Generating Performance Portable Code with Lift

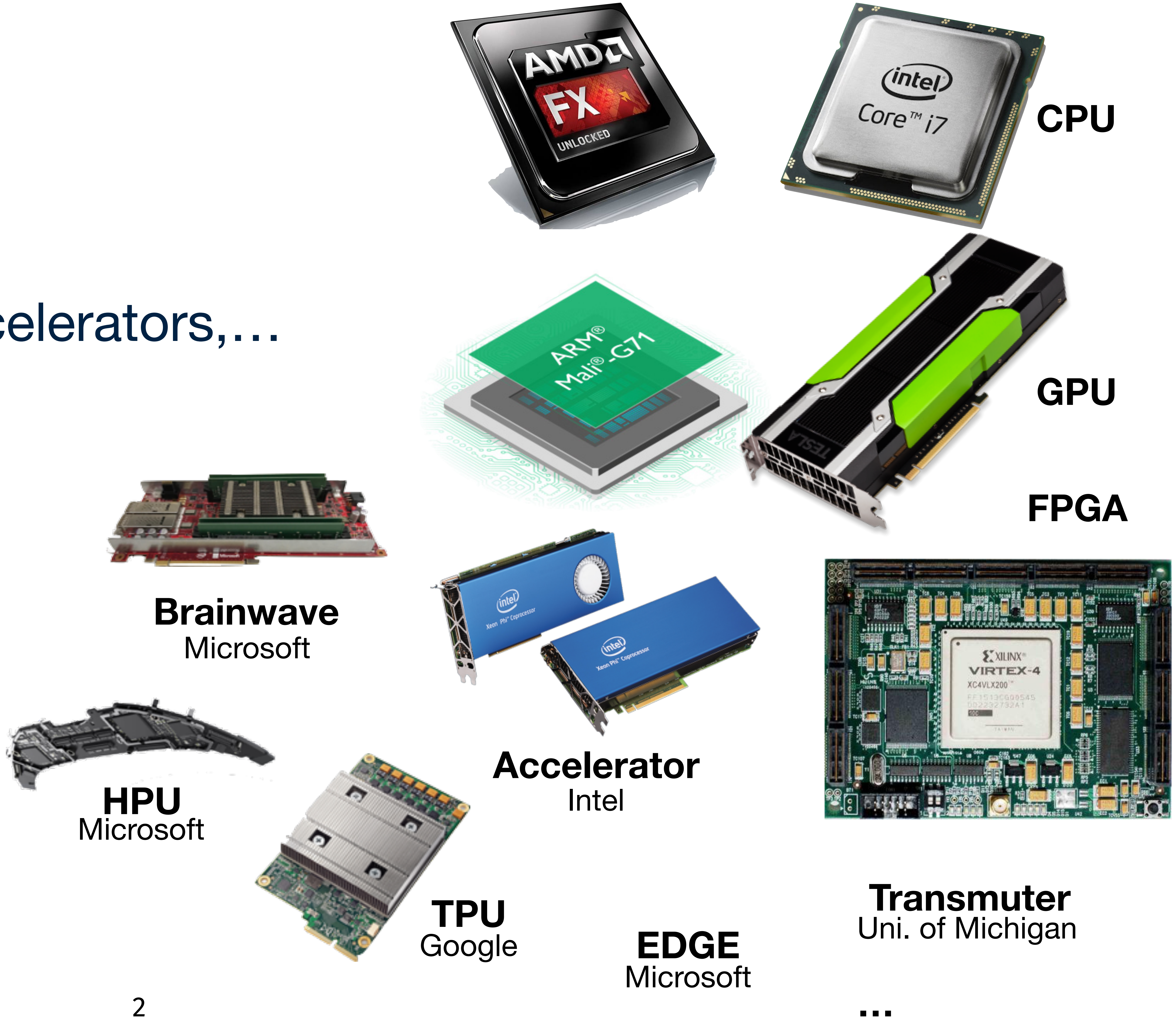
Christophe Dubach
Michel Steuwer
and the Lift team

Shonan Meeting 134
3rd September 2018

University of Glasgow
THE UNIVERSITY of EDINBURGH

Diversity is everywhere

- Parallel processors everywhere
- Many different types:
  CPUs, GPUs, FPGAs, special Accelerators,…
- Parallel programming is hard
- Optimising is even harder



Holy Grail: Performance Portability

- Some people think we already have this
  - e.g. OpenMP, OpenCL, OpenACC
  - It's a delusion! (or a question of definition)

- **Single-source** performance portability
  - Programs should be written once and for all
  - Exploit effectively current and future hardware
    - e.g. fast execution, low energy consumption

How to sum an array?

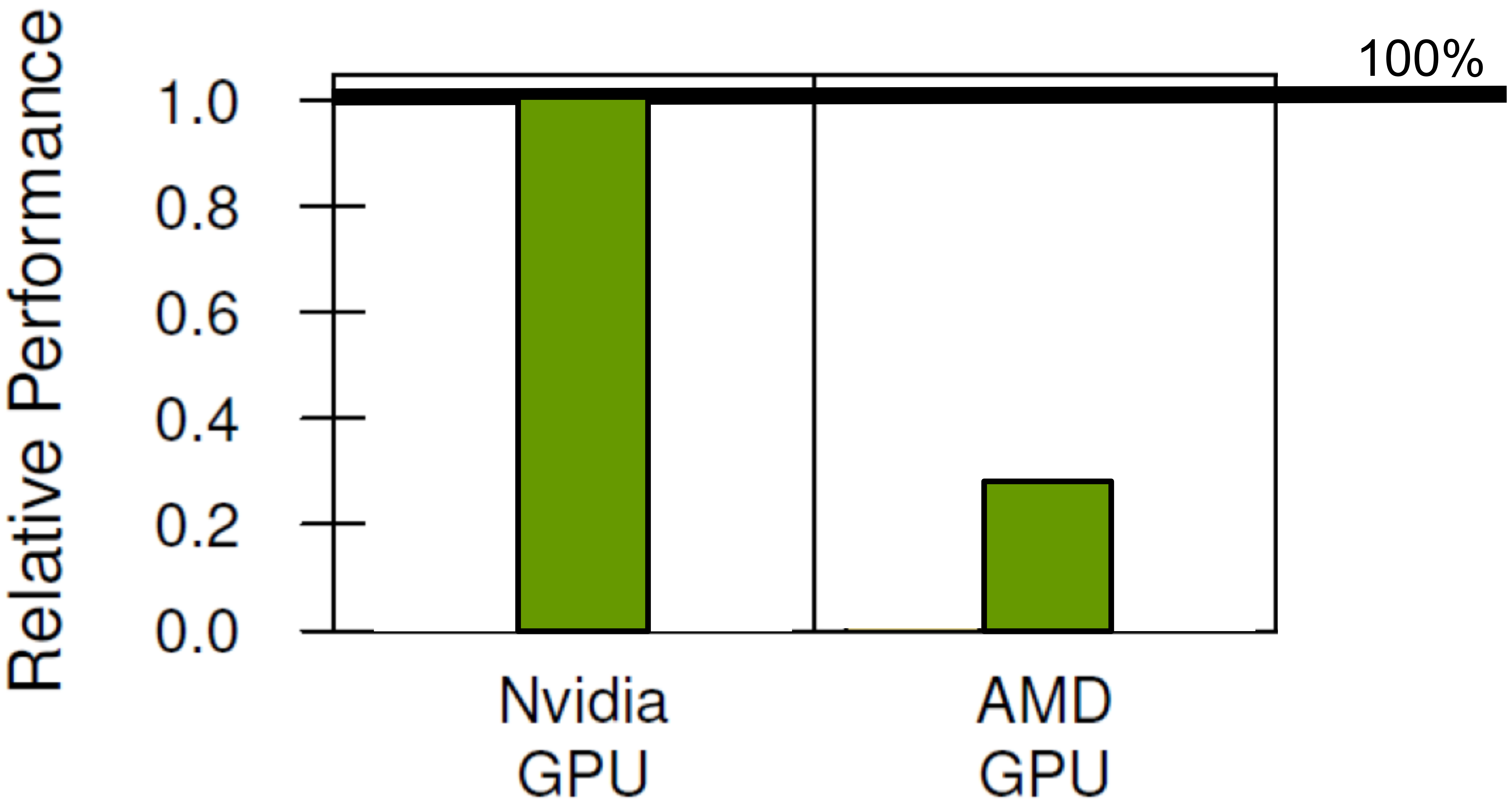
How to sum an array?

```
float acc = 0;
for (int i=0; i<N; i++)
  acc += input[i];
out[0] = acc;
```

How to really sum an array?

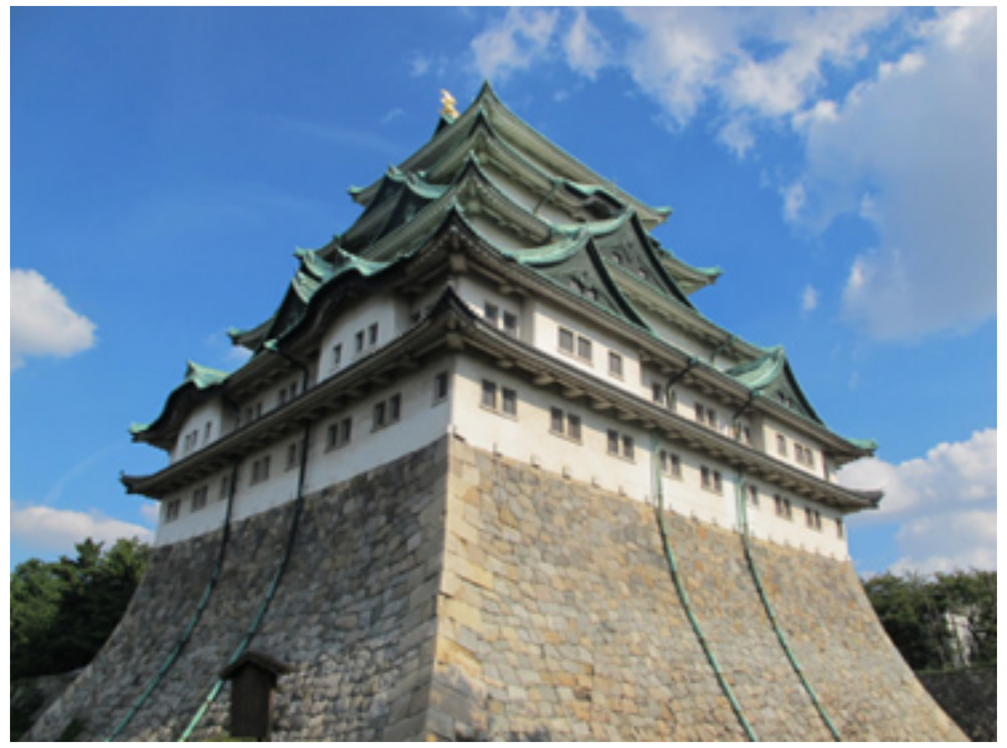
```
kernel void sum(global float* g_in, global float* g_out,
                unsigned int n, local volatile float* l_data) {

  unsigned int tid = get_local_id(0);
  unsigned int i   = get_group_id(0) * 256 + get_local_id(0);

  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_in[i];
    i += 256 * get_num_groups(0);
  }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (tid < 128)
    l_data[tid] += l_data[tid+128];
  barrier(CLK_LOCAL_MEM_FENCE);
  if (tid <  64)
    l_data[tid] += l_data[tid+ 64];
  barrier(CLK_LOCAL_MEM_FENCE)

  if (tid < 32) {
    l_data[tid] += l_data[tid+32]; l_data[tid] += l_data[tid+16];
    l_data[tid] += l_data[tid+ 8]; l_data[tid] += l_data[tid+ 4];
    l_data[tid] += l_data[tid+ 2]; l_data[tid] += l_data[tid+ 1];
  }
  if (tid == 0)
    g_out[get_group_id(0)] = l_data[0];
}
```

Performance is clearly not portable



Performance Portability needs two ingredients

- **Hardware agnostic high-level language**
  - Shield the programmer from any hardware-specific details



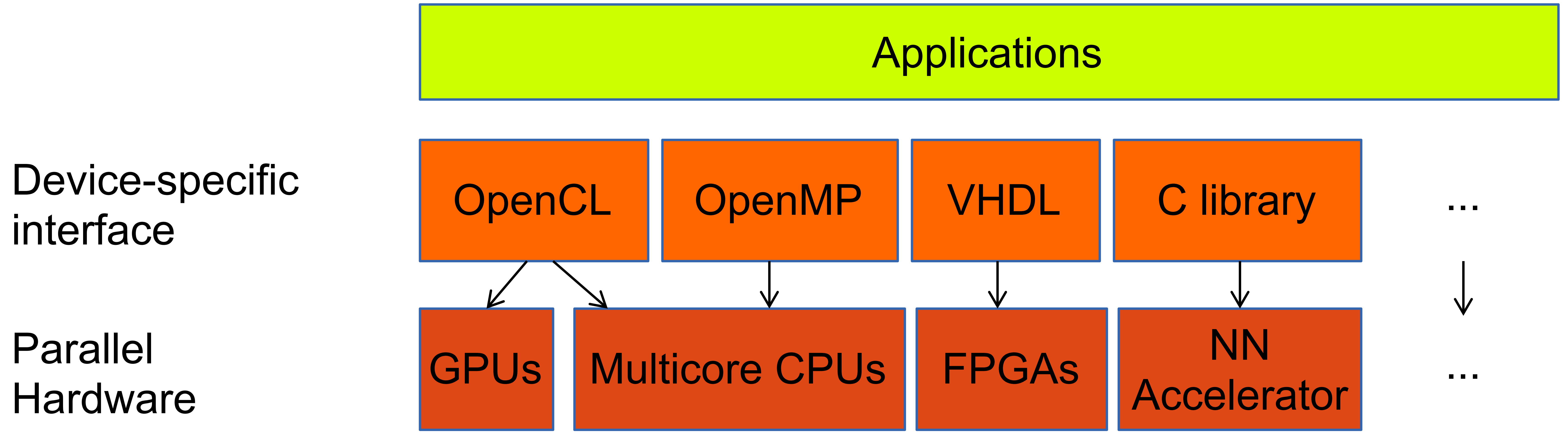
- **Generic & reusable compilation and optimisation process**
  - Express hardware-paradigms
  - Extensible mechanism
  - Express optimisations and automatic exploration

We already have performance portability for sequential machines

- **Hardware agnostic high-level language**
  - e.g. C
  - control flow, functions, data structures

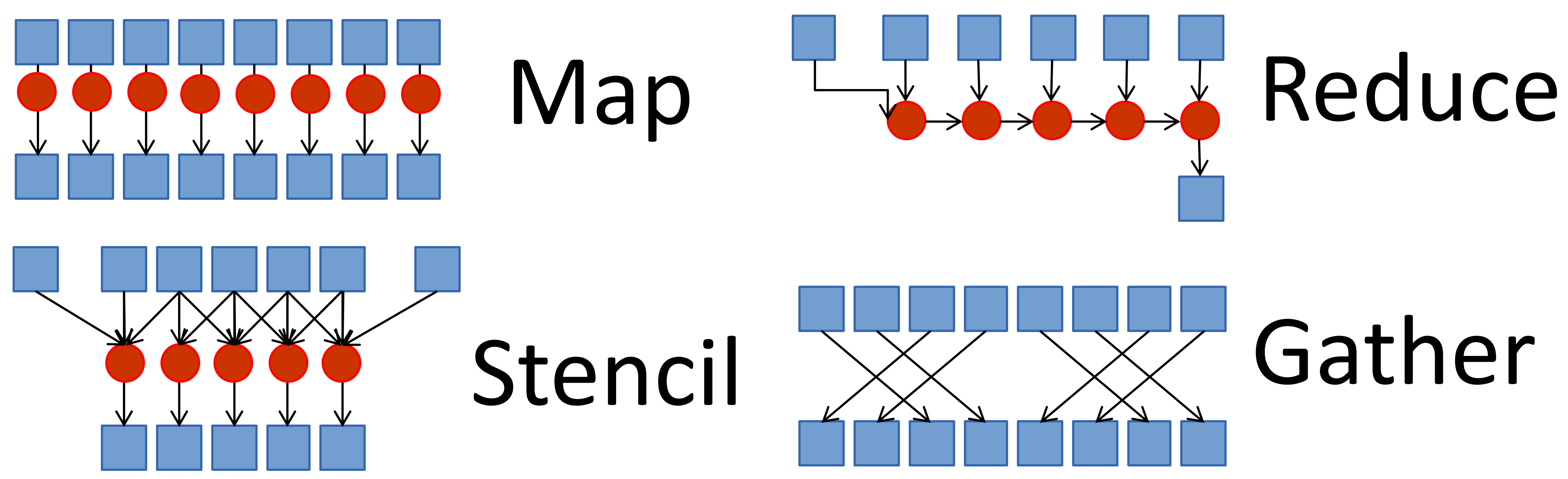
- **Generic & reusable compilation and optimisation process**
  - e.g. LLVM
  - TableGen for writing backends
  - loop optimisations

Current landscape for parallel / heterogenous machines



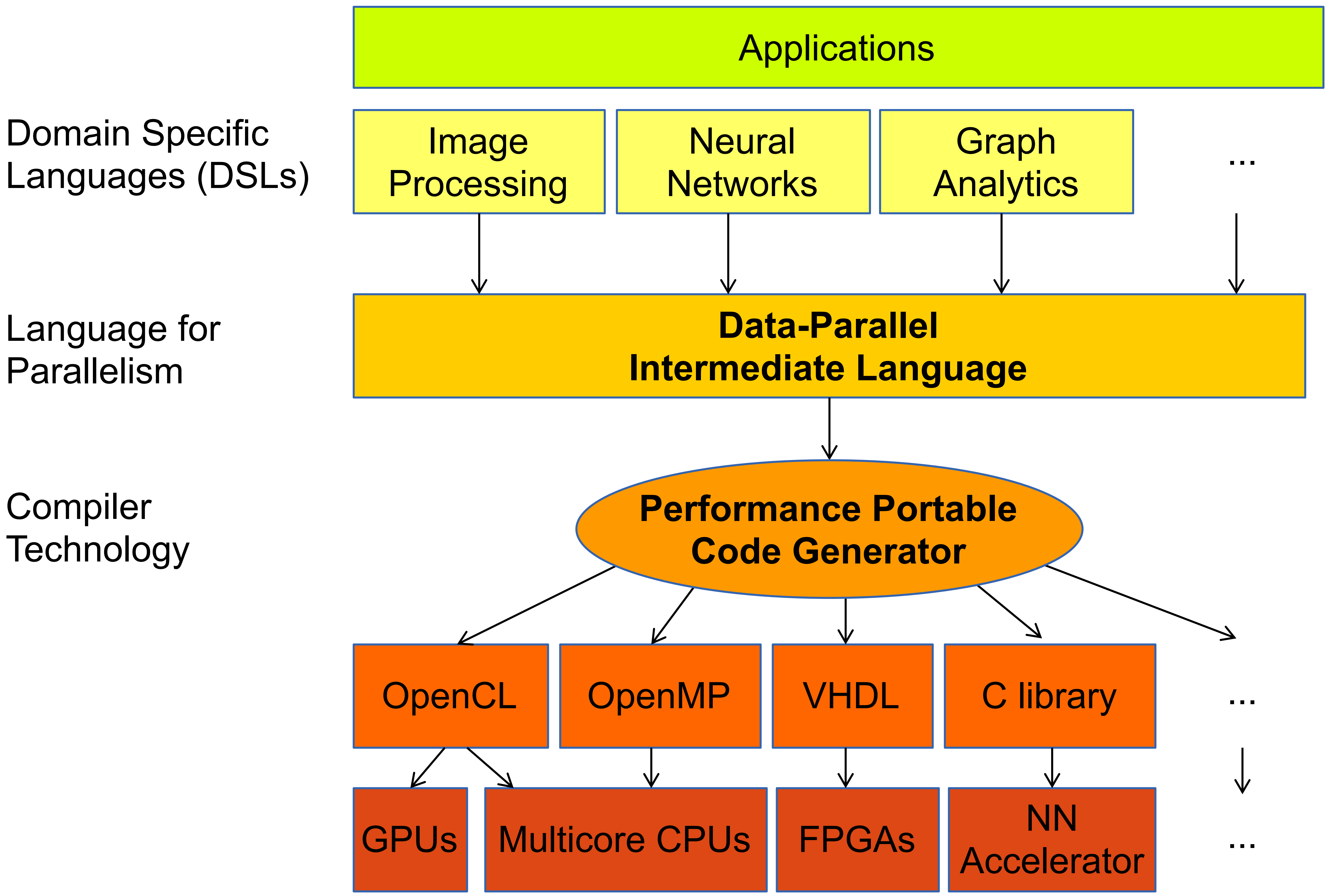
Not solved yet for parallel / heterogenous machines

- **High-level programming abstractions are here**
  - Accelerate, Futhark, LiquidMetal, Delite, Halide
  - All **functional** in nature
    - Hides the hardware!
    - High-level information available to the compiler

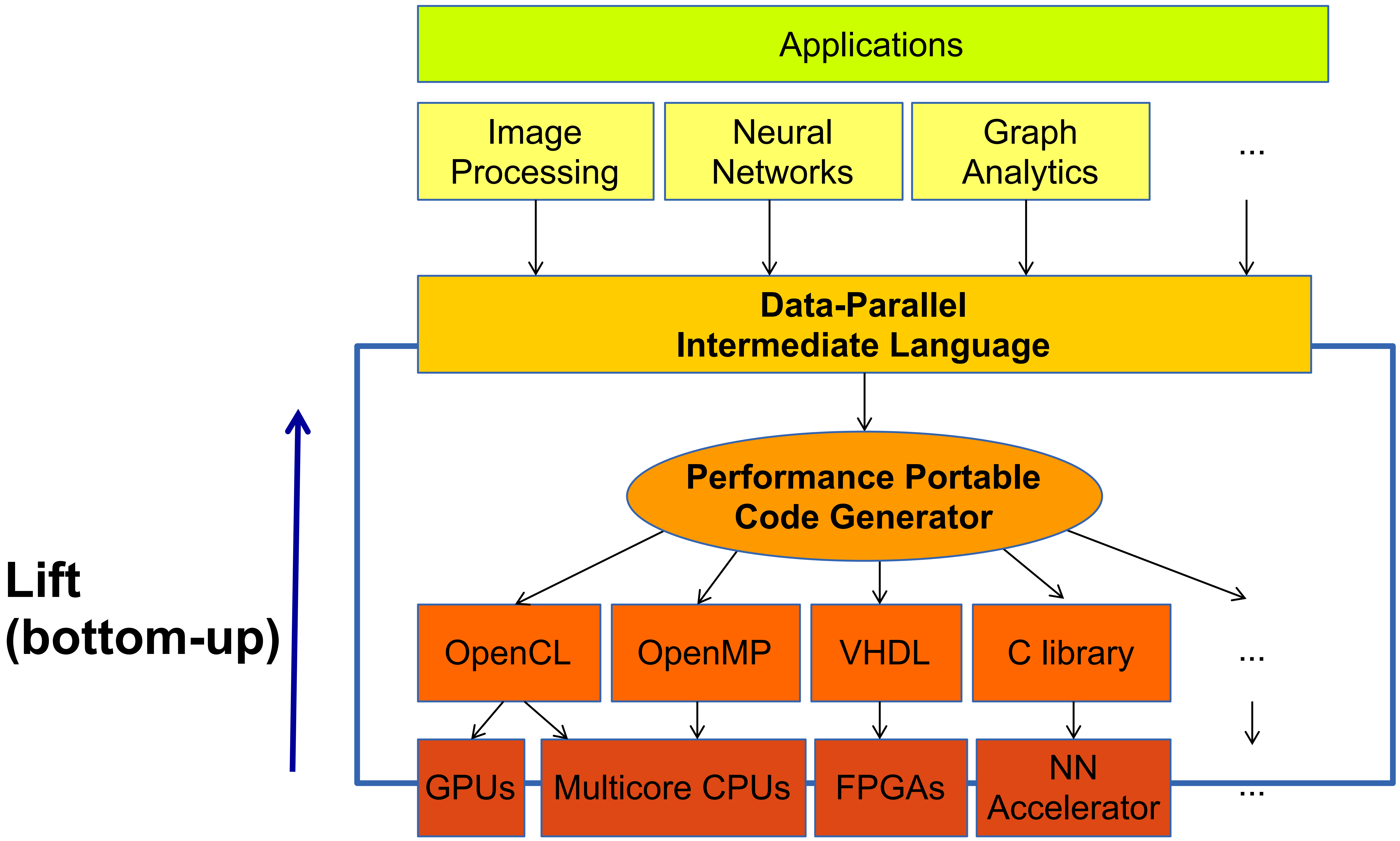


- **However, reusable and portable compilation/optimisation is lacking behind**
  - Currently, specialised backend and optimiser for each hardware target

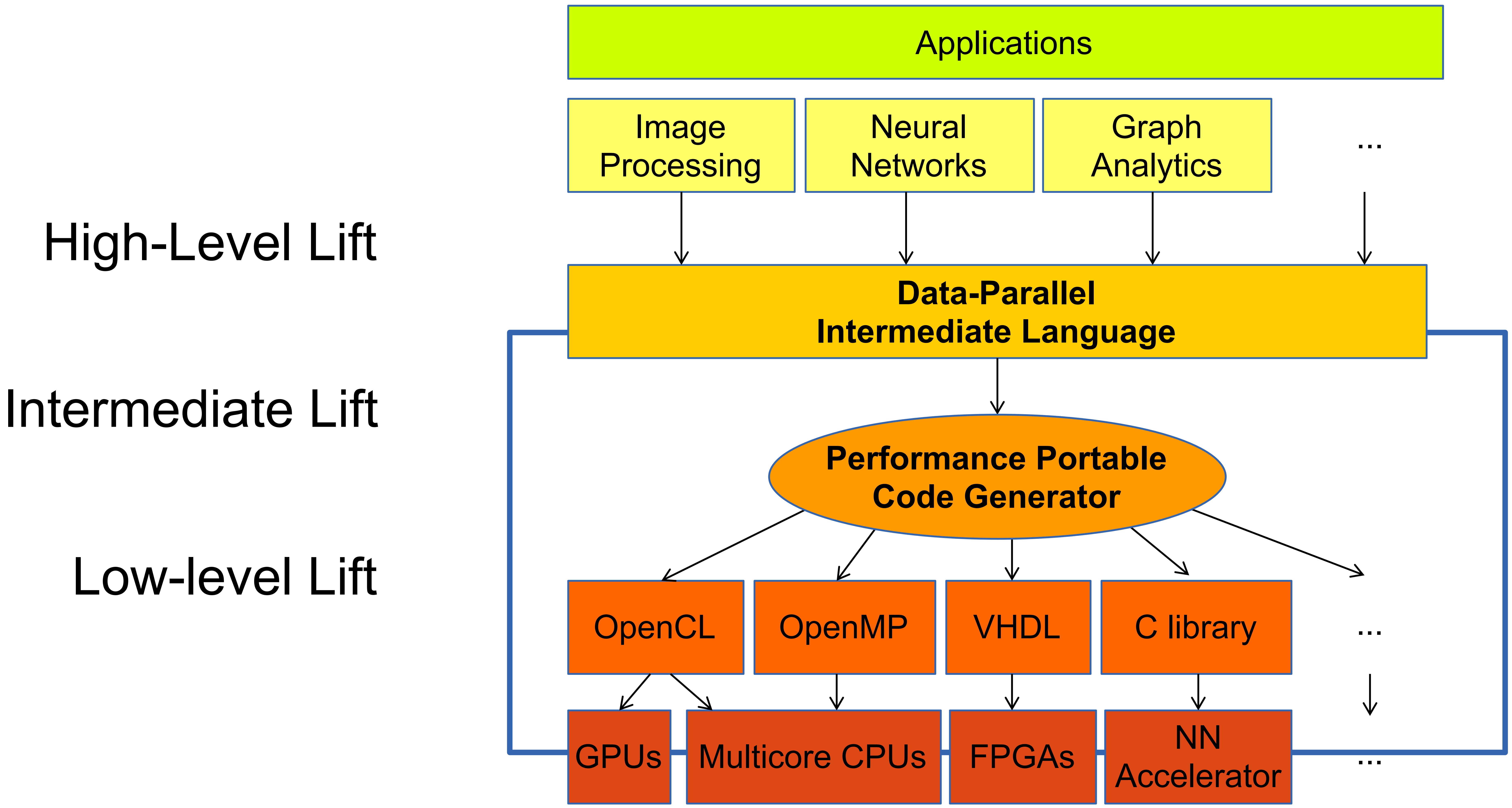
What we need


Applications
Domain Specific Languages (DSLs)
Image Processing
Neural Networks
Graph Analytics
...
Language for Parallelism
Data-Parallel Intermediate Language
Compiler Technology
Performance Portable Code Generator
OpenCL
OpenMP
VHDL
C library
...
GPUs
Multicore CPUs
FPGAs
NN Accelerator
...

