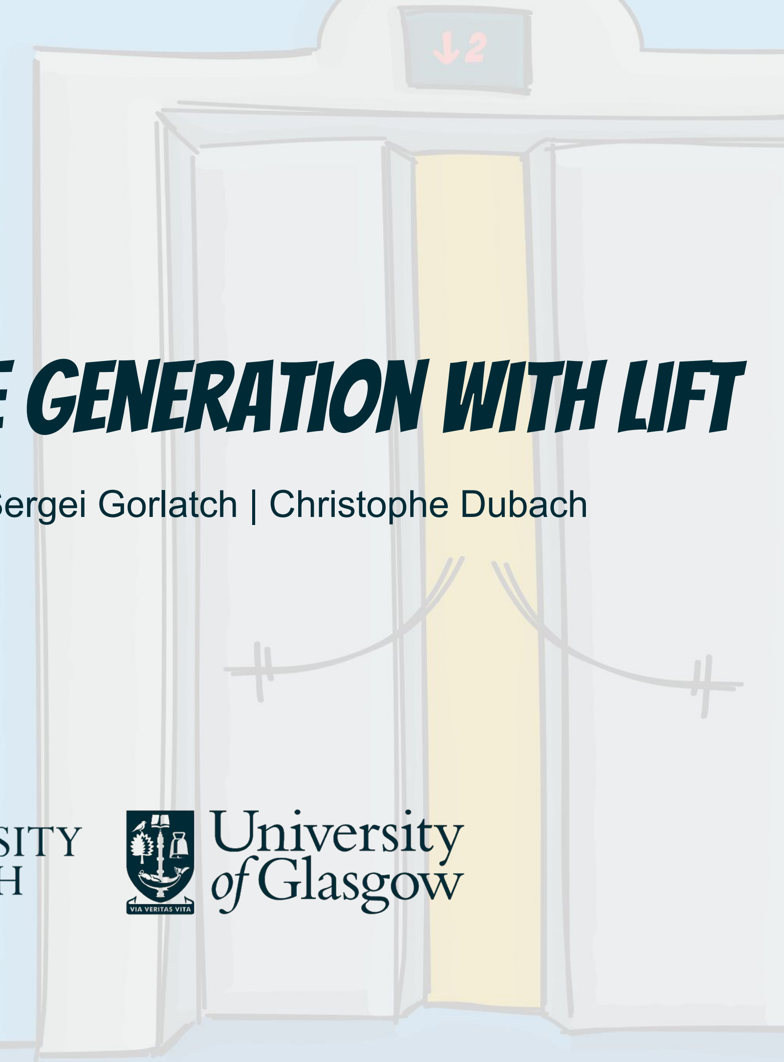


High Performance Stencil Code Generation with Lift

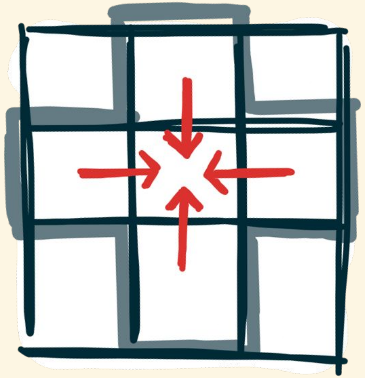
Bastian Hagedorn | Larisa Stoltzfus | Michel Steuwer | Sergei Gorlatch | Christophe Dubach

WWU MÜNSTER

THE UNIVERSITY of EDINBURGH

University of Glasgow

WHY STENCIL COMPUTATIONS?



Stencil computations are a class of kernels which update *neighboring* array elements according to a fixed pattern, called *stencil*.

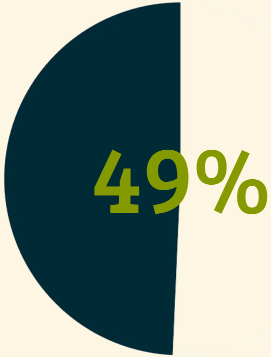
Frequently occur in:

- Medical Imaging
- Physics Simulations
- Machine Learning
- PDE Solvers

WHY STENCIL COMPUTATIONS?

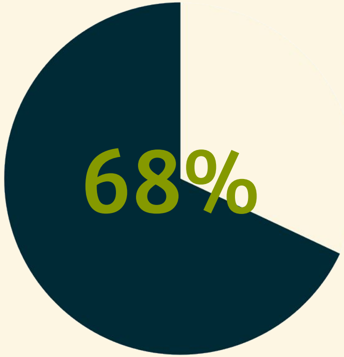
Stencil compute time:

HPC Center München



2017

HPC Center Stuttgart



2016

Frequently occur in:

- Medical Imaging
- Physics Simulations
- Machine Learning
- PDE Solvers

YET ANOTHER STENCIL PAPER?

Cache Oblivious Stencil Computations

Effective Automatic Parallelization of Ste

Performance Modeling and Automat Optimization for Iterative Stencil Lo

3.5-D Blocking Optimization for Stencil Computations on Modern CPUs and GPUs

Auto-Generation and Auto-Tuning of GPU Clusters

Understanding the Performance of Stencil Computations on Intel's Xeon Phi

Locality Aware Concurrent Start for Stencil Applications

YASK–Yet Another Stencil Kernel: a f HPC stencil code-generation and

A Stencil Framework to Realize Large-scale Computations Beyond Devi Capacity on GPU Supercomputers

