16:N])
                                mm512_loadu_ps(C[i, 16 * jo:16 * jo + 16], C_reg_1[i, 0:16])
                        for i in par(0, 6):
                            mm512_storesu_ps(C[i, 16 * jo:16 * jo + 16], C_reg[i, jo, 0:16])
                    else:
                        if N / 16 == 6:
                            C_reg: R[6, 7, 16] @ AVX512
                            C_reg_1: R[6, 16] @ AVX512
                            for i in par(0, 6):
                                for jo in par(0, 6):
                                    mm512_loadu_ps(C_reg[i, jo, 0:16], C[i, 16 * jo:16 * jo + 16])
                                    mm512_fmadd_ps(A[i, k:k + 1], B[i, 16 * jo:16 * jo + 16], C[i, 16:N])
                                    mm512_loadu_ps(C[i, 16 * jo:16 * jo + 16], C_reg_1[i, 0:16])
                            for i in par(0, 6):
                                mm512_storesu_ps(C[i, 16 * jo:16 * jo + 16], C_reg[i, jo, 0:16])
@proc
def sgemm_sys_atl(M: size, N: size, K: size, A: f32[M, K] @ DRAM,
                  B: f32[K, N] @ DRAM, C: f32[M, N] @ DRAM):
    assert M >= 1
    assert N >= 1
    assert K >= 1
    assert stride(A, 1) == 1
    assert stride(B, 1) == 1
    assert stride(C, 1) == 1
    A1_cache: f32[1264, 512] @ DRAM_STATIC
    B1_cache: f32[512, 64] @ DRAM_STATIC
    for ko in par(0, K / 512):
        for io in par(0, M / 264):
            for i1 in par(0, N / 64):
                B2_cache: f32[512, 64] @ DRAM_STATIC
                for i0 in par(0, K - 512 * (K / 512)):
                    for i1 in par(0, N / 64):
                        B2_cache[i0, i1] = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
                sgemm_above_kernel(
                    M % 264, N % 64, 512, A[264 * (M / 264):M, 512 * ko:512 * ko + 512],
                    B2_cache[0:512, 0:N - 64 * (N / 64)], C[264 * (M / 264):M, 64 * (N / 64):N])
                if M % 264 > 0:
                    if N % 64 > 0:
                        for ko in par(0, K / 512):
                            B4_cache: f32[512, 64] @ DRAM_STATIC
                            for i0 in par(0, N / 64):
                                for i1 in par(0, N / 64):
                                    B4_cache[i0, i1] = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
                            sgemm_above_kernel(
                                M % 264, N % 64, 512, A[264 * (M / 264):M, 512 * ko:512 * ko + 512],
                                B4_cache[0:512, 0:N - 64 * (N / 64)], C[264 * (M / 264):M, 64 * (N / 64):N])
                if K % 512 > 0:
                    if M % 264 > 0:
                        for ko in par(0, K / 512):
                            B5_cache: f32[512, 64] @ DRAM_STATIC
                            for i0 in par(0, N / 64):
                                for i1 in par(0, N / 64):
                                    B5_cache[i0, i1] = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
                            sgemm_above_kernel(
                                264, 64, K % 512, A[264 * io:264 * io + 264, 512 * (K / 512):K],
                                B5_cache[0:K - 512 * (K / 512), 0:N - 64 * (N / 64)], C[264 * io:264 * io + 264, 64 * jo:64 * jo + 64])
                    if N % 64 > 0:
                        for ko in par(0, K / 512):
                            for i0 in par(0, M / 6):
                                right_panel_kernel_scheduled(
                                    M % 64, K, A[6 * io:6 * io + 6, 0:K], B[0:K, 64 * (N / 64):N], C[6 * io:6 * io + 6, 64 * (N / 64):N])
                    if M % 6 > 0:
                        for ko in par(0, N / 64):
                            for i0 in par(0, N / 64):
                                bottom_panel_kernel_scheduled(