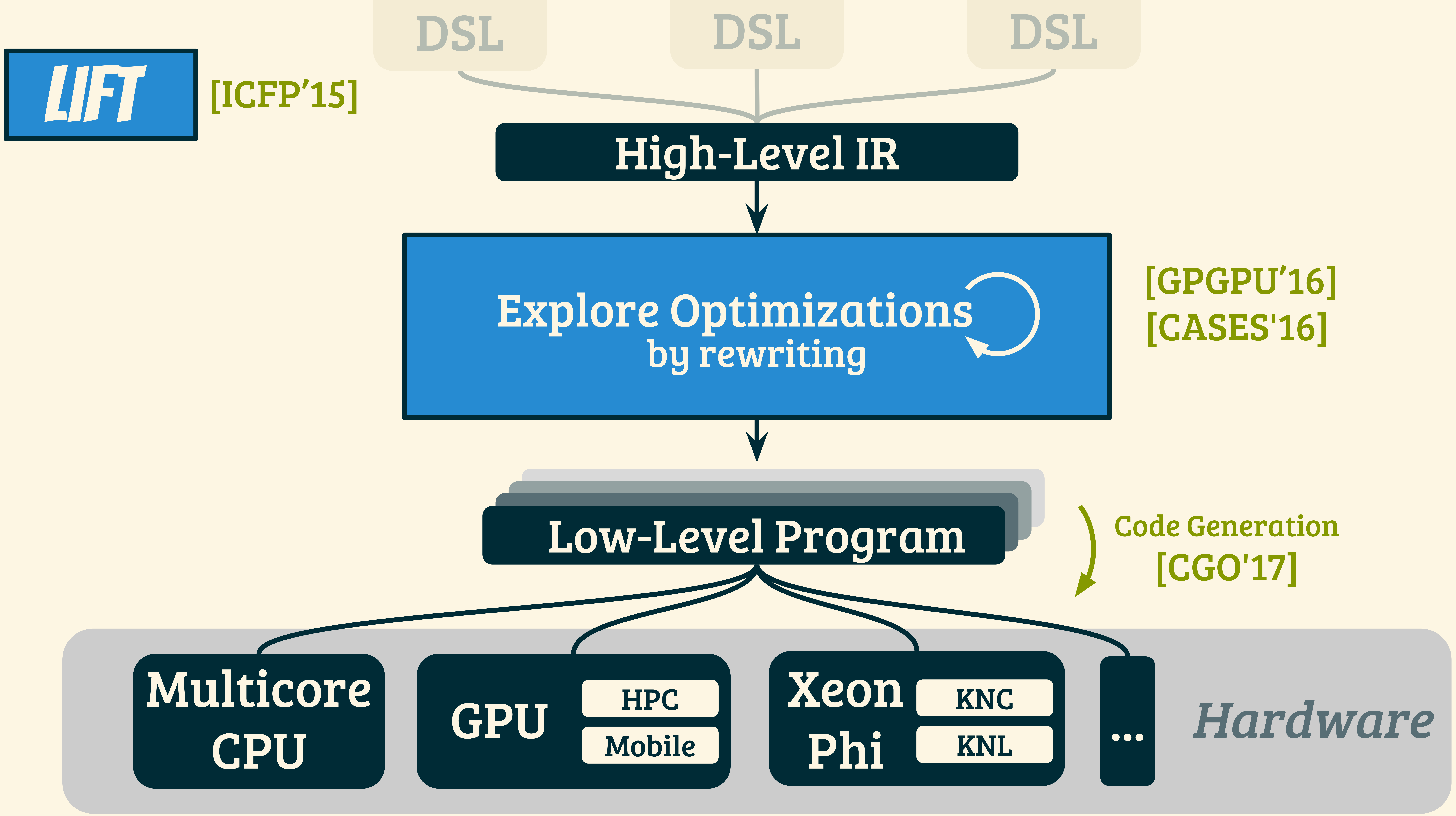
Bottom up Approach



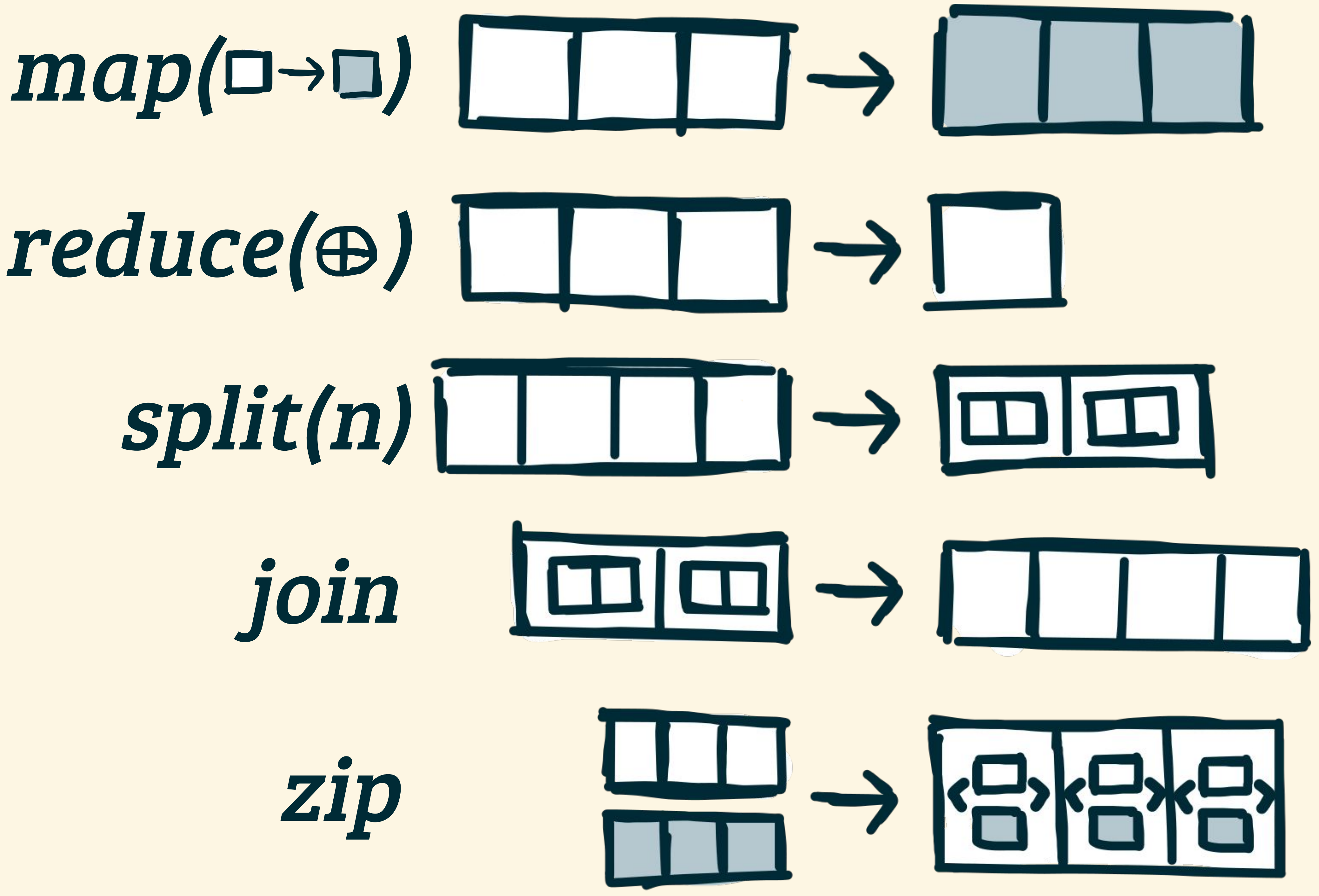
Lift: Layered language Approach



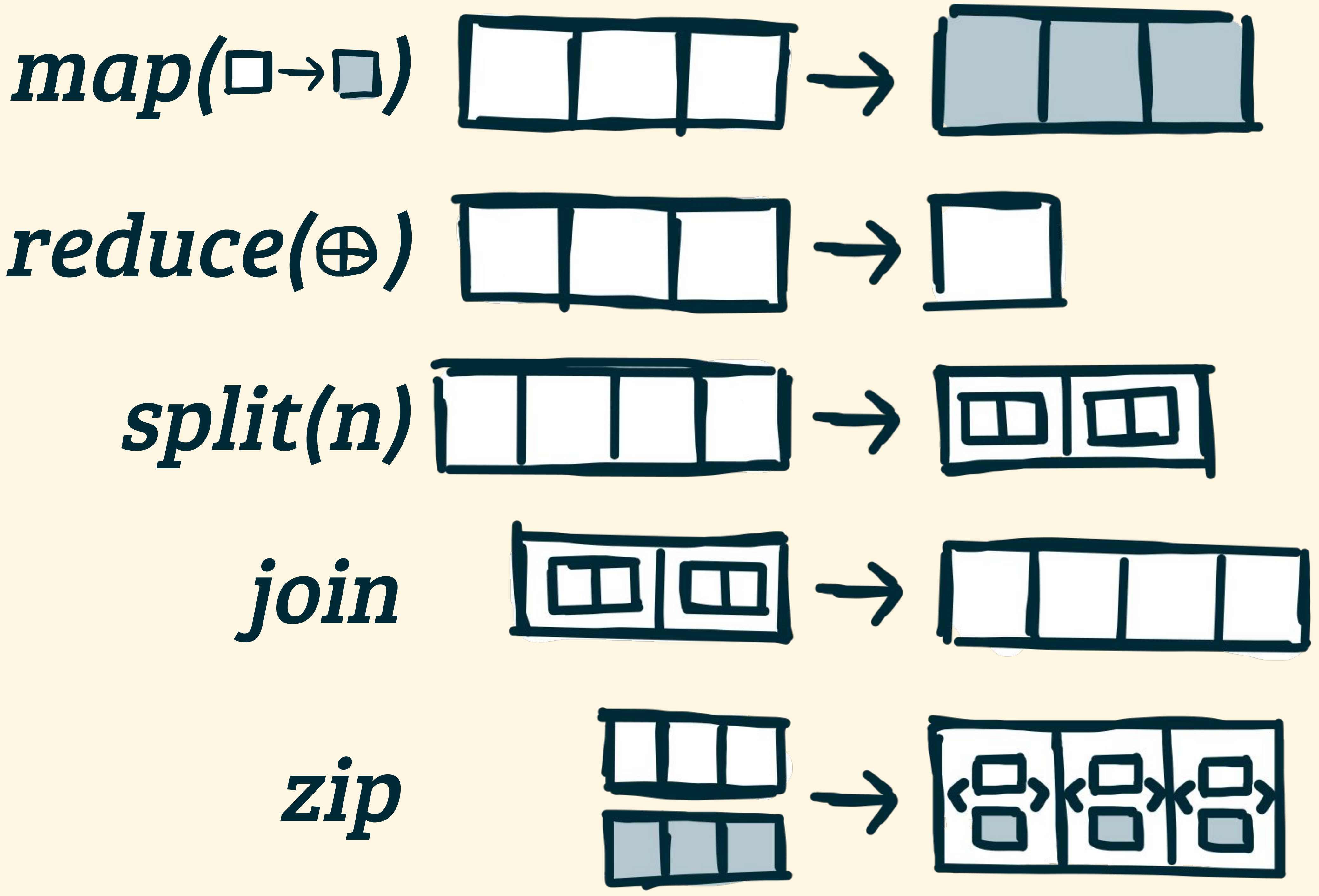

LIFT
2. HIGH-LEVEL PROGRAMMING
1. LOW-LEVEL OPTIMIZATIONS
G. HIGH PERFORMANCE


DSL
DSL
DSL
LIFT
[ICFP'15]
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

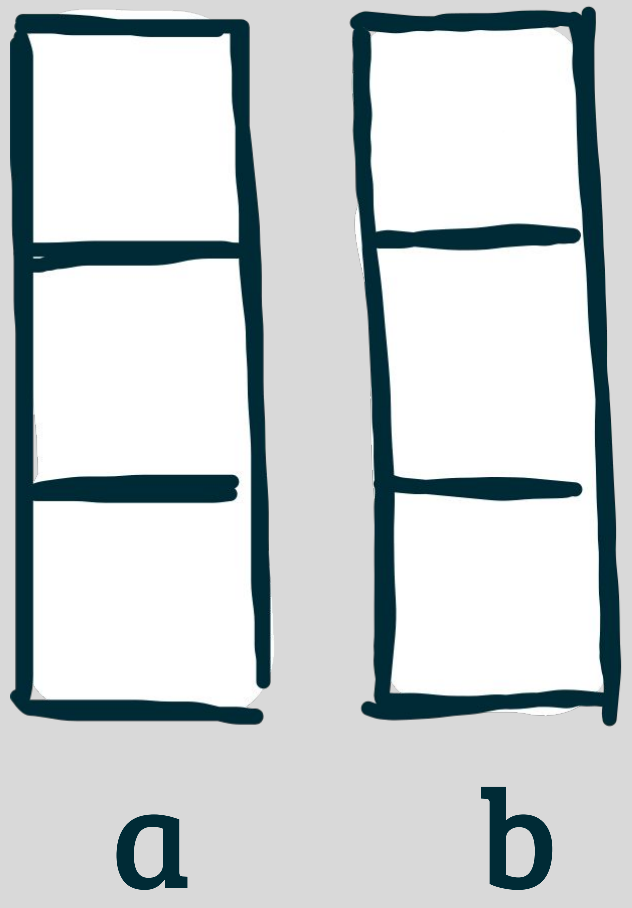
LIFT'S HIGH-LEVEL PRIMITIVES



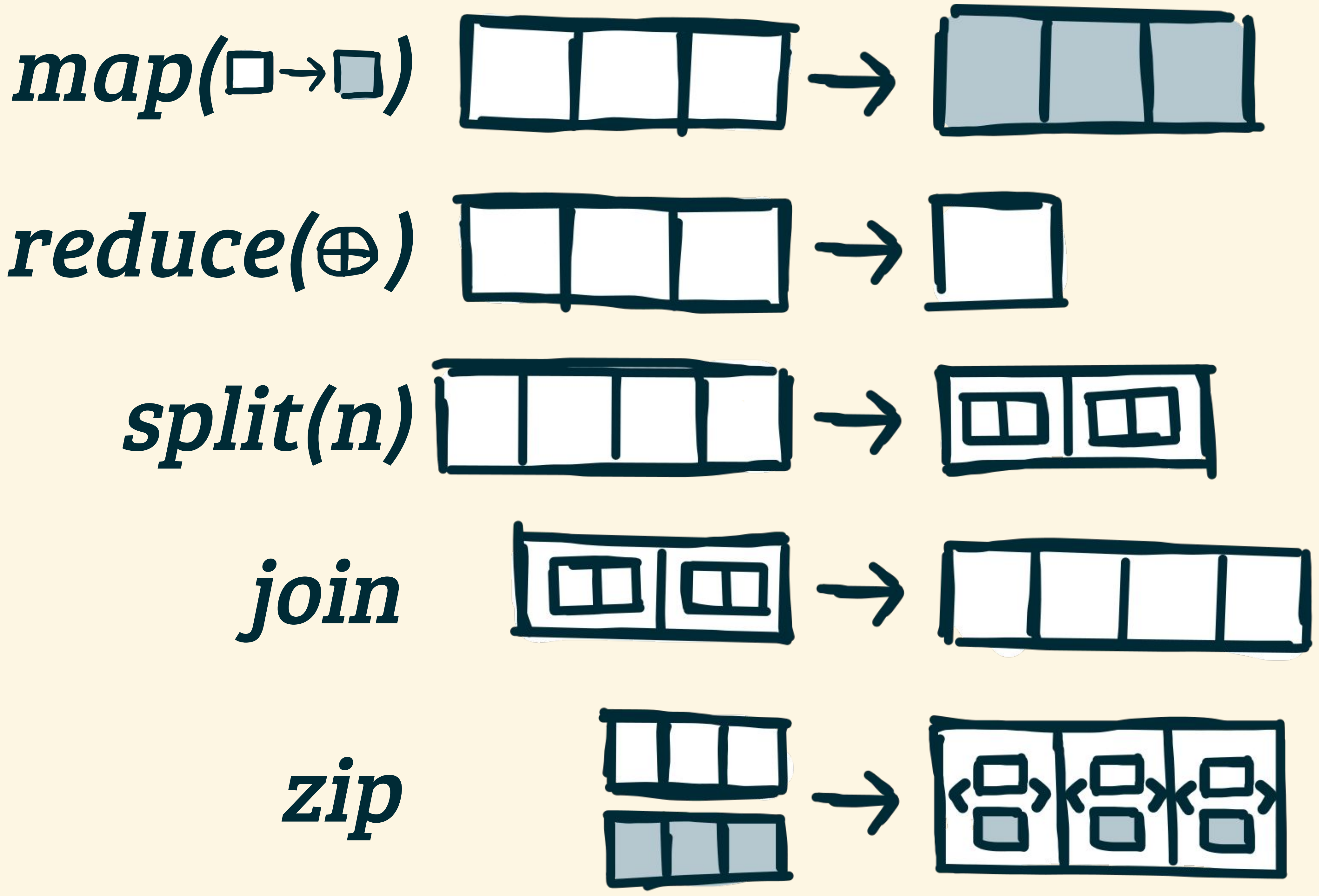
LIFT'S HIGH-LEVEL PRIMITIVES

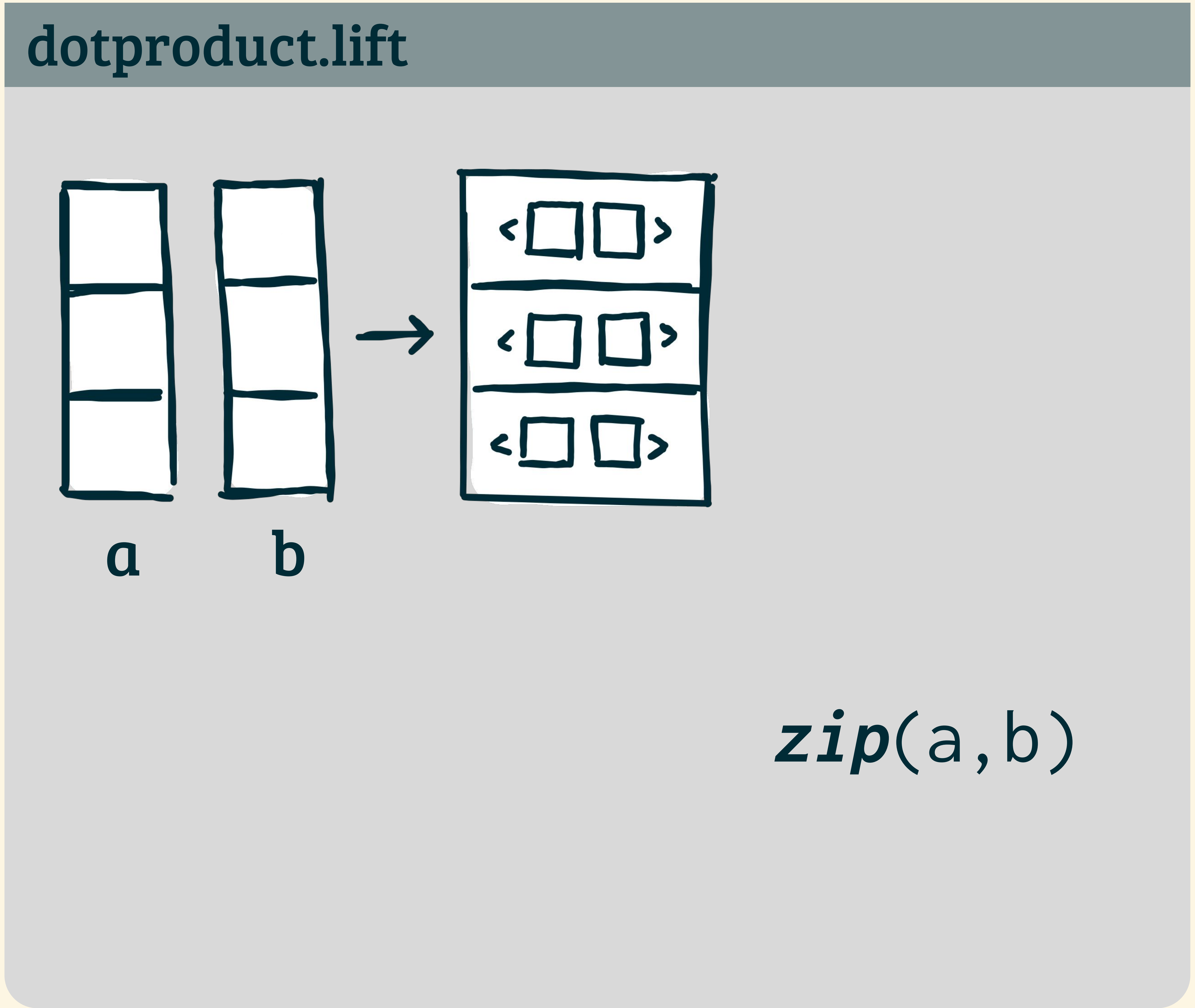


dotproduct.lift

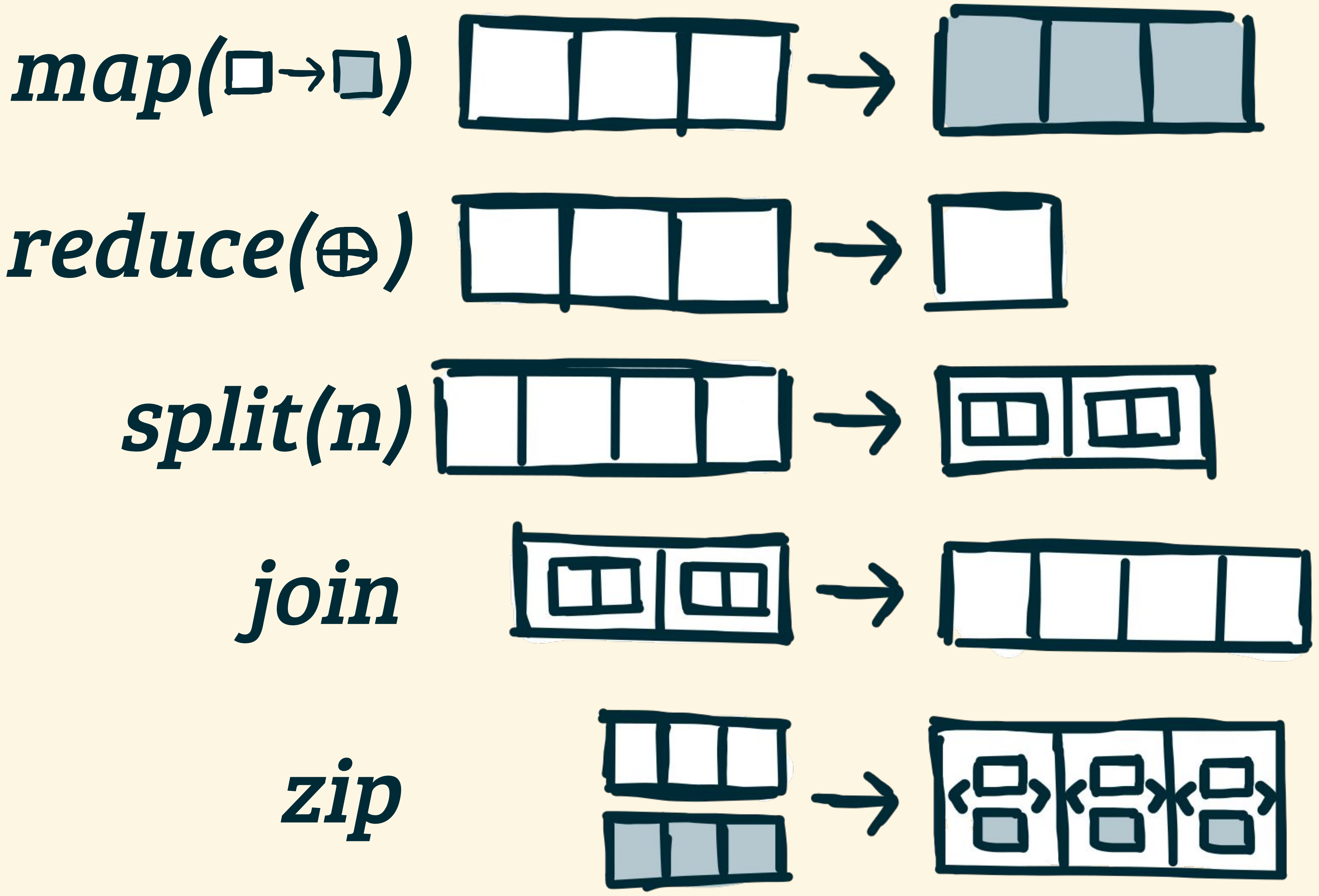


LIFT'S HIGH-LEVEL PRIMITIVES

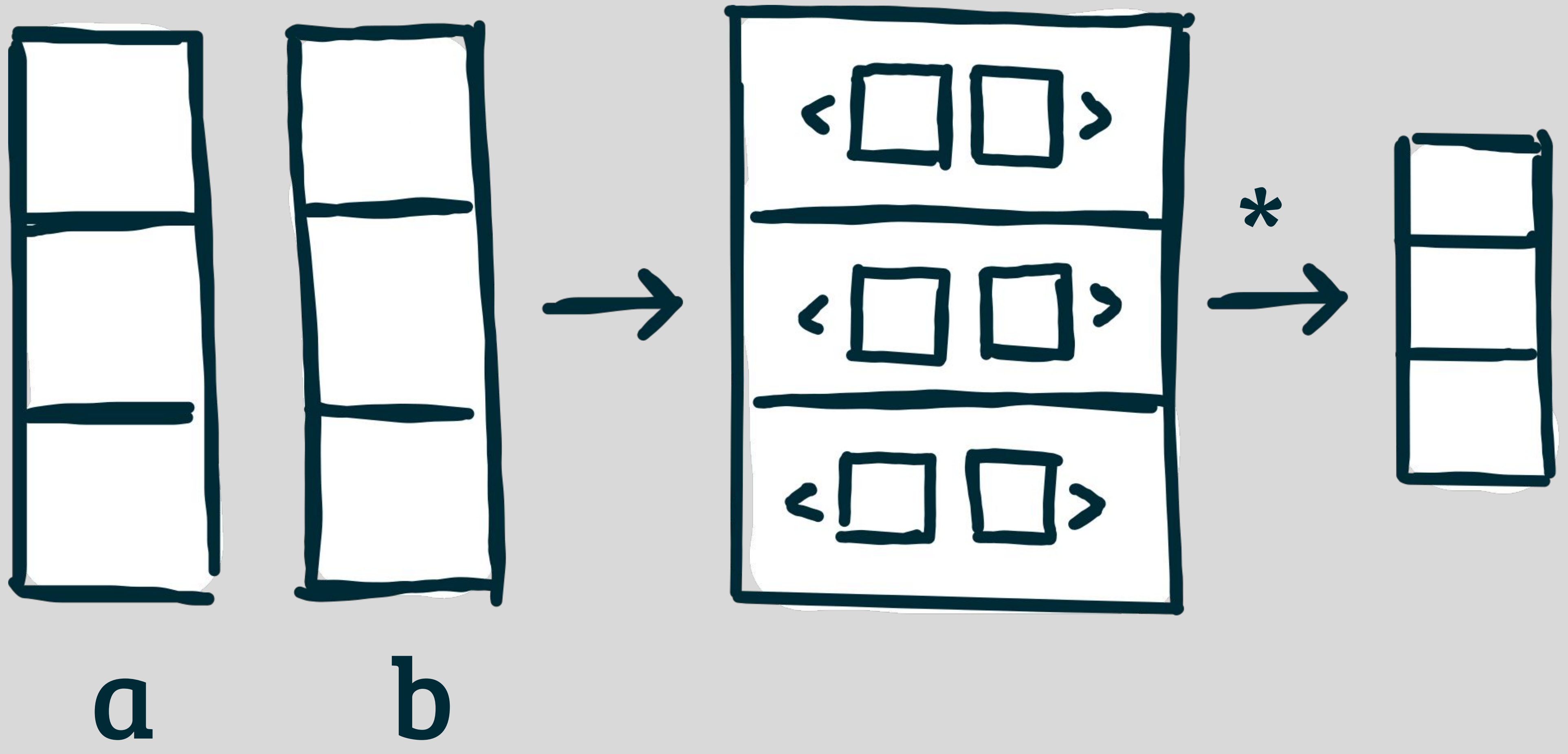




LIFT'S HIGH-LEVEL PRIMITIVES



dotproduct.lift




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

LIFT'S HIGH-LEVEL PRIMITIVES

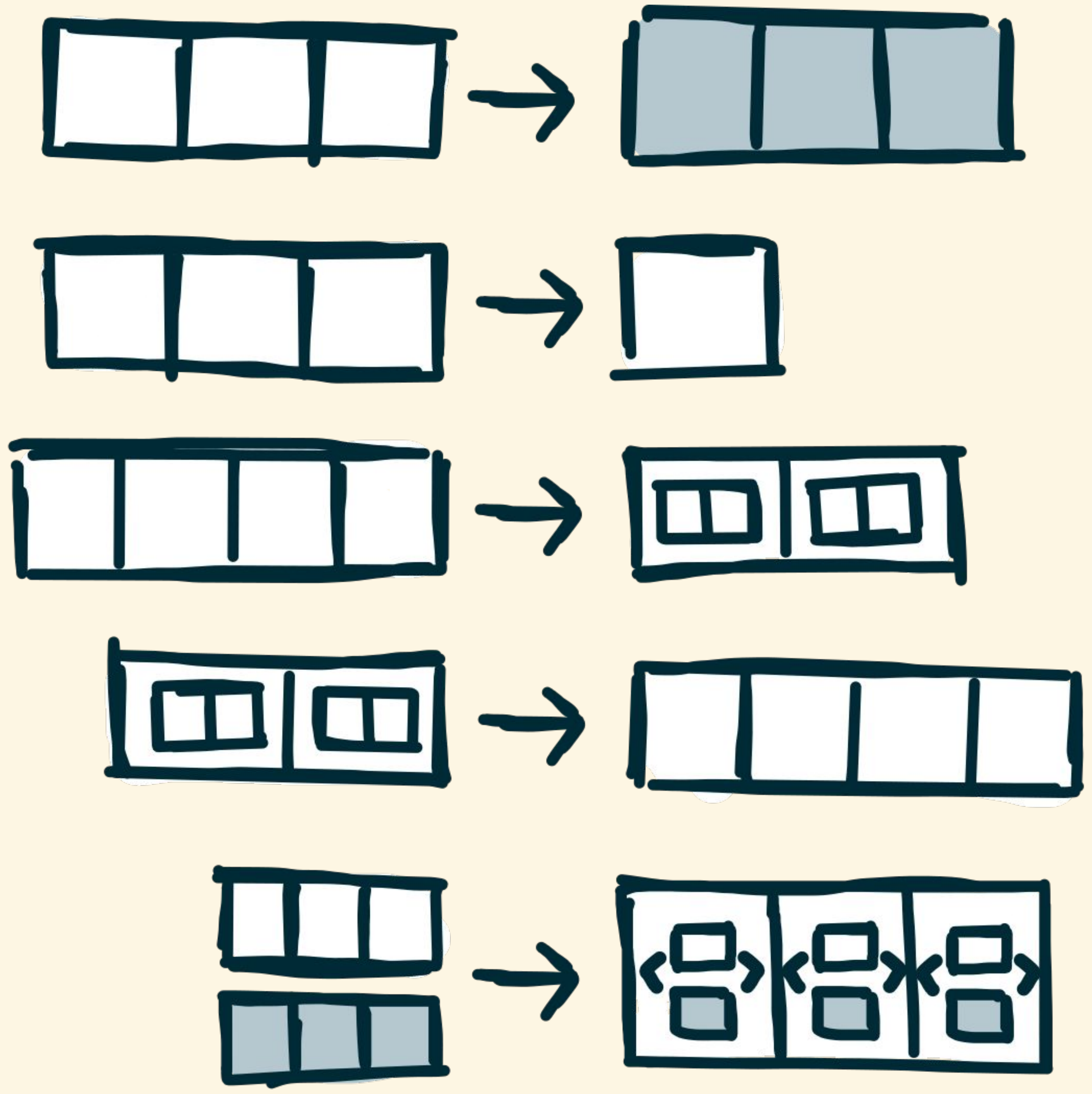
map(□→□)

reduce(⊕)

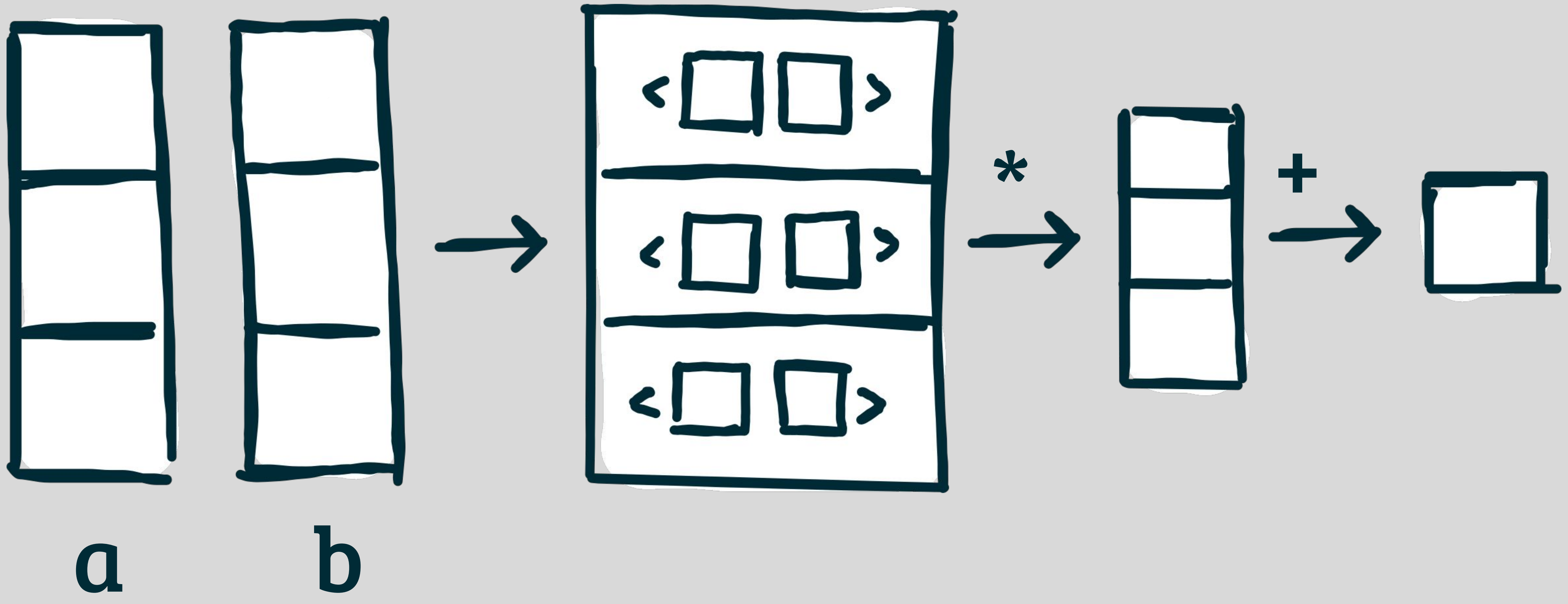
split(n)

join

zip

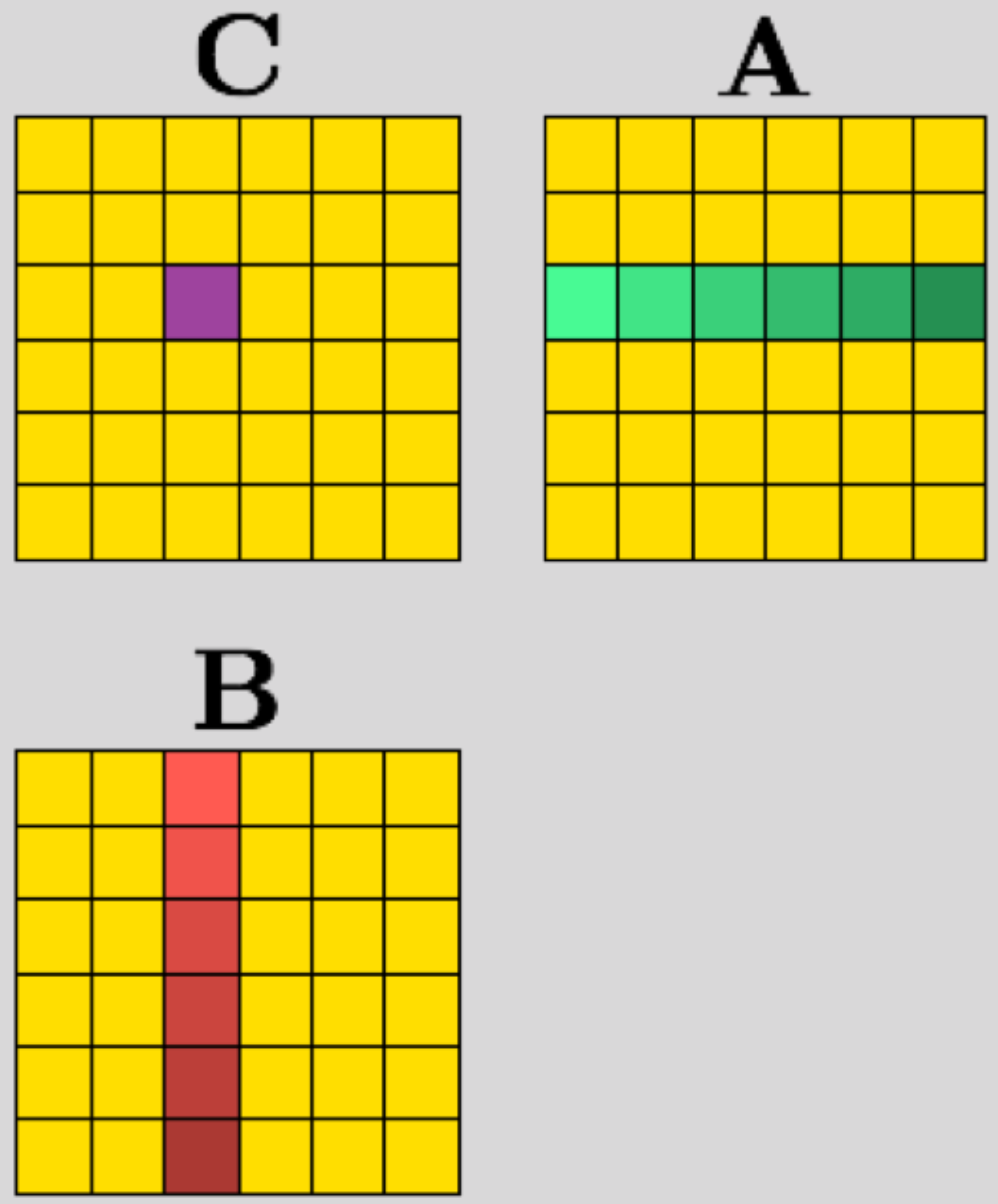


dotproduct.lift




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

LIFT'S HIGH-LEVEL PRIMITIVES



matrixMult.lift

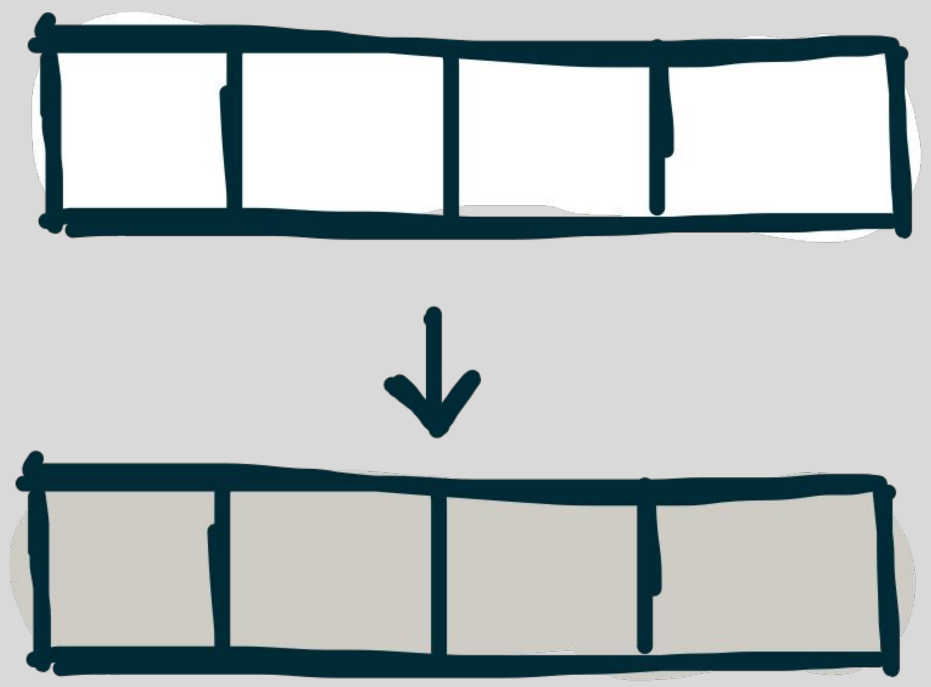
```
map(λ rowA ↦
   map(λ colB ↦
      dotProduct(rowA, colB)
     , transpose(B))
