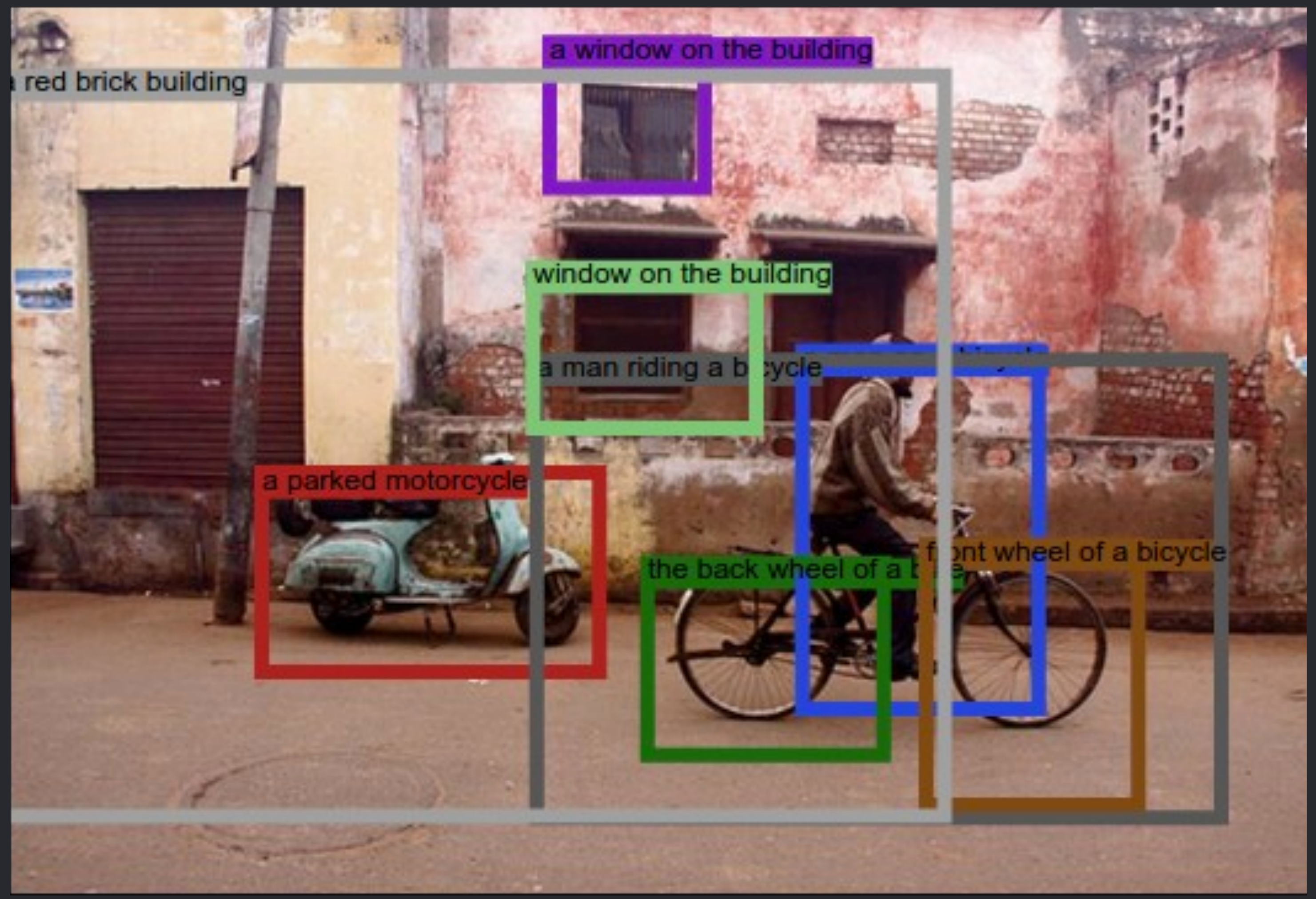
Automatically Scheduling Halide Image Processing Pipelines

Ravi Teja Mullapudi (CMU)
Andrew Adams (Google)
Dillon Sharlet (Google)
Jonathan Ragan-Kelley (Stanford)
Kayvon Fatahalian (CMU)

High demand for efficient image processing



Scheduling image processing algorithms

Algorithm description

```
Var x, y;
Func f, g;
g(x,y) = f(x,y) + …
h(x) = g(x,y) + …
```

Scheduling image processing algorithms

Algorithm description

```
Var x, y;
Func f, g;
g(x,y) = f(x,y) + …
h(x) = g(x,y) + …
```

Schedule (machine mapping)

```
parallelize y loop
tile output dims
vectorize y loop
```

Implementation

Scheduling image processing algorithms

Image processing
algorithm developers

Algorithm
description

```
Var x, y;
Func f, g;
g(x,y) = f(x,y) + …
h(x) = g(x,y) + …
```

Schedule
(machine mapping)

```
parallelize y loop
tile output dims
vectorize y loop
```

Implementation

Few developers have the skill set to author highly optimized schedules

Image processing algorithm developers

Algorithm description

```
Var x, y;
Func f, g;
g(x,y) = f(x,y) + …
h(x) = g(x,y) + …
```

Schedule (machine mapping)

```
parallelize y loop
tile output dims
vectorize y loop
```

> 10x Faster Implementation

Contribution: automatic scheduling of image processing pipelines

Image processing algorithm developers

Algorithm description

```
Var x, y;
Func f, g;
g(x,y) = f(x,y) + …
h(x) = g(x,y) + …
```

Generates expert-quality schedules in seconds

Scheduling Algorithm

> 10x Faster Implementation

Why is it challenging to schedule image processing pipelines?

Algorithm: 3x3 box blur

`in`

Algorithm: 3x3 box blur

`in`

`bx`

```
bx(x, y)  = (in(x-1, y) + in(x, y) + in(x+1, y))/3
```

Algorithm: 3x3 box blur

in

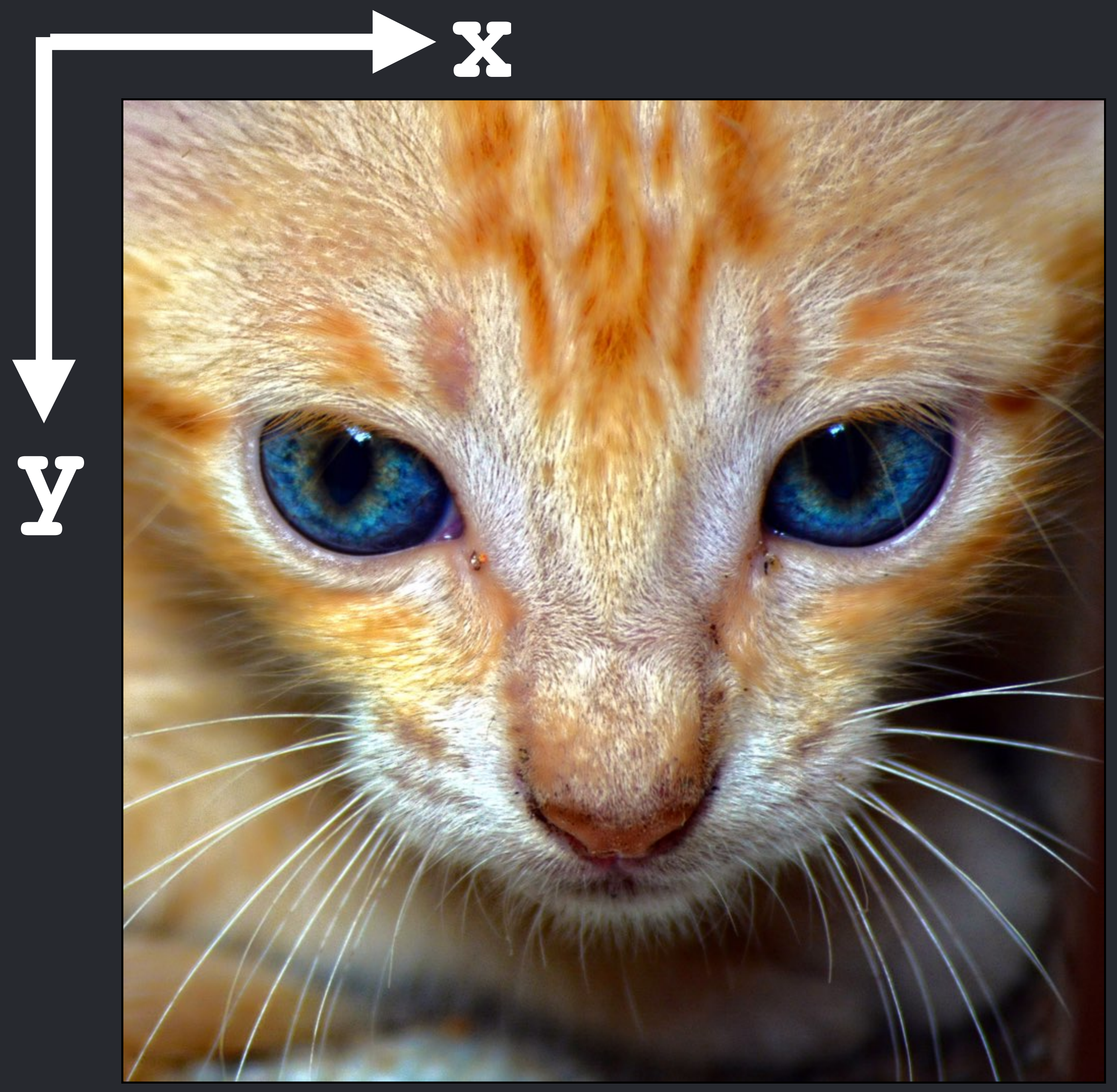
bx

out

```
bx(x, y)  = (in(x-1, y) + in(x, y) + in(x+1, y)) / 3
out(x, y) = (bx(x, y-1) + bx(x, y) + bx(x, y+1)) / 3
```

A basic (slow) schedule

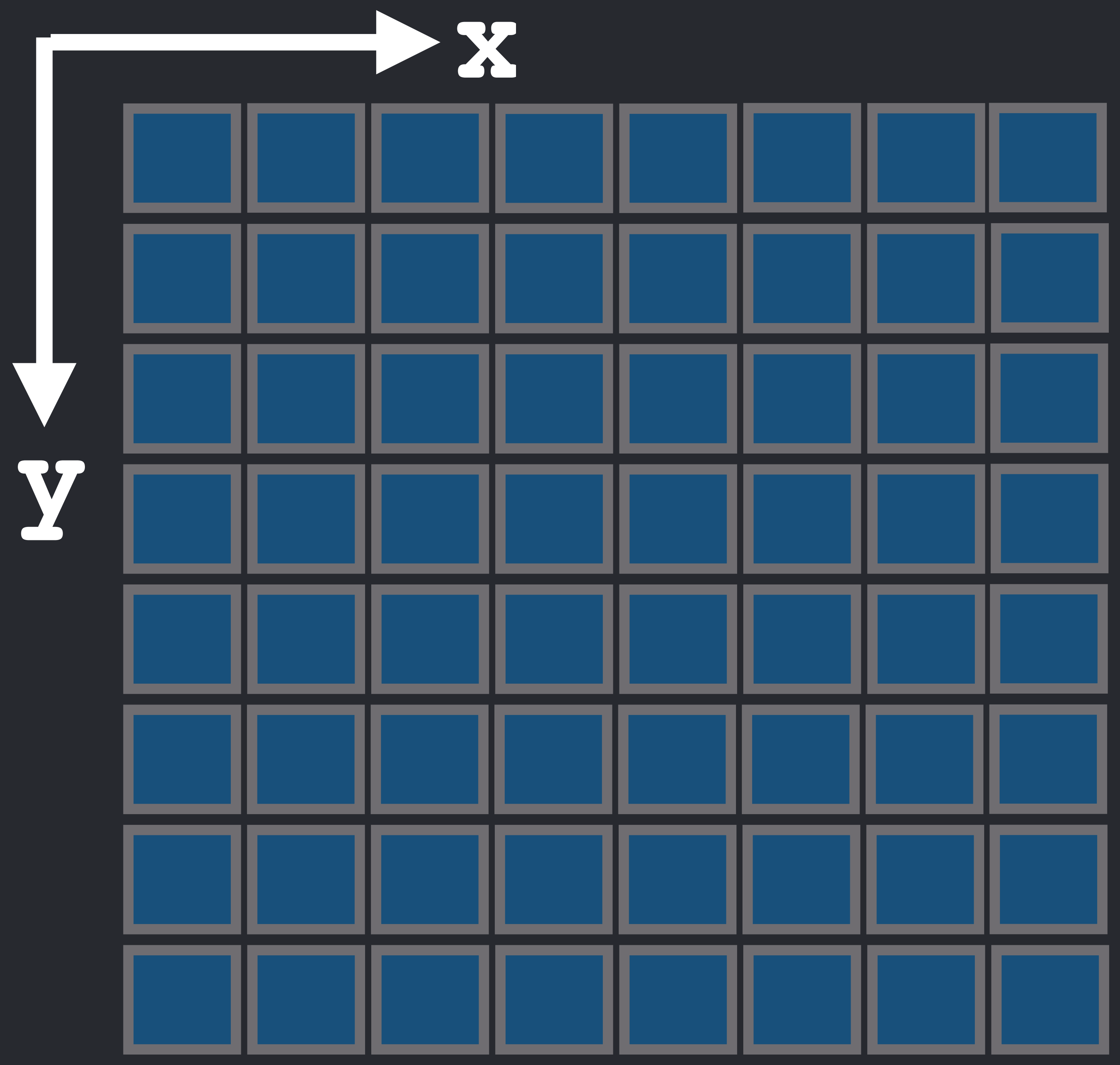
compute all pixels of bx, in parallel
compute all pixels of by, in parallel



in

A basic (slow) schedule

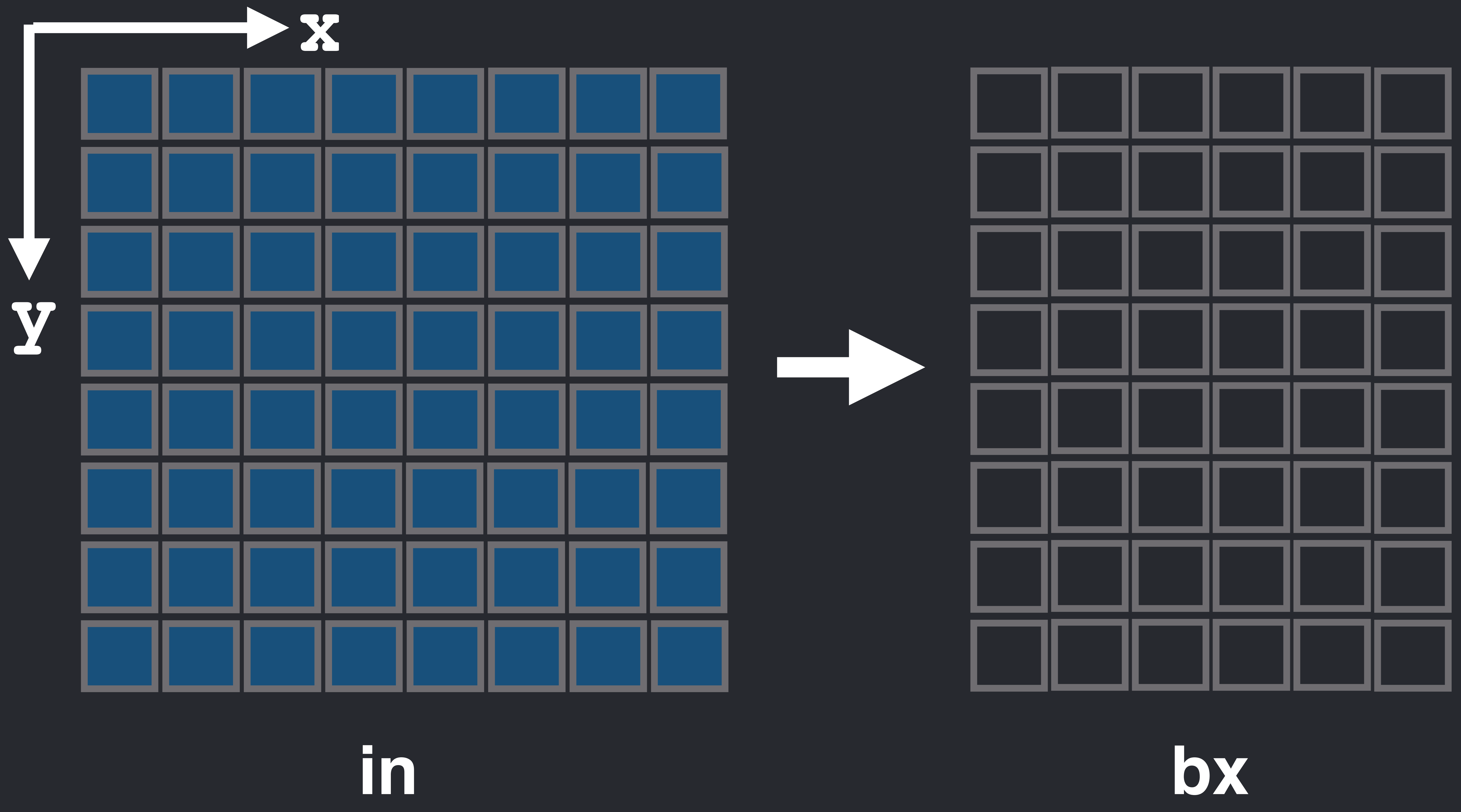
compute all pixels of bx, in parallel
compute all pixels of by, in parallel



in

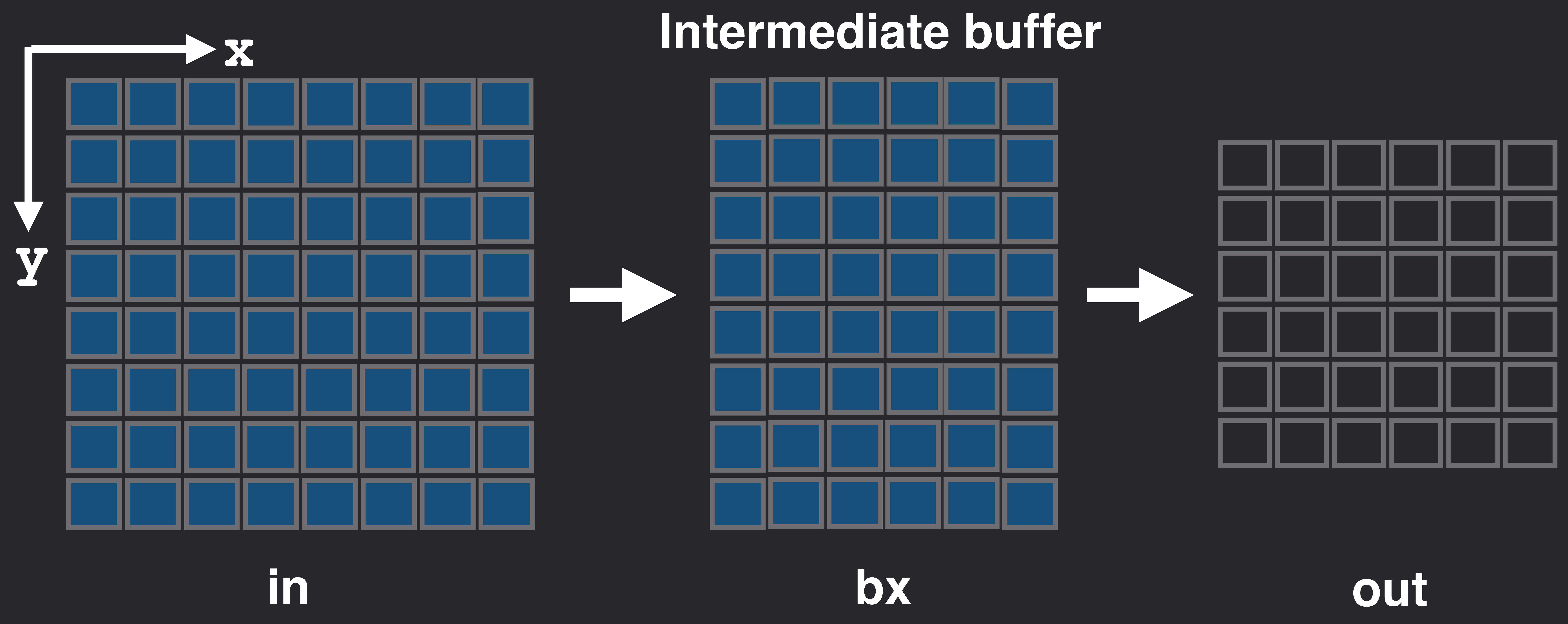
A basic (slow) schedule

compute all pixels of bx, in parallel
compute all pixels of by, in parallel



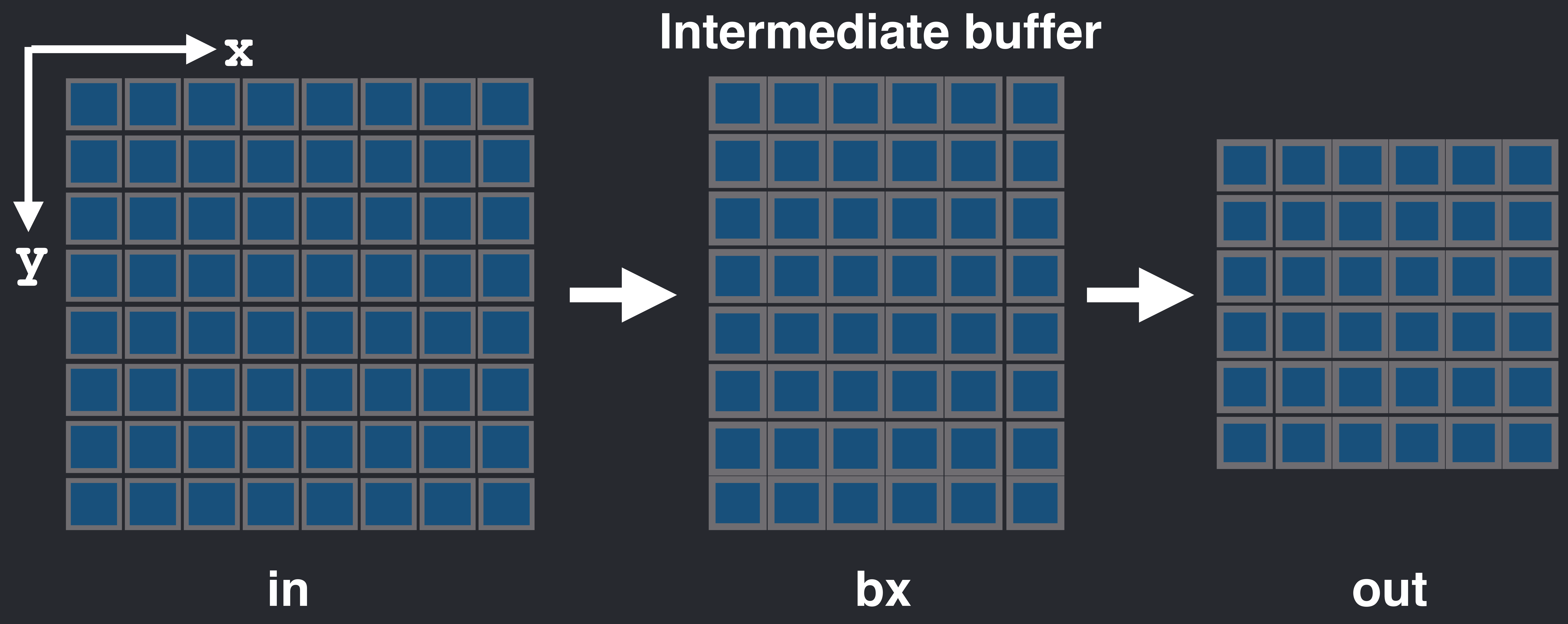
A basic (slow) schedule

compute all pixels of bx, in parallel
compute all pixels of by, in parallel



A basic (slow) schedule

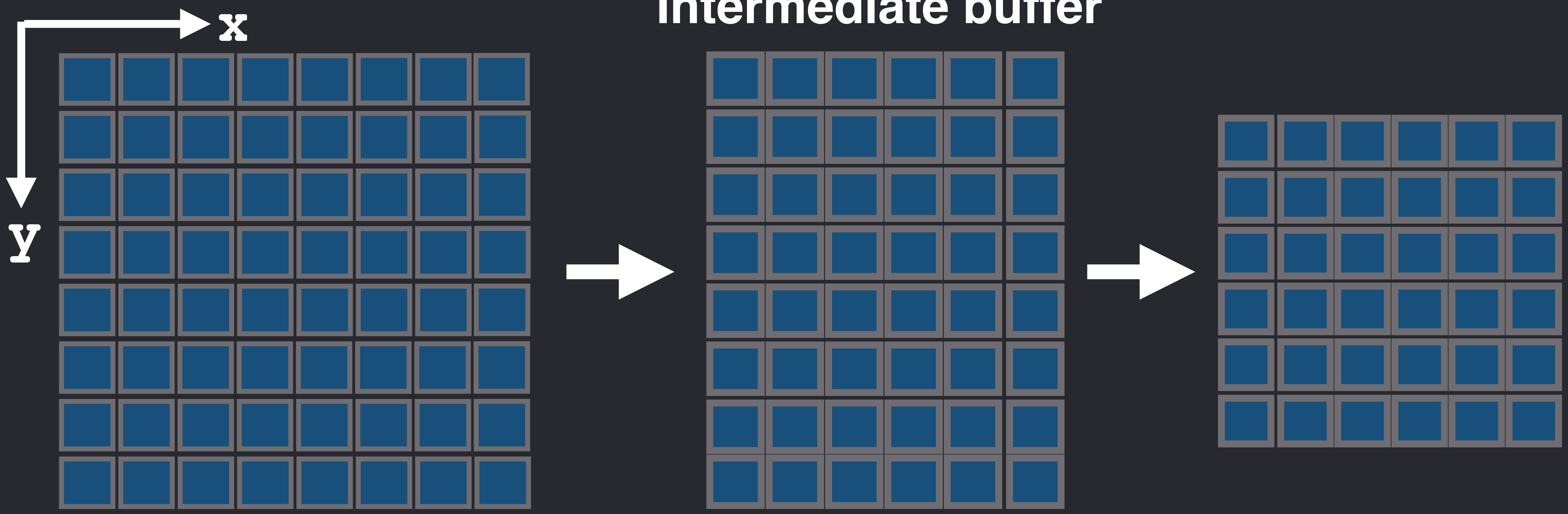
compute all pixels of bx, in parallel
compute all pixels of by, in parallel