                                    M % 6, K, A[6 * (M / 6):M, 0:K], B[0:K, 64 * jo:64 * jo + 64], C[6 * (M / 6):M, 64 * jo:64 * jo + 64])
                    if N % 64 > 0:
                        for ko in par(0, K / 512):
                            for i0 in par(0, M % 6):
                                for i1 in par(0, N % 64):
                                    for ji in par(0, N / 64):
                                        C[ii + M / 6 * 6, ji + N / 64 * 64] += A[ii + M / 6 * 6, k] * B[k, ji + N / 64 * 64]
                    if K % 512 > 0:
                        if M % 264 > 0:
                            for ko in par(0, N / 64):
                                B7_cache: f32[512, 64] @ DRAM_STATIC
                                for i0 in par(0, K - 512 * (K / 512)):
                                    for i1 in par(0, N / 64):
                                        B7_cache[i0, i1] = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
                            sgemm_above_kernel(
                                M % 264, N % 64, K % 512, A[264 * io:264 * io + 264, 512 * (K / 512):K],
                                B6_cache[0:K - 512 * (K / 512), 0:N - 64 * (N / 64)], C[264 * io:264 * io + 264, 64 * (N / 64):N])
                        if N % 64 > 0:
                            for ko in par(0, N / 64):
                                for i0 in par(0, K / 512):
                                    for i1 in par(0, M / 264):
                                        for i2 in par(0, N / 64):
                                            A1_cacheli0, i1 = A[264 * io + i0, 512 * ko + i1]
                                            for jo in par(0, N / 64):
                                                for i0 in par(0, K / 512):
                                                    for i1 in par(0, M / 264):
                                                        for i2 in par(0, N / 64):
                                                            B1_cacheli0, i1 = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
                                                            sgemm_above_kernel(
                                                                264, 64, 512, A1_cacheli0, B1_cacheli0, B1_cache[0:512, 0:N - 64 * (N / 64)])
                                            if M % 264 > 0:
                                                if N % 64 > 0:
                                                    for ko in par(0, K / 512):
                                                        B8_cache: f32[512, 64] @ DRAM_STATIC
                                                        for i0 in par(0, N / 64):
                                                            for i1 in par(0, N / 64):
                                                                B8_cache[i0, i1] = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
                                                        sgemm_above_kernel(
                                                            M % 264, N % 64, K % 512, A[264 * (M / 264):M, 512 * ko:512 * ko + 512],
                                                            B8_cache[0:K - 512 * (K / 512), 0:N - 64 * (N / 64)], C[264 * (M / 264):M, 64 * jo:64 * jo + 64])

```

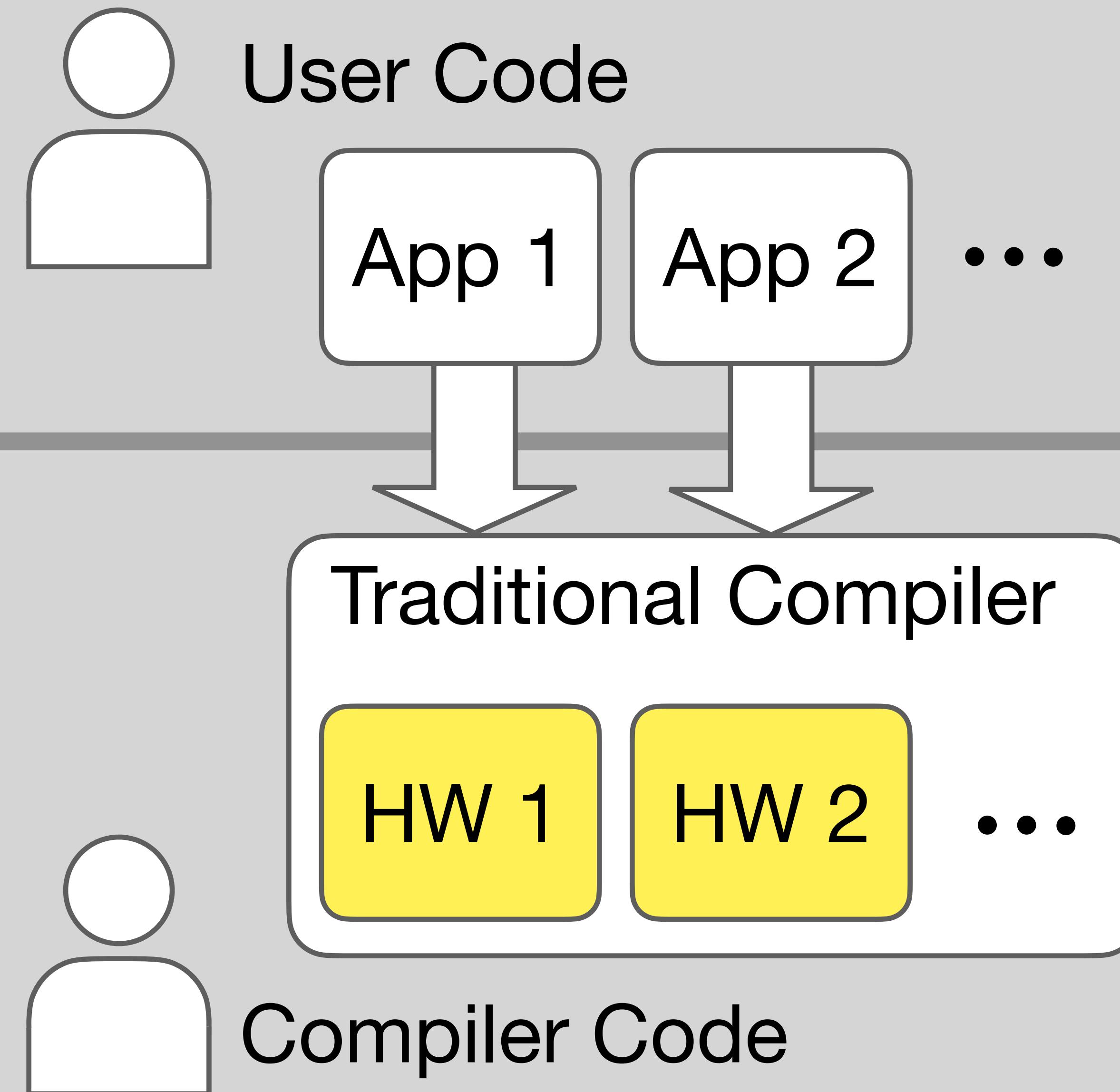
Exocompilation

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

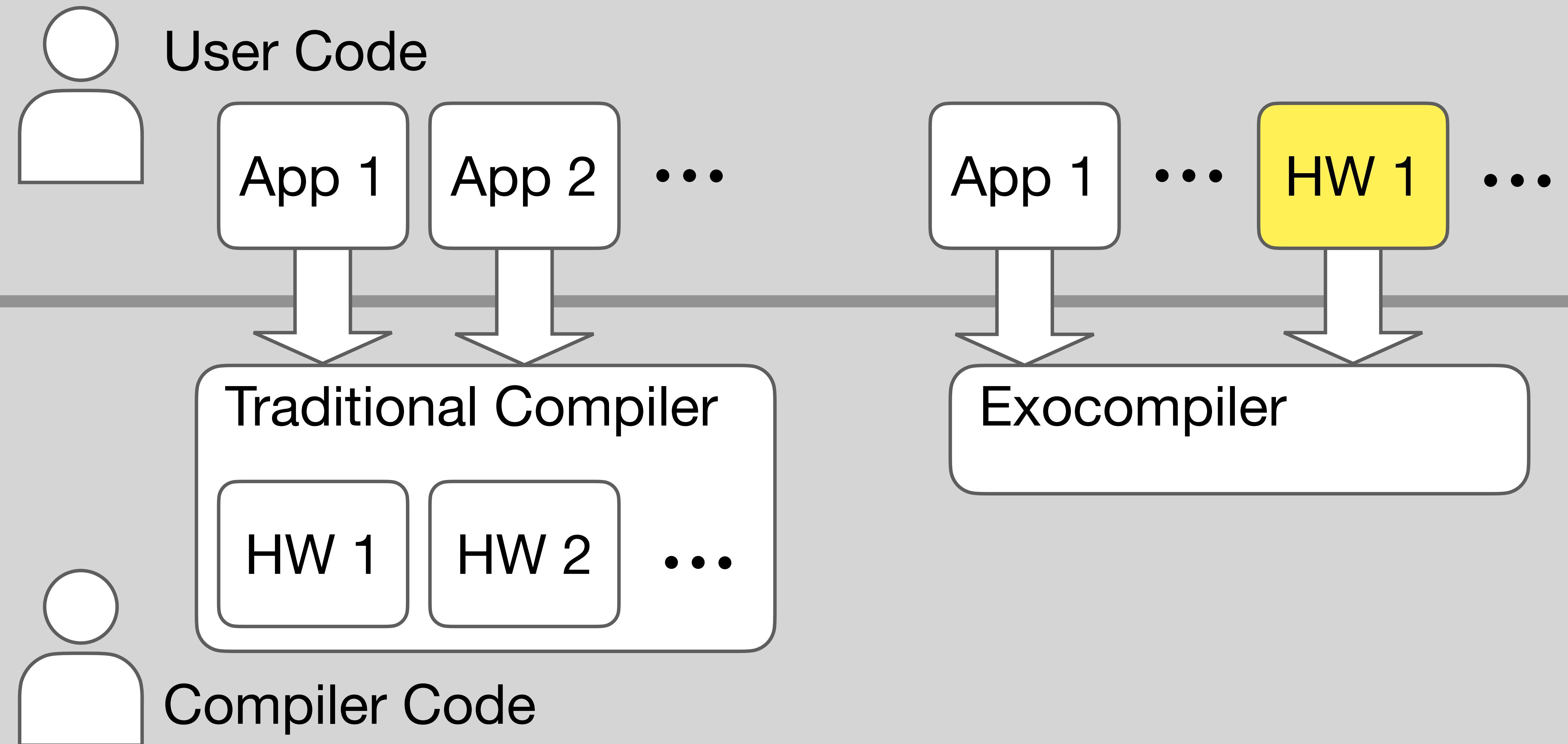
HW Targets as
Libraries

User Scheduling

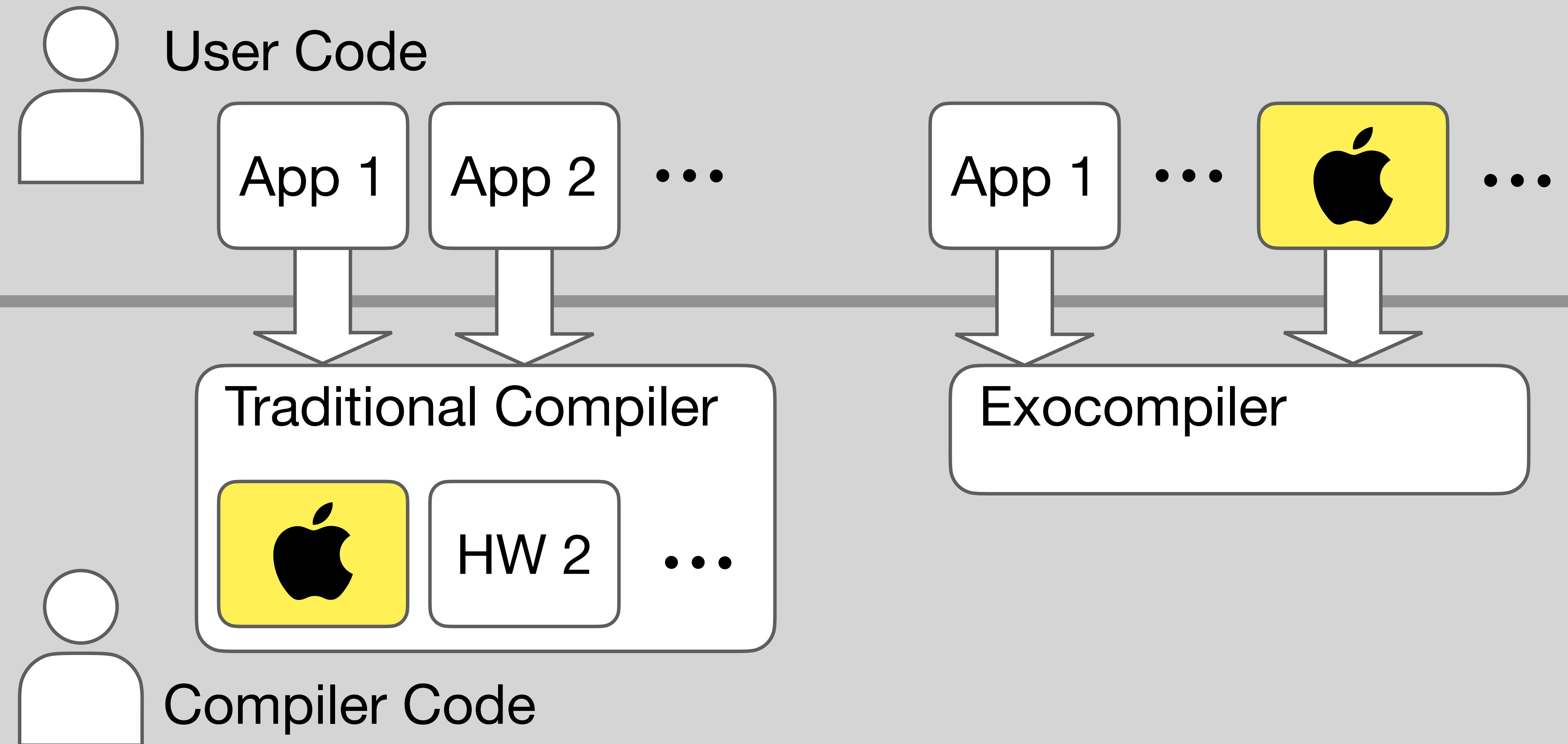
HW Backends as User-Level Libraries



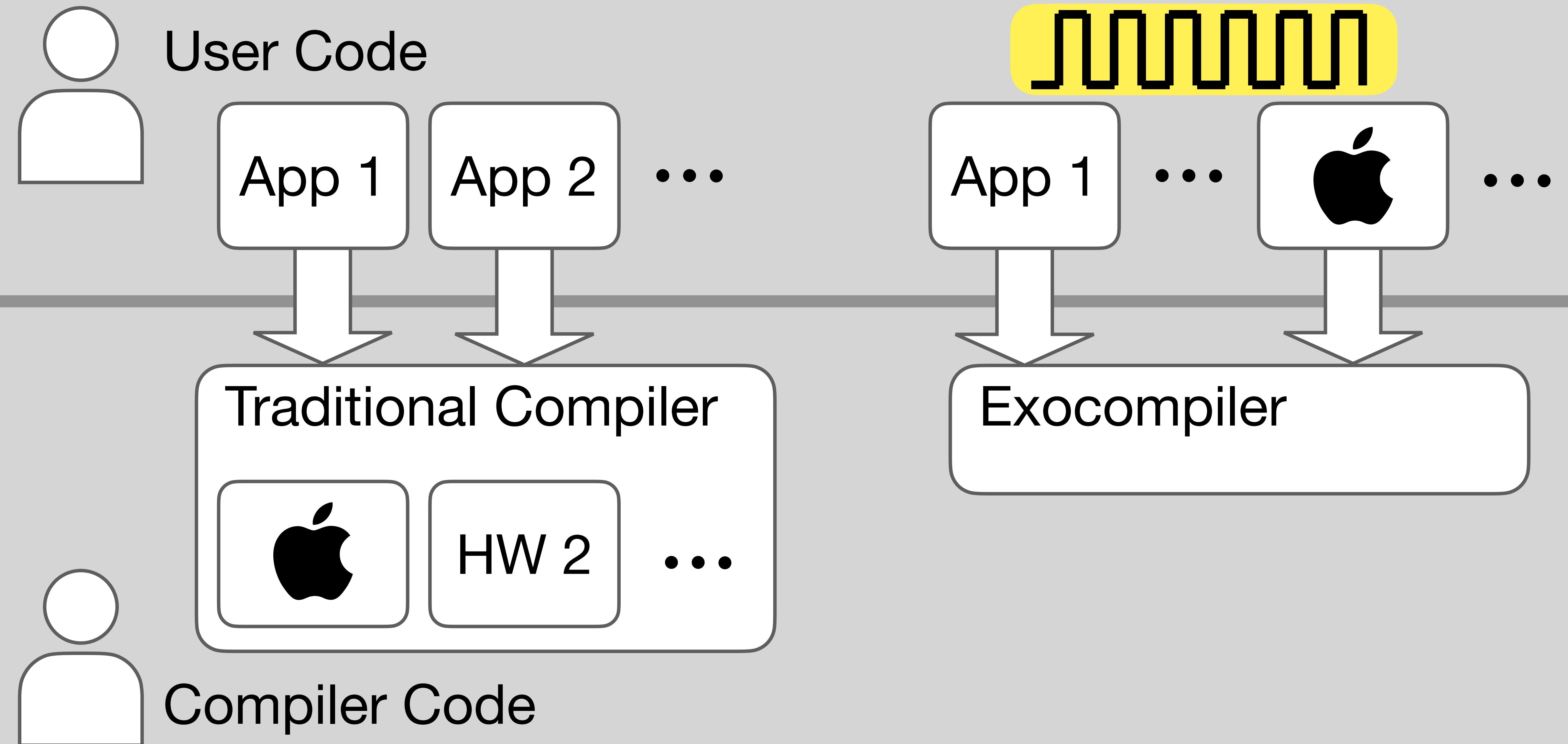
HW Backends as User-Level Libraries



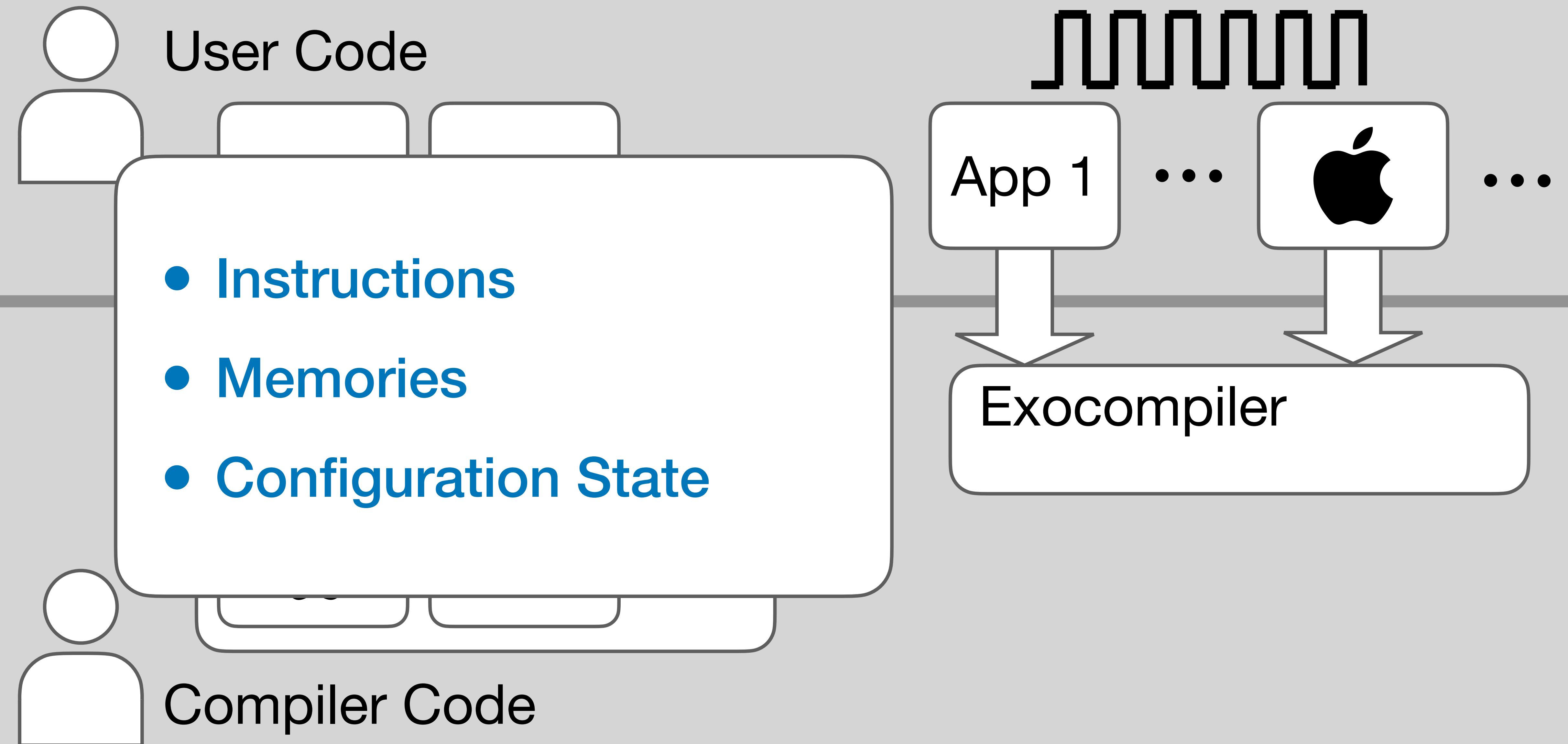
HW Backends as User-Level Libraries



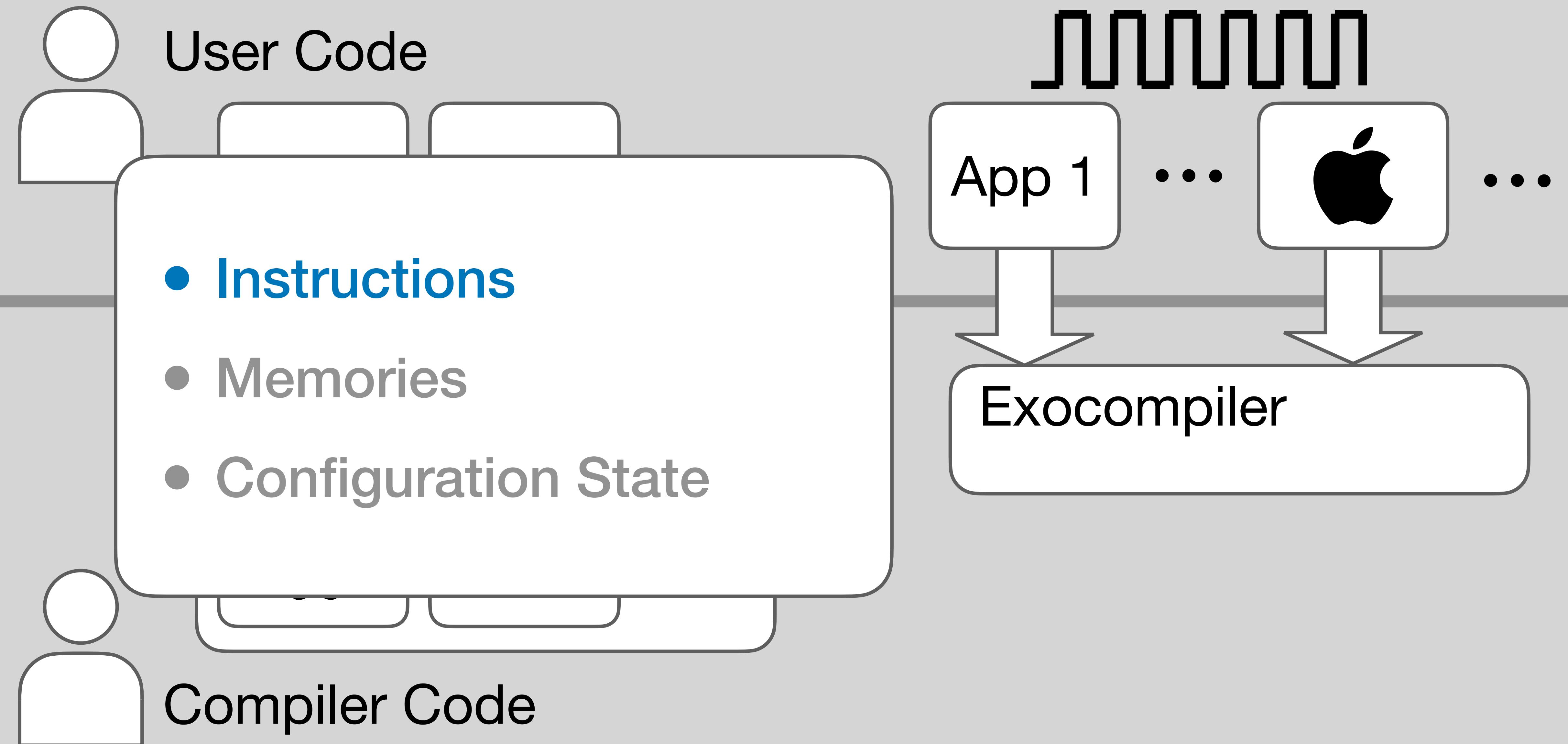
HW Backends as User-Level Libraries



HW Backends as User-Level Libraries



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(...)  
    ...
```