  , A)
```


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

Implementation Choices as Rewrite Rules

Divide & Conquer

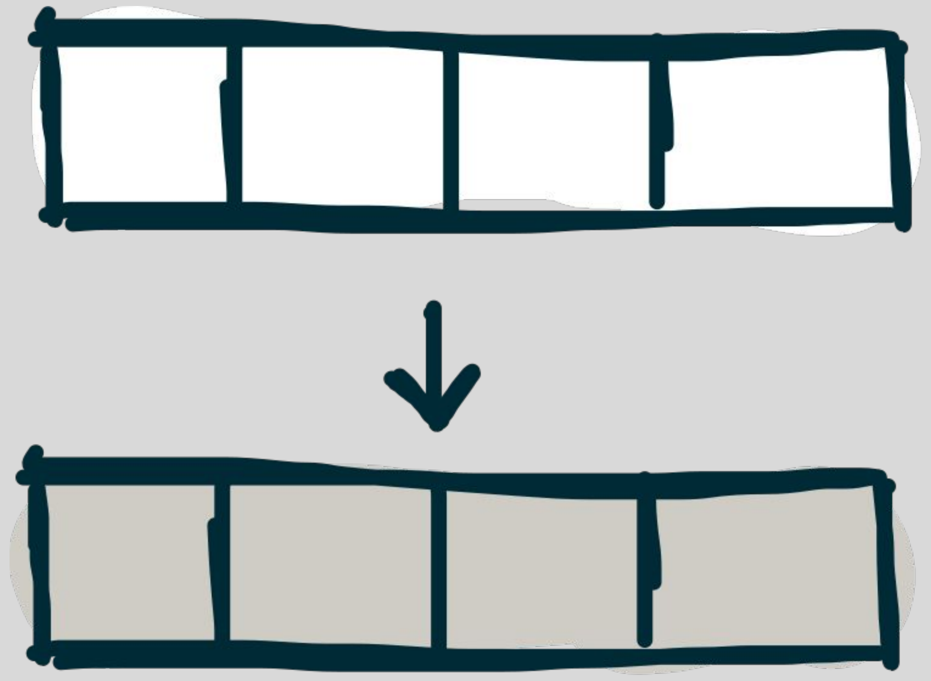
map(f, A)

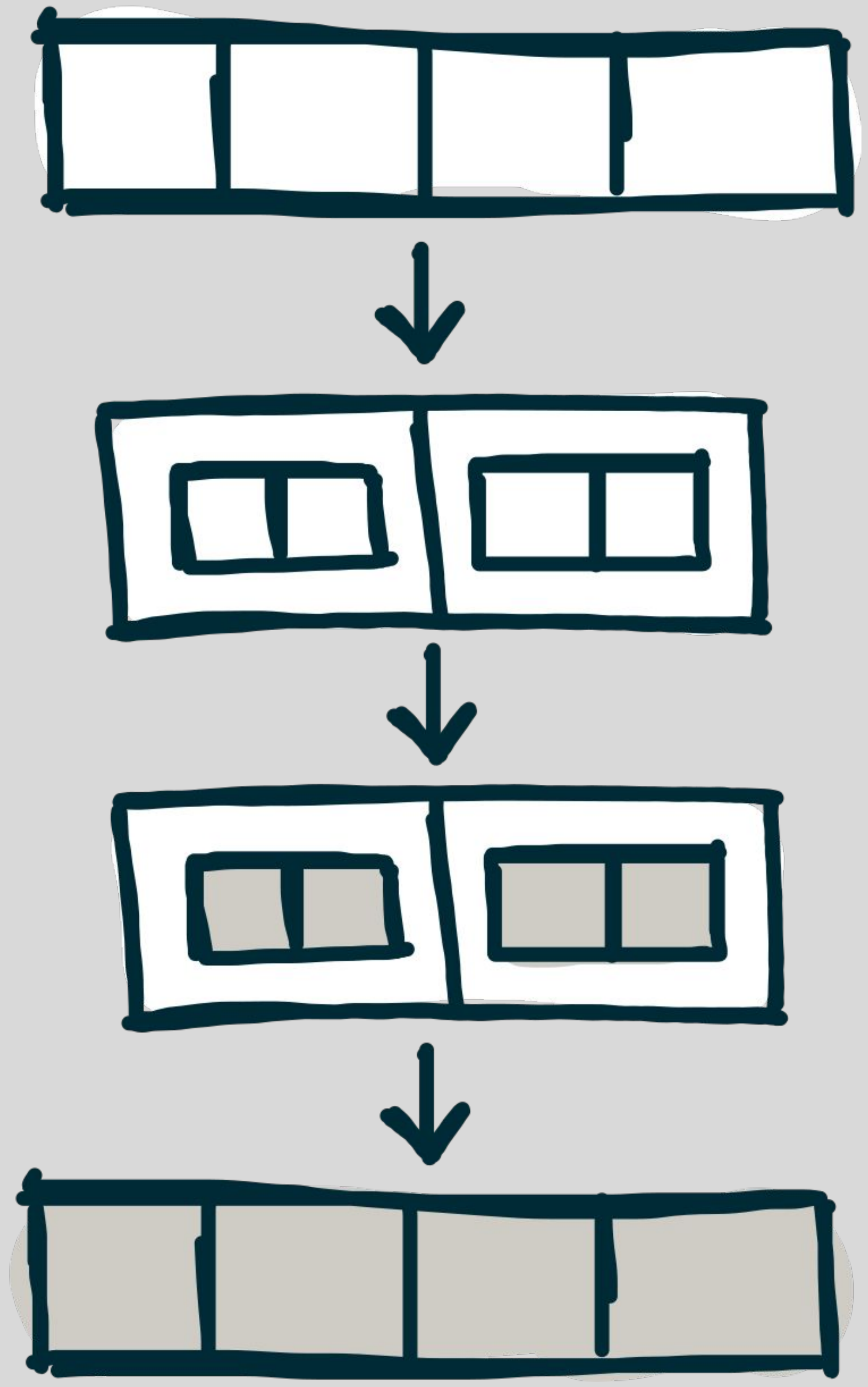


IMPLEMENTATION CHOICES AS REWRITE RULES

Divide & Conquer

$map(f, A) \mapsto join(map(map(f), split(n, A)))$





Correctness of Rewrite Rules

$$join\ (map\ (map\ f)\ (split\ n\ [x_1, \ldots, x_n]))$$

def. of *split*

$$= join\ (map\ (map\ f)\ \big[[x_1, \ldots, x_n], \ldots, [x_{m-n}, \ldots, x_m]\big])$$

def. of *map*

$$= join\ \big[(map\ f)\ [x_1, \ldots, x_n], \ldots, (map\ f)\ [x_{m-n}, \ldots, x_m]\big]$$

def. of *map*

$$= join\ \big[[f\ x_1, \ldots, f\ x_n], \ldots, [f\ x_{m-n}, \ldots, f\ x_m]\big]$$

def. of *join*

$$= [f\ x_1, \ldots, f\ x_n] = map\ f\ [x_1, \ldots, x_n]$$

See also: *The Algebra of Programming* by Richard Bird and Oege De Moor

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, x) ↦ mapGlobal(f, x) | mapWorkgroup(f, x) | mapLocal(f, x) | mapSeq(f, x)

Local / global memory:

mapLocal(f, x) ↦ toLocal(mapLocal(f, x)) mapLocal(f, x) ↦ toGlobal(mapLocal(f, x))

Vectorization:

map(f, x) ↦ joinVec(map(mapVec(f), splitVec(n, x)))