A basic (slow) schedule

compute all pixels of bx, in parallel
compute all pixels of by, in parallel

Intermediate buffer

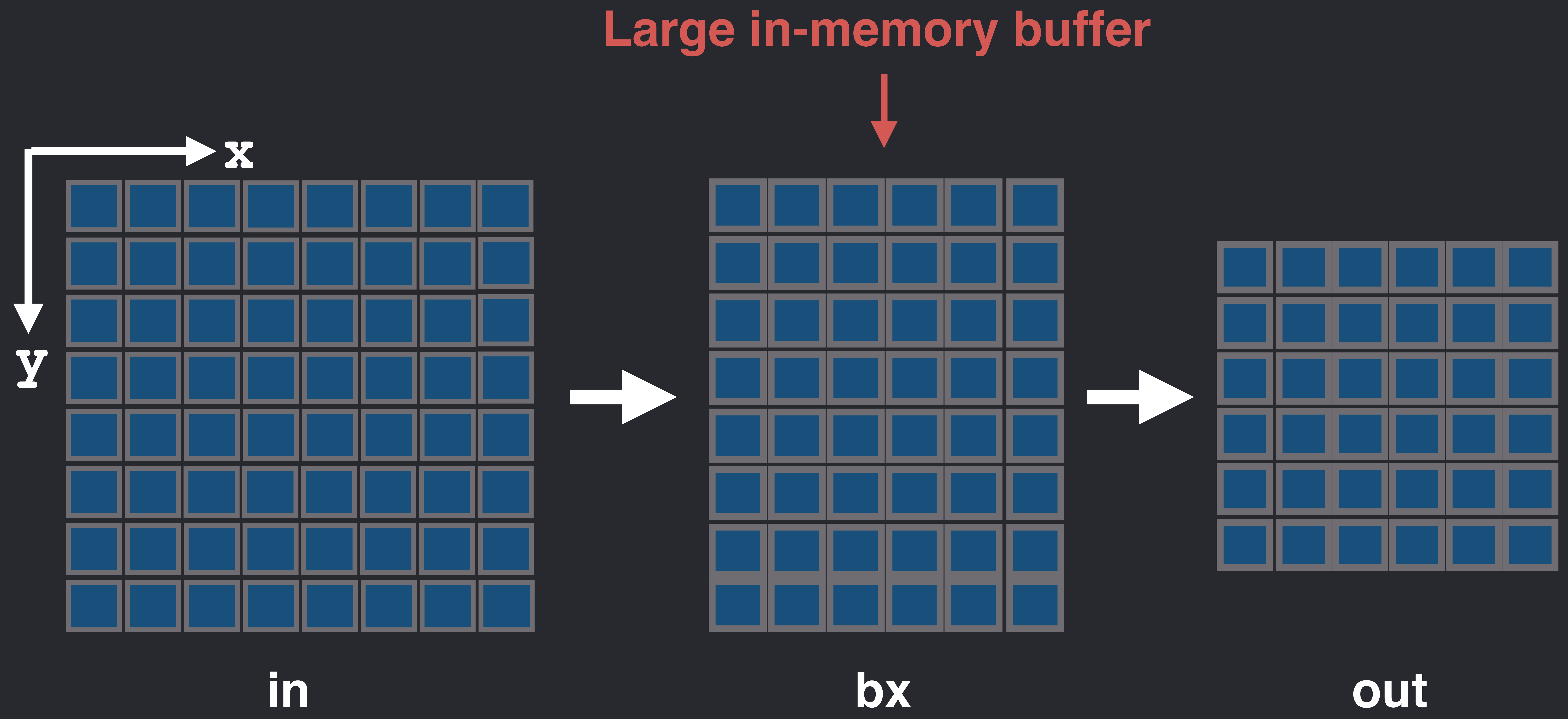


in

bx

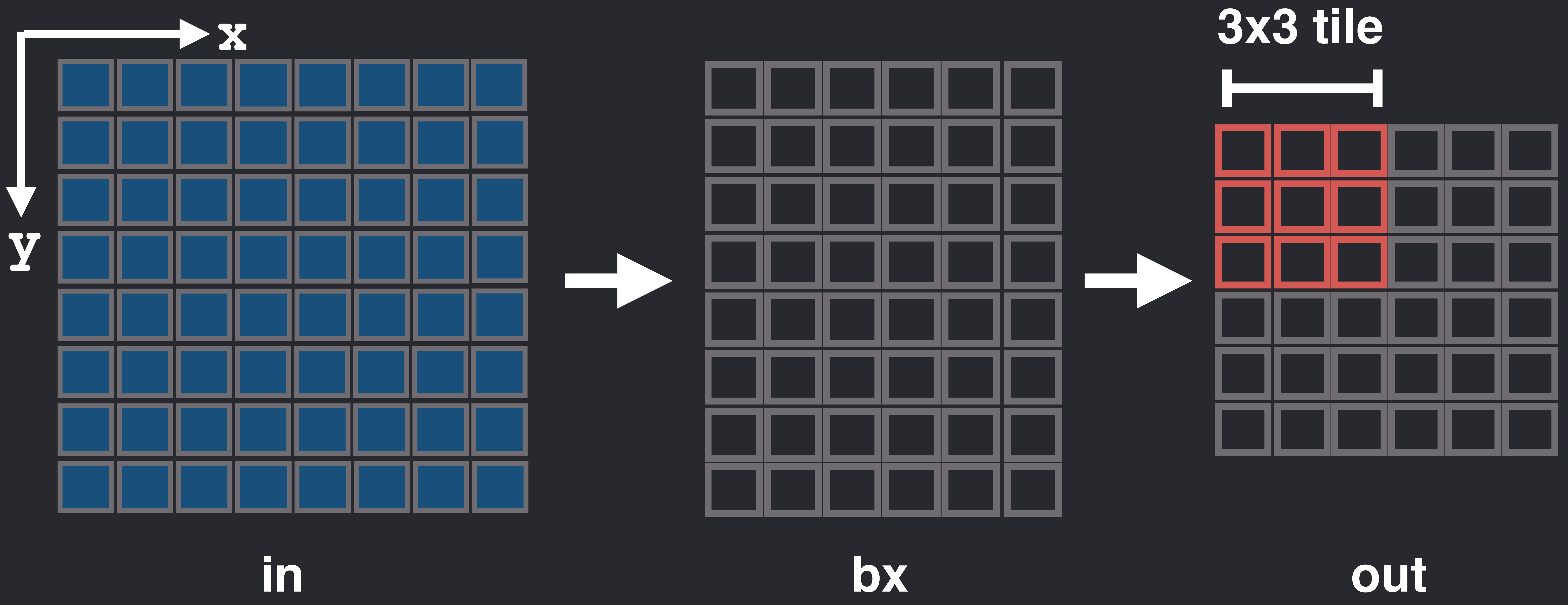
out

Low performance: bandwidth bound



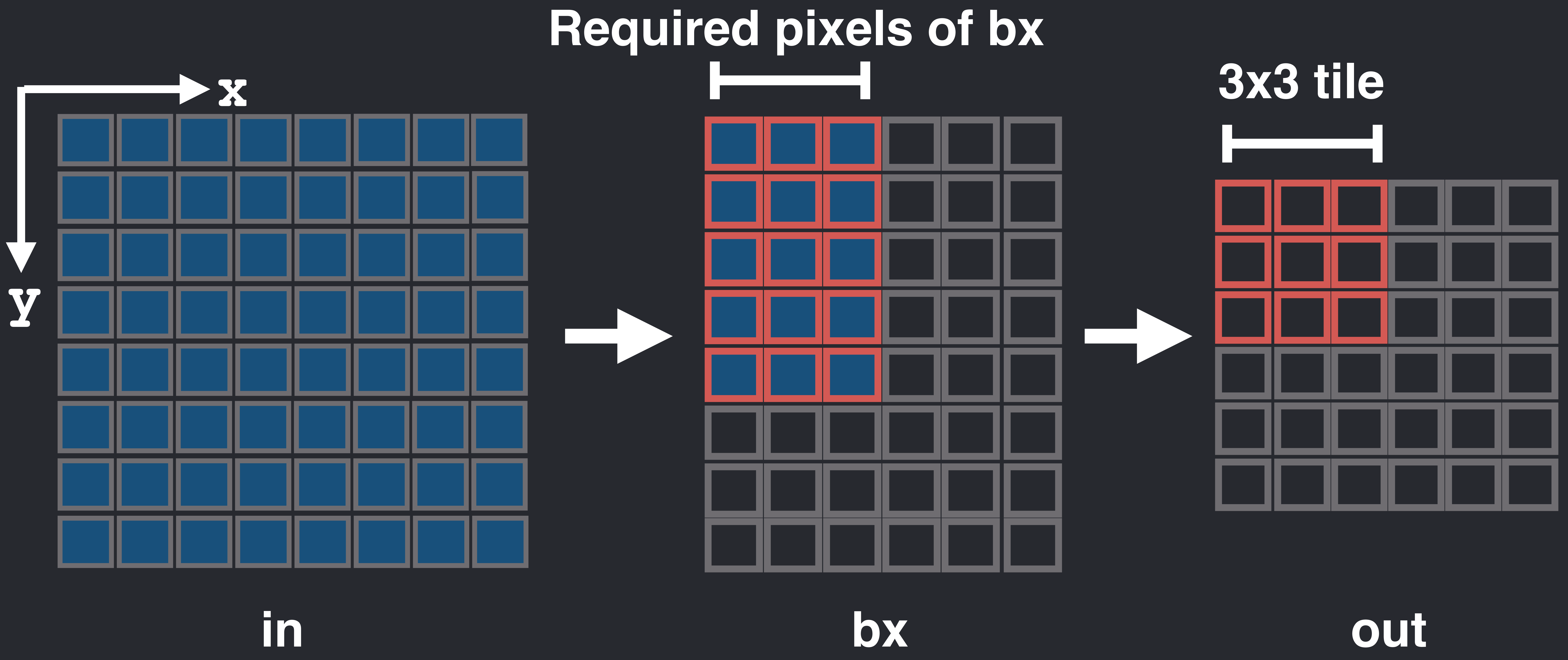
Tiling to improve data locality

for each 3x3 tile, in parallel
    compute required pixels of bx
    compute pixels of out in tile



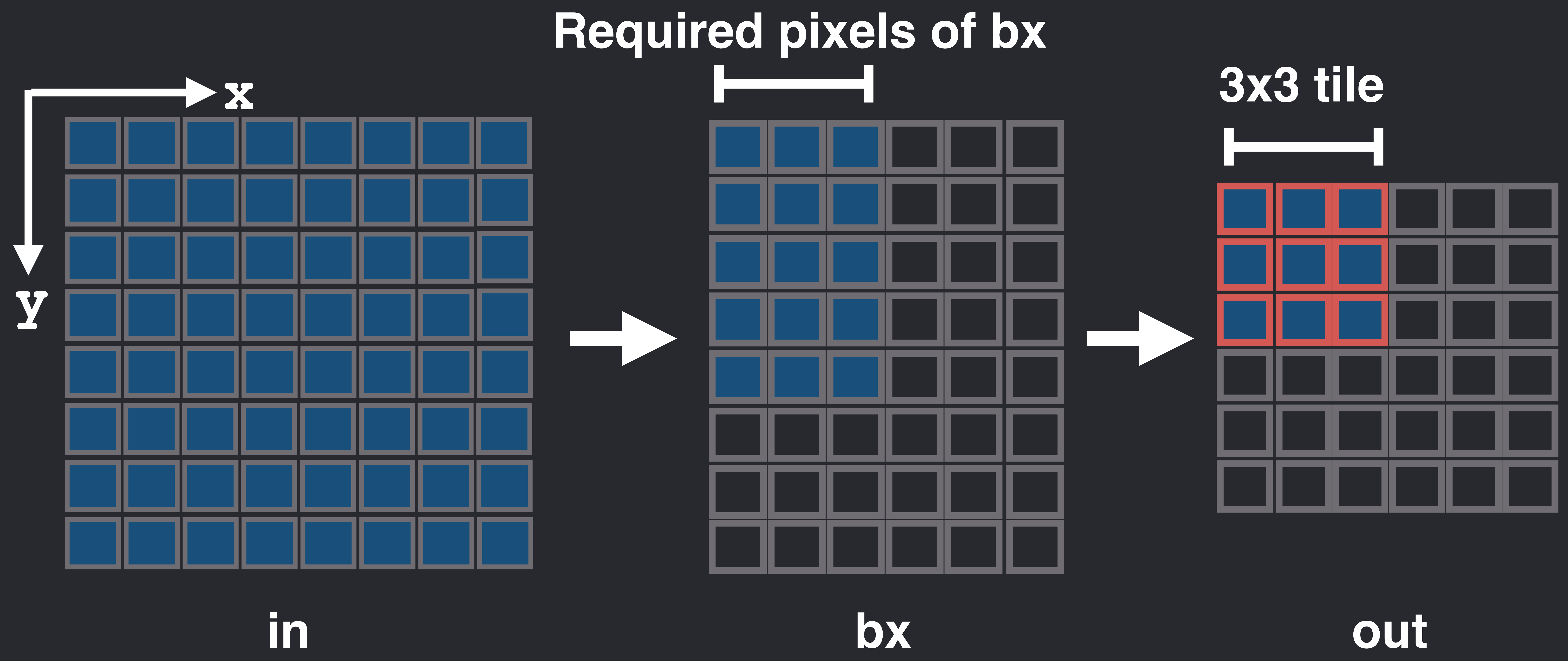
Tiling to improve data locality

for each 3x3 tile, in parallel
    compute required pixels of bx
    compute pixels of out in tile



Tiling to improve data locality

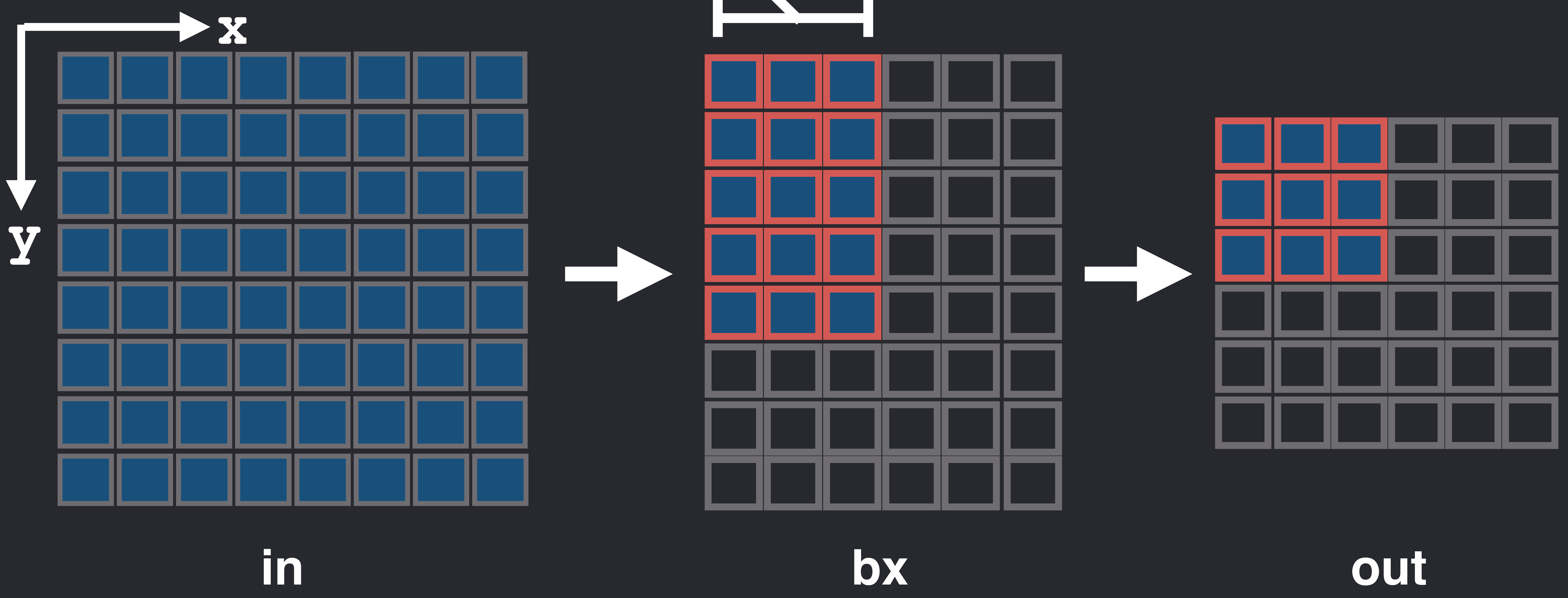
for each 3x3 tile, in parallel
    compute required pixels of bx
    compute pixels of out in tile



Tiling to improve data locality

for each 3x3 tile, in parallel
    compute required pixels of bx
    compute pixels of out in tile

Intermediate buffer:
fits in fast on-chip storage

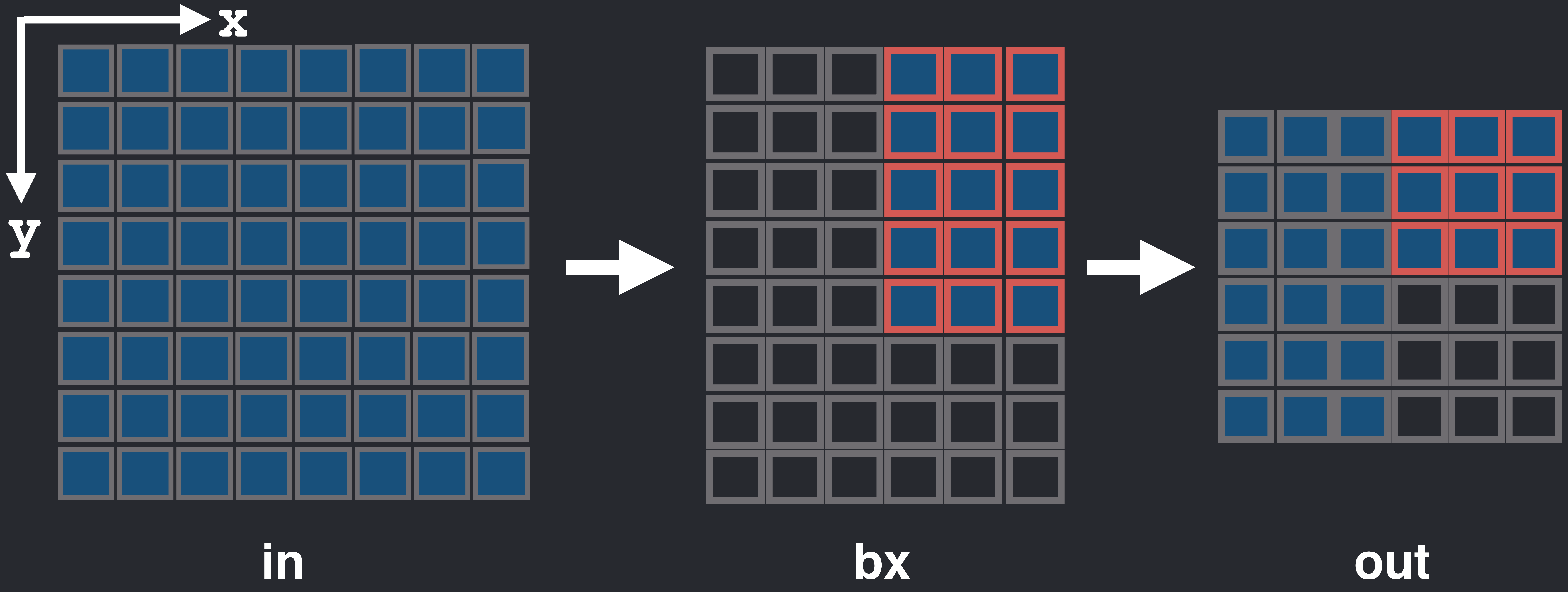


Tiling to improve data locality

for each 3x3 tile, in parallel
    compute required pixels of bx
    compute pixels of out in tile

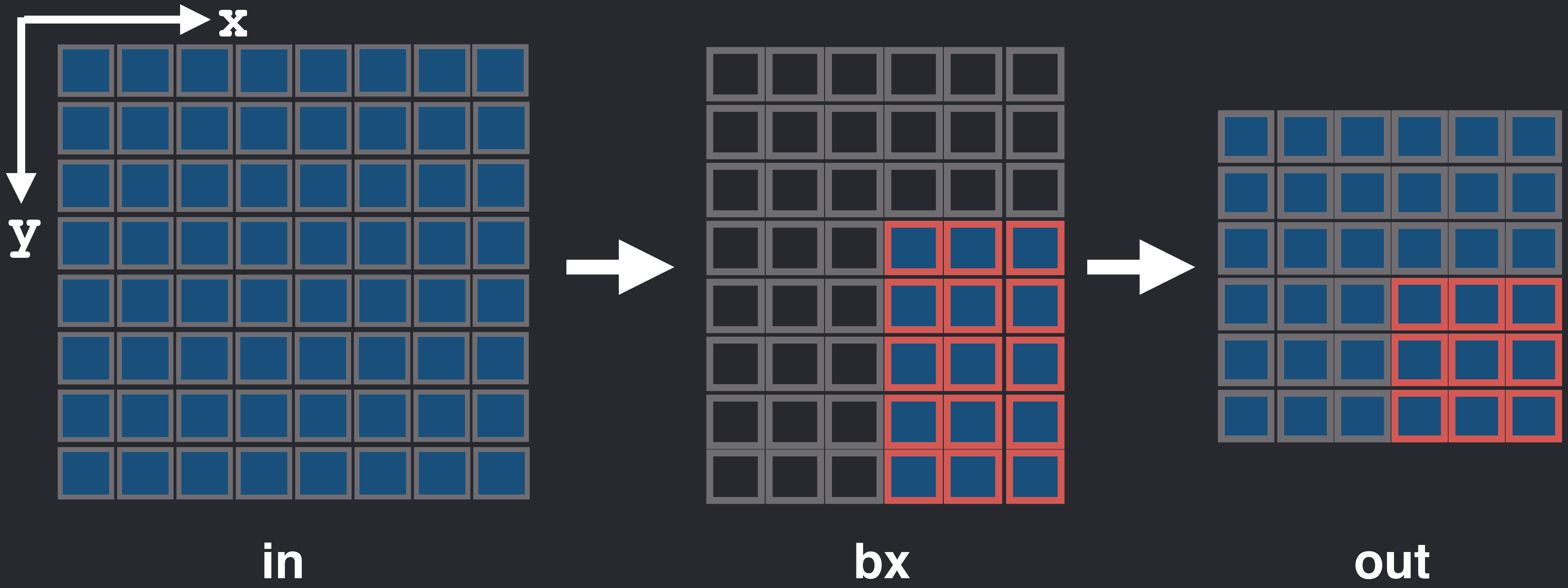
Tiling to improve data locality

for each 3x3 tile, in parallel
    compute required pixels of bx
    compute pixels of out in tile



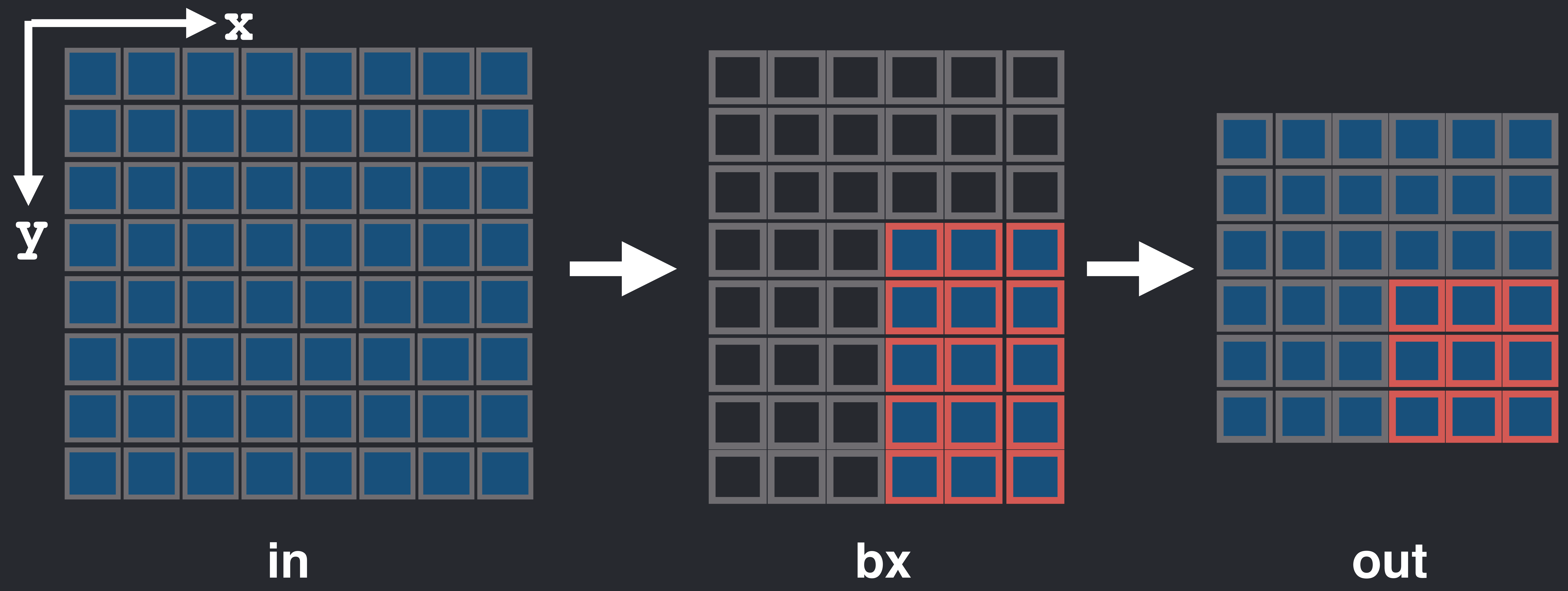
Tiling to improve data locality

```
for each 3x3 tile, in parallel
    compute required pixels of bx
    compute pixels of out in tile
```