High Performance Stencil Code Generation with Lift

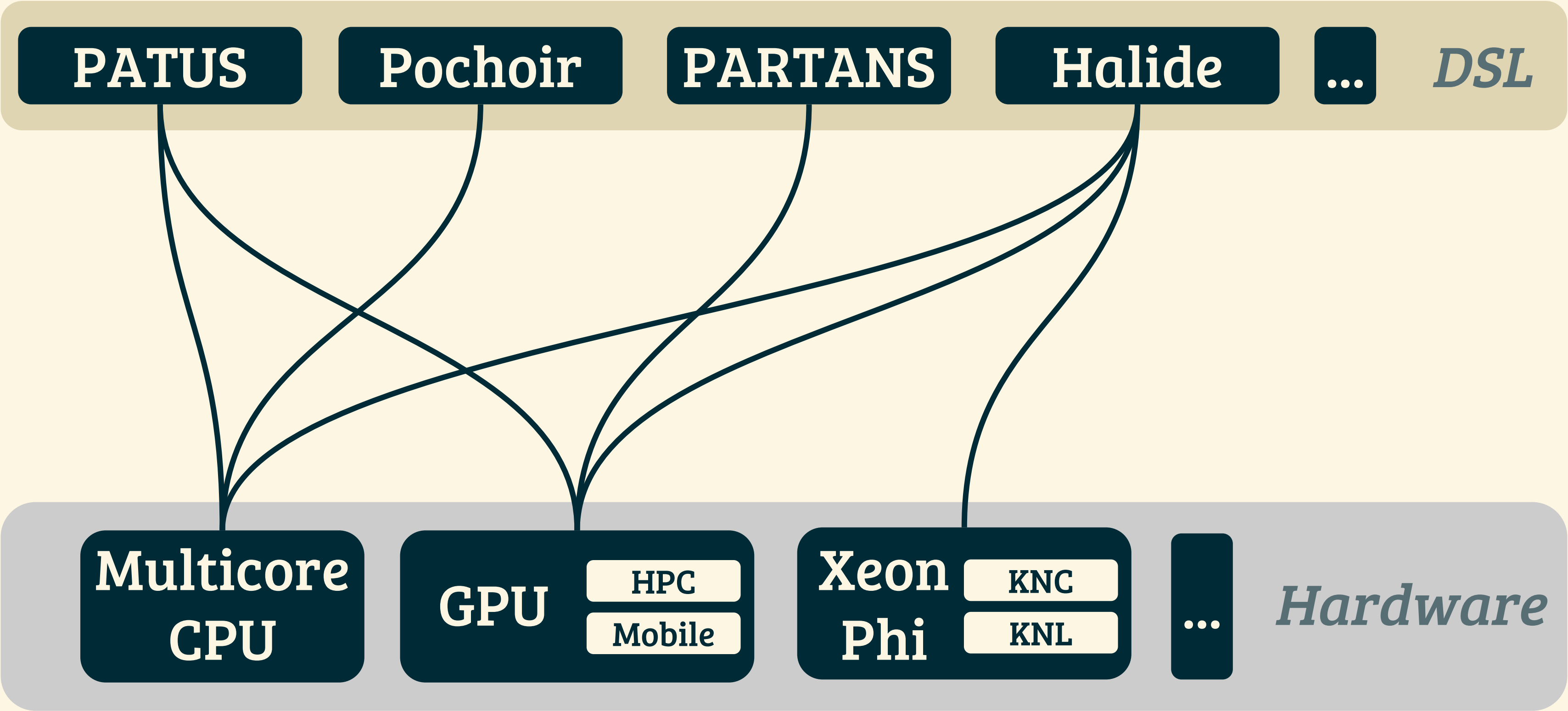
2005 2007 2009 2011 2013 2015 2018

ICS'05 PLDI'07 ICS'09 SC'10 CGO'12 CLUSTER'13 CGO'15 WOLFHPC'16 CLUSTER'17 CGO'18

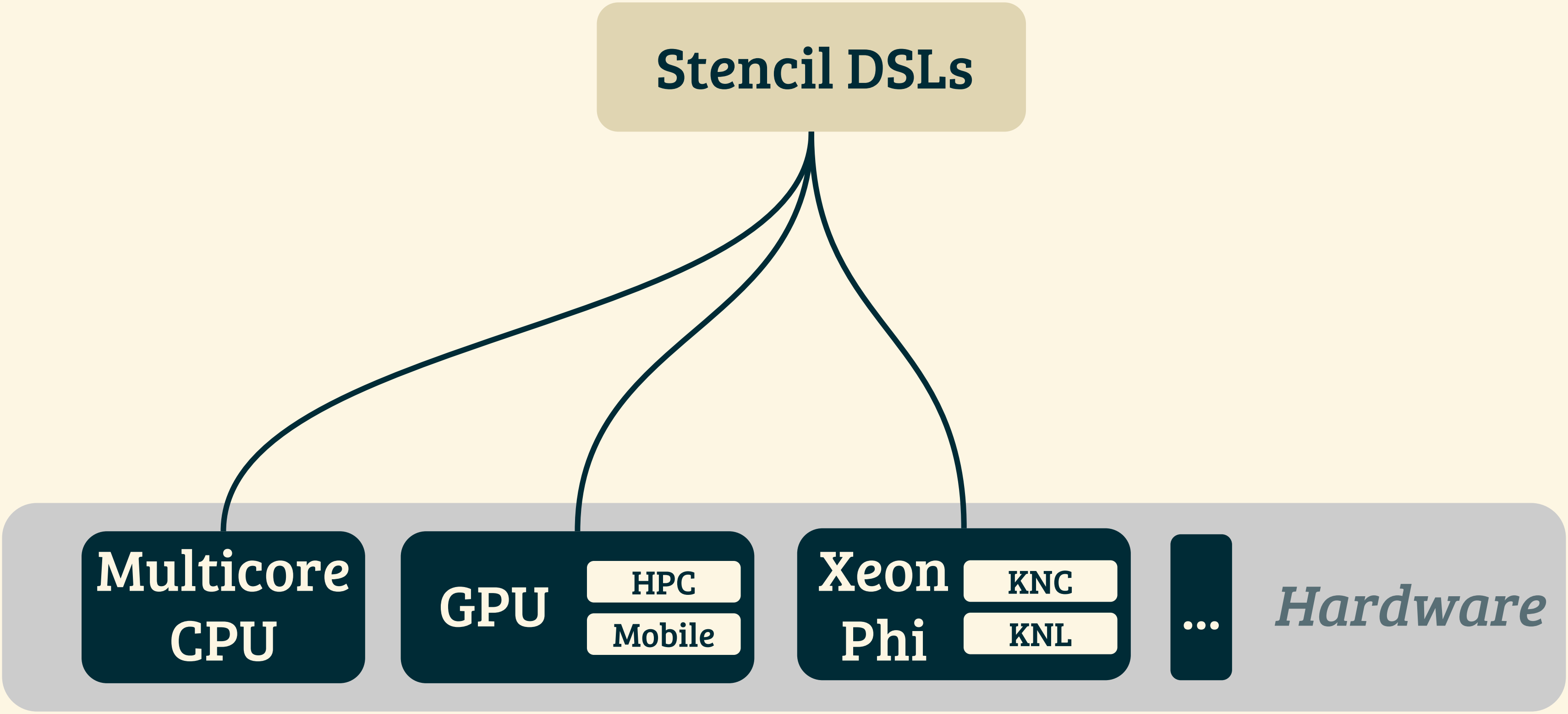
DOMAIN SPECIFIC LANGUAGES

PATUS Pochoir PARTANS Halide … *DSL*

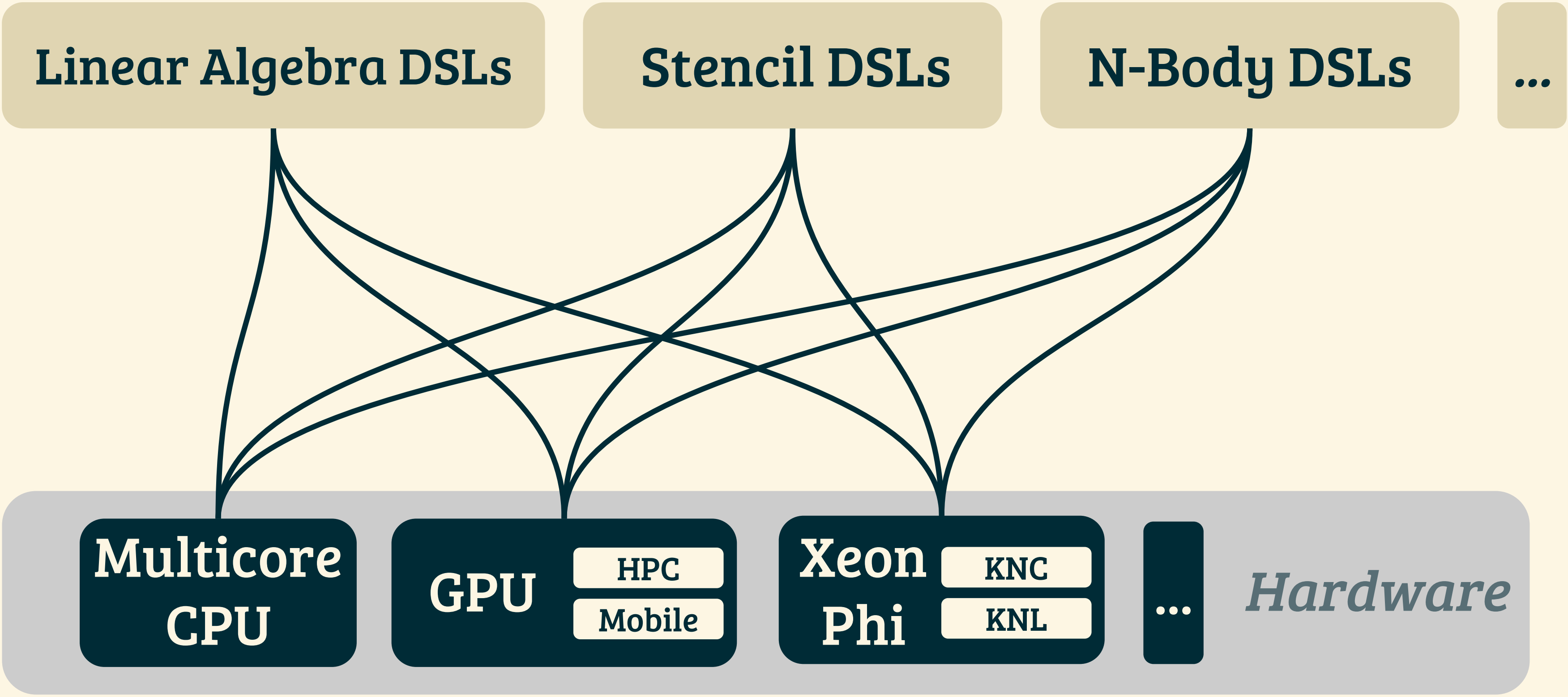
EXPLOITING DOMAIN KNOWLEDGE



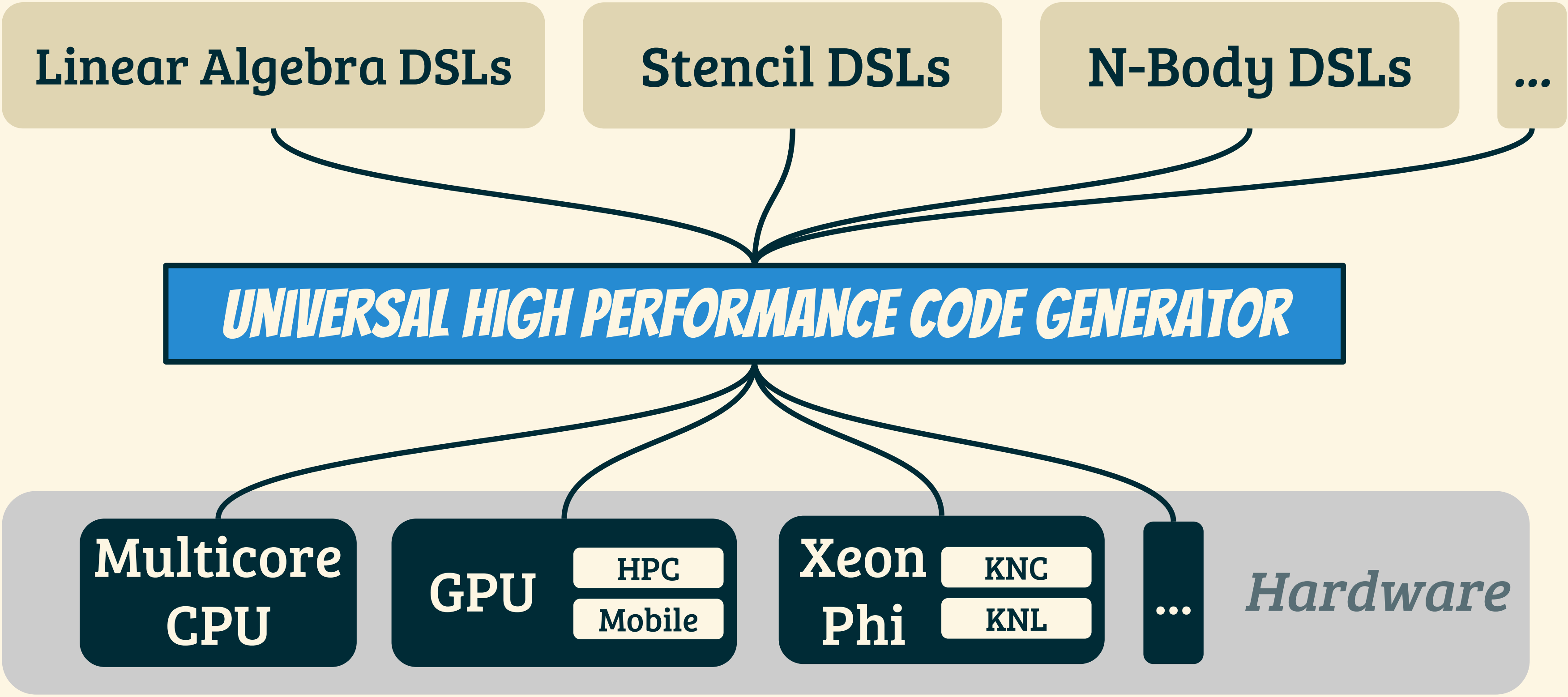
Exploiting Domain Knowledge



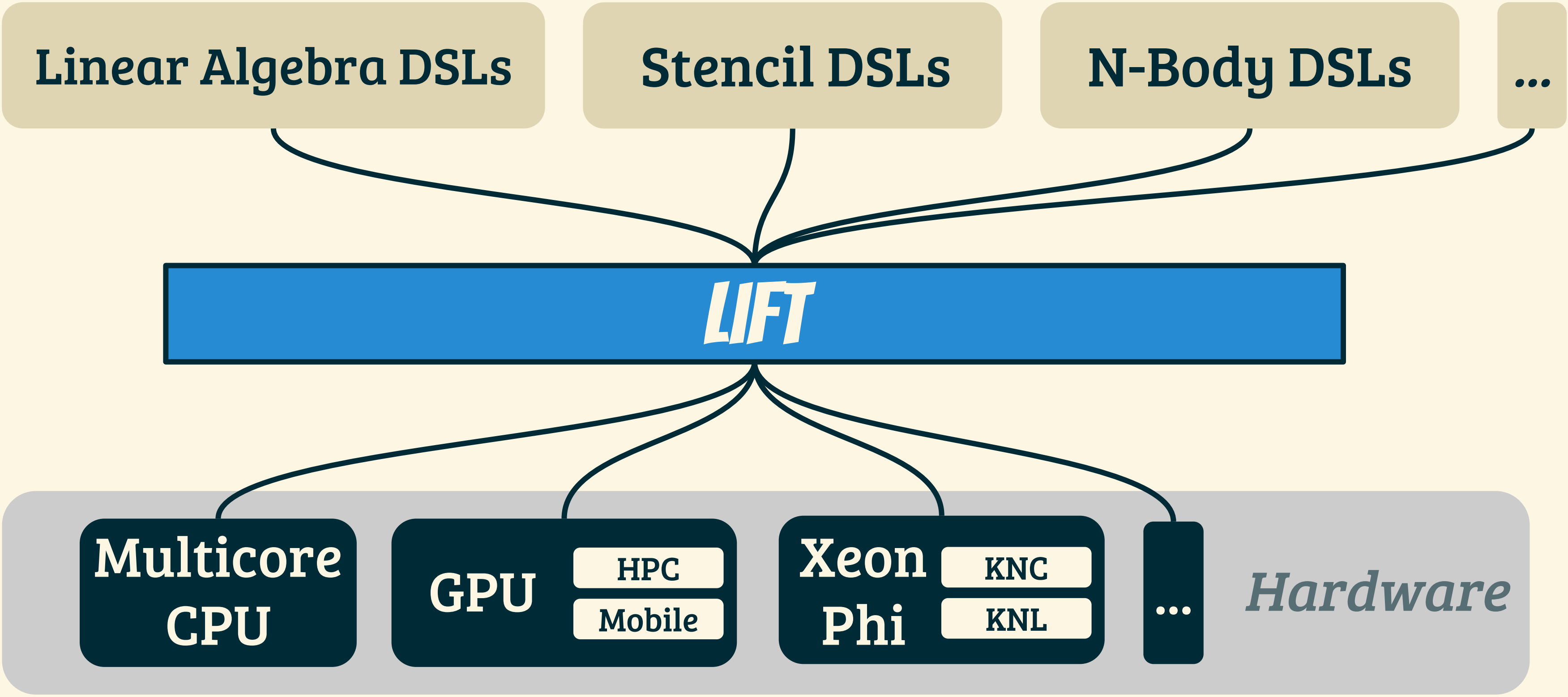
Exploiting Domain Knowledge



APPROACHING PERFORMANCE PORTABILITY

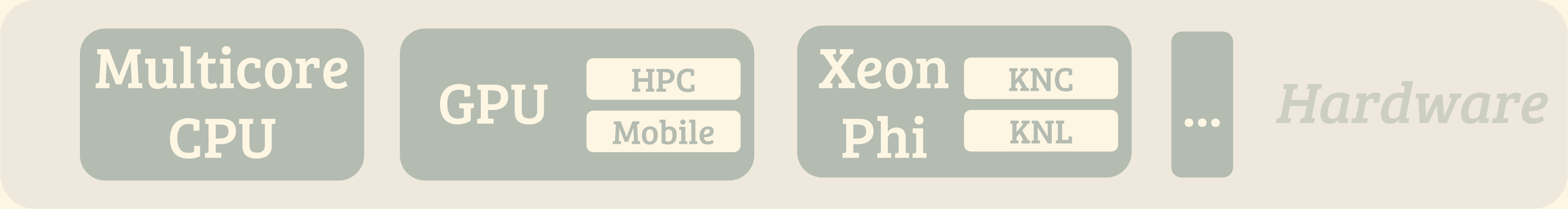


APPROACHING PERFORMANCE PORTABILITY



LIFT

DSL DSL DSL



Multicore CPU

GPU

HPC

Mobile

Xeon Phi

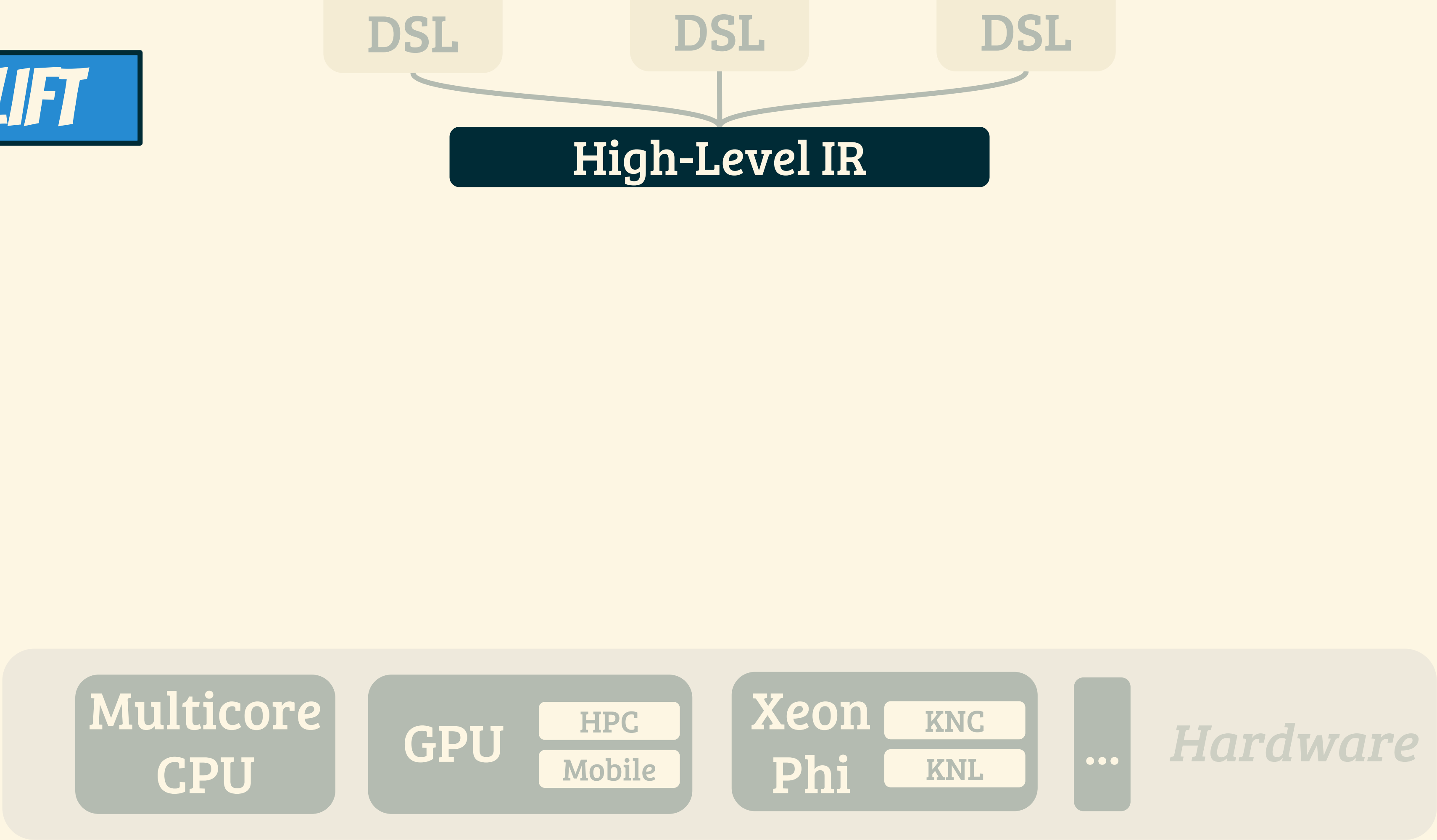
KNC

KNL

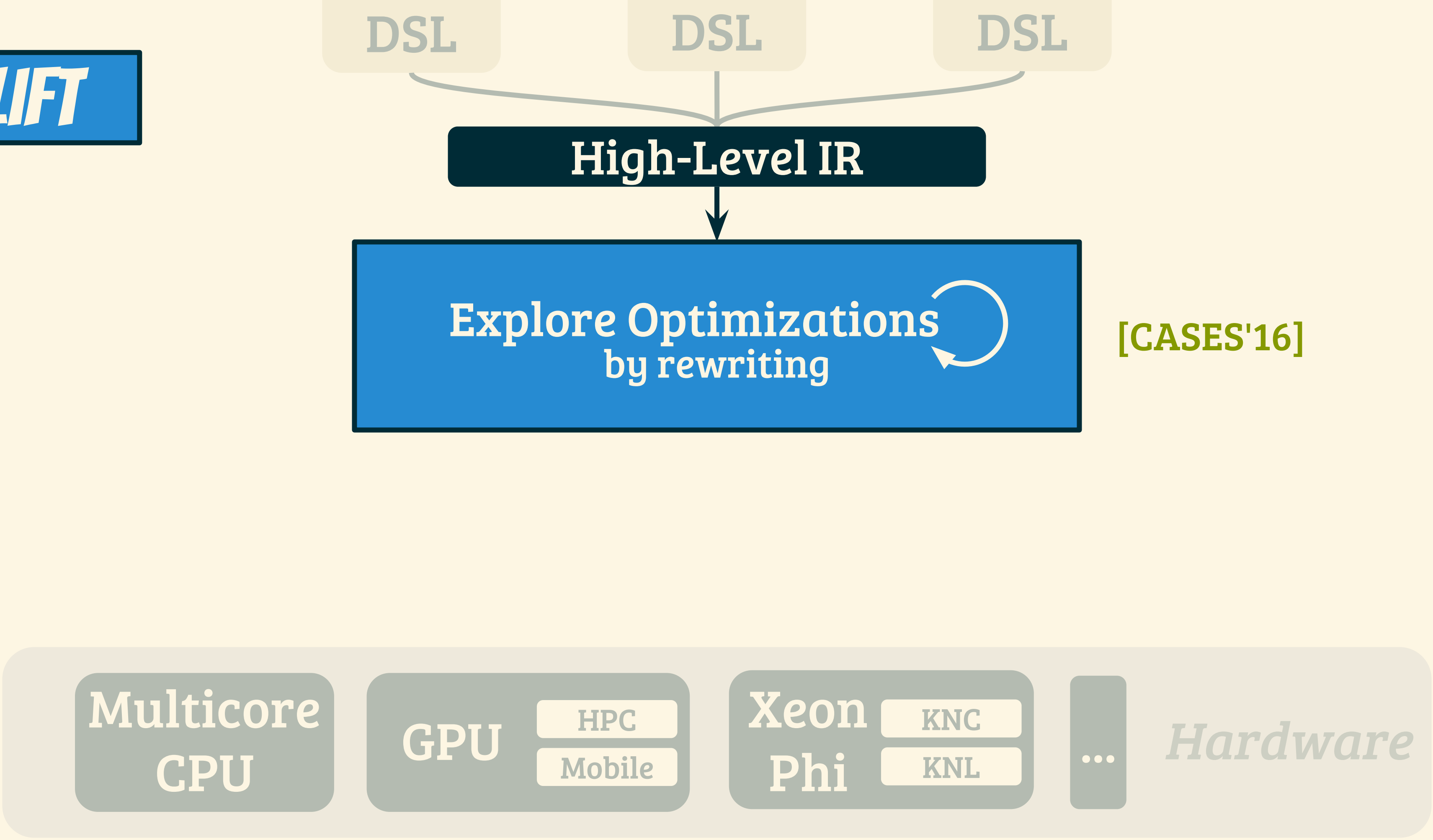
...

Hardware

LIFT


DSL
DSL
DSL
High-Level IR
Multicore CPU
GPU
HPC
Mobile
Xeon Phi
KNC
KNL
...
Hardware

LIFT



DSL

DSL

DSL

High-Level IR

Explore Optimizations
by rewriting

[CASES'16]

Multicore CPU

GPU

HPC

Mobile

Xeon Phi

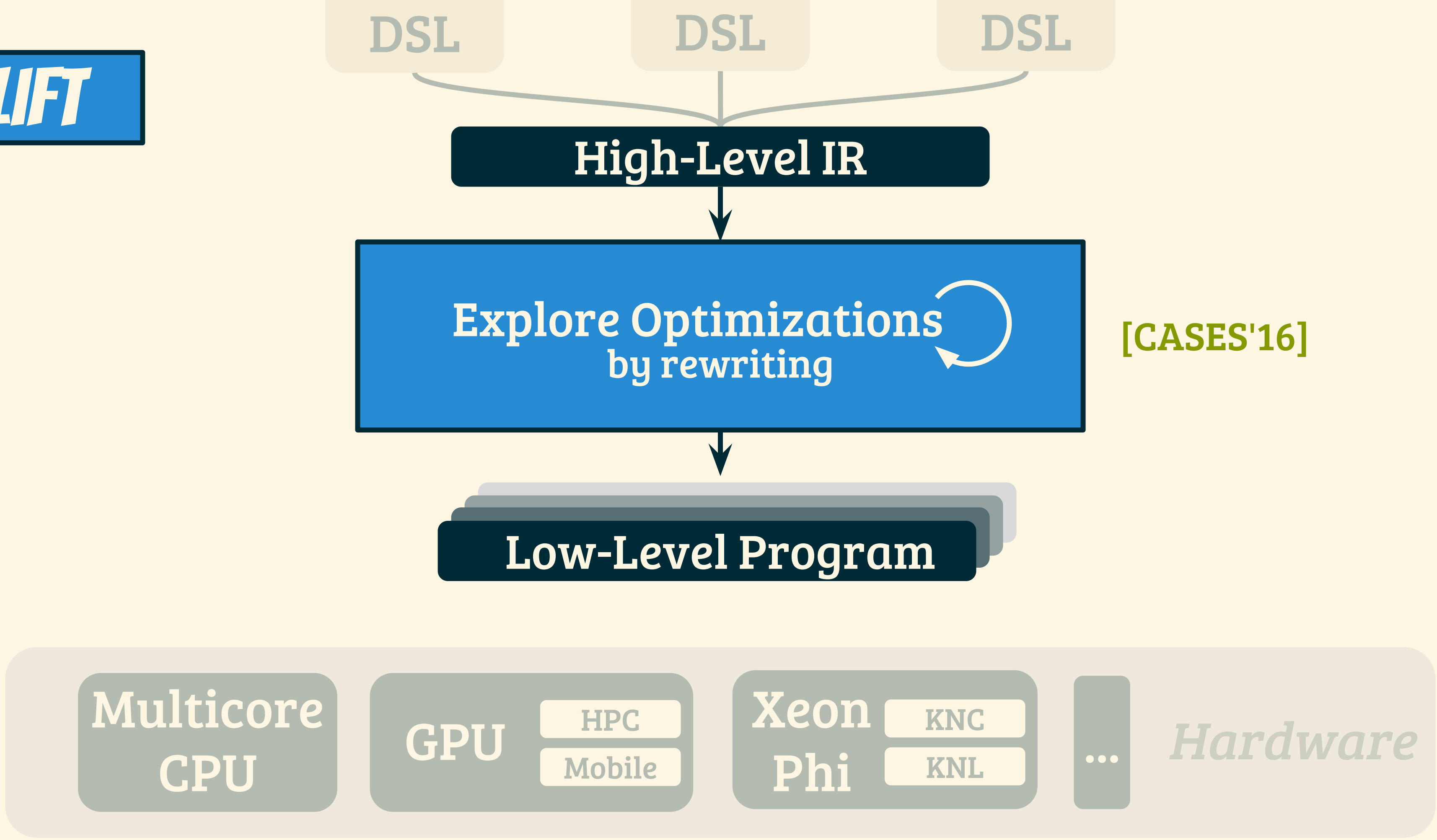
KNC

KNL

...