Optimizations as Macro Rules

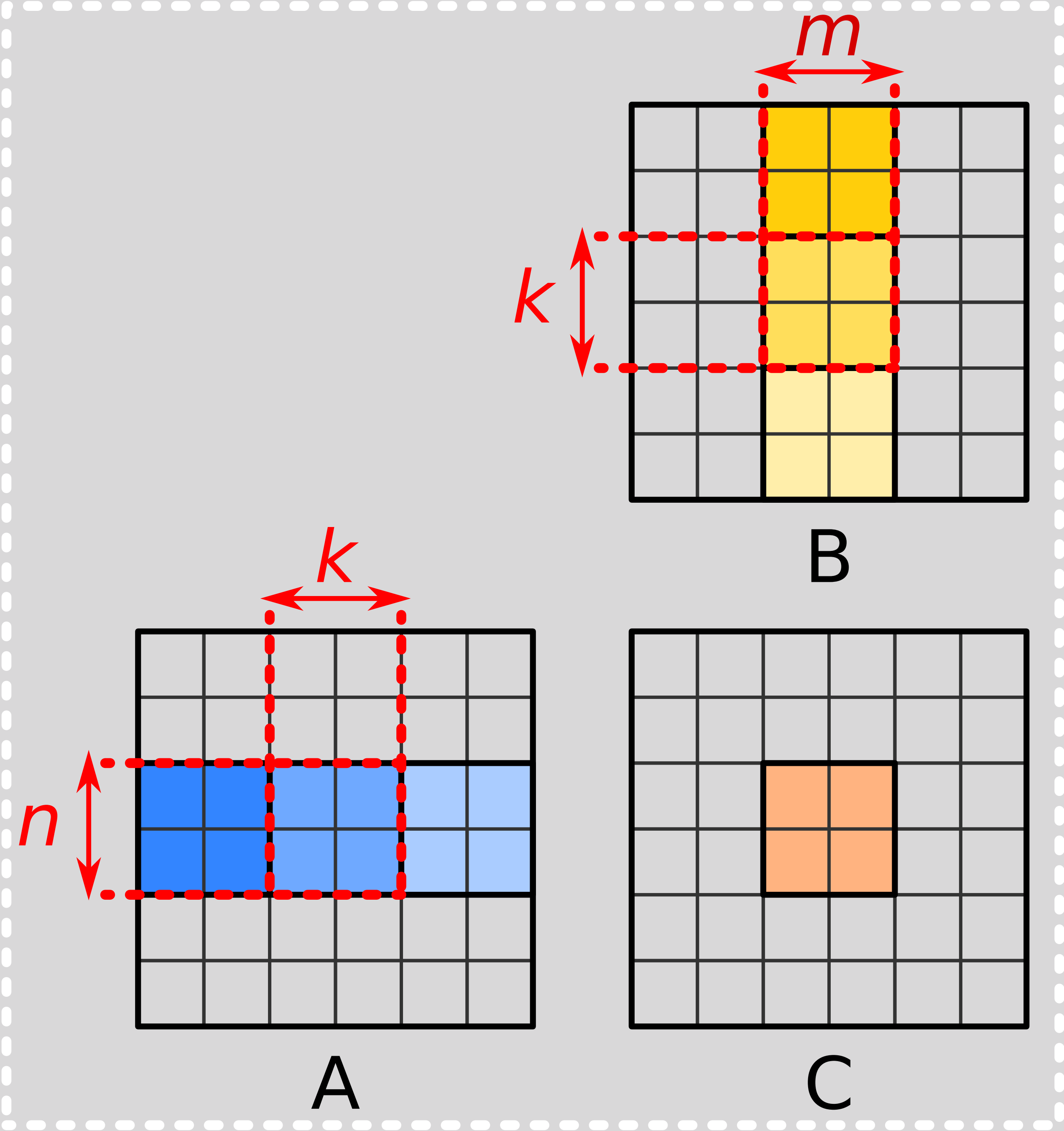
2D Tiling

Naïve matrix multiplication

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

↓ Apply tiling rules

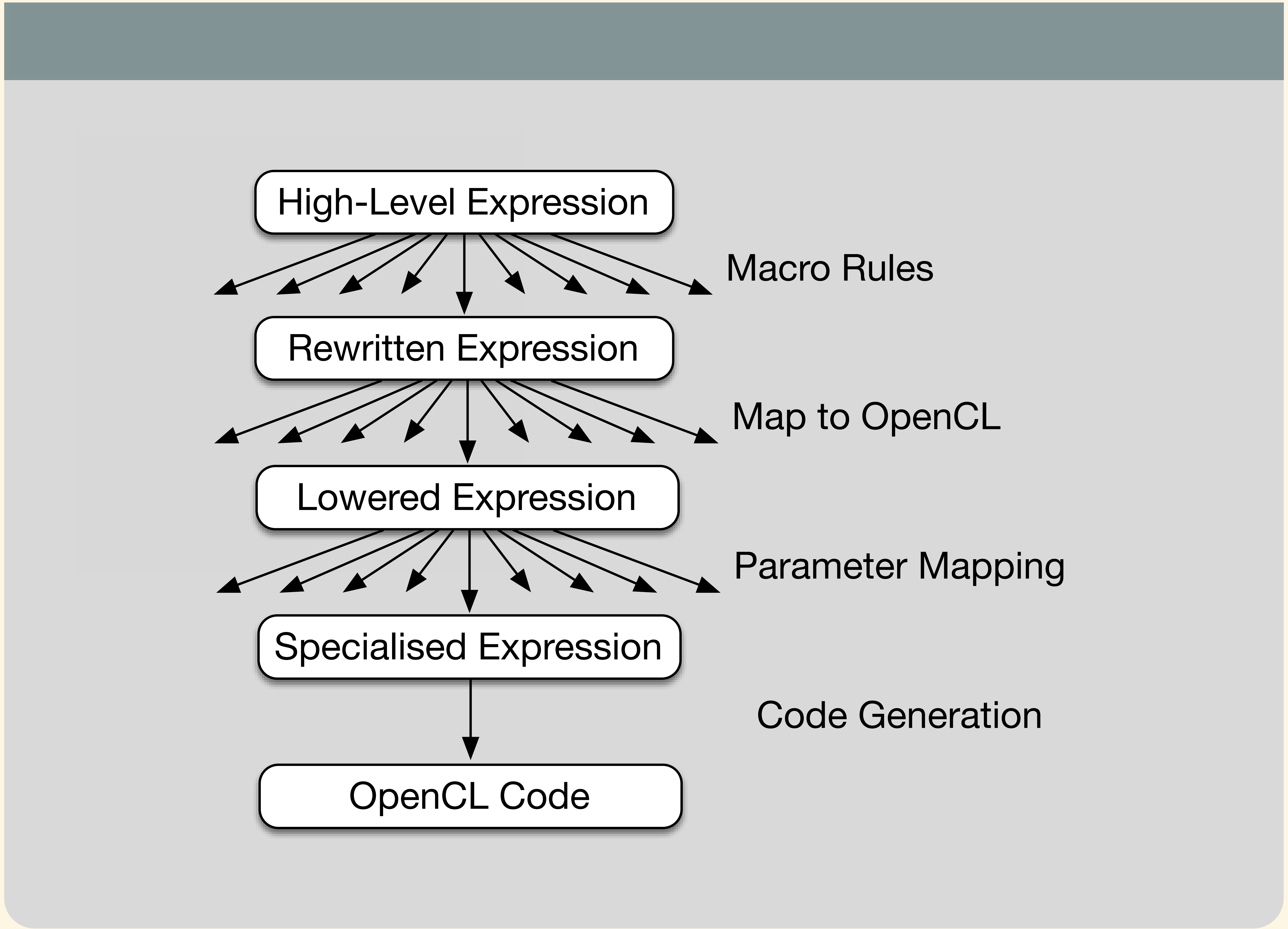
```
untile ∘ map(λ rowOfTilesA .
  map(λ colOfTilesB .
    toGlobal(copy2D) ∘
    reduce(λ (tileAcc, (tileA, tileB)) .
      map(map(+)) ∘ zip(tileAcc) ∘
      map(λ as .
        map(λ bs .
          reduce(+, 0) ∘ map(×) ∘ zip(as, bs)
        , toLocal(copy2D(tileB)))
      , toLocal(copy2D(tileA)))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



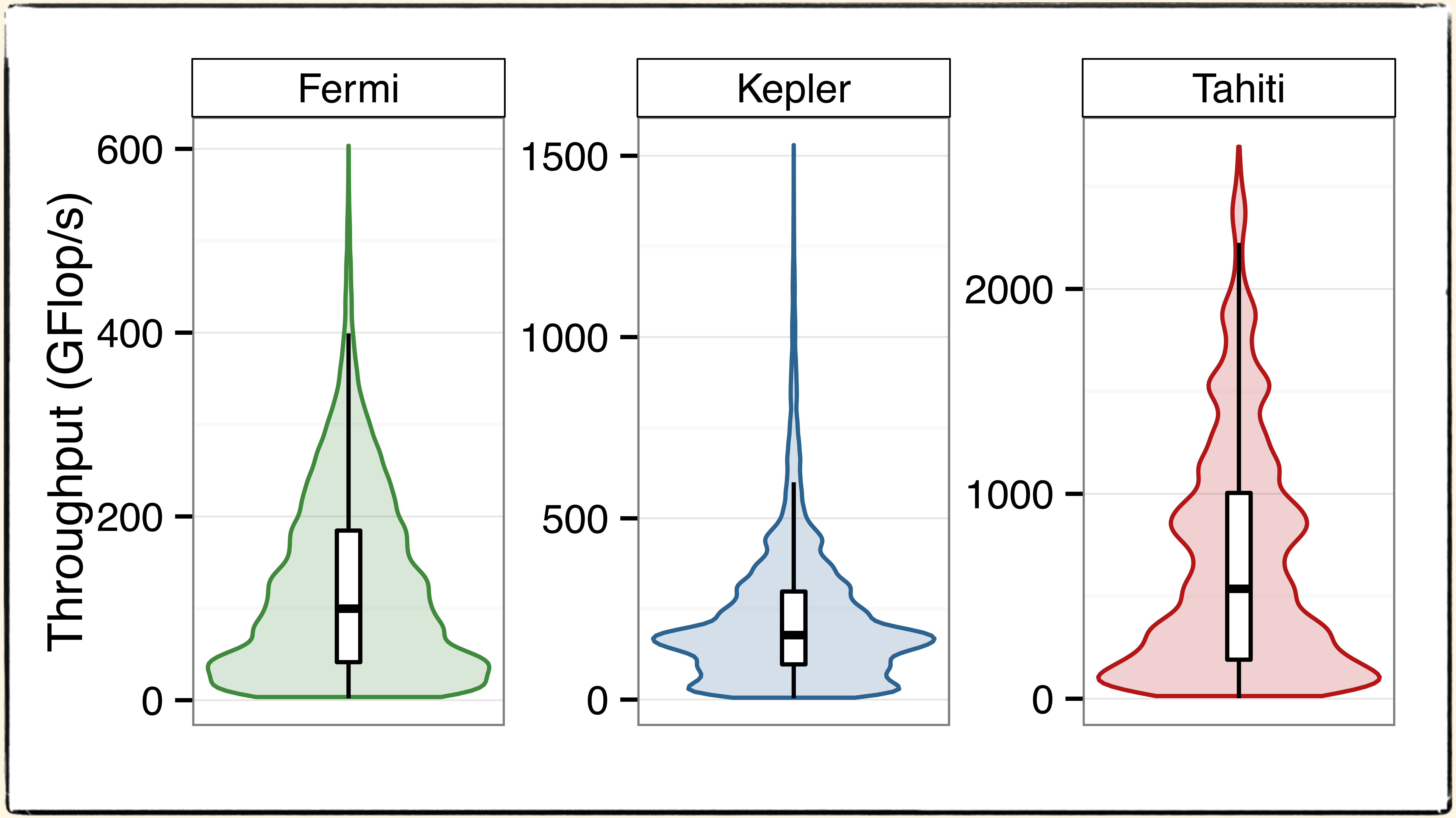
[GPGPU'16]


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

Exploration by Rewriting



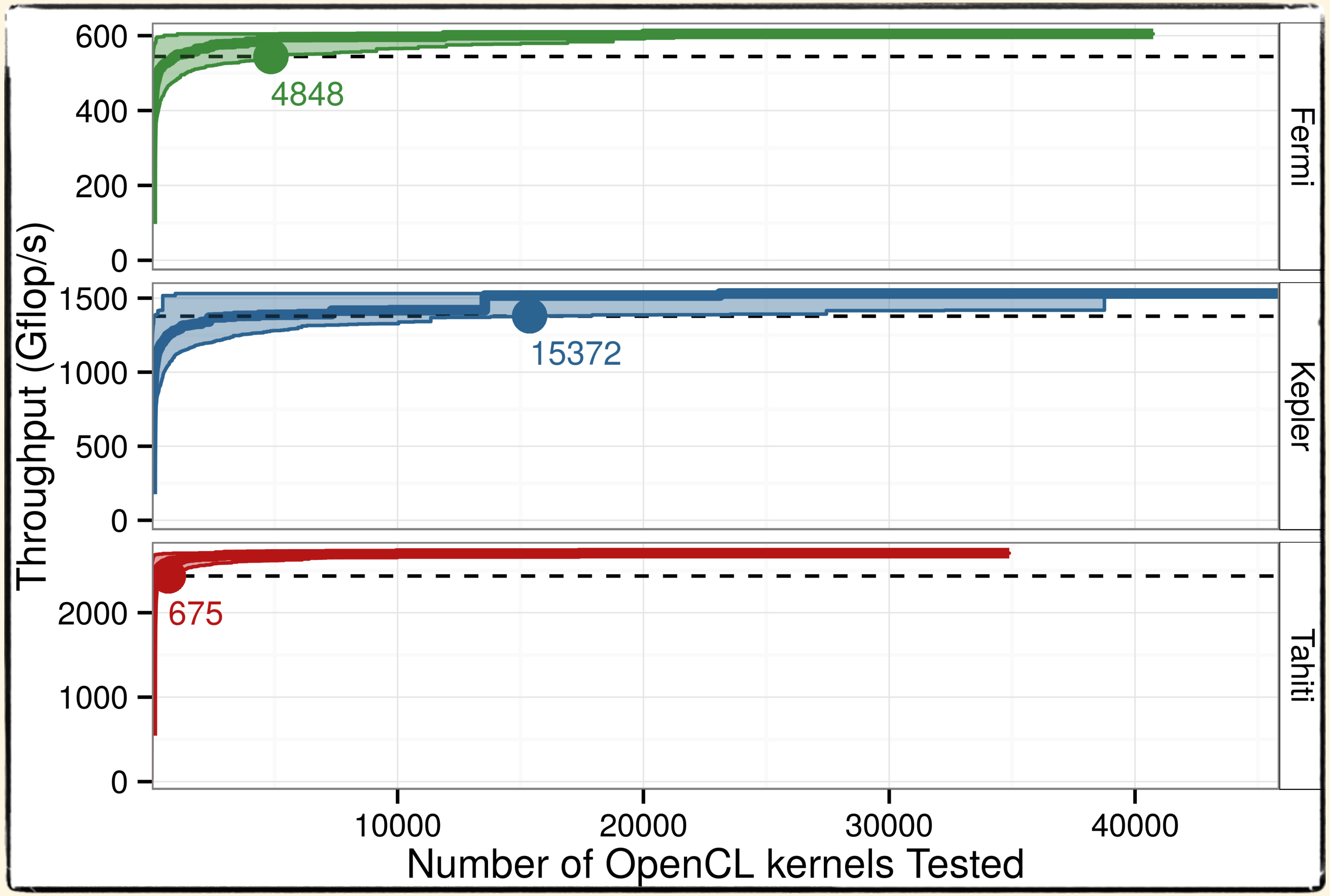
Exploration Space Matrix Multiplication



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]

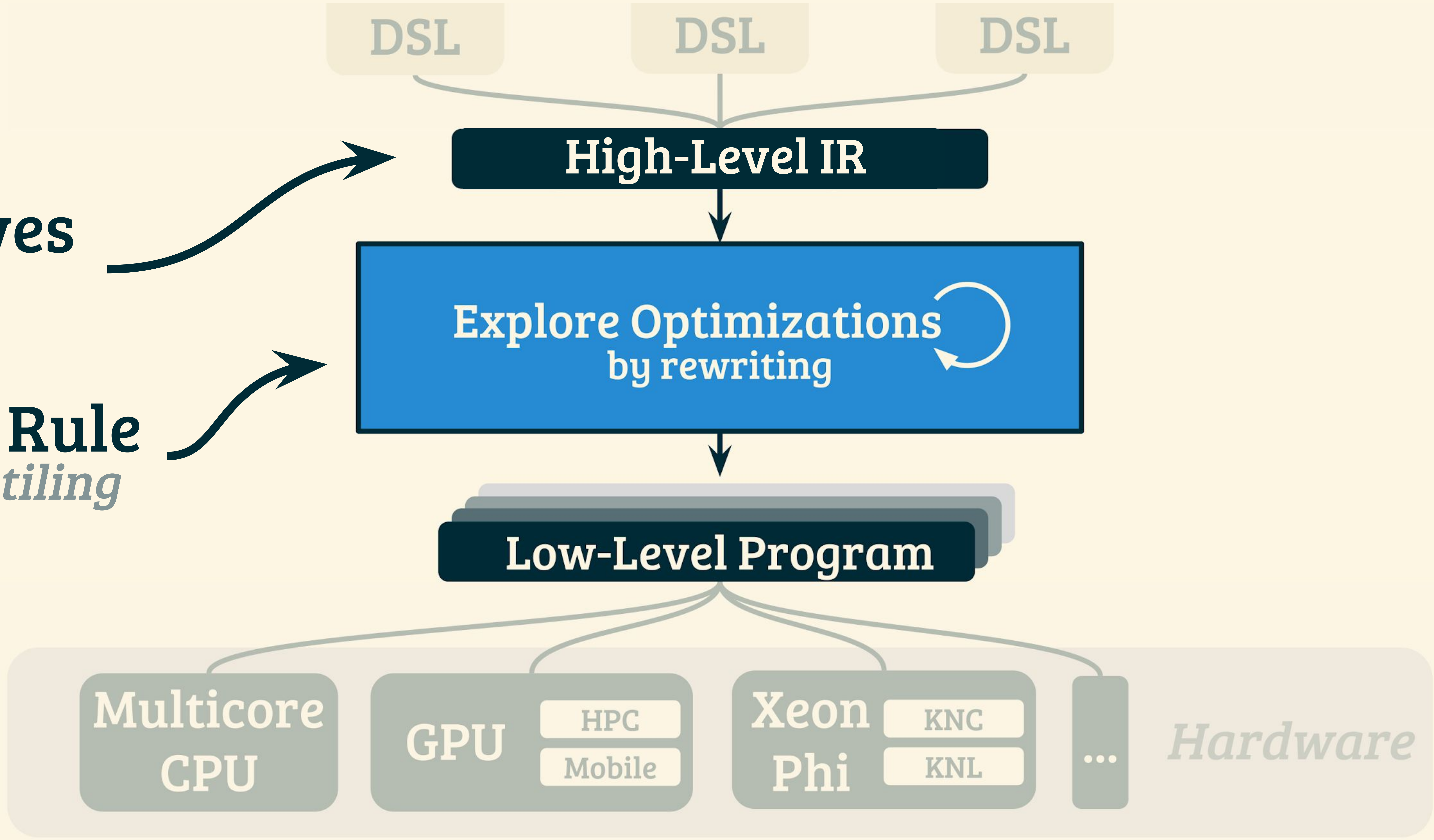
STENCIL COMPUTATIONS IN LIFT

[CGO'18] Best Paper Award

We added:

2 Primitives
pad, *slide*

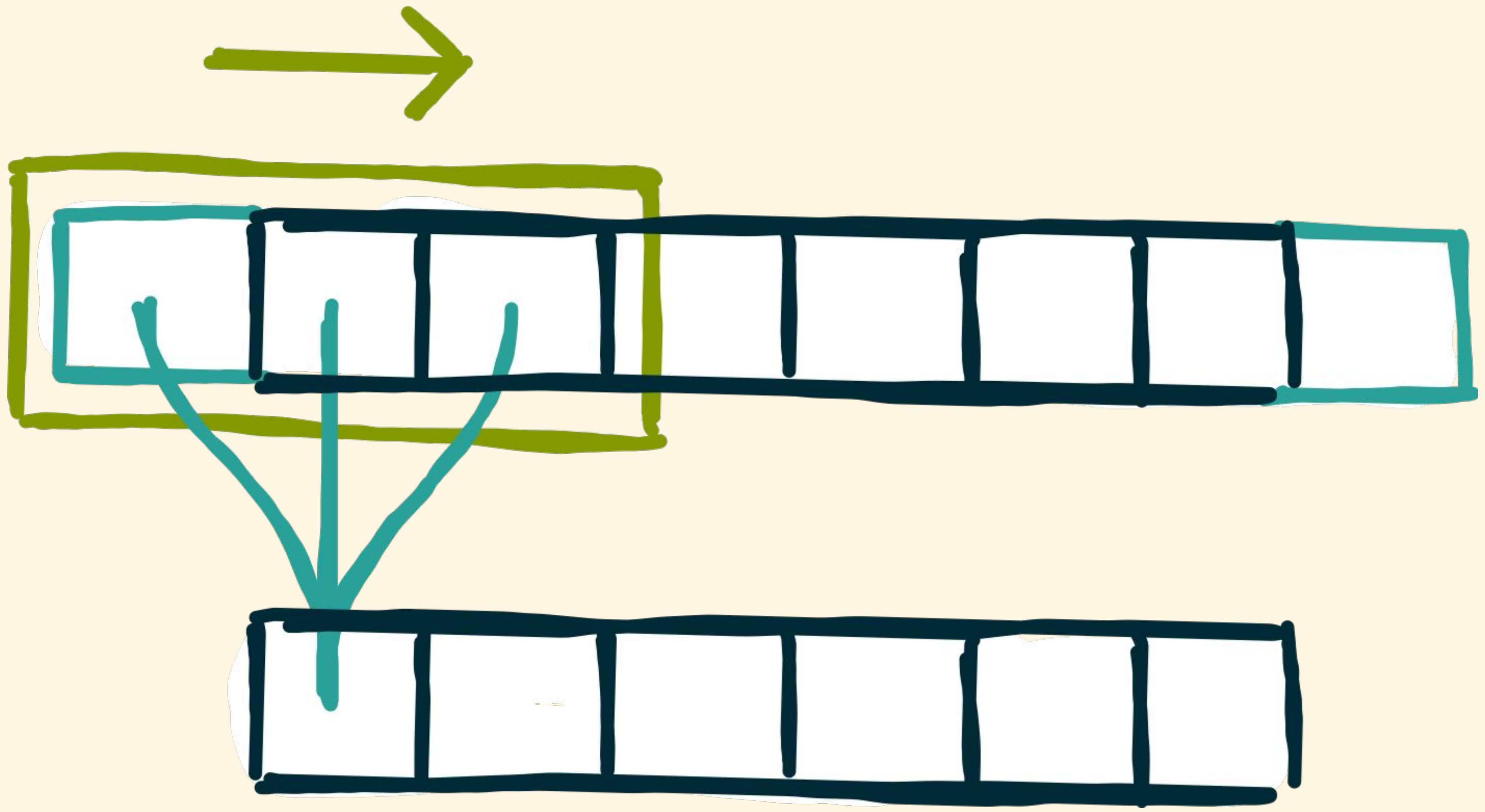
1 Rewrite Rule
overlapped tiling



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 ; }
```