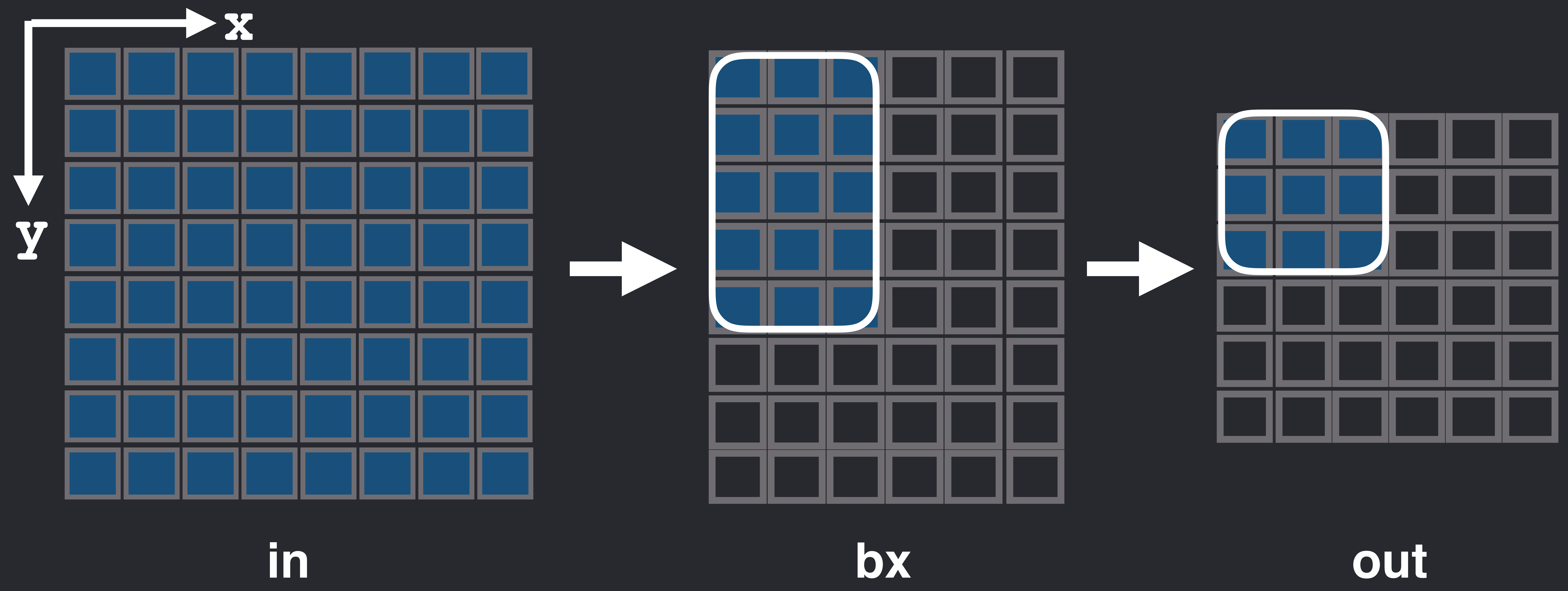


Tiling to improve data locality

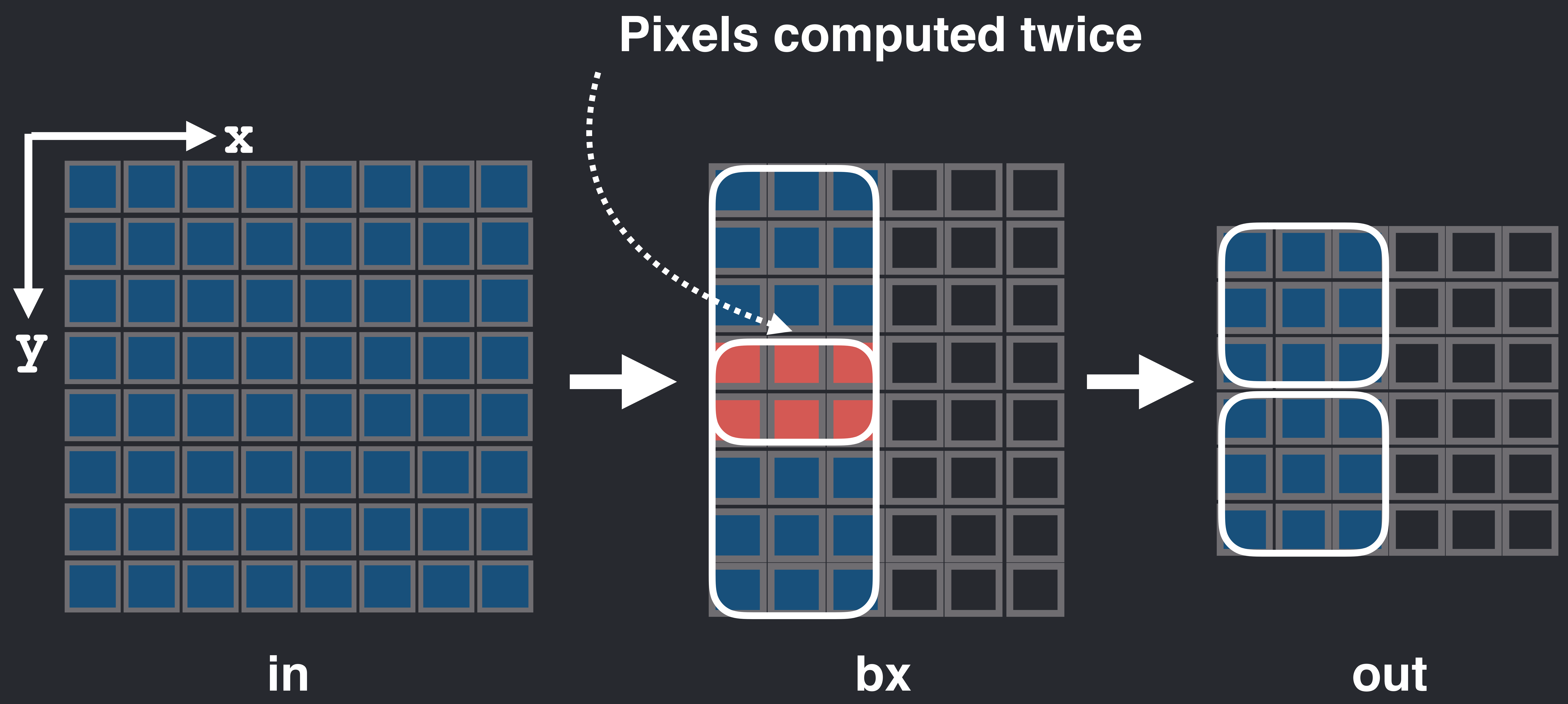
for each 3x3 tile, in parallel
    compute required pixels of bx
    compute pixels of out in tile



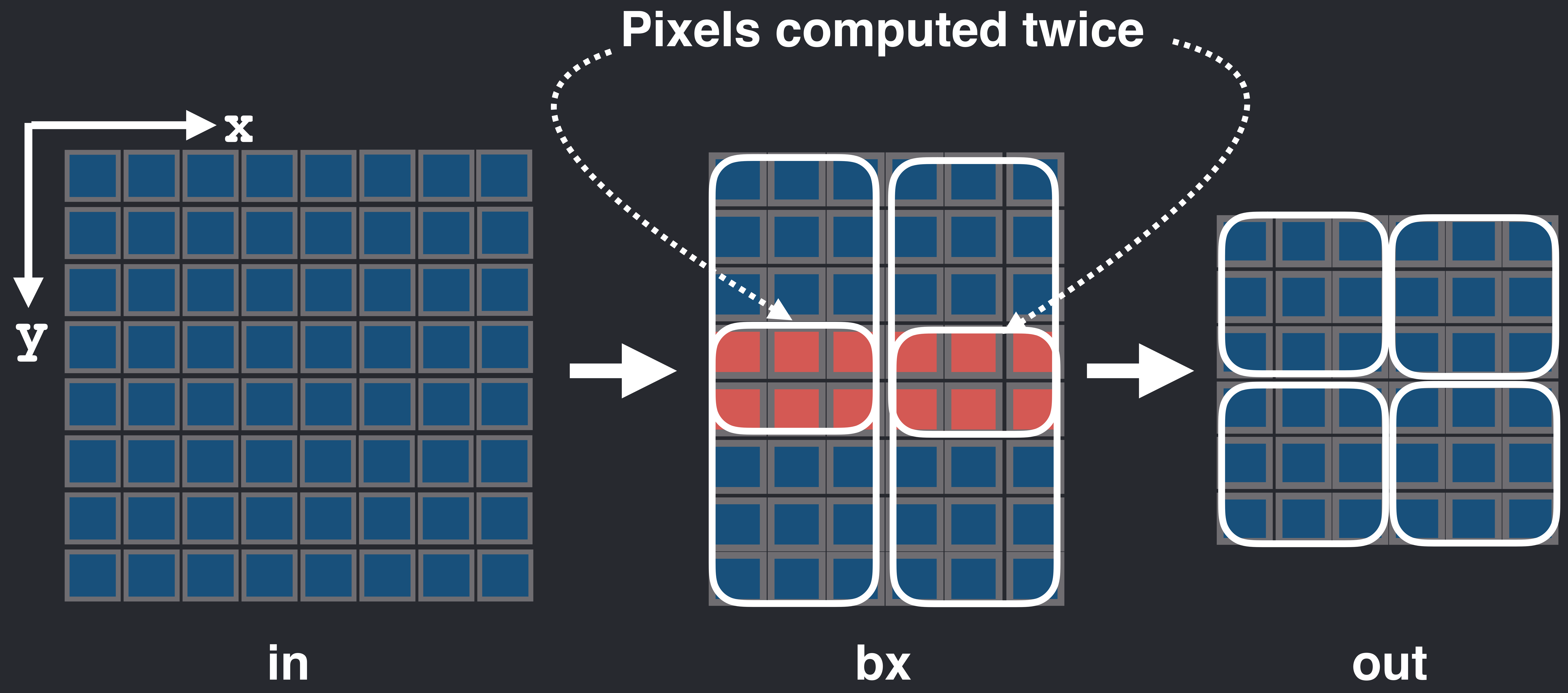
Tiling introduces redundant work



Tiling introduces redundant work

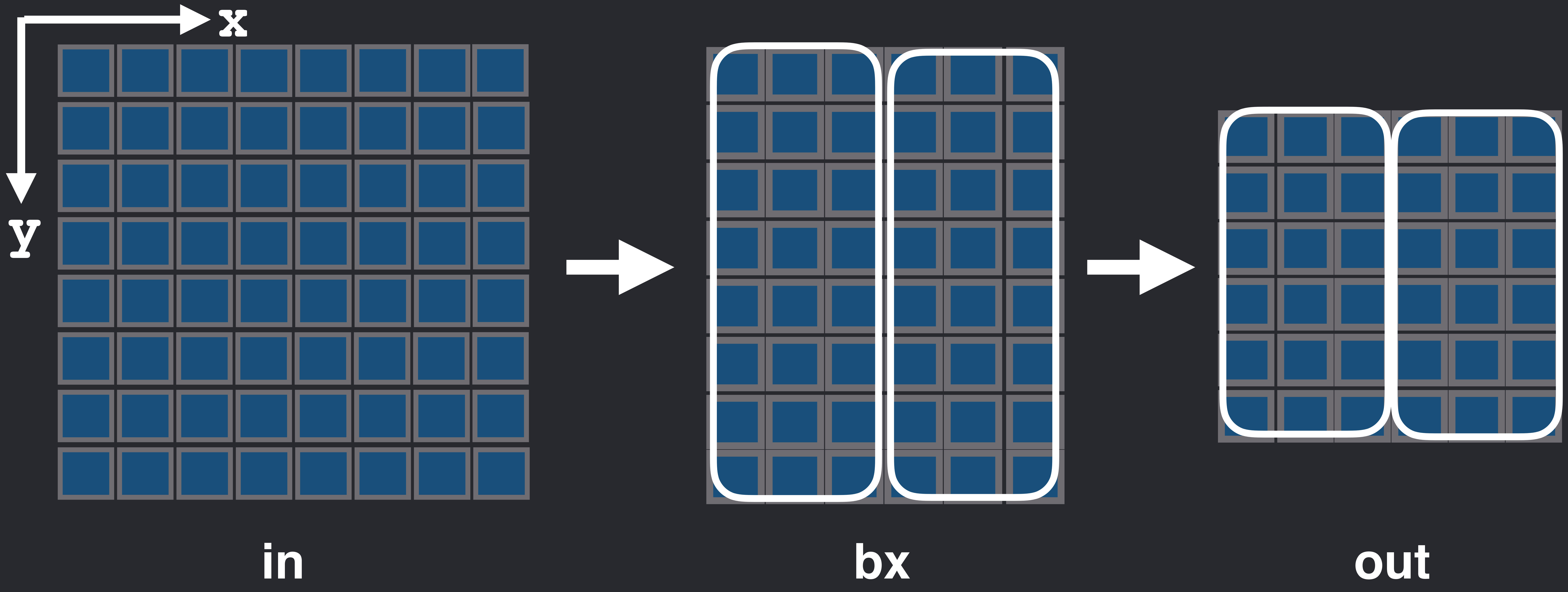


Tiling introduces redundant work



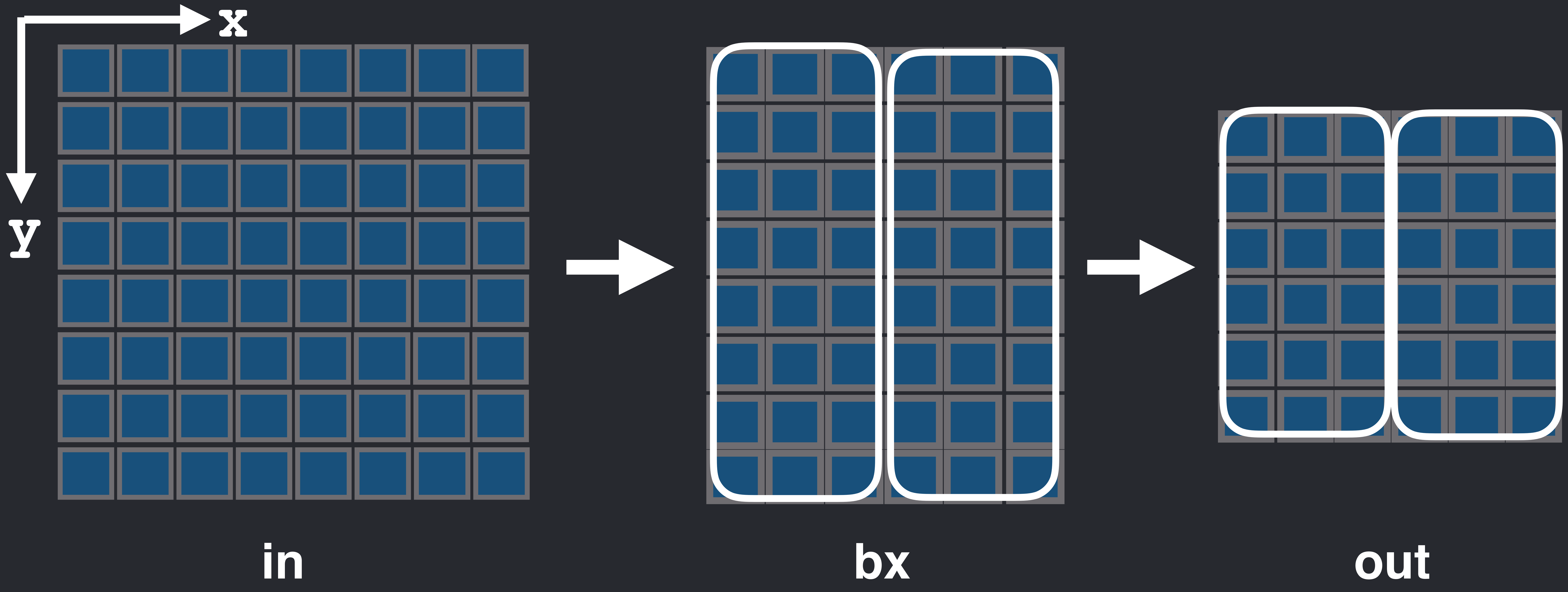
Larger tiles reduce redundant work

```
for each 3x6 tile, in parallel
    compute required pixels of bx
    compute pixels in tile of out
```



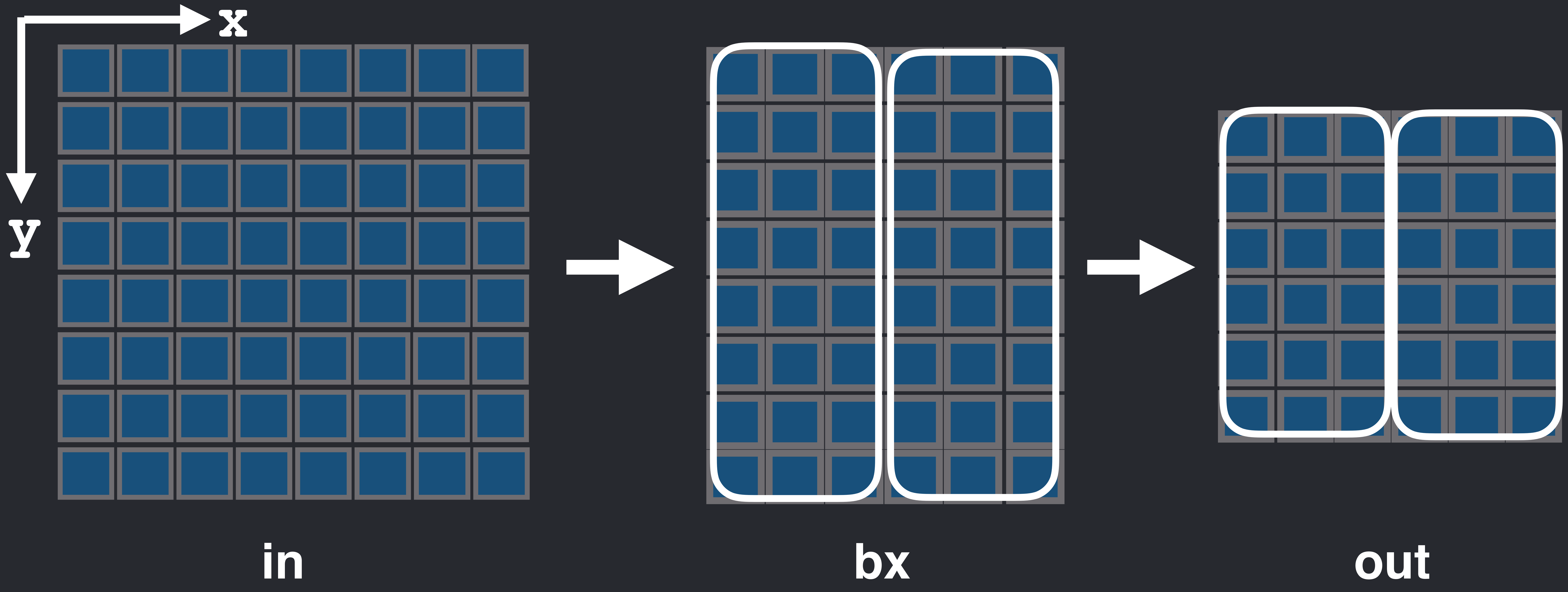
Goal: balance parallelism, locality, work

for each 3x6 tile, in parallel
    compute required pixels of bx
    compute pixels in tile of out



Goal: balance parallelism, locality, work

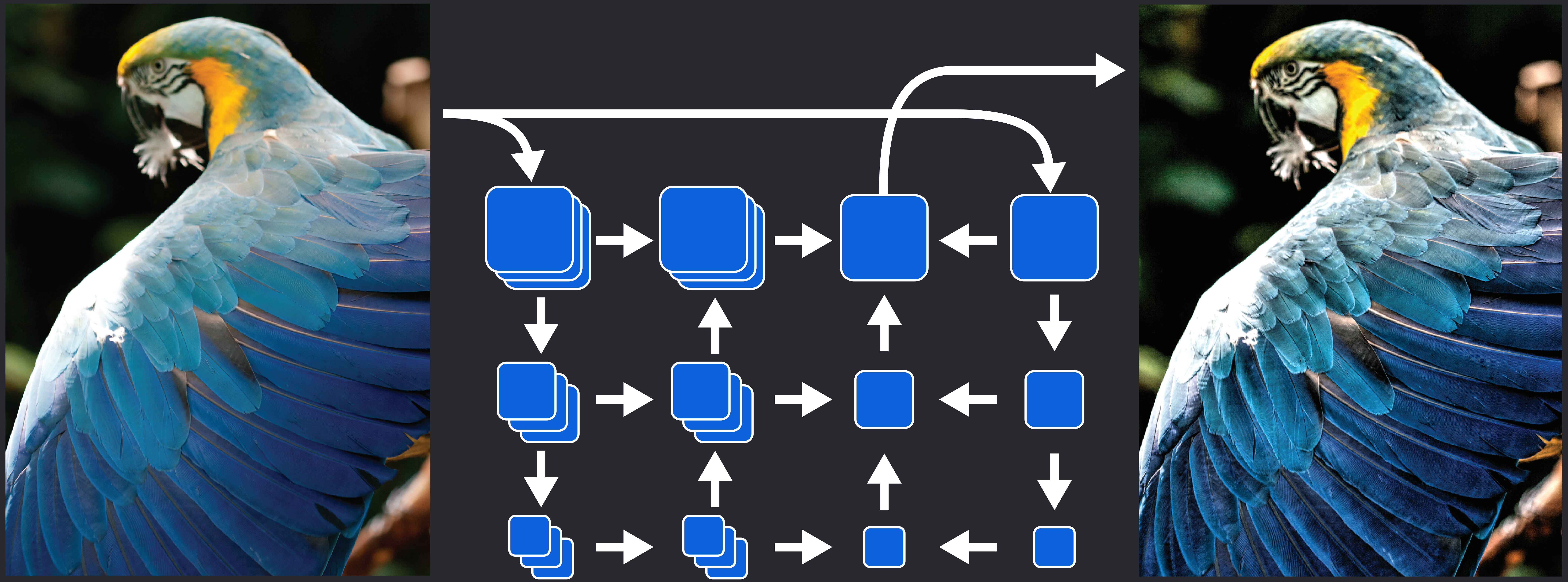
for each 3x6 tile, in parallel
    compute required pixels of bx
    compute pixels in tile of out



Represent image processing pipelines as graphs

DAG representation of the two-stage blur pipeline

Real world pipelines are complex graphs



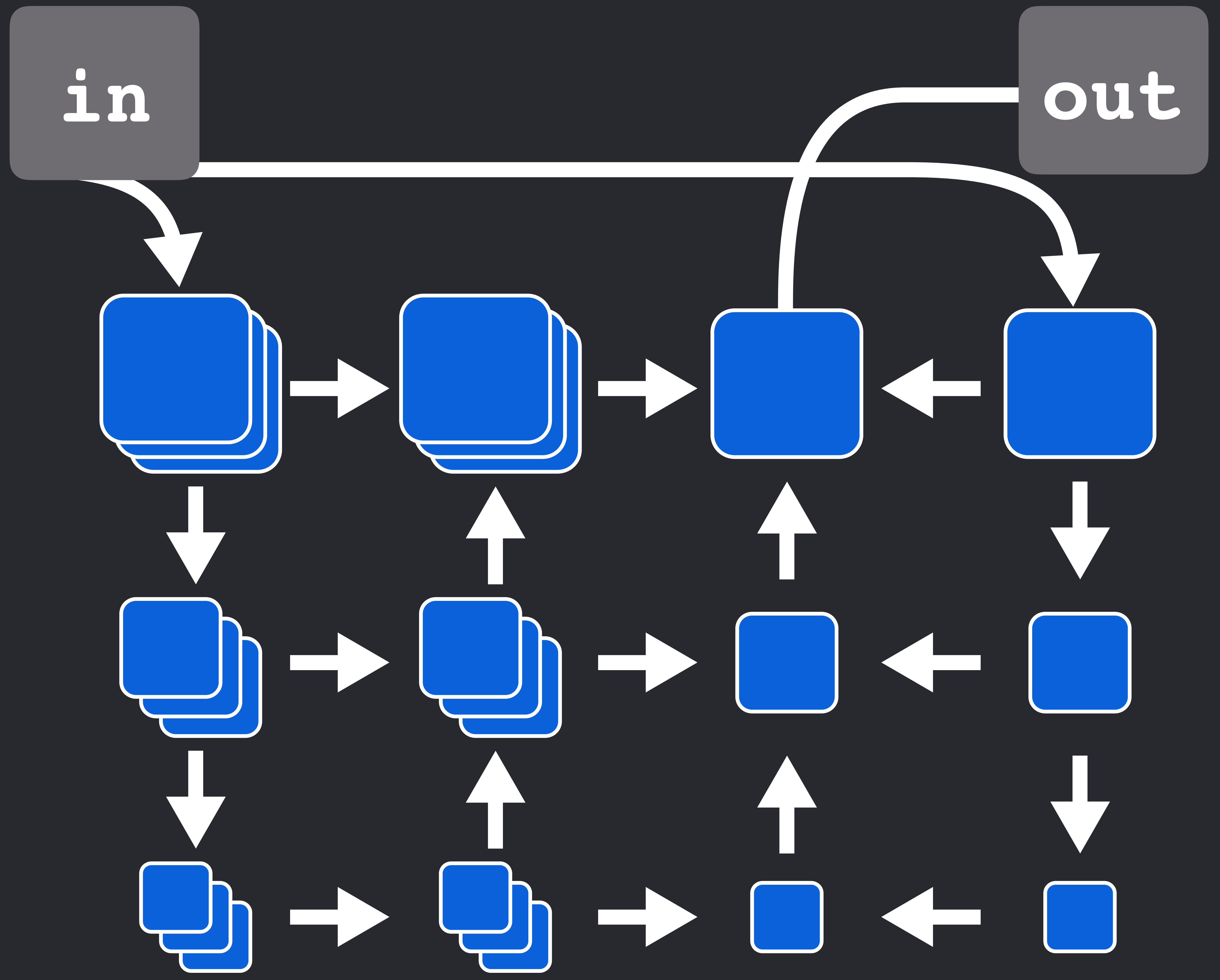
Local Laplacian filters

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

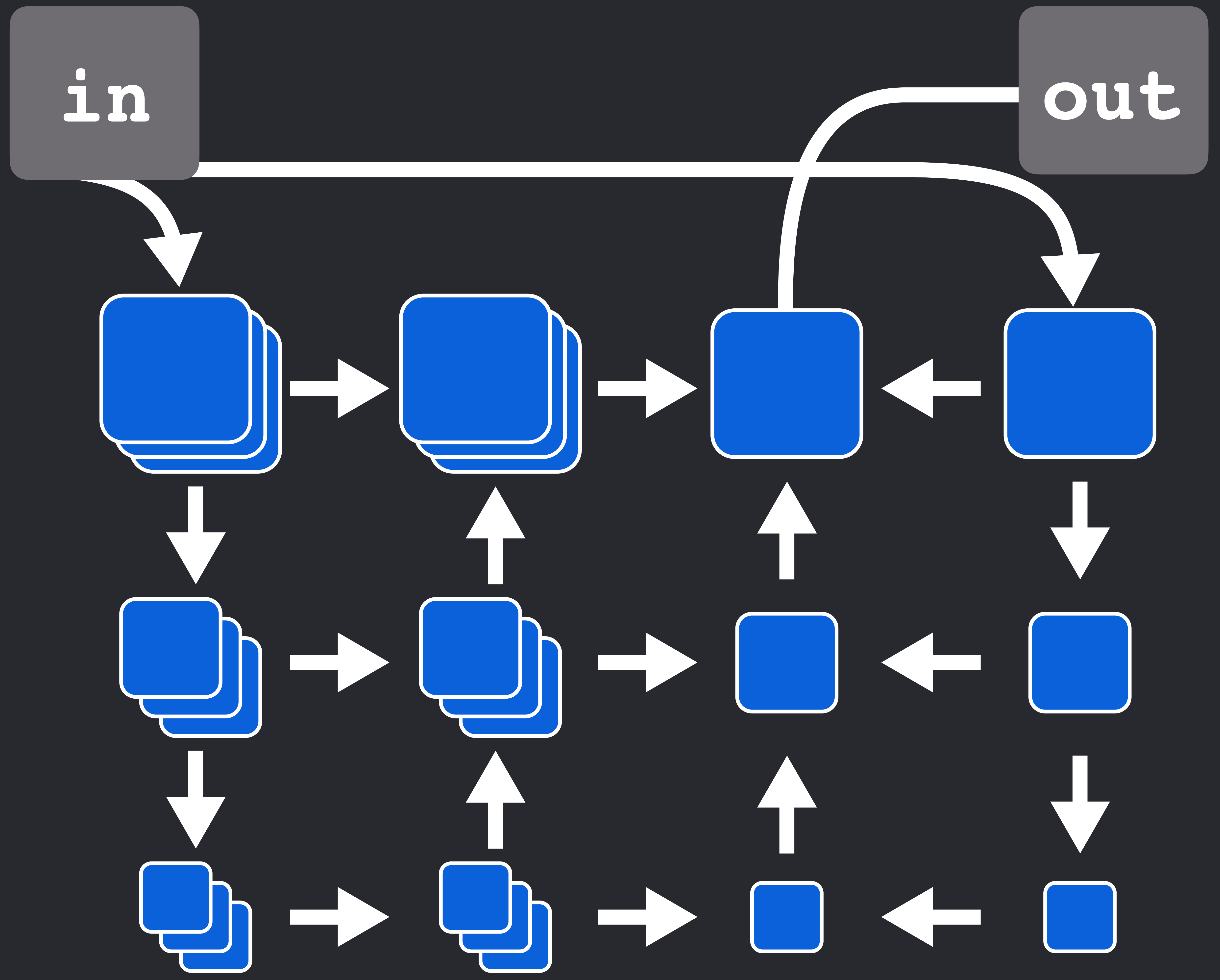
100 stages

Google Nexus HDR+ mode: over 2000 stages!

