University of Glasgow

Lift: Code Generation by Rewriting Algorithmic Skeletons

Michel Steuwer — *michel.steuwer@glasgow.ac.uk*
and the LIFT team

INSPIRING PEOPLE

1968 - Go To Statement Considered Harmful

EWD215 - 0

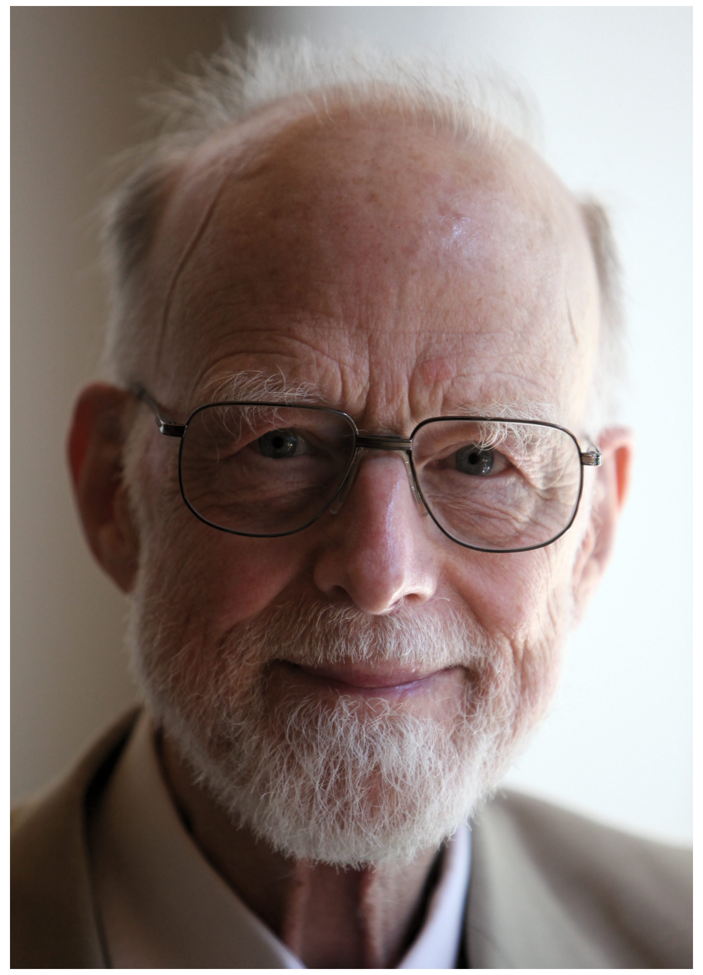
<u>A Case against the GO TO Statement.</u>

by Edsger W.Dijkstra
Technological University
Eindhoven, The Netherlands

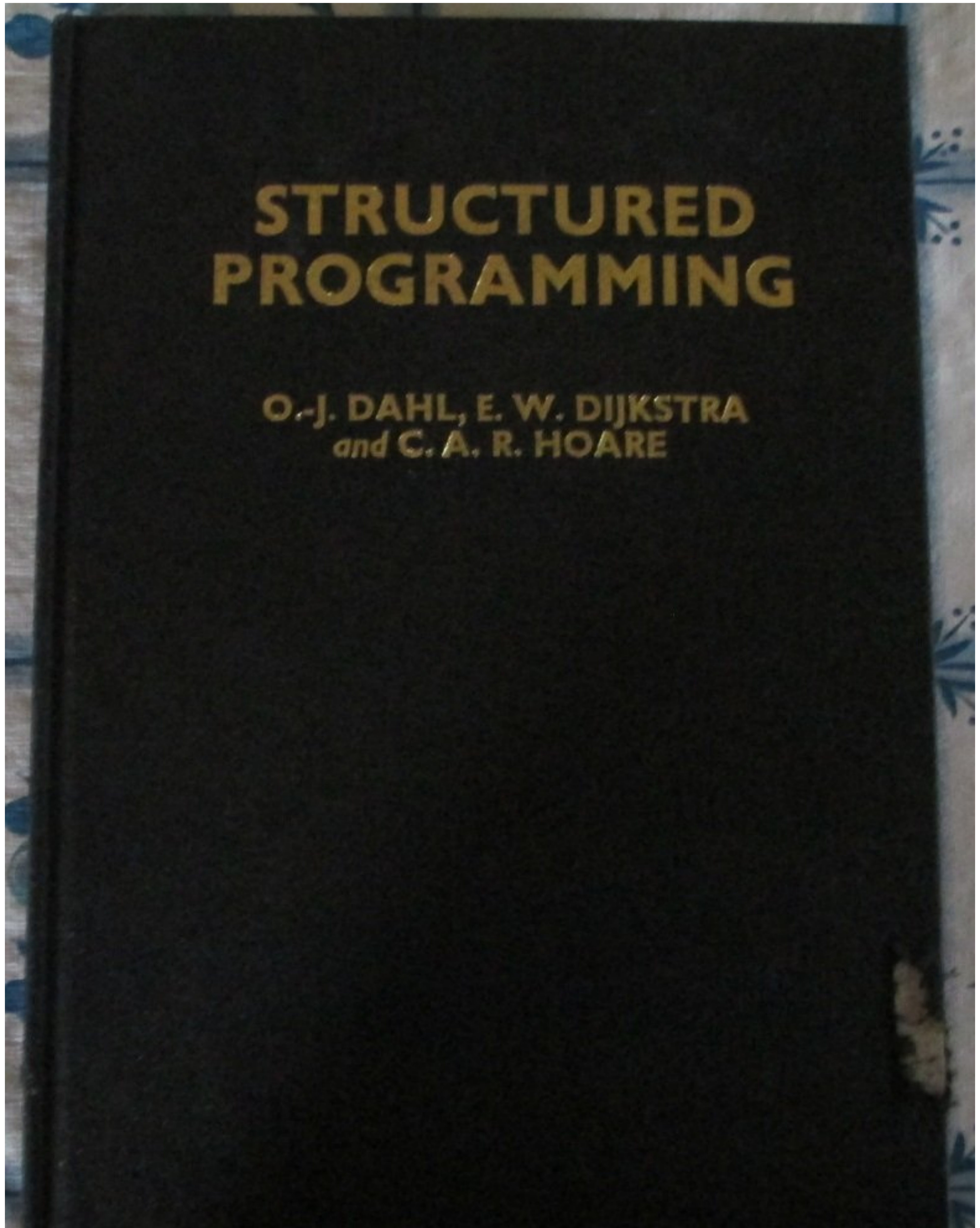
Since a number of years I am familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce. Later I discovered why the use of the go to statement has such disastrous effects and did I become convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except -perhaps- plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control

1972 - Structured Programming



```
procedure matmult (A, B, C, m, n, p);
        array A, B, C; integer m, n, p;
begin integer i, j, k;
                for i: = 1 step 1 until m do
                for j: = 1 step 1 until n do
                begin C[i, j]: = 0;
                        for k: = 1 step 1 until p do
                        C[i, j]: = C[i, j] + A[i, k] × B[k, j]
                end
end;
```



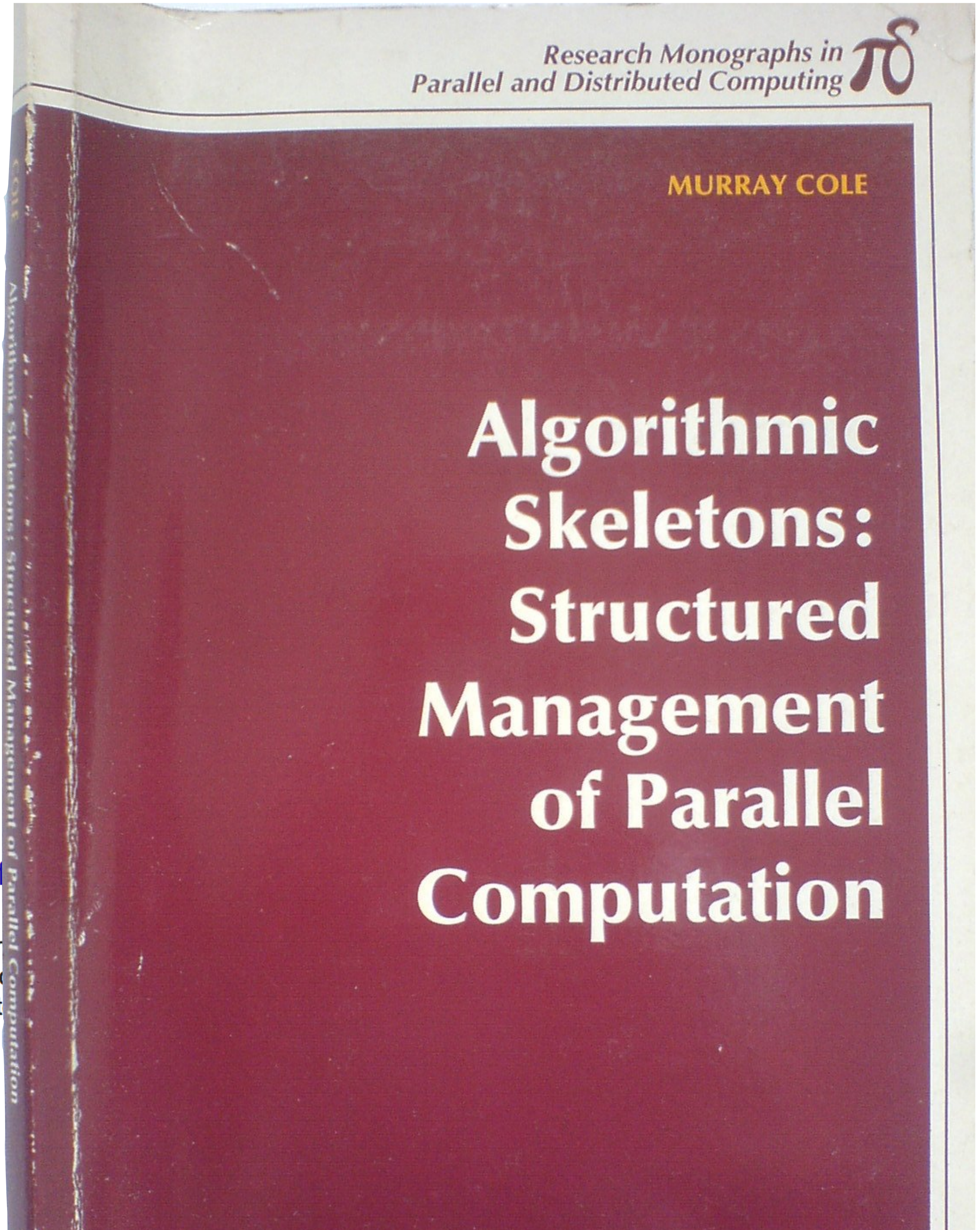
1989 - Structured *Parallel* Programming

$$
\begin{aligned}
D\_C\ indivisible\ split\ join\ f &= F \\
\text{where } F\ P &= f\ P,\ \text{ if } indivisible\ P \\
&= join\,(map\ F\ \ (split\ P))\,,\ \text{otherwise}
\end{aligned}
$$

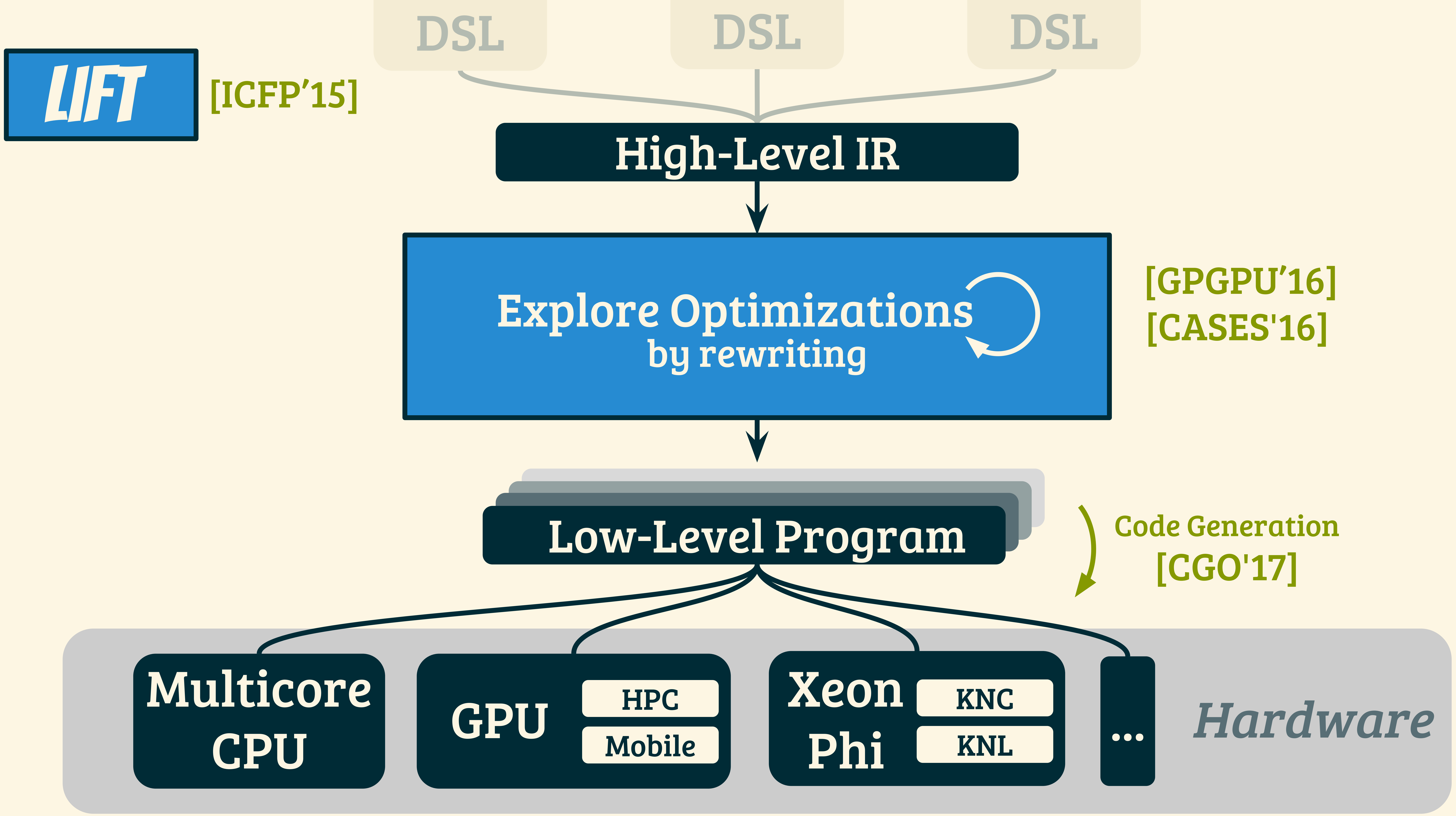
[BOOK] Algorithmic skeletons: structured management of

MI **Cole** - 1989 - homepages.inf.ed.ac.uk

Abstract In the past, most significant improvements in computer perform
achieved as a result of advances in simple device technology. The intro
scale parallelism at the inter-processor level now represents a viable alt

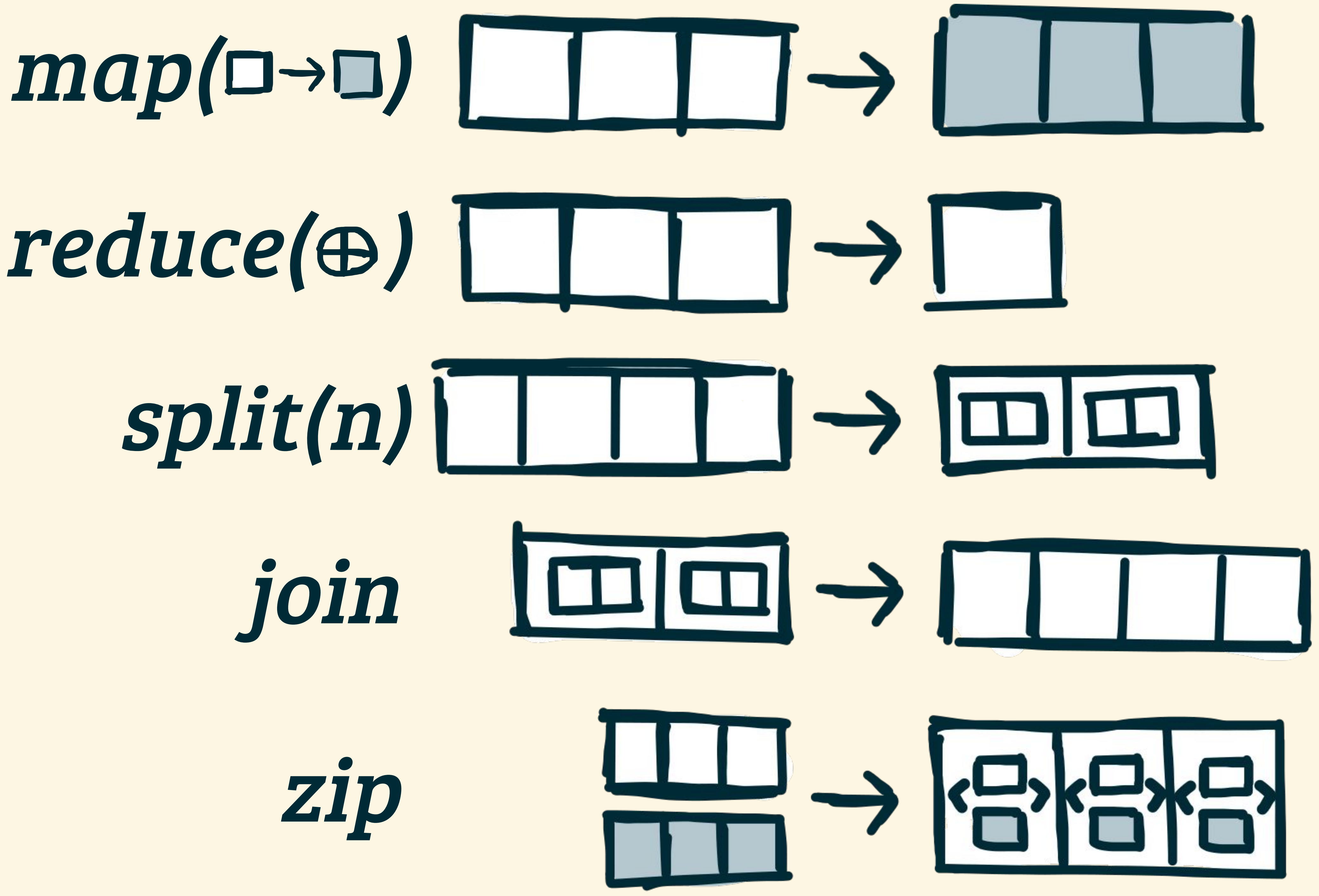



LIFT
2. HIGH-LEVEL PROGRAMMING
1. LOW-LEVEL OPTIMIZATIONS
G. HIGH PERFORMANCE

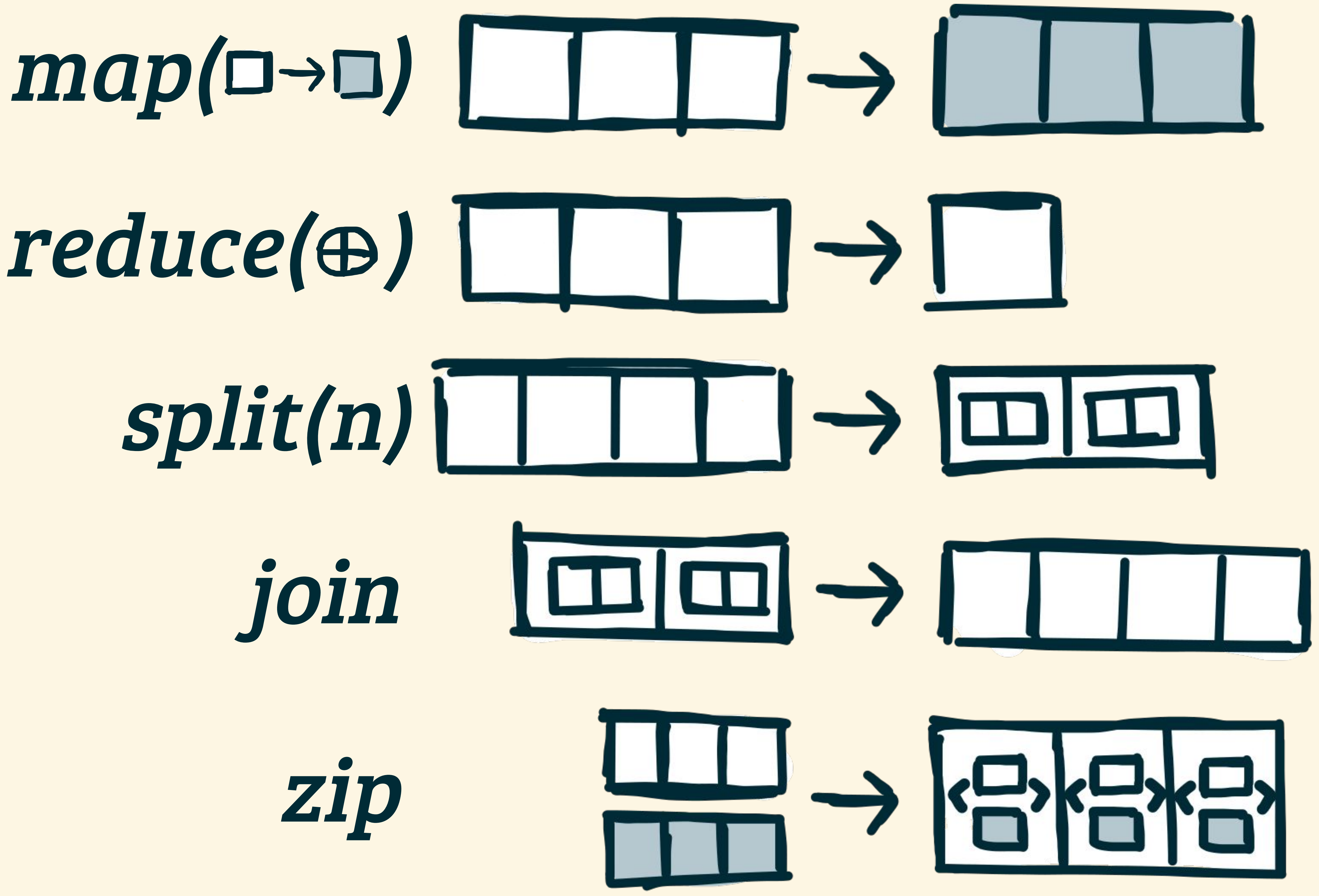


LIFT
[ICFP'15]
DSL
DSL
DSL
High-Level IR
Explore Optimizations
by rewriting
[GPGPU'16]
[CASES'16]
Low-Level Program
Code Generation
[CGO'17]
Multicore
CPU
GPU
HPC
Mobile
Xeon
Phi
KNC
KNL
...
Hardware