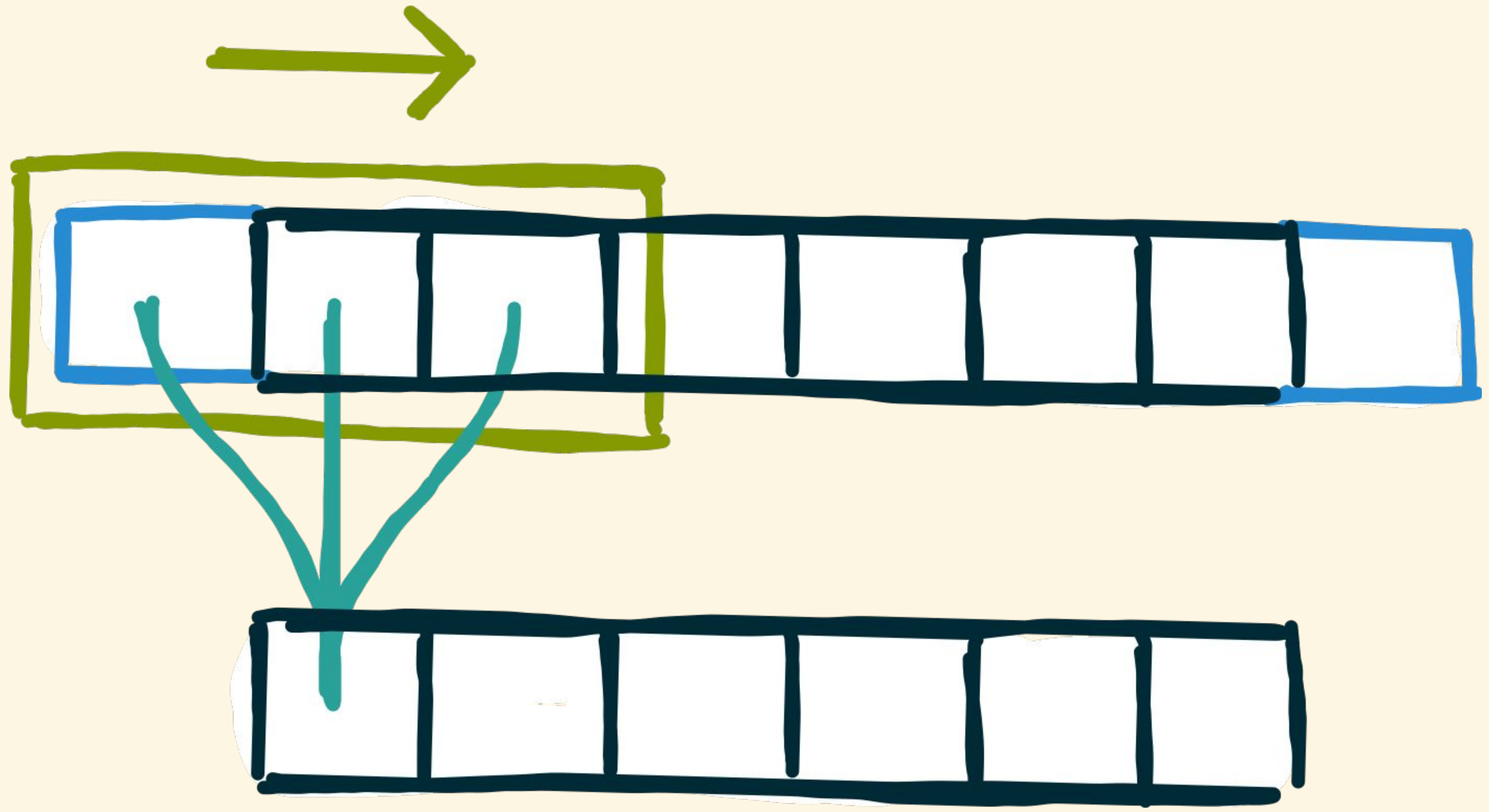


(a) access neighborhoods for every element

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 ; }
```



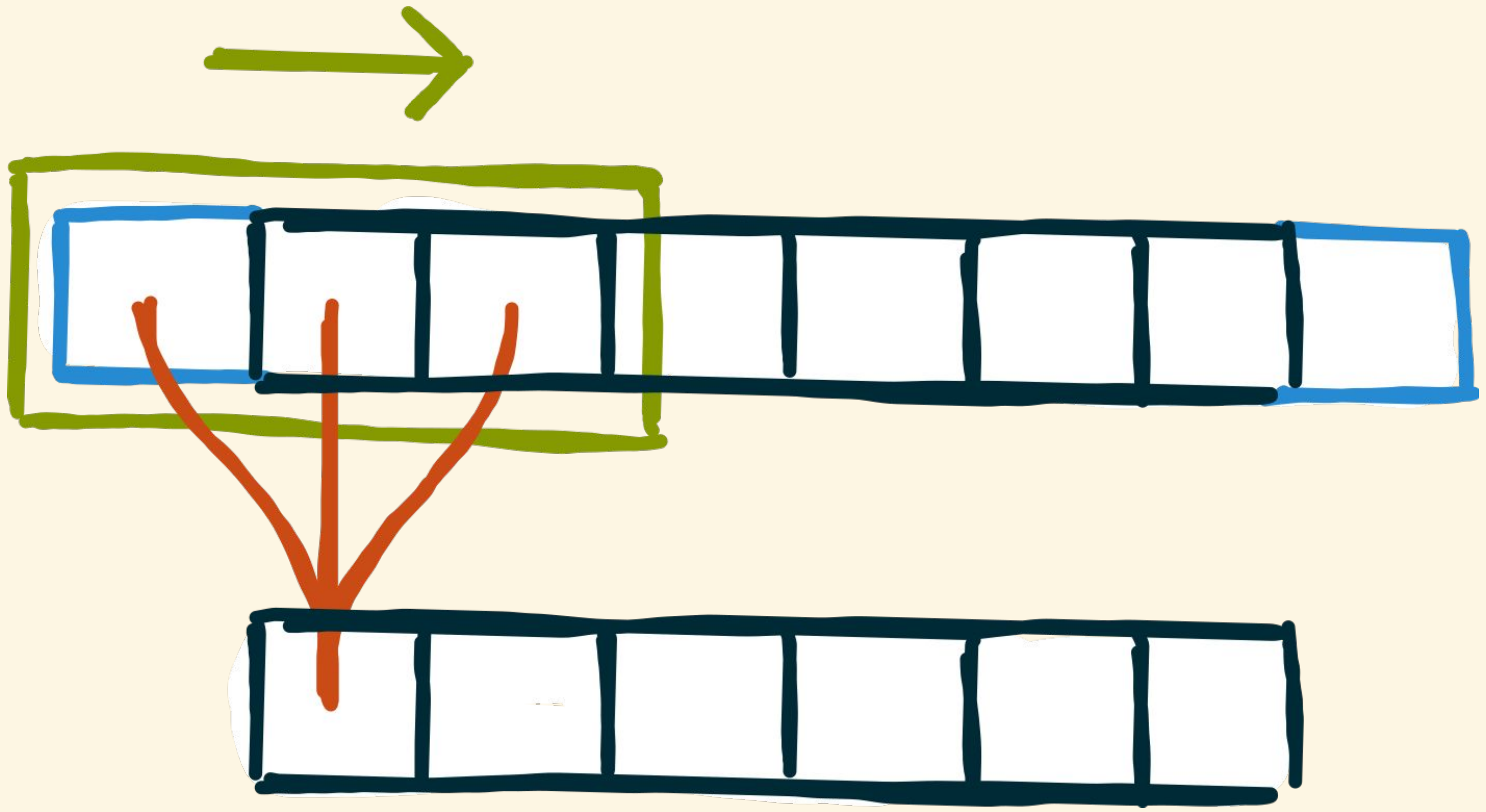
(a) access **neighborhoods** for every element

(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 ; }
```

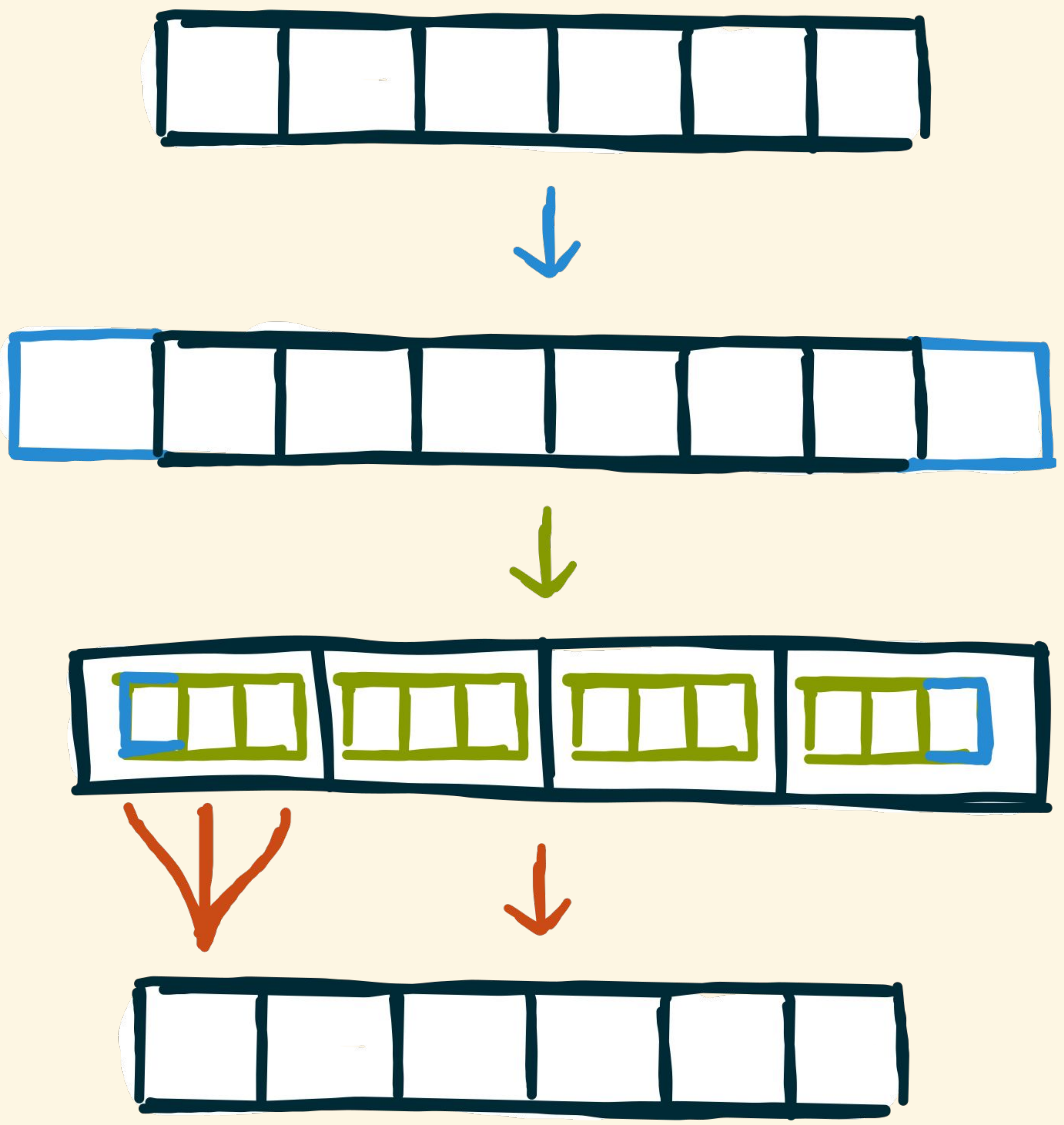


(a) access neighborhoods for every element
(b) specify boundary handling
(c) apply stencil function to neighborhoods

DECOMPOSING STENCIL COMPUTATIONS

3-point-stencil.c

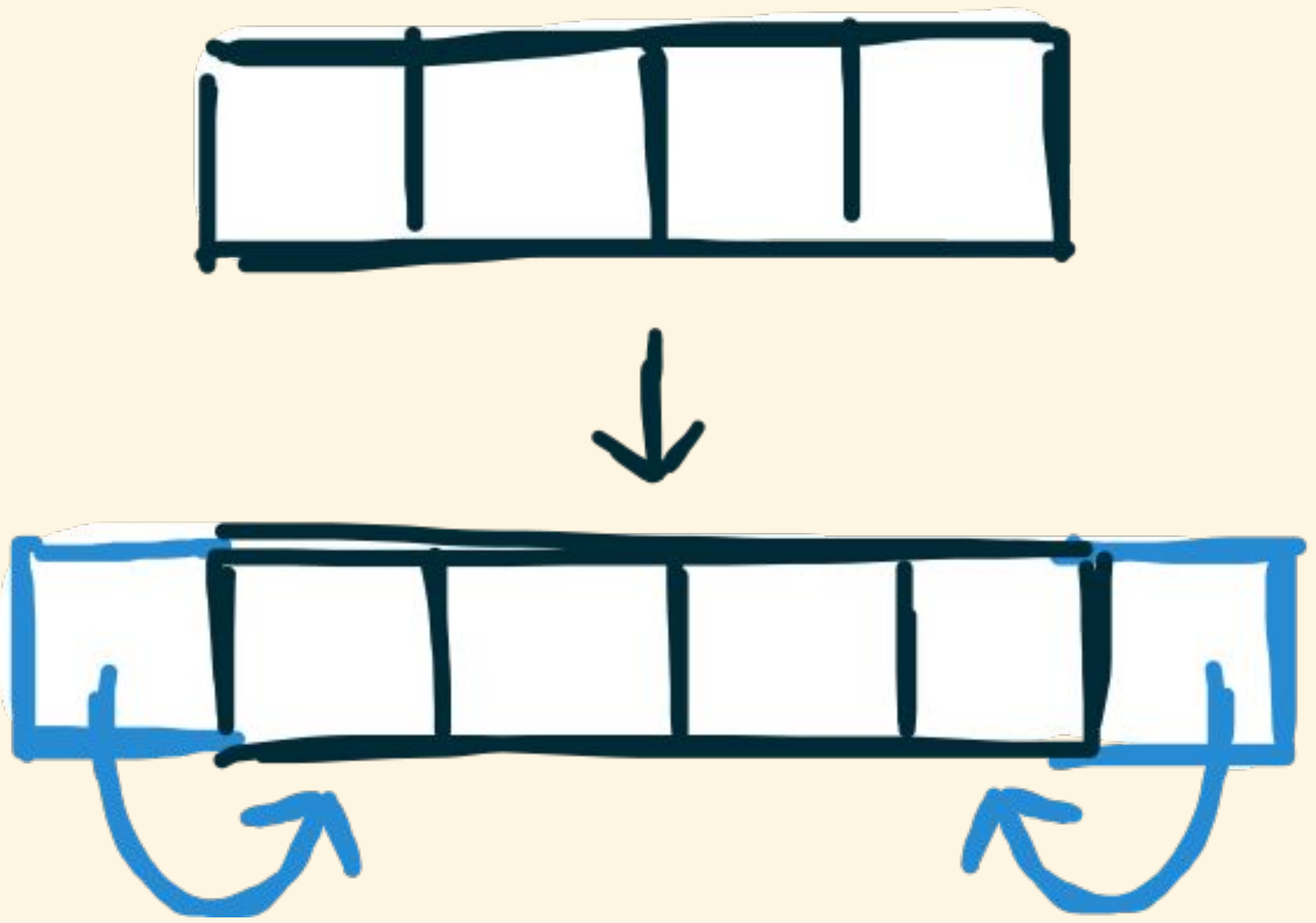
```
for (int i = 0; i < N ; i ++) {
    int sum = 0;
    for ( int j = -1; j <= 1; j ++) {   // ( a )
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;        // ( b )
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[ pos ]; }              // ( c )
 B[ i ] = sum ; }
```



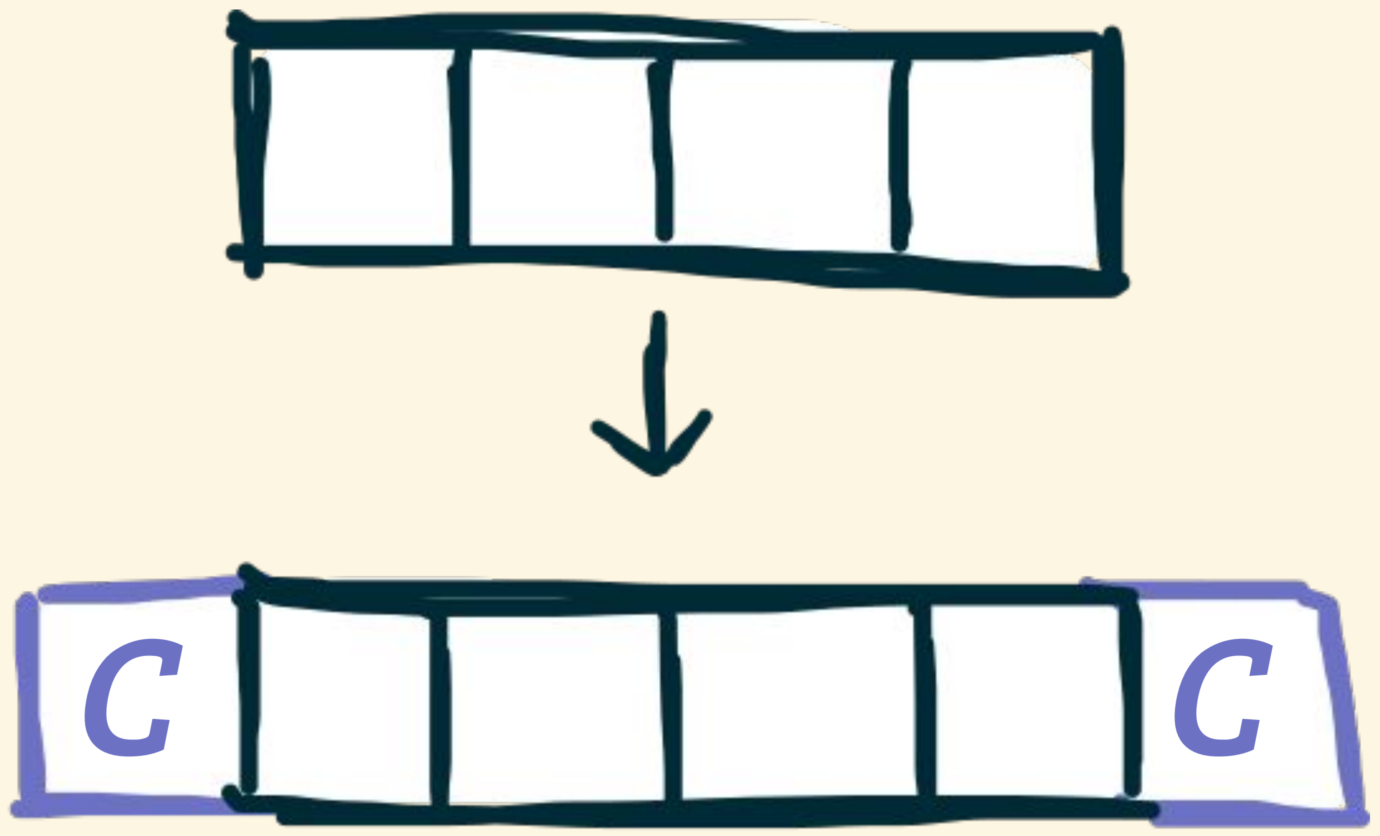
(a) access **neighborhoods** for every element
(b) specify **boundary handling**
(c) apply **stencil function** to neighborhoods

BOUNDARY HANDLING USING PAD

pad (reindexing)



pad (constant)



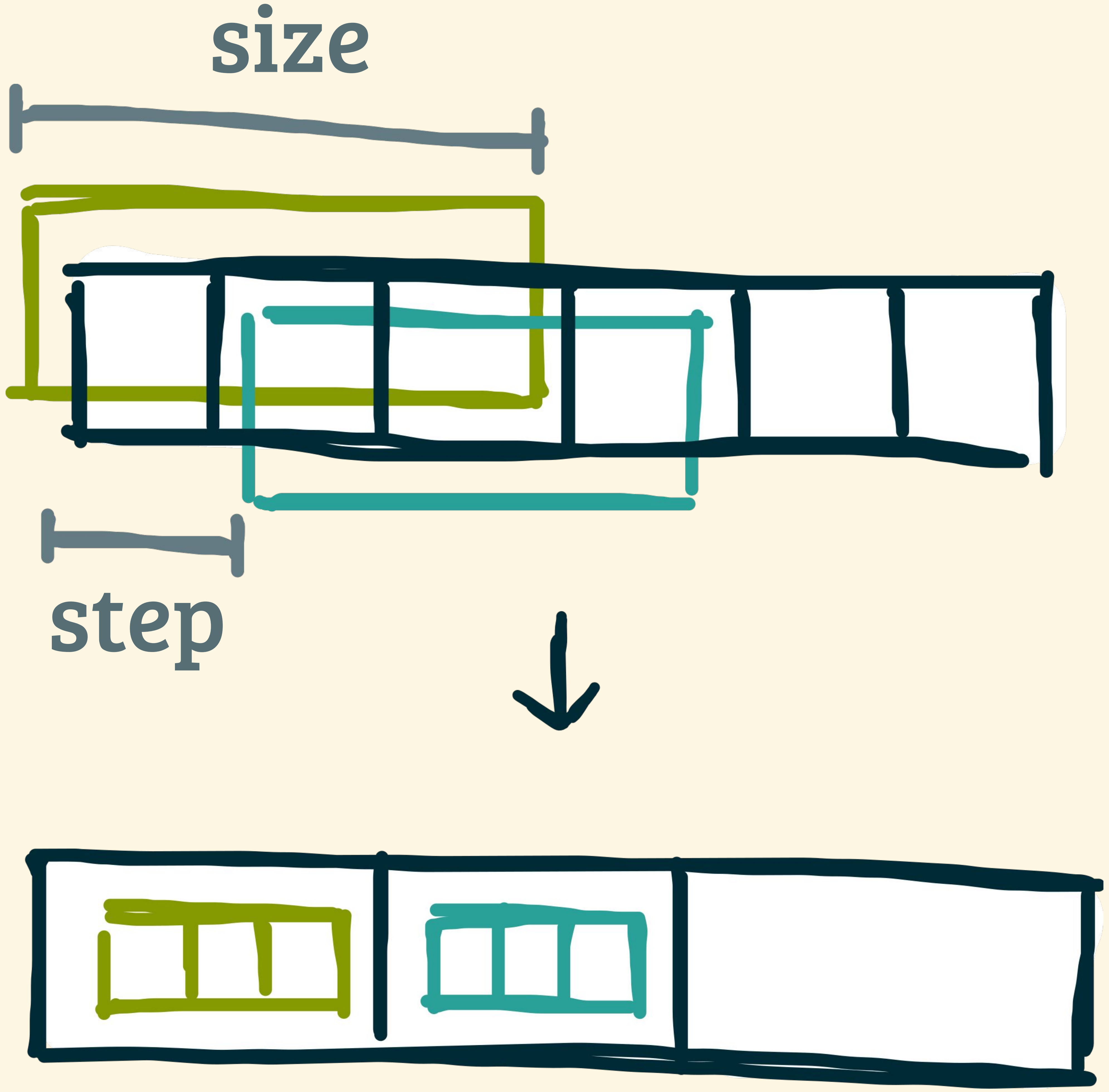
pad-reindexing.lift

```
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]
```