Hardware

LIFT



DSL DSL DSL

High-Level IR

Explore Optimizations
by rewriting

[CASES'16]

Low-Level Program

Multicore CPU

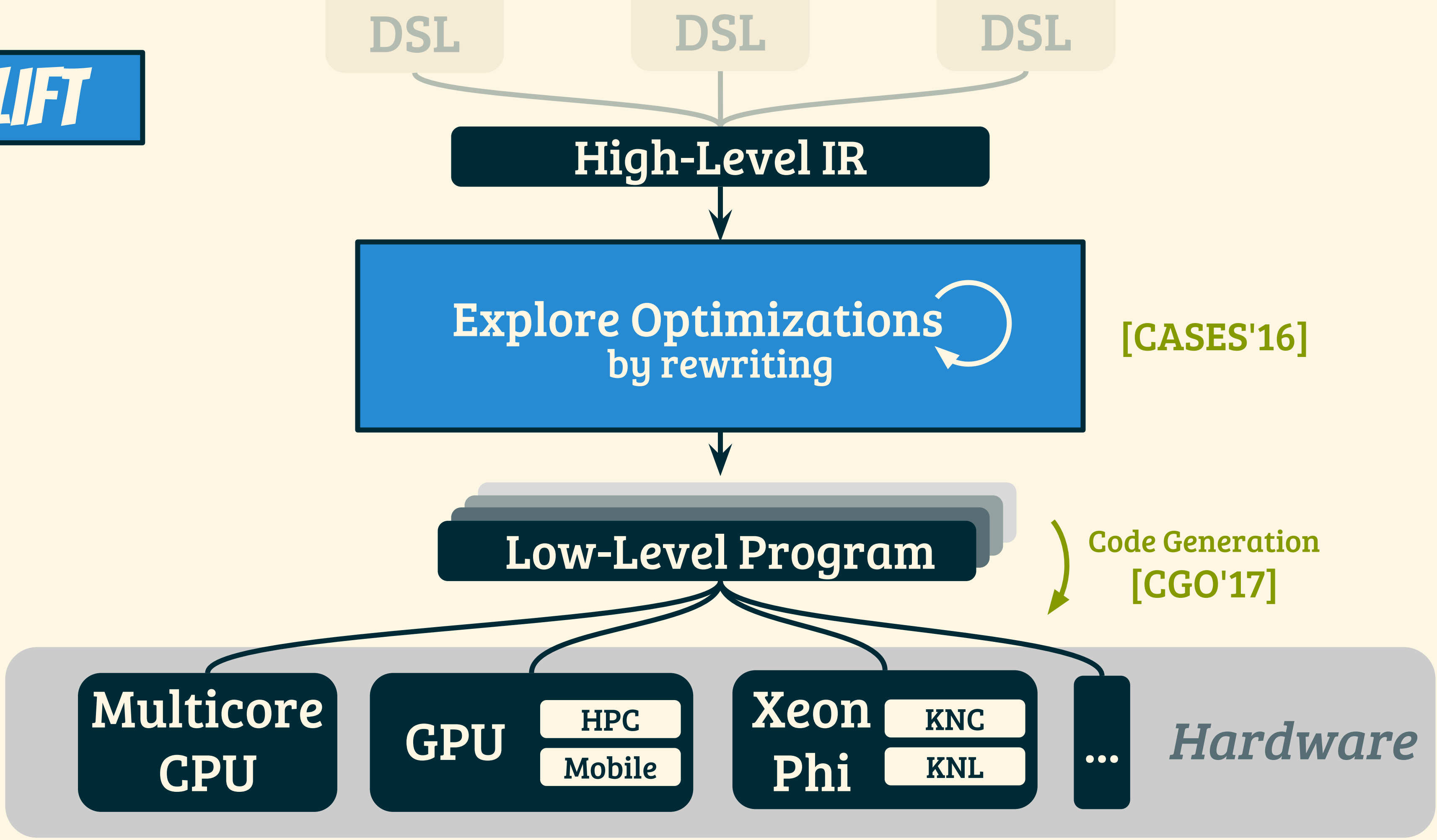
GPU
HPC
Mobile

Xeon Phi
KNC
KNL

...

Hardware

LIFT



DSL

DSL

DSL

High-Level IR

Explore Optimizations
by rewriting

[CASES'16]

Low-Level Program

Code Generation
[CGO'17]

Multicore
CPU

GPU

HPC

Mobile

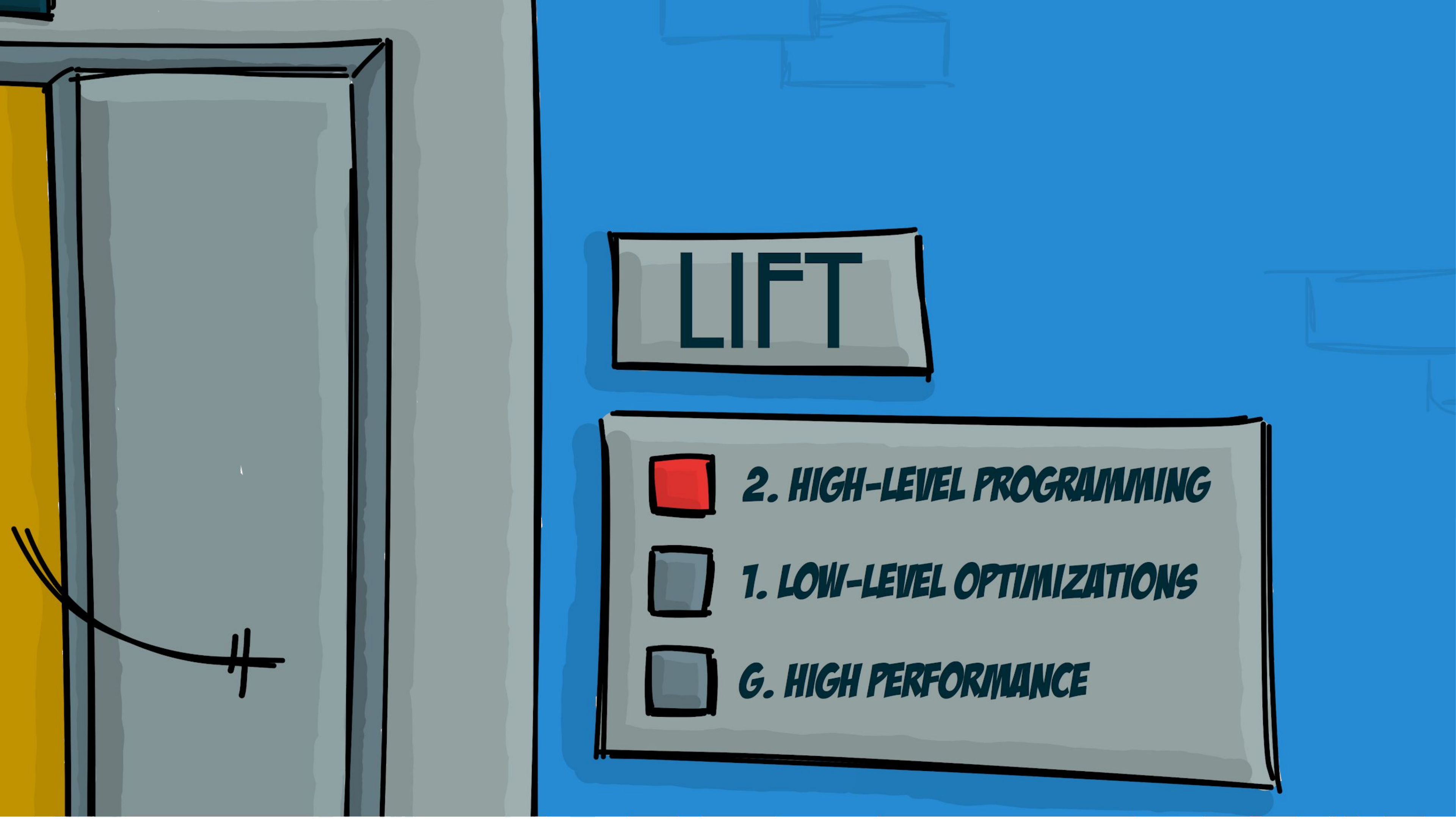
Xeon
Phi

KNC

KNL

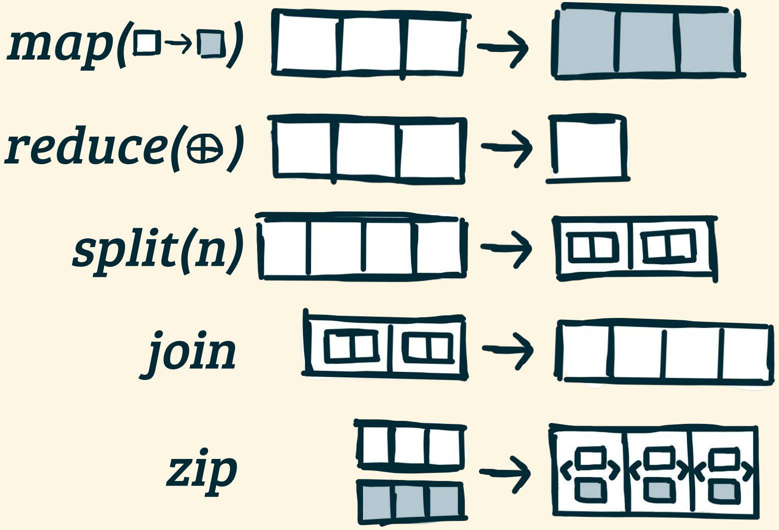
...

Hardware



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

LIFT'S HIGH-LEVEL PRIMITIVES



map(□→■)

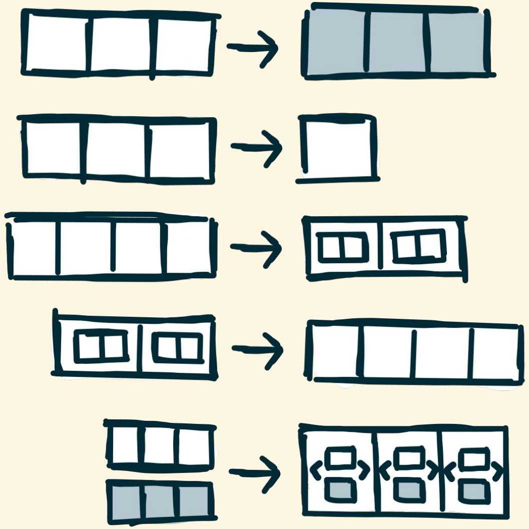
reduce(⊕)

split(n)

join

zip

LIFT'S HIGH-LEVEL PRIMITIVES



map(□→■)

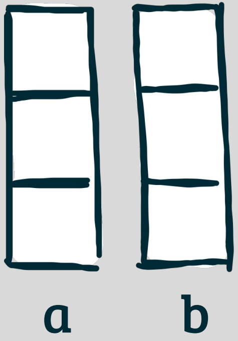
reduce(⊕)

split(n)

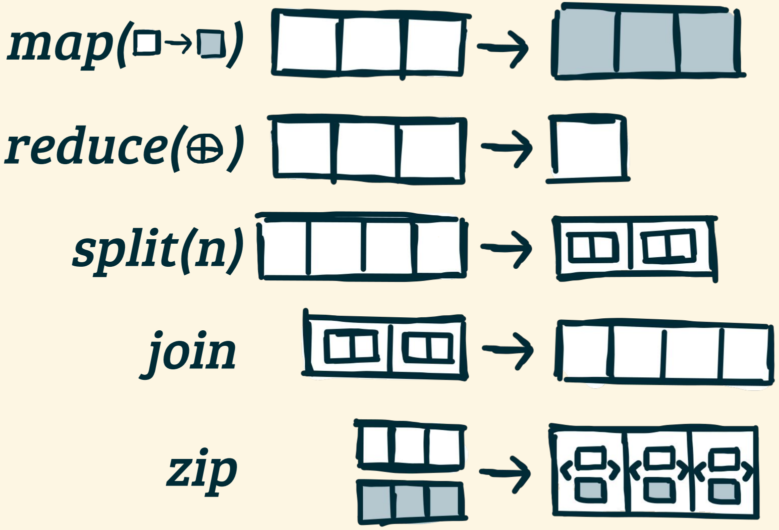
join

zip

dotproduct.lift



LIFT'S HIGH-LEVEL PRIMITIVES



map(□→▨)

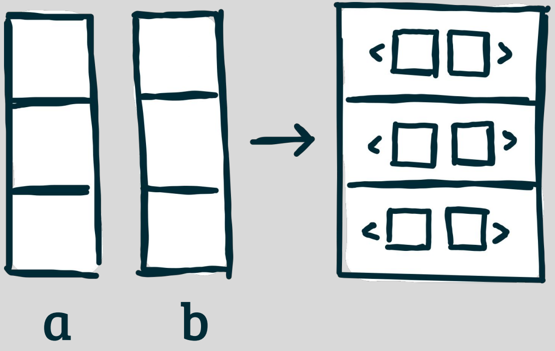
reduce(⊕)

split(n)

join

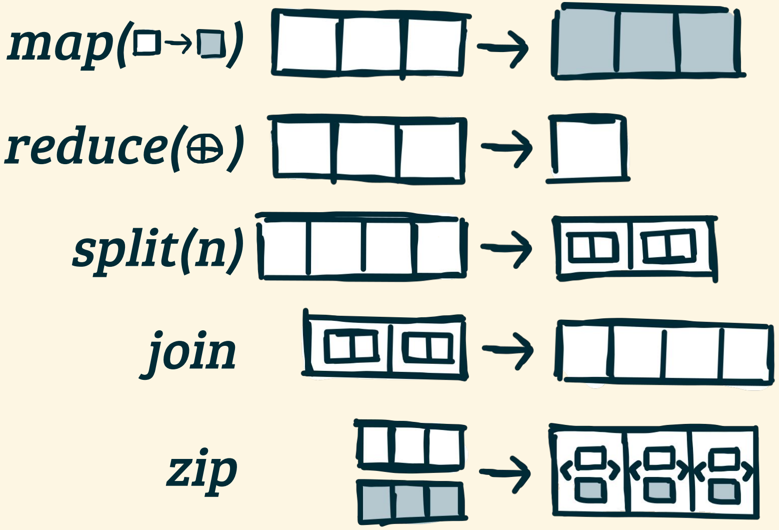
zip

dotproduct.lift



```
zip(a,b)
```

LIFT'S HIGH-LEVEL PRIMITIVES



map(□→▨)

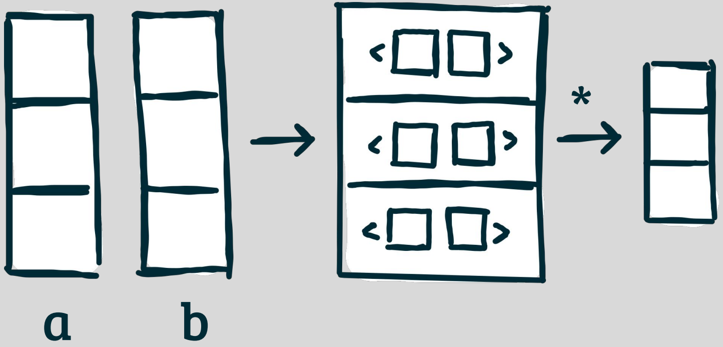
reduce(⊕)

split(*n*)

join

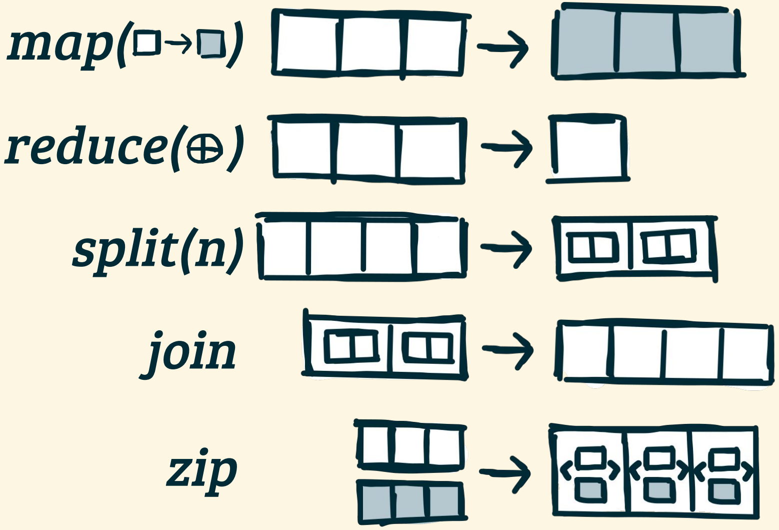
zip

dotproduct.lift



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

LIFT'S HIGH-LEVEL PRIMITIVES



map(□→■)

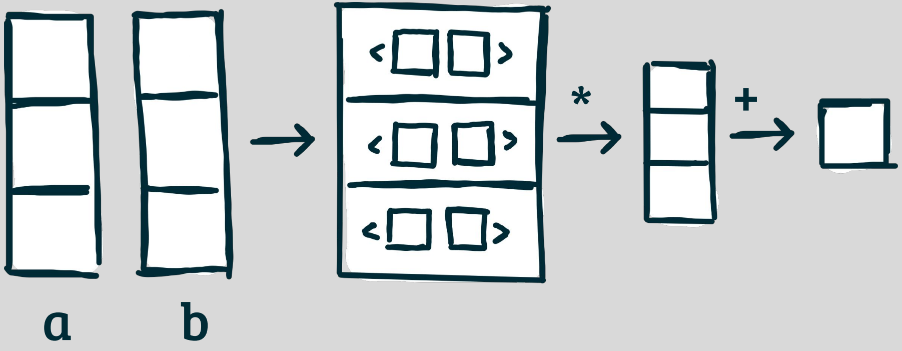
reduce(⊕)

split(n)

join

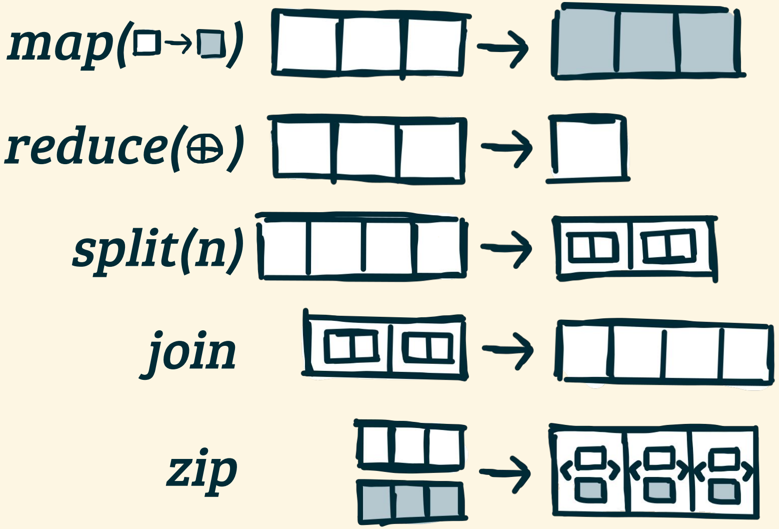
zip

dotproduct.lift



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

LIFT'S HIGH-LEVEL PRIMITIVES



map(□→□)

reduce(⊕)

split(n)

join

zip

stencil.lift?

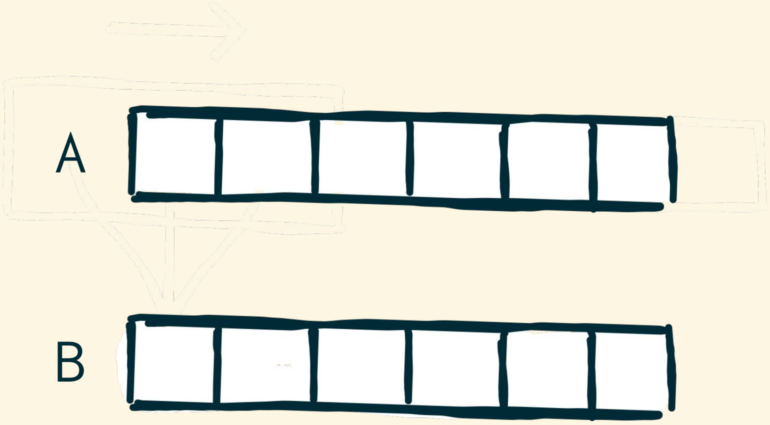
Can we express stencil computations in Lift?