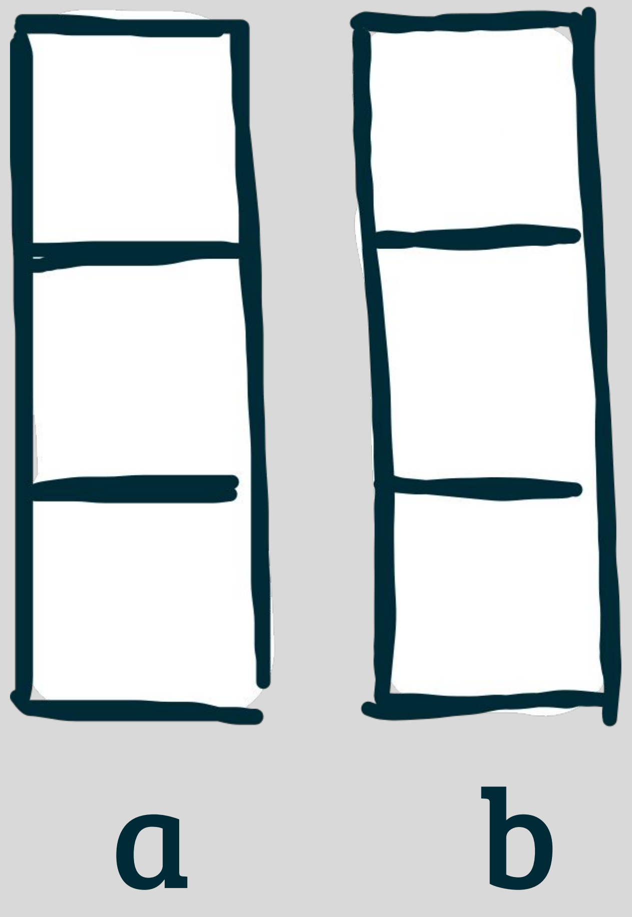
LIFT'S HIGH-LEVEL PRIMITIVES



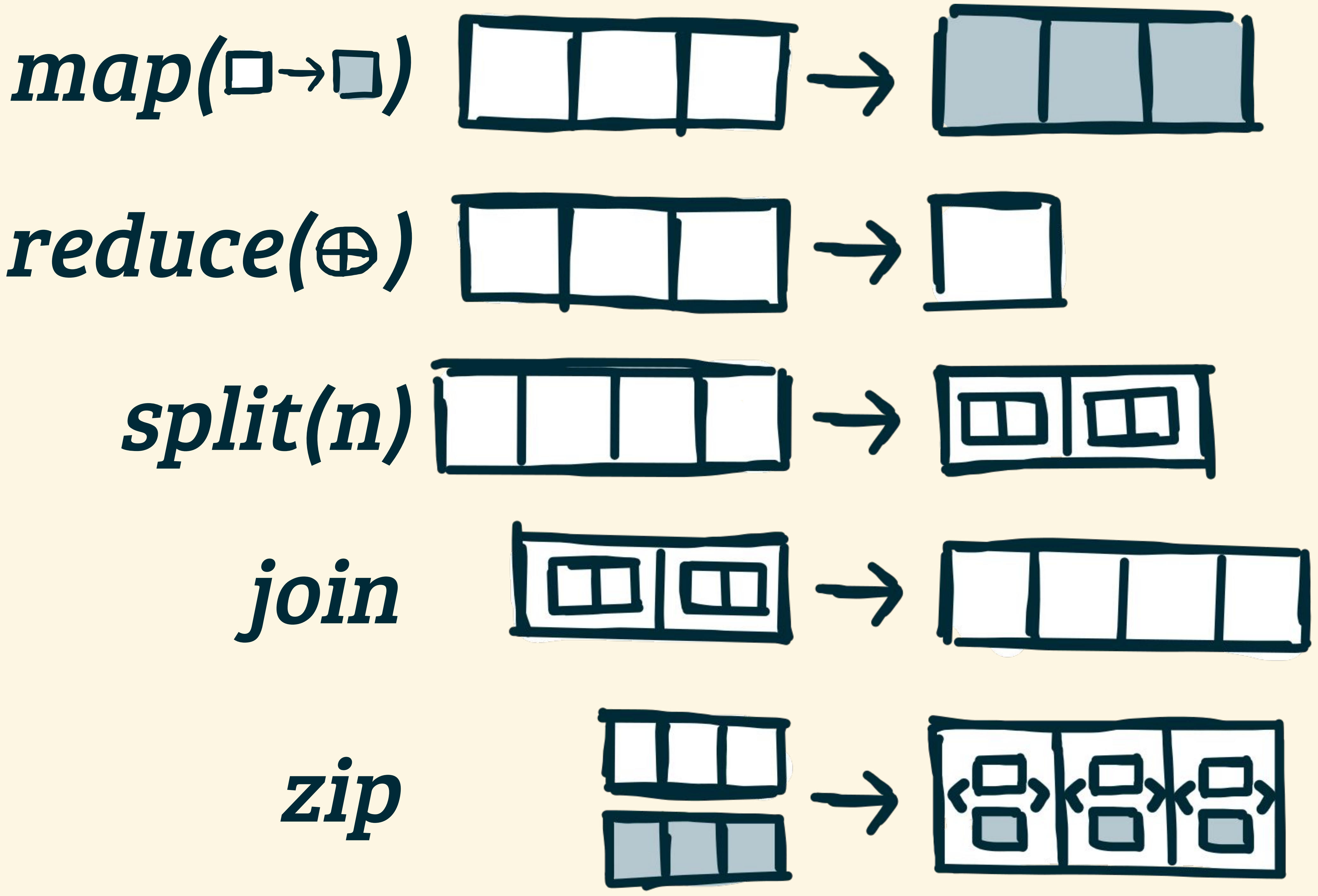
LIFT'S HIGH-LEVEL PRIMITIVES



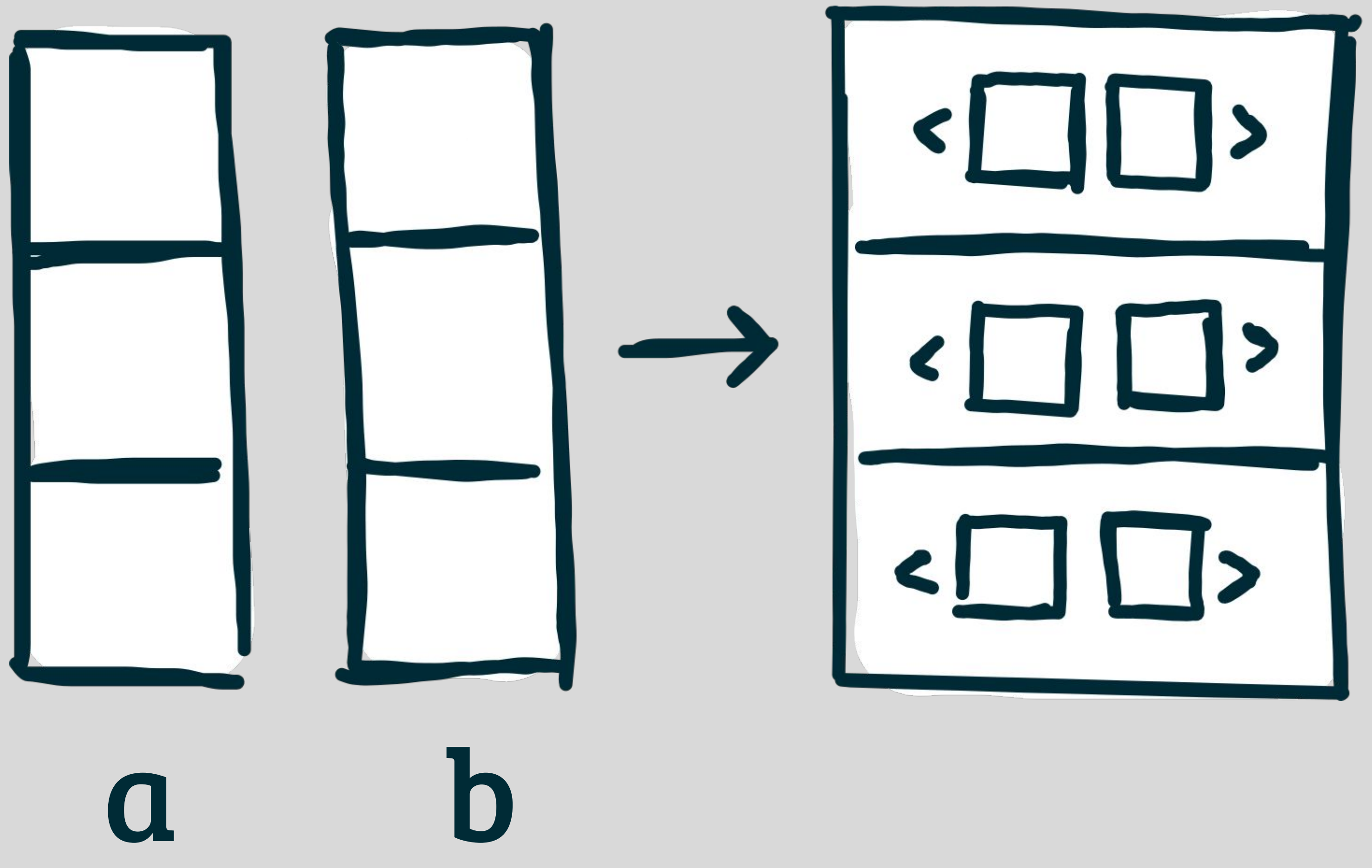
dotproduct.lift



LIFT'S HIGH-LEVEL PRIMITIVES

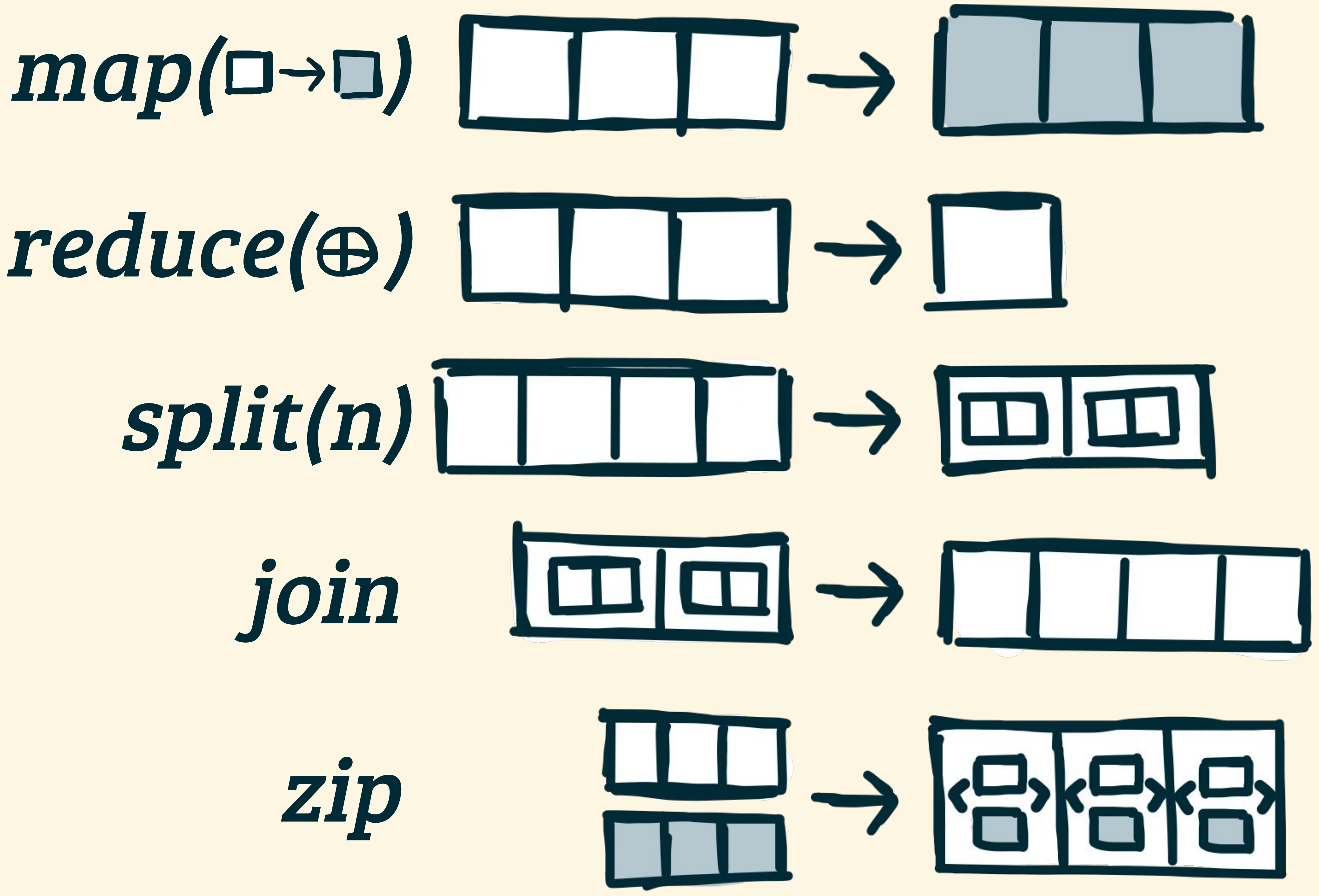


dotproduct.lift

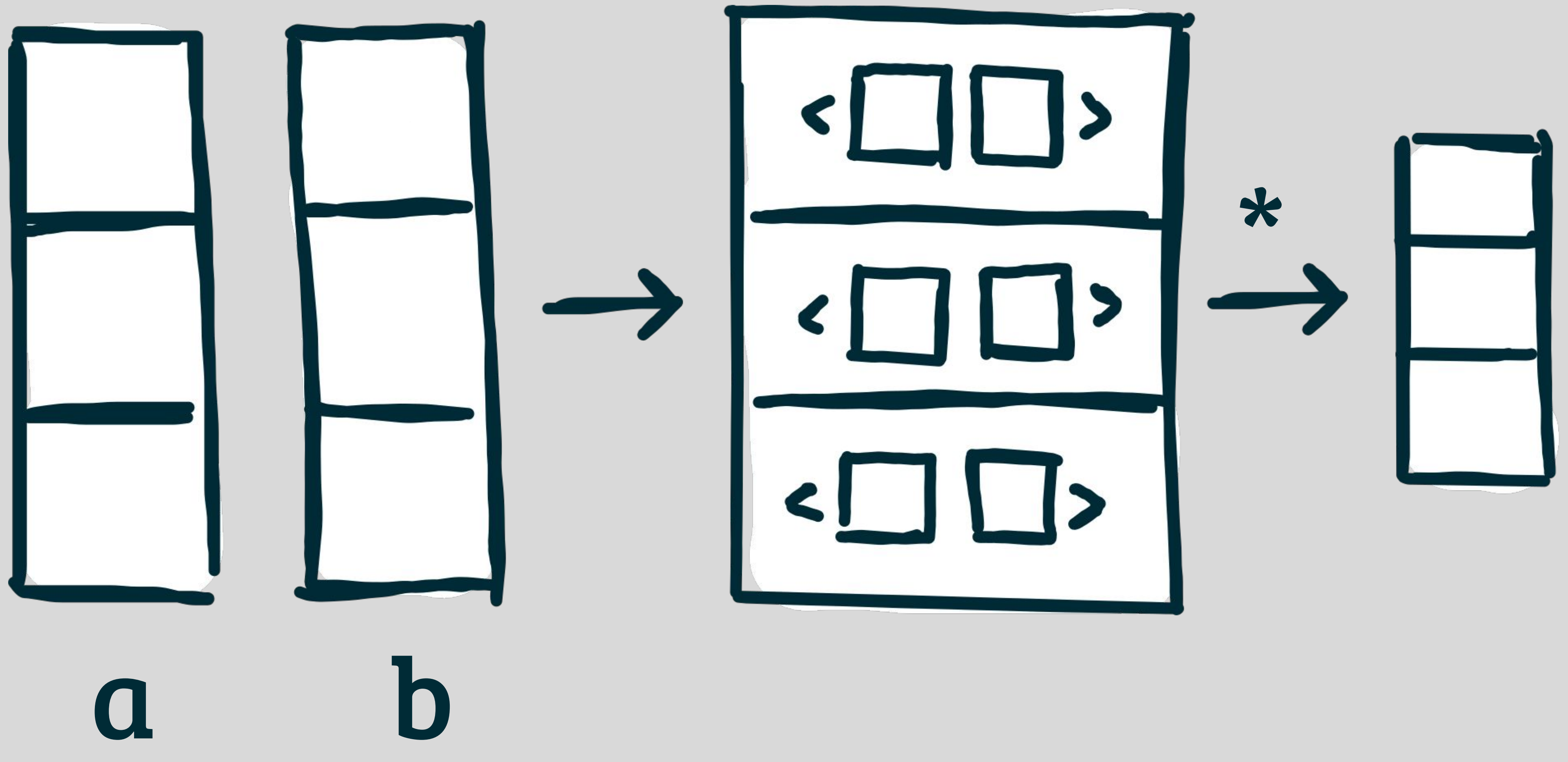


zip(a,b)