**Generate
Code**

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

```
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]
```

```

emm_sys_atl = (
SGEMM
    .rename('sgemm_sys_atl')
# Split all loops
    .split('k', K_L1_BLK, ['ko', 'ki'], tail='cut_and_guard')
    .split('i', M_L1_BLK, ['io', 'ii'], tail='cut_and_guard')
    .split('j', N_L1_BLK, ['jo', 'ji'], tail='cut_and_guard')
# Explode into 8 cases
    .fission_after('for io in _: _', n_lifts=2)
    .fission_after('for jo in _: _', n_lifts=4)
# Case 1:
    .reorder('ki', 'io')
    .reorder('ii', 'jo')
    .reorder('ki', 'jo')
    .replace(SGEMM_WINDOW, 'for ki in _: _ #0')
# Case 2:
    .lift_if('if N % _ > 0: _ #0', n_lifts=4)
    .replace(SGEMM_WINDOW, 'for ki in _: _ #0')
# Case 3:
    .lift_if('if M % _ > 0: _ #0', n_lifts=2)
    .reorder('ki', 'jo')
    .replace(SGEMM_WINDOW, 'for ki in _: _ #0')
# Case 4:
    .lift_if('if M % _ > 0: _ #1', n_lifts=2)
    .lift_if('if N % _ > 0: _ #1', n_lifts=3)
    .replace(SGEMM_WINDOW, 'for ki in _: _ #0')
# Case 5:
    .replace(SGEMM_WINDOW, 'for ki in _: _ #0')
# Case 6:
    .lift_if('if N % _ > 0: _ #2', n_lifts=3)
    .replace(SGEMM_WINDOW, 'for ki in _: _ #0')
# Case 7:
    .lift_if('if M % _ > 0: _ #2')
    .reorder('ki', 'jo')
    .replace(SGEMM_WINDOW, 'for ki in _: _ #0')
# Case 8:
    .lift_if('if M % _ > 0: _ #3')
    .lift_if('if N % _ > 0: _ #3', n_lifts=2)
    .replace(SGEMM_WINDOW, 'for ki in _: _ #0')
## Case 1 memory staging
    .stage_window('A1_cache', 'A[_] #0', DRAM_STATIC)
    .stage_window('B1_cache', 'B[_] #0', DRAM_STATIC)
    .par_to_seq('for ko in _: _ #0')
    .par_to_seq('for io in _: _ #0')
    .par_to_seq('for jo in _: _ #0')
    .lift_alloc('A1_cache: _', n_lifts=3)
    .lift_alloc('B1_cache: _', n_lifts=3)
    .fission_after('for i0 in _: _ #0')
## Case 2 memory staging
    .stage_window('B2_cache', 'B[_] #1', DRAM_STATIC)
    .bound_alloc('B2_cache: _', [None, '64'])
    .lift_alloc('B2_cache: _')
    .fission_after('for i0 in _: _ #2')
## Case 3 memory staging
    .stage_window('B3_cache', 'B[_] #2', DRAM_STATIC)
## Case 4 memory staging
    .stage_window('B4_cache', 'B[_] #3', DRAM_STATIC)
    .bound_alloc('B4_cache: _', [None, '64'])
## Case 5 memory staging
    .stage_window('B5_cache', 'B[_] #4', DRAM_STATIC)
    .bound_alloc('B5_cache: _', ['512', None])
## Case 6 memory staging
    .stage_window('B6_cache', 'B[_] #5', DRAM_STATIC)
    .bound_alloc('B6_cache: _', ['512', '64'])
    # .lift_alloc('B6_cache: _')
    # .fission_after('for i0 in _: _ #6')
## Case 7 memory staging
    .stage_window('B7_cache', 'B[_] #6', DRAM_STATIC)
    .bound_alloc('B7_cache: _', ['512', None])
## Case 8 memory staging
    .stage_window('B8_cache', 'B[_] #7', DRAM_STATIC)
    .bound_alloc('B8_cache: _', ['512', '64'])
## Replace SGEMM WINDOW with optimized form
# These must come AFTER bound_alloc since the internal check-effects
# is a whole program analysis that is VERY expensive
    .repeat(Procedure.call_eqv, sgemm_above_kernel, 'SGEMM_WINDOW(_)')
# Clean up
    .simplify()

```

User Scheduling

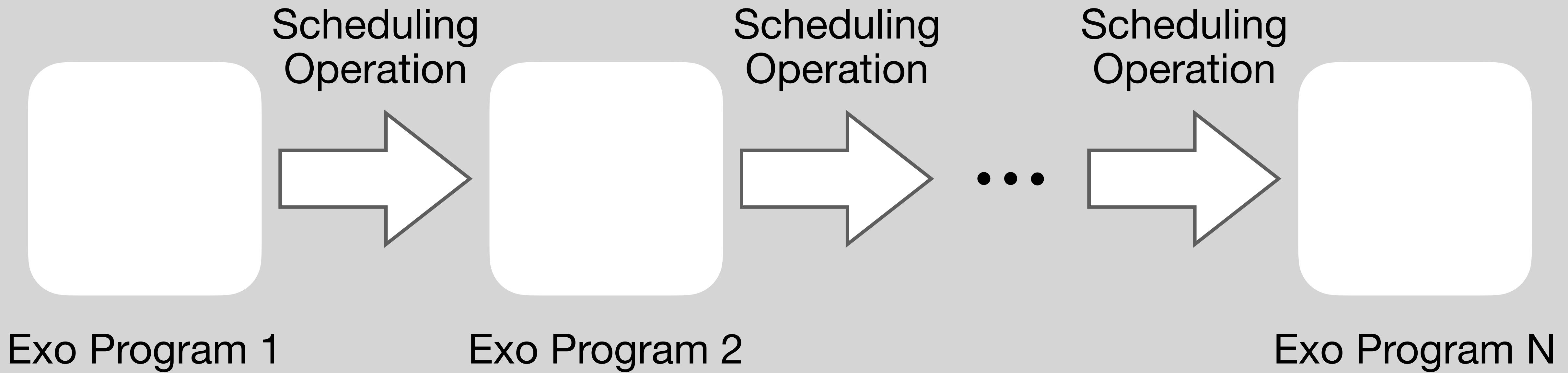
A new approach

Scheduling as Rewriting, not Annotation

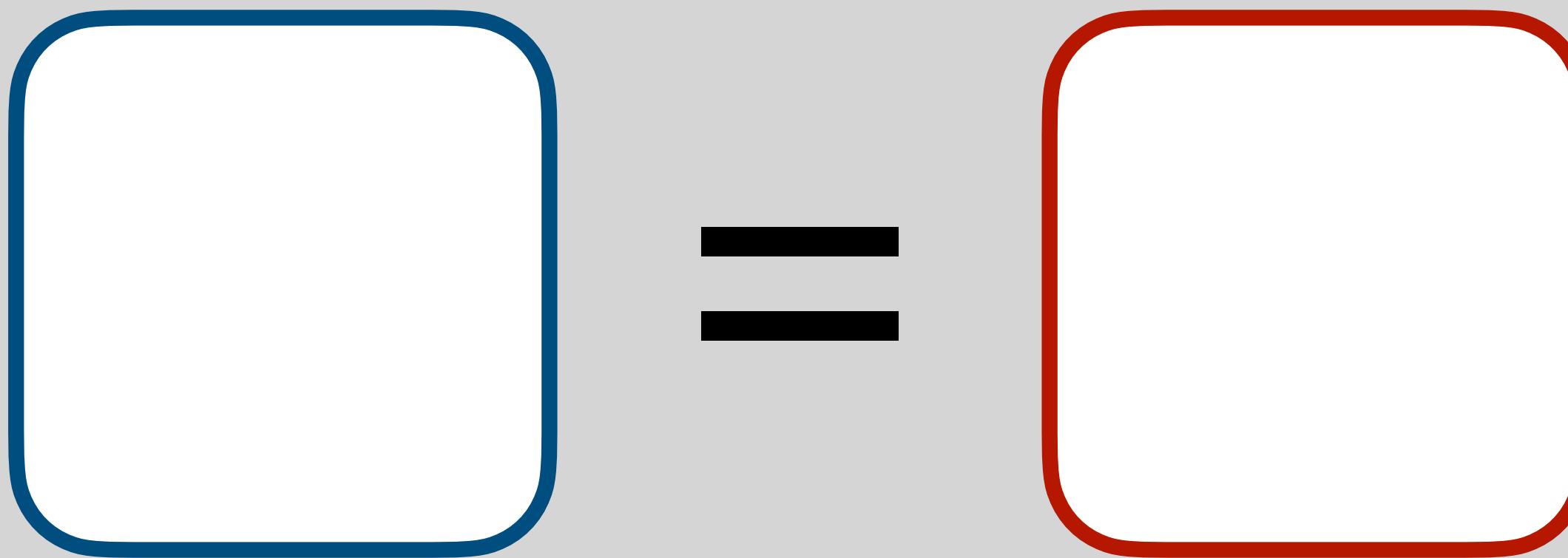
Scheduling of Sub-procedures

```
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]
```