Let's look at a simple stencil example...

WHAT ARE STENCIL COMPUTATIONS?

3-point-stencil.c

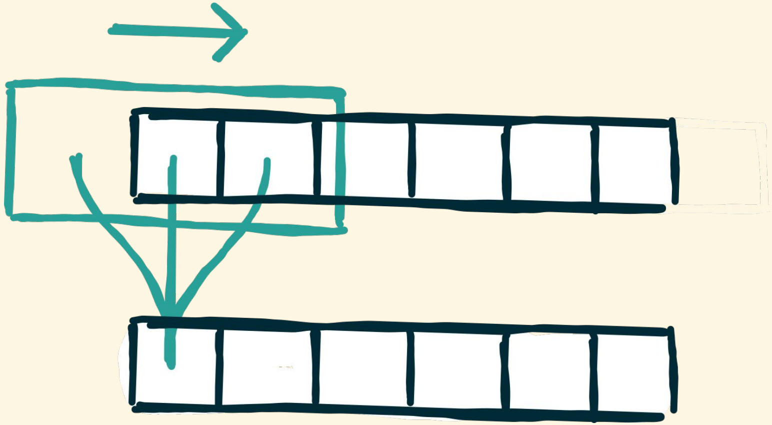
```
for (int i = 0; i < N ; i ++) {
    int sum = 0;
    for ( int j = -1; j <= 1; j ++) {
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[ pos ]; }
 B[ i ] = sum ; }
```



WHAT ARE STENCIL COMPUTATIONS?

3-point-stencil.c

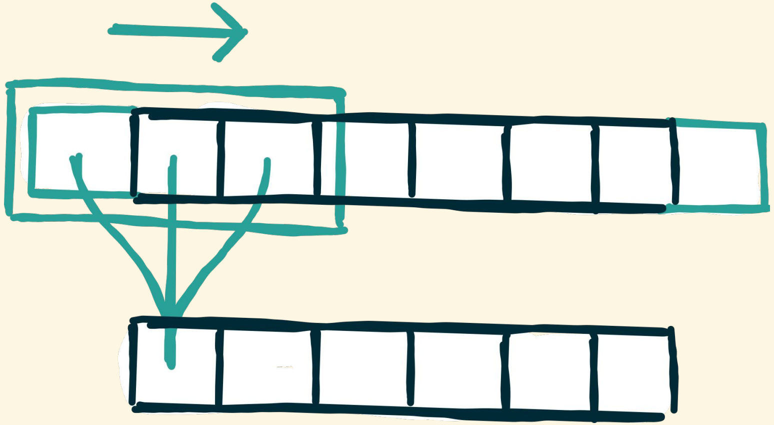
```c
for (int i = 0; i < N ; i ++) {
    int sum = 0;
    for ( int j = -1; j <= 1; j ++) {
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[ pos ]; }
 B[ i ] = sum ; }
```



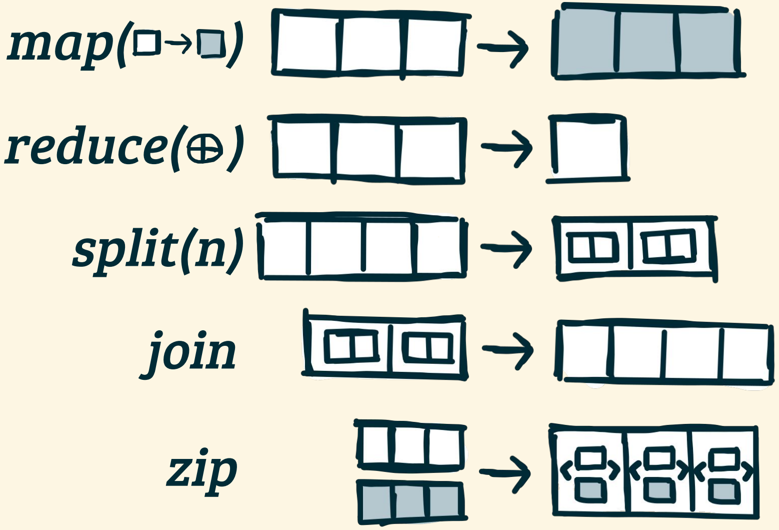
What are stencil computations?

3-point-stencil.c

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



STENCIL COMPUTATIONS IN LIFT



map(□→■)

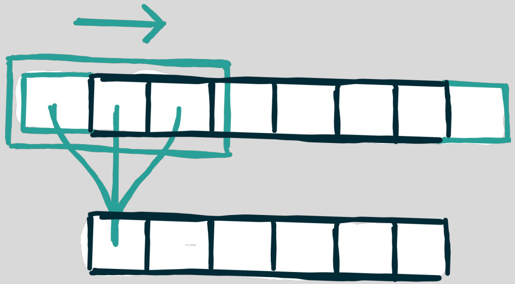
reduce(⊕)

split(n)

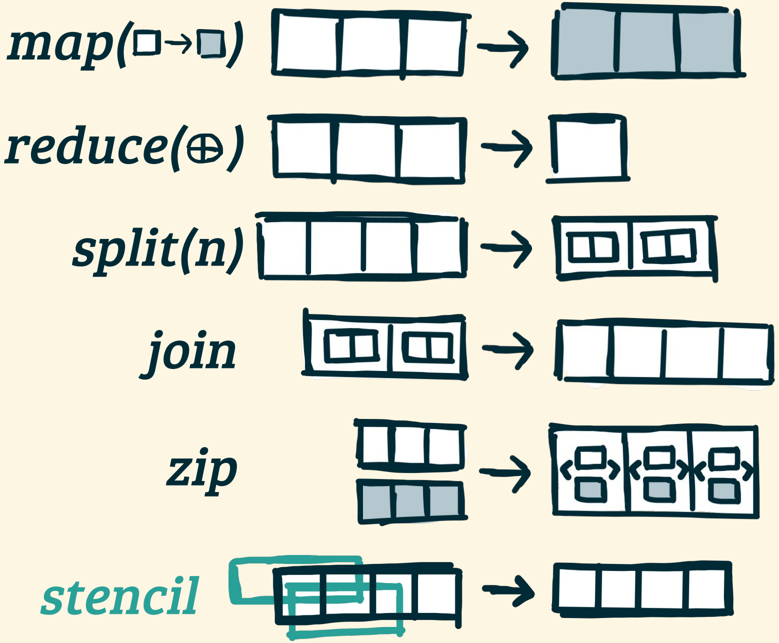
join

zip

3-point-stencil.lift



STENCIL COMPUTATIONS IN LIFT



map(□→■)

reduce(⊕)

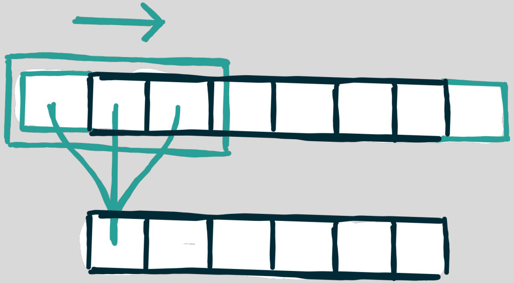
split(n)

join

zip

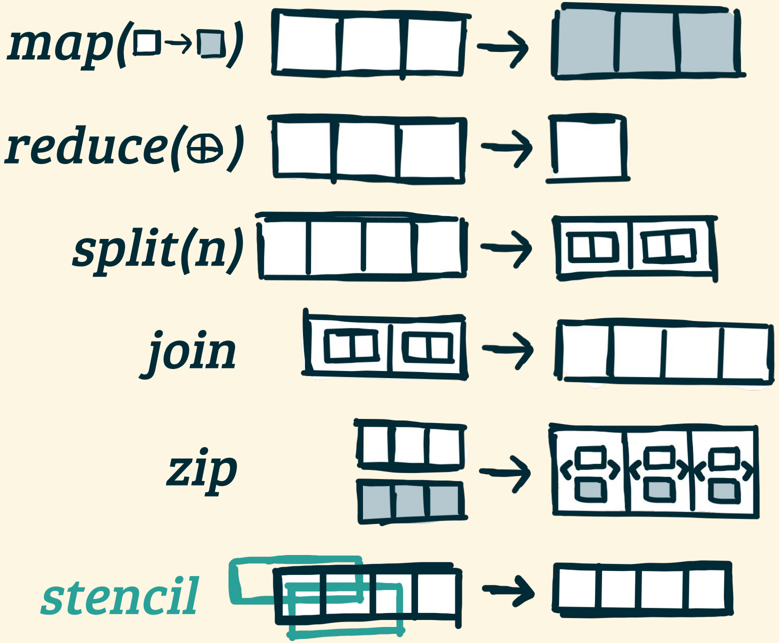
stencil

3-point-stencil.lift



Add specialized primitive: Job done?

STENCIL COMPUTATIONS IN LIFT



map(□→▨)

reduce(⊕)

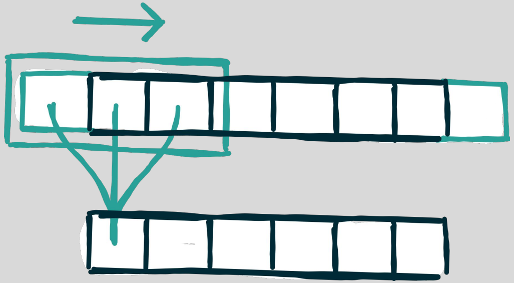
split(n)

join

zip

stencil

3-point-stencil.lift



Add specialized primitive: Job done?

⊘ **No Reuse**
of existing primitives and optimizations

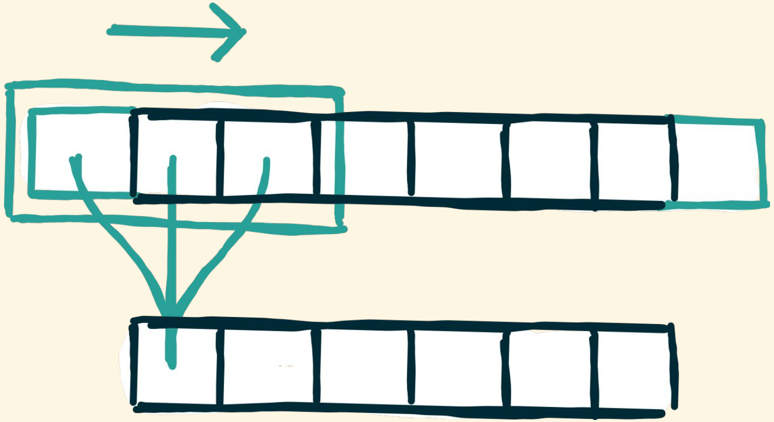
⊘ **Domain-specific**
rather than generic

⊘ **Multidimensional?**
is it composable?

Decomposing Stencil Computations

3-point-stencil.c

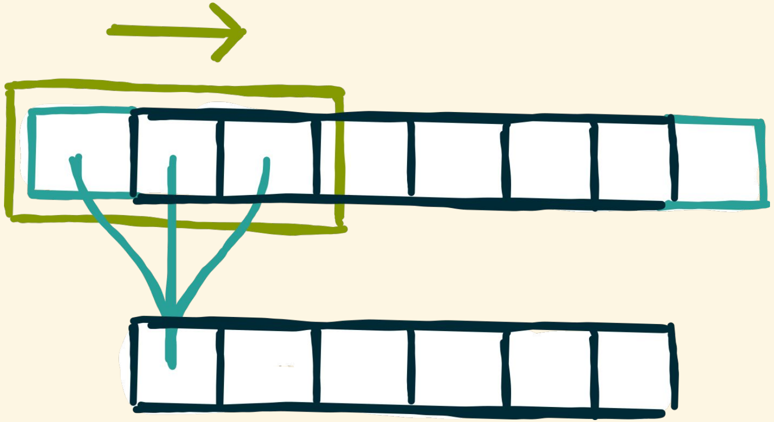
```c
for (int i = 0; i < N ; i ++) {
    int sum = 0;
    for ( int j = -1; j <= 1; j ++) {
        int pos = i + j;
        pos = pos < 0 ? 0 : pos;
        pos = pos > N - 1 ? N - 1 : pos;
        sum += A[ pos ]; }
 B[ i ] = sum ; }
```



DECOMPOSING STENCIL COMPUTATIONS

3-point-stencil.c

```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 ; }
```

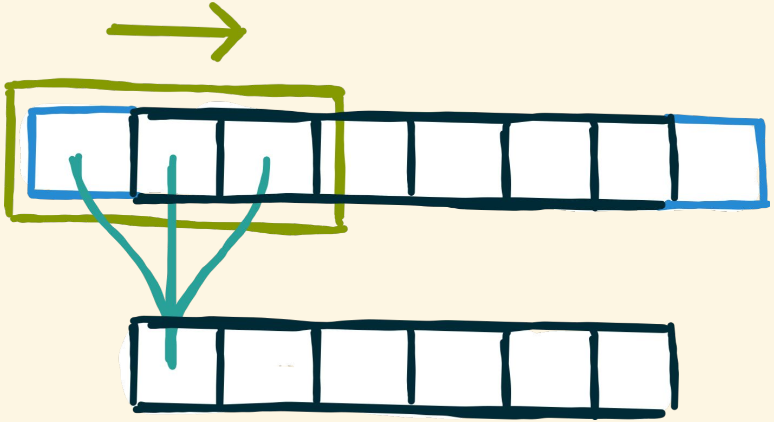


(a) access neighborhoods for every element

DECOMPOSING STENCIL COMPUTATIONS

3-point-stencil.c

```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 ; }
```



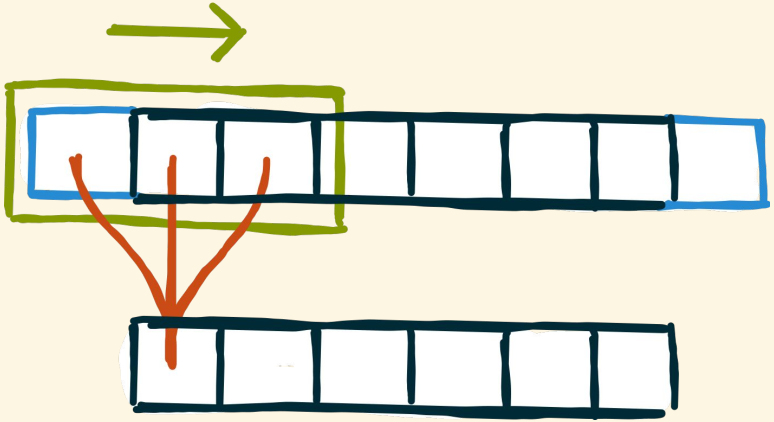
(a) access neighborhoods for every element

(b) specify boundary handling

DECOMPOSING STENCIL COMPUTATIONS

3-point-stencil.c

```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

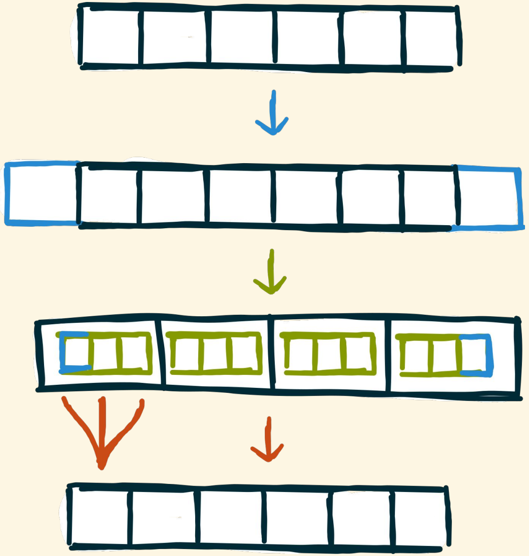
(b) specify **boundary handling**

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

DECOMPOSING STENCIL COMPUTATIONS

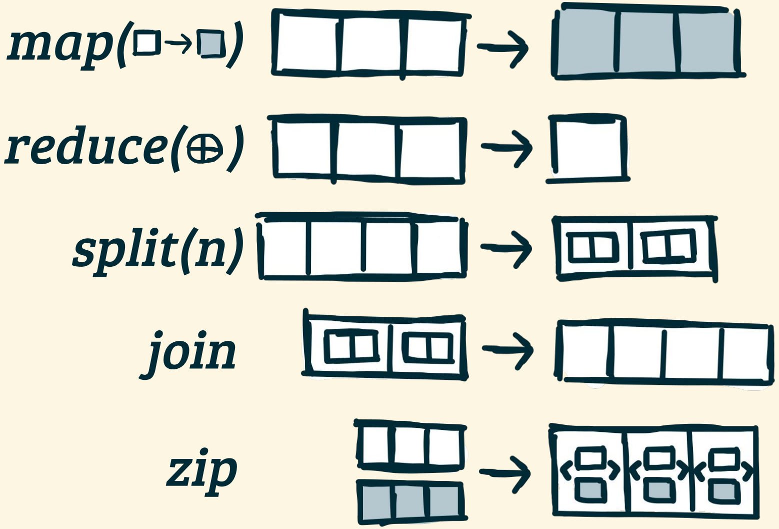
3-point-stencil.c

```
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

STENCIL COMPUTATIONS IN LIFT



map(□→□)

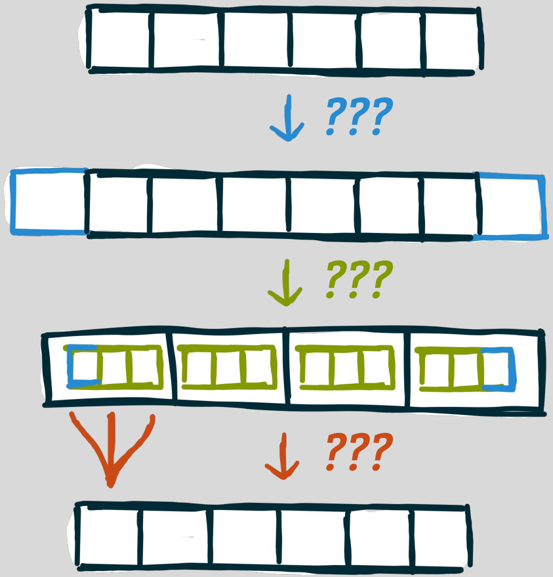
reduce(⊕)

split(n)