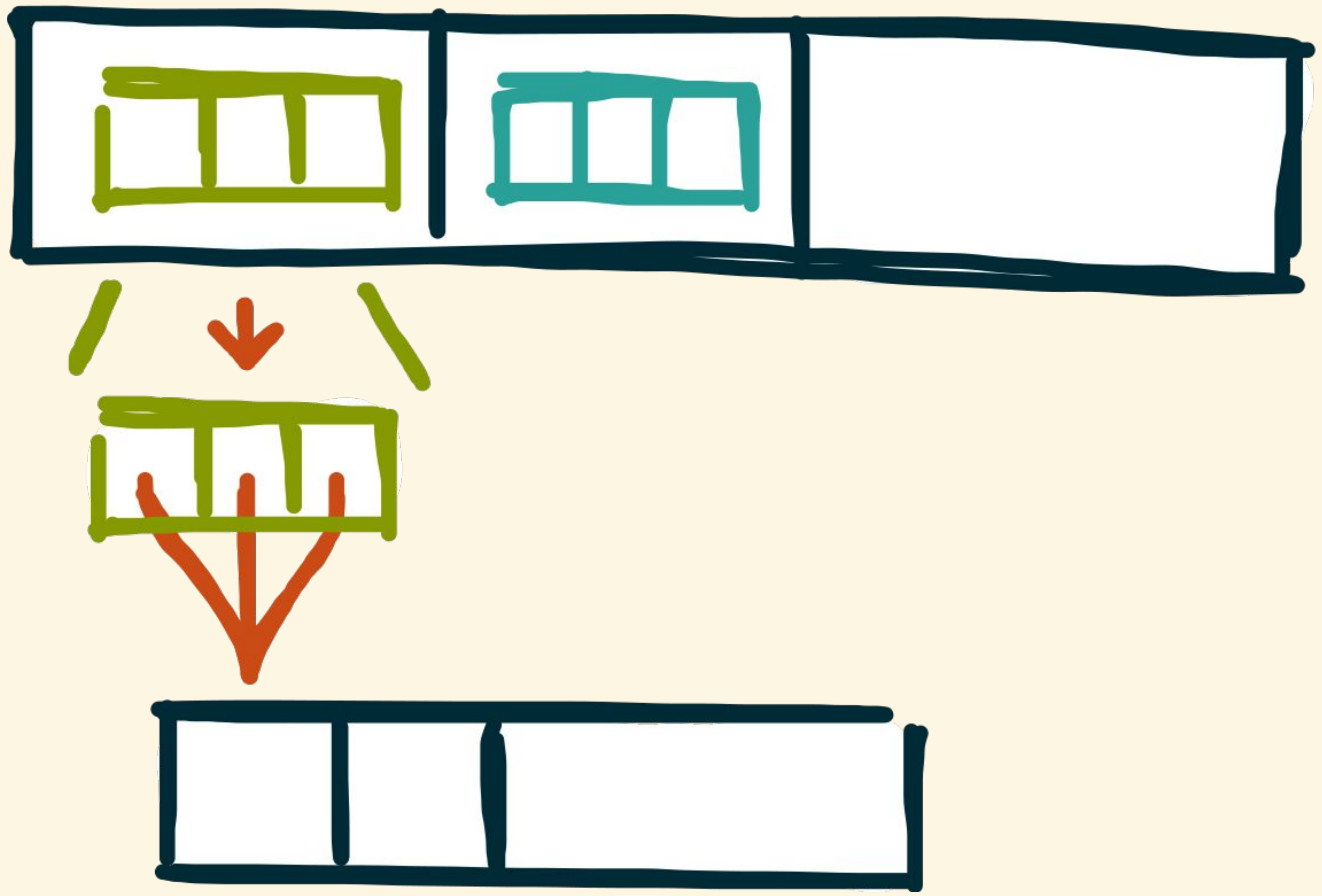
Neighborhood Creation Using slide



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

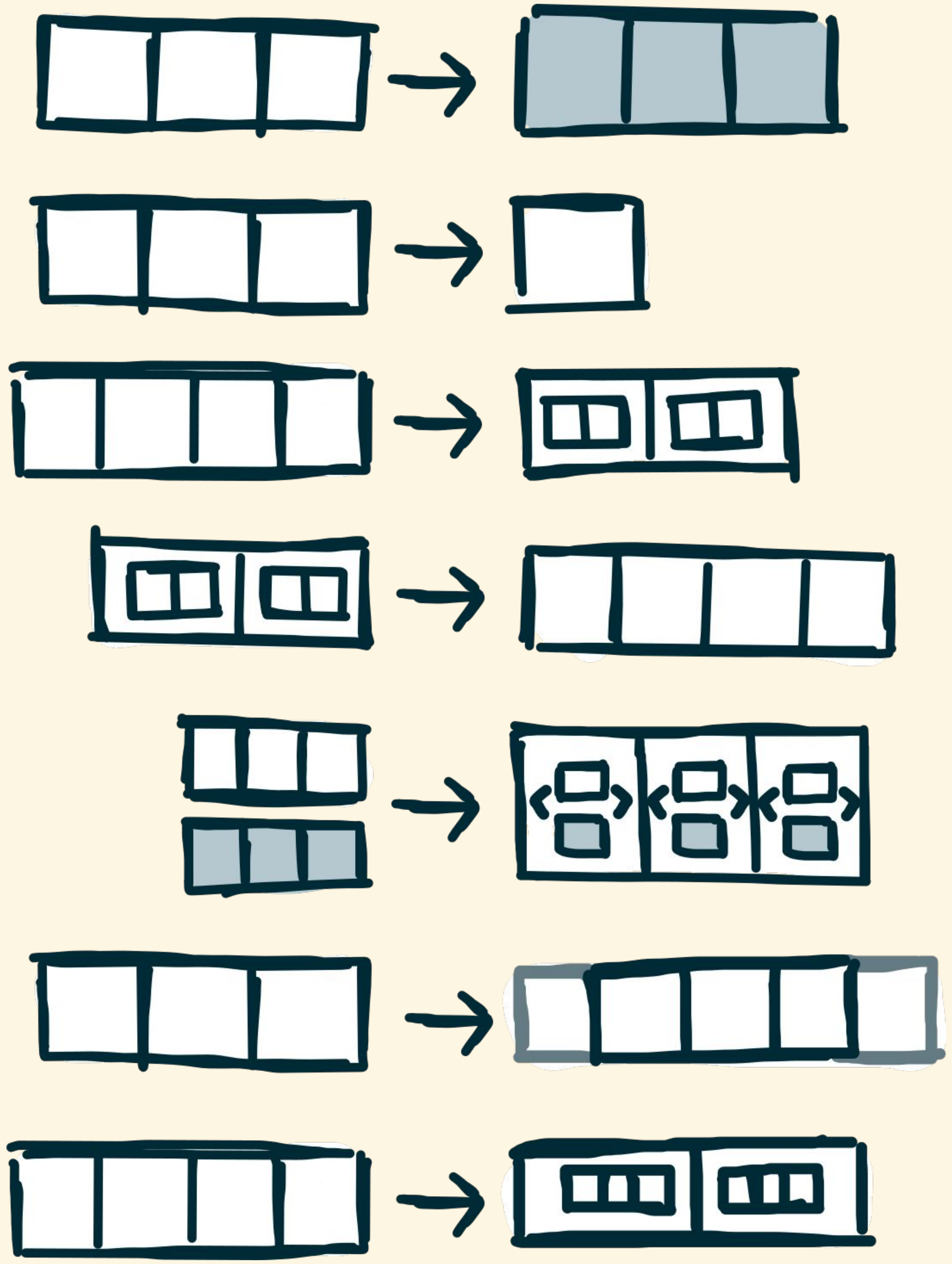


sum-neighborhoods.lift

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

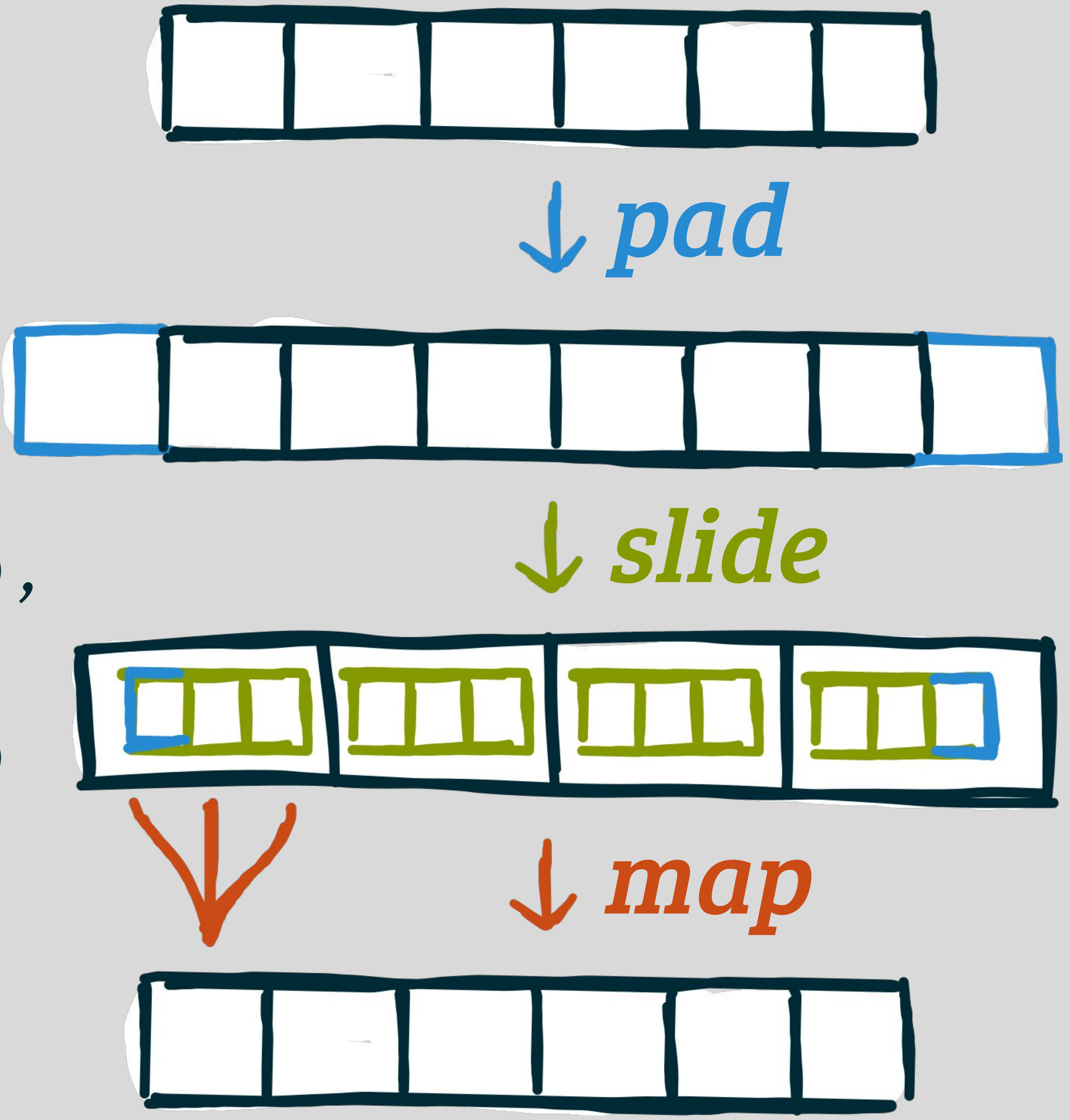
PUTTING IT TOGETHER

map(□→□)
reduce(⊕)
split(n)
join
zip
pad(l,r,b)
slide(n,s)



stencil1D.lift

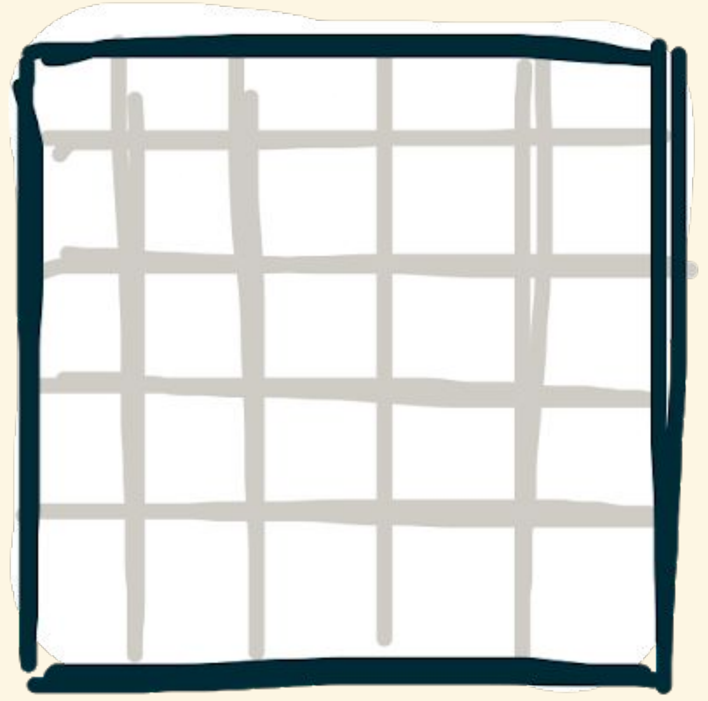
```
def stencil1D =
 fun(A =>
  map(reduce(add, 0.0f),
   slide(3,1,
    pad(1,1,clamp,A))))
```



Multidimensional Stencil Computations

are expressed as compositions of intuitive, generic 1D primitives

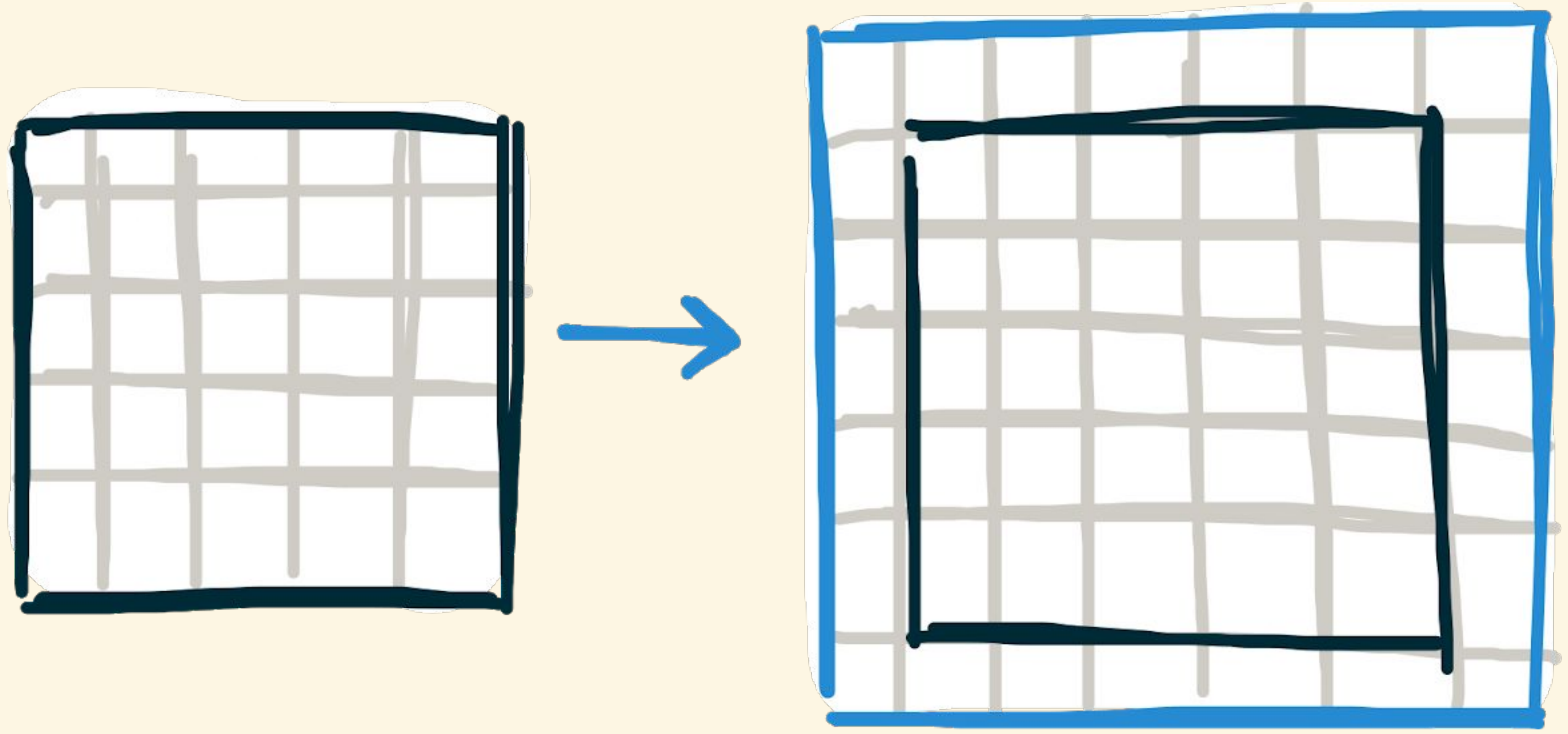
Decompose to Re-Compose



Multidimensional Stencil Computations

are expressed as compositions of intuitive, generic 1D primitives

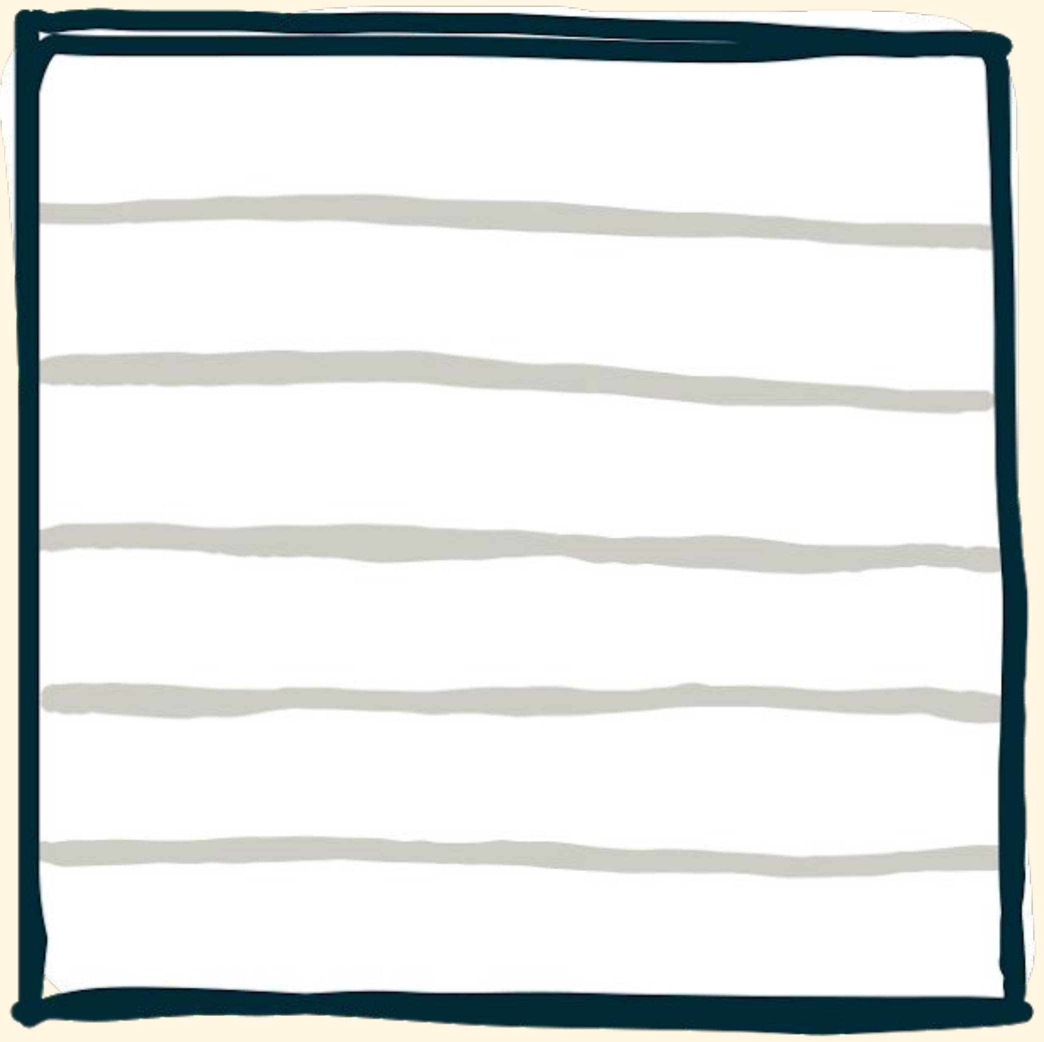
Decompose to Re-Compose



$\textbf{pad}_2(1, 1, \mathit{clamp}, \mathrm{input})$

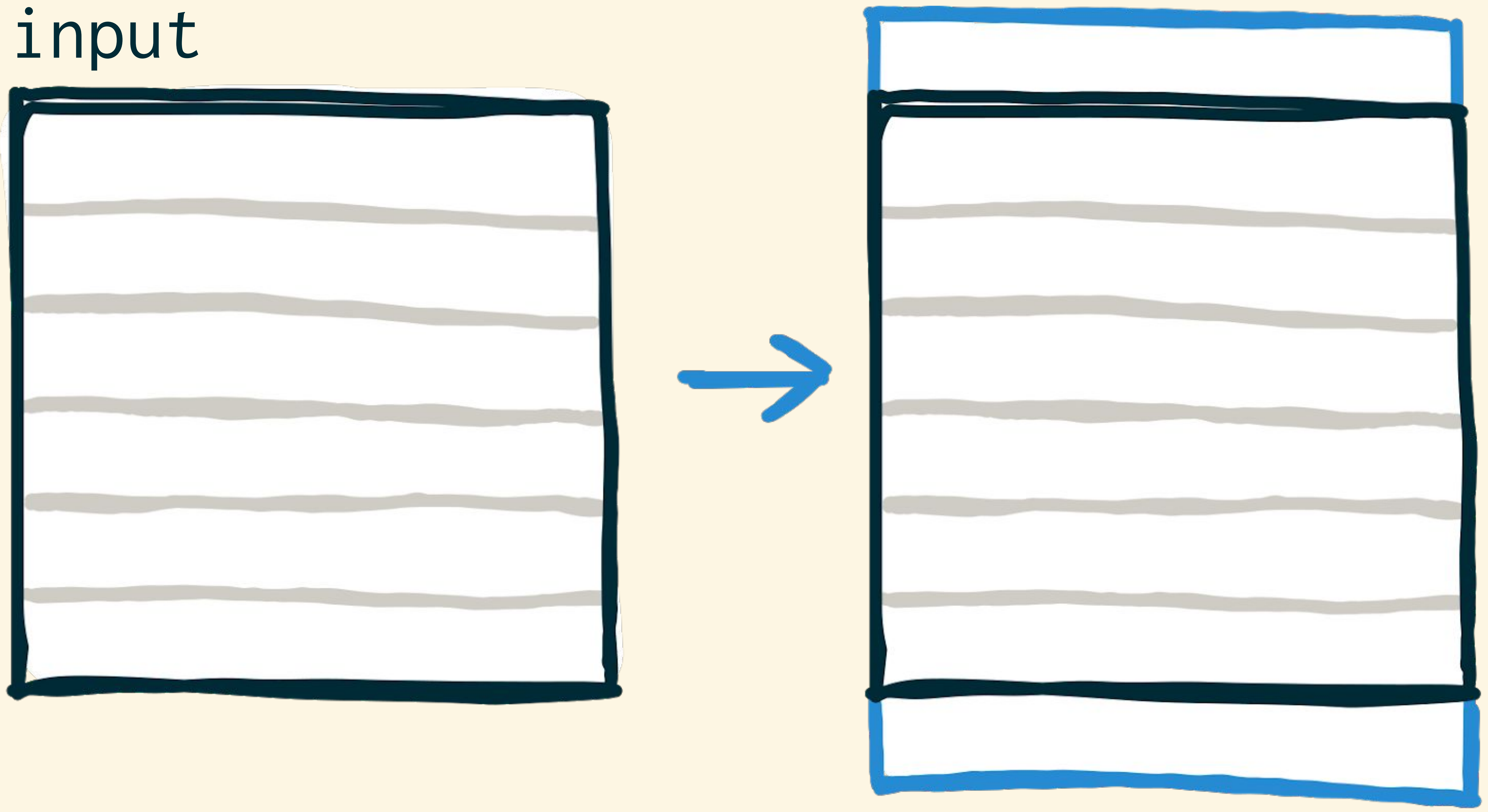
Multidimensional boundary handling using pad_2

input



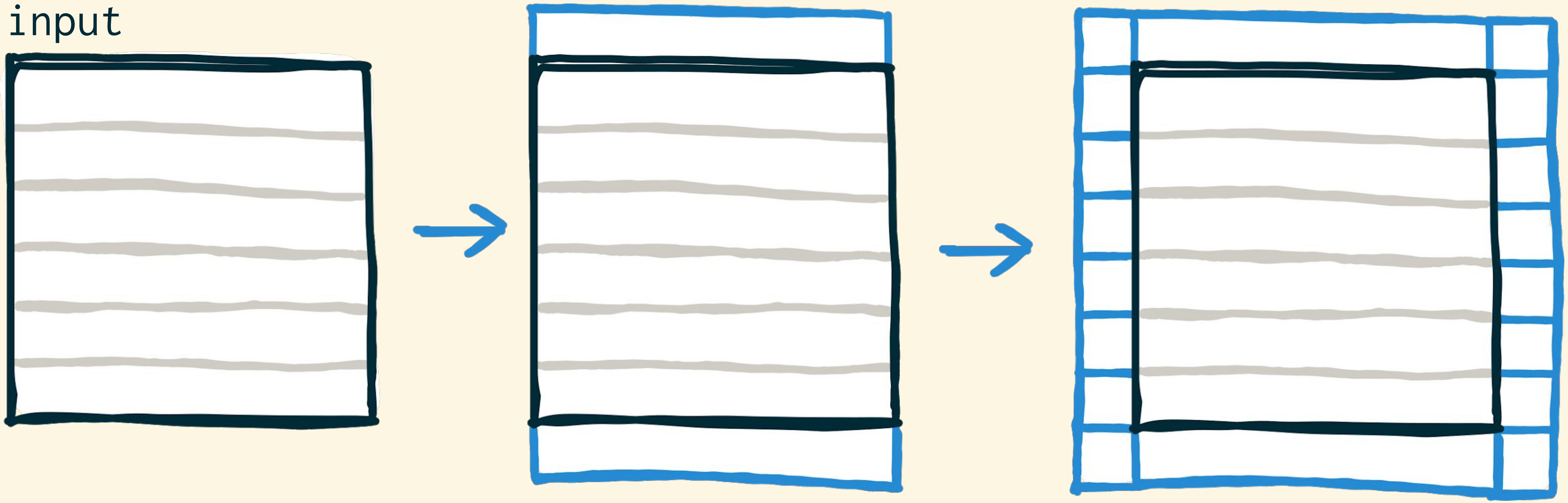
$pad_2 =$

MULTIDIMENSIONAL BOUNDARY HANDLING USING PAD$_2$



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

Multidimensional Boundary Handling using pad_2

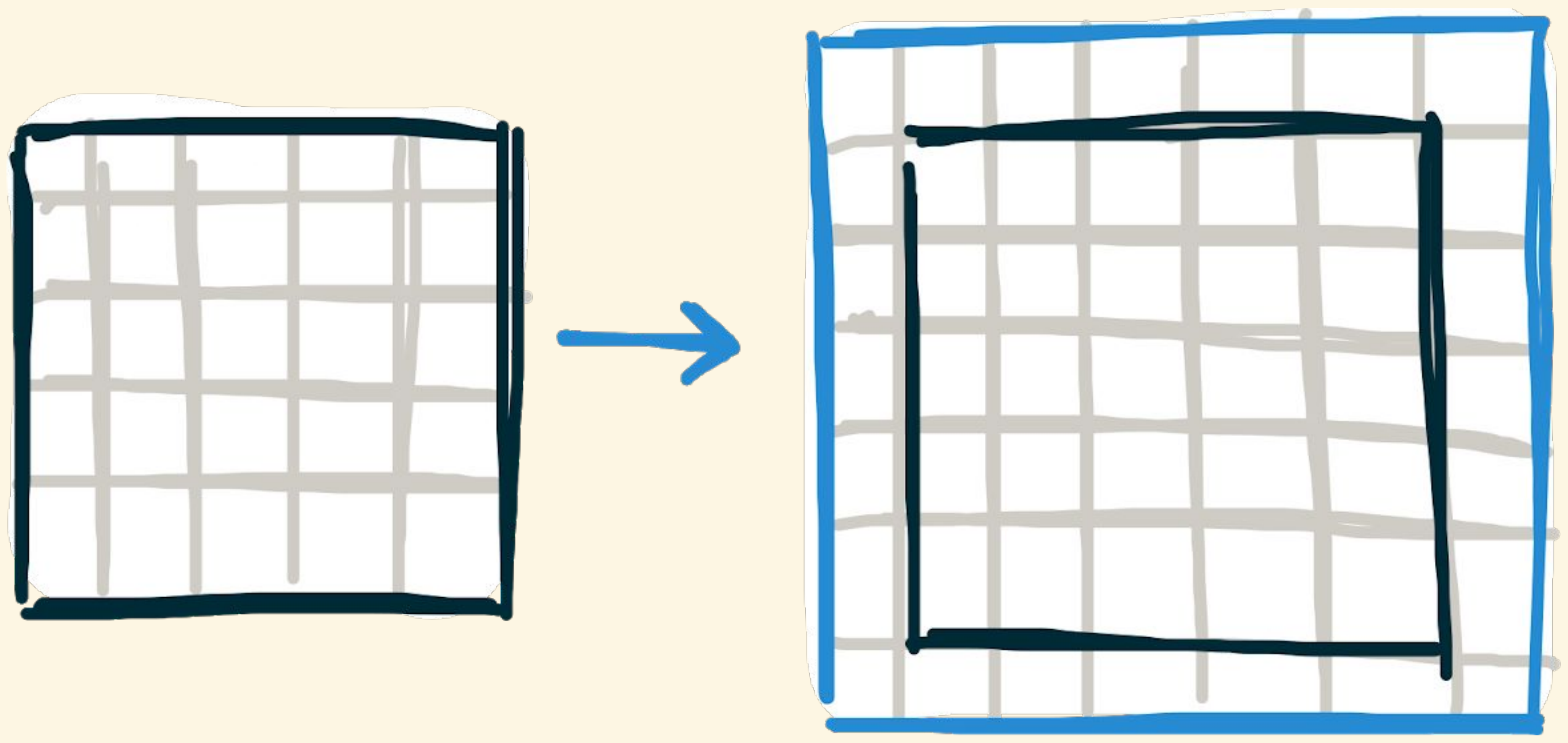


$$\boldsymbol{pad_2} = \mathtt{map}(\mathtt{pad}(l,r,b,\mathtt{pad}(l,r,b,\mathtt{input})))$$

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

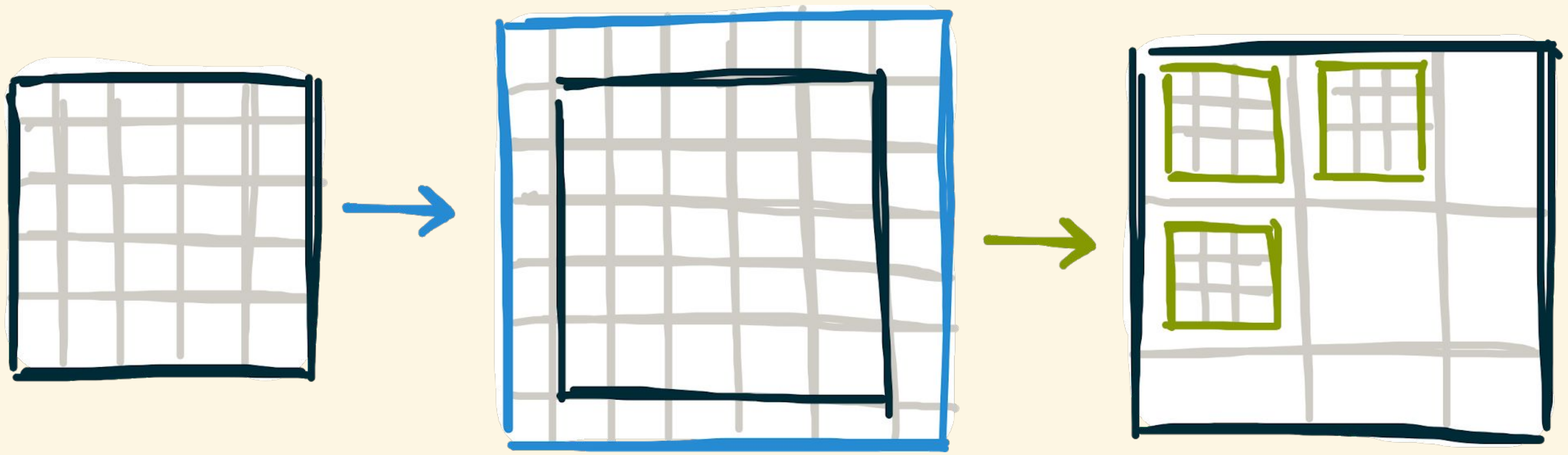


pad_2(1,1,*clamp*,input)