LIFT'S HIGH-LEVEL PRIMITIVES

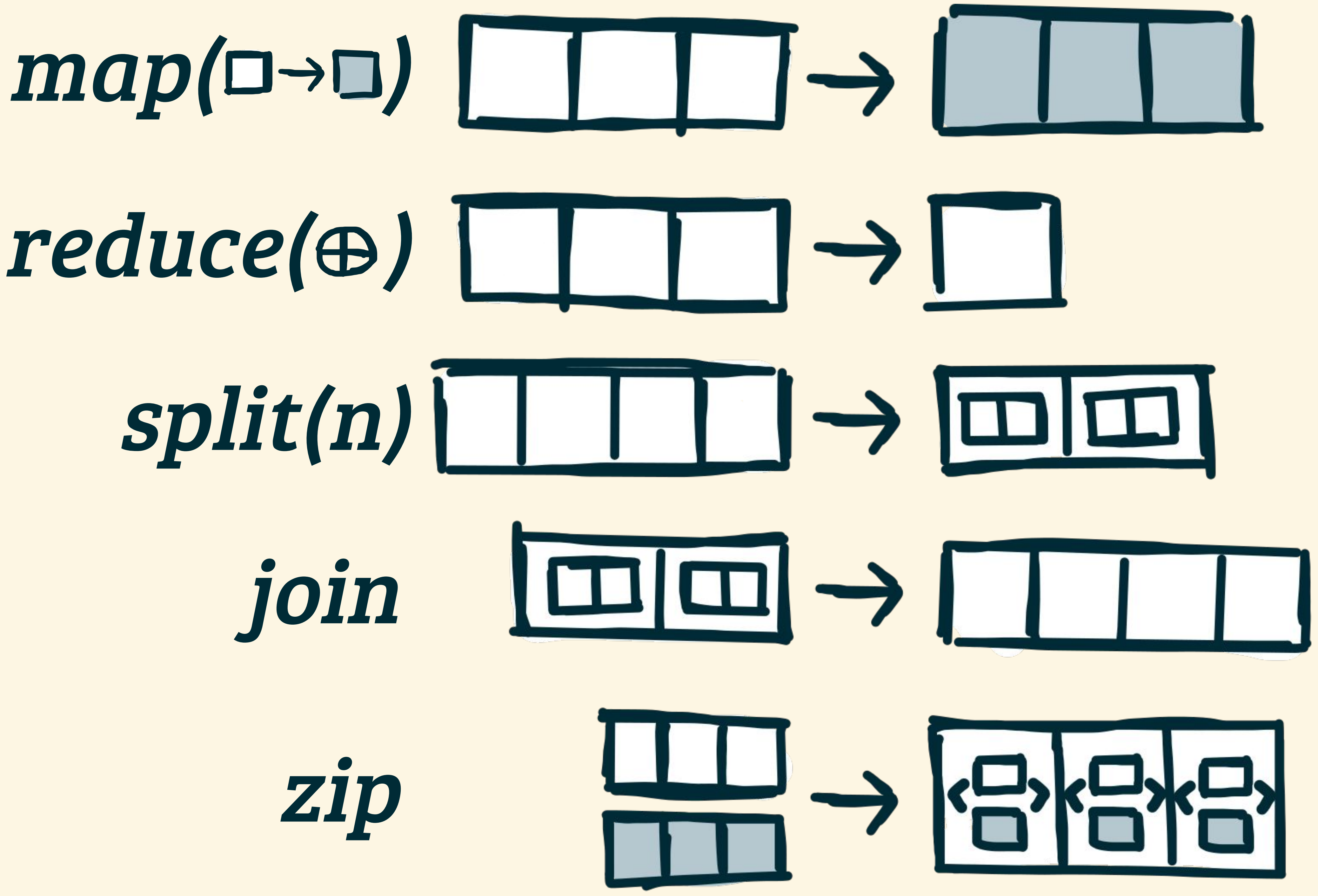


dotproduct.lift

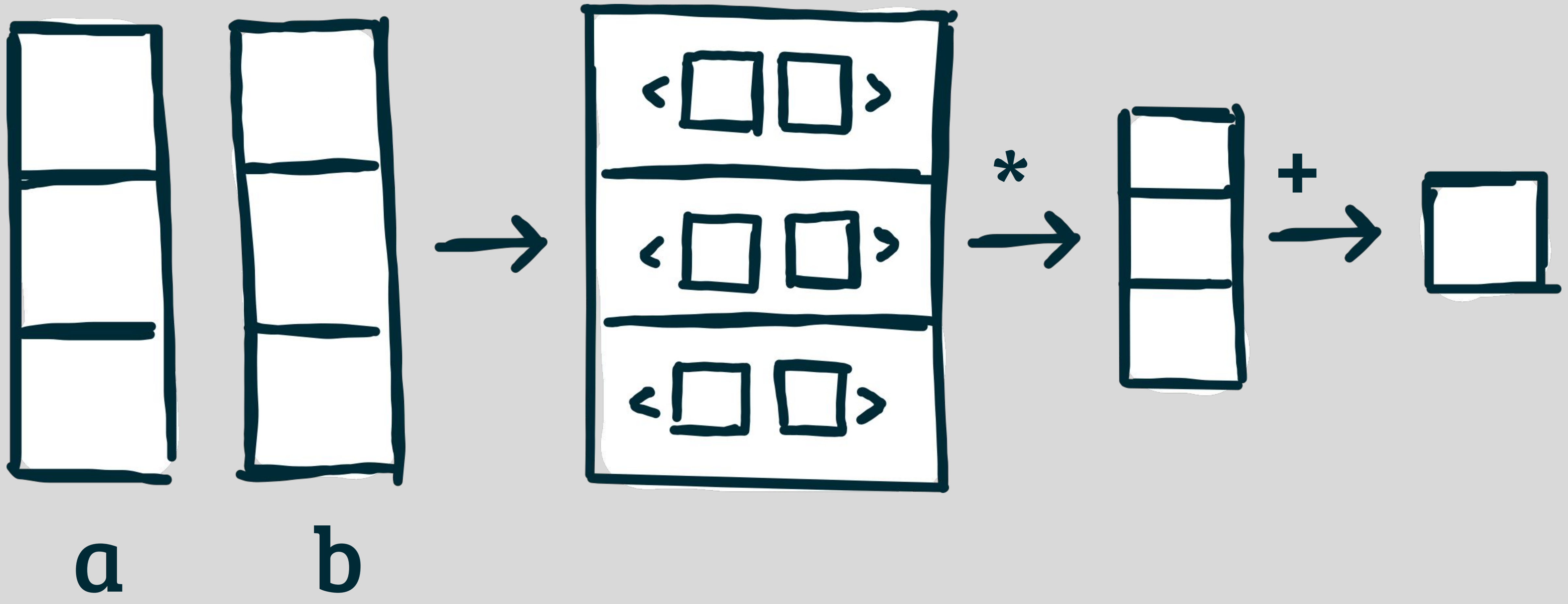



```
map(*, zip(a,b))
```

LIFT'S HIGH-LEVEL PRIMITIVES

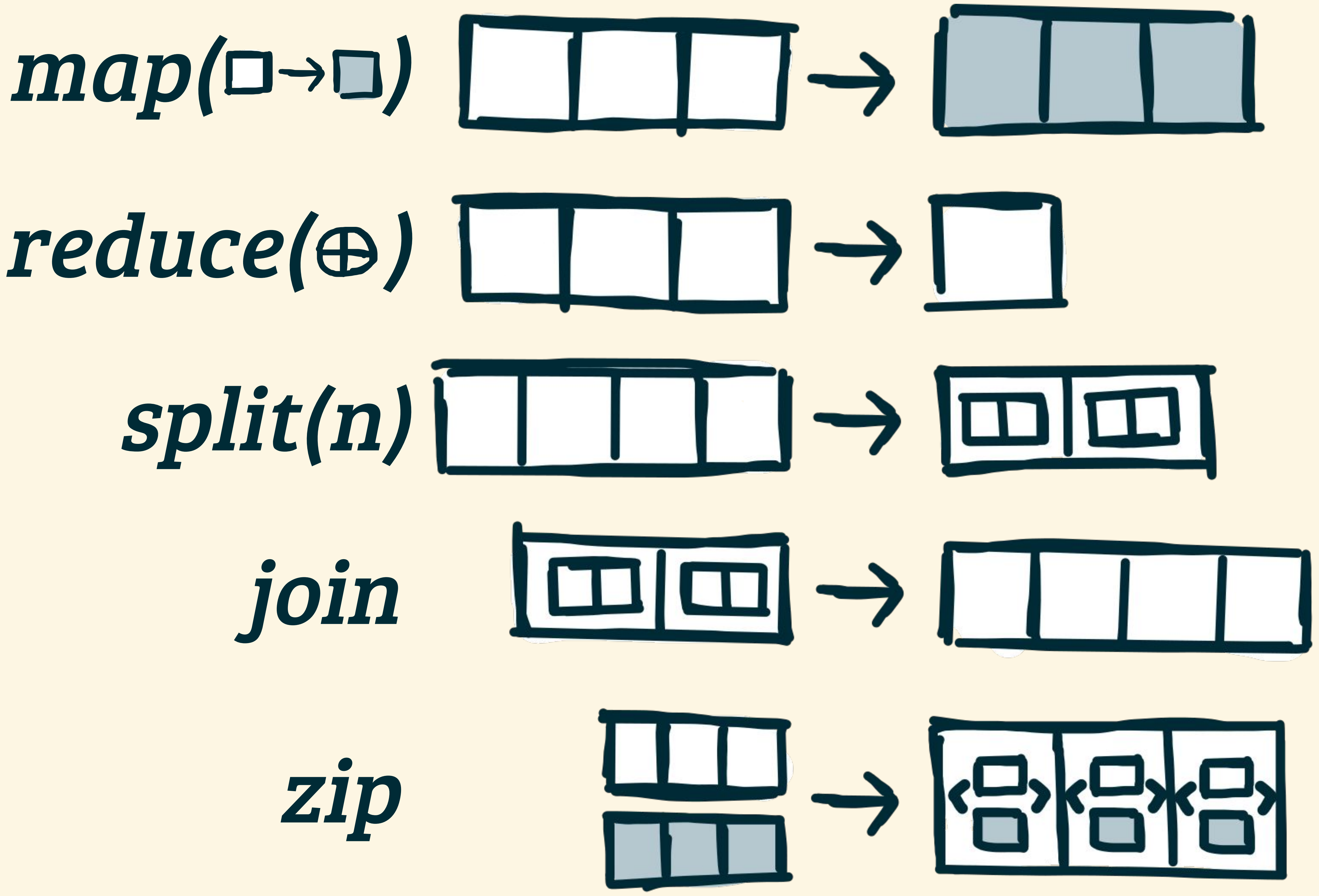


dotproduct.lift




```
reduce(+,0, map(*, zip(a,b)))
```

LIFT'S HIGH-LEVEL PRIMITIVES



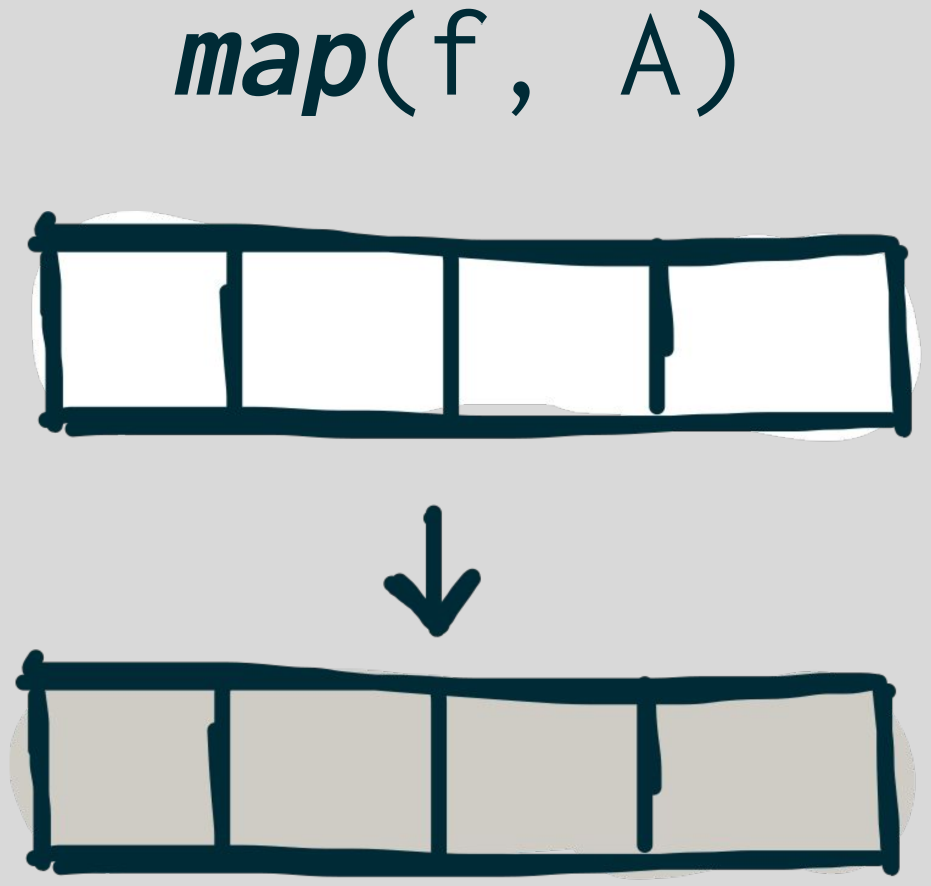
matrixMult.lift

```
map(λ rowA ↦
   map(λ colB ↦
      dotProduct(rowA, colB)
    , transpose(B))
  , A)
```


LIFT
2. HIGH-LEVEL PROGRAMMING
1. LOW-LEVEL OPTIMIZATIONS
G. HIGH PERFORMANCE

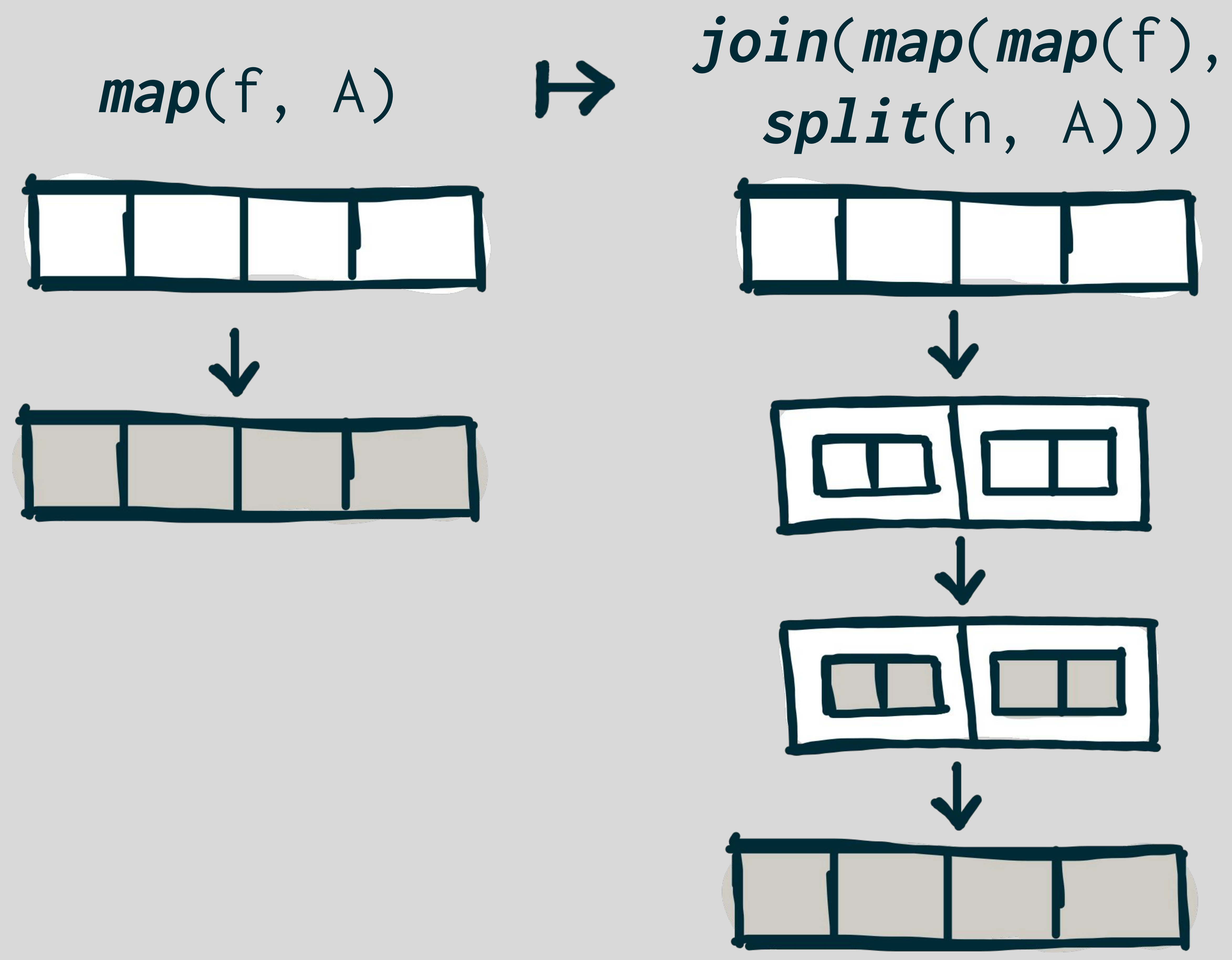
Implementation Choices as Rewrite Rules

Divide & Conquer



Implementation Choices as Rewrite Rules

Divide & Conquer



LIFT'S LOW LEVEL (OPENCL) PRIMITIVES

Lift primitive	OpenCL concept
mapGlobal	Work-items
mapWorkgroup mapLocal	Work-groups
mapSeq reduceSeq	Sequential implementations
toLocal, toGlobal	Memory areas
mapVec, splitVec, joinVec	Vectorisation

Rewriting into OpenCL

Map rules:

map f ↦ mapGlobal f | mapWorkgroup f | mapLocal f | mapSeq f

Local / global memory:

mapLocal f ↦ toLocal (mapLocal f)   mapLocal f ↦ toGlobal (mapLocal f)

Vectorization:

map f ↦ joinVec ○ map (mapVec f) ○ splitVec n

OPTIMIZATIONS AS MACRO RULES

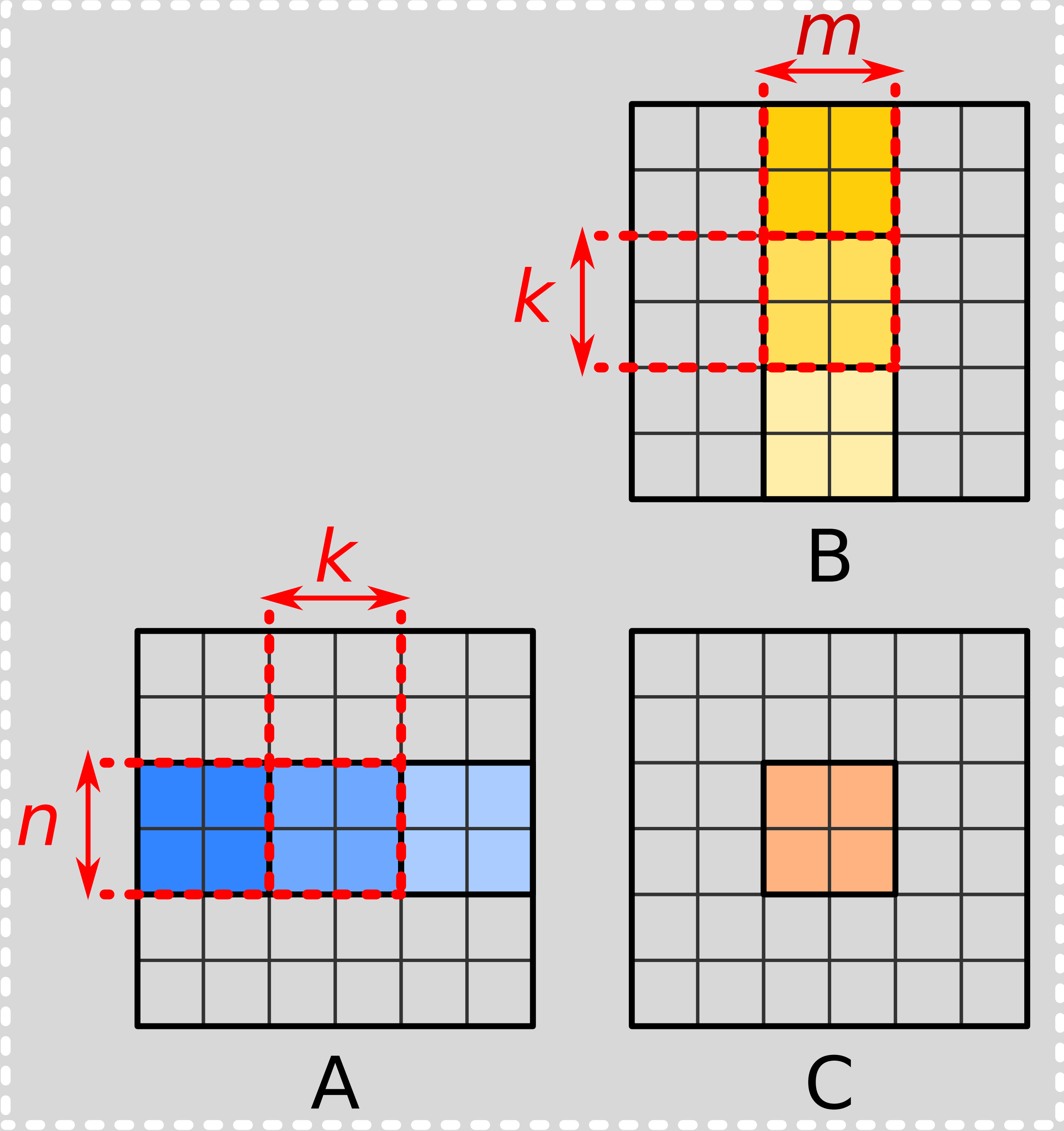
2D Tiling

Naïve matrix multiplication

```
map(λ arow .
  map(λ bcol .
    reduce(+, 0) ∘ map(×) ∘ zip(arow, bcol)
  , transpose(B))
, A)
```

↓ Apply tiling rules

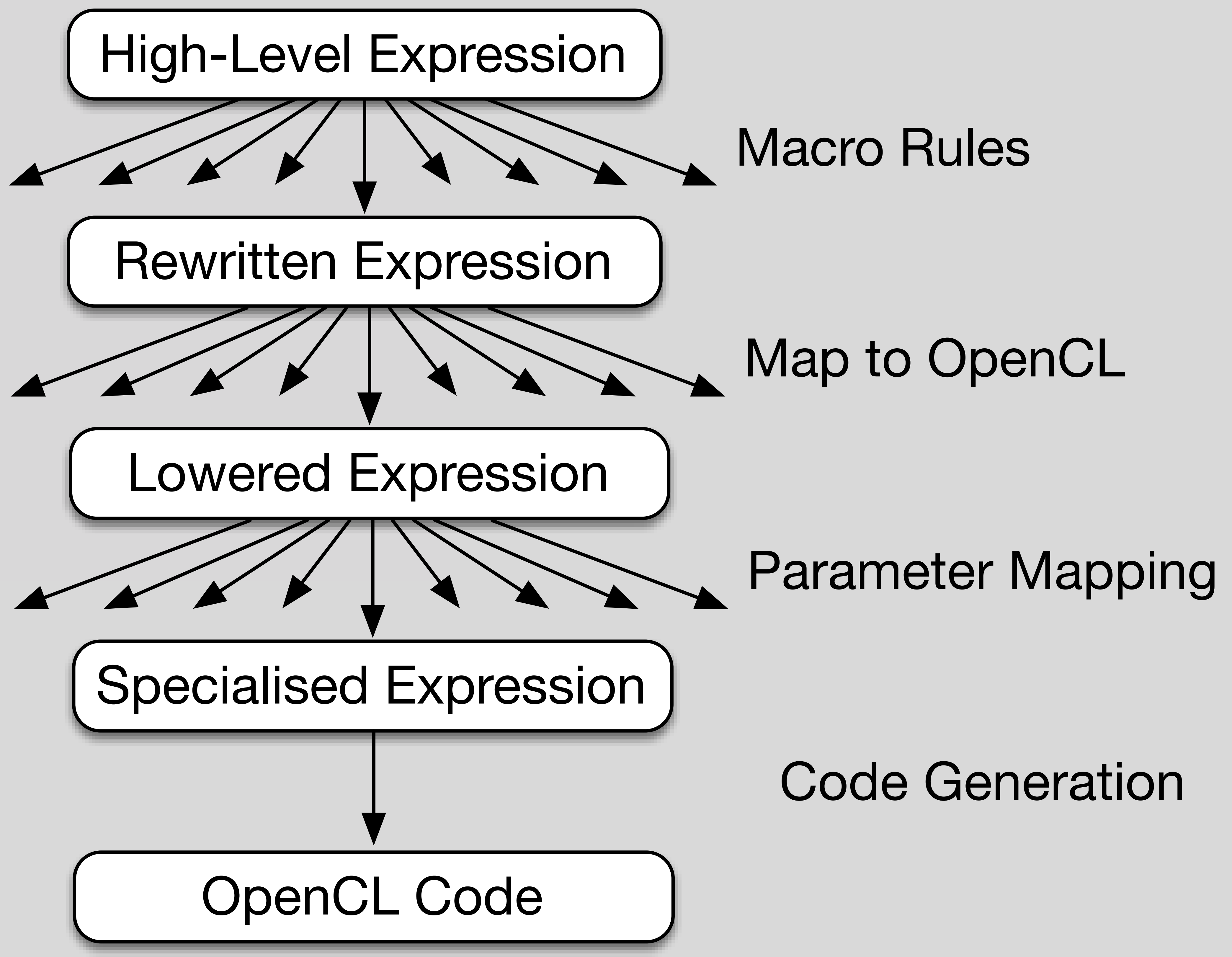
```
untile ∘ map(λ rowOfTilesA .
  map(λ colOfTilesB .
    toGlobal(copy2D) ∘
    reduce(λ (tileAcc, (tileA, tileB)) .
      map(map(+)) ∘ zip(tileAcc) ∘
      map(λ as .
        map(λ bs .
          reduce(+, 0) ∘ map(×) ∘ zip(as, bs)
        , toLocal(copy2D(tileB)))
      , toLocal(copy2D(tileA)))
    ,0, zip(rowOfTilesA, colOfTilesB))
  ) ∘ tile(m, k, transpose(B))
) ∘ tile(n, k, A)
```