Key aspects of scheduling

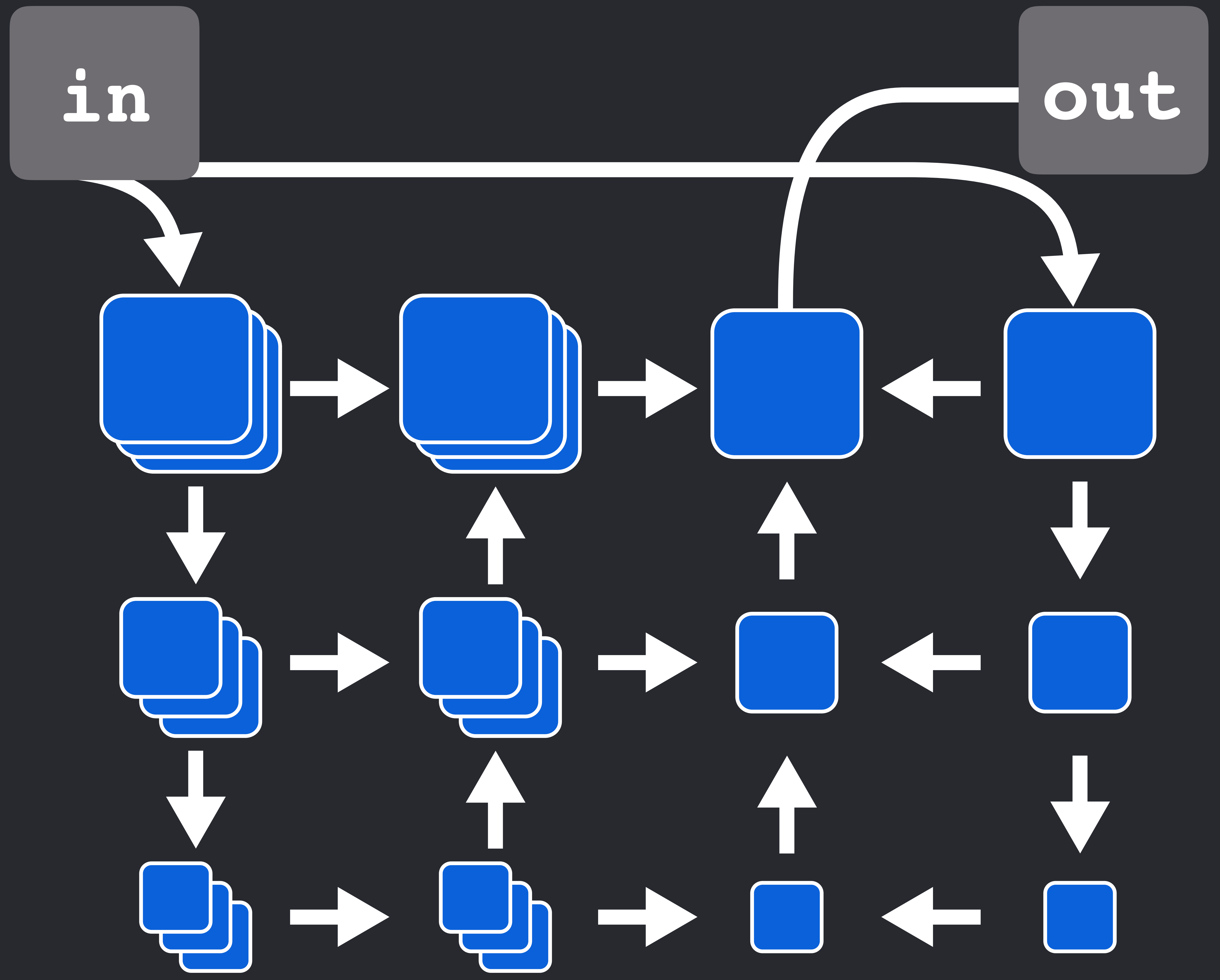


Key aspects of scheduling



Deciding which stages to interleave for better data locality

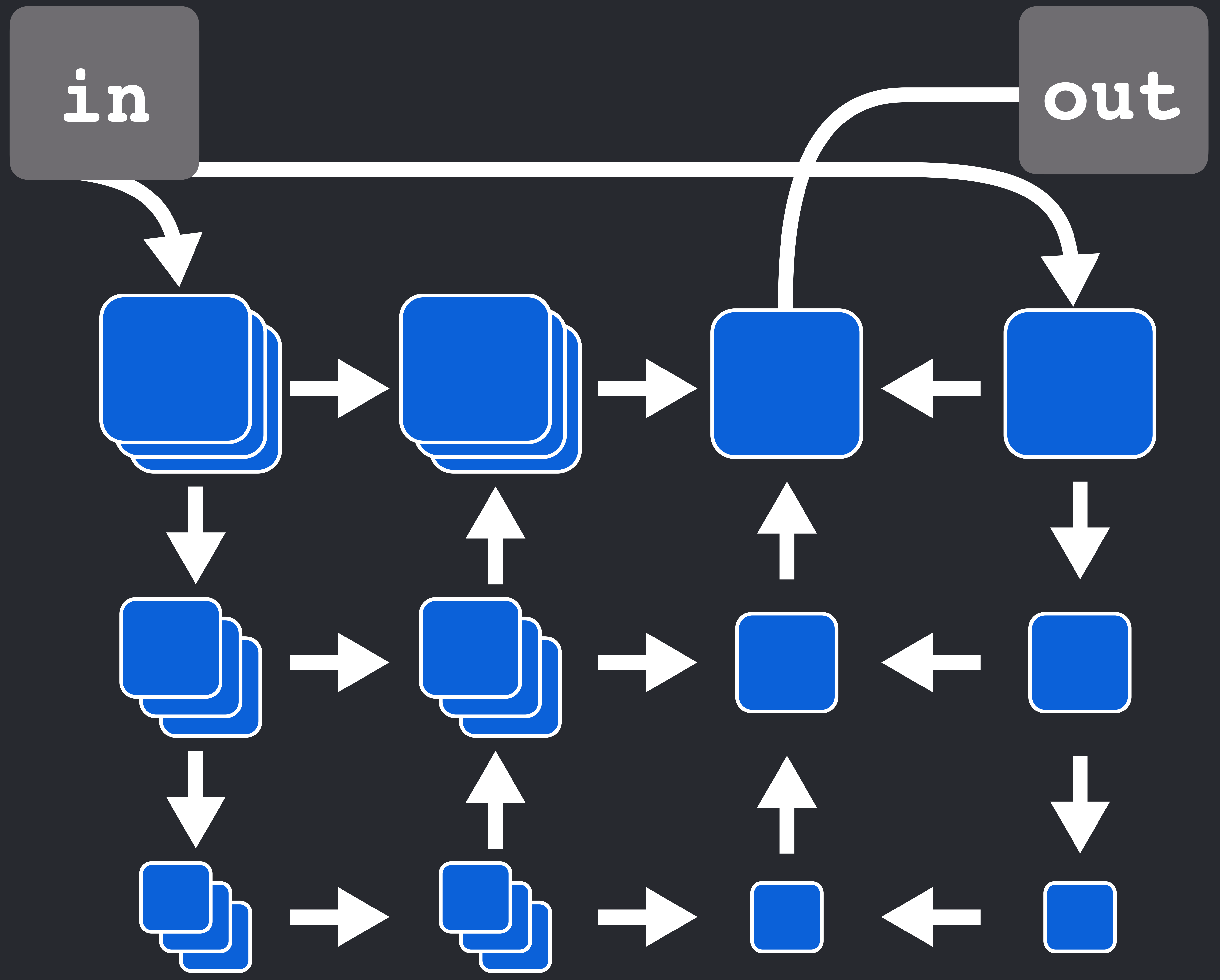
Key aspects of scheduling



Deciding which stages to interleave for better data locality

Picking tiles sizes to trade-off locality and re-computation

Key aspects of scheduling



Deciding which stages to interleave for better data locality

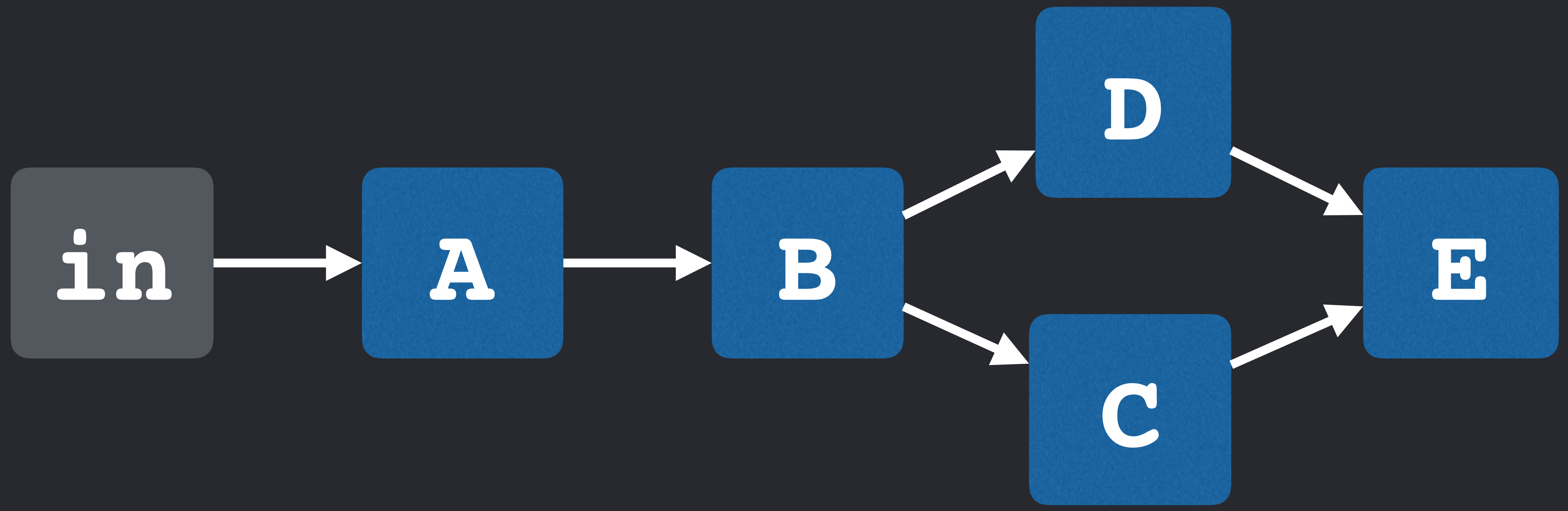
Picking tiles sizes to trade-off locality and re-computation

Maintain ability to execute in parallel

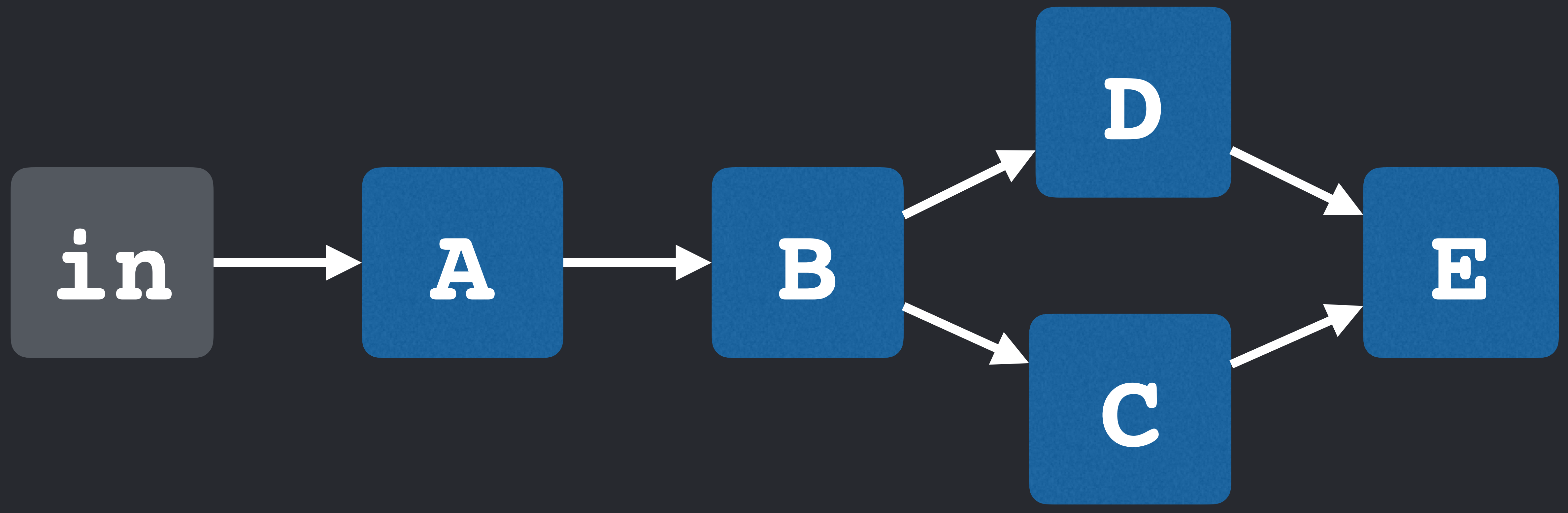
An Algorithm for Scheduling Image Processing Pipelines

Algorithm

Input: DAG of pipeline stages



Algorithm



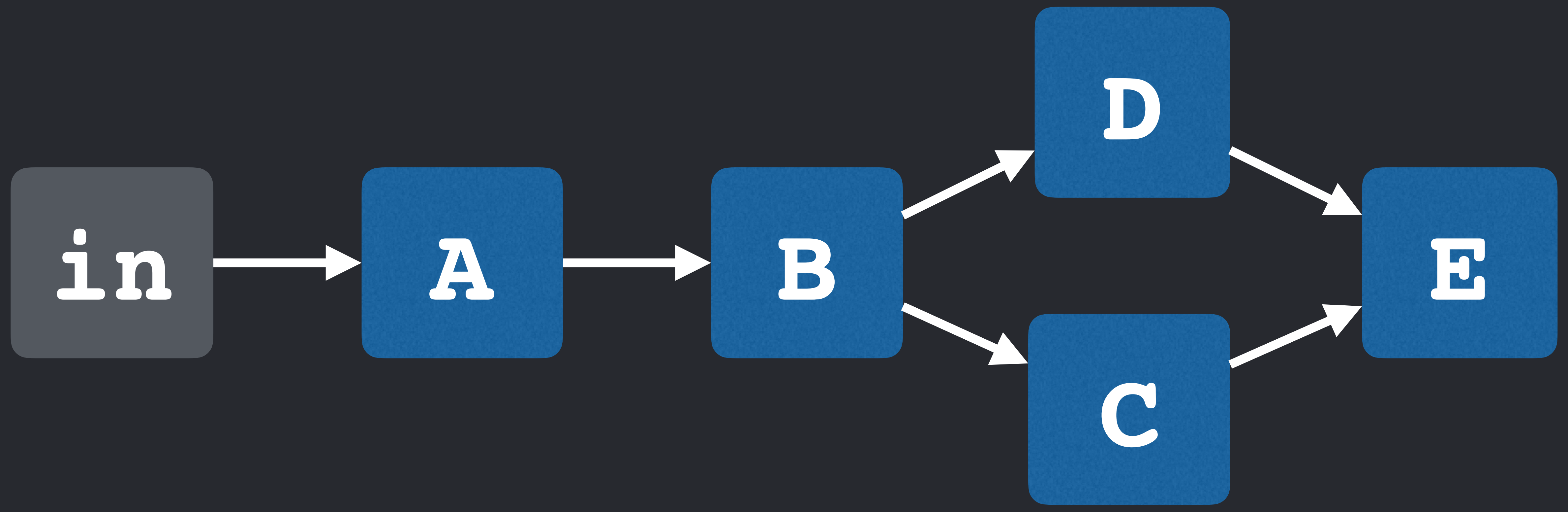
Input: DAG of pipeline stages

Output: Optimized schedule

for each 8x128 tile in parallel
    compute required pixels of A
    compute pixels in tile of B

for each 8x8 tile in parallel
    compute required pixels of C
    compute required pixels of D
    compute pixels in tile of E

Algorithm



Input: DAG of pipeline stages

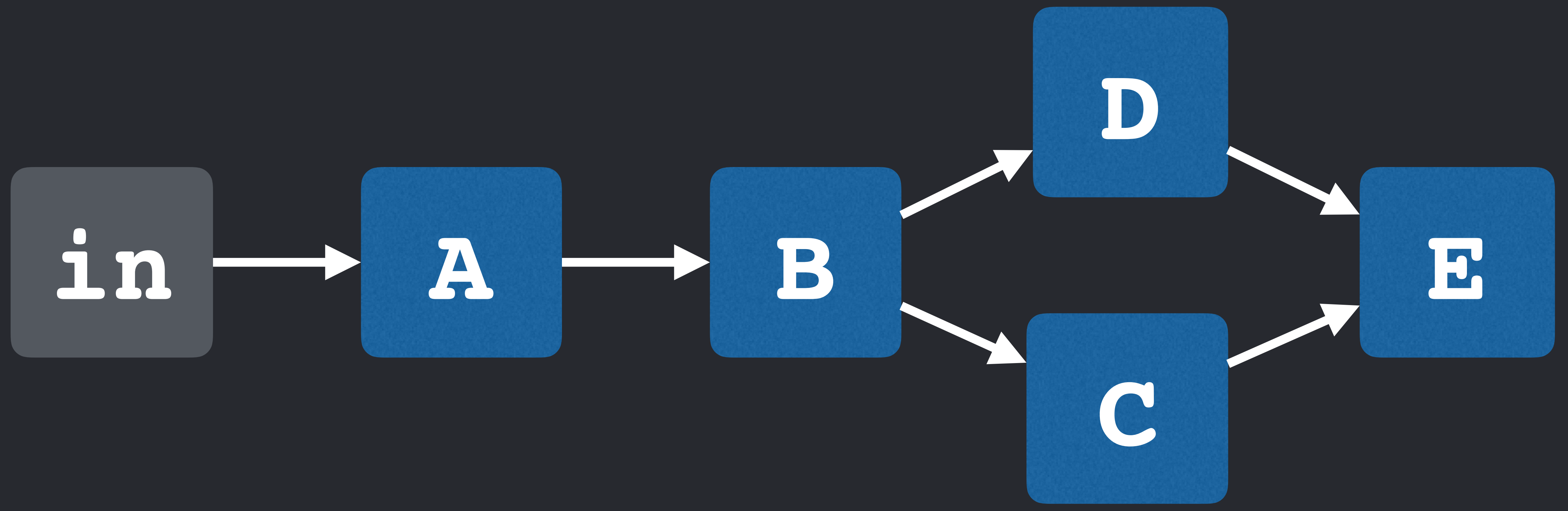
Output: Optimized schedule

```
for each 8x128 tile in parallel
    compute required pixels of A
    compute pixels in tile of B

for each 8x8 tile in parallel
    compute required pixels of C
    compute required pixels of D
    compute pixels in tile of E
```

Algorithm

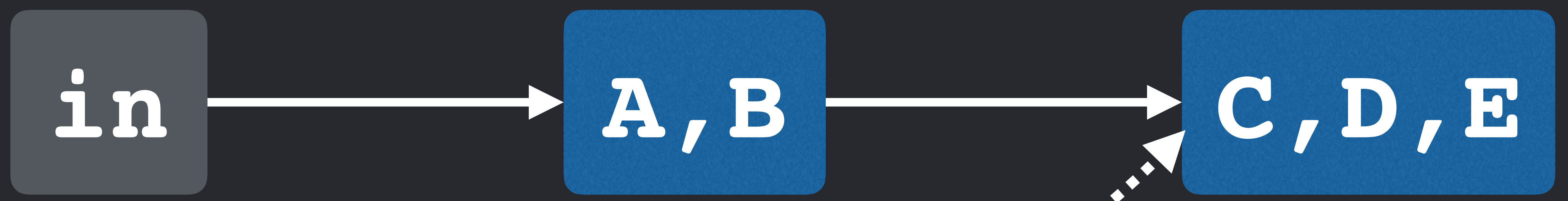
Input: DAG of pipeline stages



Output: Optimized schedule

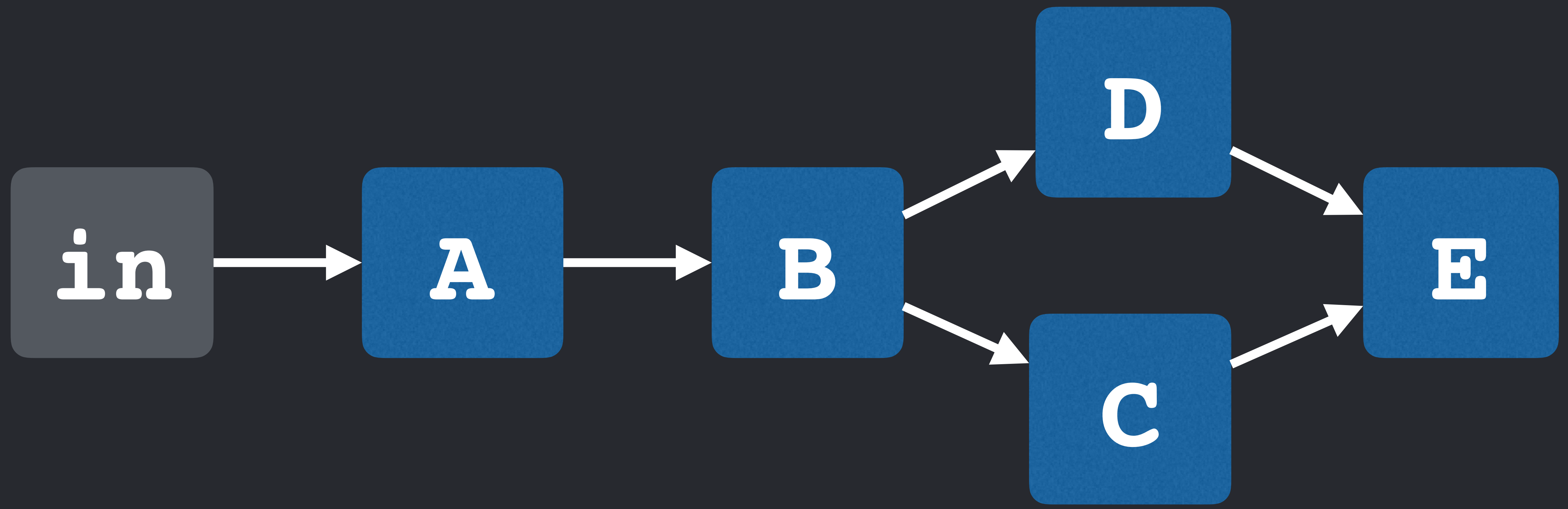
for each 8x128 tile in parallel
    compute required pixels of A
    compute pixels in tile of B

for each 8x8 tile in parallel
    compute required pixels of C
    compute required pixels of D
    compute pixels in tile of E



Algorithm

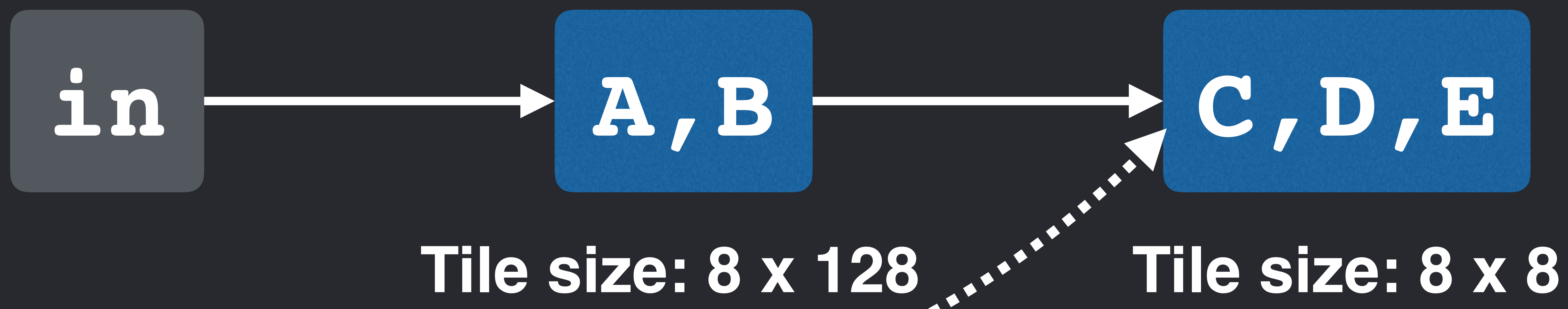
Input: DAG of pipeline stages



Output: Optimized schedule

```
for each 8x128 tile in parallel
    compute required pixels of A
    compute pixels in tile of B

for each 8x8 tile in parallel
    compute required pixels of C
    compute required pixels of D
    compute pixels in tile of E
```

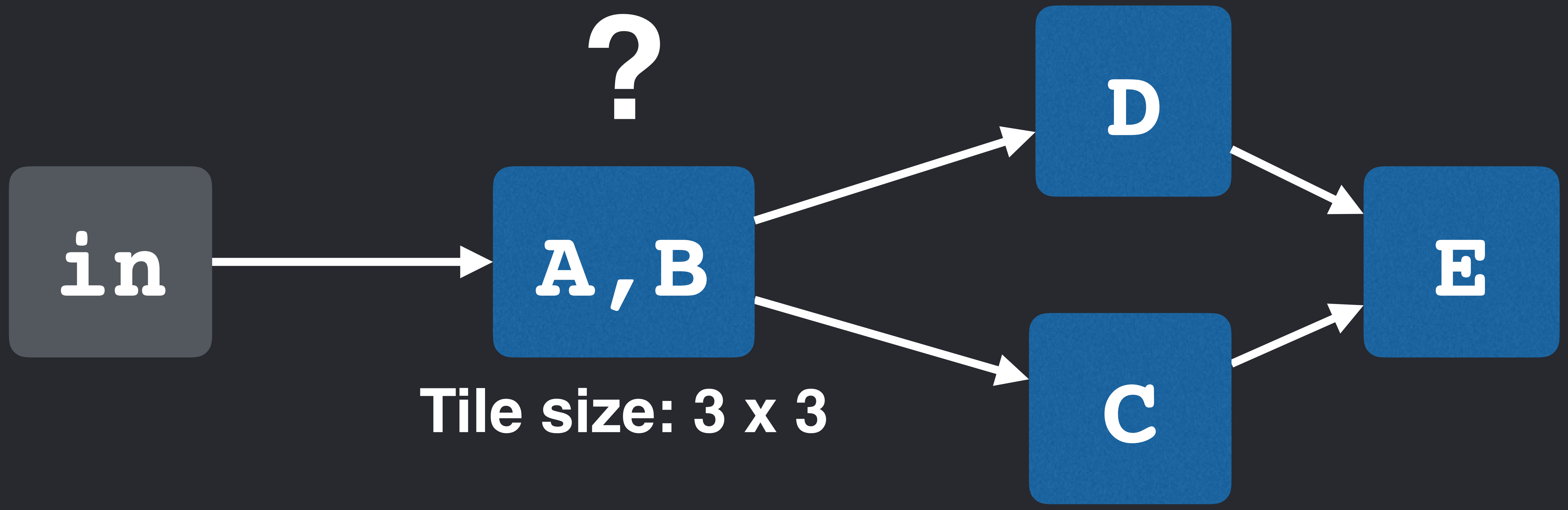


Scheduling the DAG for better locality

Determine which stages to group together?

How to tile stages in each group?

When to group stages?