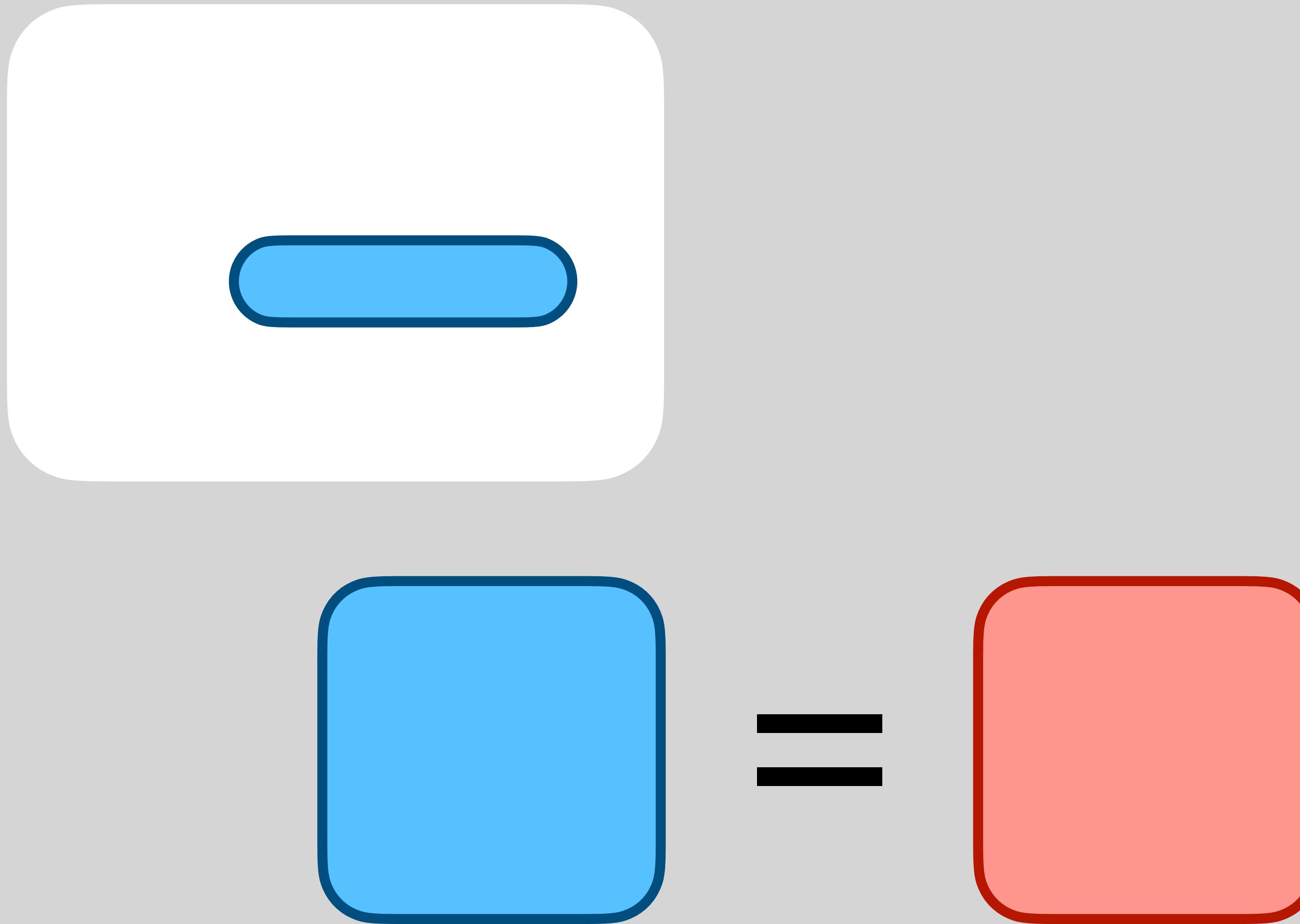
Scheduling as Program Rewriting



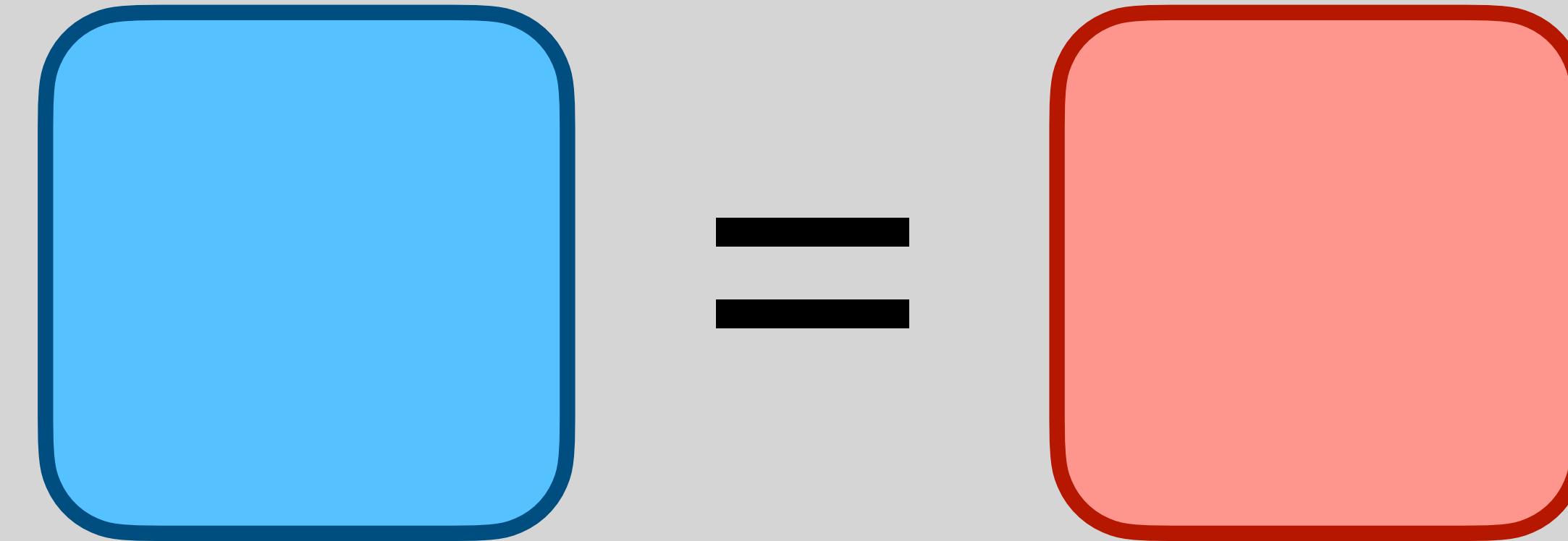
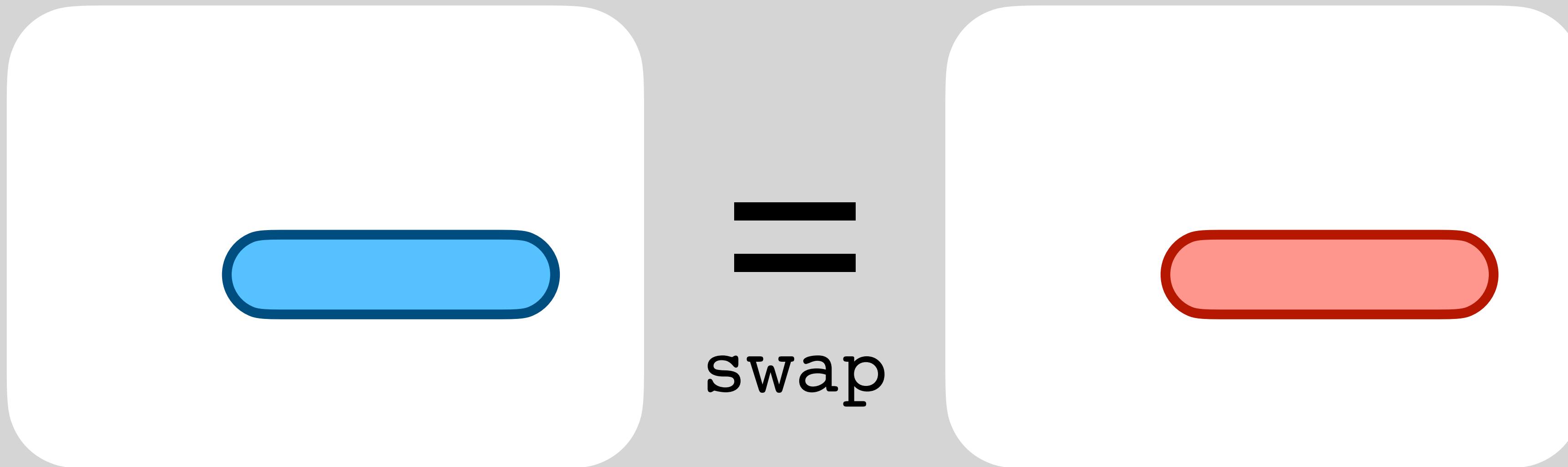
Scheduling as Program Rewriting