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

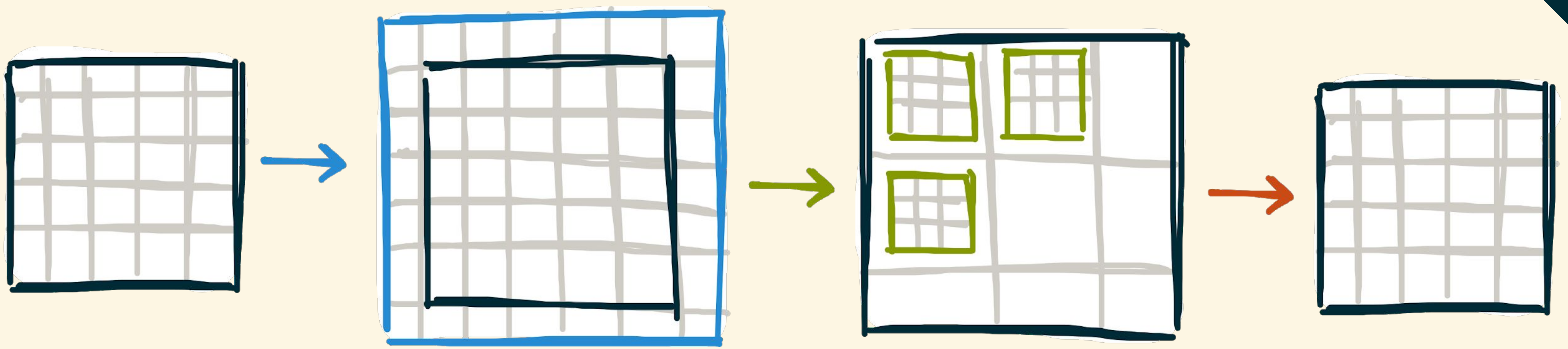


$slide_2(3,1,\ pad_2(1,1,clamp,\mathrm{input}))$

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

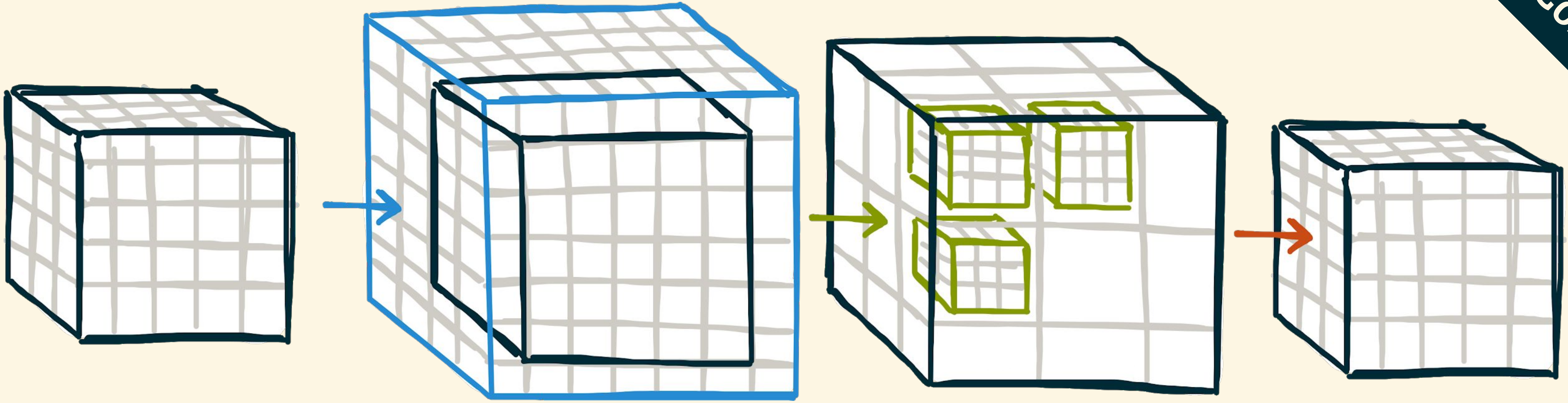


$\mathbf{map}_2(\mathit{sum},\ \mathbf{slide}_2(3,1,\ \mathbf{pad}_2(1,1,\mathit{clamp},\mathrm{input})))$

Multidimensional Stencil Computations

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

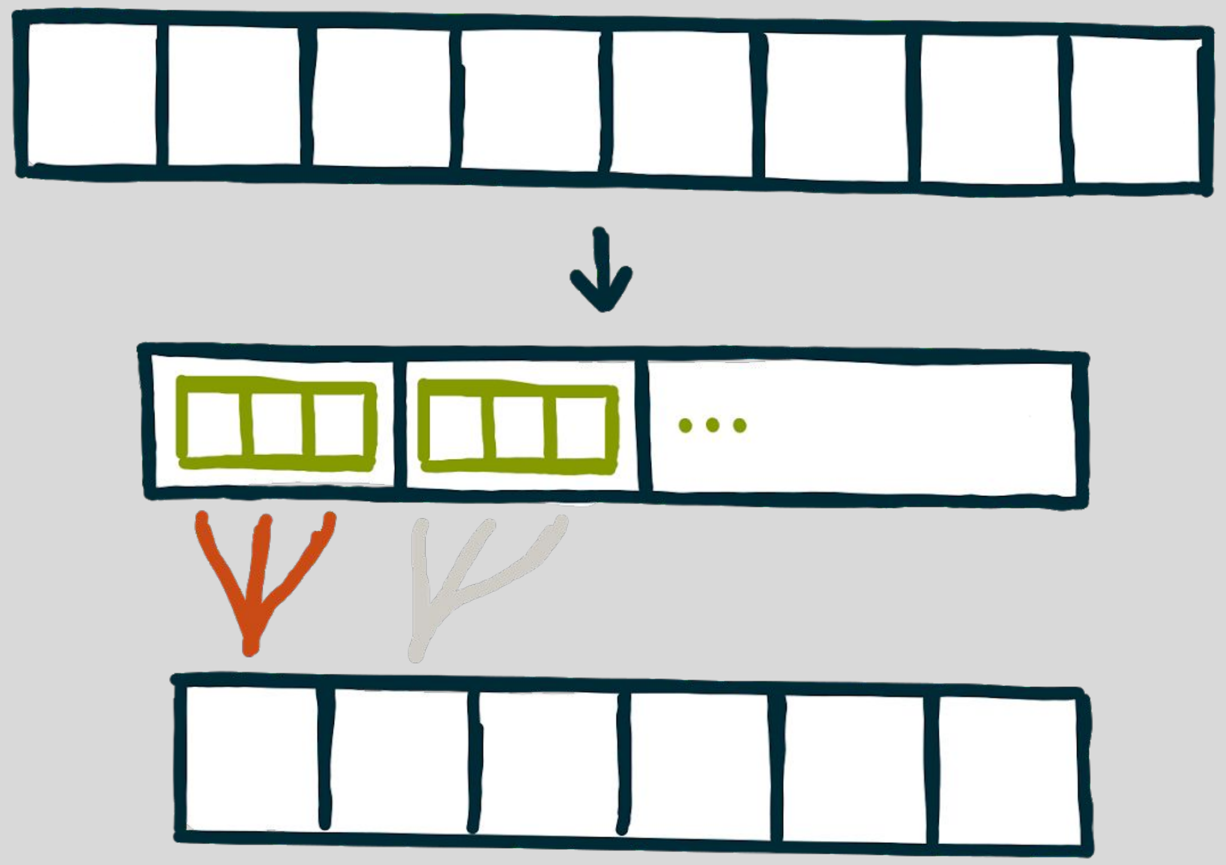


$map_3(sum,\ slide_3(3,1,\ pad_3(1,1,clamp,input)))$

Overlapped Tiling as a Rewrite Rule

overlapped tiling rule

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

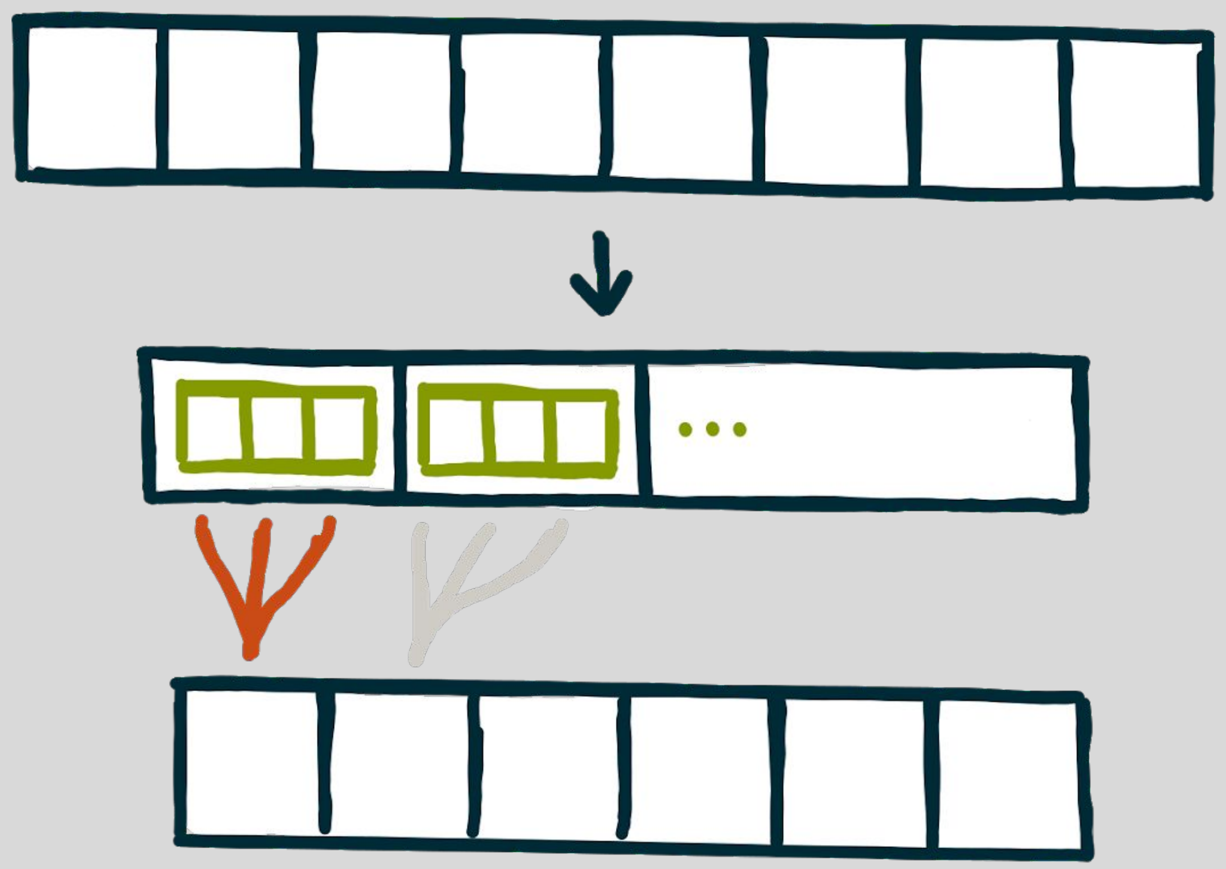


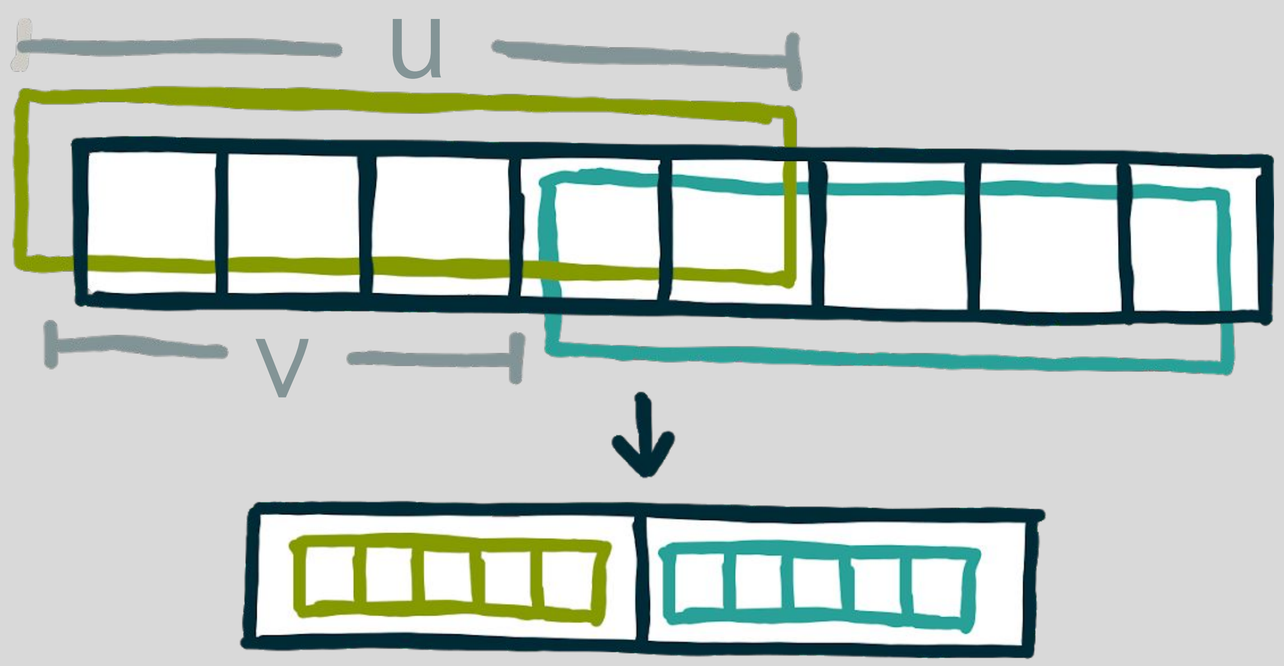
OVERLAPPED TILING AS A REWRITE RULE

overlapped tiling rule

`map(f, slide(3,1,input))` ↦

`slide(u,v,input)`

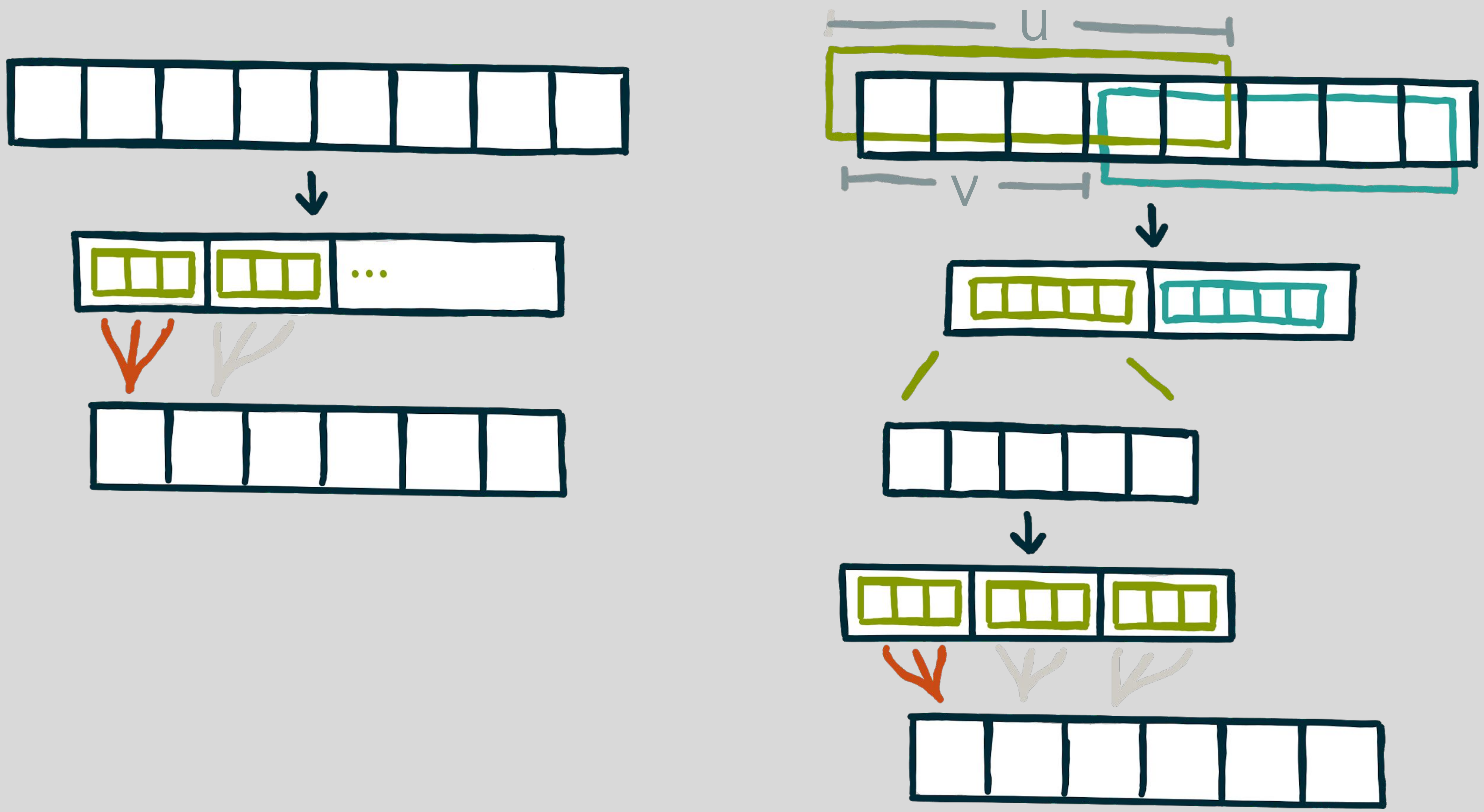




Overlapped Tiling as a Rewrite Rule

overlapped tiling rule

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



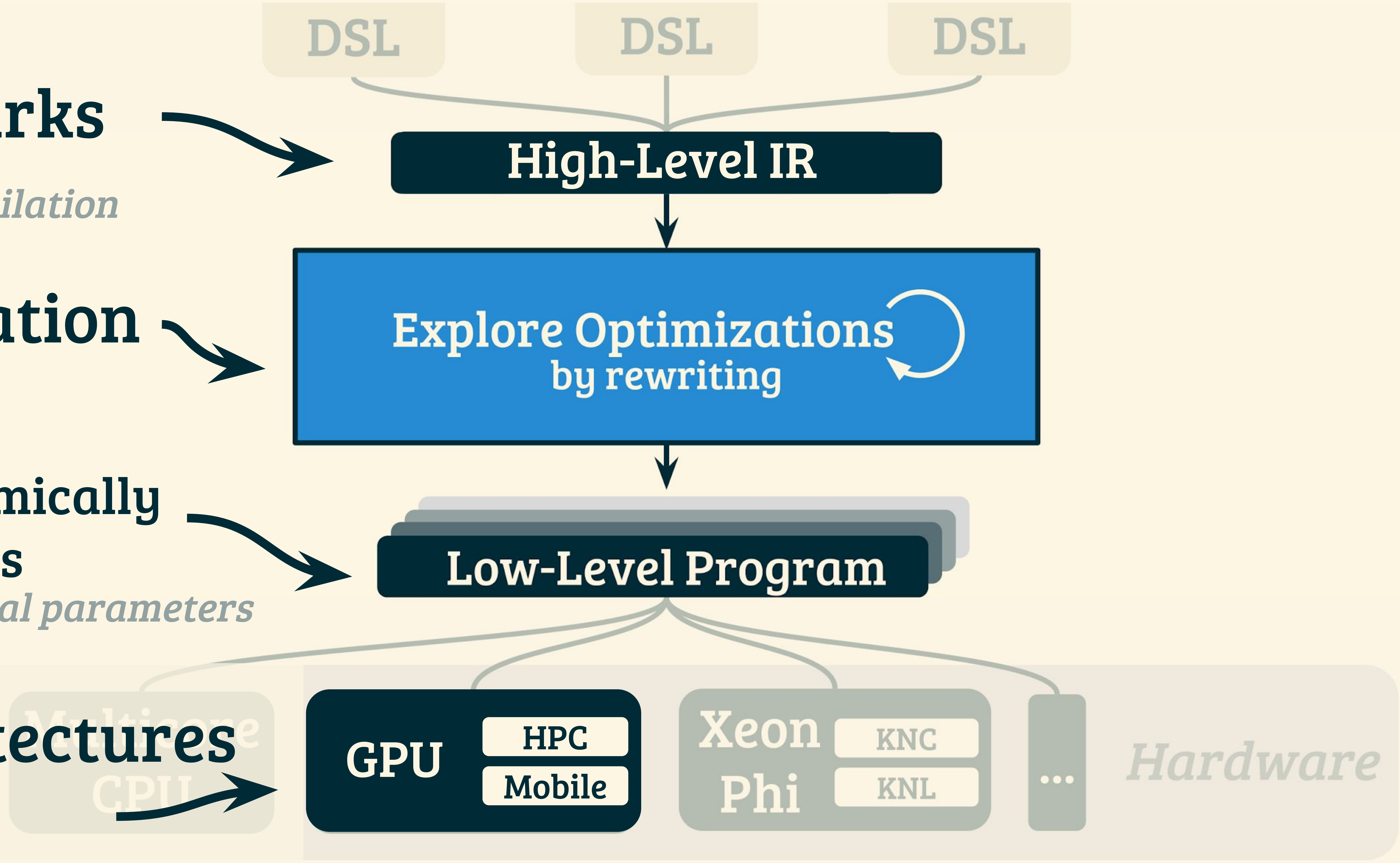
EXPERIMENTAL EVALUATION

14 Benchmarks
6 hand-optimized
8 polyhedral compilation

< 3h Exploration
per benchmark

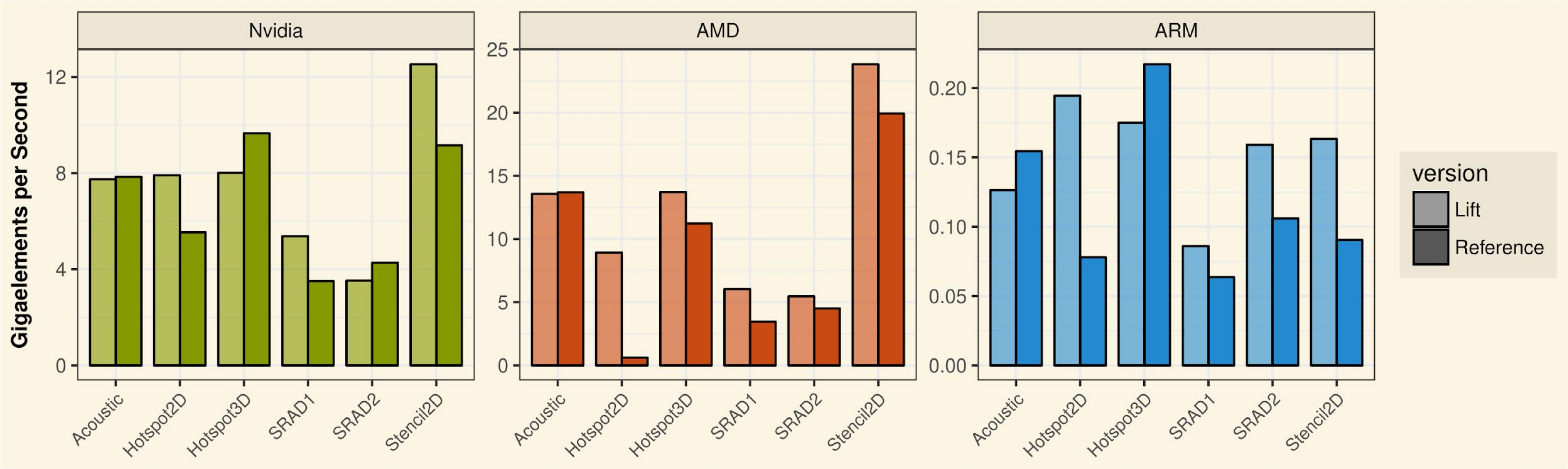
up to 20 algorithmically different variants
+ auto-tuning of numerical parameters