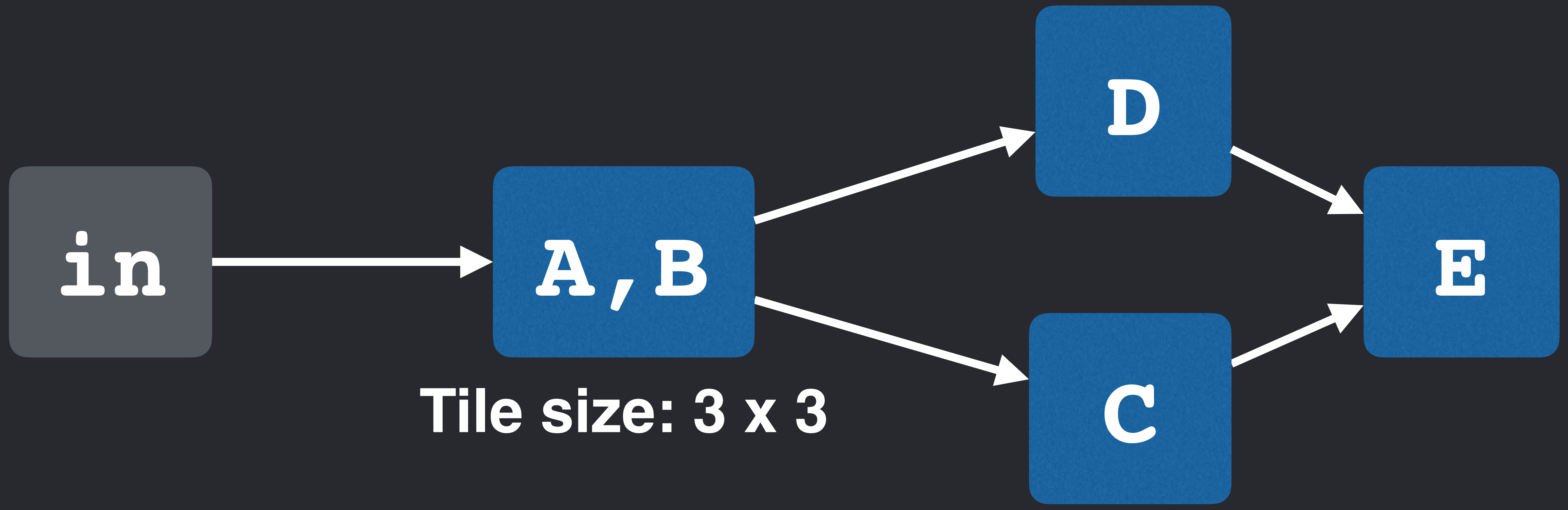



```
for each 3x3 tile in parallel
    compute required pixels of A
    compute pixels in tile of B

compute all pixels of C, in parallel
compute all pixels of D, in parallel
compute all pixels of E, in parallel
```

Grouping A and B together can either improve or degrade performance

Quantifying the cost of a group



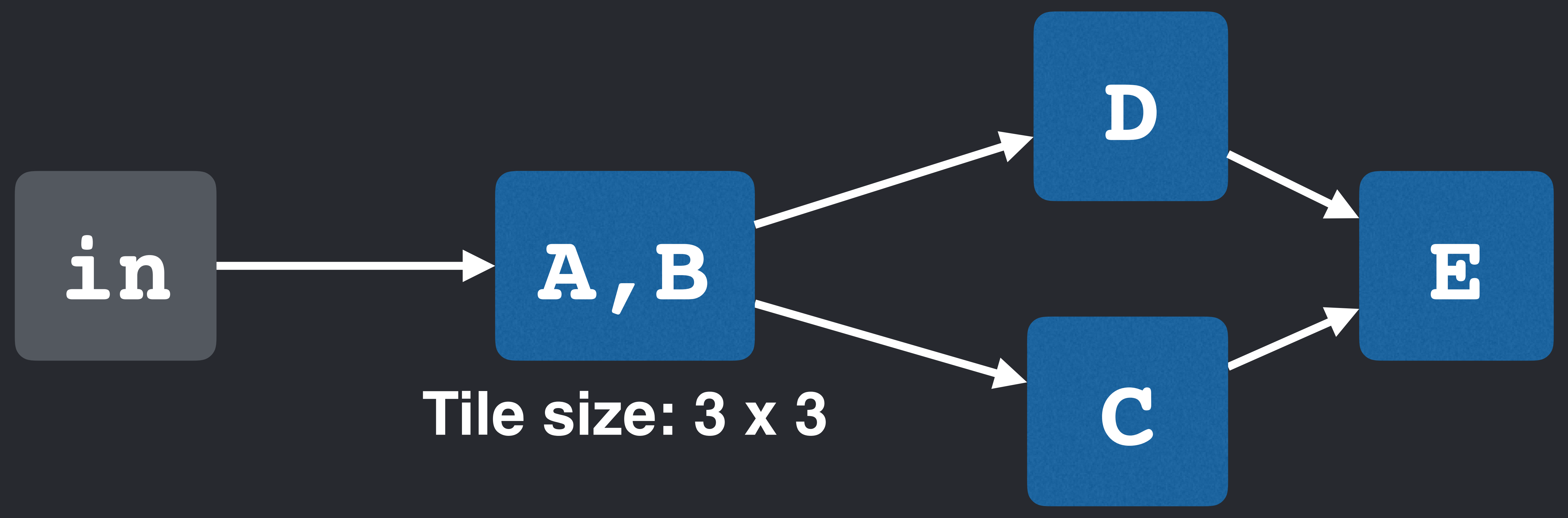
Tile size: 3 x 3

```
for each 3x3 tile in parallel
    compute required pixels of A
    compute pixels in tile of B

compute all pixels of C, in parallel
compute all pixels of D, in parallel
compute all pixels of E, in parallel
```

Cost = Cost of arithmetic + Cost of memory

Quantifying the cost of a group

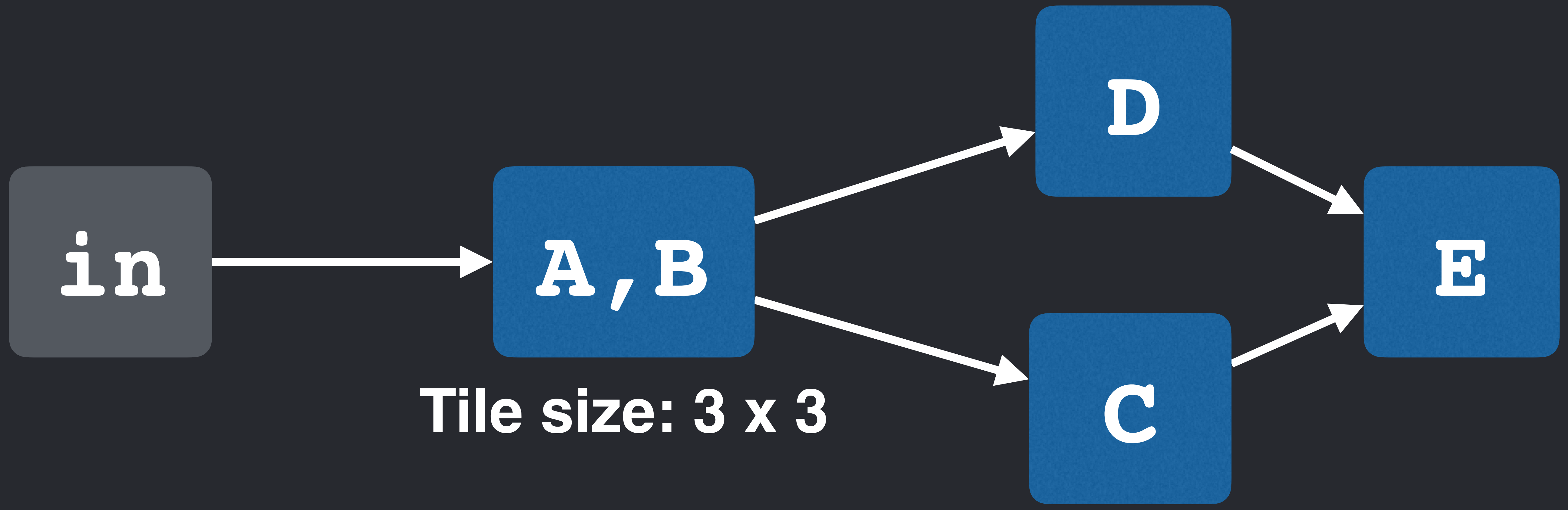


for each 3x3 tile in parallel
    compute required pixels of A
    compute pixels in tile of B

compute all pixels of C, in parallel
compute all pixels of D, in parallel
compute all pixels of E, in parallel

Cost = (Number of arithmetic operations) +
(Number of memory accesses) x (LOAD COST)

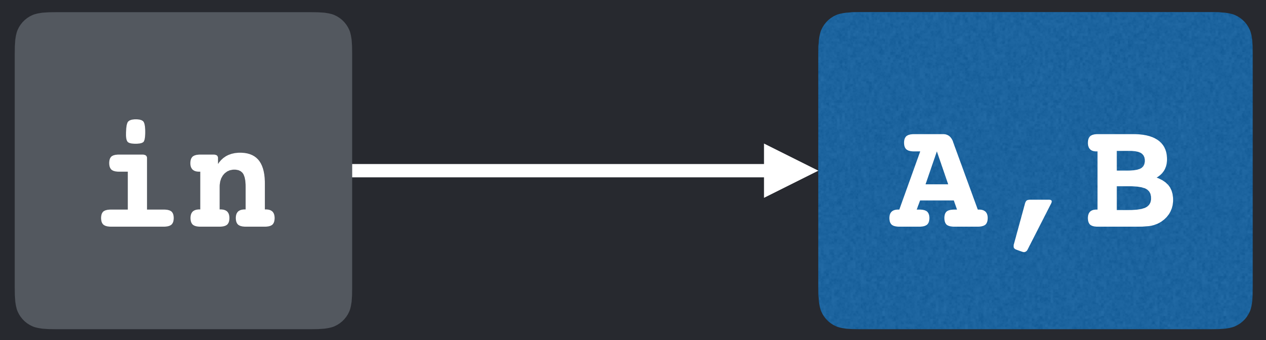
Quantifying the cost of a group



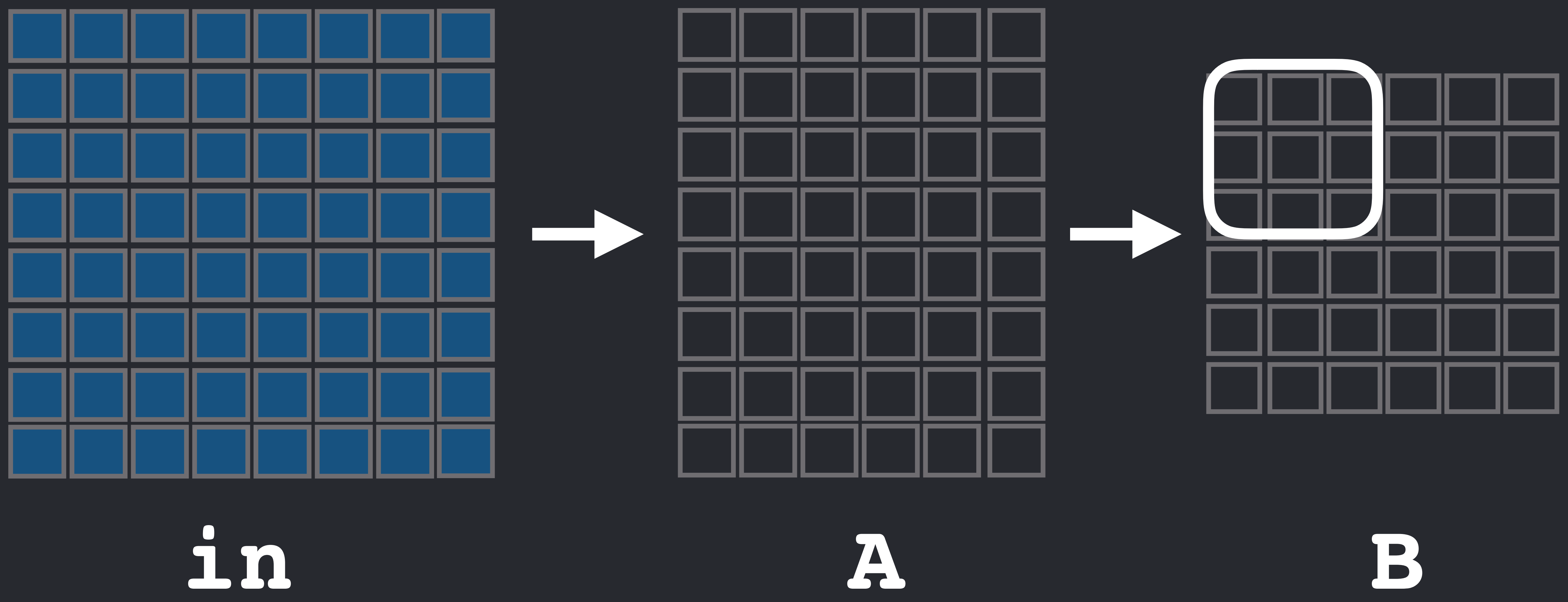

```
for each 3x3 tile in parallel
    compute required pixels of A
    compute pixels in tile of B
```

Cost = (Number of arithmetic operations) +
(Number of memory accesses) x (LOAD COST)

Estimating cost using interval analysis

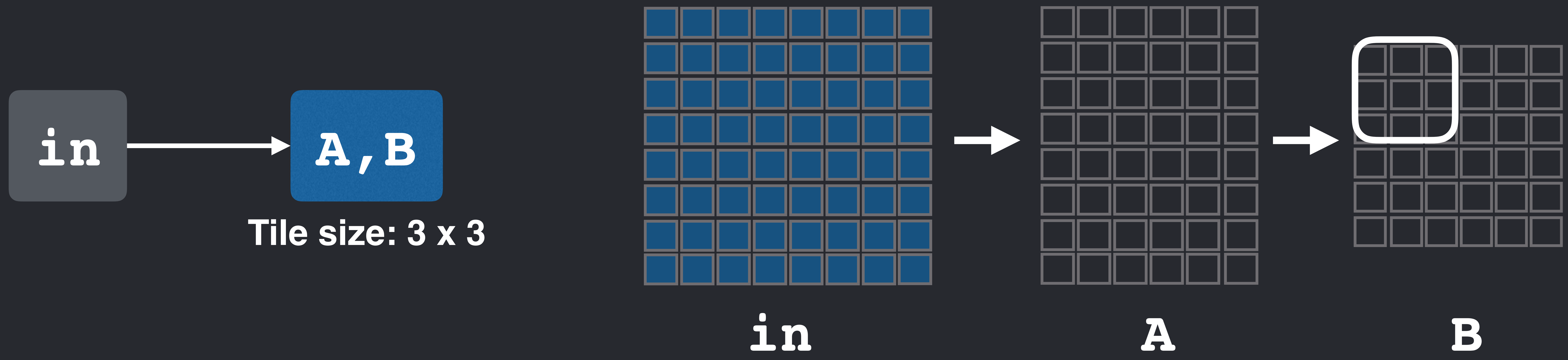


Tile size: 3 x 3



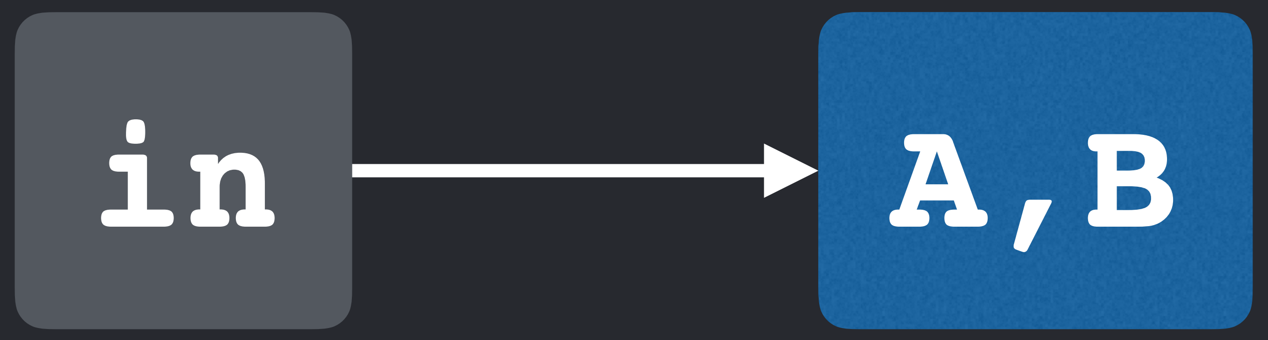
Cost = (Number of arithmetic operations) +
(Number of memory accesses) x (LOAD COST)

Estimating cost using interval analysis

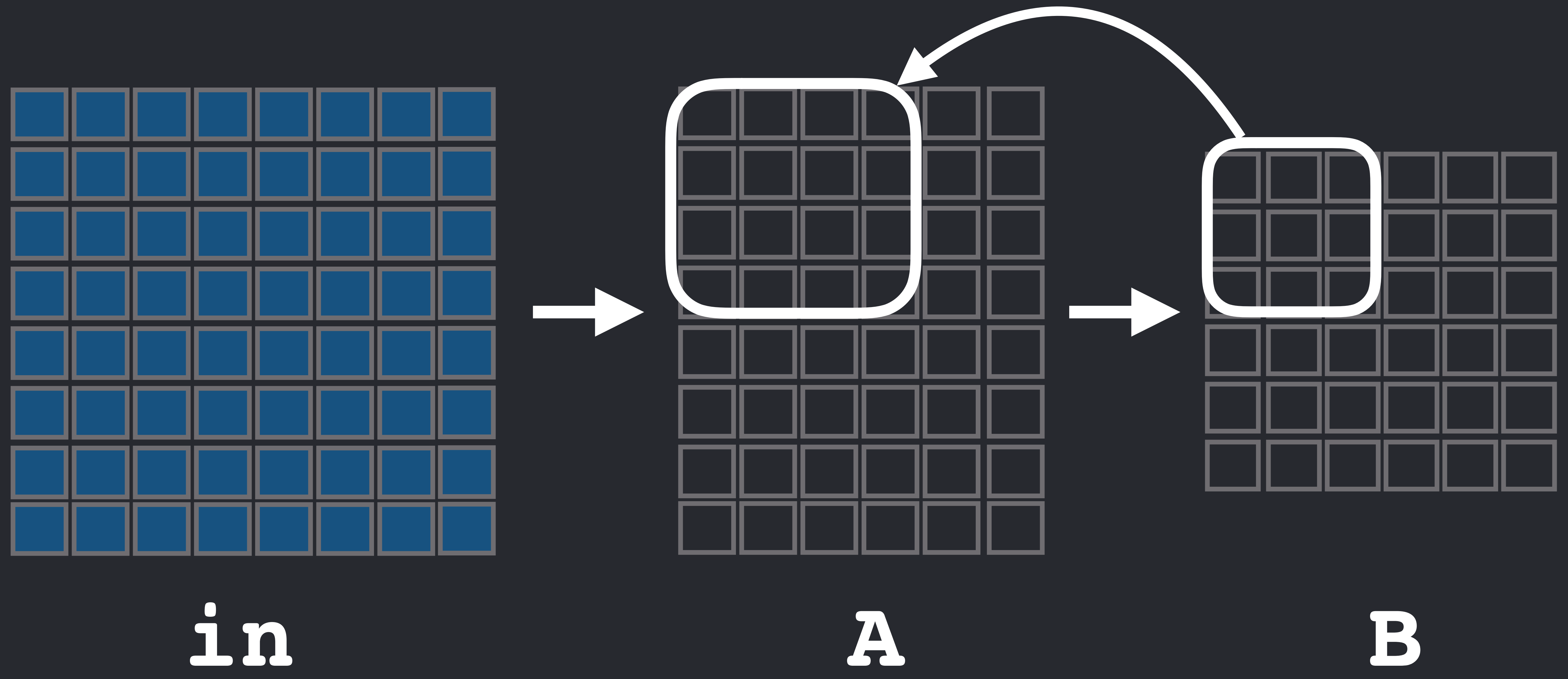


Cost = (Number of arithmetic operations) +
(Number of memory accesses) x (LOAD COST)

Estimating cost using interval analysis

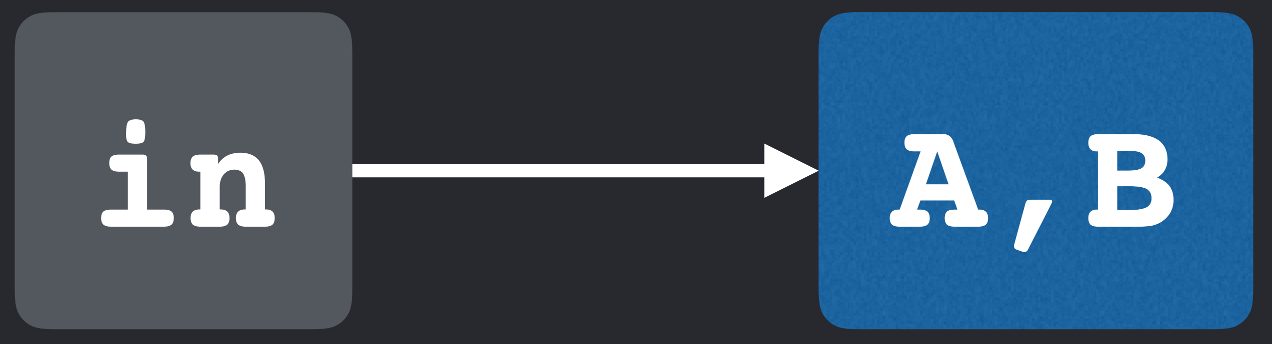


Tile size: 3 x 3

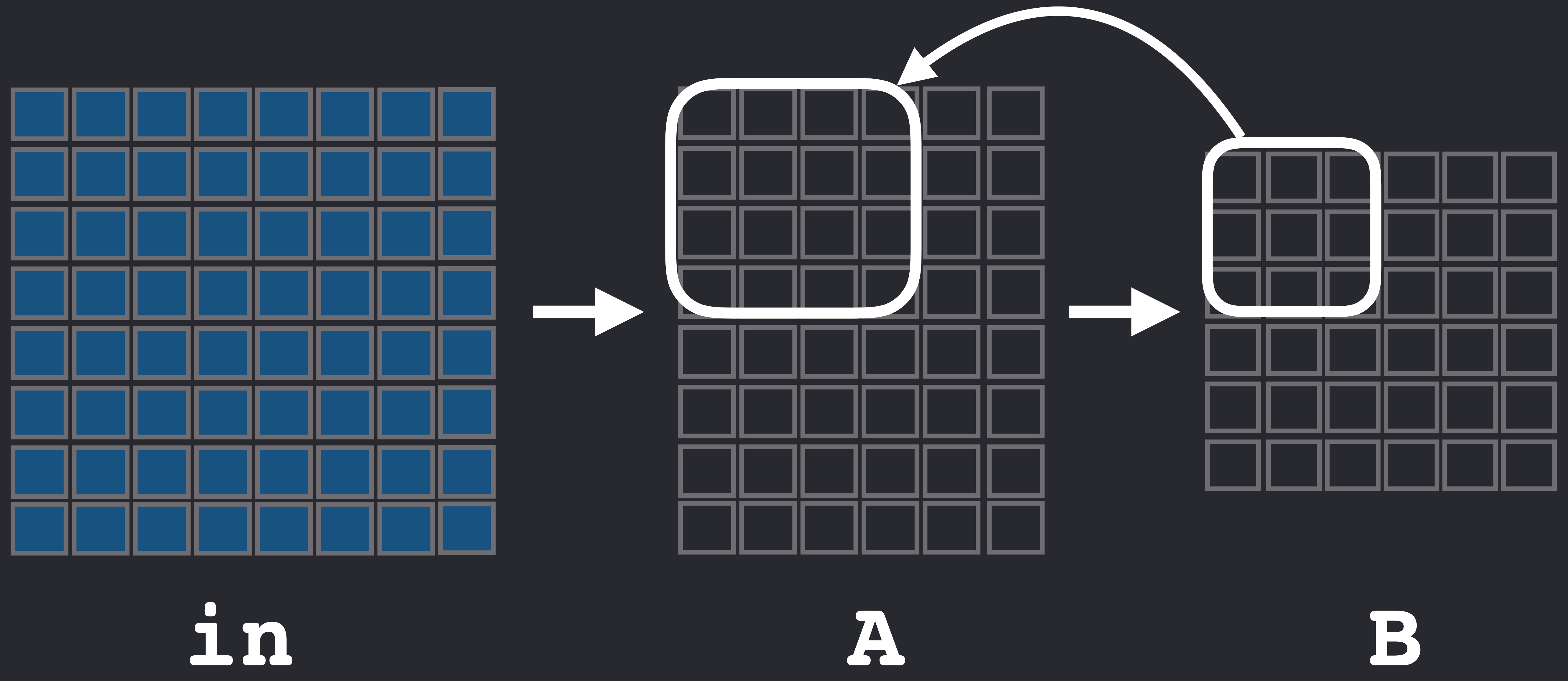


Cost = (Number of arithmetic operations) +
(Number of memory accesses) x (LOAD COST)

Estimating cost using interval analysis

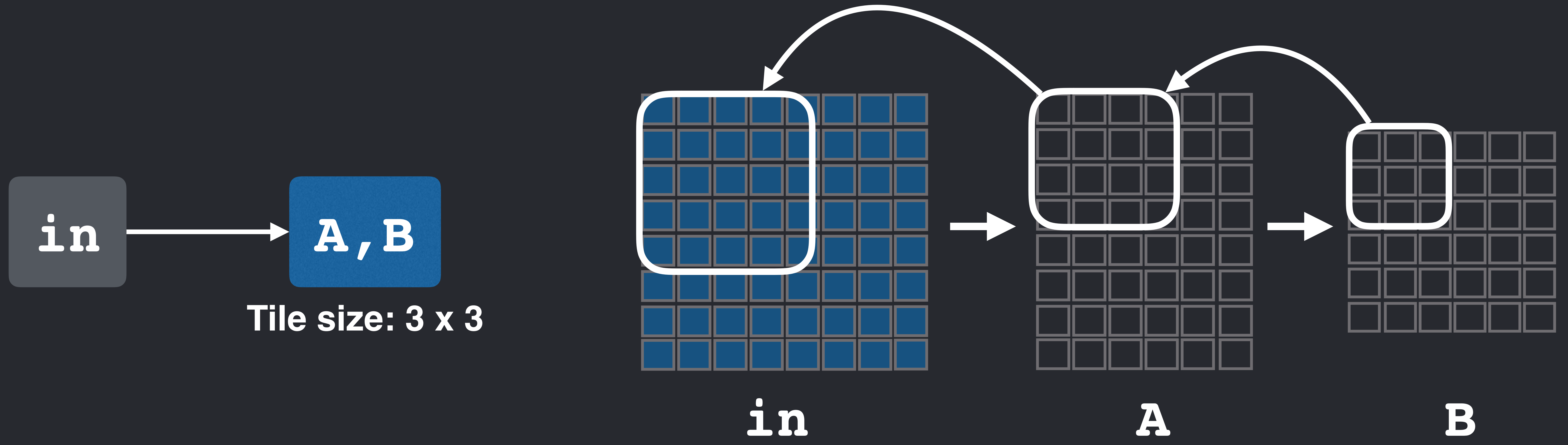


Tile size: 3 x 3



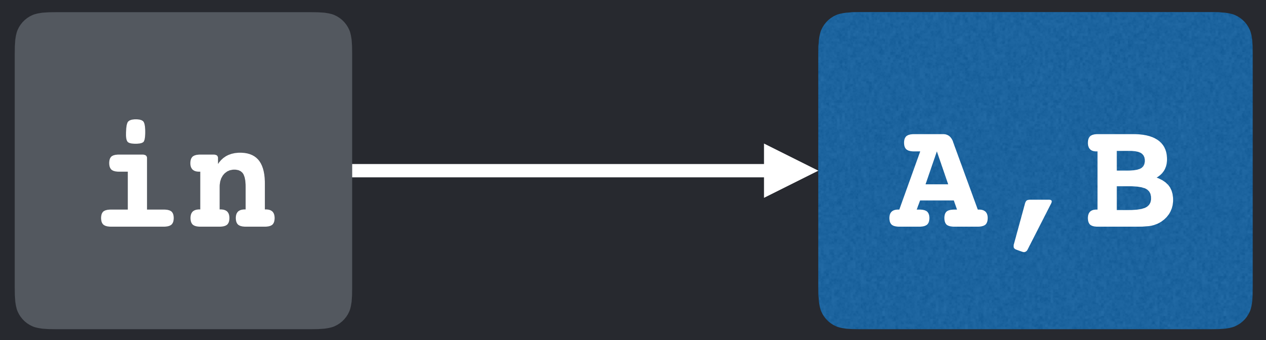
Cost = (Number of arithmetic operations) +
(Number of memory accesses) x (LOAD COST)