Scheduling as Program Rewriting



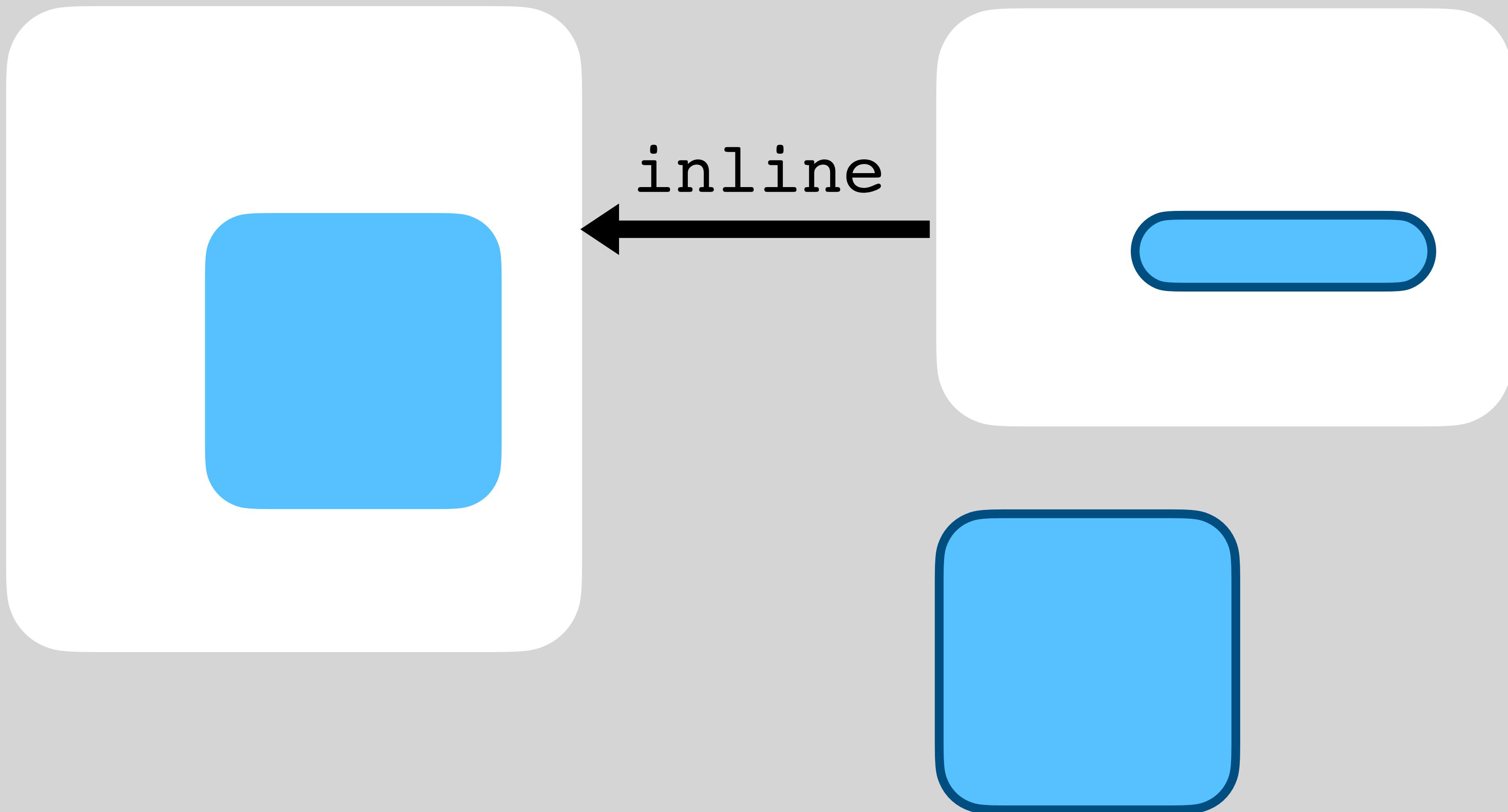
Scheduling as Program Rewriting



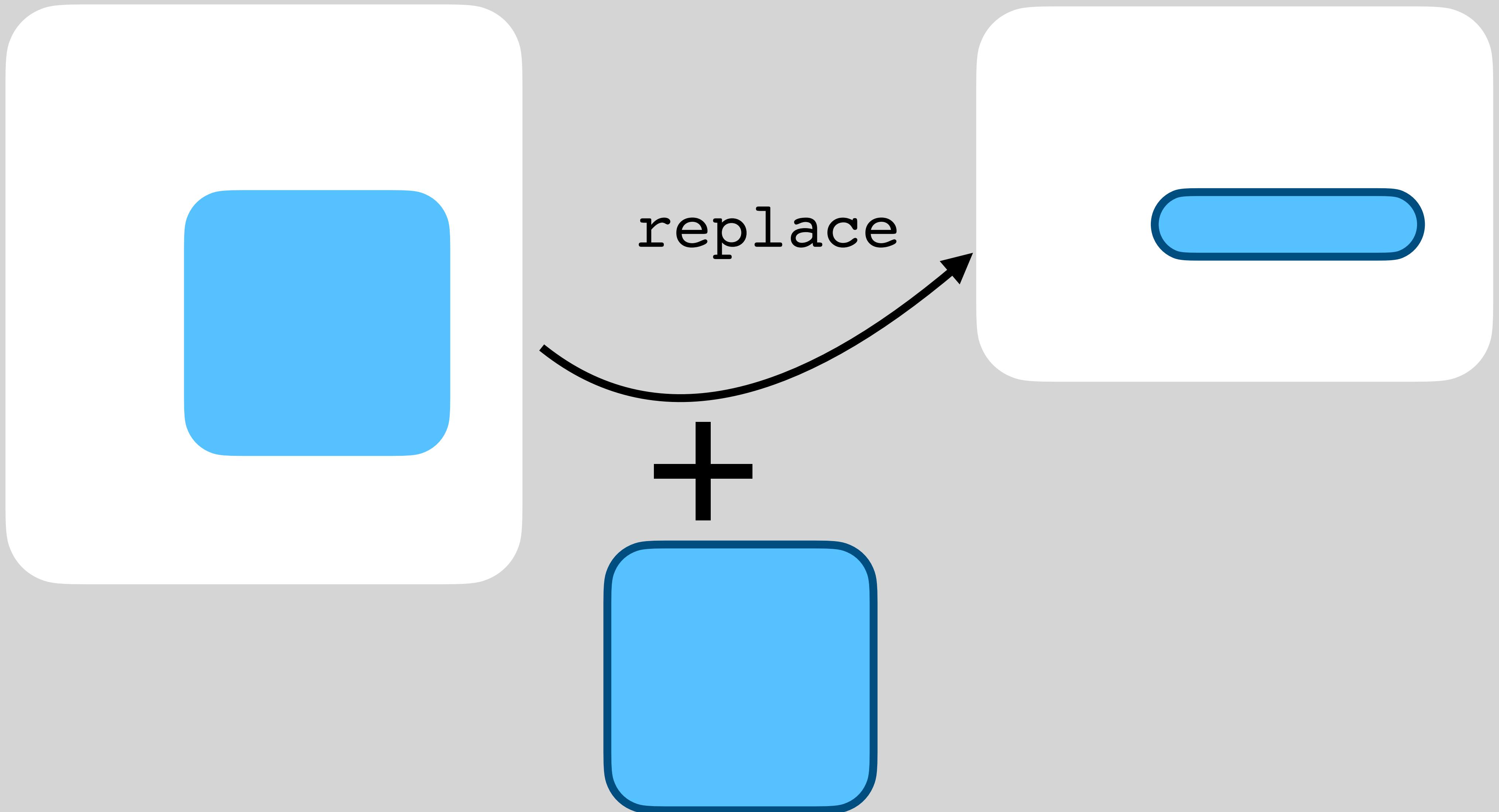
Scheduling Sub-procedures



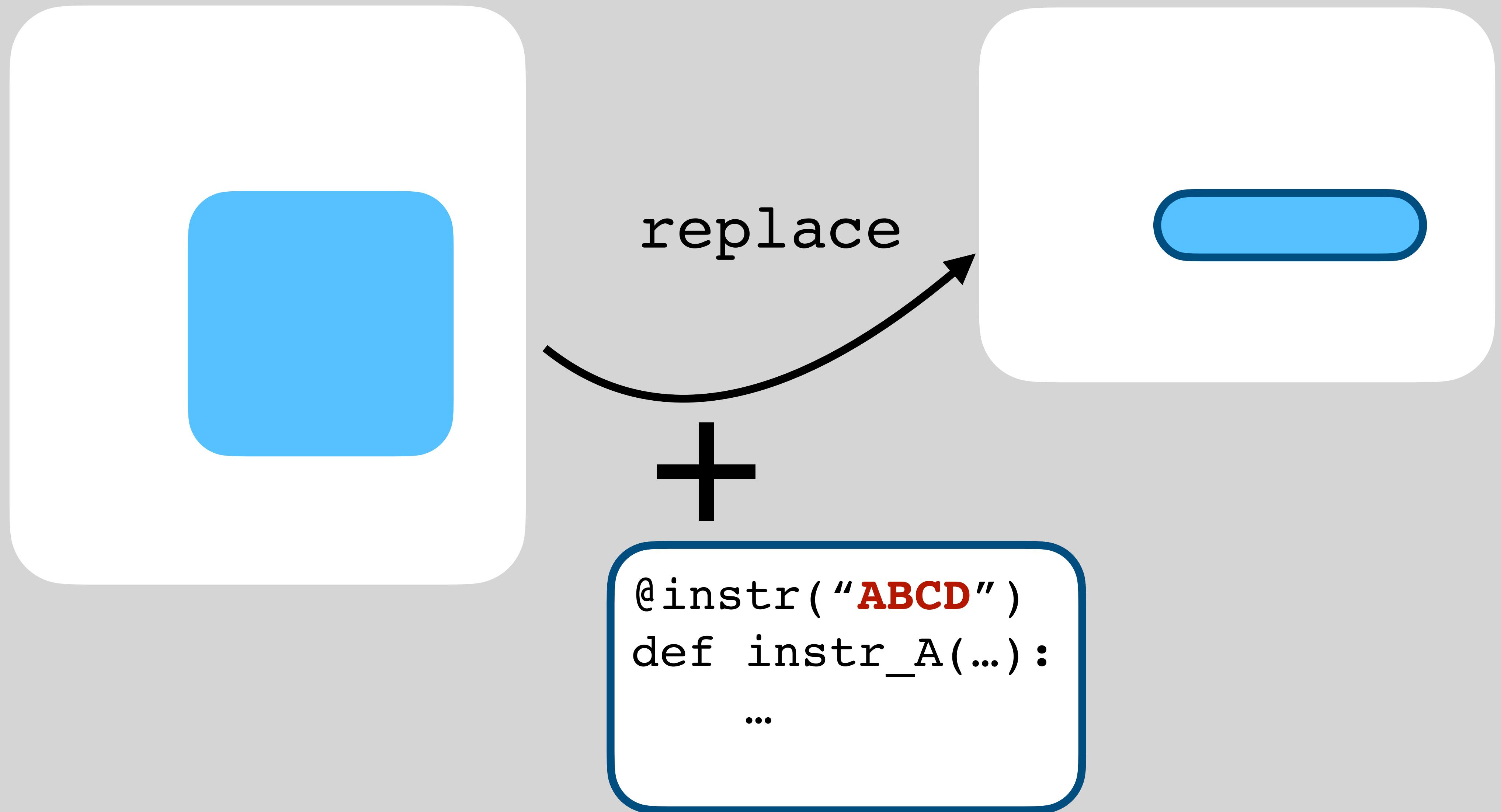
Scheduling Sub-procedures



Scheduling Sub-procedures



Scheduling Sub-procedures



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

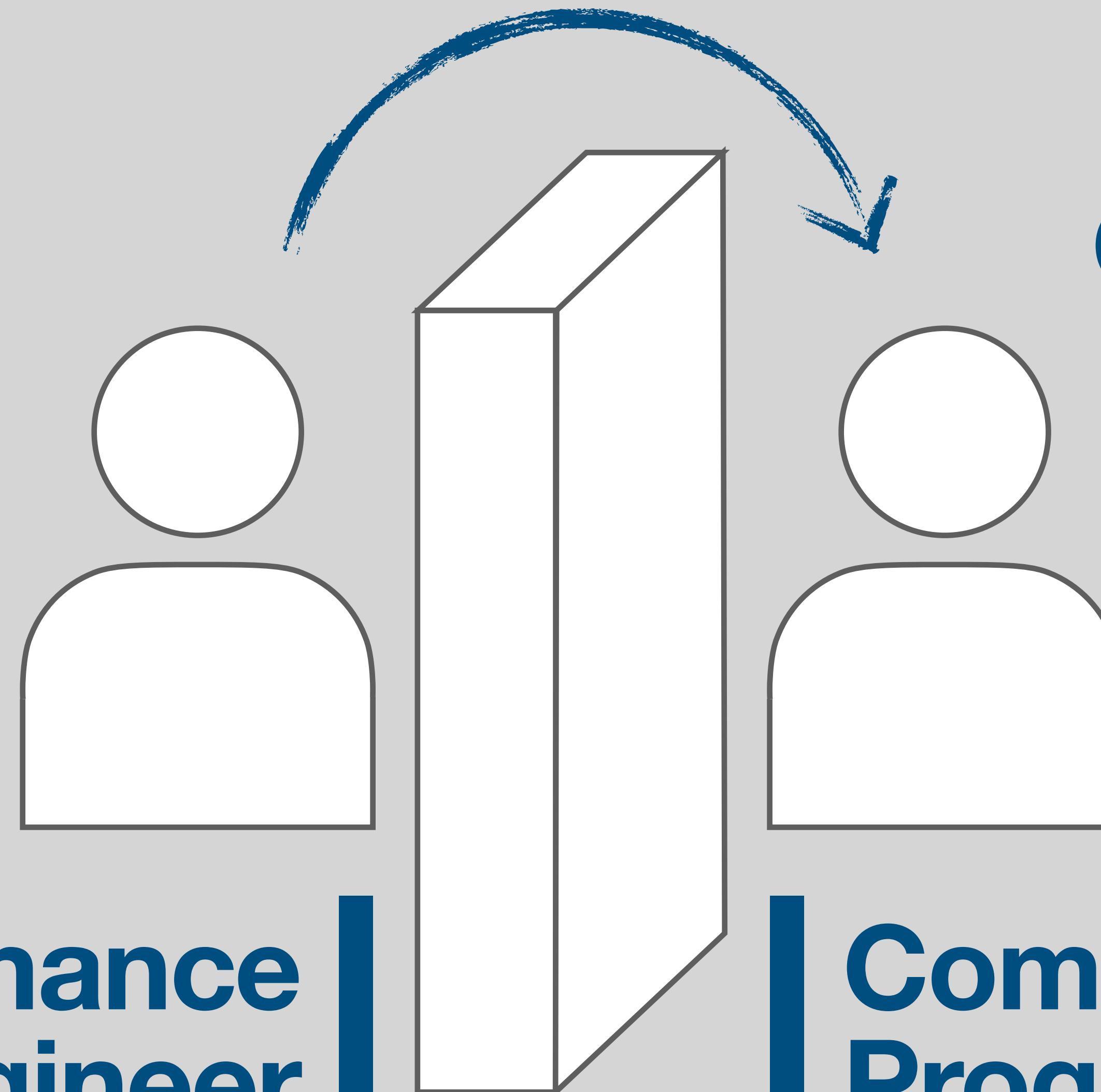
lift_alloc

Safety checked using SMT verification

stage_mem
bind_config
reorder_dim
expand_dim
add_guard

partition
remove_loop
...