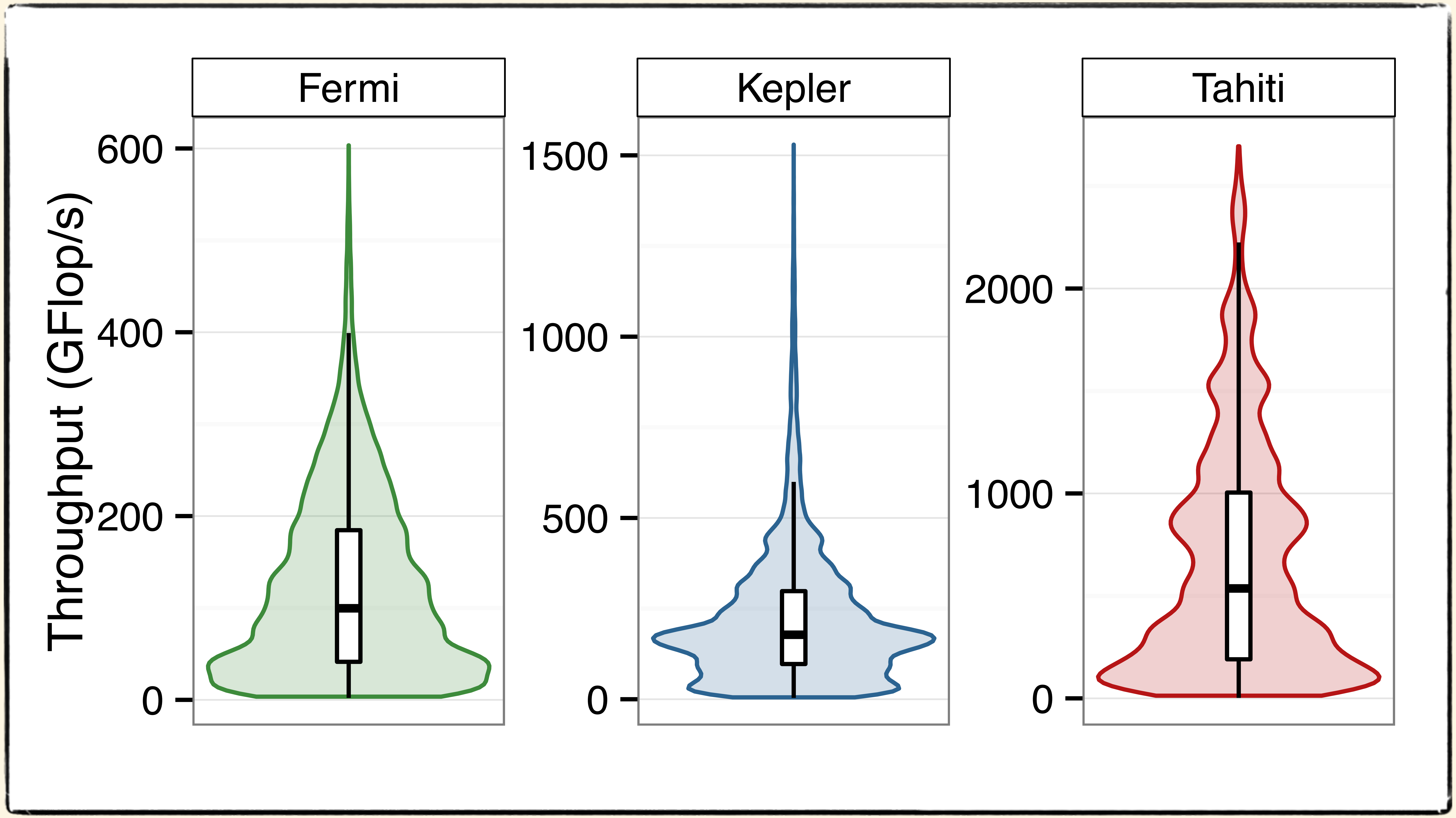
[GPGPU'16]


LIFT
2. HIGH-LEVEL PROGRAMMING
1. LOW-LEVEL OPTIMIZATIONS
G. HIGH PERFORMANCE

Exploration by Rewriting



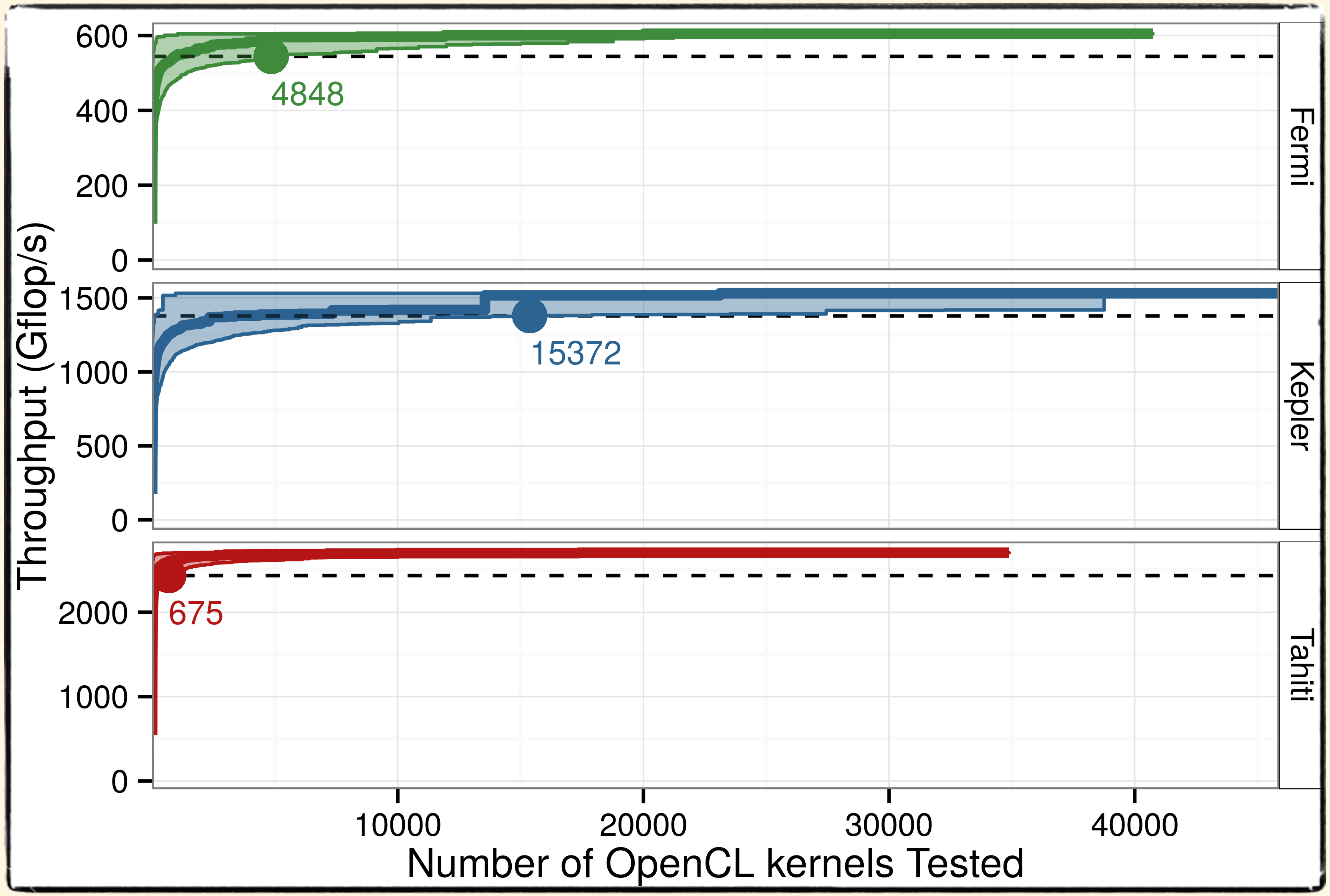
Exploration space matrix multiplication



Only few generated code with very good performance

[GPGPU'16]

Even randomised search works well!



Still: One can expect to find a good performing kernel quickly!

[GPGPU'16]

Performance Results Matrix Multiplication

Performance close or better than hand-tuned MAGMA library

[GPGPU'16]

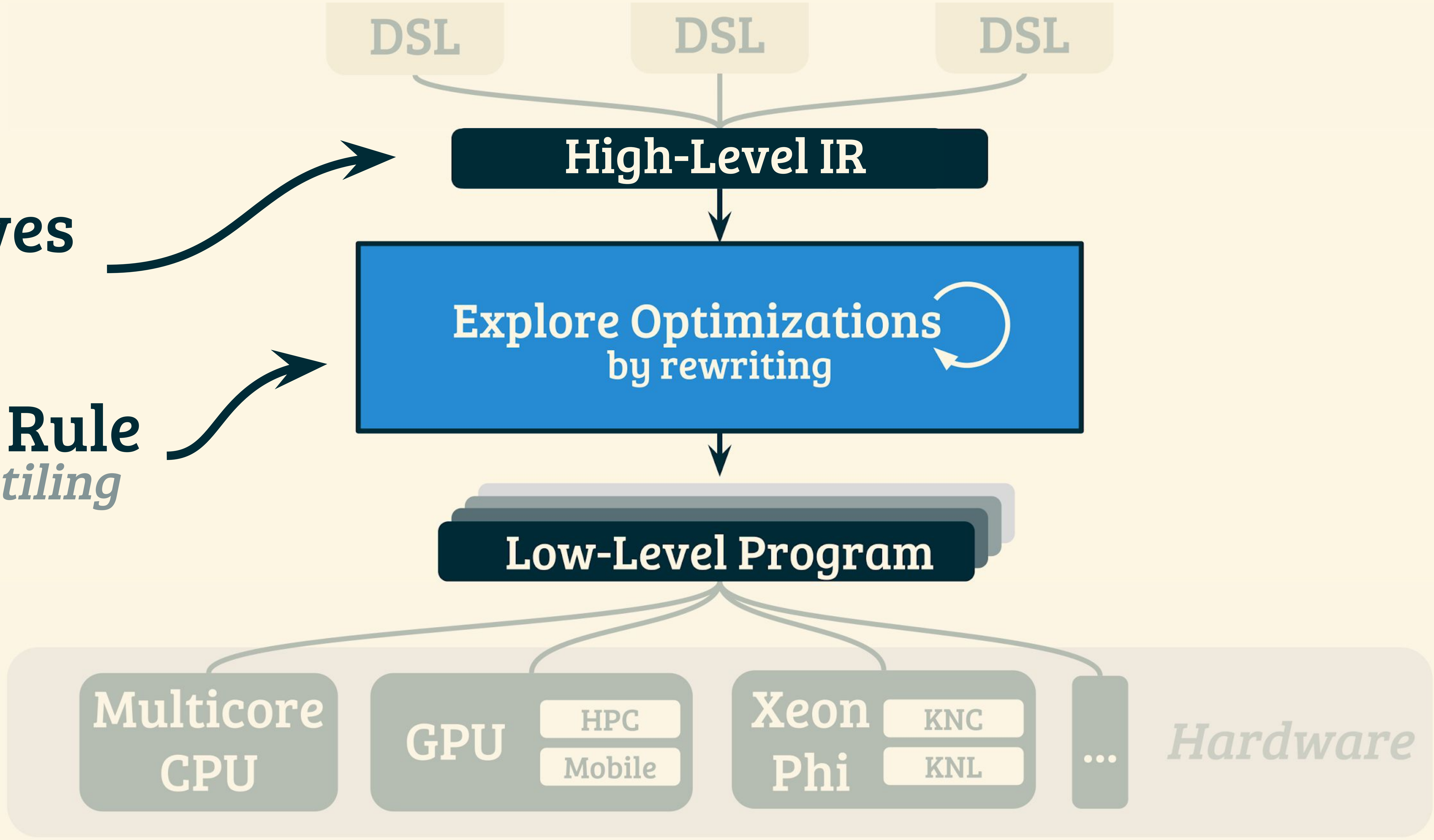
STENCIL COMPUTATIONS IN LIFT

[CGO'18] Best Paper Award

We added:

2 Primitives
pad, *slide*

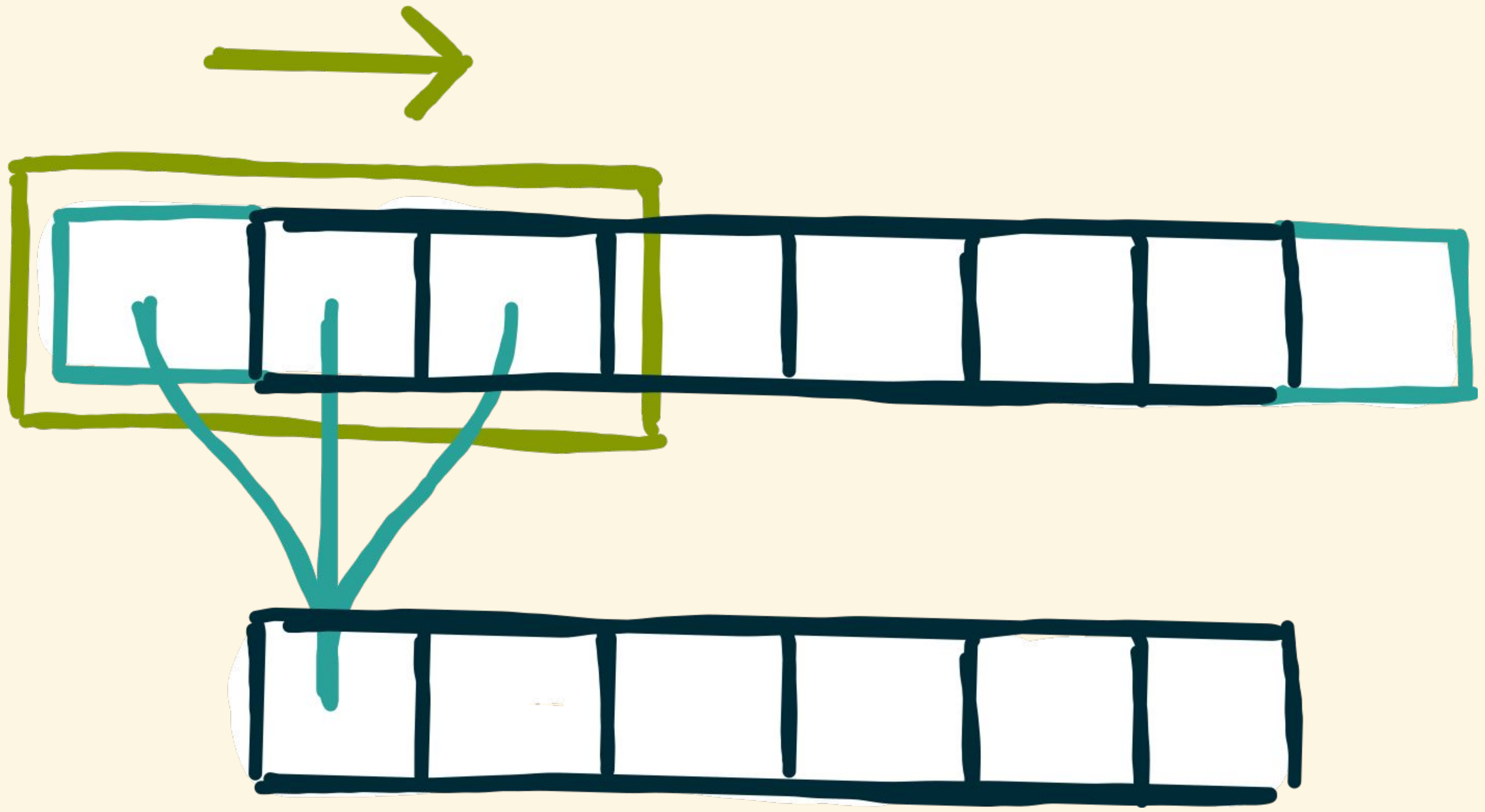
1 Rewrite Rule
overlapped tiling



DECOMPOSING STENCIL COMPUTATIONS

3-point-stencil.c

```
for (int i = 0; i < N ; i ++) {
    int sum = 0;
    for ( int j = -1; j <= 1; j ++) {   // ( a )
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[ pos ]; }
 B[ i ] = sum ; }
```

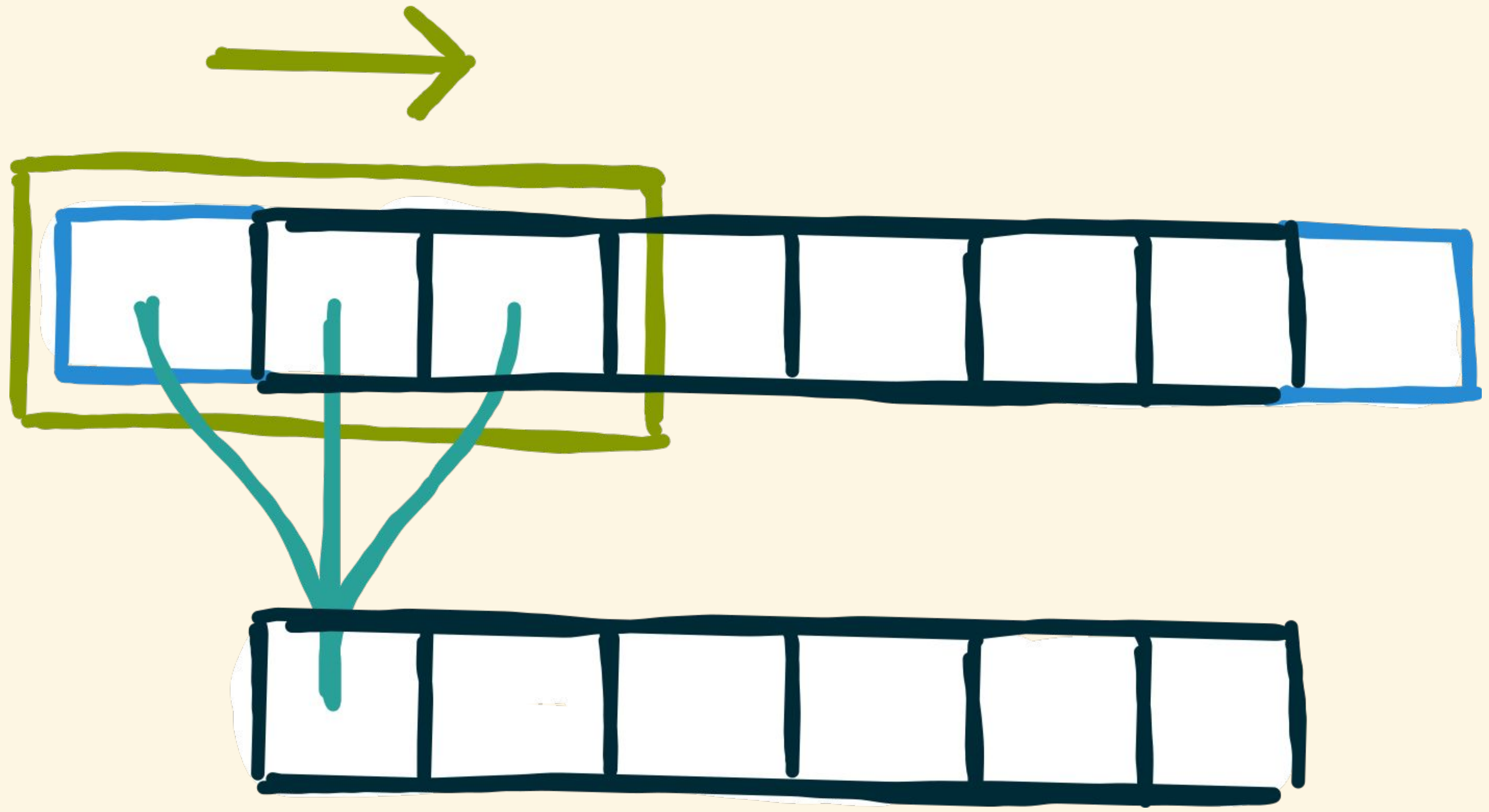


(a) access neighborhoods for every element

DECOMPOSING STENCIL COMPUTATIONS

3-point-stencil.c

```
for (int i = 0; i < N ; i ++) {
    int sum = 0;
    for ( int j = -1; j <= 1; j ++) {   // ( a )
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;        // ( b )
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[ pos ]; }
 B[ i ] = sum ; }
```



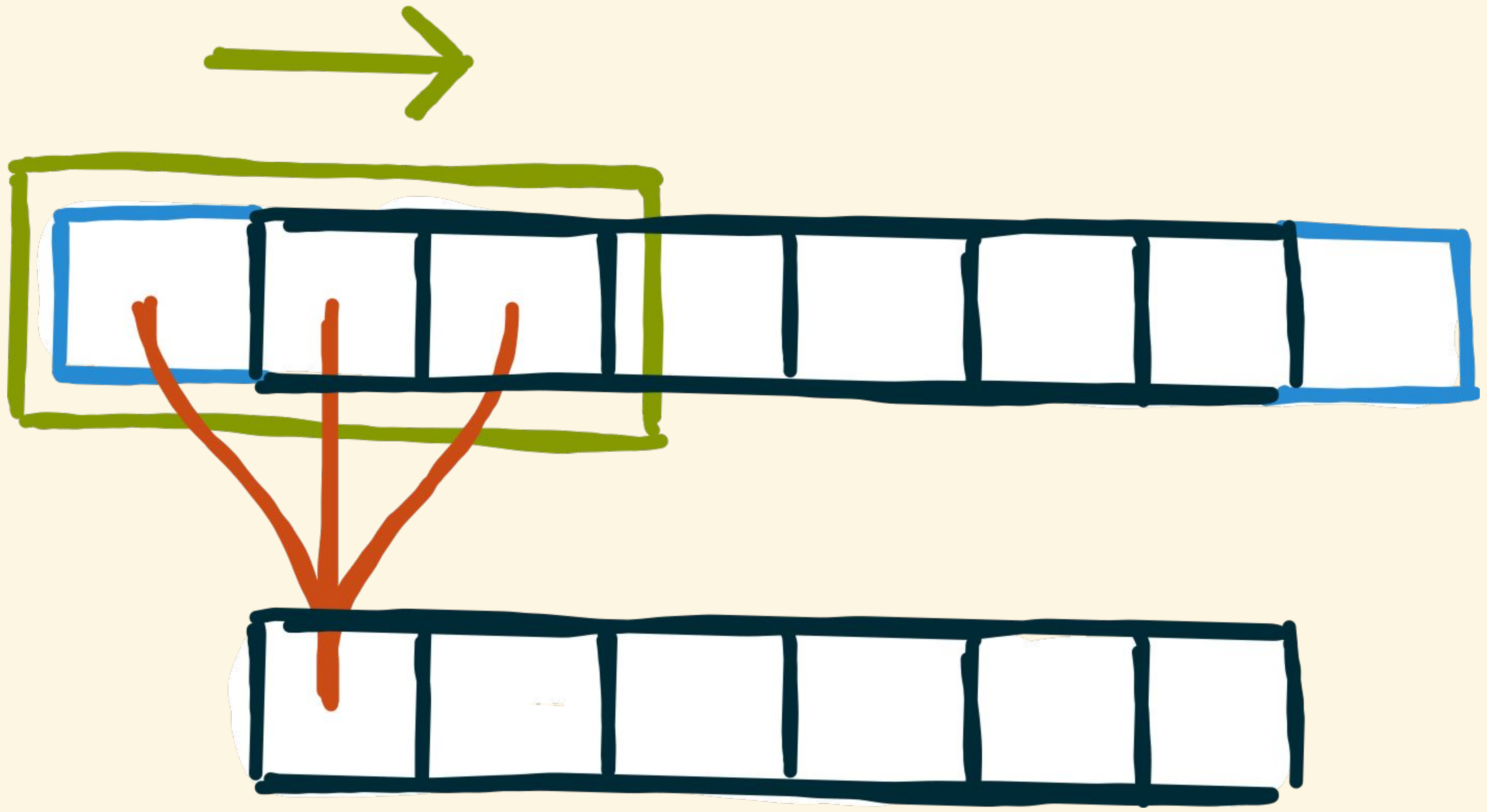
(a) access neighborhoods for every element