Estimating cost using interval analysis

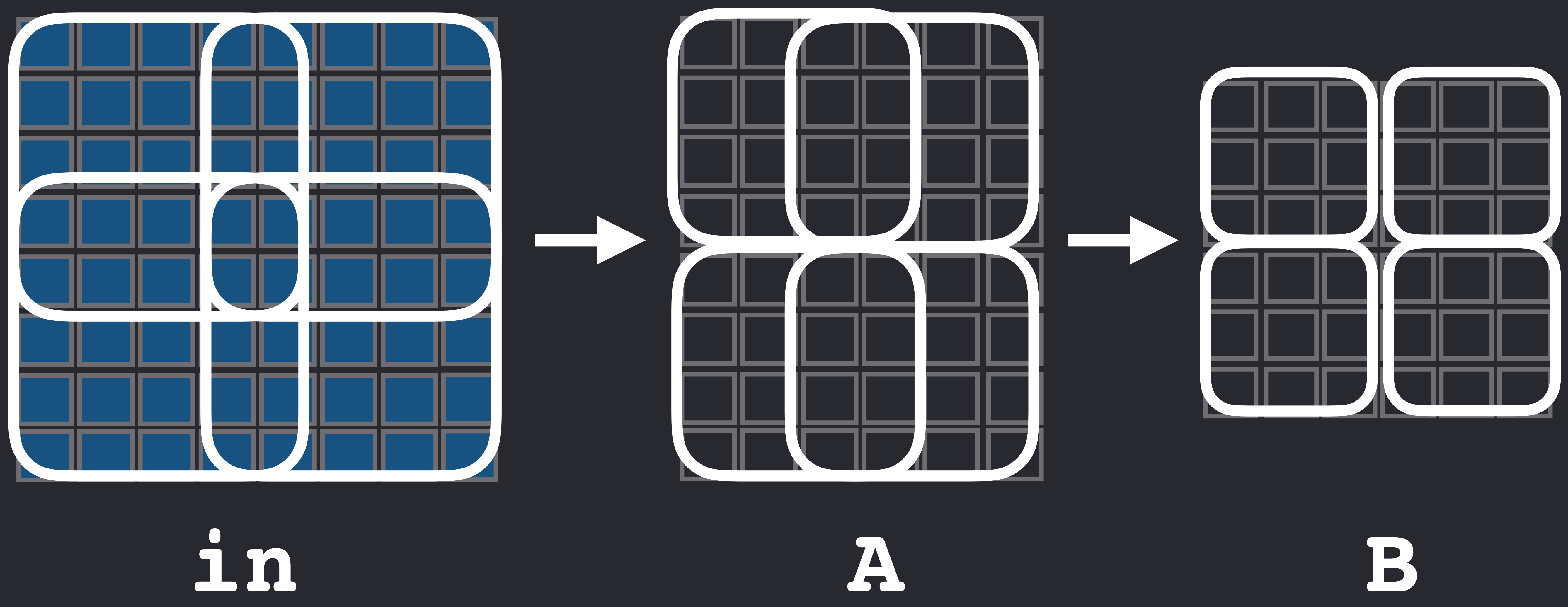


Cost = (Number of arithmetic operations) +
(Number of memory accesses) x (LOAD COST)

Estimating cost using interval analysis

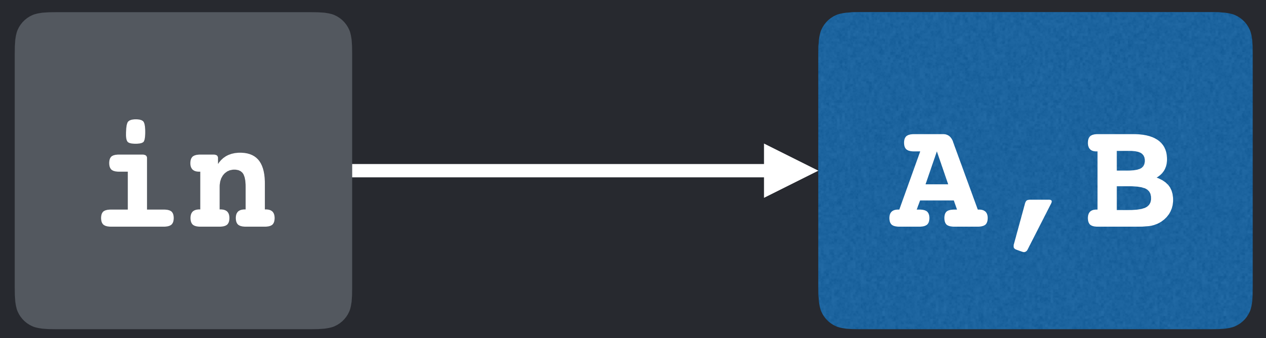


Tile size: 3 x 3

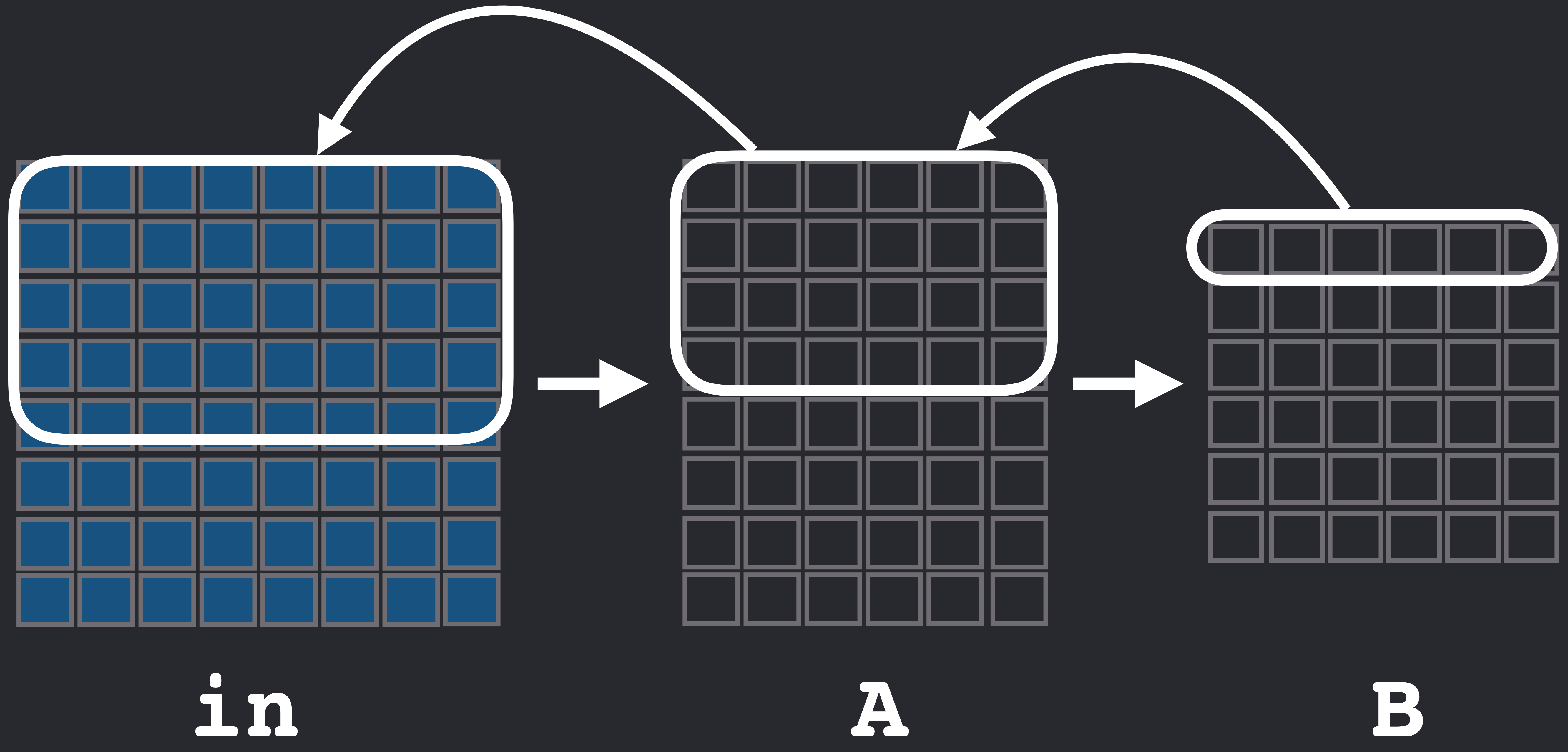


Cost = Number of tiles x Cost per tile

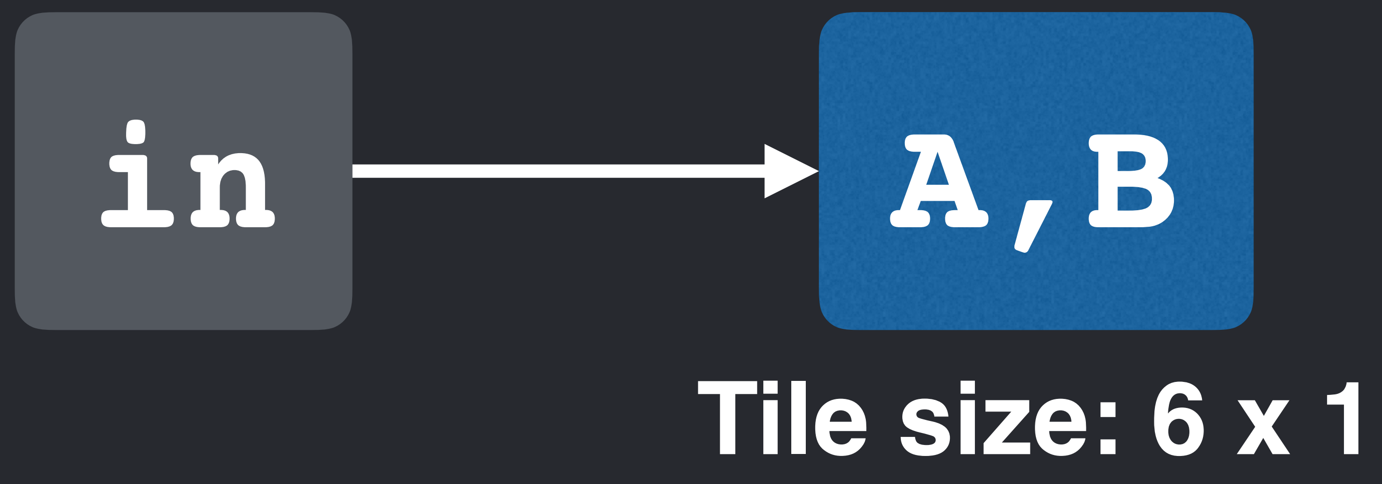
Search for best tile sizes

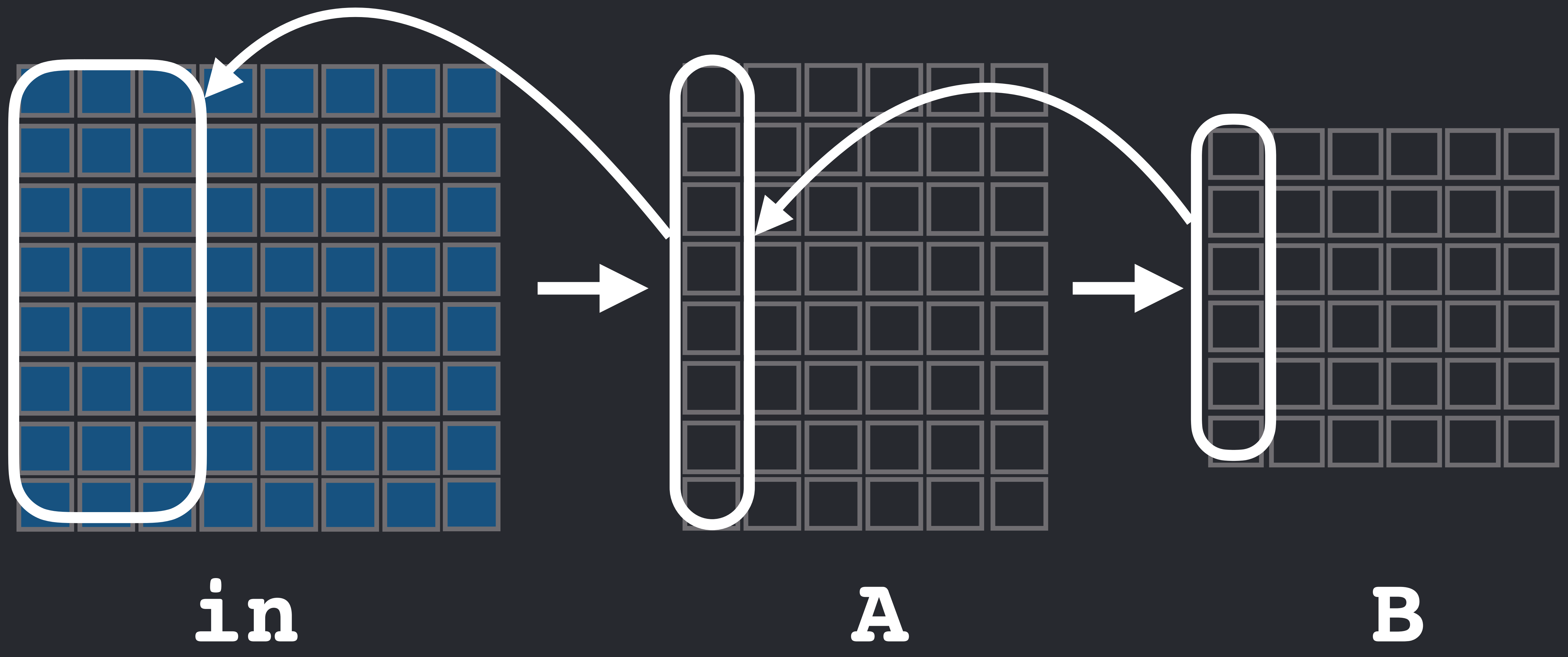


Tile size: 1 x 6

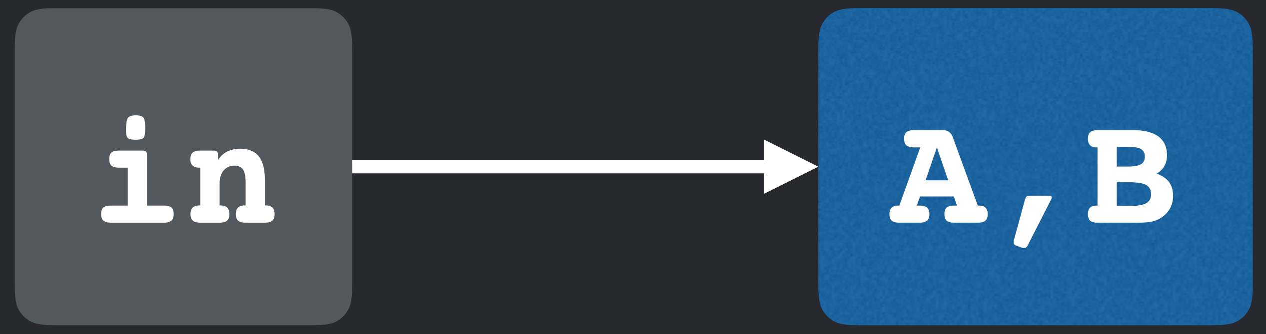


Search for best tile sizes

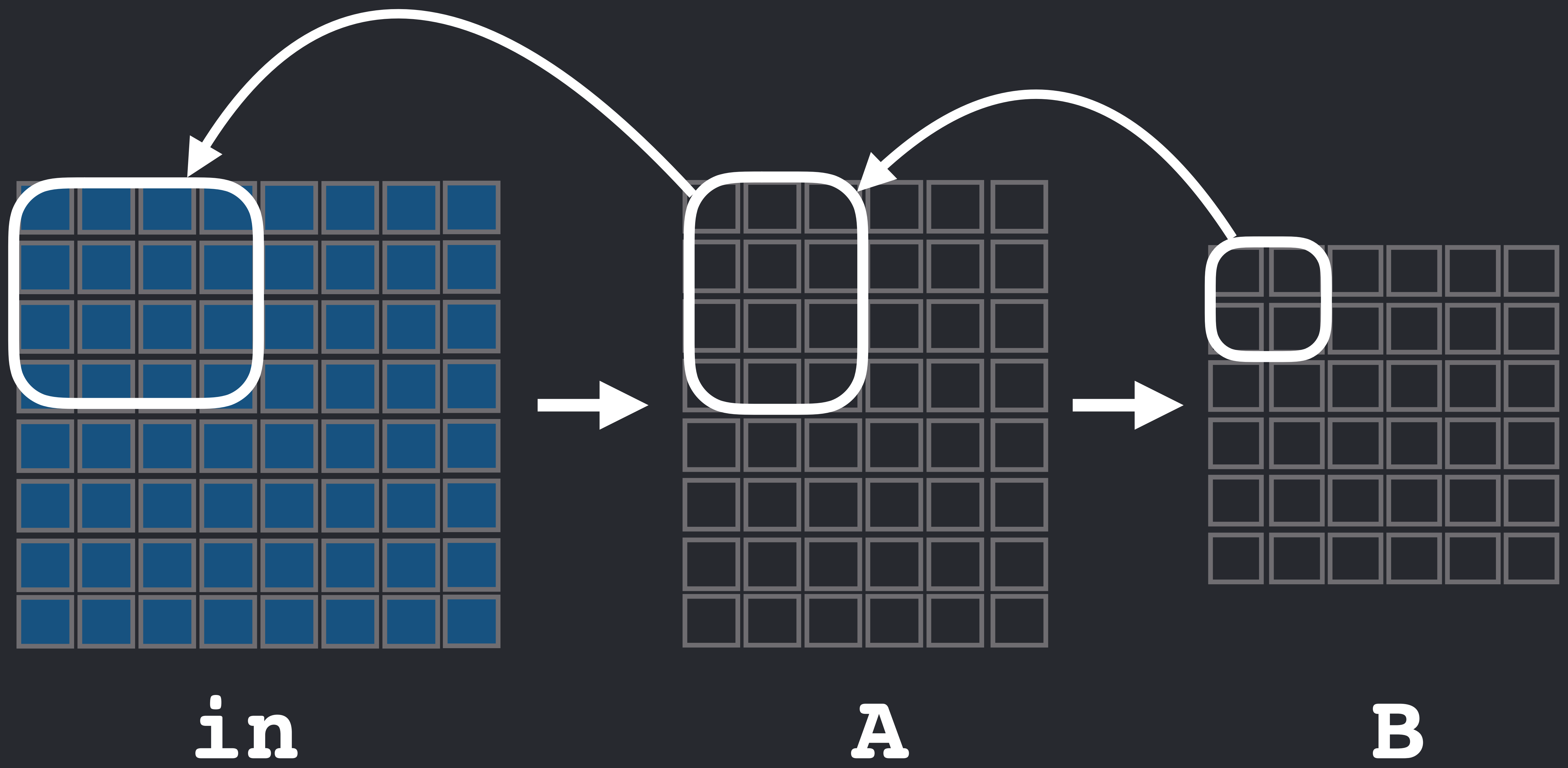




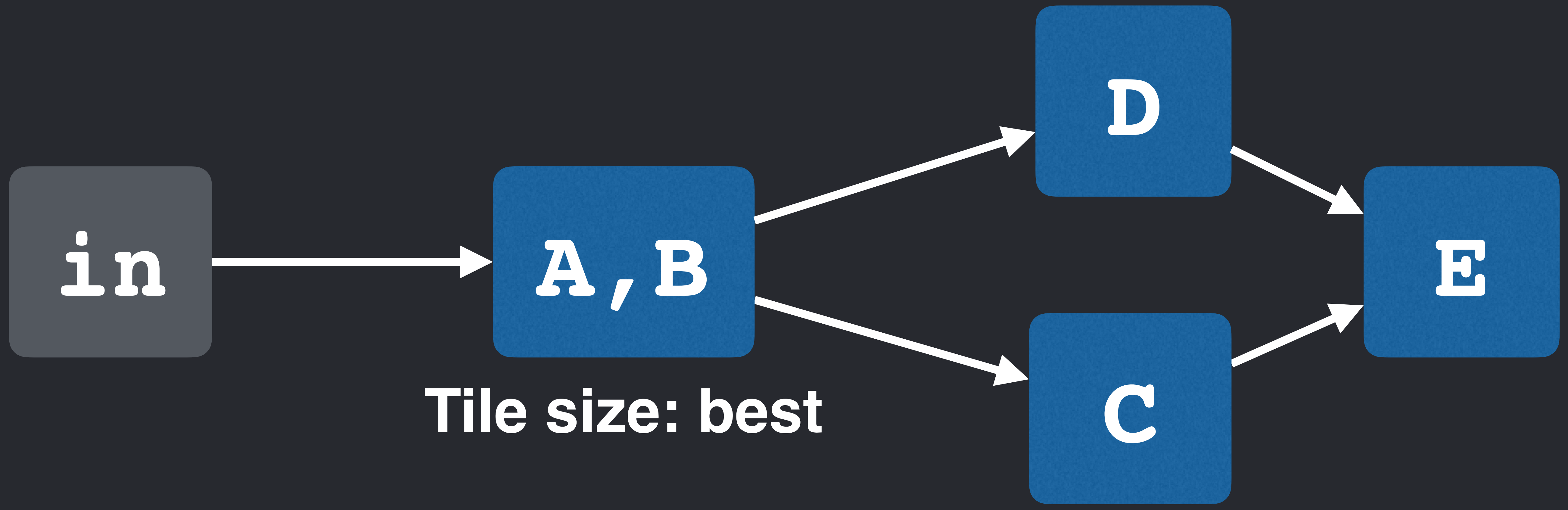
Search for best tile sizes



Tile size: 2 x 2



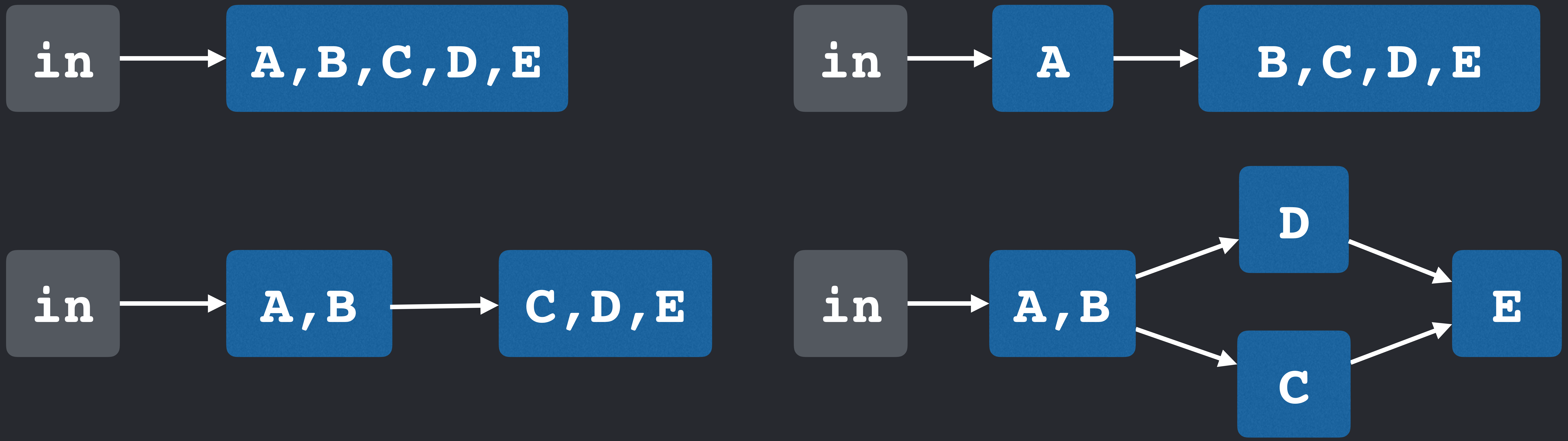
When to group stages?