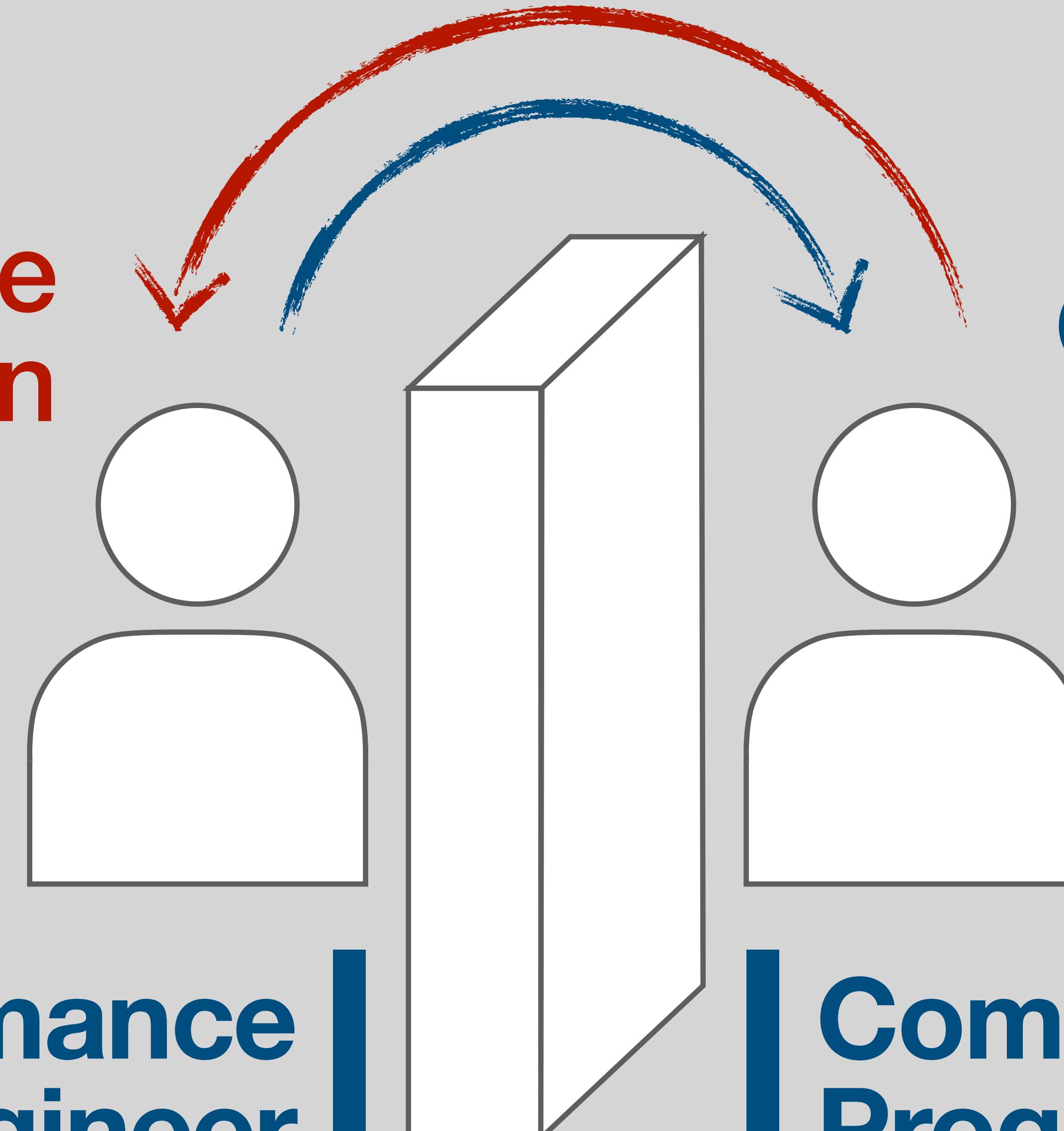
**Performance
Engineer**



Correctness

**Compiler
Programmer**

**Performance
Optimization**



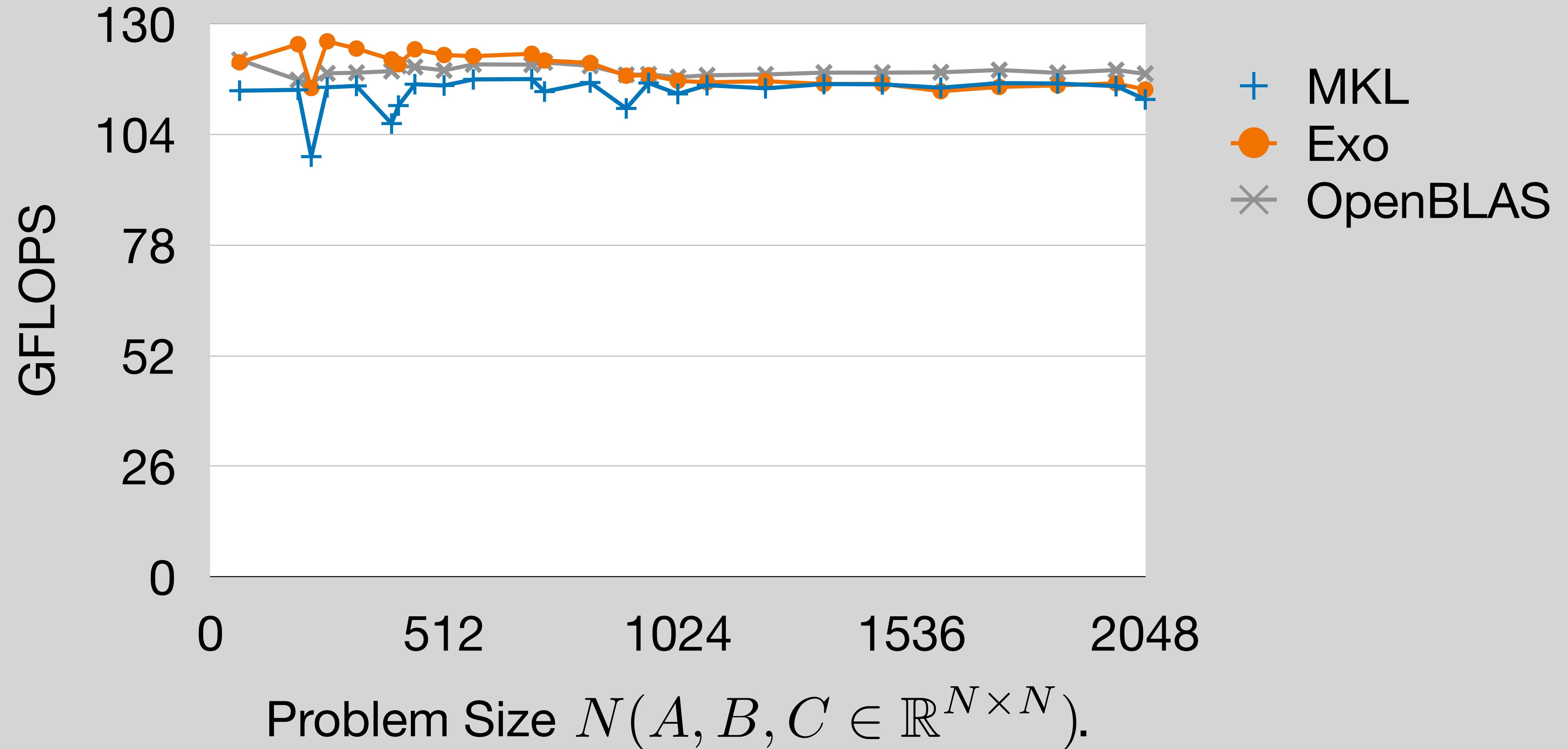
**Performance
Engineer**

**Compiler
Programmer**

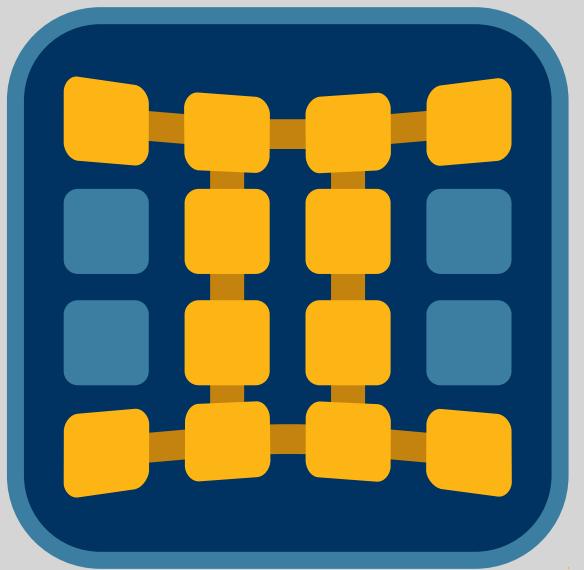
Exo Performance Results

AVX-512 MatMul

Matches best possible performance



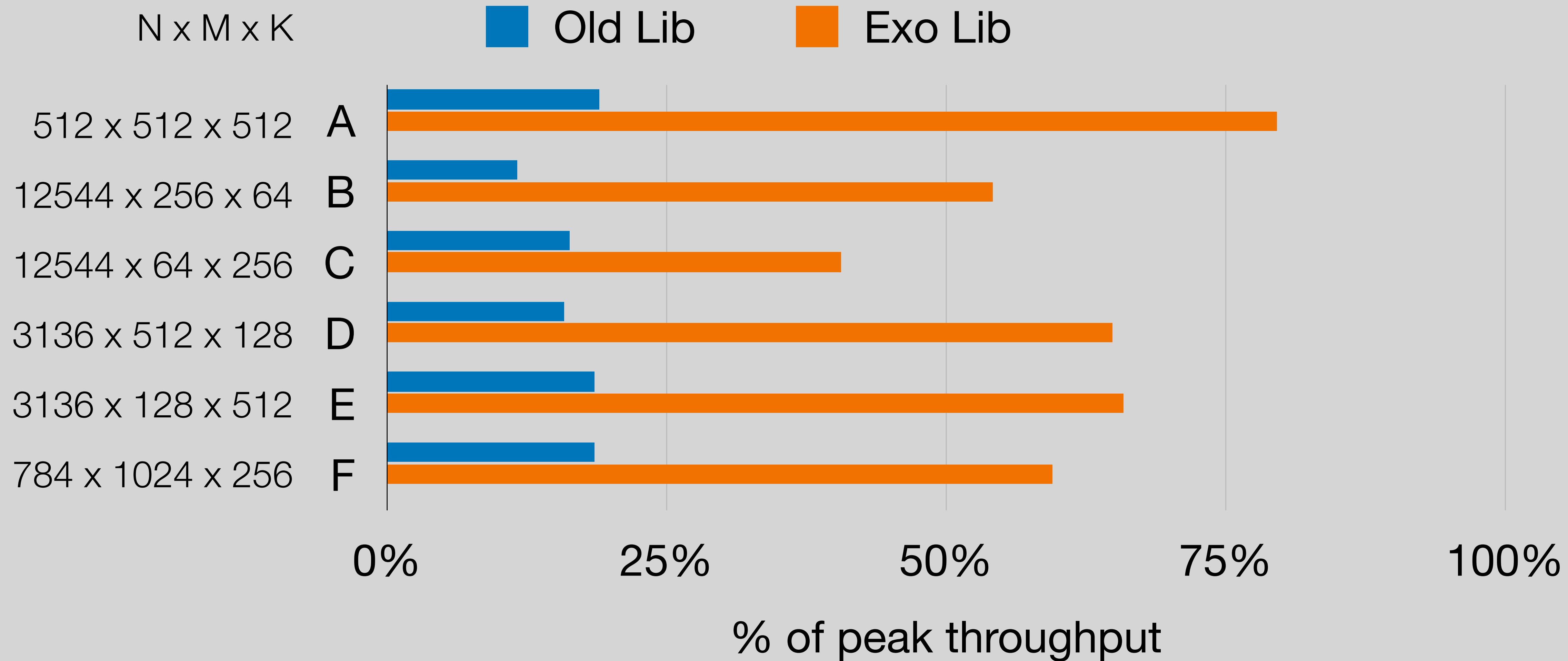
UC Berkeley



GEMINI

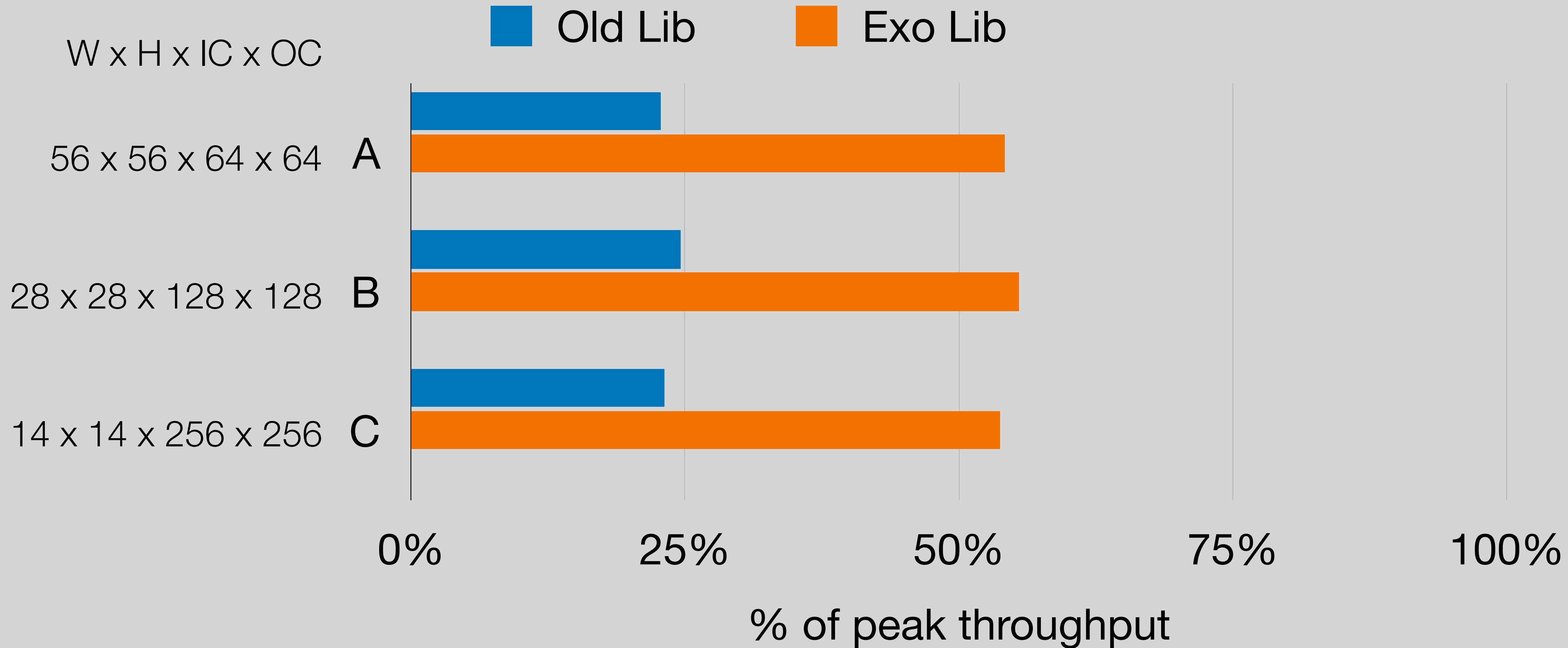
GEMMINI MatMul

2.3x - 3.7x faster than original HPC Library

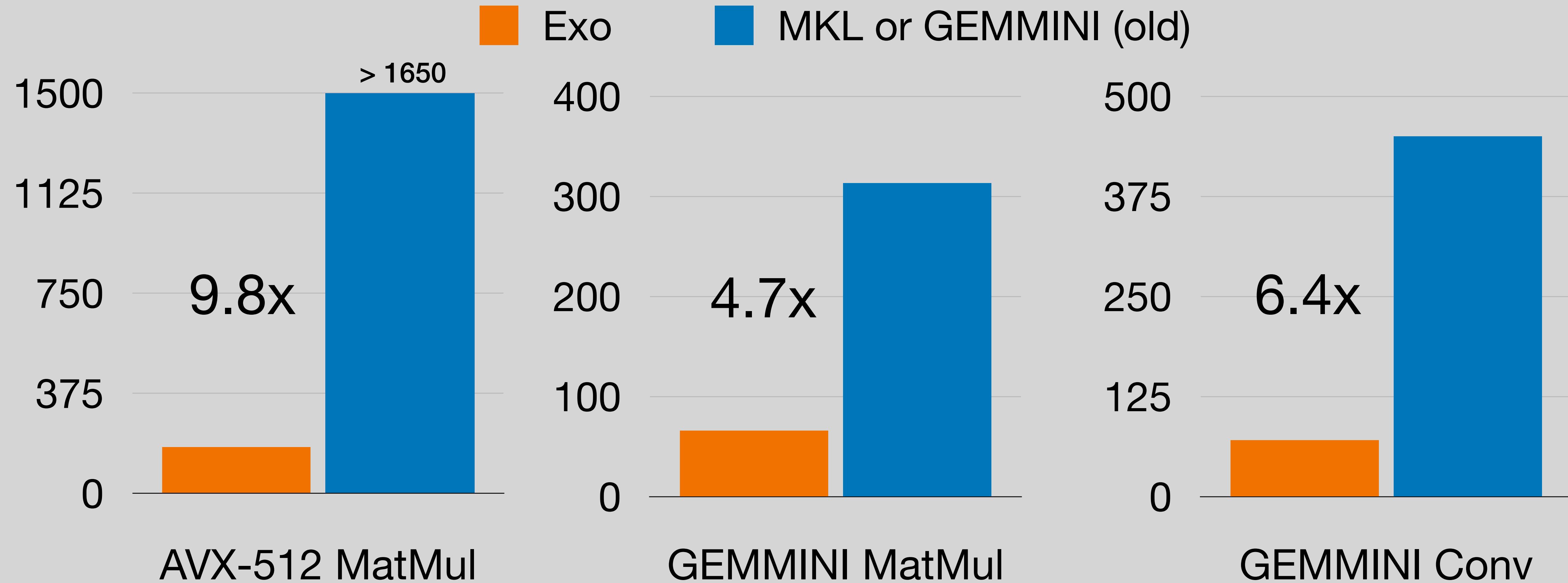


GEMMINI Conv

Similar Story for Convolution Layers



Exo vs. Existing - Lines of Code



HW / SW Co-Design