(b) specify boundary handling

DECOMPOSING STENCIL COMPUTATIONS

3-point-stencil.c

```
for (int i = 0; i < N ; i ++) {
    int sum = 0;
    for ( int j = -1; j <= 1; j ++) {    // ( a )
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;         // ( b )
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[ pos ]; }               // ( c )
 B[ i ] = sum ; }
```



(a) access **neighborhoods** for every element

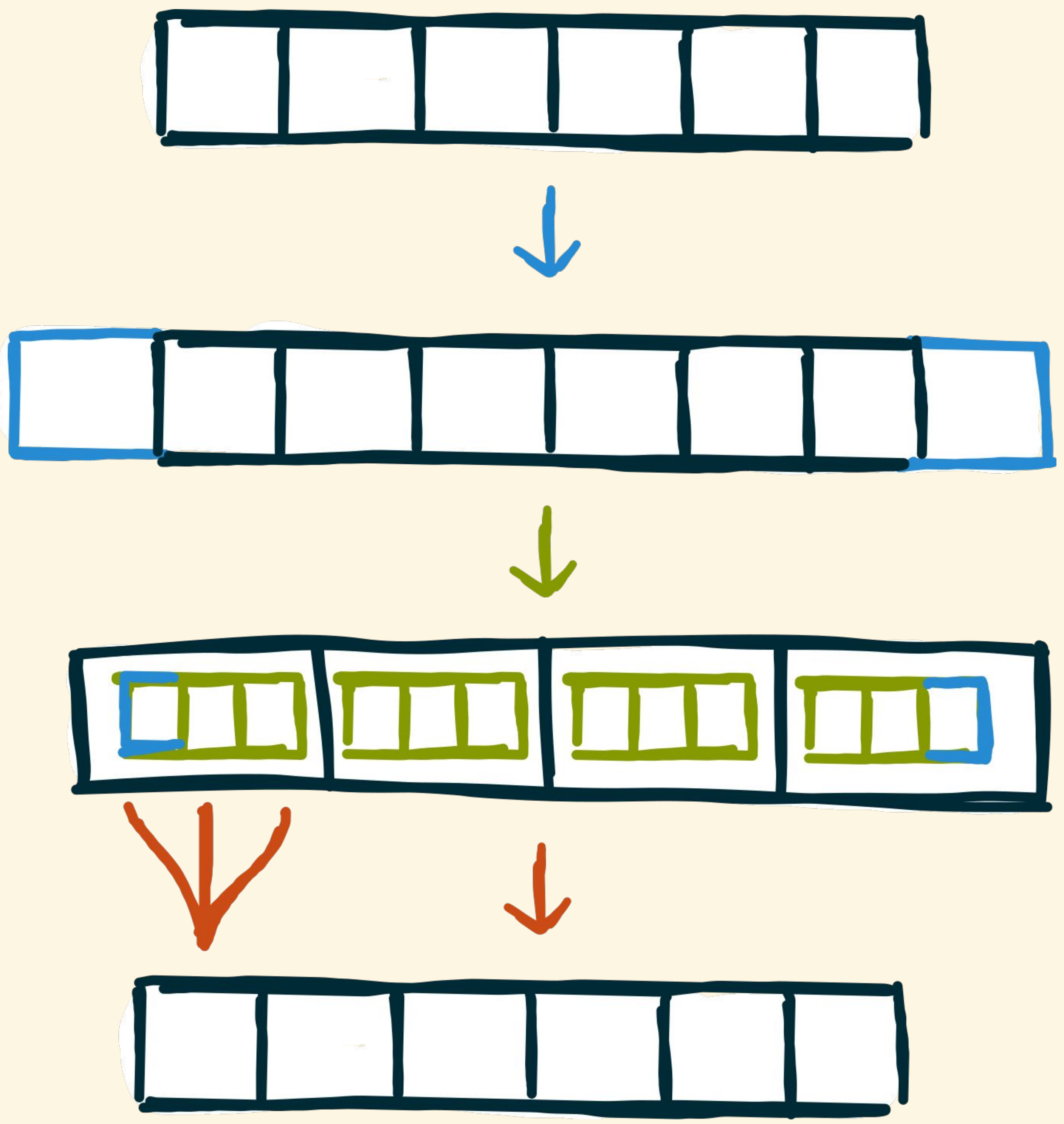
(b) specify **boundary handling**

(c) apply **stencil function** to neighborhoods

DECOMPOSING STENCIL COMPUTATIONS

3-point-stencil.c

```
for (int i = 0; i < N ; i ++) {
    int sum = 0;
    for ( int j = -1; j <= 1; j ++) {   // ( a )
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;        // ( b )
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[ pos ]; }              // ( c )
 B[ i ] = sum ; }
```

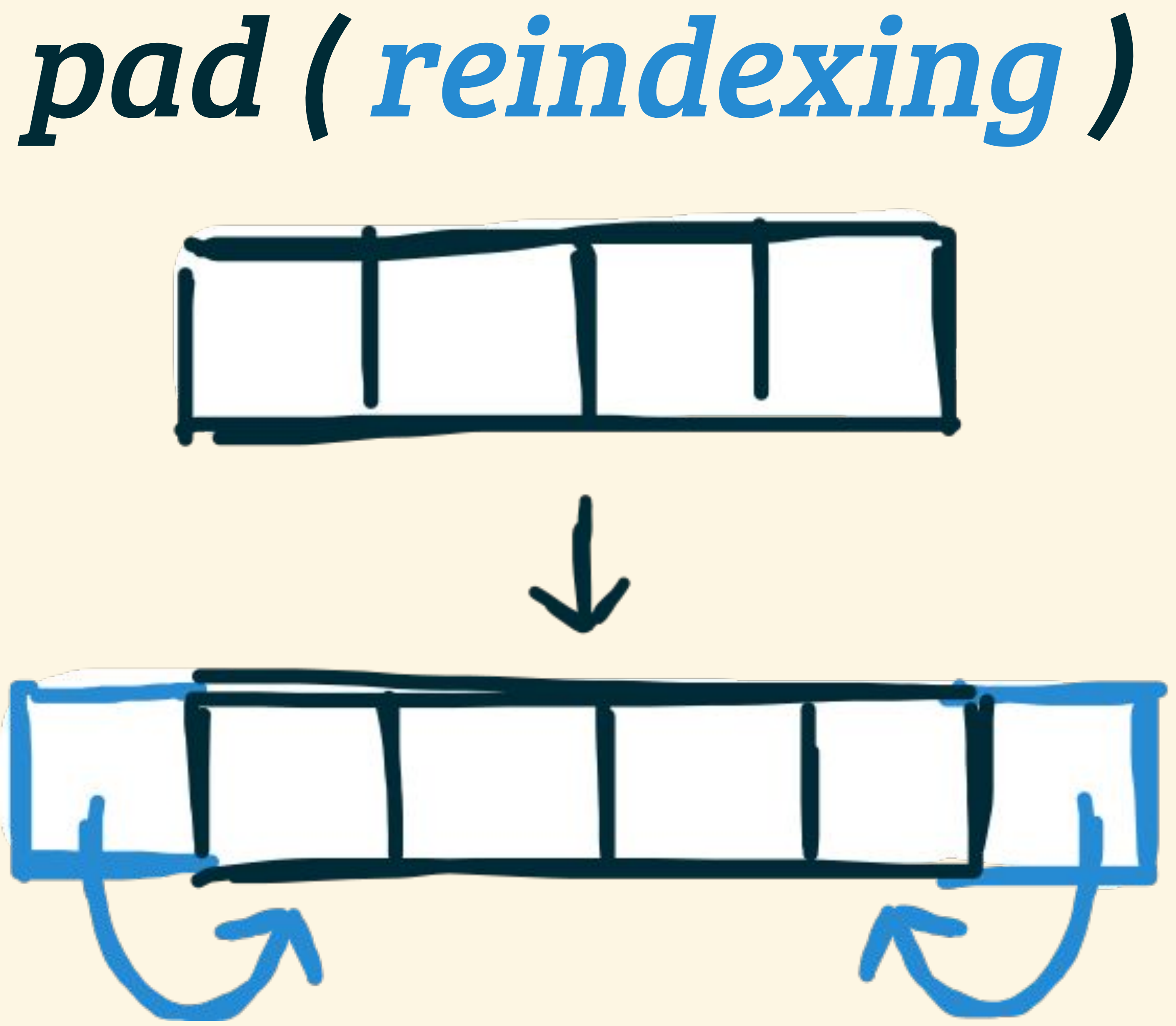


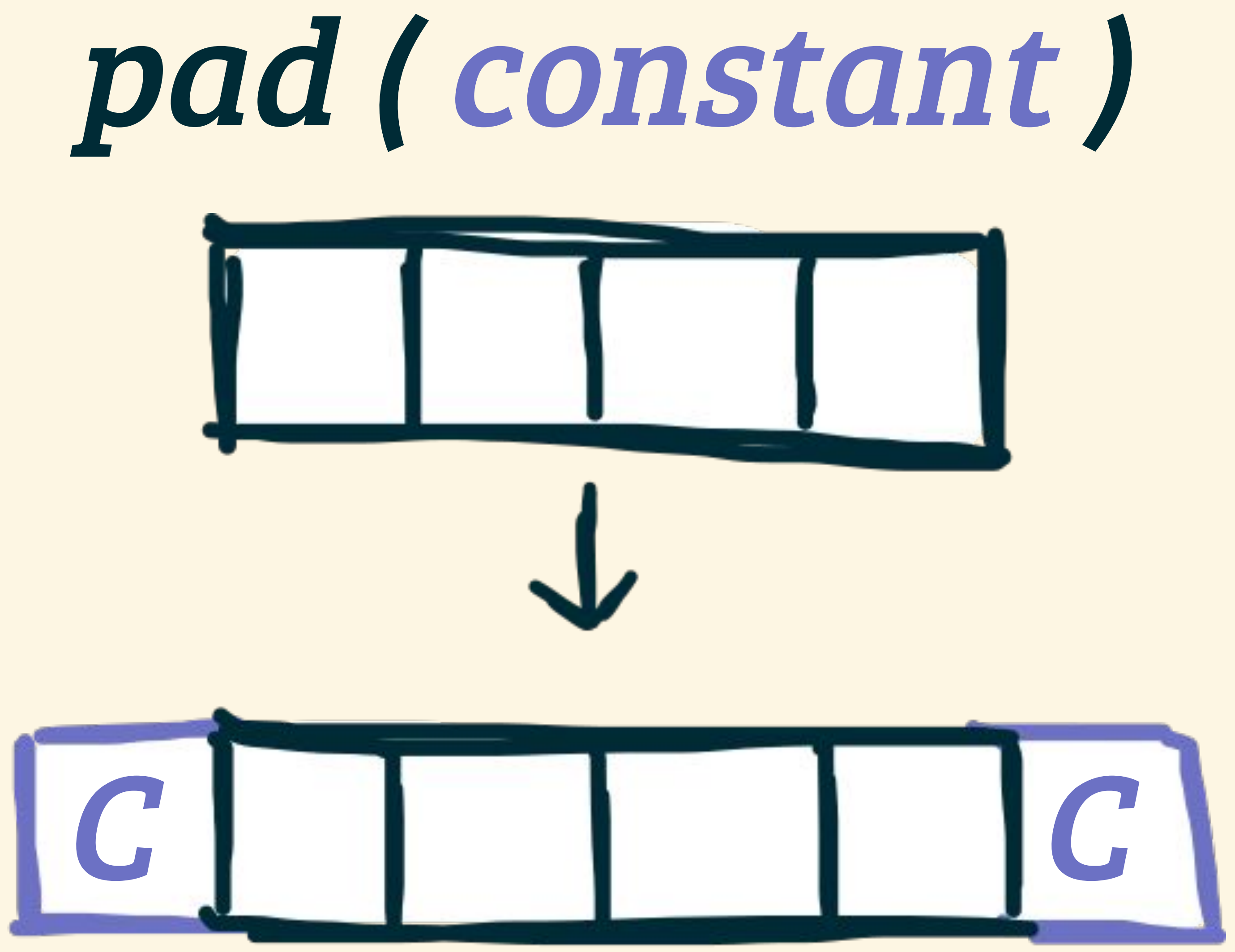
(a) access neighborhoods for every element

(b) specify boundary handling

(c) apply stencil function to neighborhoods

BOUNDARY HANDLING USING PAD





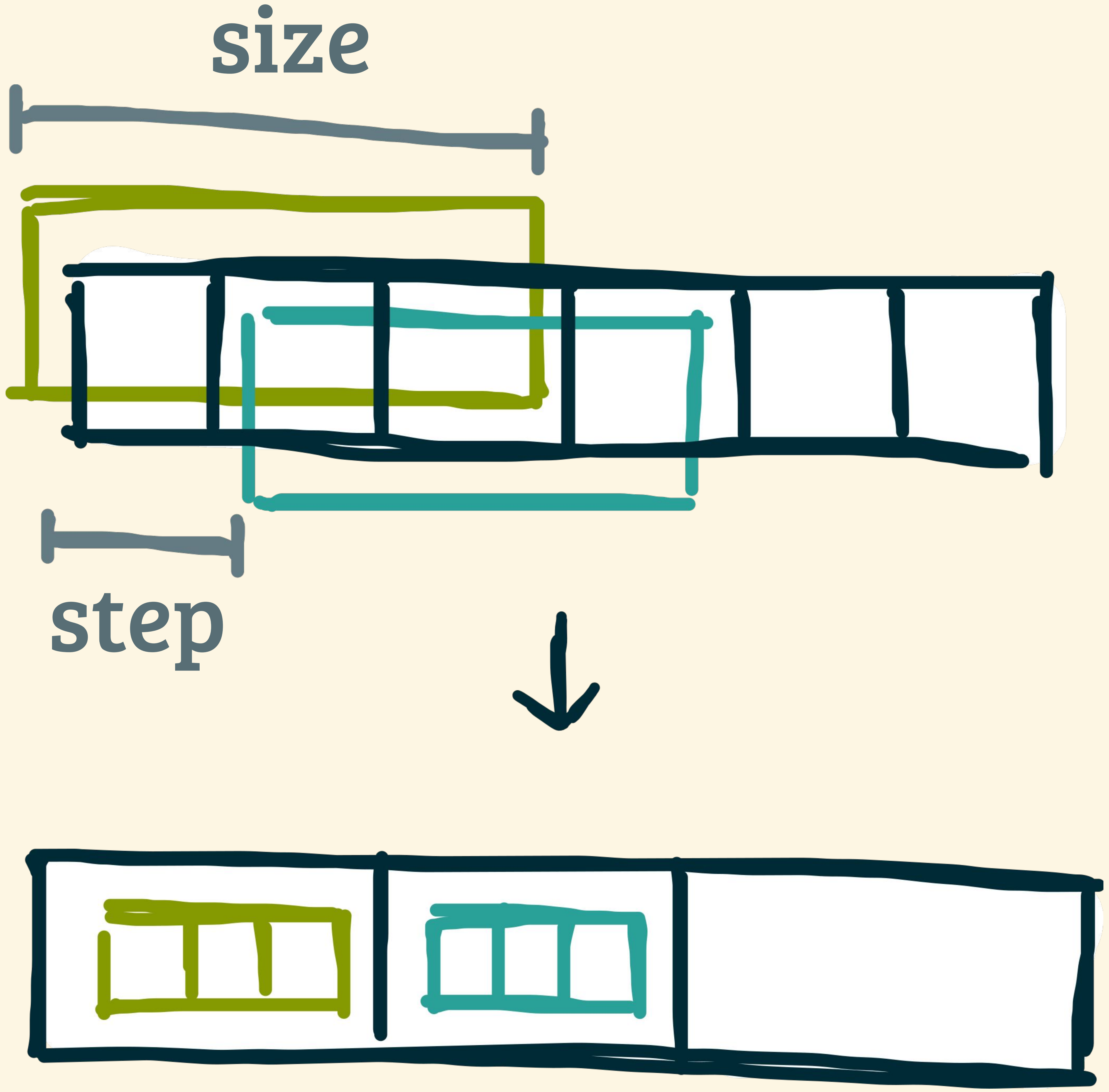
pad-reindexing.lift

```
clamp(i, n) = (i <  0) ? 0  :
              ((i >= n) ? n-1:i)

pad(1,1,clamp, [a,b,c,d]) =
    [a,a,b,c,d,d]
```

pad-constant.lift

```
constant(i, n) = C


pad(1,1,constant, [a,b,c,d]) =
    [C,a,b,c,d,C]
```

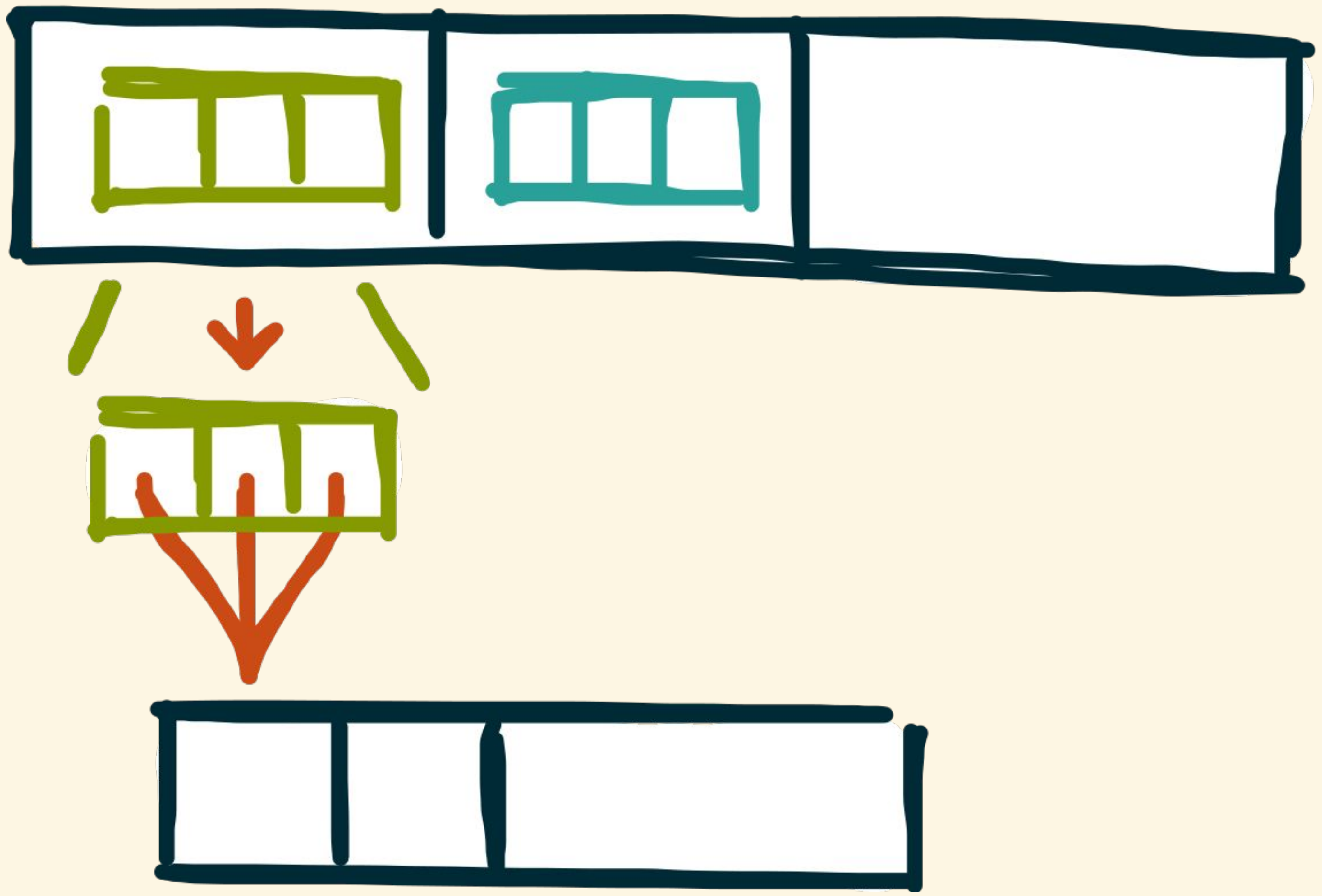
NEIGHBORHOOD CREATION USING SLIDE



slide-example.lift

```
slide(3,1,[a,b,c,d,e]) =
    [[a,b,c],[b,c,d],[c,d,e]]
```