Benefit(A,B) = Cost(A) + Cost(B) - Cost(A,B)

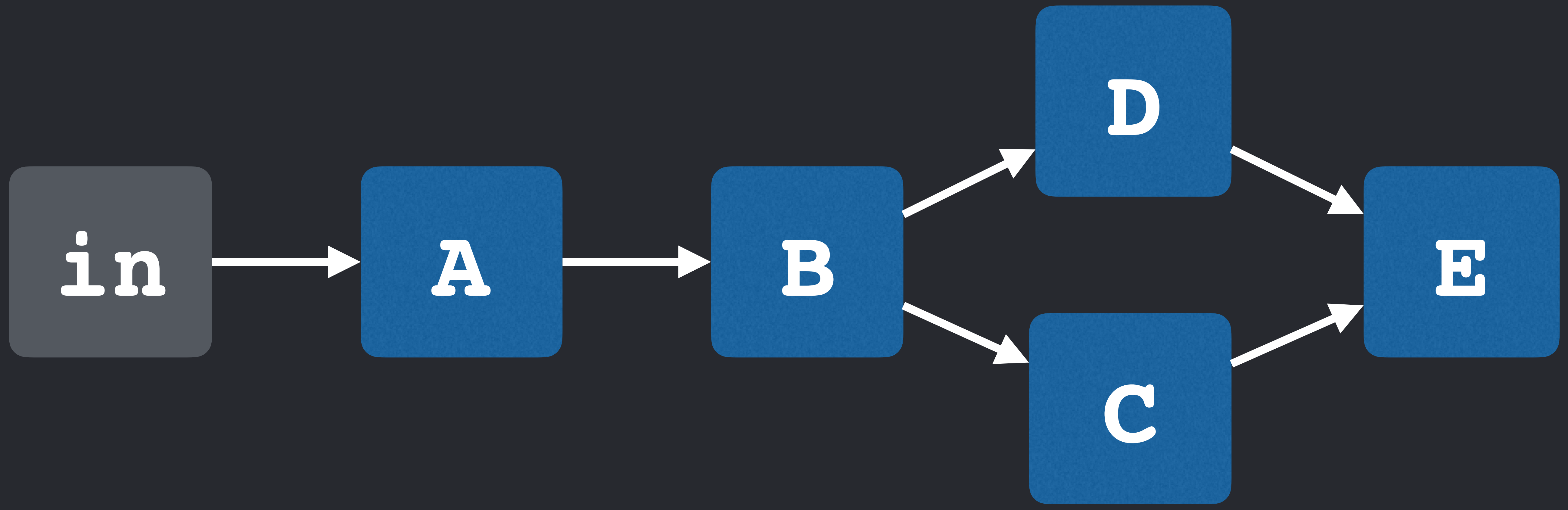
Tile size: best

Exhaustive search is infeasible



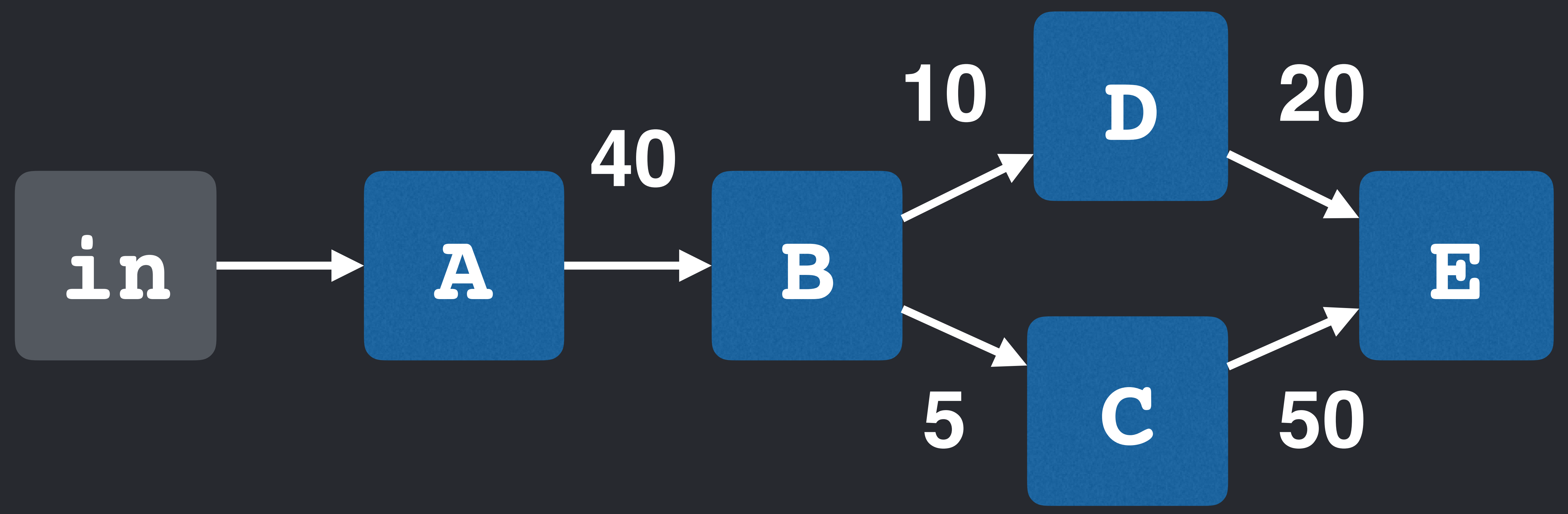
Exponential number of possible groupings

Greedy grouping algorithm



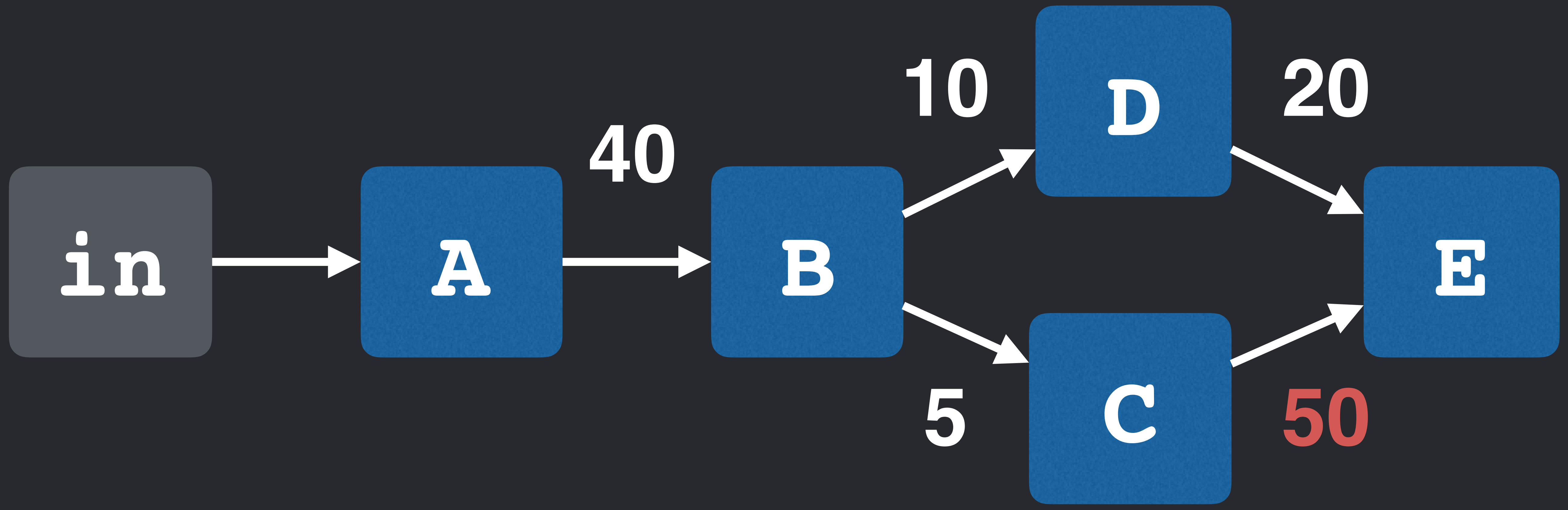
compute all pixels of A, in parallel
compute all pixels of B, in parallel
compute all pixels of C, in parallel
compute all pixels of D, in parallel
compute all pixels of E, in parallel

Greedy grouping algorithm



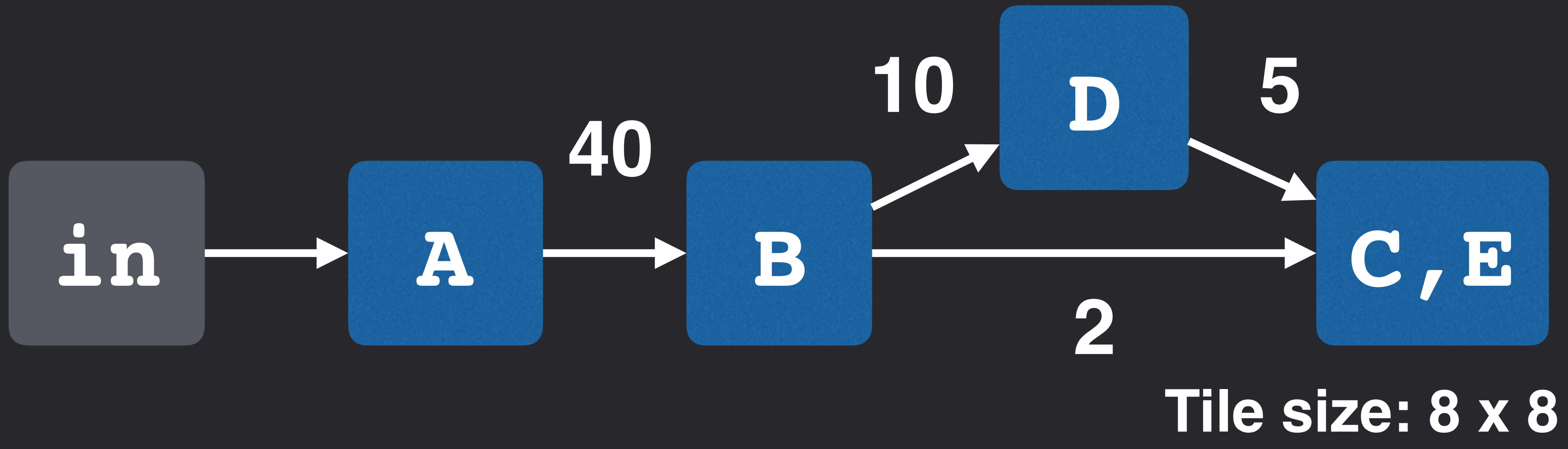
compute all pixels of A, in parallel
compute all pixels of B, in parallel
compute all pixels of C, in parallel
compute all pixels of D, in parallel
compute all pixels of E, in parallel

Greedy grouping algorithm



compute all pixels of A, in parallel
compute all pixels of B, in parallel
compute all pixels of C, in parallel
compute all pixels of D, in parallel
compute all pixels of E, in parallel

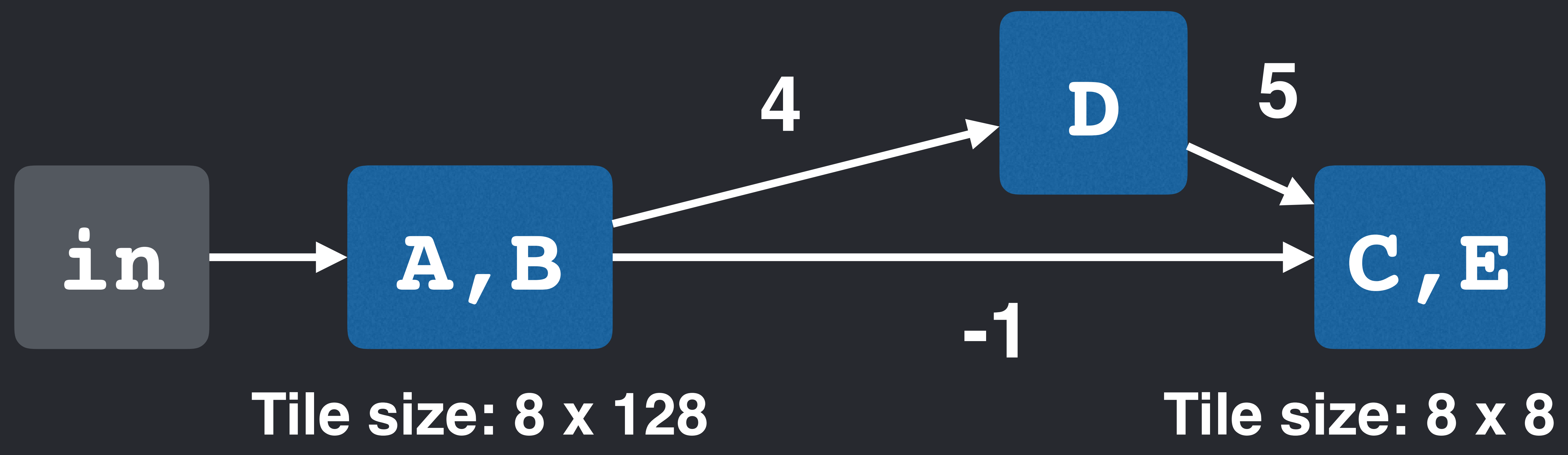
Greedy grouping algorithm



compute all pixels of A, in parallel
compute all pixels of B, in parallel
compute all pixels of D, in parallel

for each 8x8 tile in parallel
    compute required pixels of C
    compute pixels in tile of E

Greedy grouping algorithm

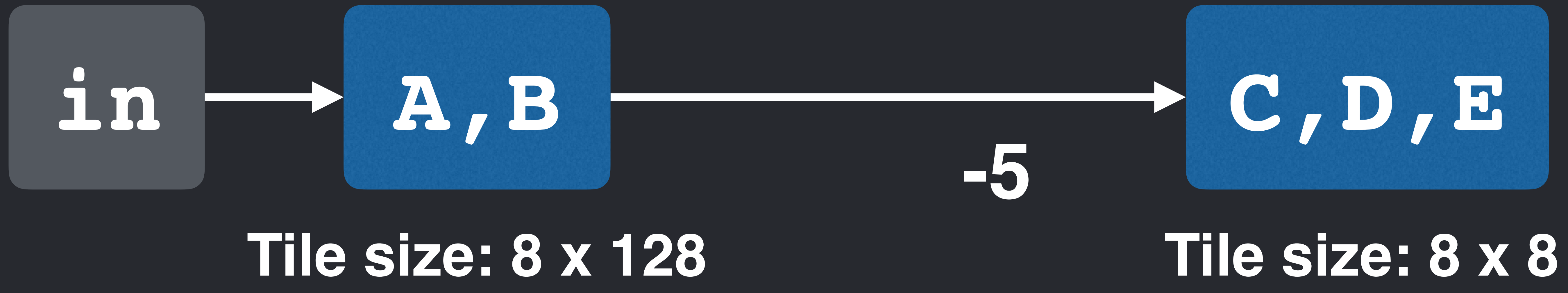


for each 8x128 tile in parallel
    compute required pixels of A
    compute pixels in tile of B

compute all pixels of D, in parallel

for each 8x8 tile in parallel
    compute required pixels of C
    compute pixels in tile of E

Greedy grouping algorithm



for each 8x128 tile in parallel
    compute required pixels of A
    compute pixels in tile of B

for each 8x8 tile in parallel
    compute required pixels of C
    compute required pixels of
    compute pixels in tile of E

Auto scheduler implementation details

- Multi-core parallelism, vectorization, loop reordering, and unrolling

```
for each 8x128 tile in parallel
    vectorize compute required pixels of A unroll x by 4
    vectorize compute required pixels of B
    vectorize compute pixels in tile of D

for each 8x8 tile in parallel
    vectorize compute required pixels of C unroll y by 2
    vectorize compute pixels in tile of E
```

Evaluation

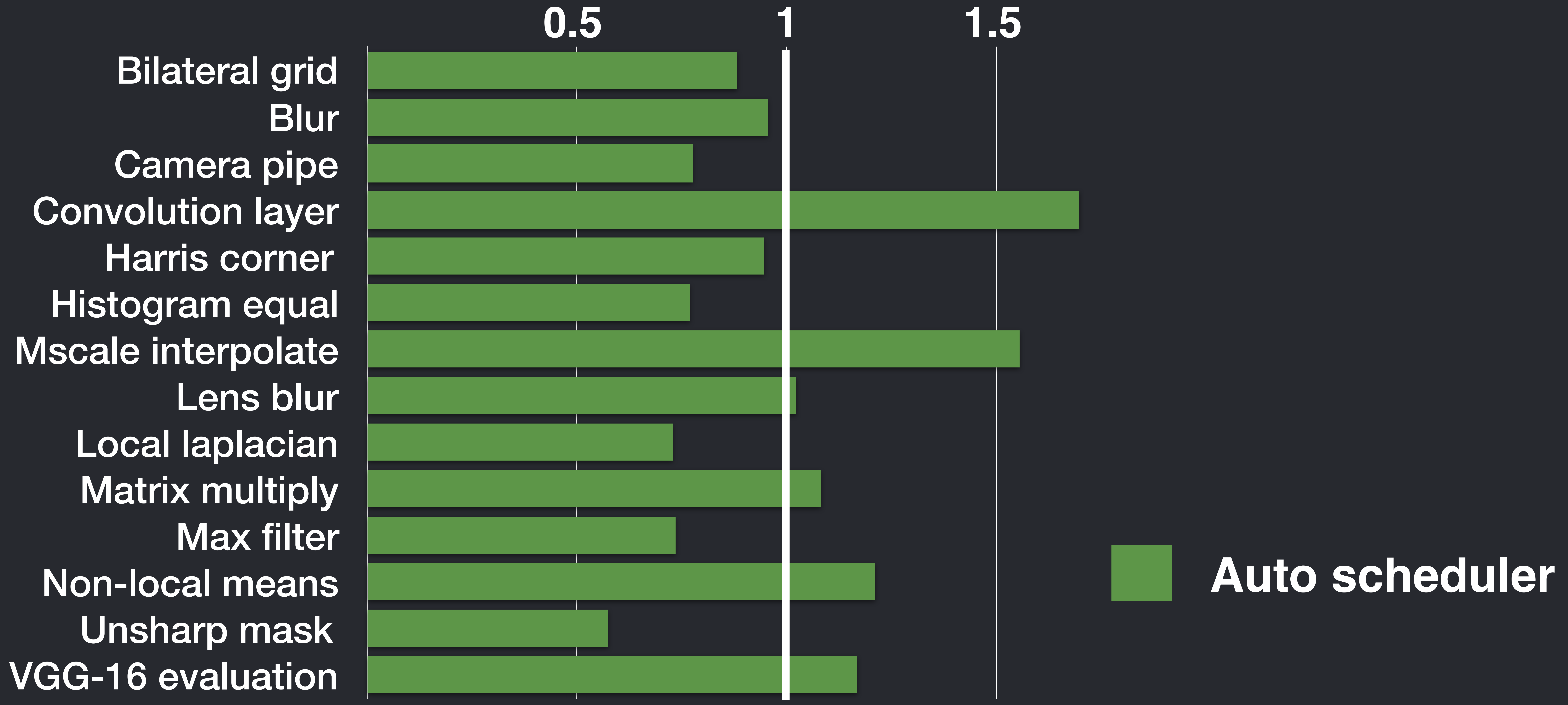
Benchmarks of varying complexity and structure

Benchmark	Stages
Blur	3
Unsharp mask	9
Harris corner detection	13
Camera RAW processing	30
Non-local means denoising	13
Max-brightness filter	9
Multi-scale interpolation	52
Local-laplacian filter	103
Synthetic depth-of-field	74
Bilateral filter	8
Histogram equalization	7
VGG-16 deep network eval	64

Auto scheduler generates schedules in seconds

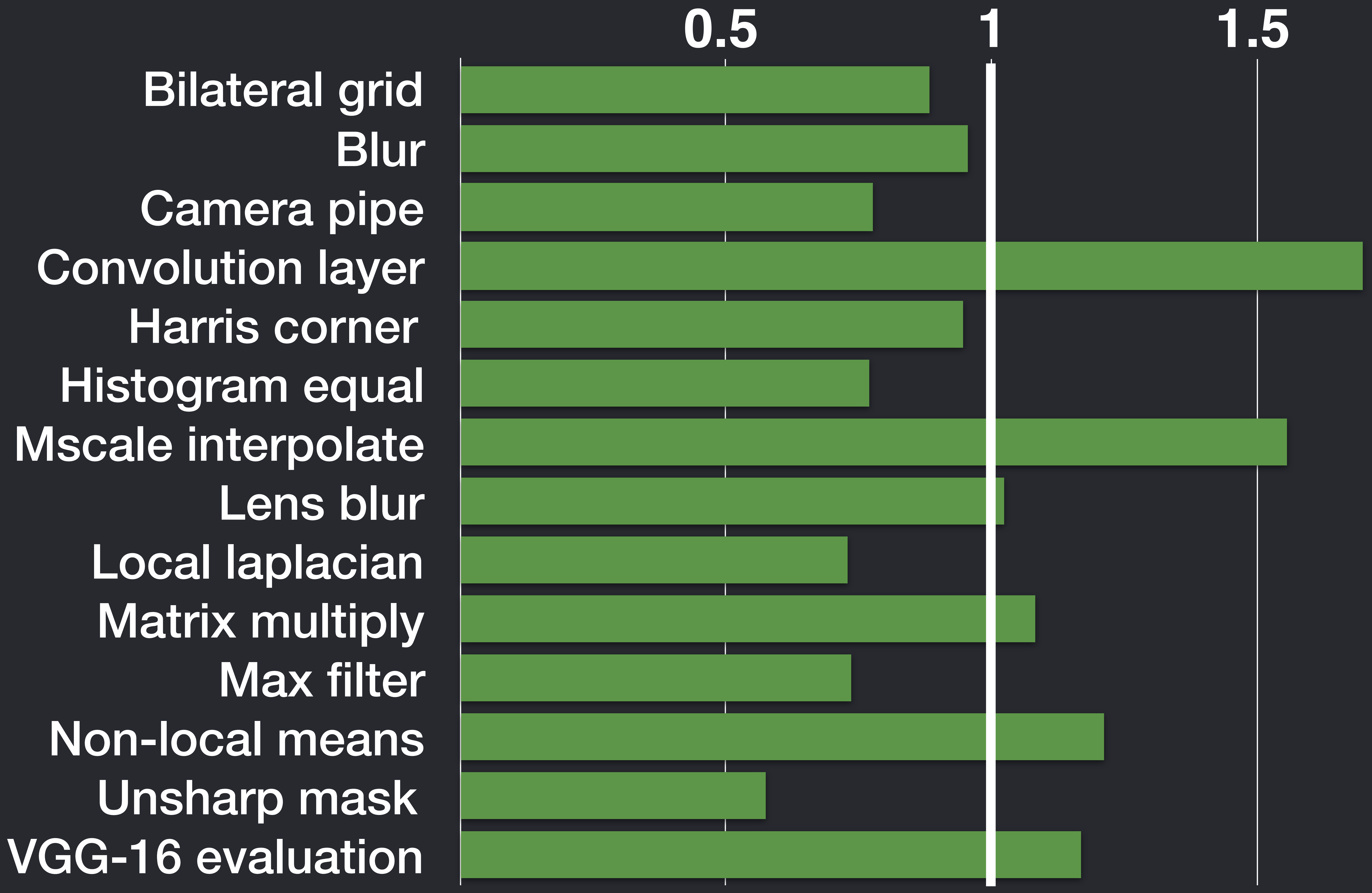
Benchmark	Stages	Compile time (s)
Blur	3	<1
Unsharp mask	9	<1
Harris corner detection	13	<1
Camera RAW processing	30	<1
Non-local means denoising	13	<1
Max-brightness filter	9	<1
Multi-scale interpolation	52	2.6
Local-laplacian filter	103	3.9
Synthetic depth-of-field	74	55
Bilateral filter	8	<1
Histogram equalization	7	<1
VGG-16 deep network eval	64	6.9

Auto scheduler performs comparably to experts



Performance relative to experts (6 core Xeon CPU)

Auto scheduler performs comparably to experts

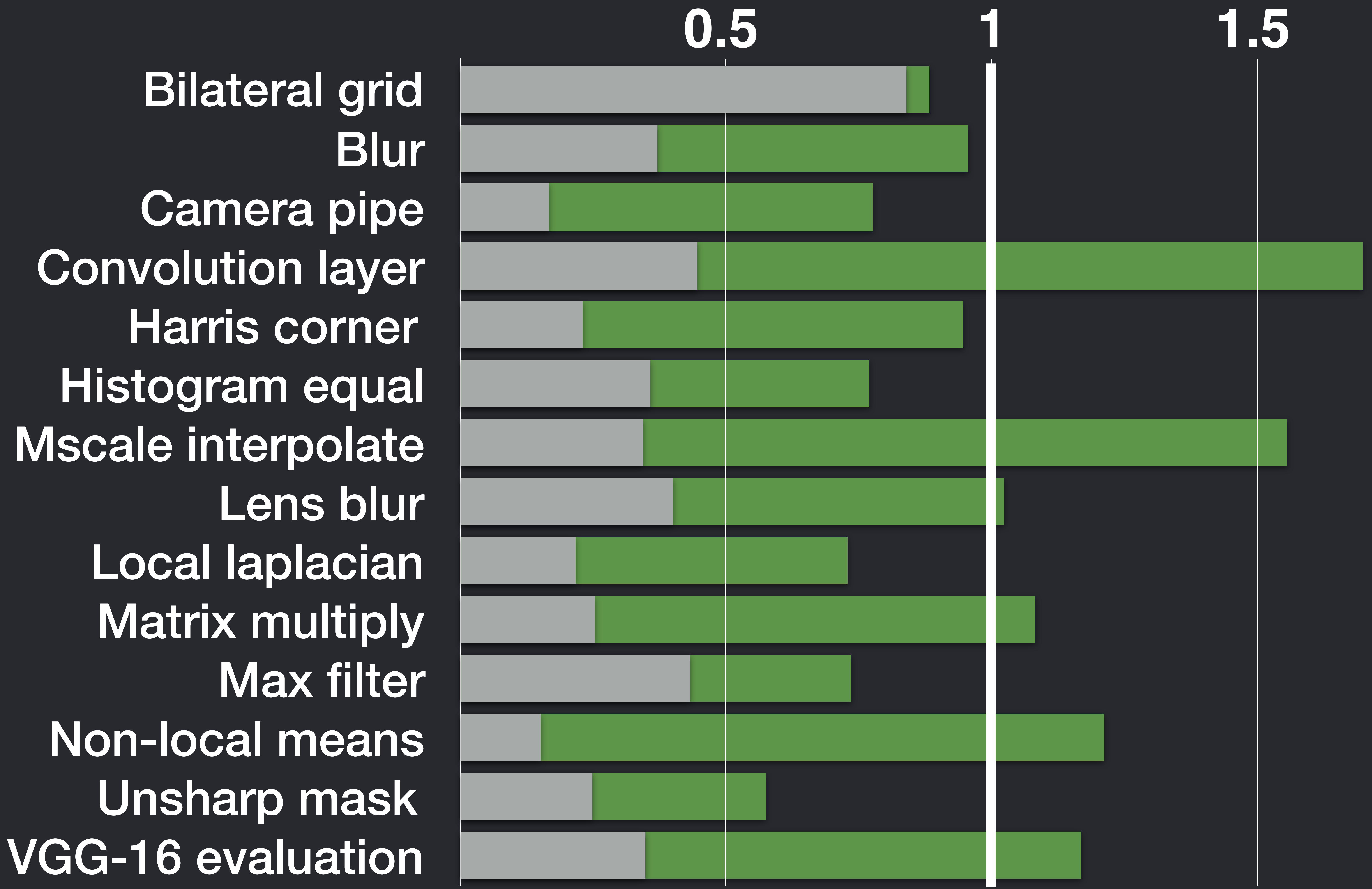


On 8 of the 14 benchmarks performance within 10% of experts or better

Auto scheduler

Performance relative to experts (6 core Xeon CPU)

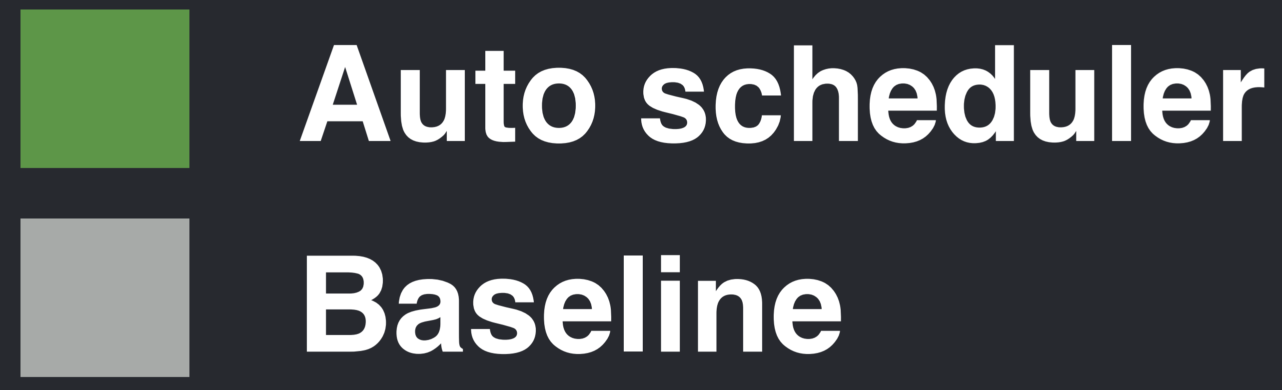
Auto scheduler performs comparably to experts