Benefits to Software Maintenance

ISA Change

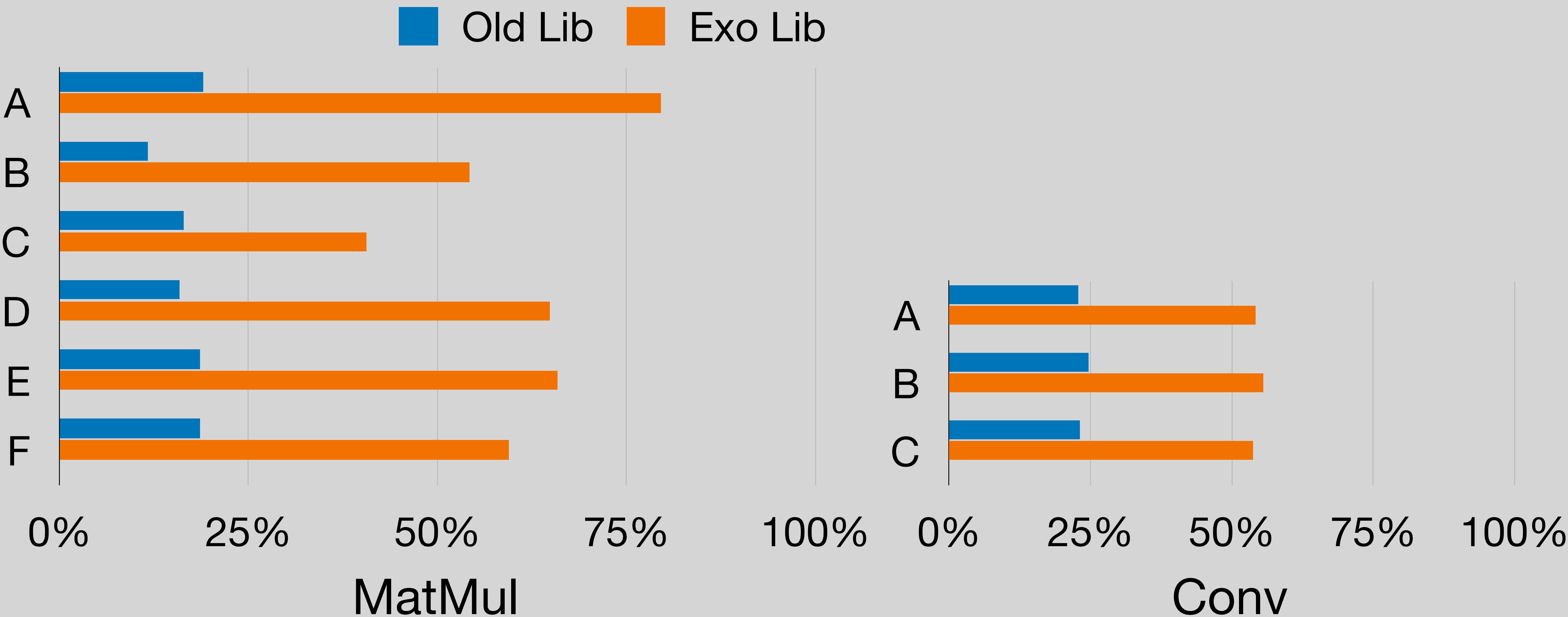


5 lines of Exo code changed

46 lines of old handwritten GEMMINI
library changed

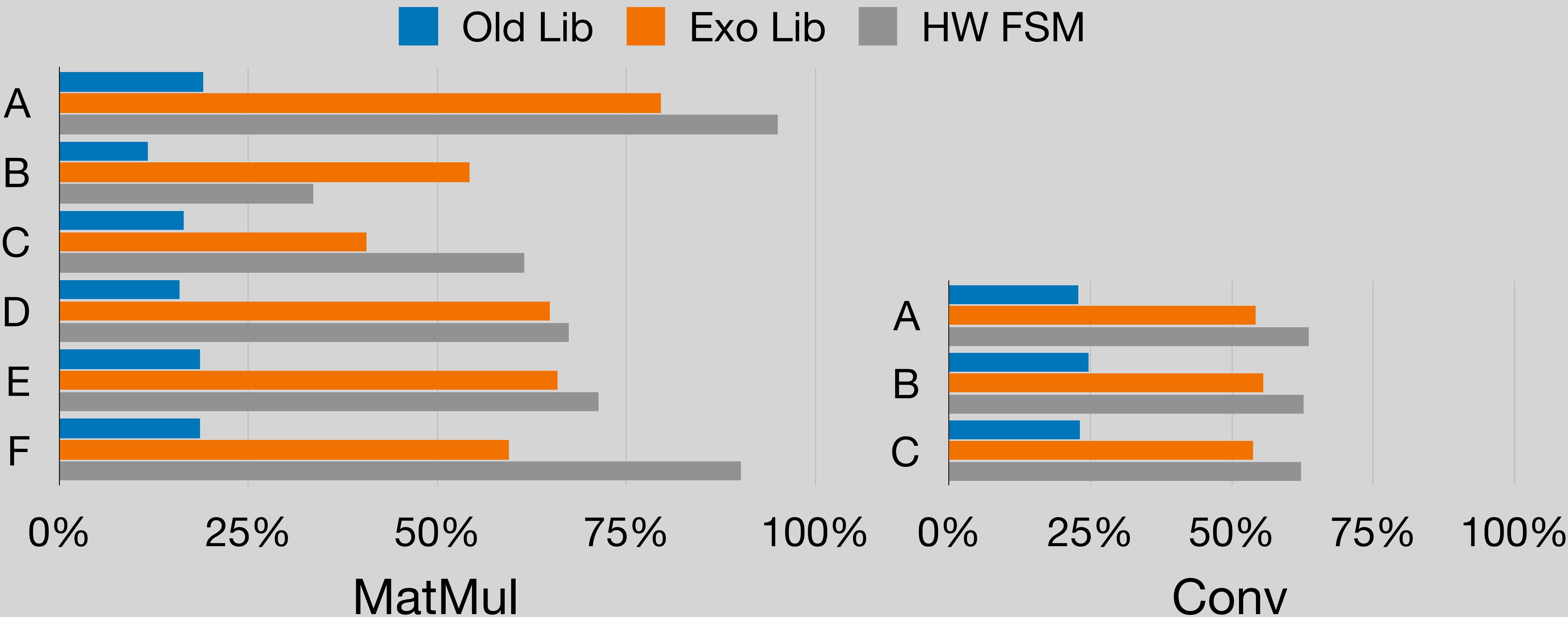
GEMMINI MatMul + Conv

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

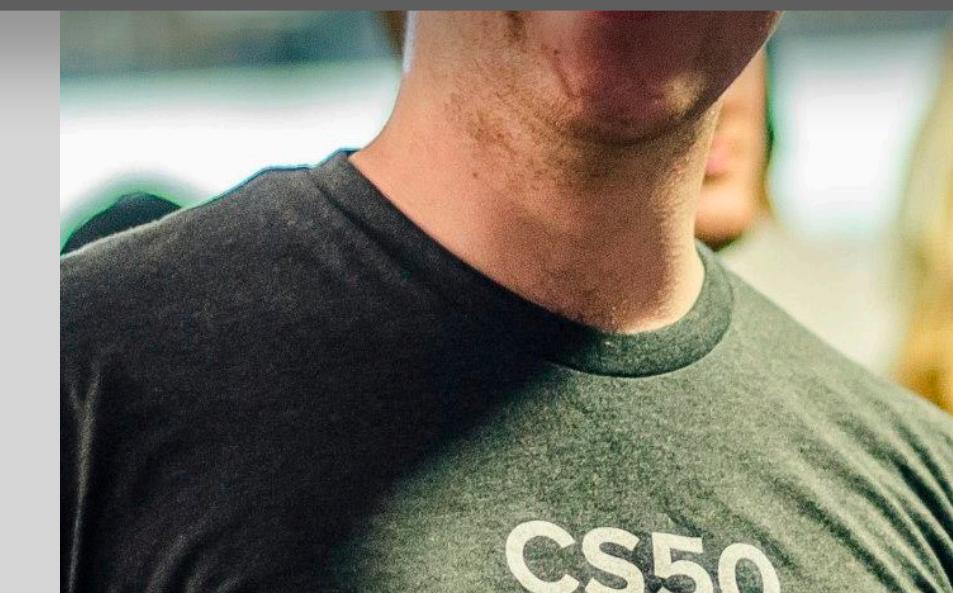
<https://exo-lang.dev>



Gilbert
Bernstein



Yuka Ikarashi



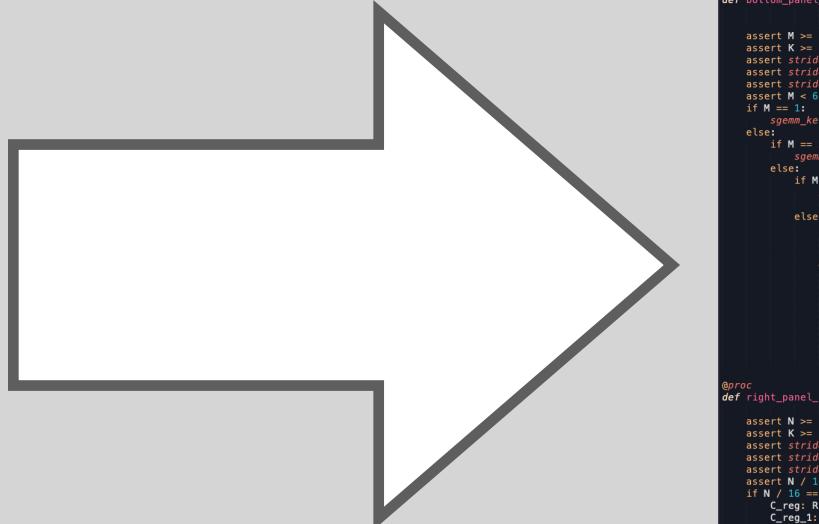
Alex Reinking



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

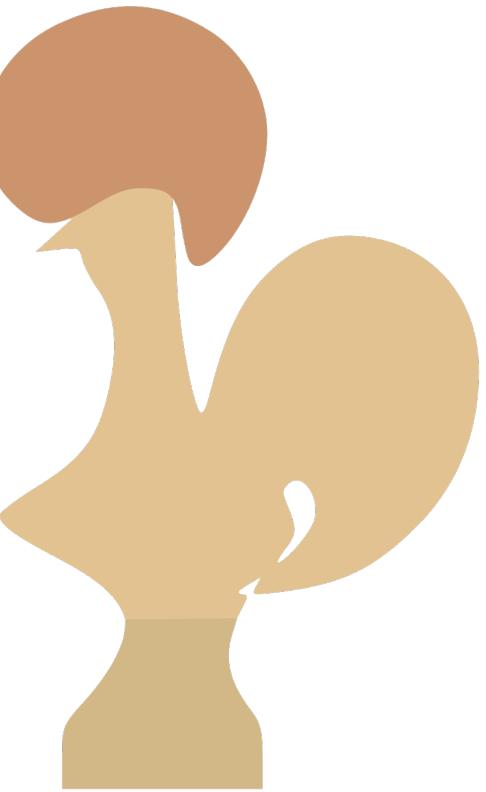
```
for i in seq(0,N):  
    for j in seq(0,M):  
        for k in seq(0,K):  
            C[i,j] += A[i,k] *
```