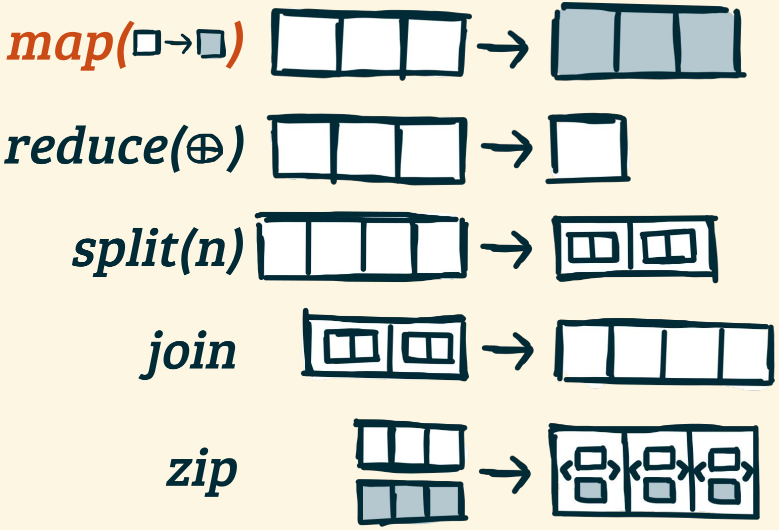
join

zip

3-point-stencil.lift

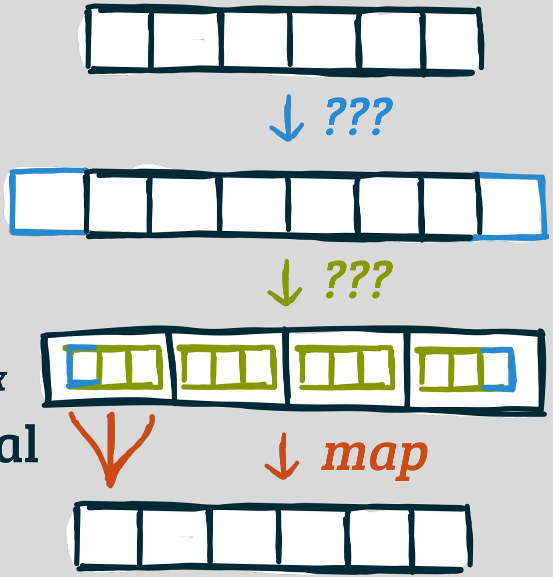


STENCIL COMPUTATIONS IN LIFT



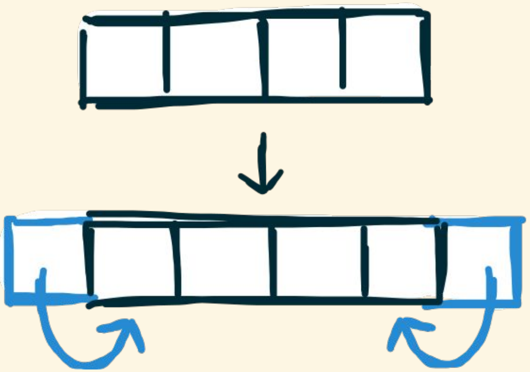
3-point-stencil.lift

- ✓ **Reuse *map***
  allows to reuse existing rewrite rules
- ✓ **Simplicity**
  one primitive per task
- ✓ **Multidimensional**
  easily composable



Boundary Handling using pad

pad (reindexing)

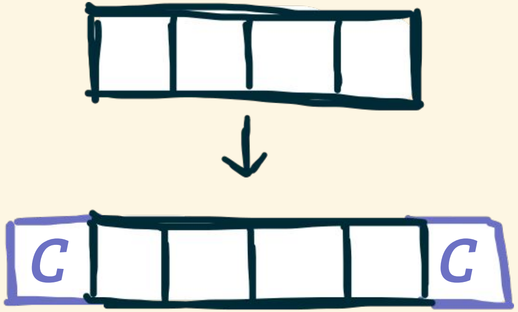


pad-reindexing.lift

```
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)



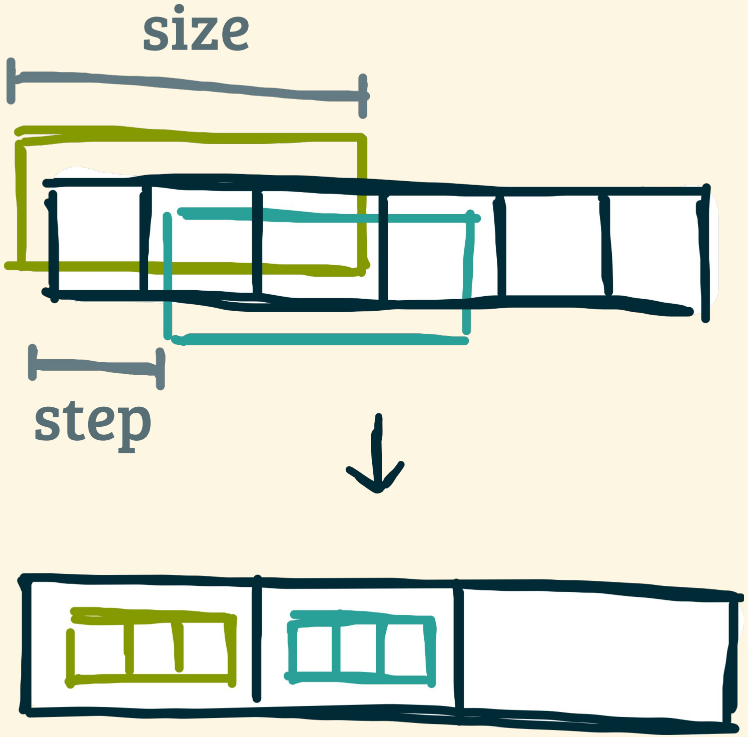
pad-constant.lift

```
constant(i, n) = C



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

Neighborhood Creation using slide

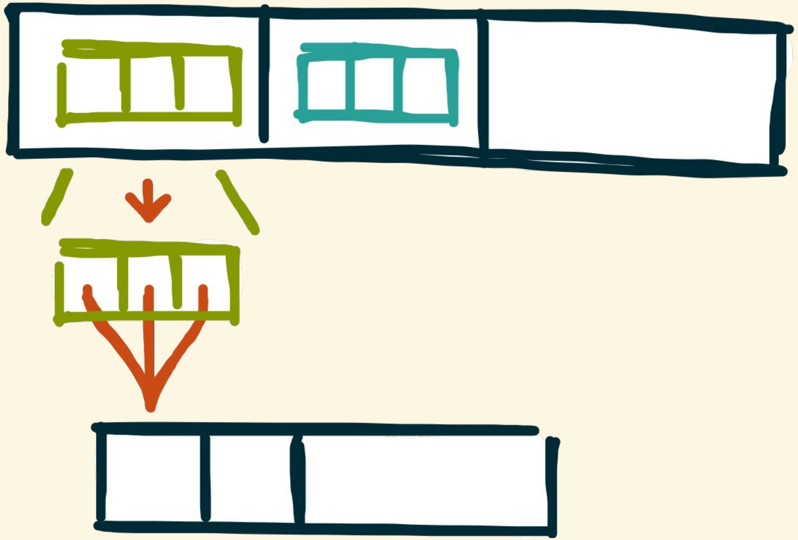


slide-example.lift

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

    [[a,b,c],[b,c,d],[c,d,e]]
```

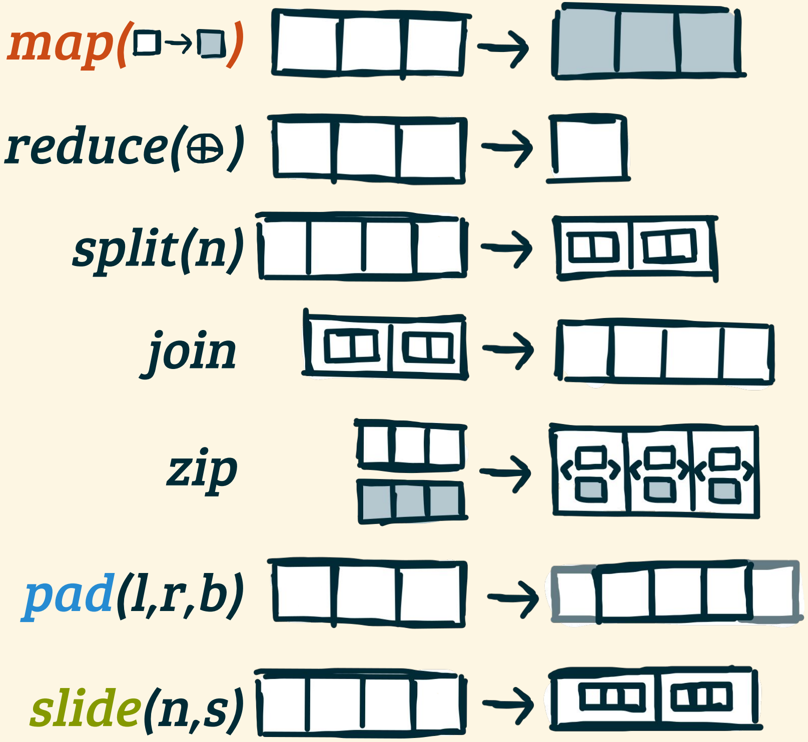
APPLYING STENCIL FUNCTION USING *MAP*



sum-neighborhoods.lift

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

PUTTING IT TOGETHER



map(□→□)

reduce(⊕)

split(n)

join

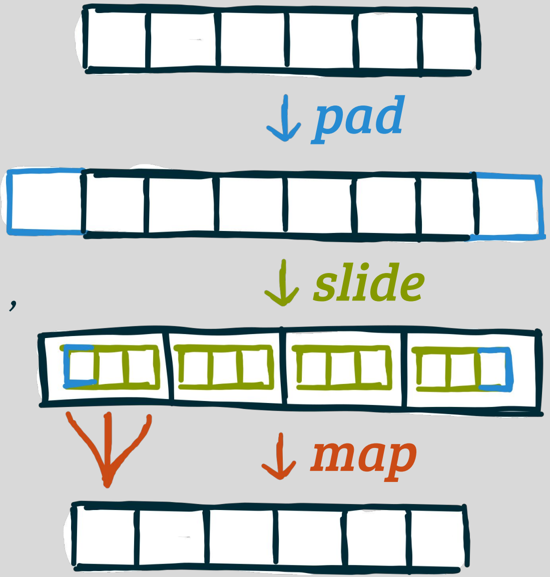
zip

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))))
```



pad

slide

map

Multidimensional Stencil Computations

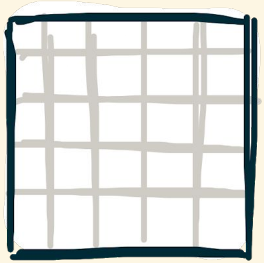
are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

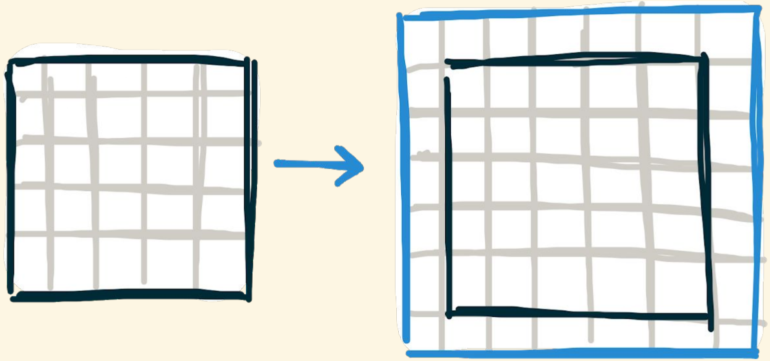
Decompose to Re-Compose



MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

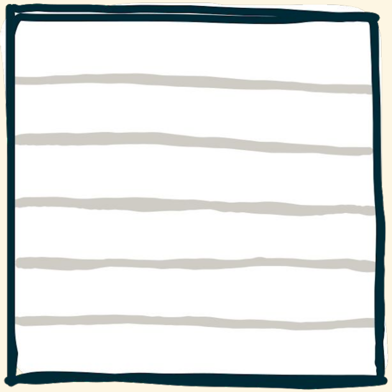
Decompose to Re-Compose



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

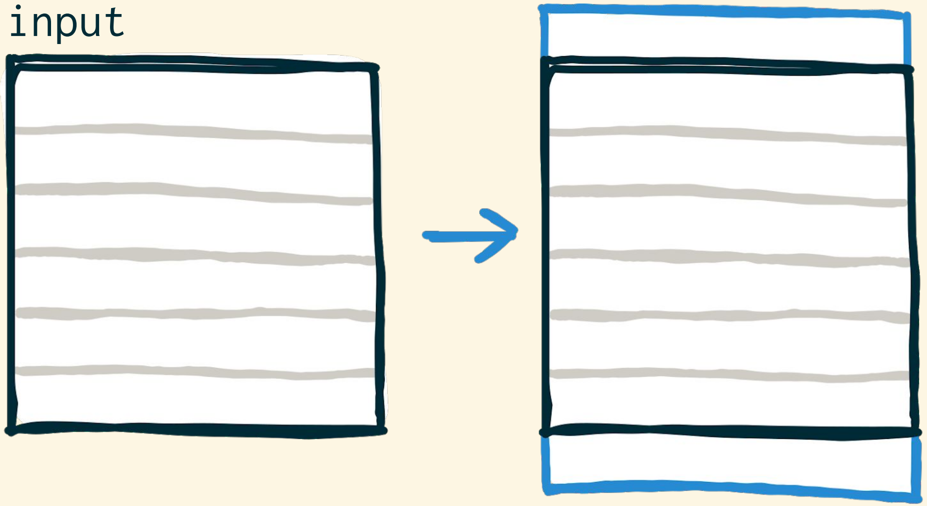
MULTIDIMENSIONAL BOUNDARY HANDLING USING PAD_2

input



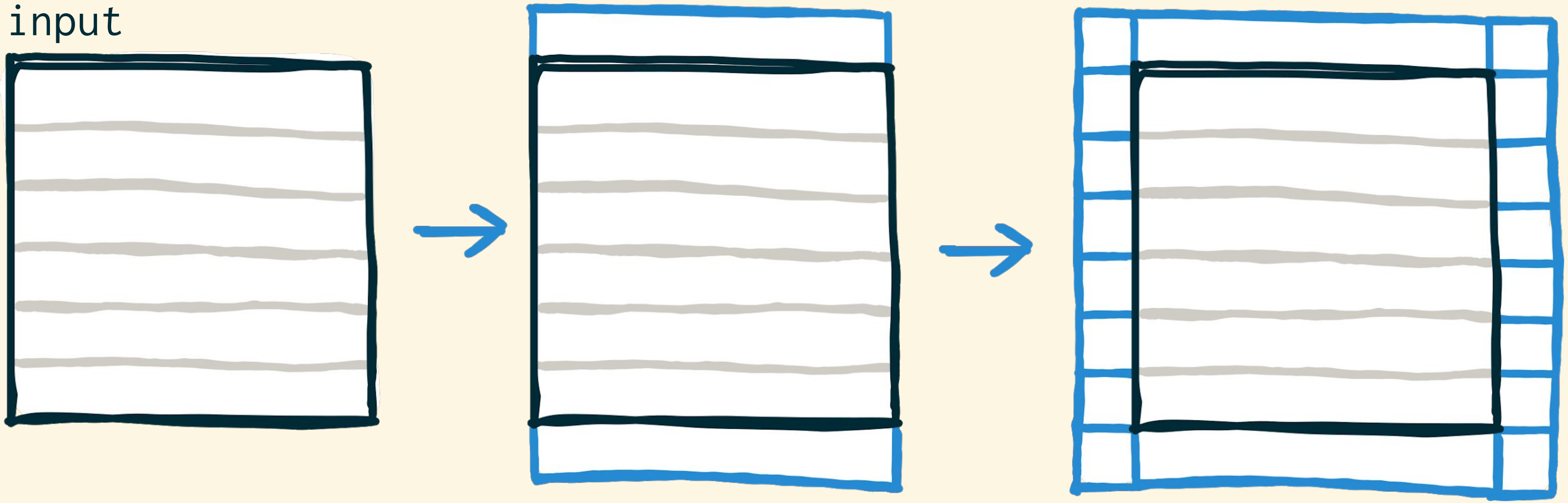
pad_2 =

MULTIDIMENSIONAL BOUNDARY HANDLING USING PAD_2



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

MULTIDIMENSIONAL BOUNDARY HANDLING USING PAD_2

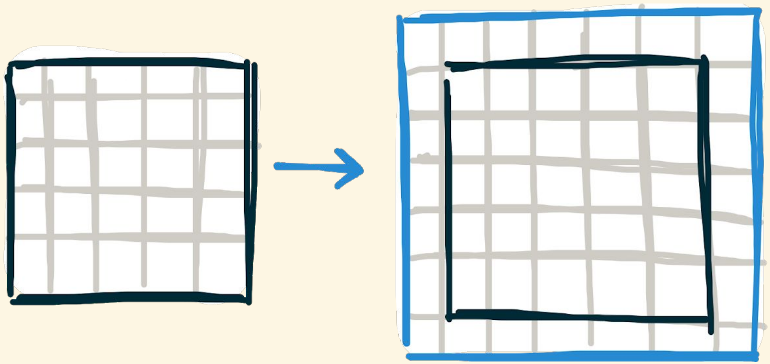


$$\mathit{pad}_2 = \mathit{map}(\mathit{pad}(l,r,b,\mathit{pad}(l,r,b,\mathrm{input})))$$

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

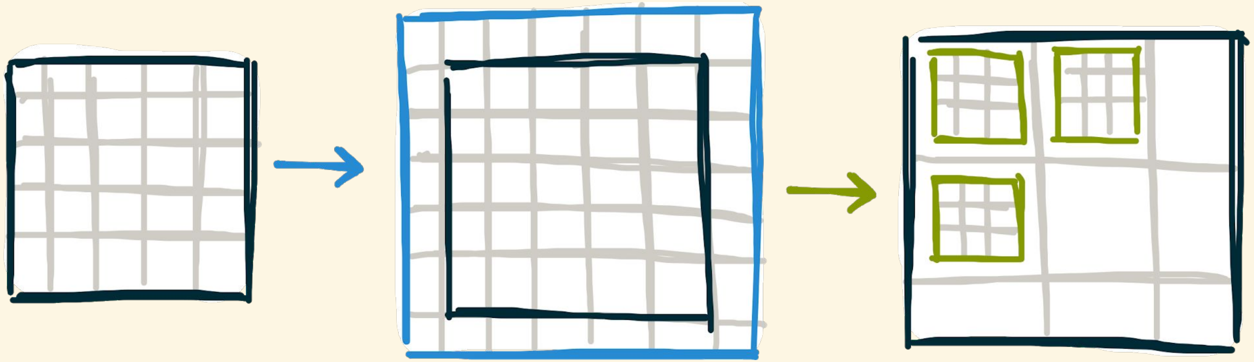


pad$_2$(1,1,*clamp*,input)