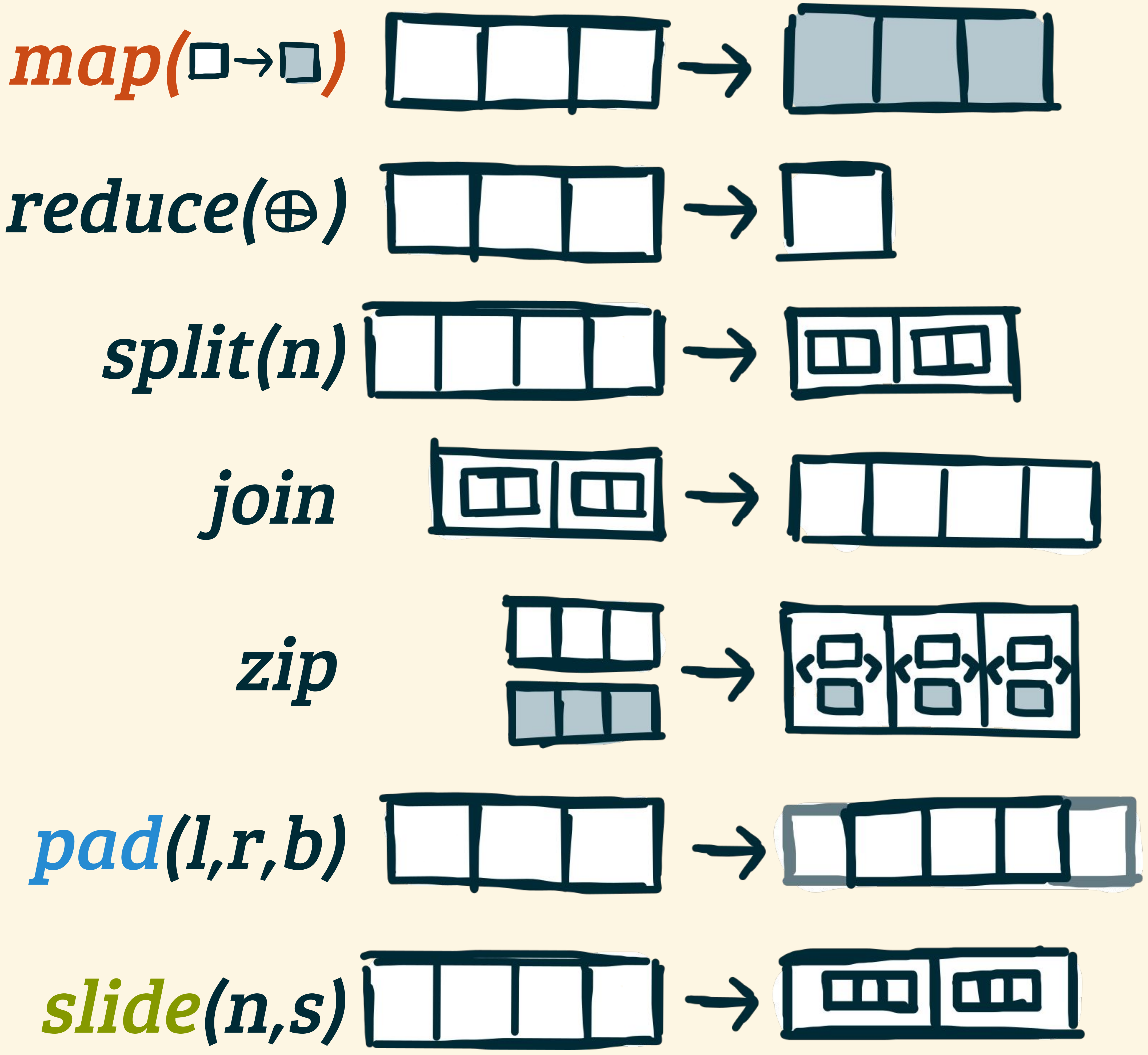
APPLYING STENCIL FUNCTION USING MAP



sum-neighborhoods.lift

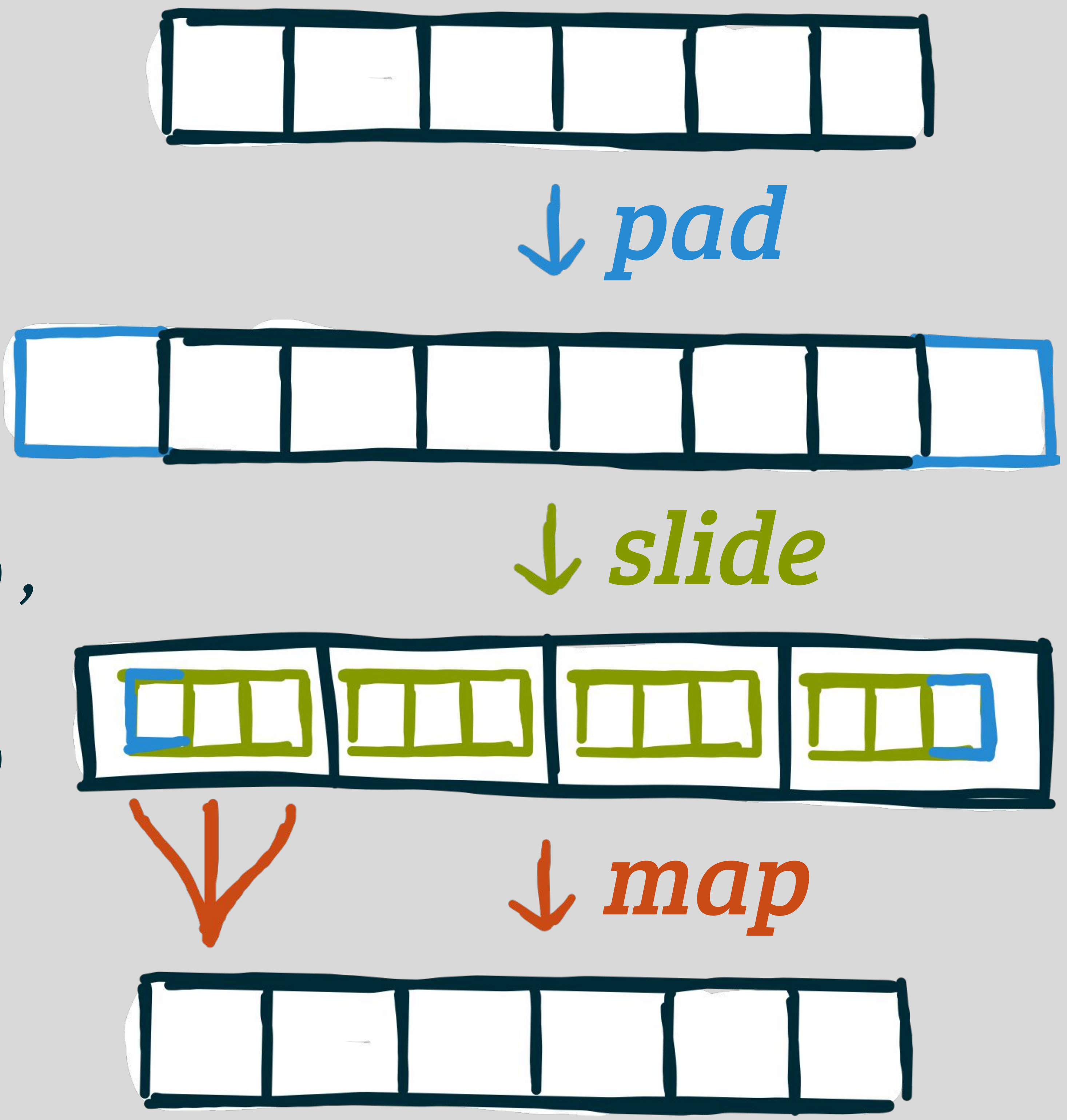
```
map(nbh =>
  reduce(add, 0.0f, nbh))
```

PUTTING IT TOGETHER



stencil1D.lift

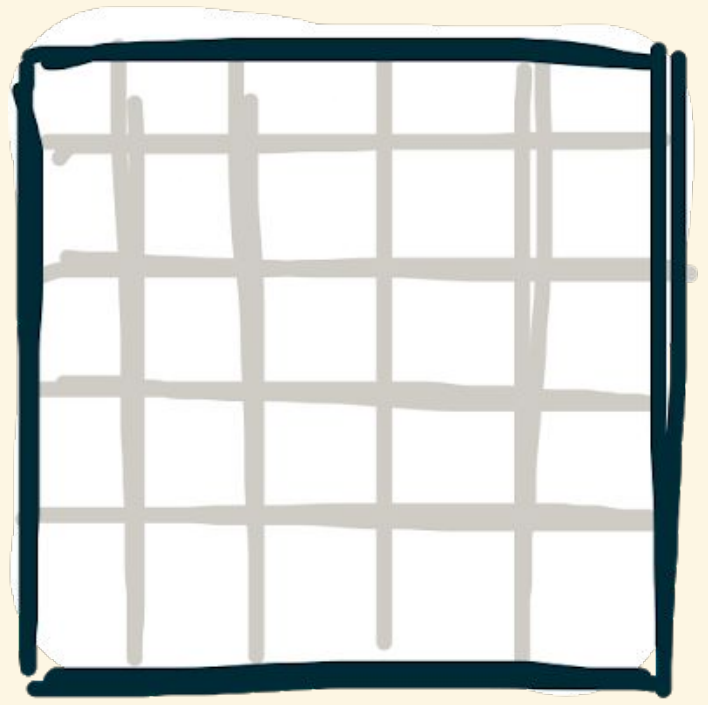
```
def stencil1D =
 fun(A =>
  map(reduce(add, 0.0f),
   slide(3,1,
    pad(1,1,clamp,A))))
```



Multidimensional Stencil Computations

are expressed as compositions of intuitive, generic 1D primitives

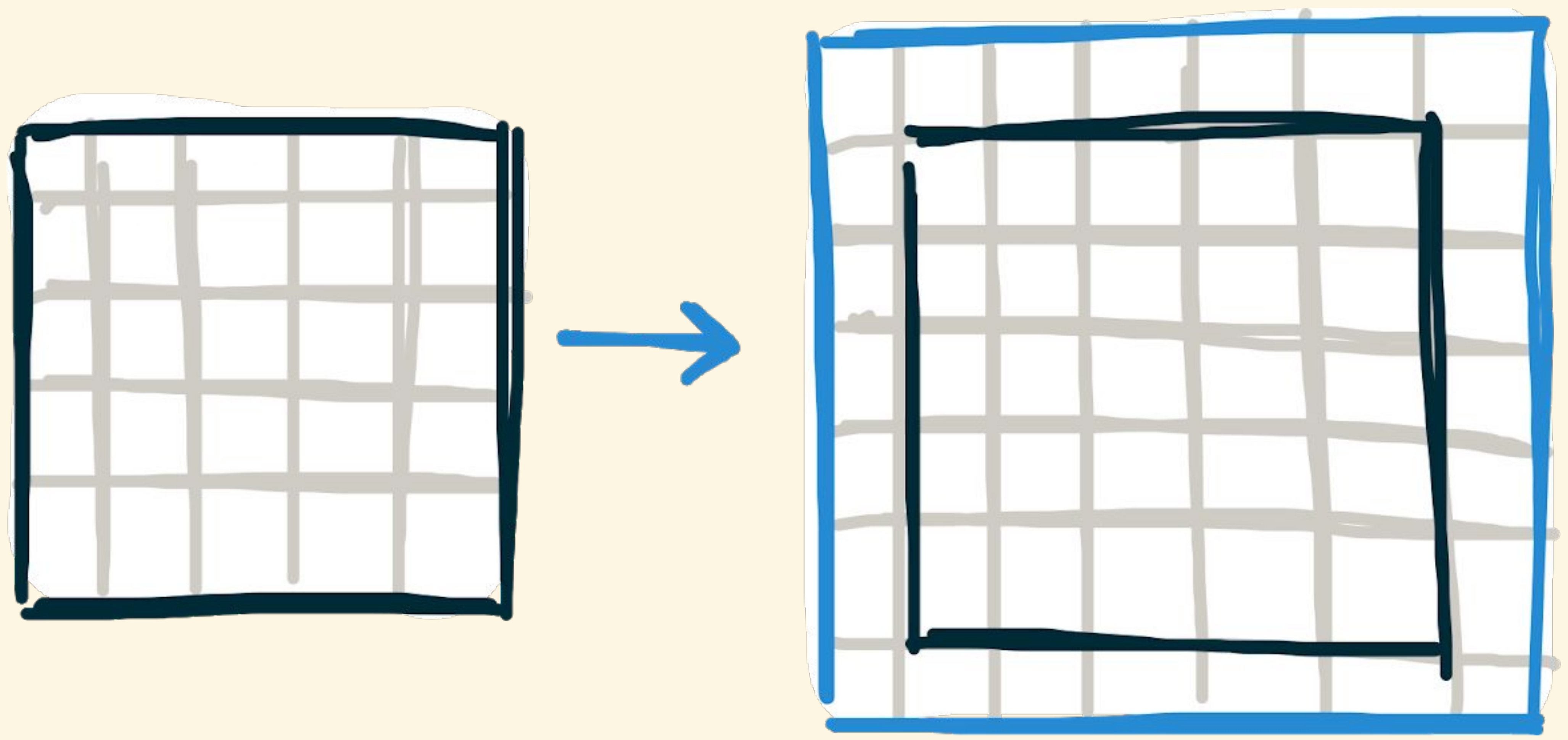
Decompose to Re-Compose



Multidimensional Stencil Computations

are expressed as compositions of intuitive, generic 1D primitives

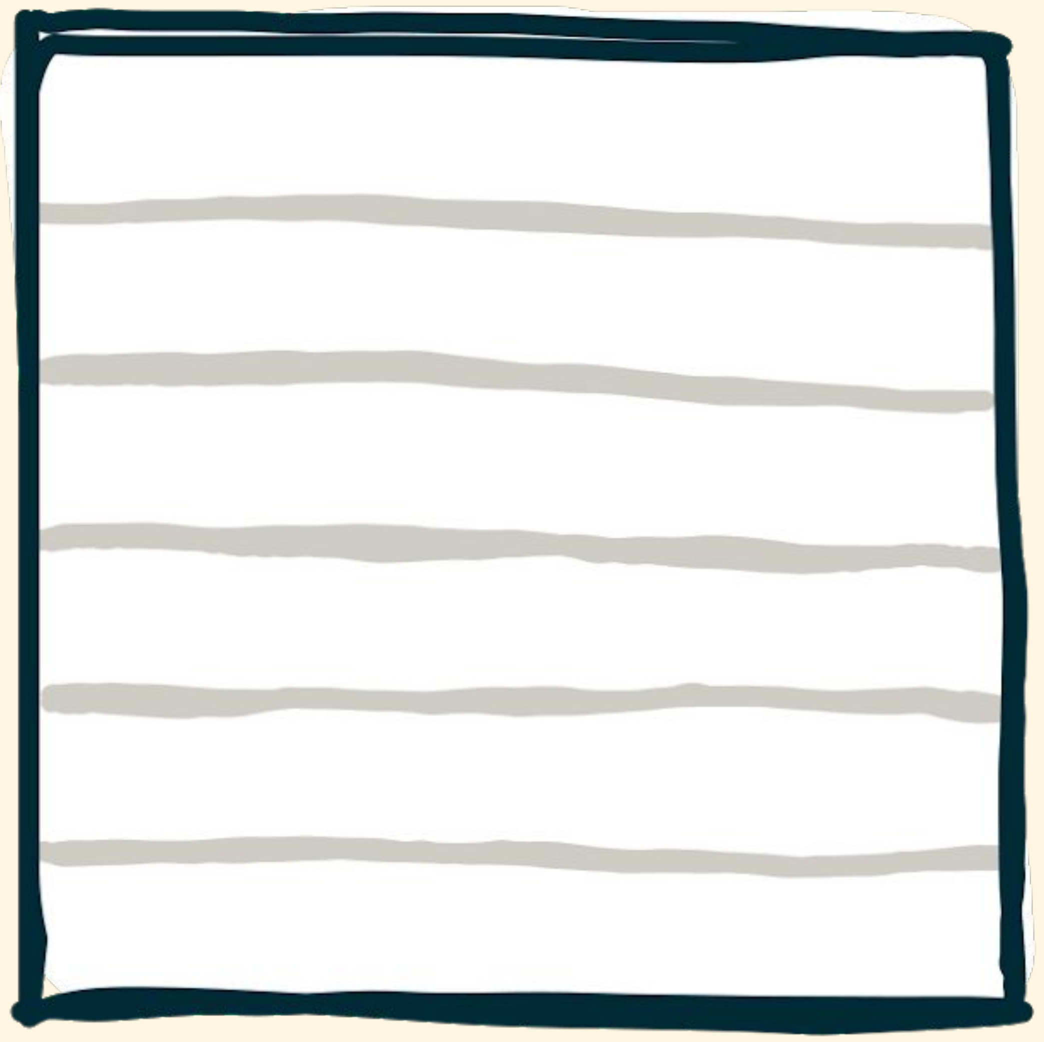
Decompose to Re-Compose



$\mathbf{pad}_2(1, 1, \mathit{clamp}, \mathtt{input})$

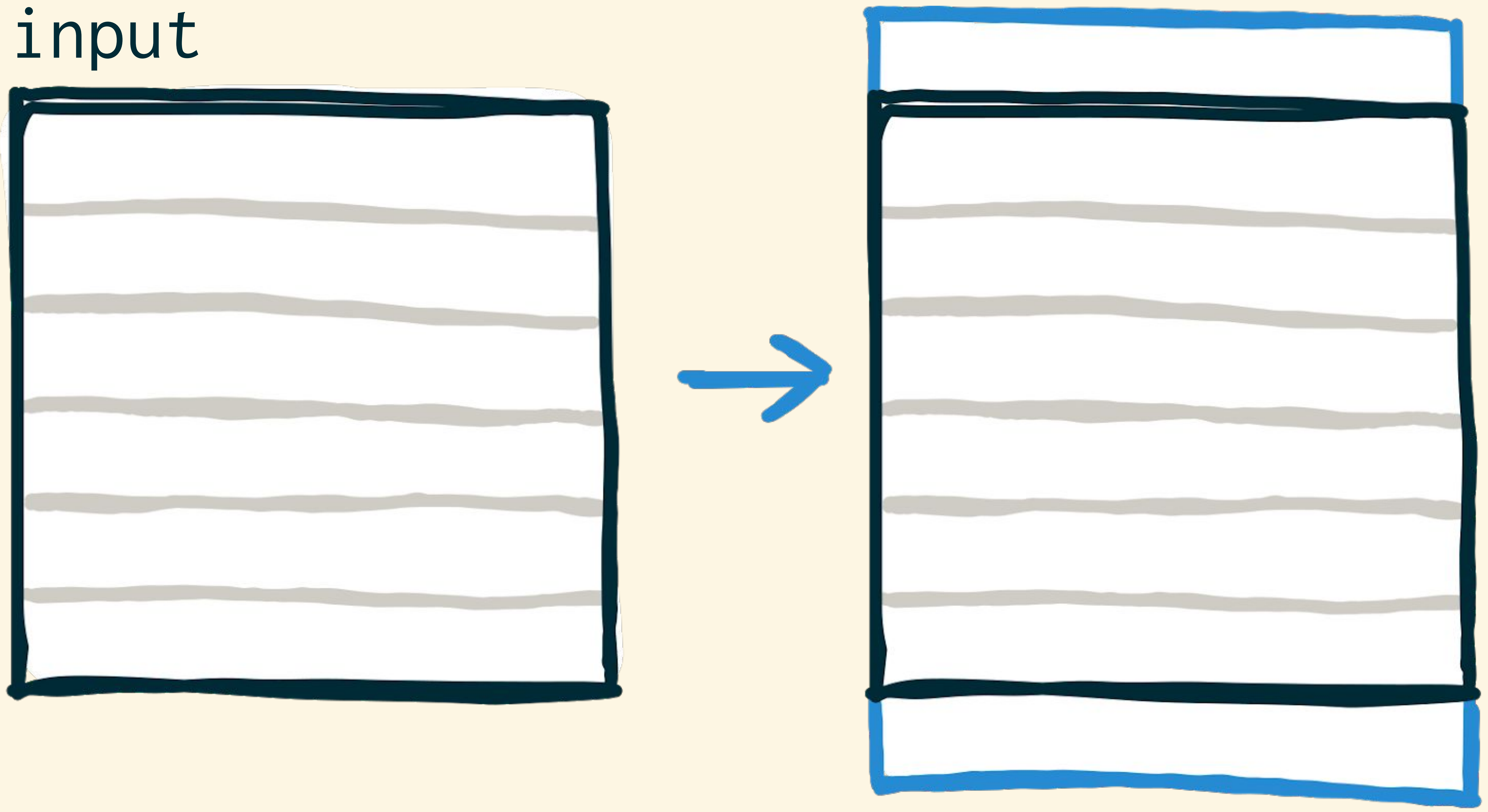
Multidimensional Boundary Handling using pad_2

input



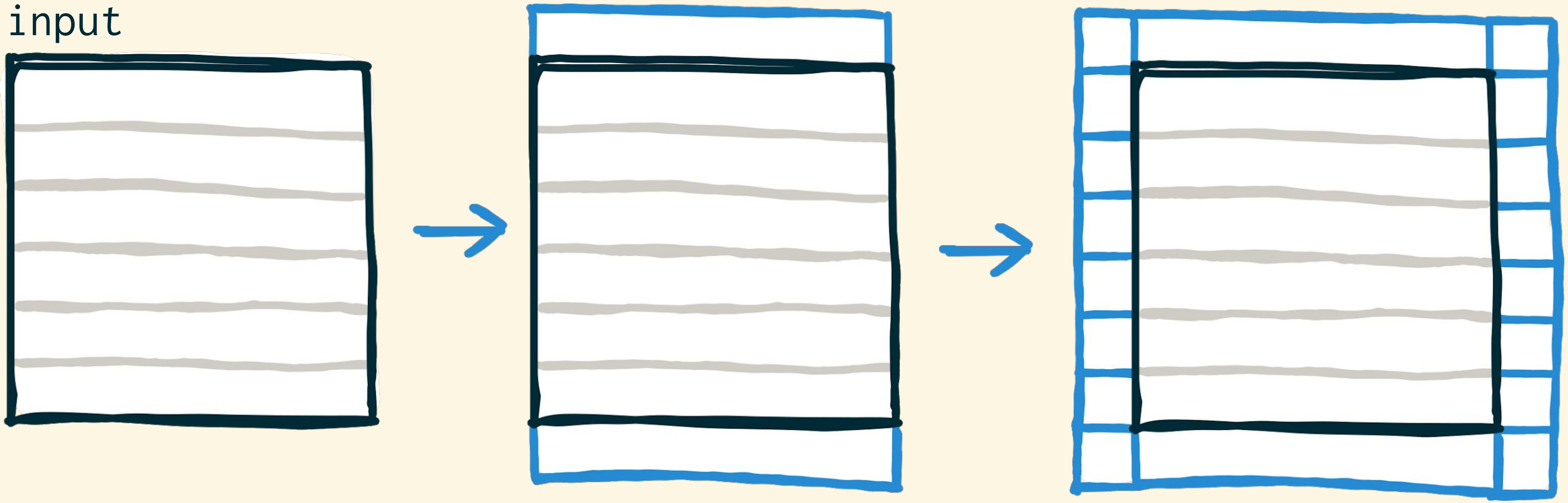
pad_2 =

MULTIDIMENSIONAL BOUNDARY HANDLING USING PAD_2



pad_2 = *pad*(l,r,b,input)