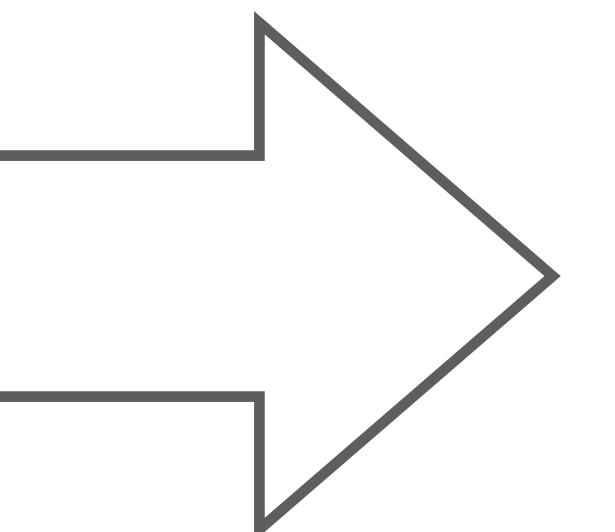
Functional Scheduling



Coq



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



$$N/16 \quad M/16 \quad K/16$$


$i=0$ $j=0$ $k=0$

```
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]
```

```

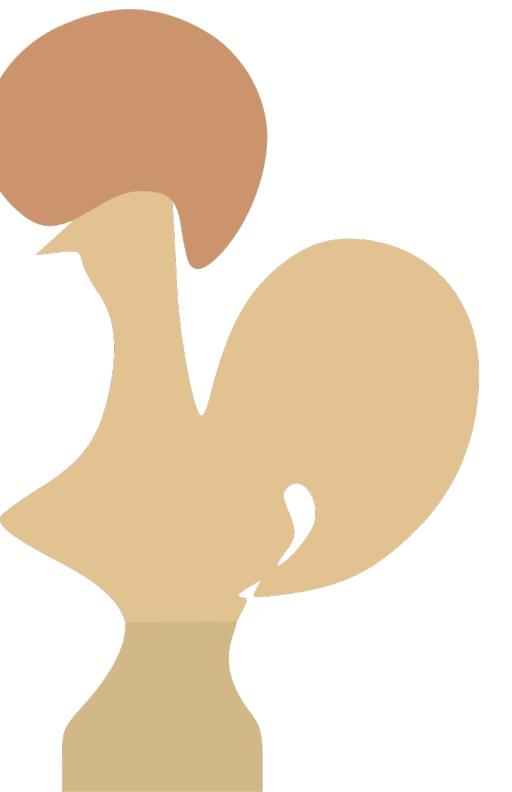
mm512_set1_ps(A, reg, A[i, k:k+1])
B_reg: R[16] @ AVX512
mm512_loadu_ps(B[reg:0:16], B[k, 16 * jo:16 * jo + 16])
mm512_set1_ps(A, reg, A[i, k:k+1])
A_reg: R[16] @ AVX512
mm512_mask_set1_ps(N % 16, A, reg2, C, reg[i, jo, 0:16])
B_reg: R[16] @ AVX512
mm512_loadu_ps(N % 16, B, reg2[0:16], B[k, 16:N])
mm512_mask_fmad_ps(N % 16, A, reg2, B, reg2, C, reg_1[i, 0:16])
for i in par(0, 6):
    mm512_loadu_ps(C, reg[i, jo, 0:16], C[i, 16 * jo:16 * jo + 16])
par(0, 6):
    A_vec: R[16] @ AVX512
    mm512_set1_ps(A, vec, A[i, k:k+1])
    for jo in par(0, 4):
        B_vec: R[16] @ AVX512
        mm512_loadu_ps(B, vec[0:16], B[k, 16 * jo:16 * jo + 16])
        mm512_fmad_ps(A, vec, B, vec, C, reg[i, jo, 0:16])
    par(0, 6):
        mm512_stores_ps(C[i, 16 * jo:16 * jo + 16], C, reg[i, jo, 0:16])
    anel_kernel_scheduled(M: size, K: size, A: [f32][M, K] @ DRAM,
    B: [f32][K, 64] @ DRAM,
    C: [f32][M, 64] @ DRAM):
        >= 1
        tride(A, 1) == 1
        tride(B, 1) == 1
        tride(C, 1) == 1
        < 6
        1:
        _kernel_avx512_1x4(K, A[0:1, 0:K], B[0:K, 0:64], C[0:1, 0:64])
        >= 2:
        sgemm_kernel_avx512_2x4(K, A[0:2, 0:K], B[0:K, 0:64], C[0:2, 0:64])
        if M == 3:
            sgemm_kernel_avx512_3x4(K, A[0:3, 0:K], B[0:K, 0:64], C[0:3, 0:64])
        else:
            if M == 4:
                sgemm_kernel_avx512_4x4(K, A[0:4, 0:K], B[0:K, 0:64],
                C[0:4, 0:64])
            else:
                if M == 5:
                    sgemm_kernel_avx512_5x4(K, A[0:5, 0:K], B[0:K, 0:64],
                    C[0:5, 0:64])
                else:
                    for k in par(0, K):
                        for i in par(0, M):
                            for j in par(0, 64):
                                Ci, j1 == A[i, k] * B[k, j]
        nel_kernel_scheduled(N: size, K: size, A: [f32][6, K] @ DRAM,
        B: [f32][K, N] @ DRAM, C: [f32][6, N] @ DRAM):
            >= 1
            tride(A, 1) == 1
            tride(B, 1) == 1
            tride(C, 1) == 1
            < 6
            4
            g: R[5, 1, 16] @ AVX512
            g: R[6, 16] @ AVX512
            i in par(0, 6):
                mm512_maskz_loadu_ps(N, C, reg_1[i, 0:16], C[i, 0:N])
            k in par(0, K):
                for i in par(0, 6):
                    A_reg: R[16] @ AVX512
                    mm512_set1_ps(A, reg, A[i, k:k+1])
                    B_reg: R[16] @ AVX512
                    mm512_loadu_ps(B, reg[0:16], B[k, 16 * jo:16 * jo + 16])
                    mm512_fmad_ps(A, reg, B, reg, C, reg[i, jo, 0:16])
                par(0, 6):
                    for jo in par(0, 3):
                        mm512_stores_ps(C[i, 16 * jo:16 * jo + 16], C, reg[i, jo, 0:16])
                    mm512_mask_stores_ps(N % 16, C[i, 16:N], C, reg_1[i, 0:16])
                for i in par(0, 6):
                    for jo in par(0, 1):
                        mm512_stores_ps(C[i, 16 * jo:16 * jo + 16], C, reg[i, jo, 0:16])
                    mm512_mask_stores_ps(N % 16, C[i, 16:N], C, reg_1[i, 0:16])
                else:
                    if N / 16 == 2:
                        C, reg: R[6, 3, 16] @ AVX512
                        C, reg_1: R[6, 16] @ AVX512
                        for i in par(0, 6):
                            for jo in par(0, 2):
                                mm512_loadu_ps(C, reg[i, jo, 0:16],
                                C[i, 16 * jo:16 * jo + 16])
                                mm512_maskz_loadu_ps(N % 16, C, reg_1[i, 0:16], C[i, 32:N])
                        k in par(0, K):
                            for i in par(0, 6):
                                for jo in par(0, 2):
                                    A, reg: R[16] @ AVX512
                                    mm512_set1_ps(A, reg, A[i, k:k+1])
                                    B, reg: R[16] @ AVX512
                                    mm512_loadu_ps(B, reg[0:16],
                                    B[k, 16 * jo:16 * jo + 16])
                                    mm512_fmad_ps(A, reg, B, reg, C, reg[i, jo, 0:16])
                                A, reg2: R[16] @ AVX512
                                mm512_set1_ps(A, reg2, A[i, k:k+1])
                                B, reg2: R[16] @ AVX512
                                mm512_loadu_ps(N % 16, A, reg2[0:16], B[k, 32:N])
                                mm512_fmad_ps(N % 16, A, reg2, B, reg2, C, reg_1[i, 0:16])
                            for i in par(0, 6):
                                for jo in par(0, 2):
                                    mm512_stores_ps(C[i, 16 * jo:16 * jo + 16],
                                    C, reg[i, jo, 0:16])
                                mm512_mask_stores_ps(N % 16, C[i, 16:N], C, reg_1[i, 0:16])
                            else:
                                if N / 16 == 3:
                                    C, reg: R[6, 4, 16] @ AVX512
                                    C, reg_1: R[6, 16] @ AVX512
                                    for i in par(0, 6):
                                        for jo in par(0, 3):
                                            mm512_loadu_ps(C, reg[i, jo, 0:16],
                                            C[i, 16 * jo:16 * jo + 16])
                                            mm512_fmad_ps(A, reg, B, reg, C, reg[i, jo, 0:16])
                                    if N / 16 == 4:
                                        for k in par(0, K):
                                            for ii in par(0, N / 64):
                                                bottom_panel_kernel_scheduled(
                                                M % 6, K, A[i, * :M / 6, 0:K], B[0:K, 64 * jo:64 * jo + 64],
                                                C[i, * :M / 6, 0:6, 64 * (N / 64):N])
                                    if N / 64 > 0:
                                        for k in par(0, K):
                                            for ii in par(0, N / 64):
                                                for jj in par(0, N / 64):
                                                    Ci[i + M / 6 * 6, jj + N / 64 * 64] += A[i + M / 6 * 6, jj + N / 64 * 64]
        @proc def sgemm_above_kernel(M: size, N: size, K: size, A: [f32][M, K] @ DRAM,
        B: [f32][K, N] @ DRAM, C: [f32][M, N] @ DRAM):
            assert M >= 1
            assert N >= 1
            assert K >= 1
            assert stride(A, 1) == 1
            assert stride(B, 1) == 1
            assert stride(C, 1) == 1
            for i in par(0, M / 6):
                for jo in par(0, N / 64):
                    sgemm_kernel_avx512_6x4(K, A[6 * io:6 * io + 6, 0:K], B[0:K, 64 * jo:64 * jo + 64],
                    C[6 * io:6 * io + 6, 64 * (N / 64):N])
            if N / 64 > 0:
                for i in par(0, M / 6):
                    right_panel_kernel_scheduled(N % 64, K, A[6 * io:6 * io + 6, 0:K],
                    B[0:K, 64 * (N / 64):N], C[6 * io:6 * io + 6, 64 * (N / 64):N])
            if M / 6 > 0:
                for jo in par(0, N / 64):
                    bottom_panel_kernel_scheduled(
                    M % 6, K, A[i, * :M / 6, 0:K], B[0:K, 64 * jo:64 * jo + 64],
                    C[i, * :M / 6, 0:6, 64 * (N / 64):N])
            if N / 64 > 0:
                for k in par(0, K):
                    for ii in par(0, N / 64):
                        for jj in par(0, N / 64):
                            Ci[i + M / 6 * 6, jj + N / 64 * 64] += A[i + M / 6 * 6, jj + N / 64 * 64]
        @proc def sgemm_sys_atl(M: size, N: size, K: size, A: [f32][M, K] @ DRAM,
        B: f32[K, N] @ DRAM, C: f32[M, N] @ DRAM):
            assert M >= 1
            assert N >= 1
            assert K >= 1
            assert stride(A, 1) == 1
            assert stride(B, 1) == 1
            assert stride(C, 1) == 1
            A1_cache: f32[264, 512] @ DRAM_STATIC
            B1_cache: f32[512, 64] @ DRAM_STATIC
            for k in par(0, K / 512):
                for io in par(0, M / 264):
                    for i in par(0, 16):
                        for j in par(0, 512):
                            A1_cache[i, 16 * jo:16 * jo + 16] = A[264 * io + 10, 512 * ko + i1]
                for jo in par(0, N / 64):
                    for i in par(0, 512):
                        for k1 in par(0, 16):
                            B1_cache[i, 16 * jo:16 * jo + 16] = B[512 * ko + i0, 64 * jo + i1]
            sgemm_above_kernel(
                264, 64, K % 512, A[264 * (M / 264):M, 512 * (K / 512):K],
                B1_cache[0:512, 0:N - 64 * (N / 64):N], C[264 * (M / 264):M, 64 * (N / 64):N])
            if N / 64 > 0:
                for k in par(0, K / 512):
                    for i0 in par(0, M / 264):
                        for i1 in par(0, N / 64):
                            B2_cache: f32[512, 64] @ DRAM_STATIC
                            for i0 in par(0, K - 512 * (K / 512)):
                                for i1 in par(0, 64):
                                    B2_cache[i0, i1] = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
            sgemm_above_kernel(
                264, 64, K % 512, A[264 * io:264 * io + 264, 512 * (K / 512):K],
                B2_cache[0:N - 64 * (N / 64):N], C[264 * io:264 * io + 264, 64 * (N / 64) + i1])
            if K % 512 > 0:
                for io in par(0, M / 264):
                    for jo in par(0, N / 64):
                        B5_cache: f32[512, 64] @ DRAM_STATIC
                        for i0 in par(0, K - 512 * (K / 512)):
                            for i1 in par(0, 64):
                                B5_cache[i0, i1] = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
            sgemm_above_kernel(
                264, 64, K % 512, A[264 * io:264 * io + 264, 512 * (K / 512):K],
                B5_cache[0:N - 64 * (N / 64):N], C[264 * io:264 * io + 264, 64 * (N / 64) + i1])
            if K % 512 == 0:
                if N / 64 > 0:
                    for i0 in par(0, M / 264):
                        for i1 in par(0, N / 64):
                            B6_cache: f32[512, 64] @ DRAM_STATIC
                            for i0 in par(0, K - 512 * (K / 512)):
                                for i1 in par(0, 64):
                                    B6_cache[i0, i1] = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
                    sgemm_above_kernel(
                    264, 64, K % 512, A[264 * io:264 * io + 264, 512 * (K / 512):K],
                    B6_cache[0:N - 64 * (N / 64):N], C[264 * io:264 * io + 264, 64 * (N / 64) + i1])
                if M / 264 > 0:
                    for jo in par(0, N / 64):
                        B7_cache: f32[512, 64] @ DRAM_STATIC
                        for i0 in par(0, K - 512 * (K / 512)):
                            for i1 in par(0, 64):
                                B7_cache[i0, i1] = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
                    sgemm_above_kernel(
                    M % 264, 64, K % 512, A[264 * (M / 264):M, 512 * (K / 512):K],
                    B7_cache[0:N - 64 * (N / 64):N], C[264 * (M / 264):M, 64 * (N / 64):N])
                if N / 64 > 0:
                    for k in par(0, K / 512):
                        for i0 in par(0, M / 264):
                            for i1 in par(0, N / 64):
                                B8_cache: f32[512, 64] @ DRAM_STATIC
                                for i0 in par(0, K - 512 * (K / 512)):
                                    for i1 in par(0, 64):
                                        B8_cache[i0, i1] = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
                    sgemm_above_kernel(
                    M % 264, 64, K % 512, A[264 * (M / 264):M, 512 * (K / 512):K],
                    B8_cache[0:N - 64 * (N / 64):N], C[264 * (M / 264):M, 64 * (N / 64):N])
            if K % 512 > 0:
                for i0 in par(0, K / 512):
                    for i1 in par(0, M / 264):
                        for i2 in par(0, N / 64):
                            B9_cache: f32[512, 64] @ DRAM_STATIC
                            for i0 in par(0, K - 512 * (K / 512)):
                                for i1 in par(0, 64):
                                    B9_cache[i0, i1] = B[512 * (K / 512) + i0, 64 * (N / 64) + i1]
            sgemm_above_kernel(
                264, 64, K % 512, A[264 * (M / 264):M, 512 * (K / 512):K],
                B9_cache[0:N - 64 * (N / 64):N], C[264 * (M / 264):M, 64 * (N / 64):N])

```

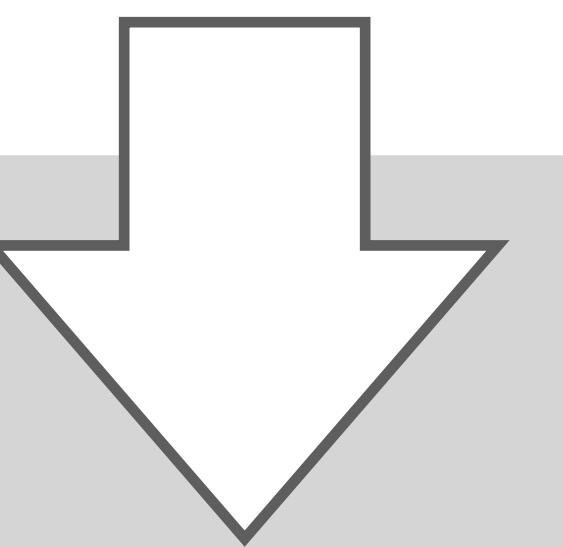
Functional Scheduling



Coq



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



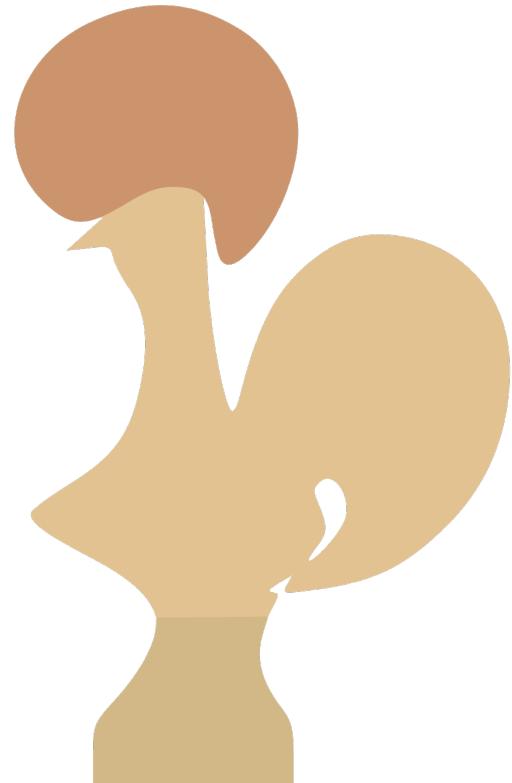
```
for i in seq(0,N):
    for j in seq(0,M):
        for k in seq(0,K):
            C[i,j] += A[i,k] * B[k,j]
```

First Formally Verified User-Schedulable Language

[POPL 2022]



Coq



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

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