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

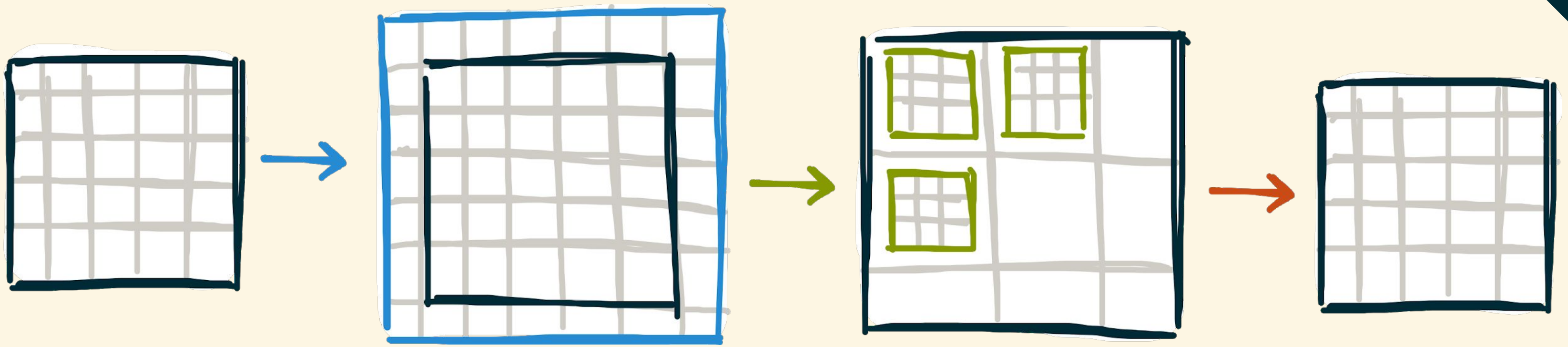


slide_2(3,1, pad_2(1,1,*clamp*,input))

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

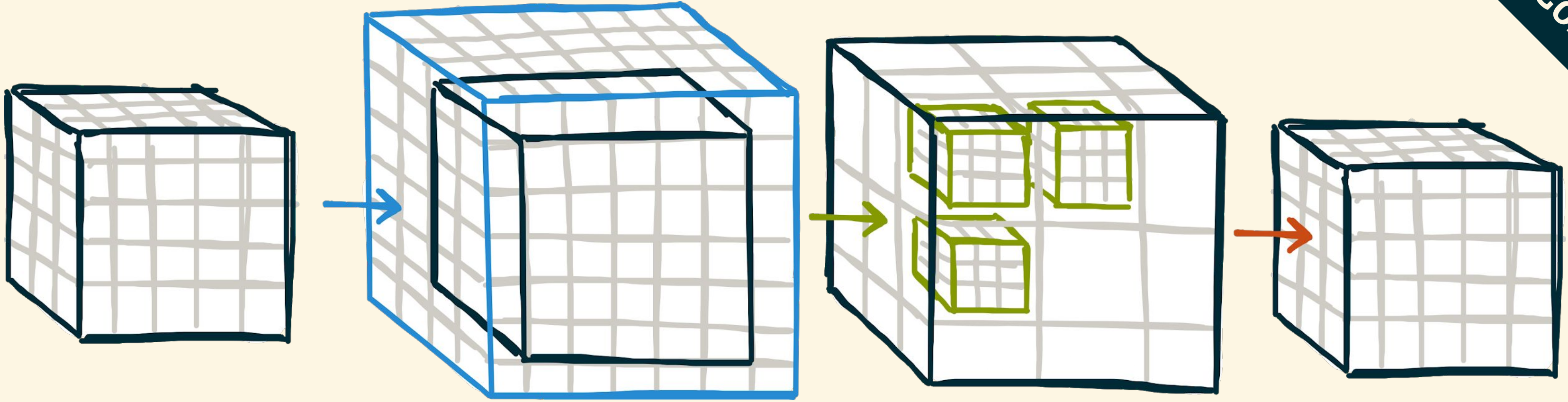


$map_2(sum,\ slide_2(3,1,\ pad_2(1,1,clamp,input)))$

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

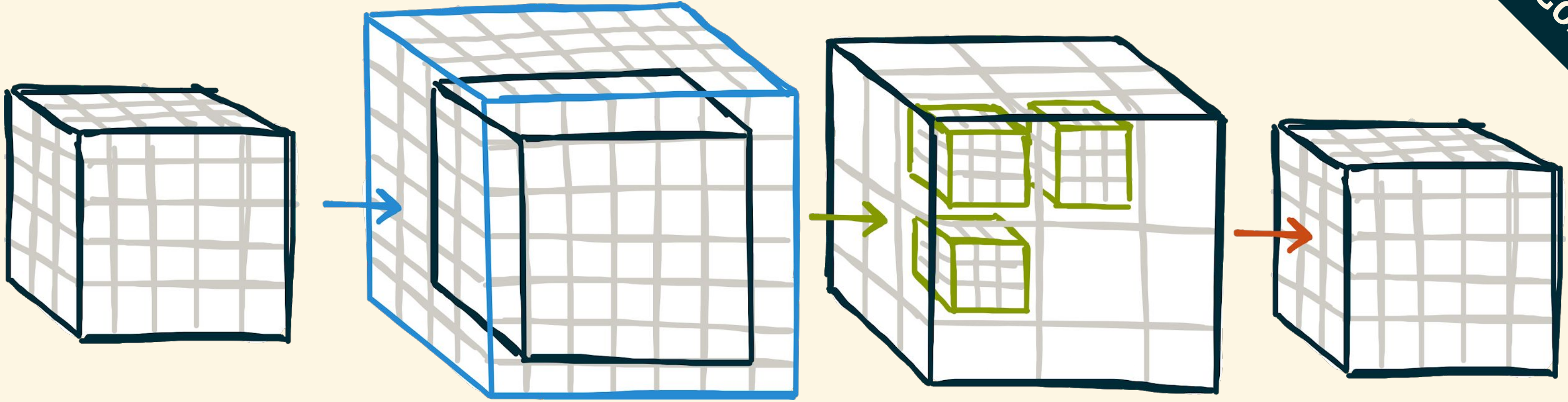


$map_3(sum,\ slide_3(3,1,\ pad_3(1,1,clamp,\text{input})))$

MULTIDIMENSIONAL STENCIL COMPUTATIONS

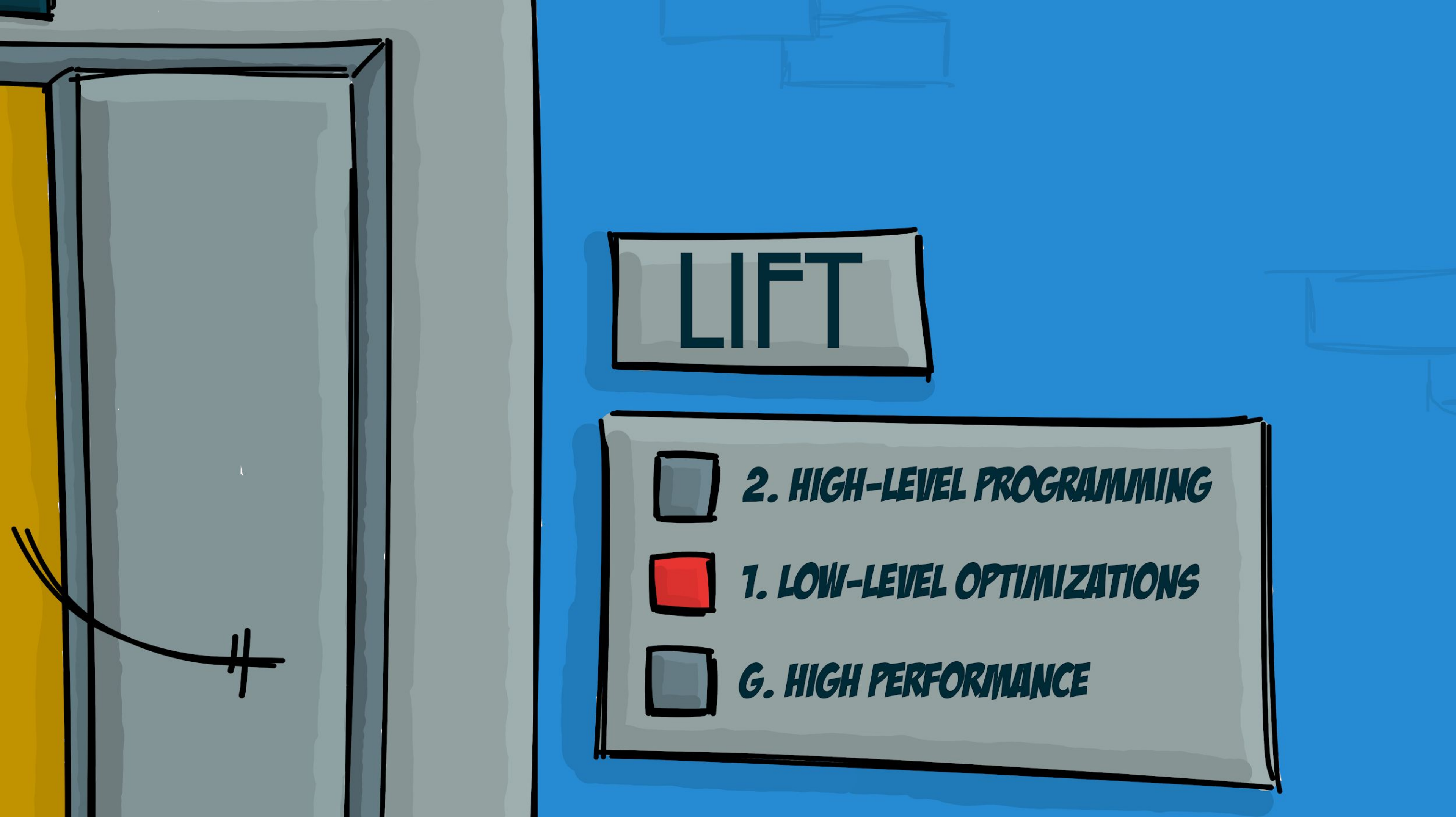
are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose



$\mathit{map}_3(\mathit{sum},\ \mathit{slide}_3(3,1,\ \mathit{pad}_3(1,1,\mathit{clamp},\mathrm{input})))$

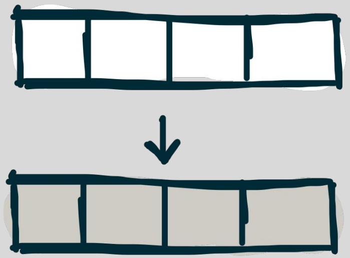
Advantages: ✓ Compact Language ✓ Reuse Rewrites ✓ Simple Compilation


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

Reusing Existing Rewrite Rules

Divide & Conquer

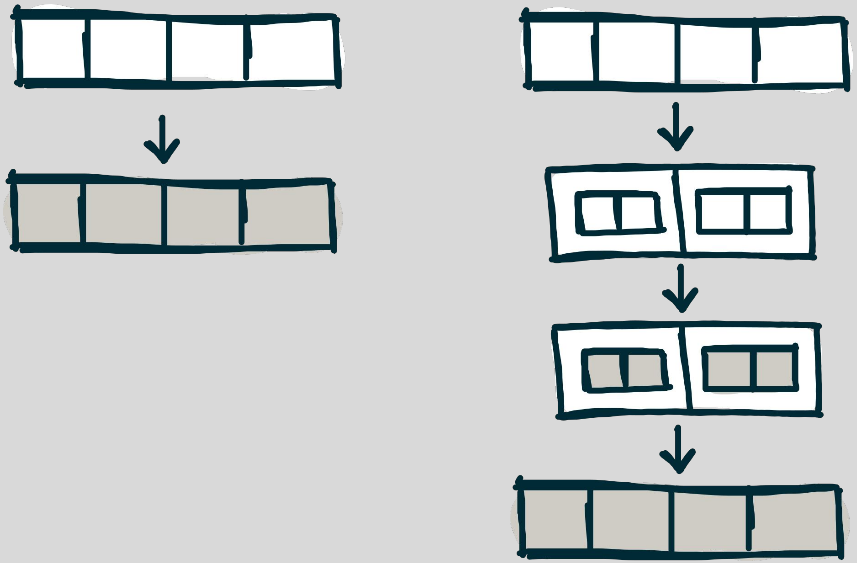
`map(f, A)`



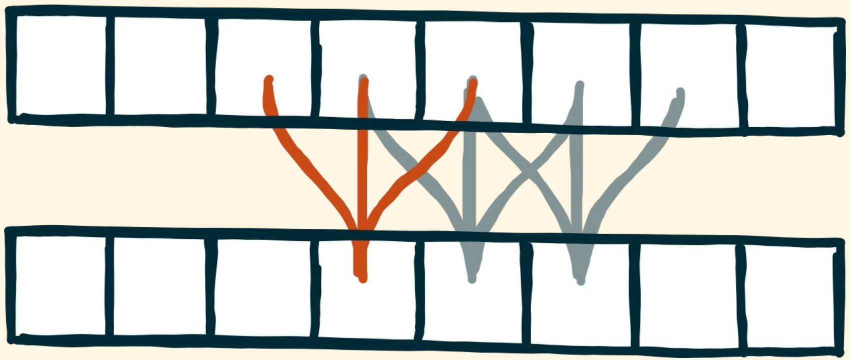
REUSING EXISTING REWRITE RULES

Divide & Conquer

```
map(f, A)   ↦   join(map(map(f),
                  split(n, A)))
```