MULTIDIMENSIONAL BOUNDARY HANDLING USING PAD$_2$

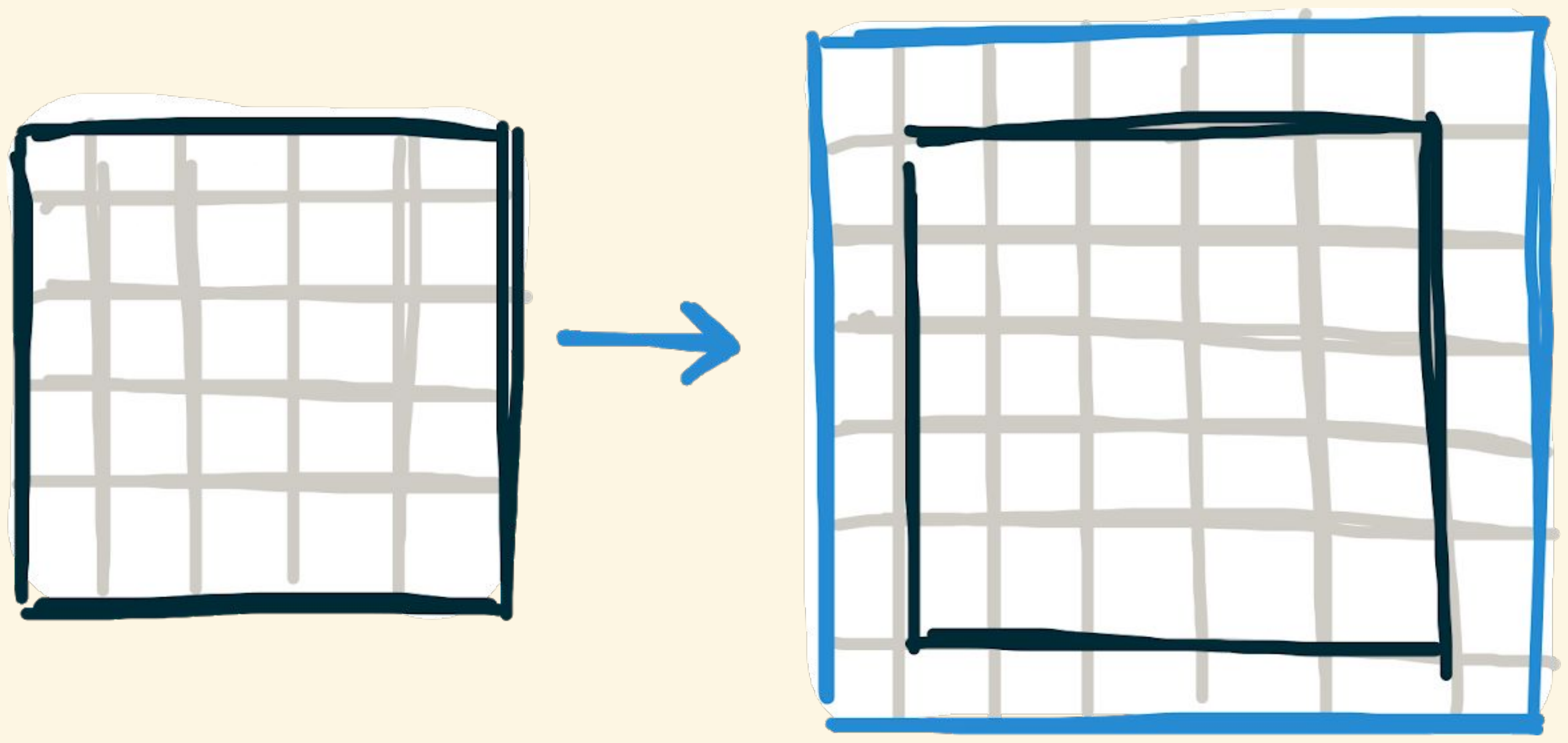


$$pad_2 = map(pad(l,r,b,pad(l,r,b,input)))$$

Multidimensional Stencil Computations

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

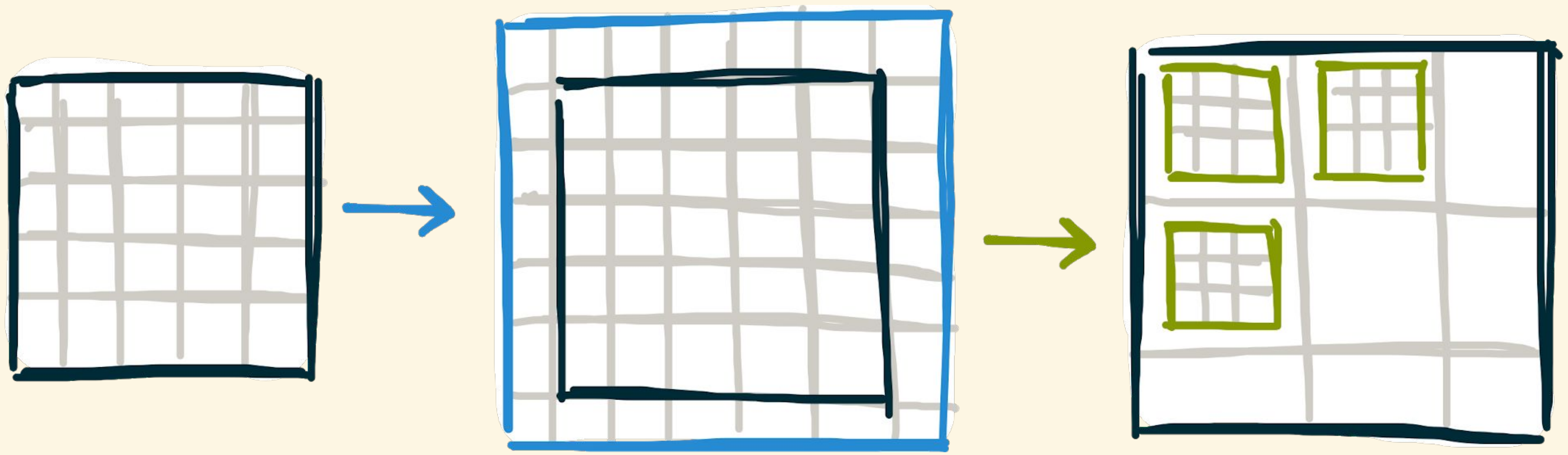


pad_2(1,1,*clamp*,input)

Multidimensional Stencil Computations

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

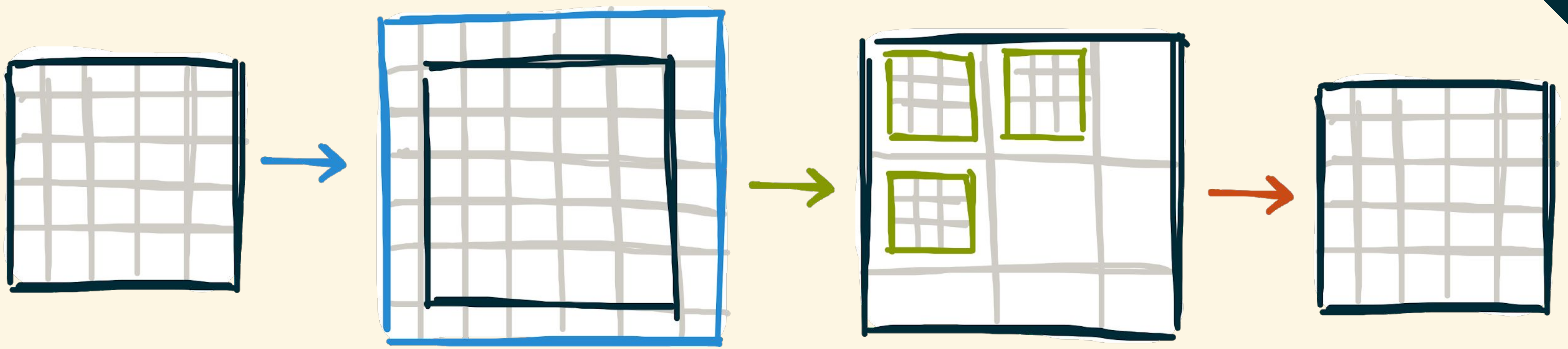


$slide_2$(3,1, pad_2(1,1,*clamp*,input))

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

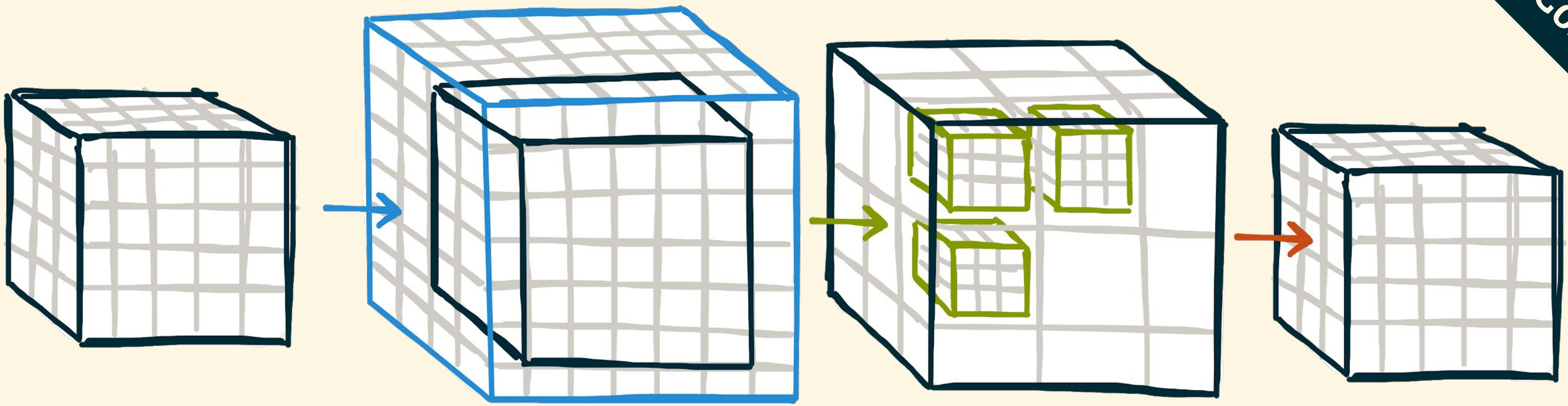


$$map_2(sum,\ slide_2(3,1,\ pad_2(1,1,clamp,input)))$$

Decompose to Re-Compose

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

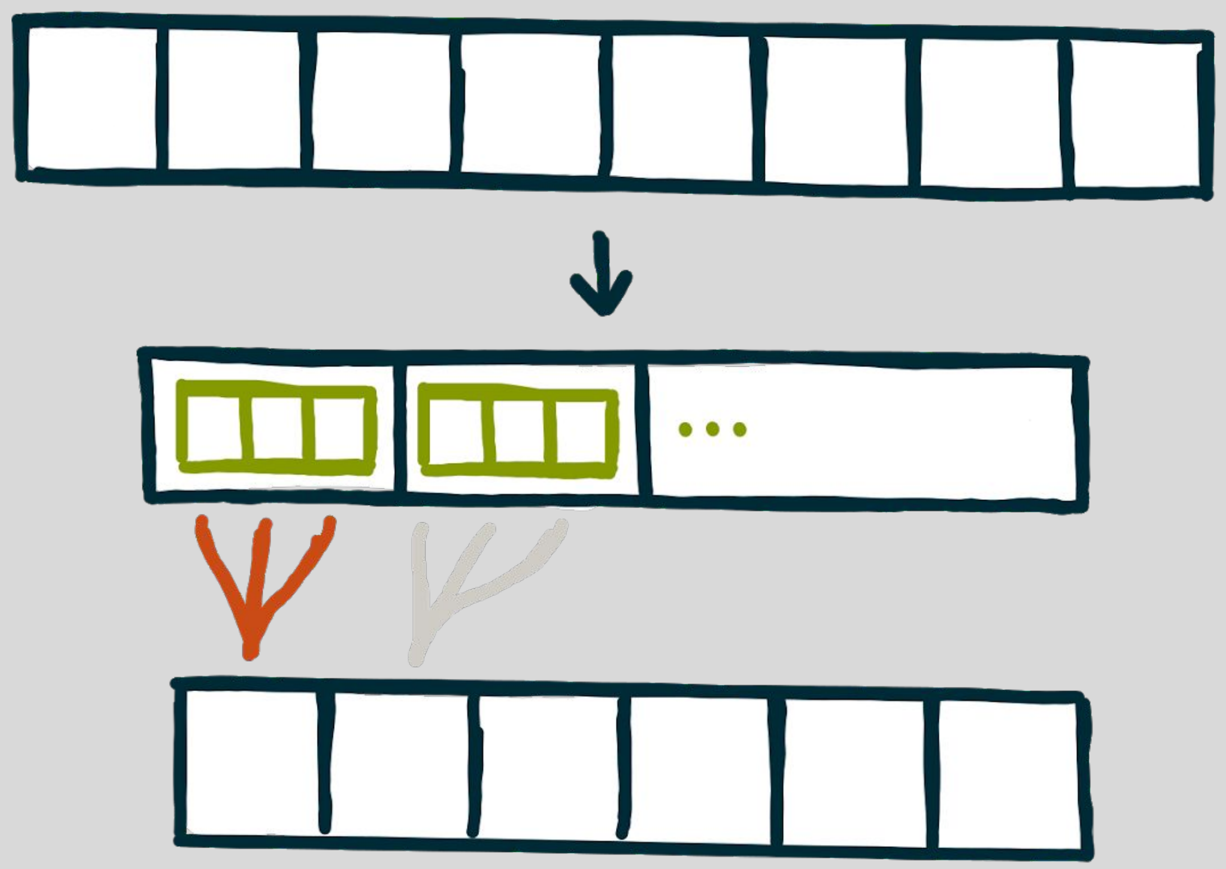


$map_3(sum,\ slide_3(3,1,\ pad_3(1,1,clamp,\mathrm{input})))$

Overlapped tiling as a rewrite rule

overlapped tiling rule

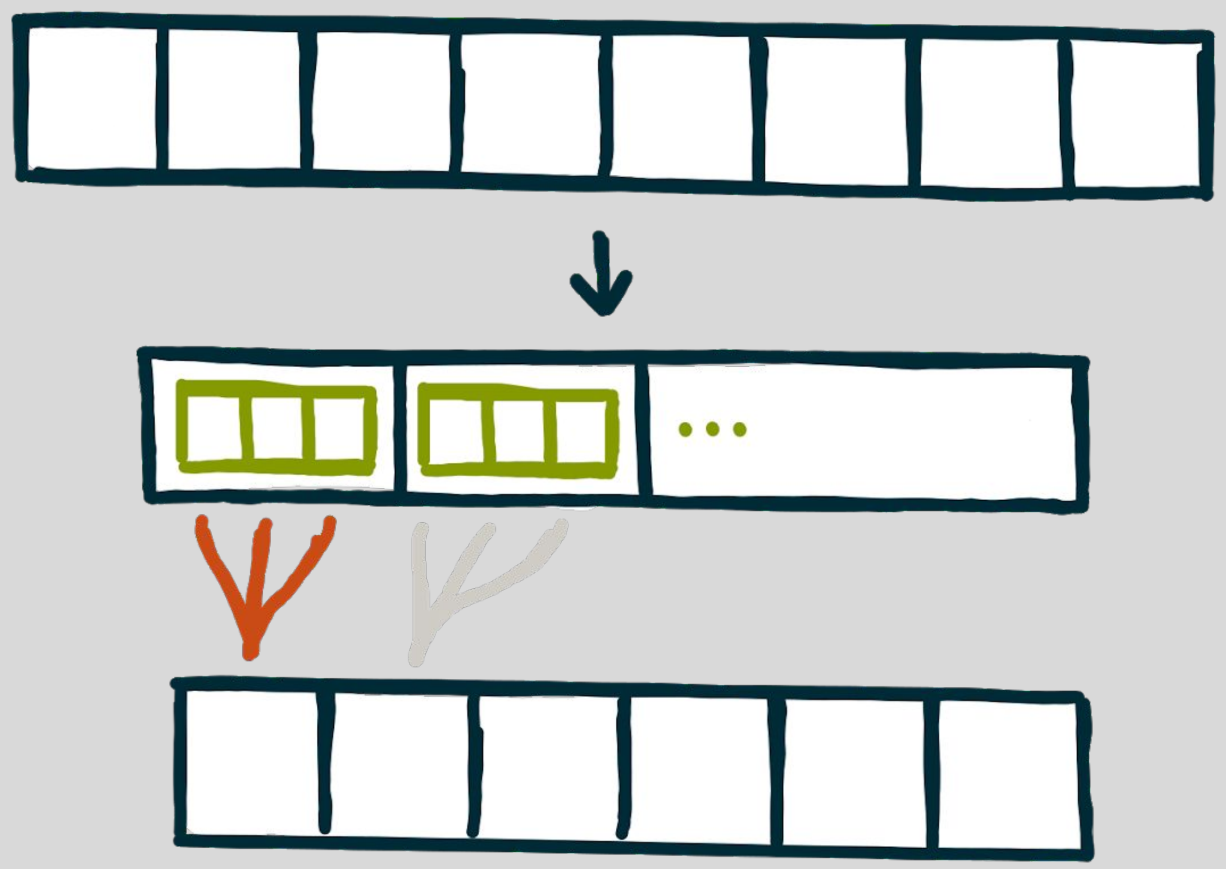
```
map(f, slide(3,1,input))
```

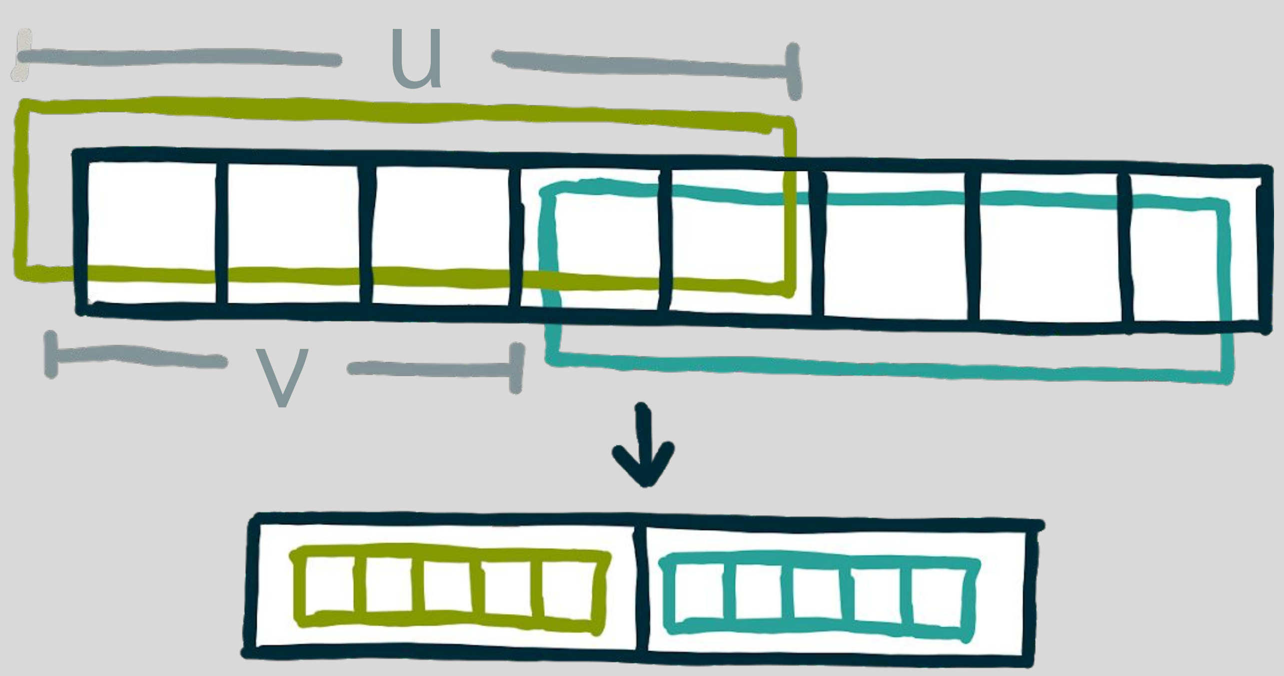


Overlapped tiling as a rewrite rule

overlapped tiling rule

map(f, *slide*(3,1,input)) ↦ *slide*(u,v,input)