3 GPU Architectures
2 Desktop GPUs
1 Mobile GPU



COMPARISON WITH HAND-OPTIMIZED CODES

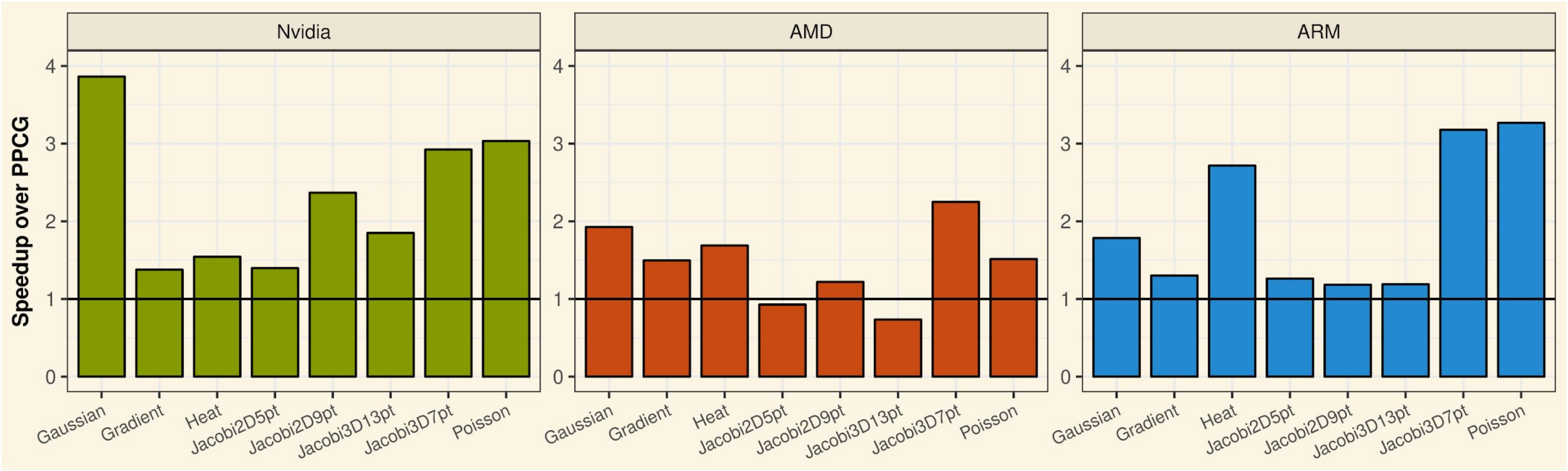
higher is better



Lift achieves the same performance
as hand optimized code

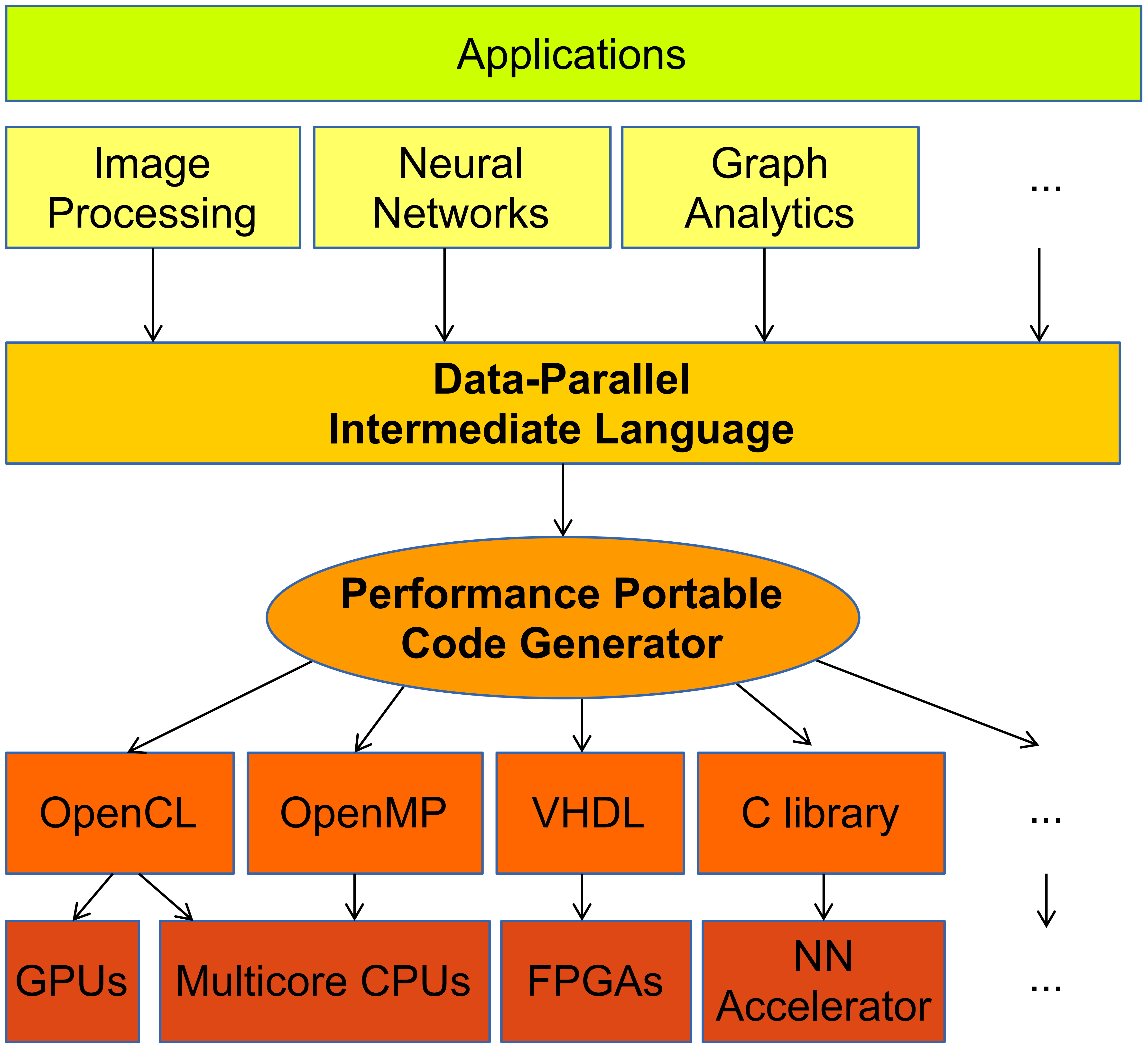
COMPARISON WITH POLYHEDRAL COMPILATION

higher is better



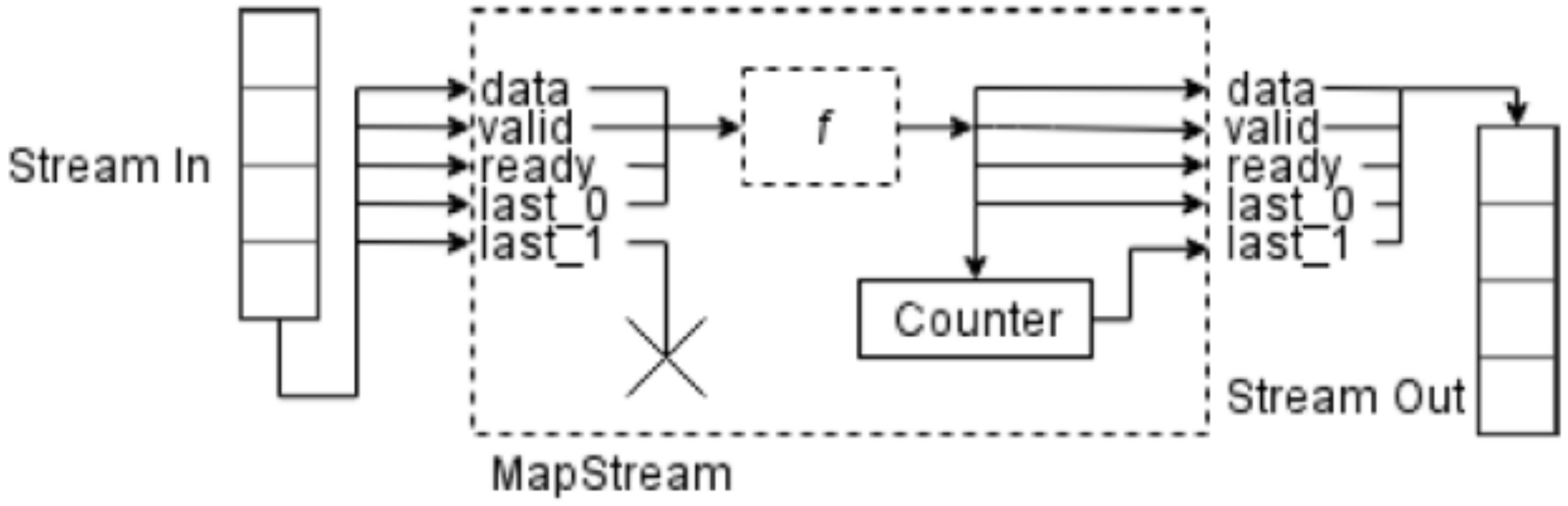
Lift outperforms state-of-the-art optimizing compilers

Lift works beyond GPUs

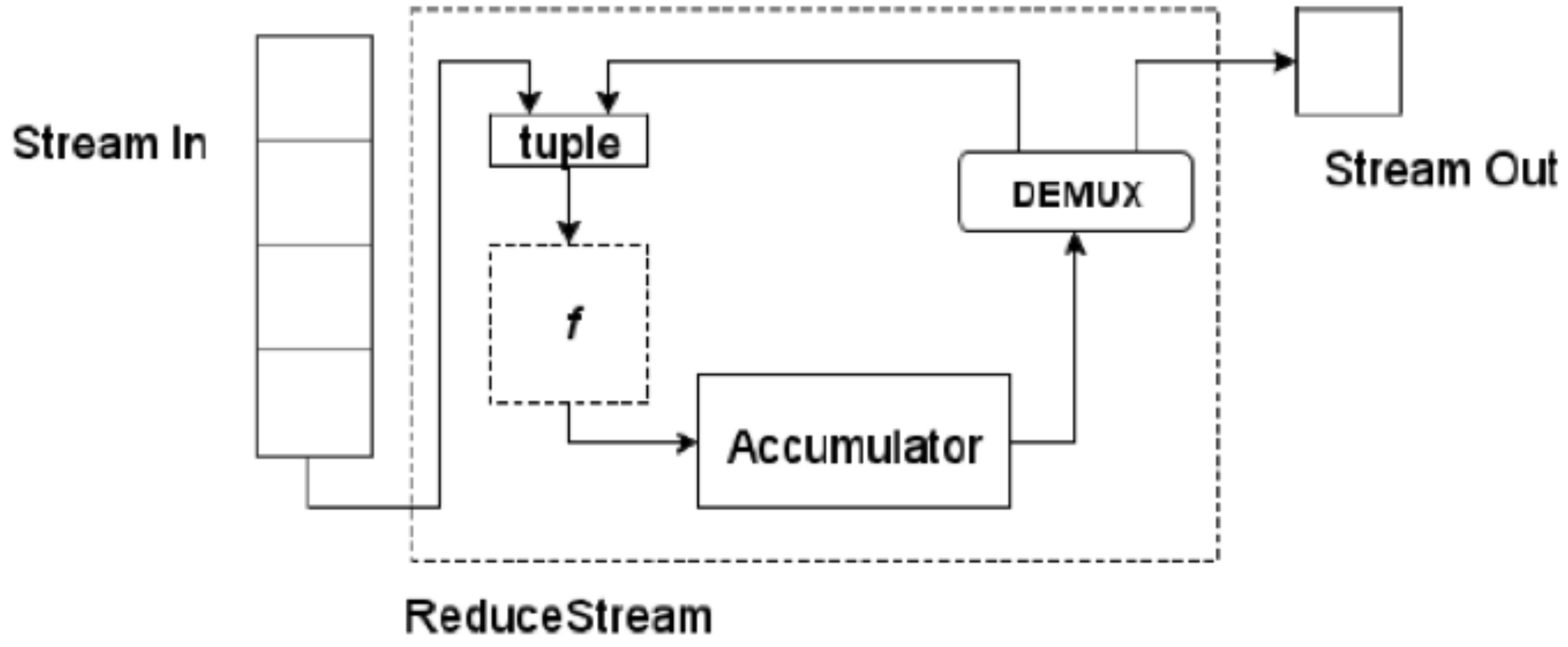


Moving onto FPGAs

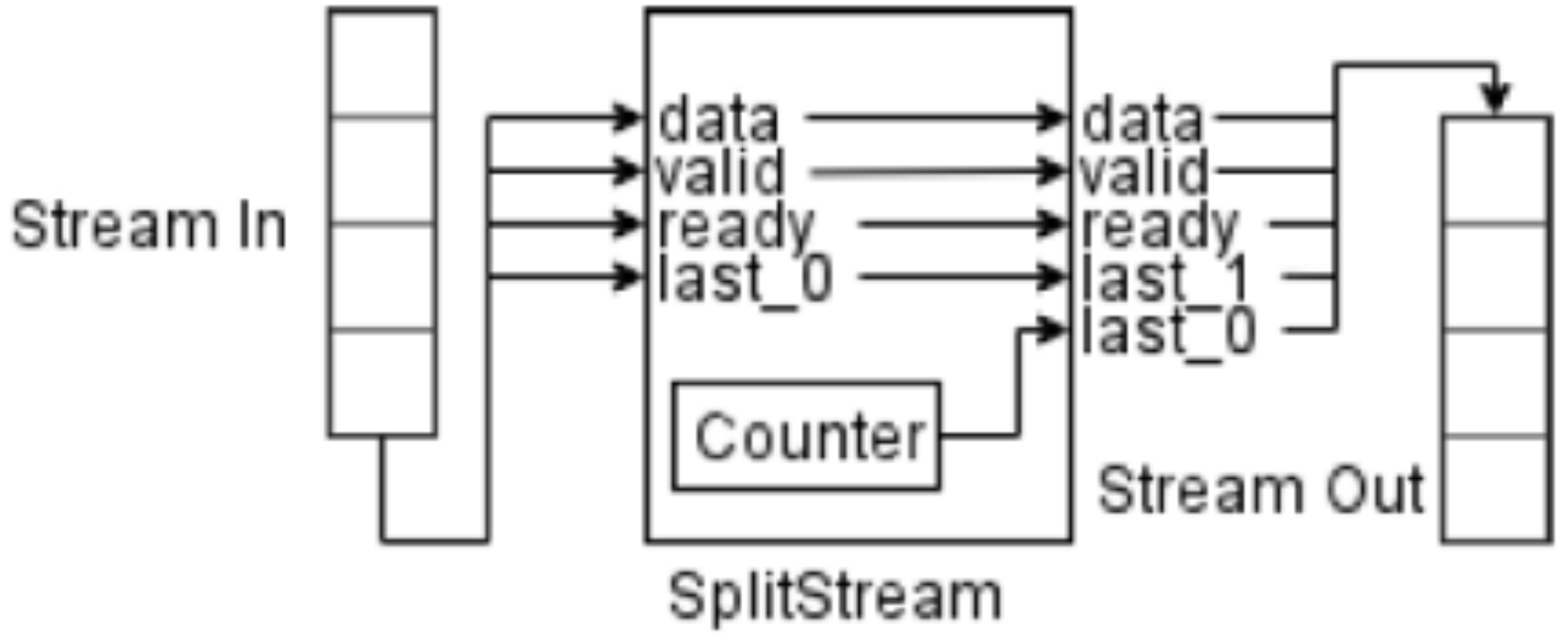
mapStream



reduceStream



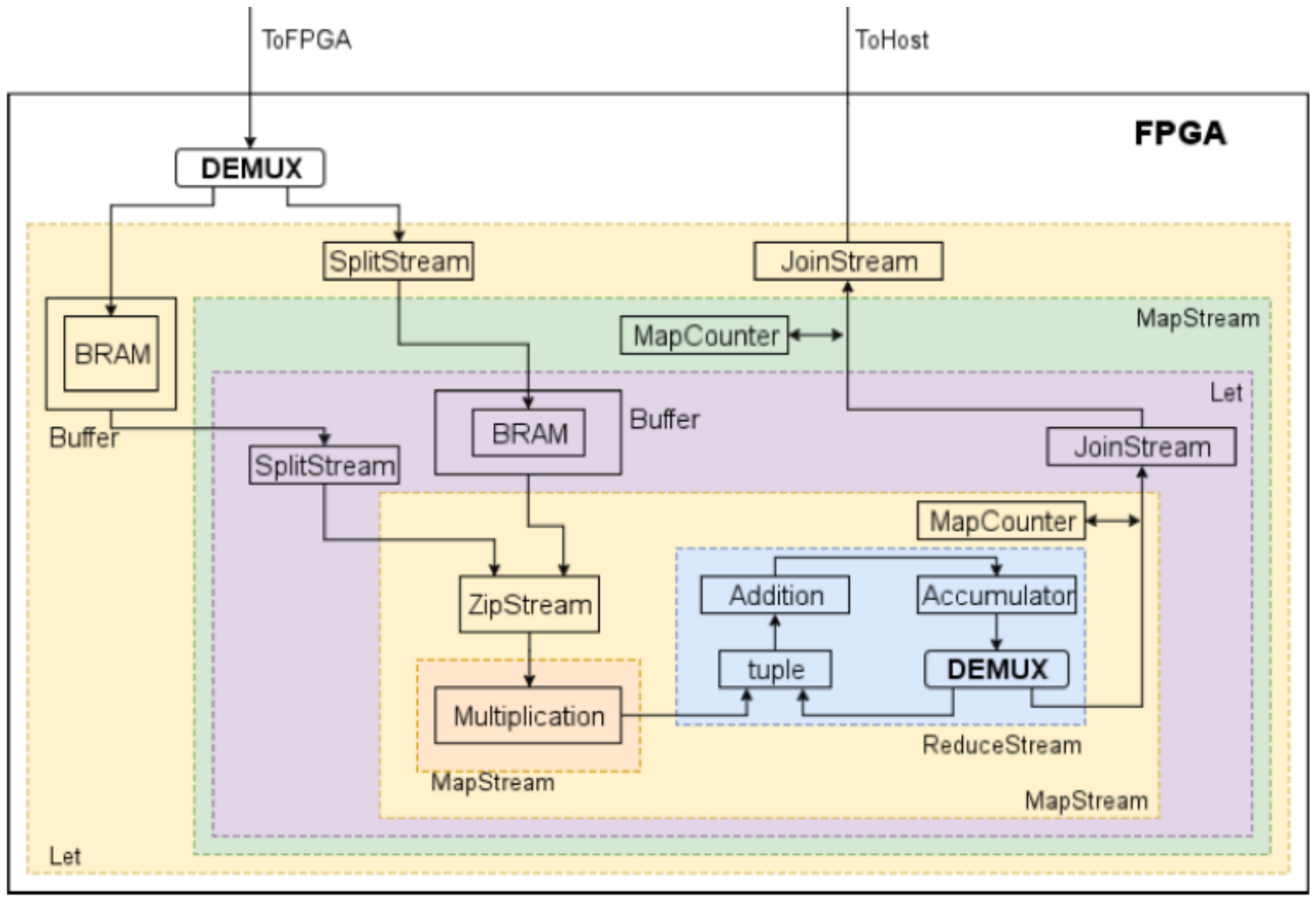
splitStream



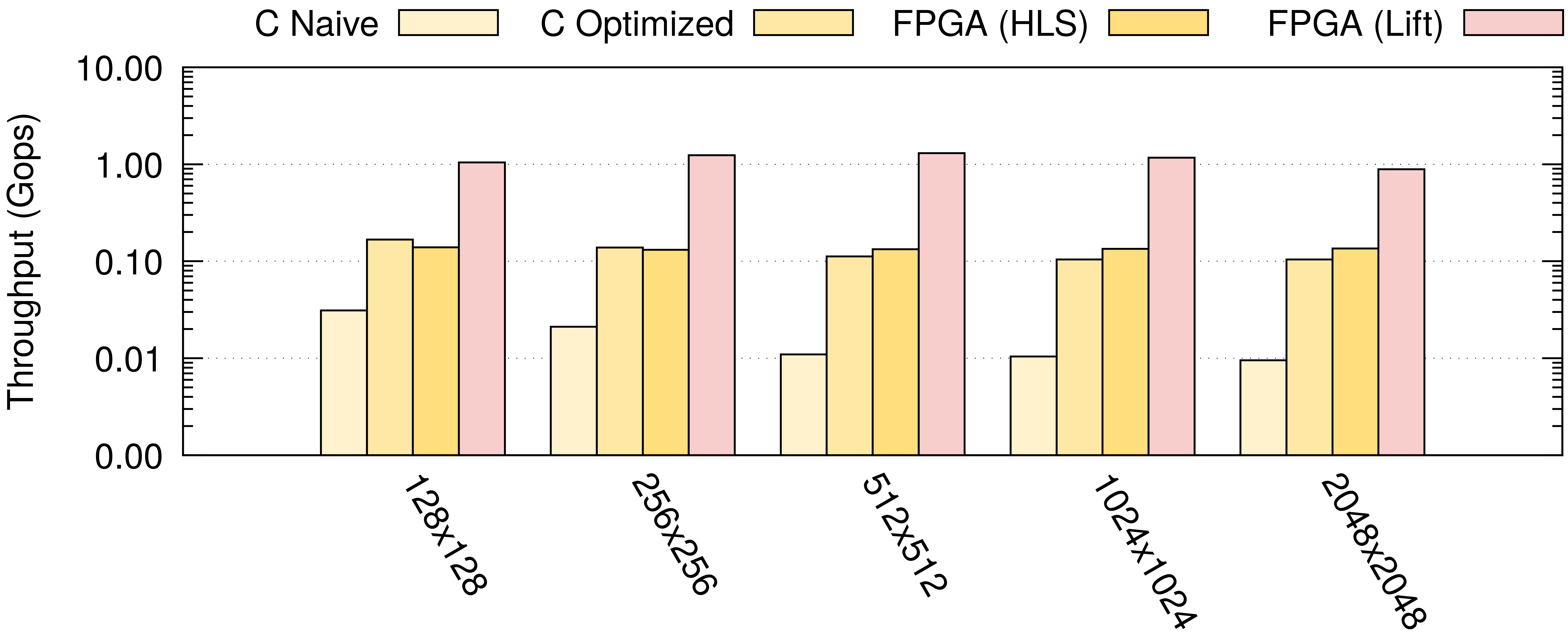
Matrix-multiplication on FPGA

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

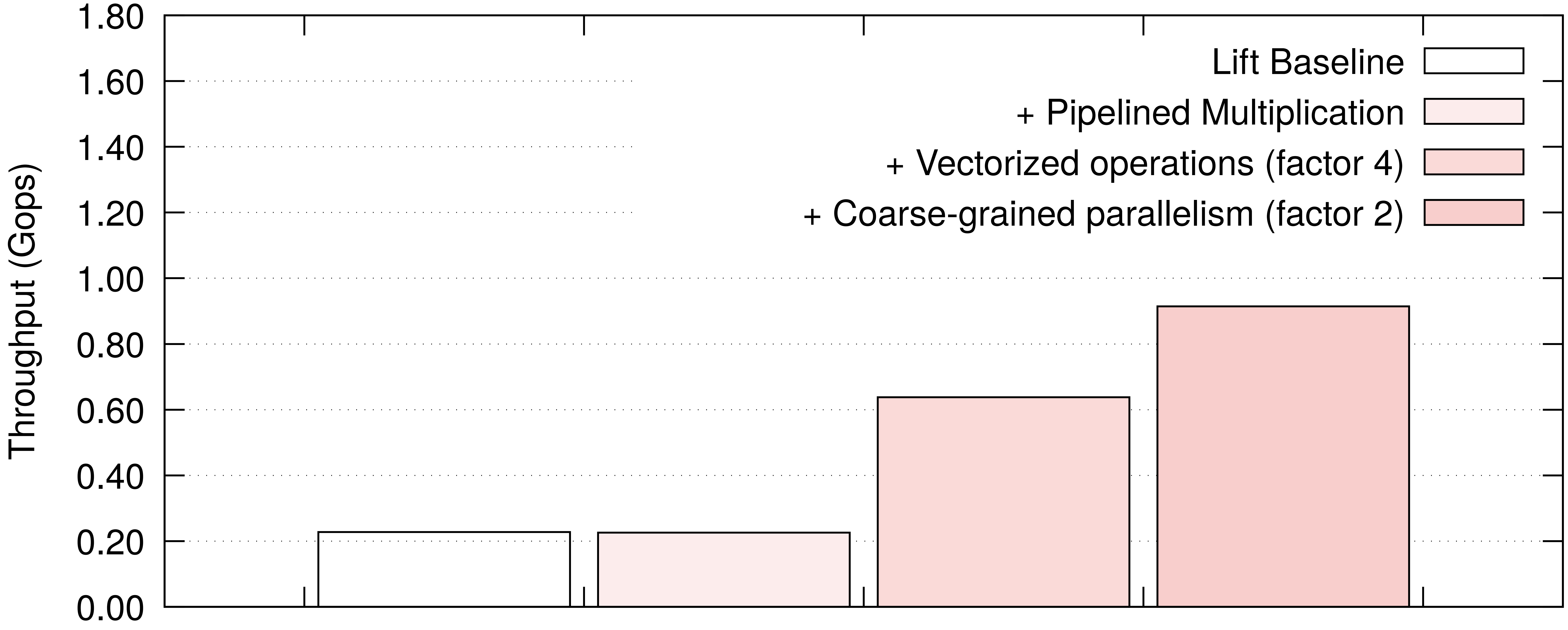
```
split(B_num_col,
  toHost(let(λ B ↦
    joinStream(
      mapStream(λ a_row ↦ let(λ a_row ↦
        joinStream(
          mapStream(λ b_col ↦ dotProduct(a_row, b_col)
          , splitStream(B_num_col, B)))
      , a_row)
    , splitStream(A_num_col, toFPGA(flatten(A))) ))
  , toFPGA(flatten(transpose(B))))))
```



Zynq 7000 results (preliminary)



Optimisation Space Exploration with Rewrites



Performance Models

- **Machine-learning based**
  - Extract features directly from high-level expression

Neural Networks with Lift

- **CNN (Convolution)**
  - Building blocks:
    - convolution, fully-connected, pooling
  - Architecture:
    - VGG, GoogleNet, ResNet
  - Optimisations example:
    - Stacked Systolic Array
    - Winograd transform
    - Weight pruning
    - Quantisation

- **RNN (Recurrent)**
  - Building block
    - LSTM (Long short-term memory)

LIFT IS OPEN SOURCE!

Artifacts Available
Artifacts Evaluated Reusable
Results Replicated

more info at:

lift-project.org

Paper Artifacts Source Code

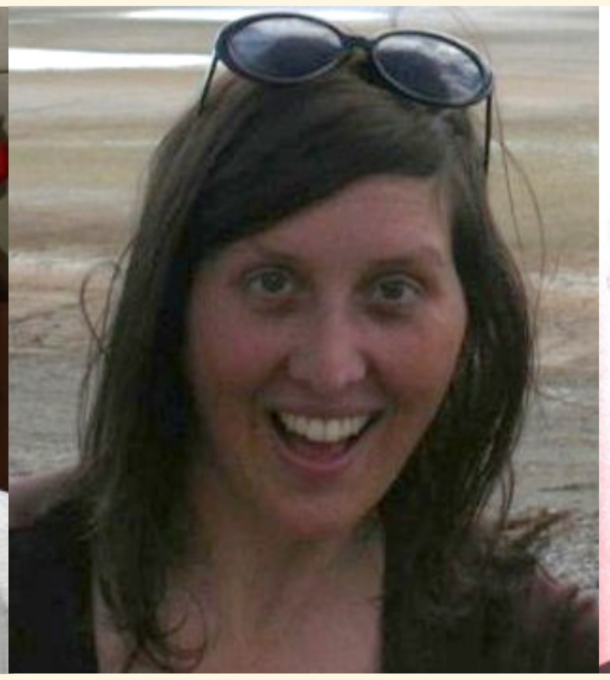

Naums Mogers


Lu Li


Christophe Dubach


Bastian Hagedorn


Toomas Remmelg


Larisa Stoltzfus


Michel Steuwer


Federico Pizzuti


Adam Harries