OPTIMIZATION: OVERLAPPED TILING



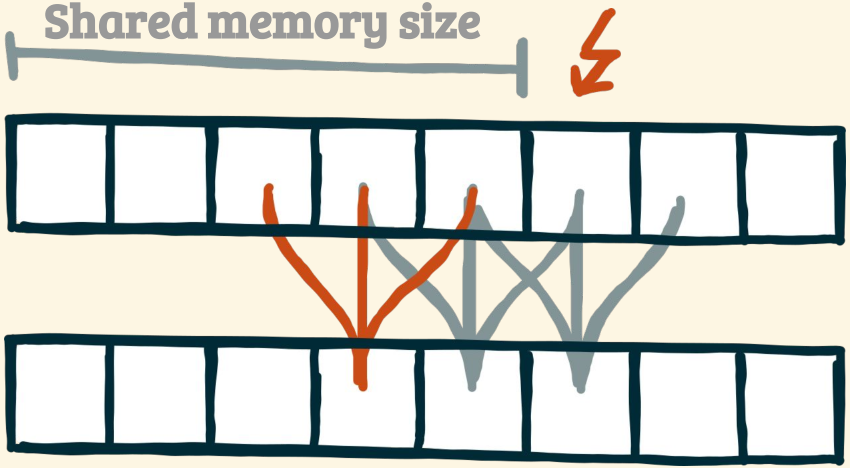
✓ **Exploit Locality**

Close neighborhoods share elements that can be grouped in tiles

✓ **Shared Memory**

Fast memory can be used to cache tiles

OPTIMIZATION: OVERLAPPED TILING



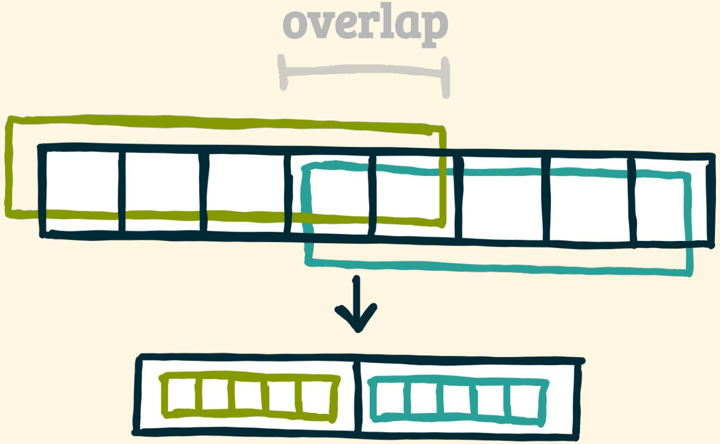
- ✓ **Exploit Locality**

  Close neighborhoods share elements that can be grouped in tiles

- ✓ **Shared Memory**

  Fast memory can be used to cache tiles

OPTIMIZATION: OVERLAPPED TILING



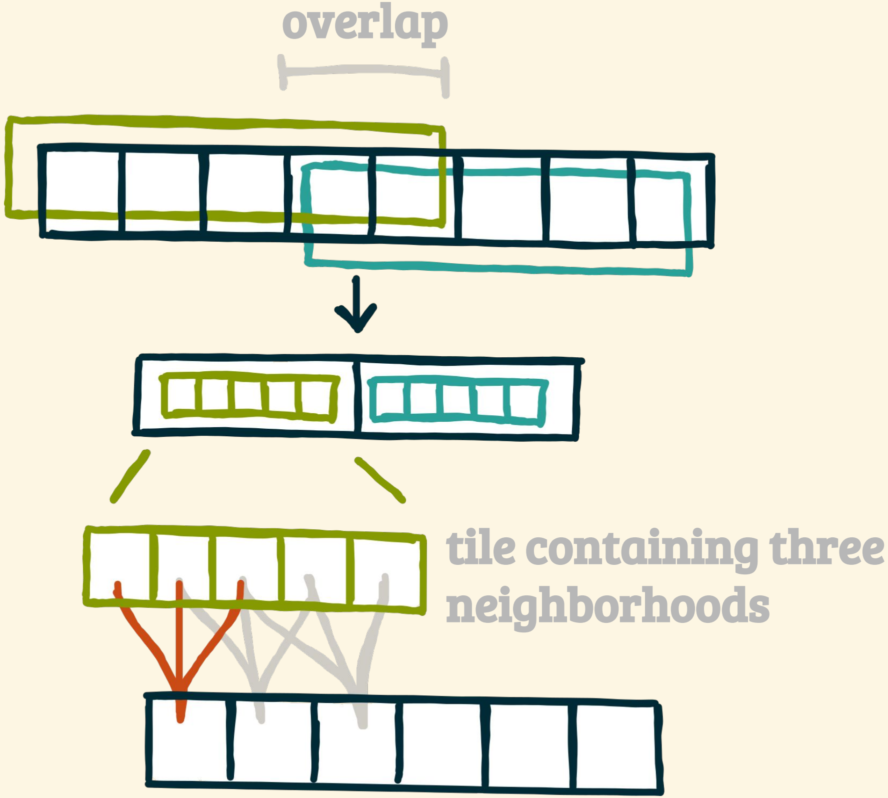
✓ **Exploit Locality**

Close neighborhoods share elements that can be grouped in tiles

✓ **Shared Memory**

Fast memory can be used to cache tiles

Optimization: Overlapped Tiling



✓ **Exploit Locality**

Close neighborhoods share elements that can be grouped in tiles

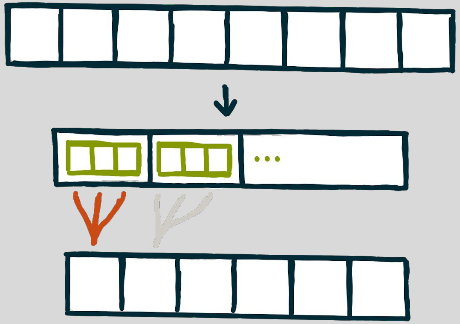
✓ **Shared Memory**

Fast memory can be used to cache tiles

Overlapped Tiling as a Rewrite Rule

overlapped tiling rule

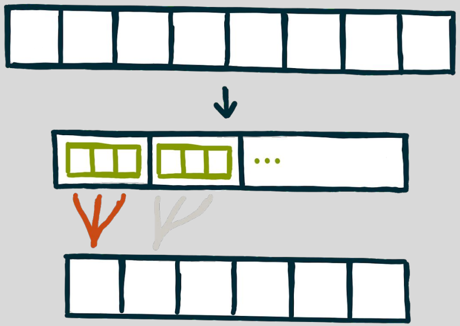
```
map(f, slide(3,1,input))
```

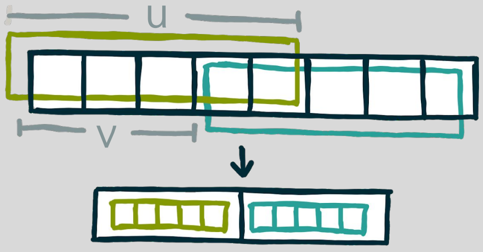


Overlapped Tiling as a Rewrite Rule

overlapped tiling rule

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

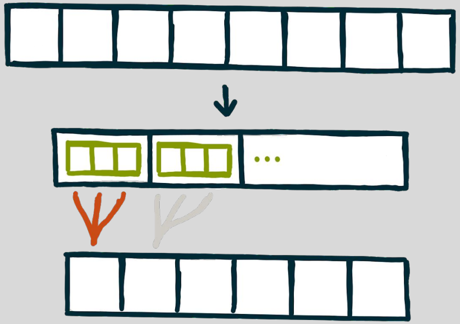


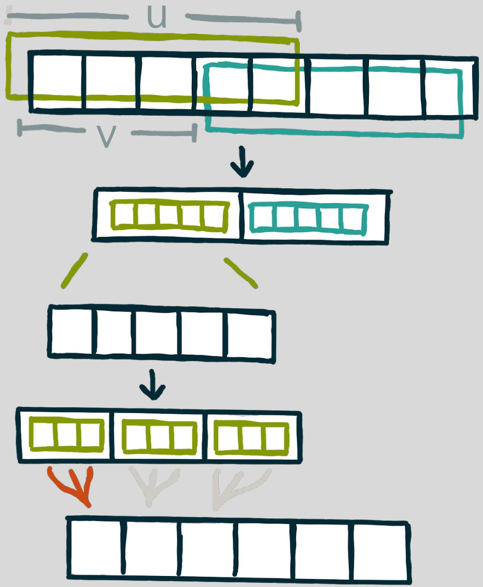


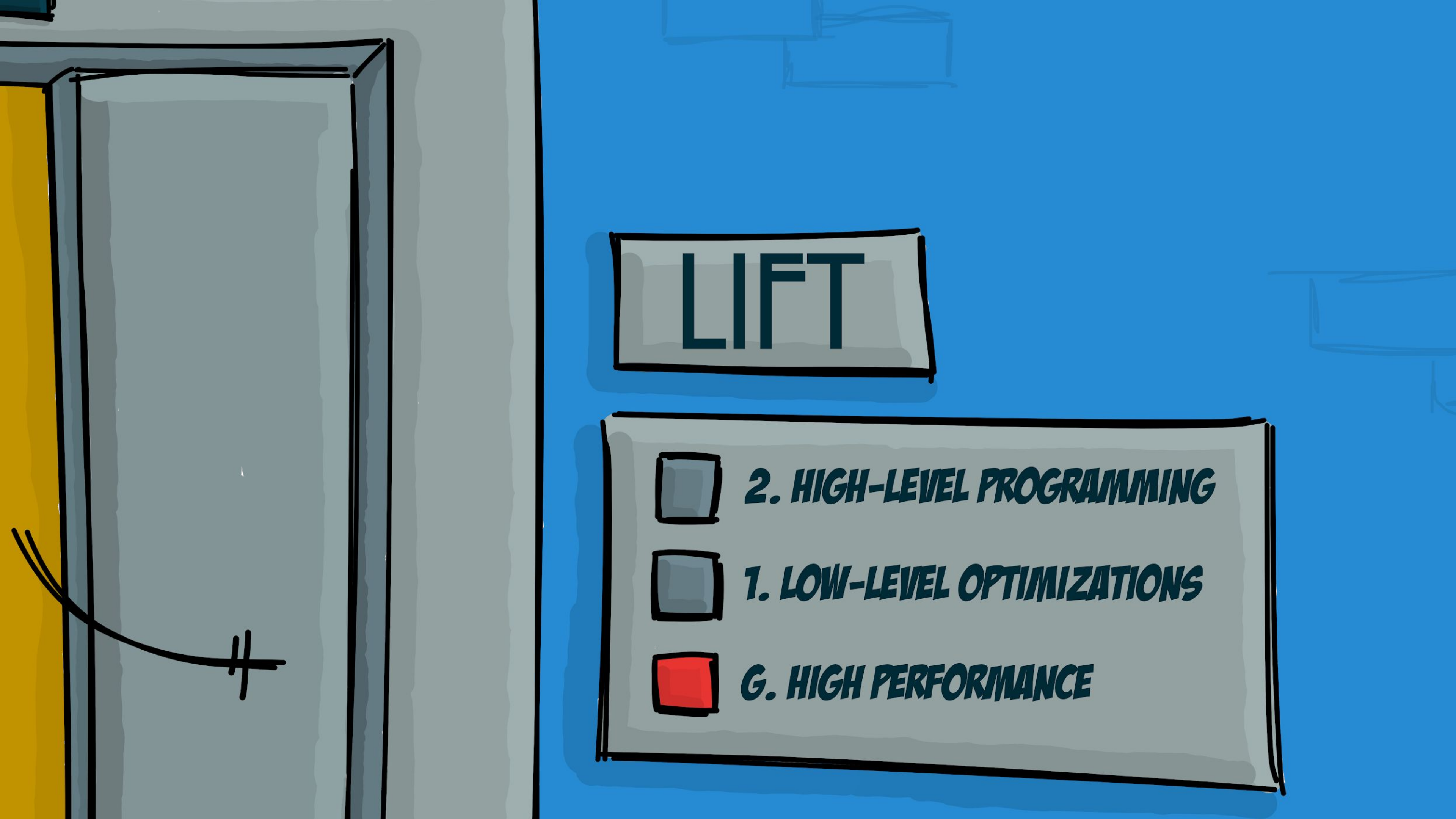
OVERLAPPED TILING AS A REWRITE RULE

overlapped tiling rule

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



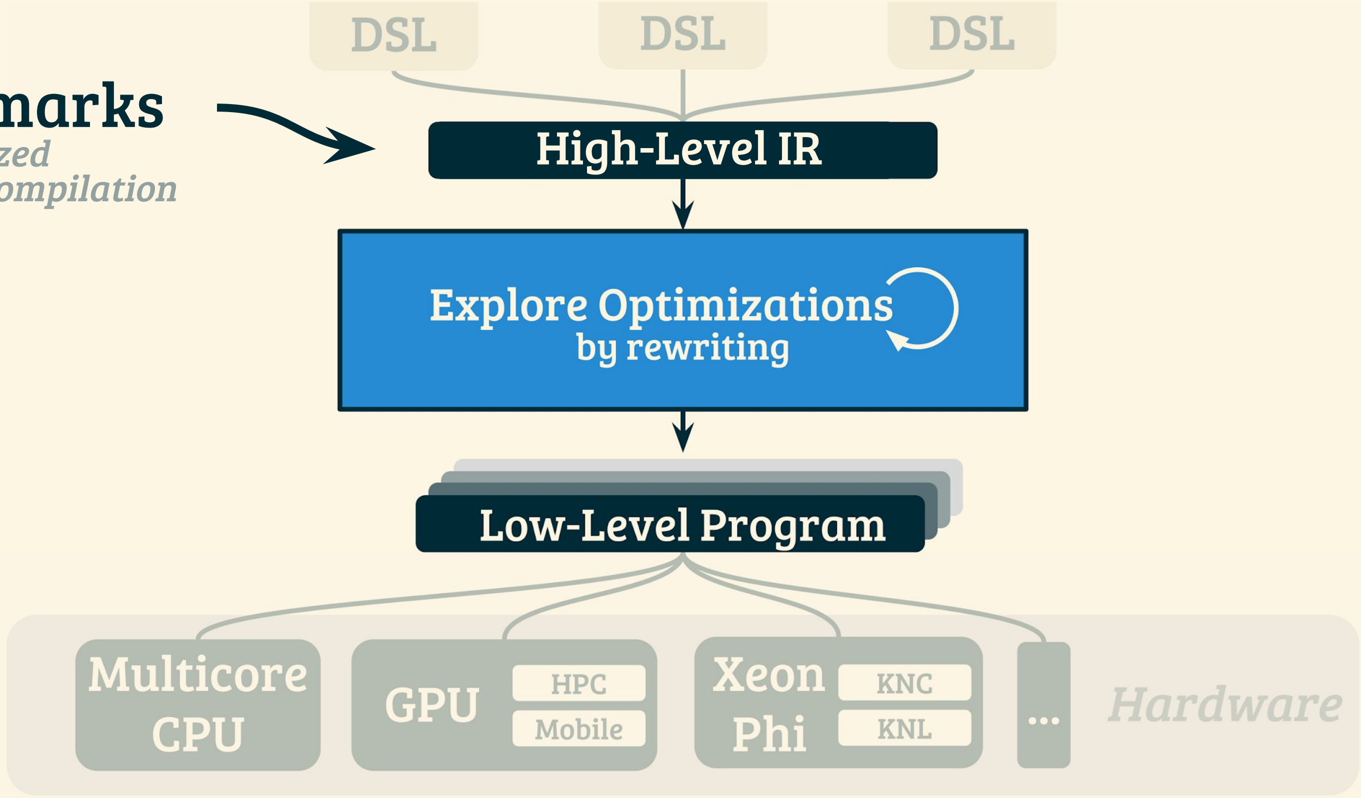



LIFT
2. HIGH-LEVEL PROGRAMMING
1. LOW-LEVEL OPTIMIZATIONS
G. HIGH PERFORMANCE

EXPERIMENTAL EVALUATION

14 Benchmarks

6 hand-optimized
8 polyhedral compilation



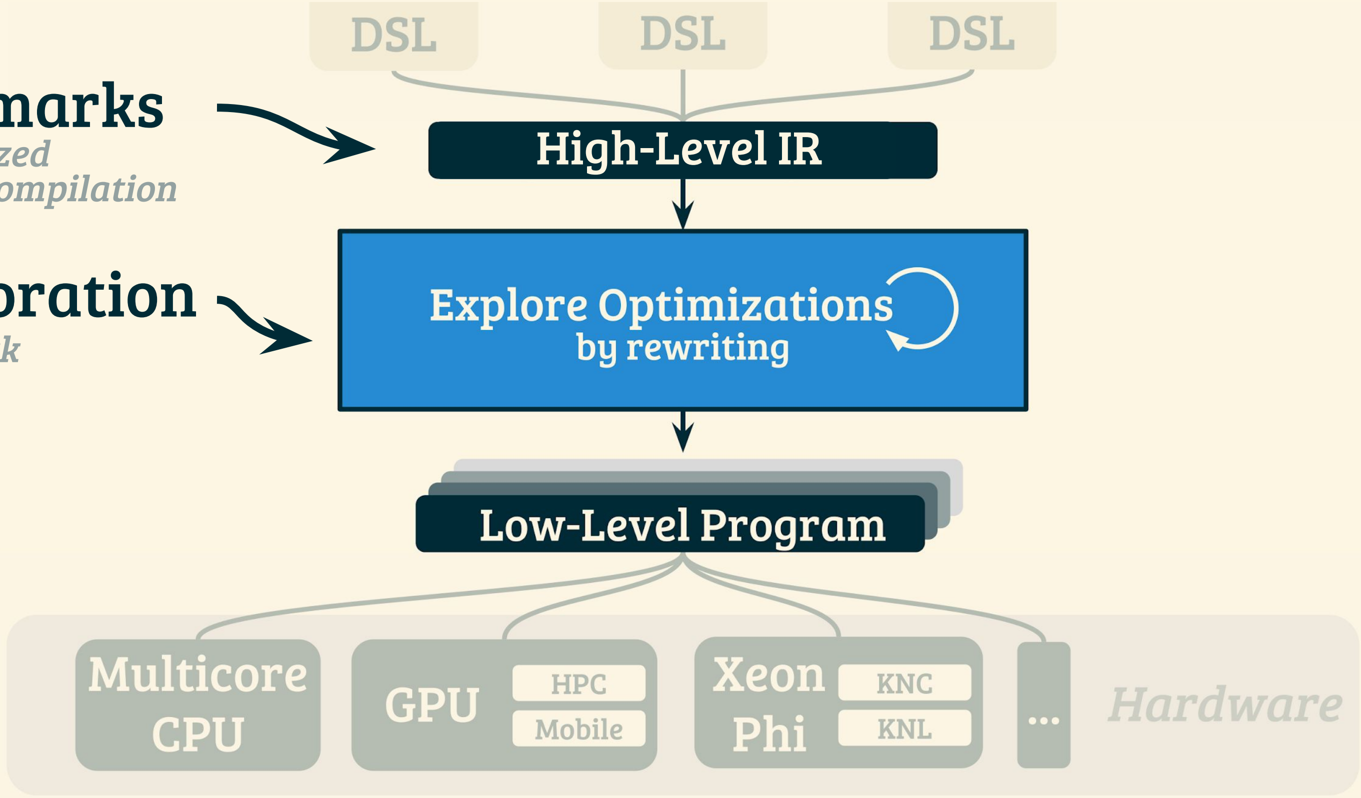
Experimental Evaluation

14 Benchmarks

6 hand-optimized
8 polyhedral compilation

< 3h Exploration

per benchmark



Experimental Evaluation

14 Benchmarks

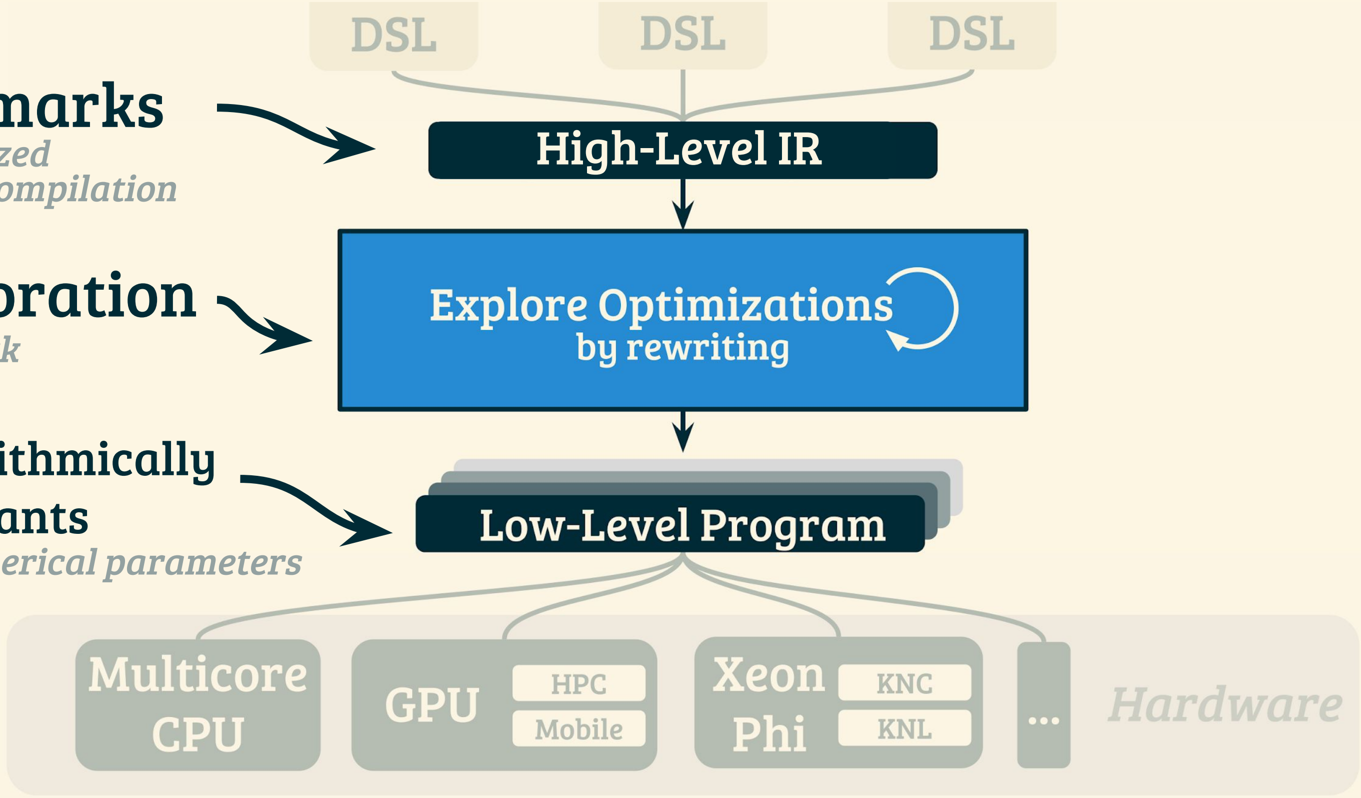
6 hand-optimized
8 polyhedral compilation