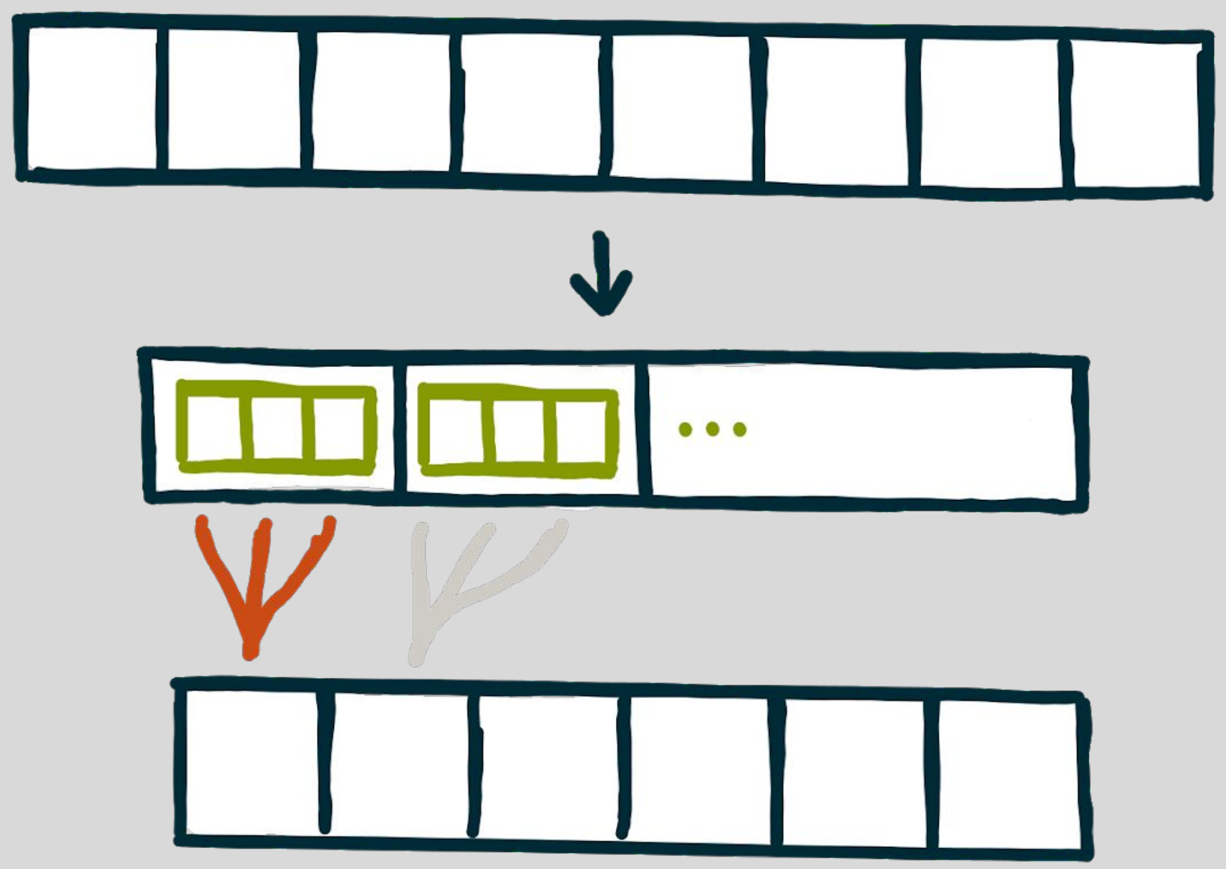


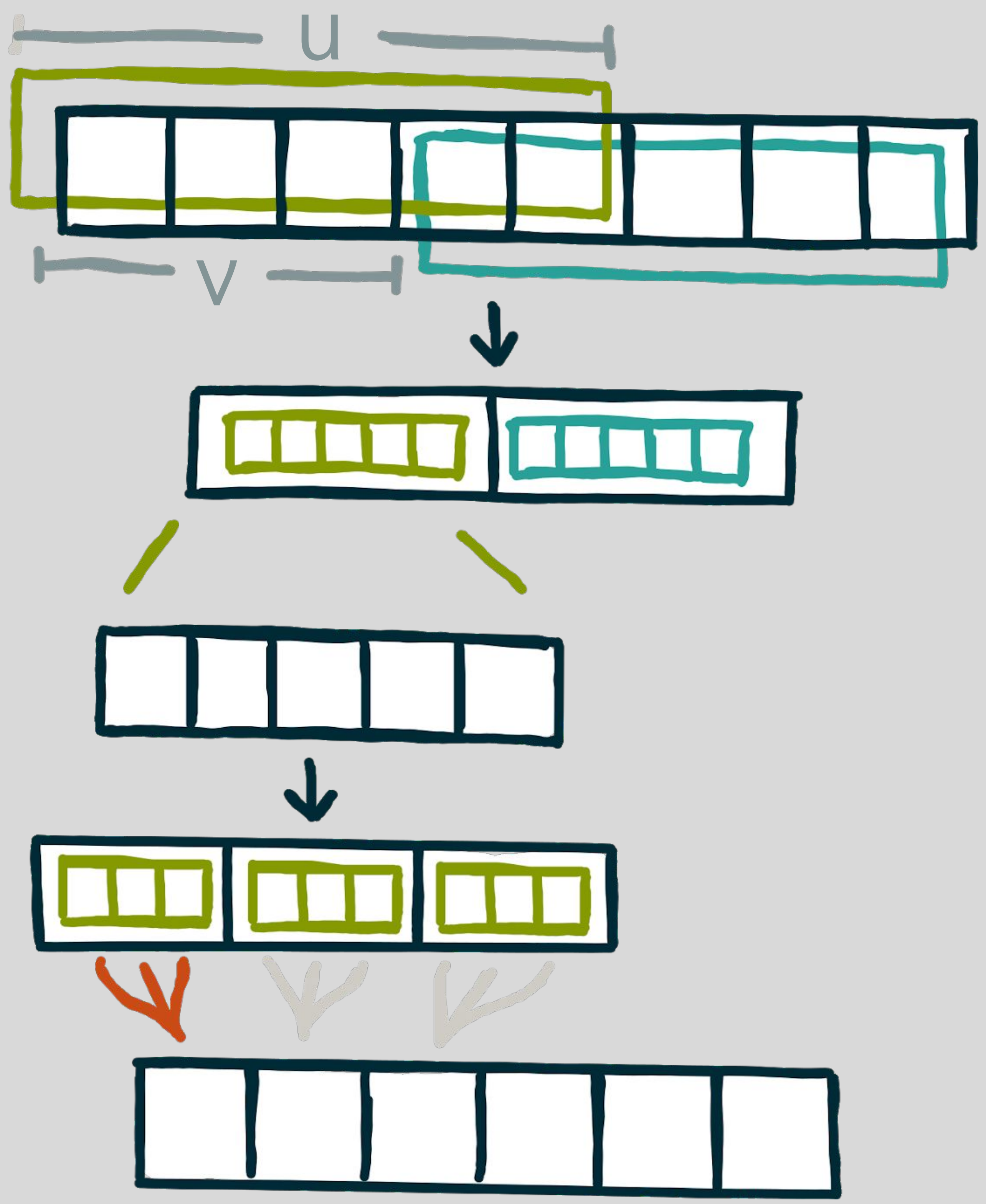


OVERLAPPED TILING AS A REWRITE RULE

overlapped tiling rule

```
map(f, slide(3,1,input))  ↦  join(map(tile ⇒
                                 map(f, slide(3,1,tile)),
                                   slide(u,v,input)))
```





EXPERIMENTAL EVALUATION

14 Benchmarks

6 hand-optimized
8 polyhedral compilation

< 3h Exploration

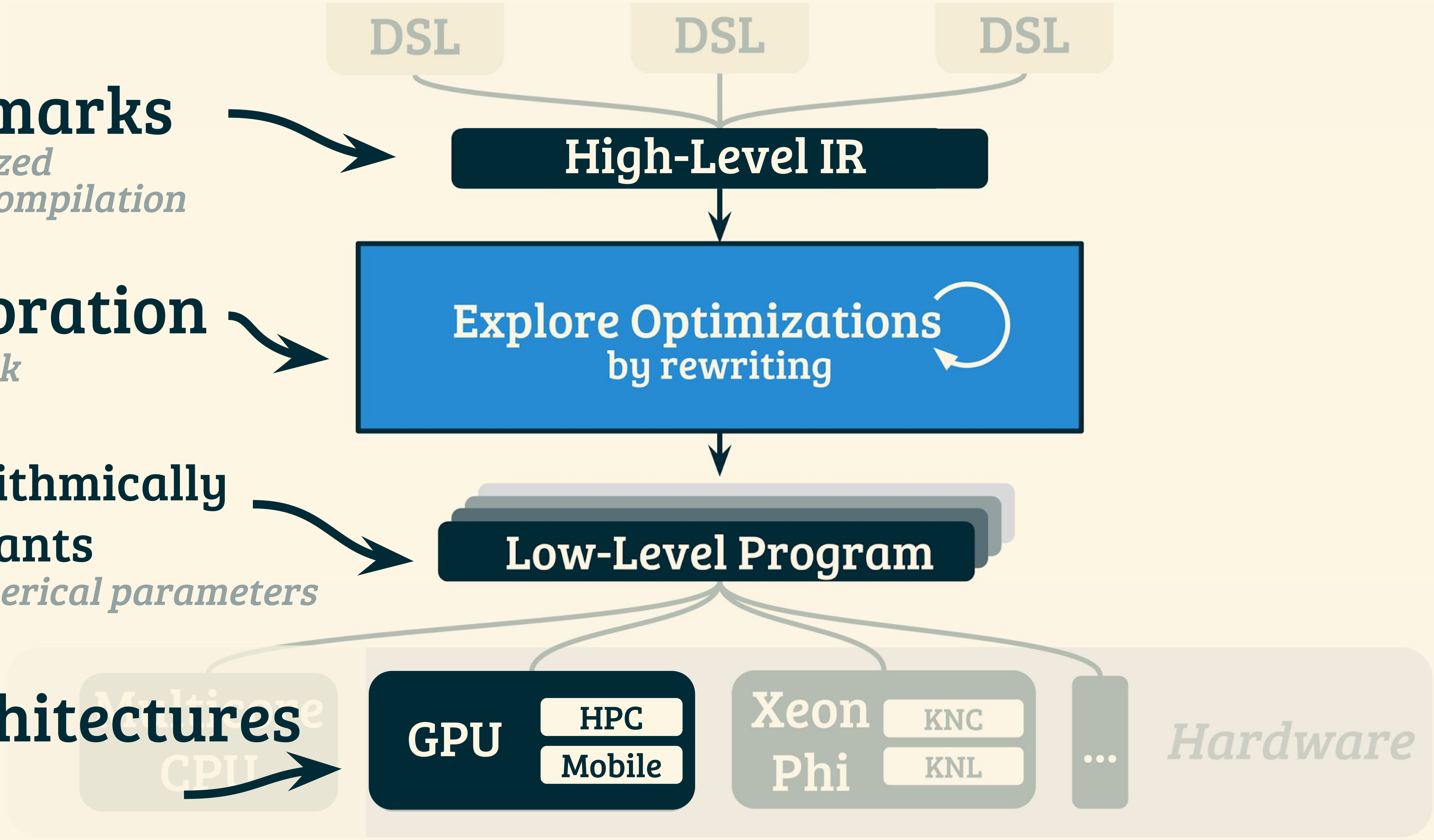
per benchmark

up to 20 algorithmically different variants

+ auto-tuning of numerical parameters

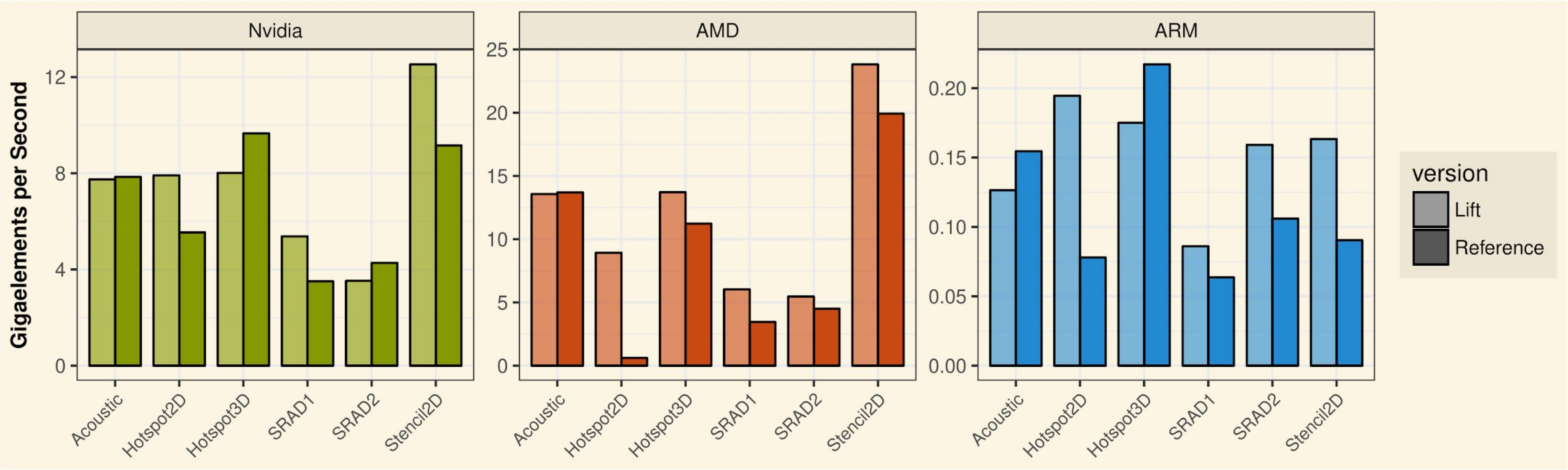
3 GPU Architectures

2 Desktop GPUs
1 Mobile GPU



COMPARISON WITH HAND-OPTIMIZED CODES

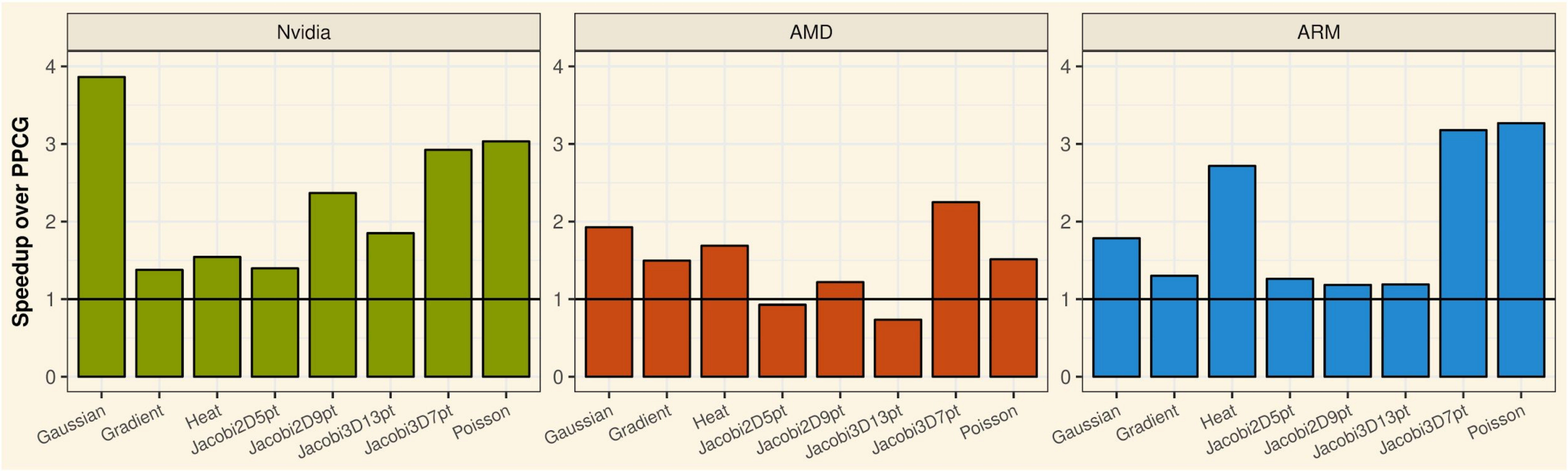
higher is better



Lift achieves the same performance
as hand optimized code

Comparison with Polyhedral Compilation

higher is better



Lift outperforms state-of-the-art optimizing compilers

LIFT IS OPEN SOURCE!

more info at:

lift-project.org

Paper Artifacts Source Code

Naums Mogers | Lu Li | Christophe Dubach | Bastian Hagedorn | Toomas Remmelg | Larisa Stoltzfus | Michel Steuwer | Federico Pizzuti | Adam Harries

ACCELERATING LEGACY CODE

[ASPLOS'18]

Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach

Philip Ginsbach
The University of Edinburgh
philip.ginsbach@ed.ac.uk

Toomas Remmelg
The University of Edinburgh
toomas.remmelg@ed.ac.uk

Michel Steuwer
University of Glasgow
michel.steuwer@glasgow.ac.uk

Bruno Bodin
The University of Edinburgh
bbodin@ed.ac.uk

Christophe Dubach
The University of Edinburgh
christophe.dubach@ed.ac.uk

Michael F. P. O'Boyle
The University of Edinburgh
mob@ed.ac.uk

Abstract

Heterogeneous accelerators often disappoint. They provide the prospect of great performance, but only deliver it when using vendor specific optimized libraries or domain specific languages. This requires considerable legacy code modifications, hindering the adoption of heterogeneous computing.

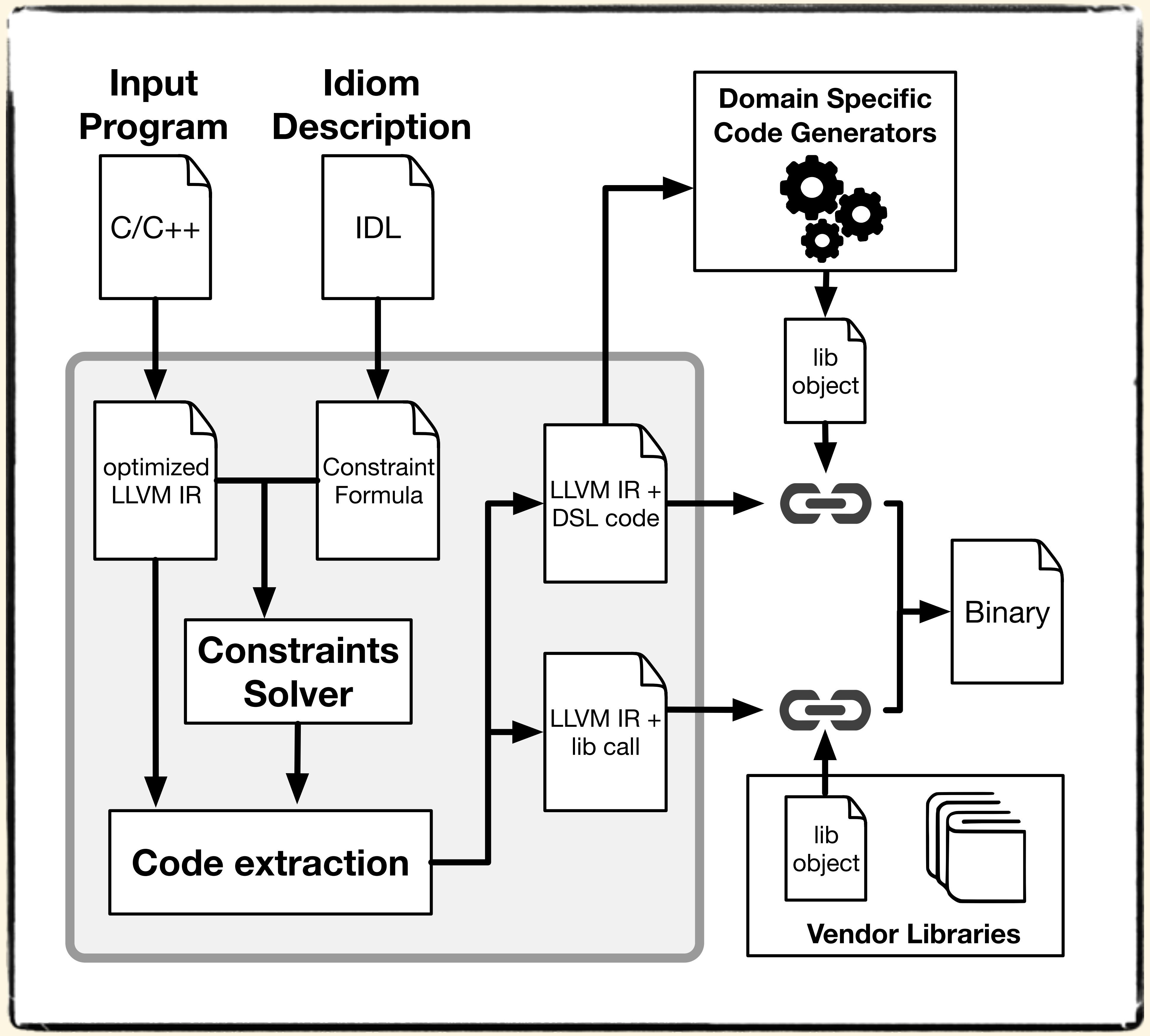
This paper develops a novel approach to automatically

1 Introduction

Heterogeneous accelerators provide the potential for great performance. However, achieving that potential is difficult. General purpose languages such as OpenCL [36] provide portability, but the achieved performance often disappoints [29]. This shortfall has led vendors to deliver specialized libraries to bridge the gap [2]. Alternatively, domain specific

IDIOM DETECTION VIA CONSTRAINT LANGUAGE

[ASPLOS'18]




```
...
```

```
Constraint SPMV
( inherits For and
```

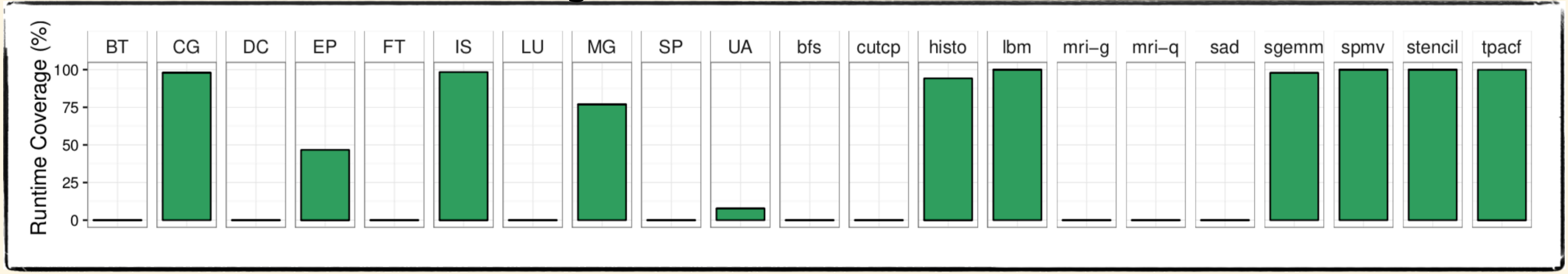
```
Constraint GEMM
( inherits ForNest(N=3) and
```

```
Constraint SESE
( {precursor} is branch instruction and
  {precursor} has control flow to {begin} and
  {end} is branch instruction and
  {end} has control flow to {successor} and
  {begin} control flow dominates {end} and
  {end} control flow post dominates {begin} and
  {precursor} strictly control flow dominates
      {begin} and
  {successor} strictly control flow post dominates
      {end} and
  all control flow from {begin} to {precursor}
      passes through {end} and
  all control flow from {successor} to {end}
      passes through {begin})
End
```

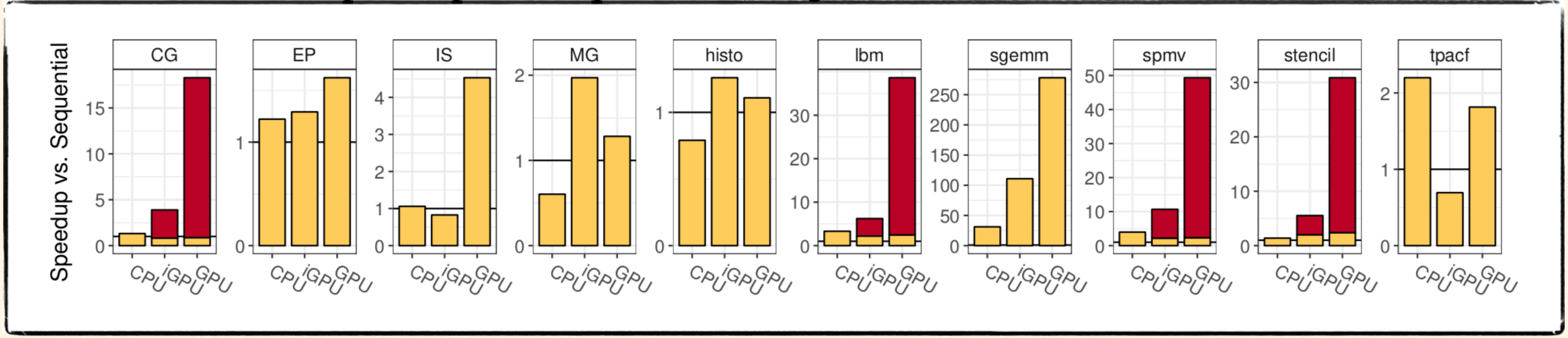
Performance Results

[ASPLOS'18]

Runtime Coverage of detected Idioms (NAS PB + Parboil)



Speedup vs. Sequential (using BLAS, Halide, Lift as backends)



University of Glasgow

The LIFT Project

Michel Steuwer — *michel.steuwer@glasgow.ac.uk*

www.lift-project.org @LIFTlang

INSPIRING PEOPLE

#UofGWorldChangers

@UofGlasgow