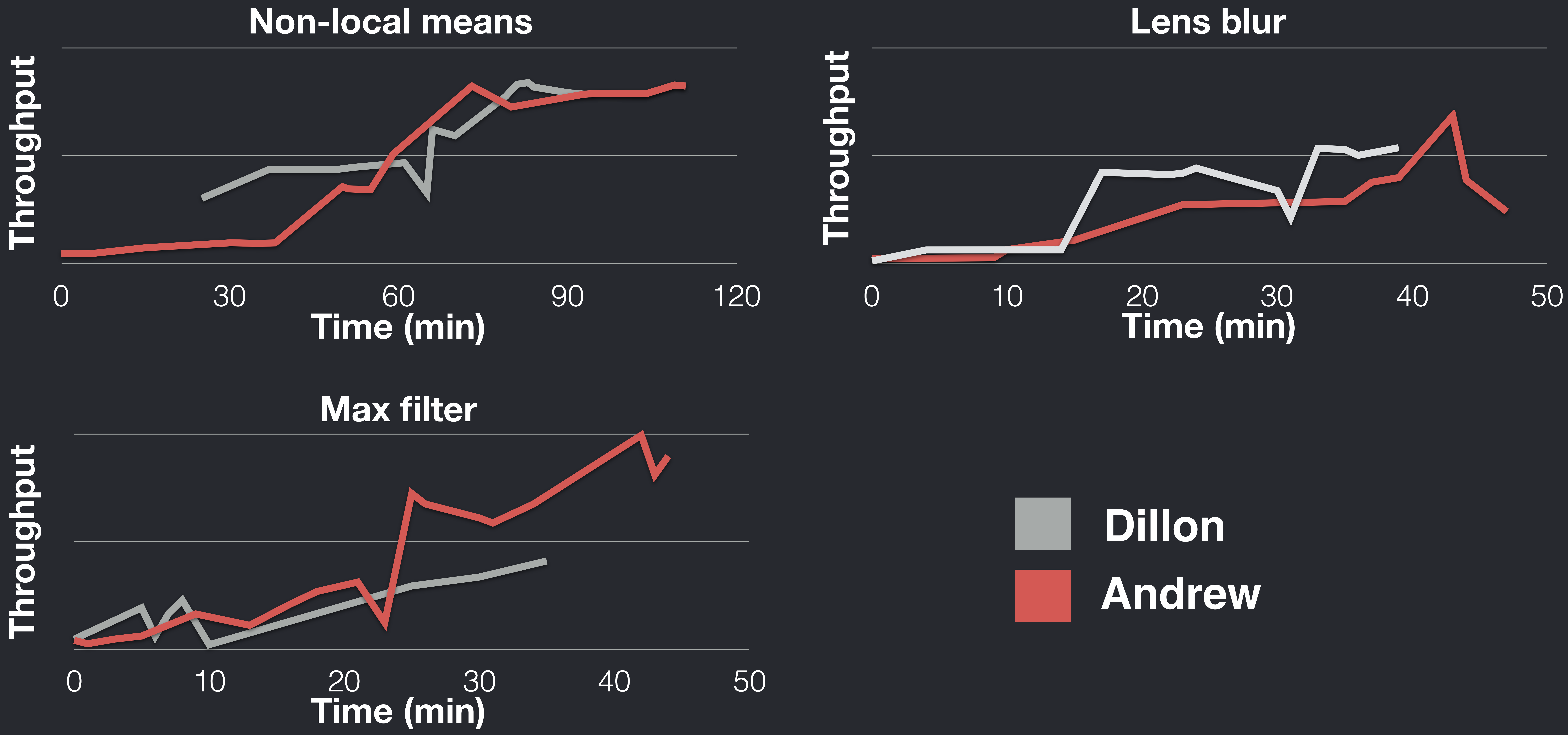
Performance relative to experts (6 core Xeon CPU)

On 8 of the 14 benchmarks performance within 10% of experts or better

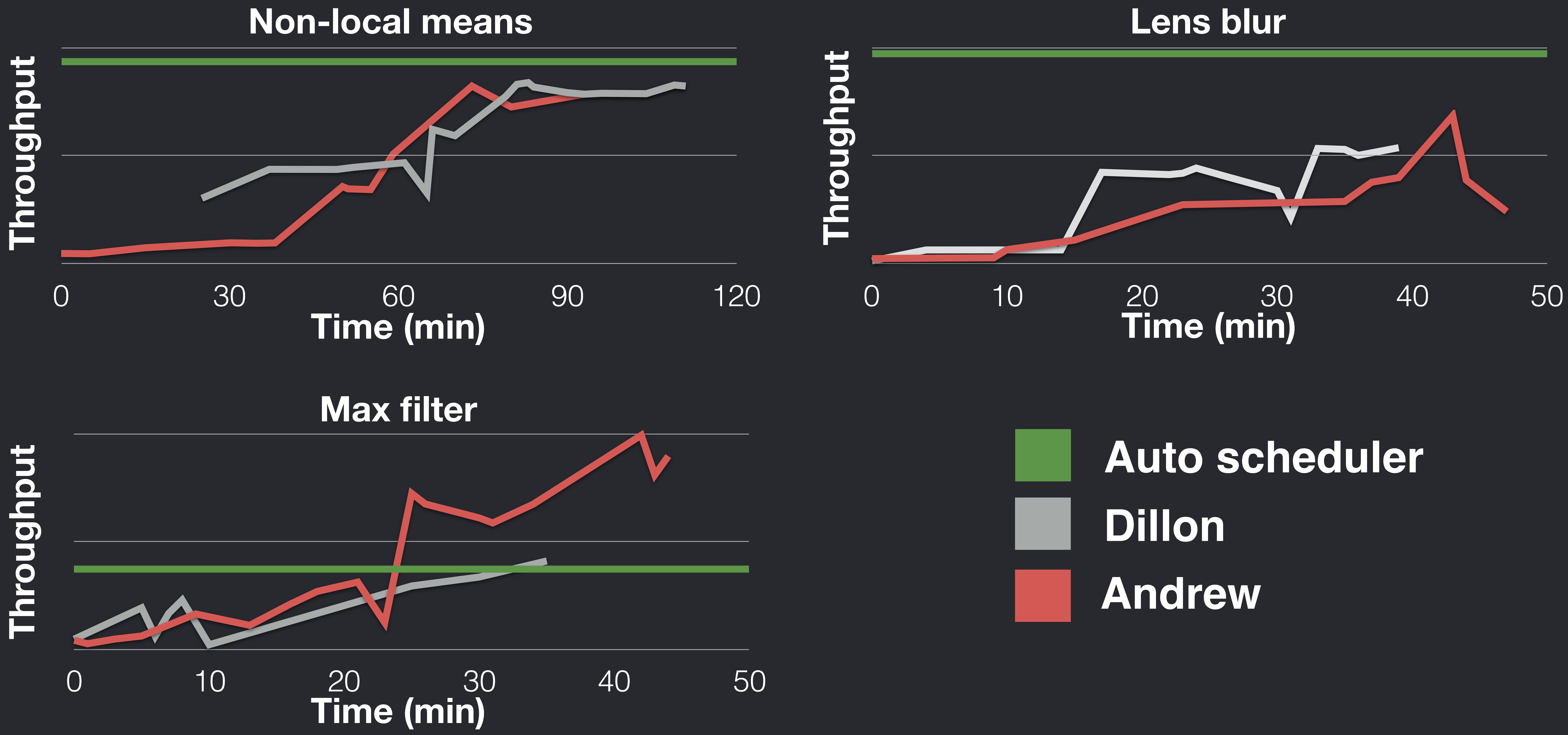
Baseline schedules exploit multi-core and vector parallelism but no grouping



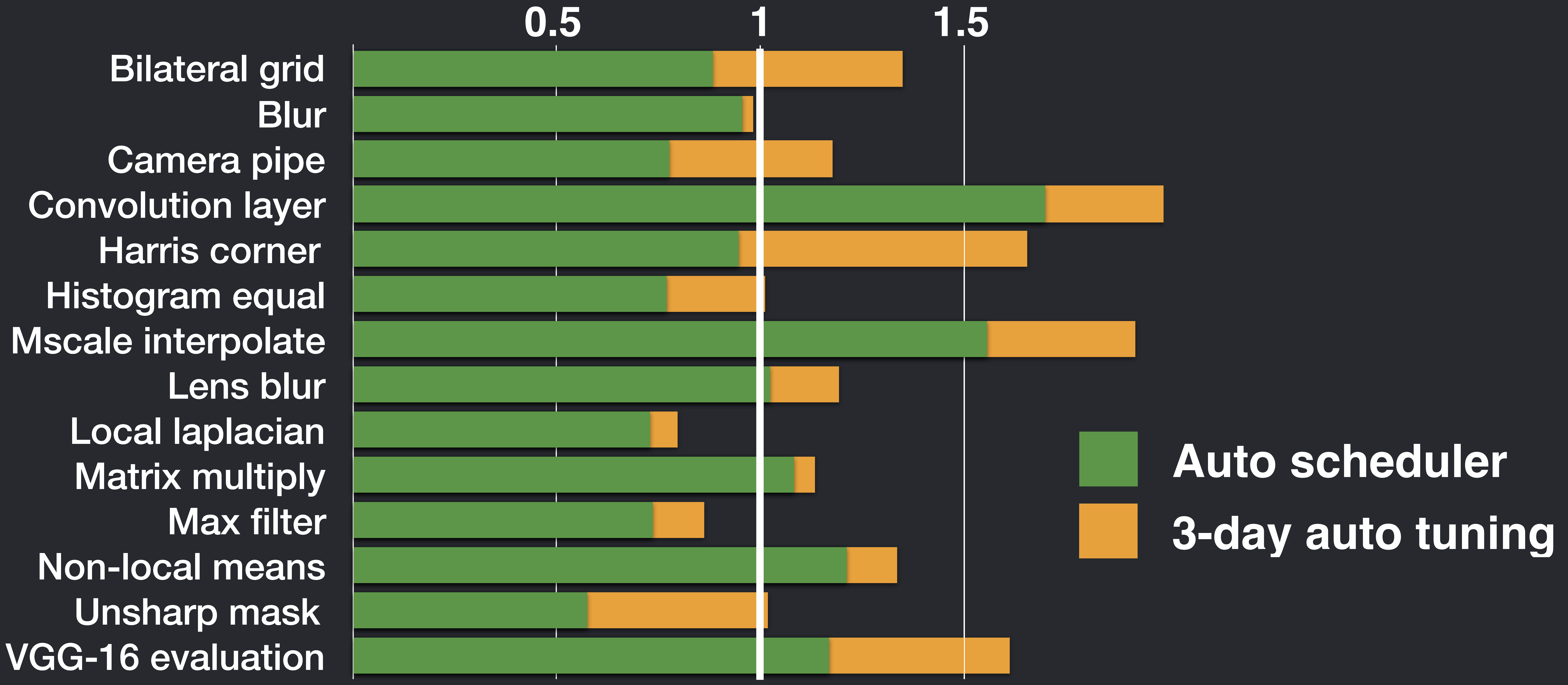
Auto scheduler can save time for experts



Auto scheduler can save time for experts

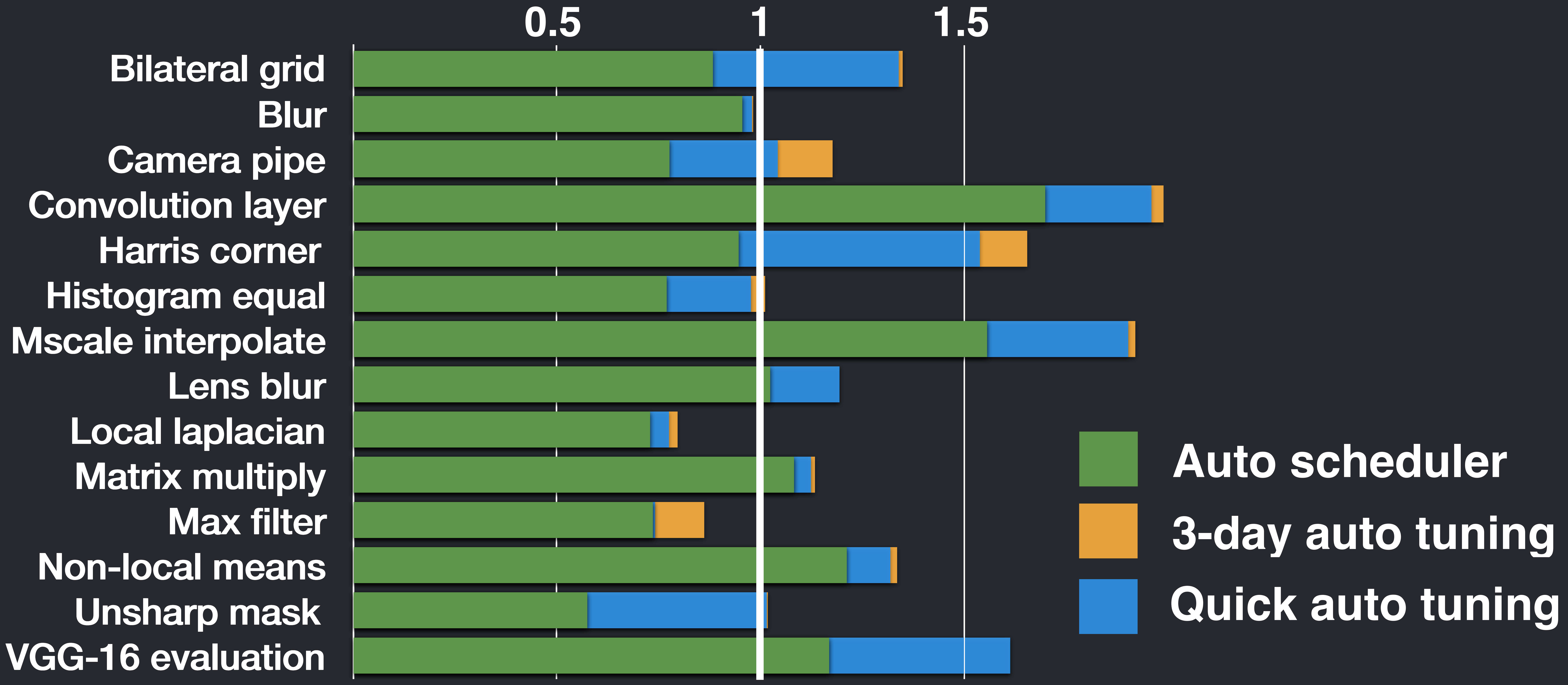


Exploring cost model parameters



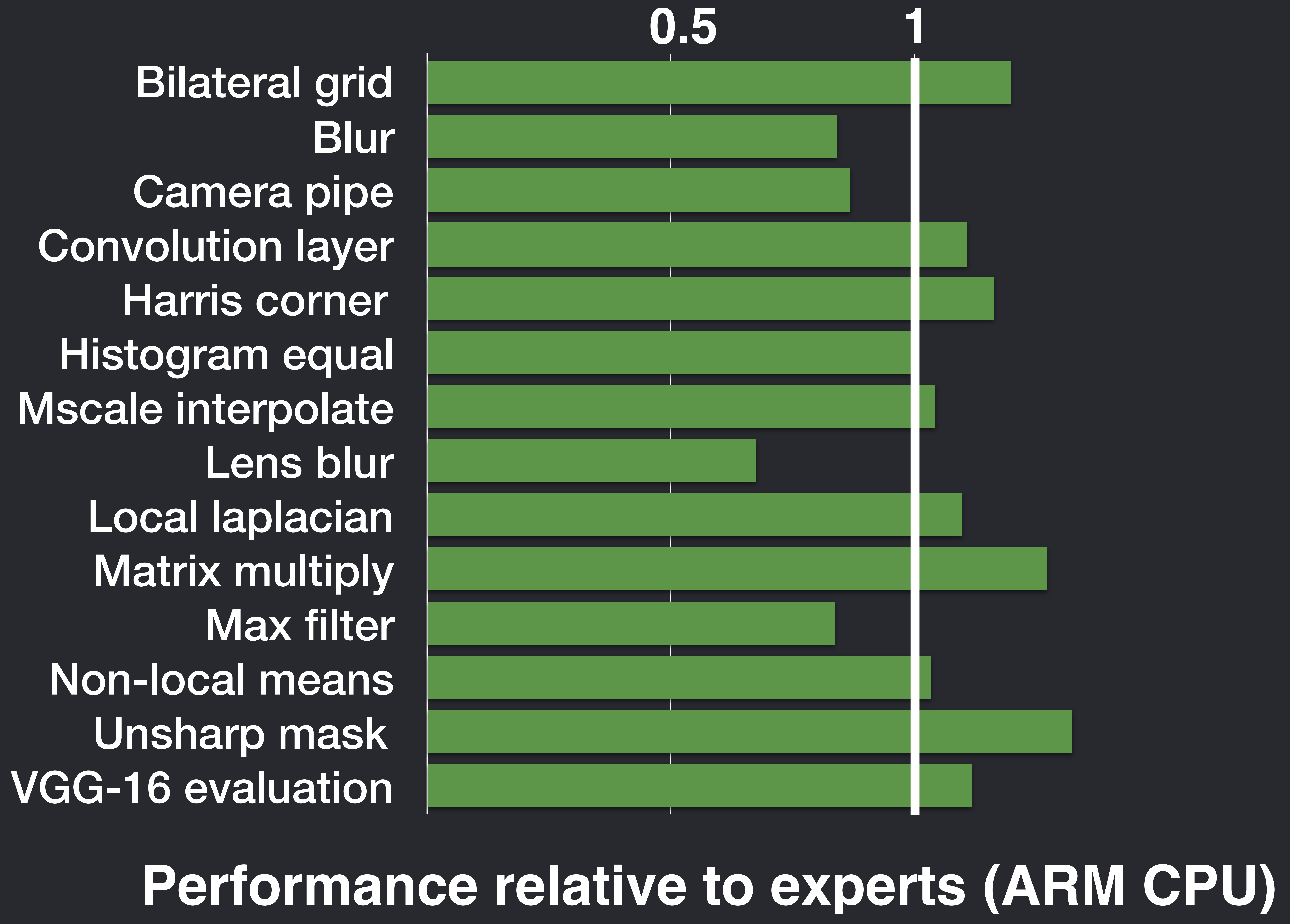
Performance relative to experts (6 core Xeon CPU)

Exploring cost model parameters

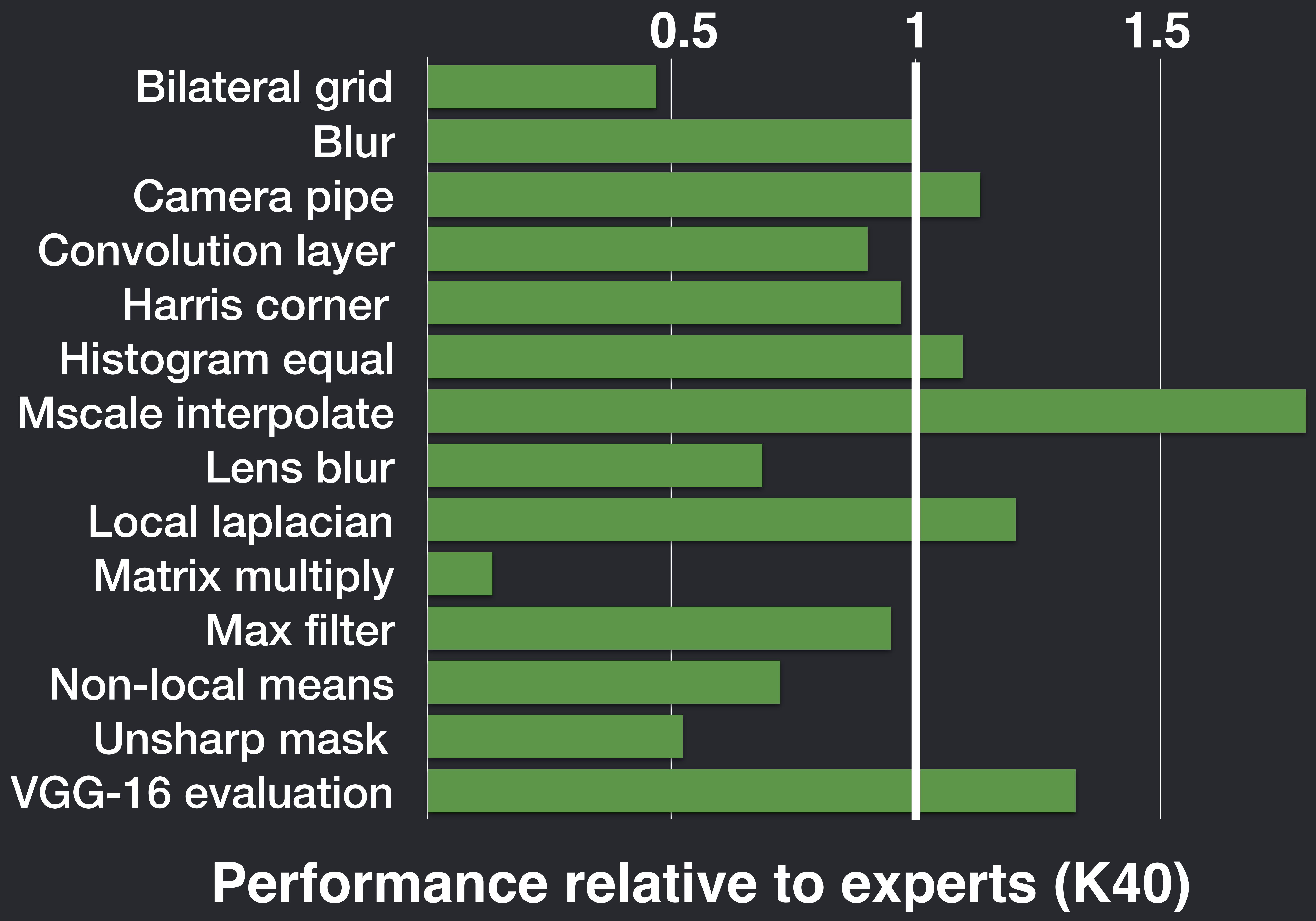


Performance relative to experts (6 core Xeon CPU)

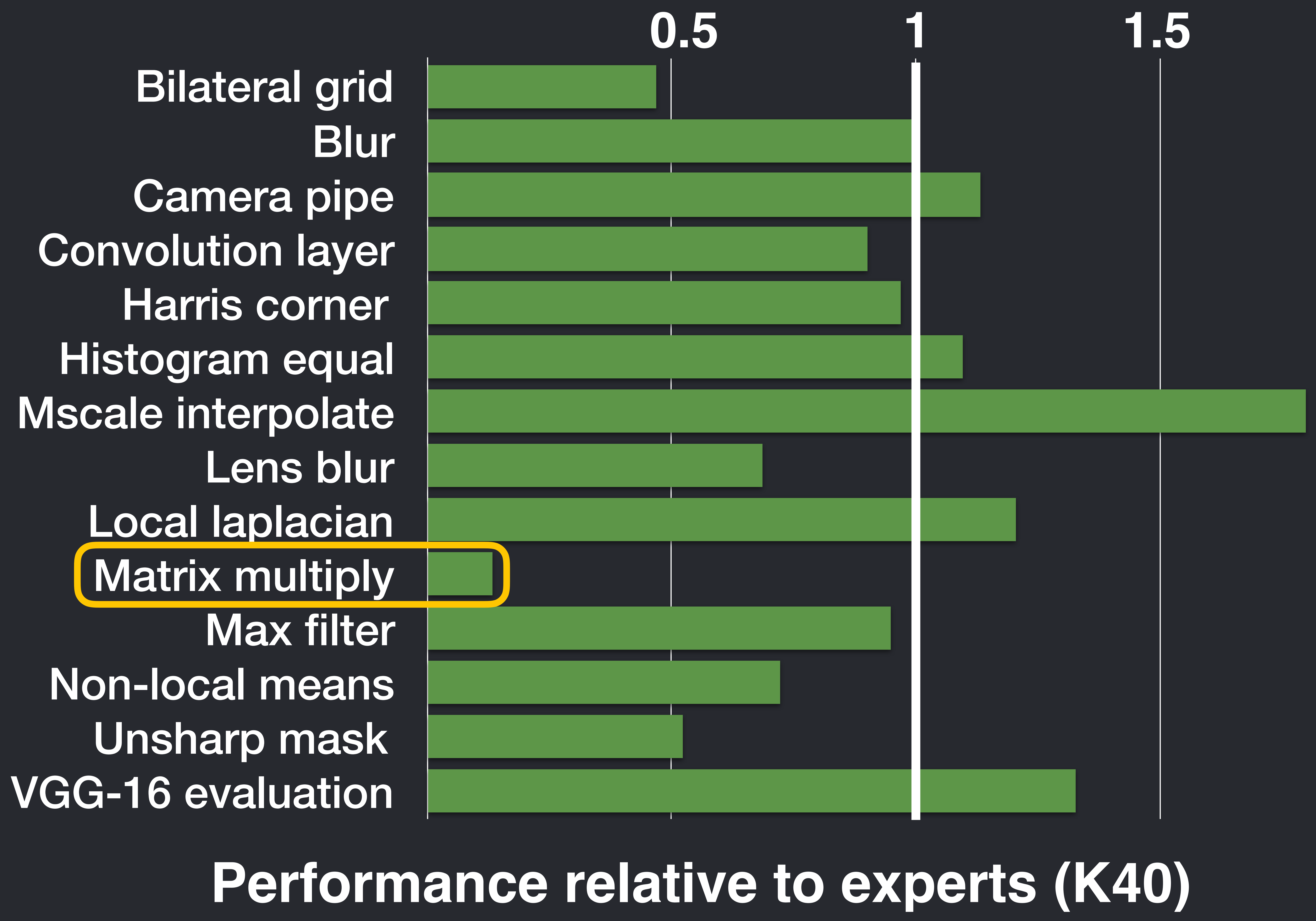
Quad core ARM performance



K40 GPU performance



K40 GPU performance



Prior work

Optimizing Halide via auto-tuning and stochastic search [Ragan-Kelley 13, Ansel 14]:

- Compilation time: hours to days
- Output up to 5-10x slower than hand-tuned implementations

Darkroom [Hegarty 14]:

- Auto-scheduling assuming applications restricted to fixed-size stencils

PolyMage [Mullapudi 15]: polyhedral-based optimization

- Greedy group-and-tile algorithm was inspired by PolyMage
- Polyhedral approach cannot analyze non-affine and data-dependent computations

Limitations

Restricted space of schedules

- Does not consider sliding windows and multi-level tiling

No human interaction with the auto scheduler

- Enable experts to guide the scheduling process

Summary

Algorithm that generates Halide schedules

- Competitive with experts
- Generated in seconds
- Pratical implementation

In the process of being merged into mainline Halide

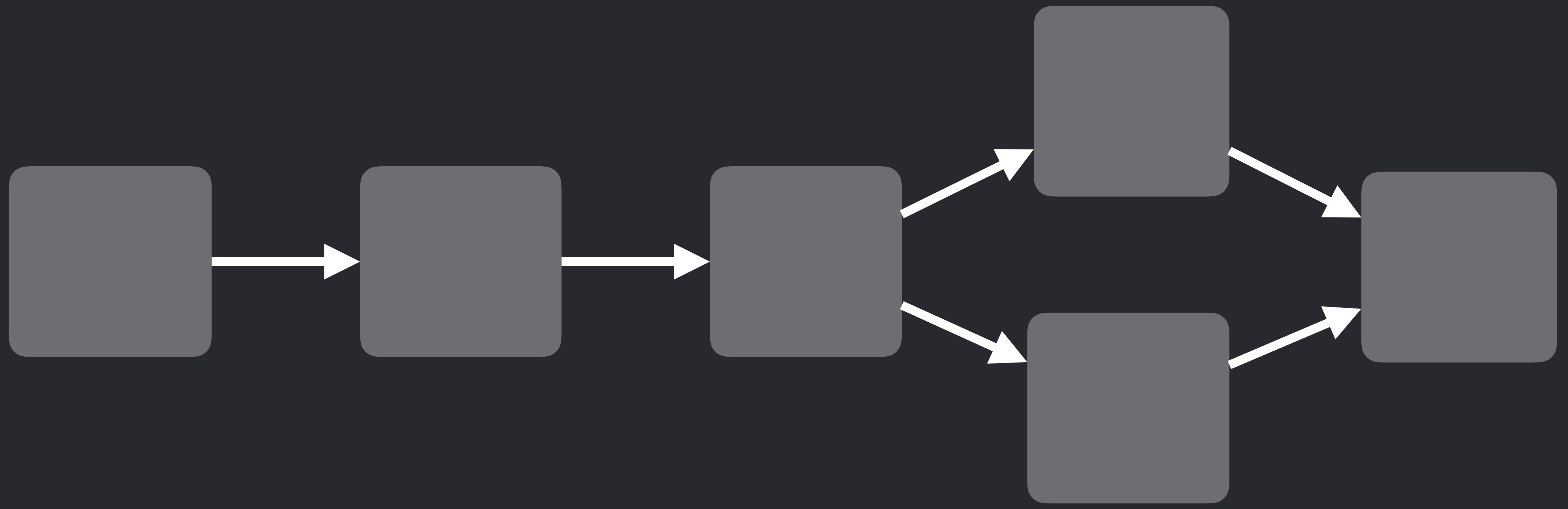
https://github.com/halide/Halide/tree/auto_scheduler

Generalizing the auto scheduler for other DSLs

Tensor Flow

Halide

Opt



Abstract analysis and scheduling techniques into components that can be used across languages

Thank you

https://github.com/halide/Halide/tree/auto_scheduler