< 3h Exploration

per benchmark

up to 20 algorithmically different variants

+ auto-tuning of numerical parameters



Experimental Evaluation

14 Benchmarks

6 hand-optimized
8 polyhedral compilation

< 3h Exploration

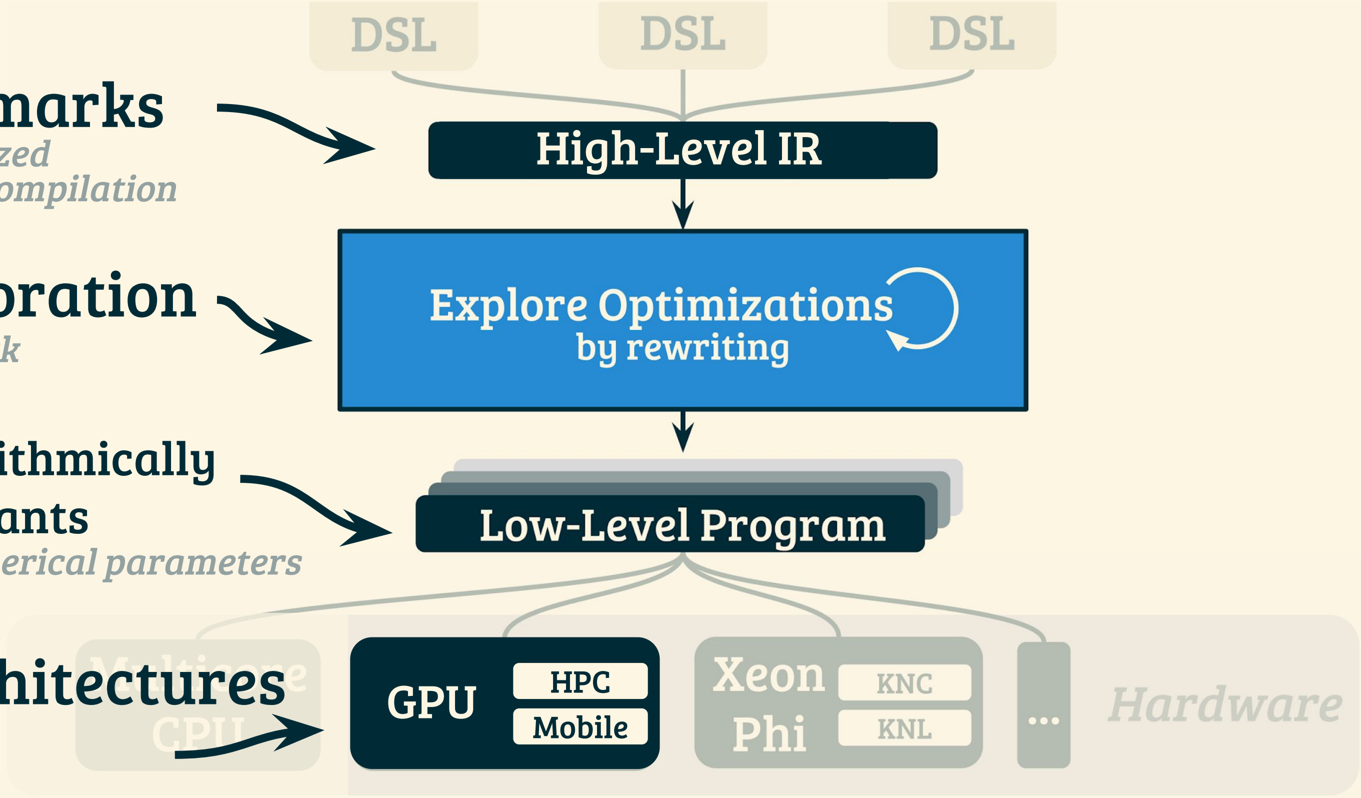
per benchmark

up to 20 algorithmically different variants

+ auto-tuning of numerical parameters

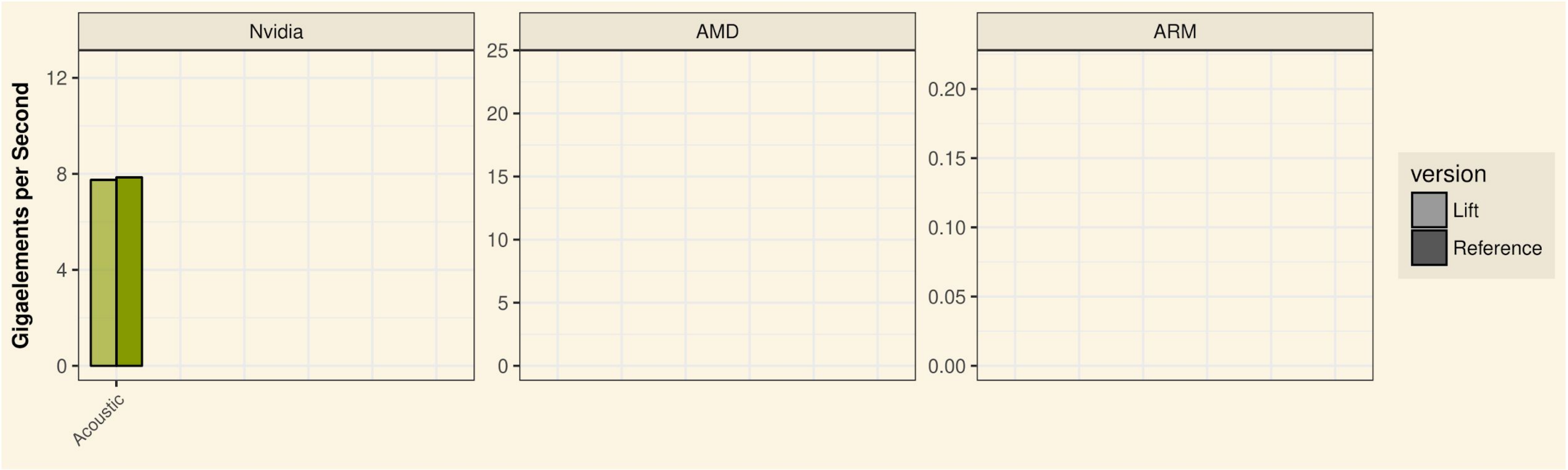
3 GPU Architectures

2 Desktop GPUs
1 Mobile GPU



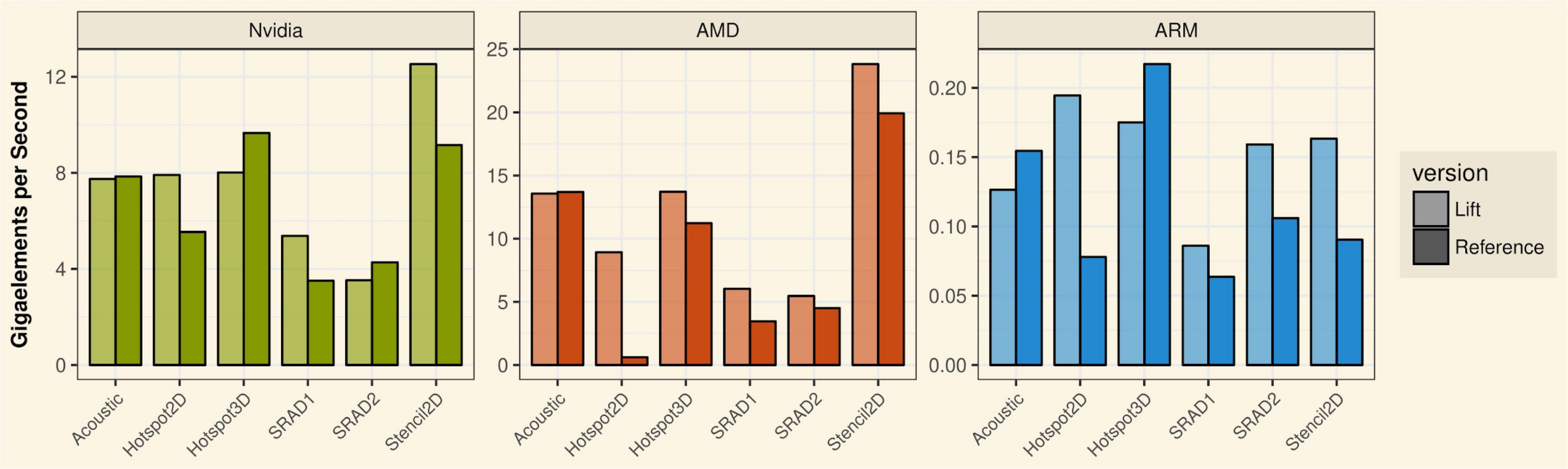
Comparison with hand-optimized codes

higher is better



Comparison with hand-optimized codes

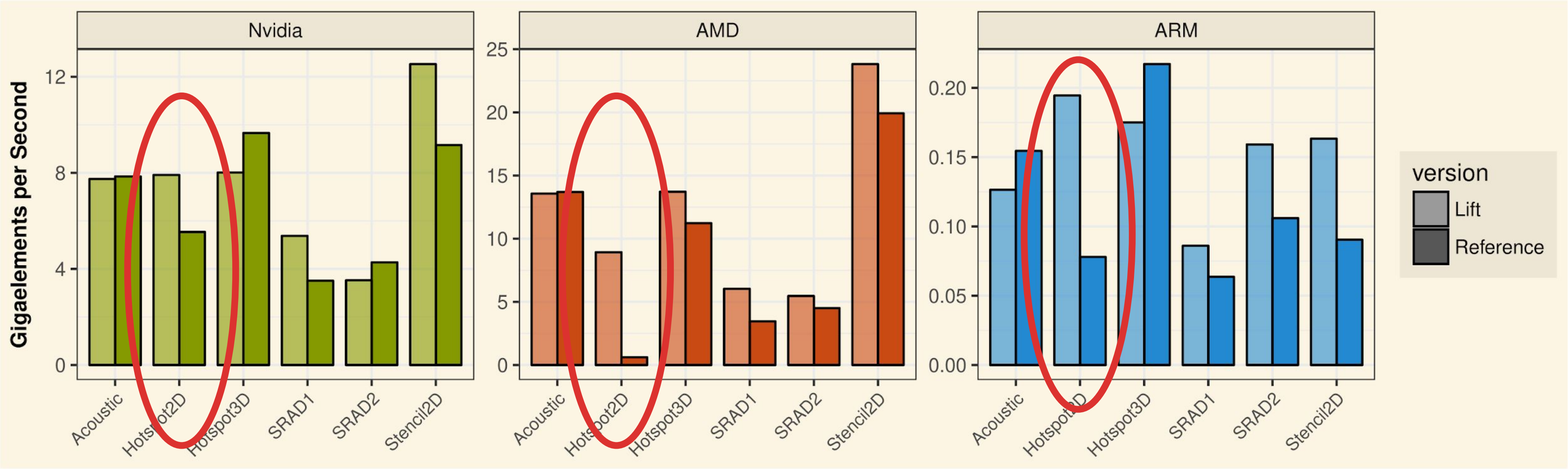
higher is better



Lift achieves the same performance
as hand optimized code

Comparison with hand-optimized codes

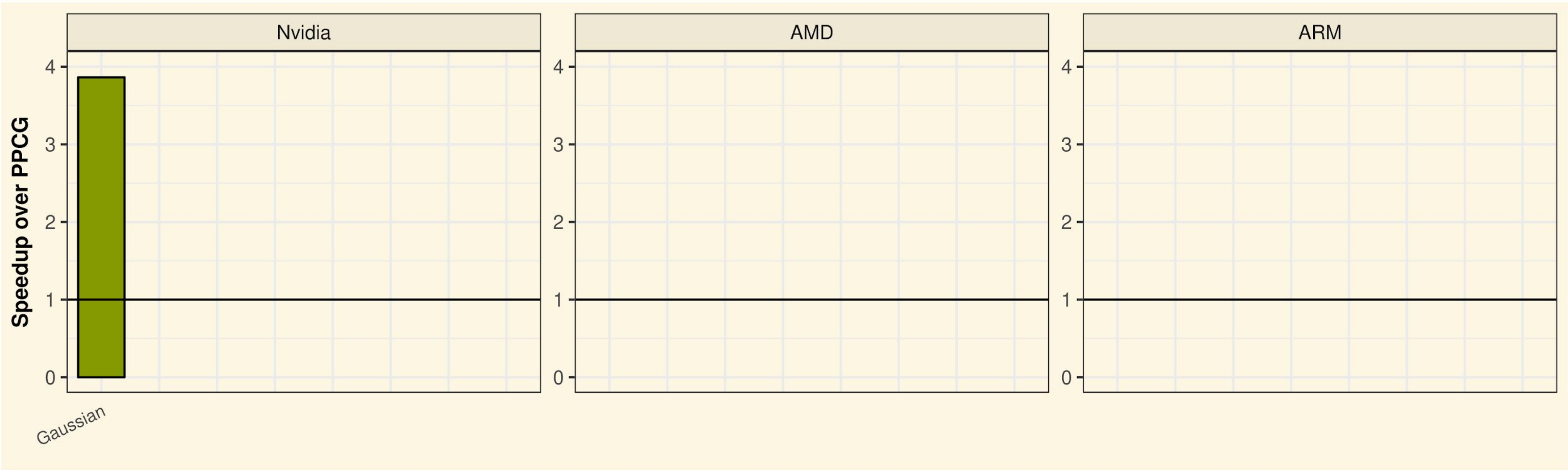
higher is better



Lift achieves the same performance
as hand optimized code

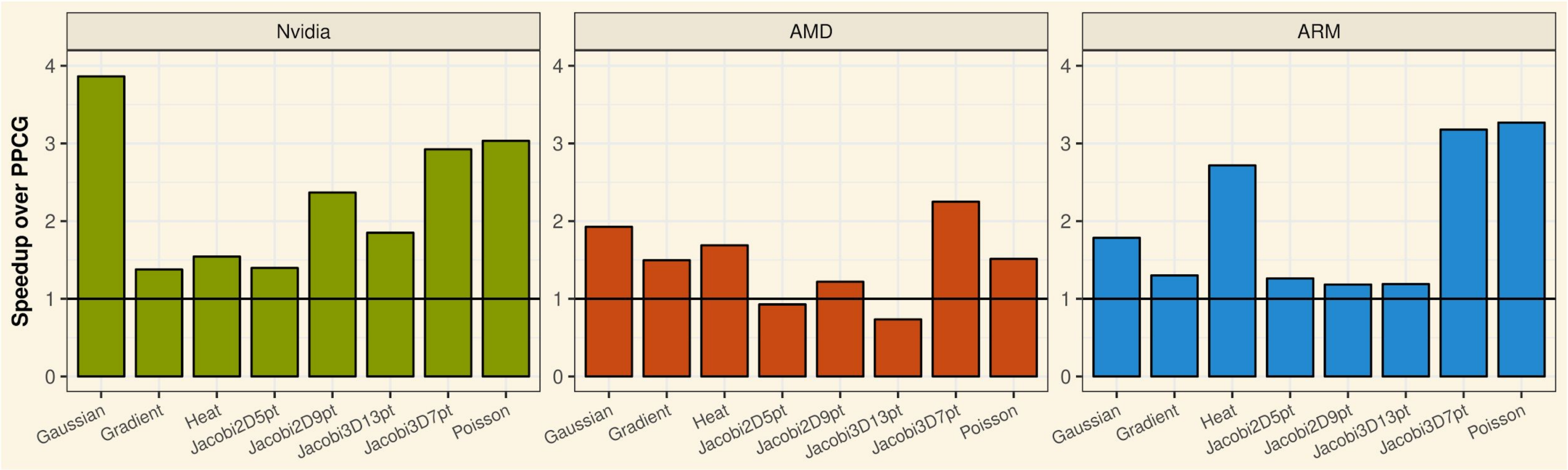
Comparison with Polyhedral Compilation

higher is better



Comparison with Polyhedral Compilation

higher is better



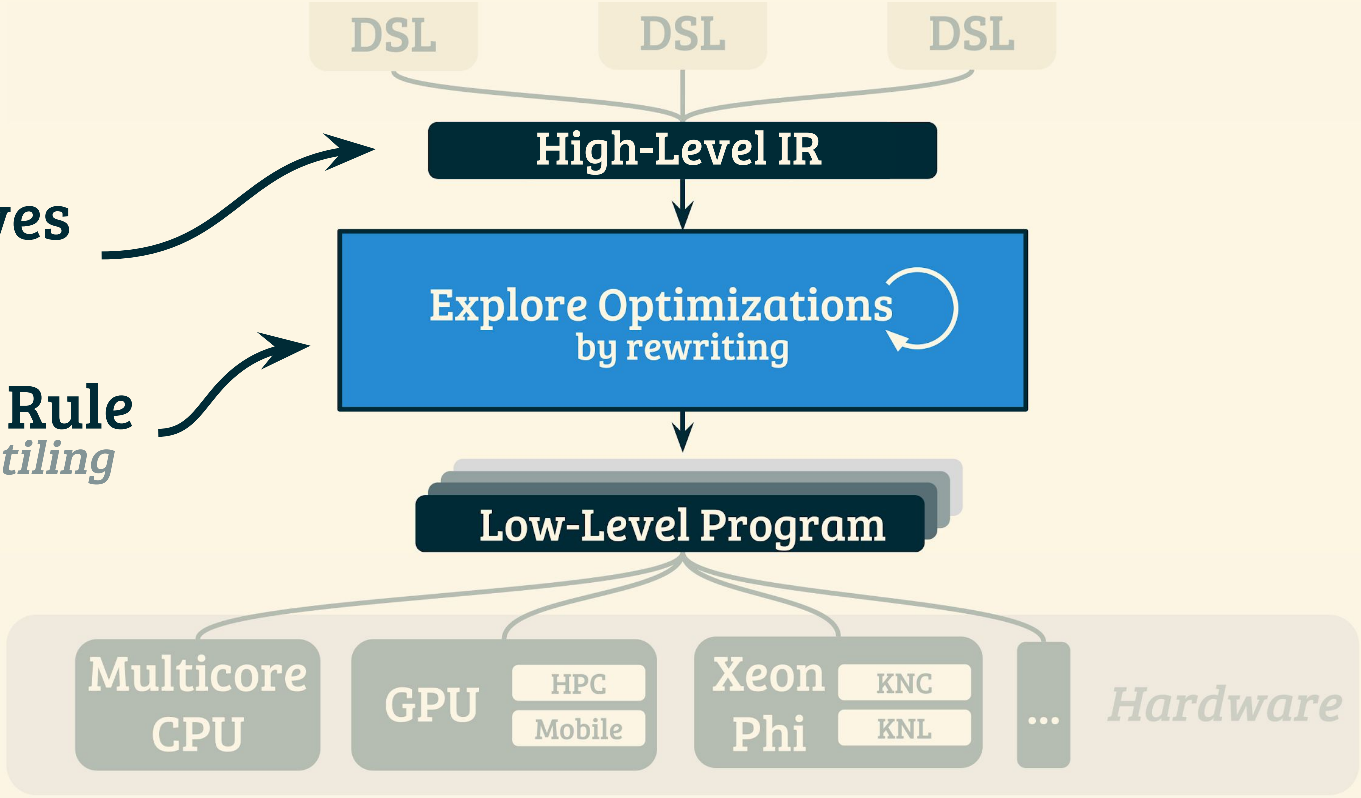
Lift outperforms state-of-the-art optimizing compilers

STENCIL COMPUTATIONS IN LIFT

We added:

2 Primitives
pad, *slide*

1 Rewrite Rule
overlapped tiling



LIFT IS OPEN SOURCE!

Artifacts Available acm · Artifacts Evaluated Reusable acm · Results Replicated acm

more info at:

lift-project.org

Paper · Artifacts Available acm CGO Artifact · Source Code

Best Paper Award (CGO'18)

Bastian Hagedorn: b.hagedorn@wwu.de

TO BE CONTINUED...