University of Glasgow

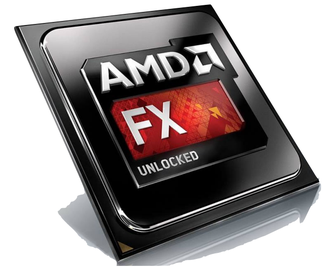
Lift & Elevate:
Generating High Performance Code with Rewrite Rules and Strategies

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

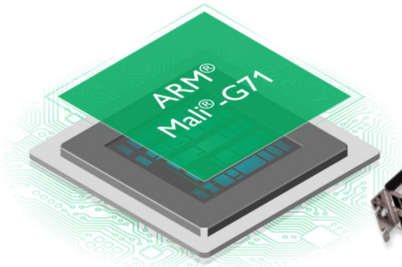
INSPIRING PEOPLE

Diversity is everywhere

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

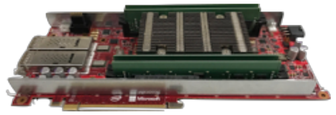


CPU

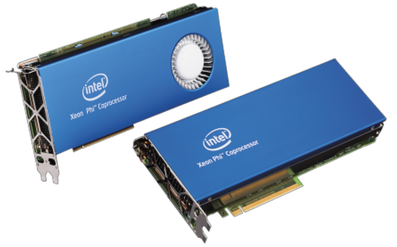


GPU

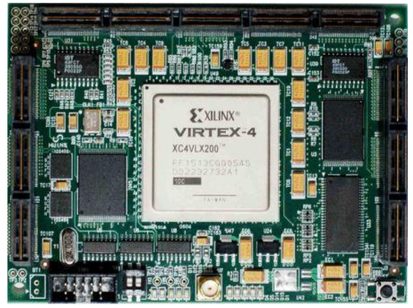
FPGA



Brainwave
Microsoft



Accelerator
Intel



Transmuter
Uni. of Michigan

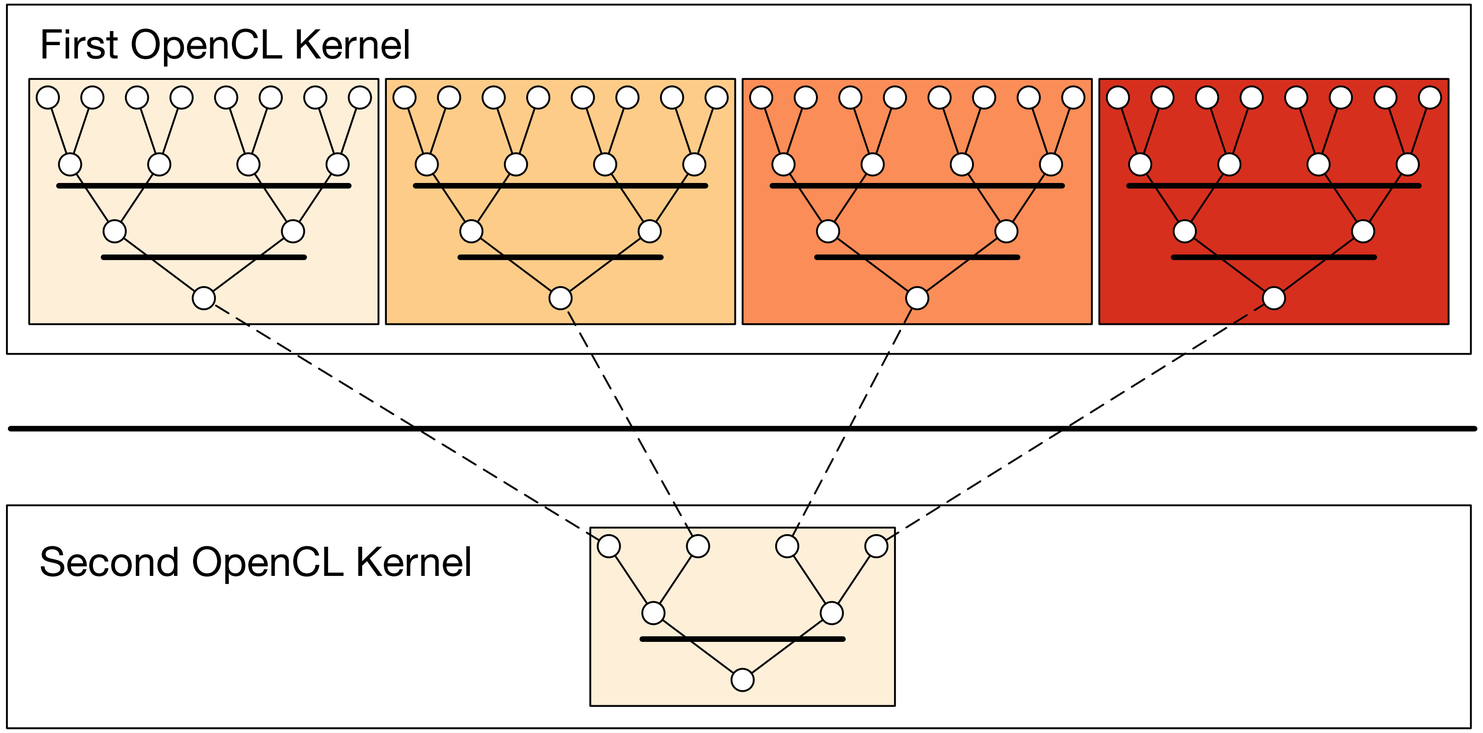
HPU
Microsoft

TPU
Google

...

Case Study: Parallel Reduction in OpenCL

- Summing up all values of an array
- Comparison of 7 implementations by Nvidia
- Investigating complexity and efficiency of optimisations



Parallel reduction with OpenCL

```
kernel void reduce(global float* g_idata, global float* g_odata,
                   unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
      barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

Parallel reduction with OpenCL

Kernel function executed in parallel by multiple **work-items**

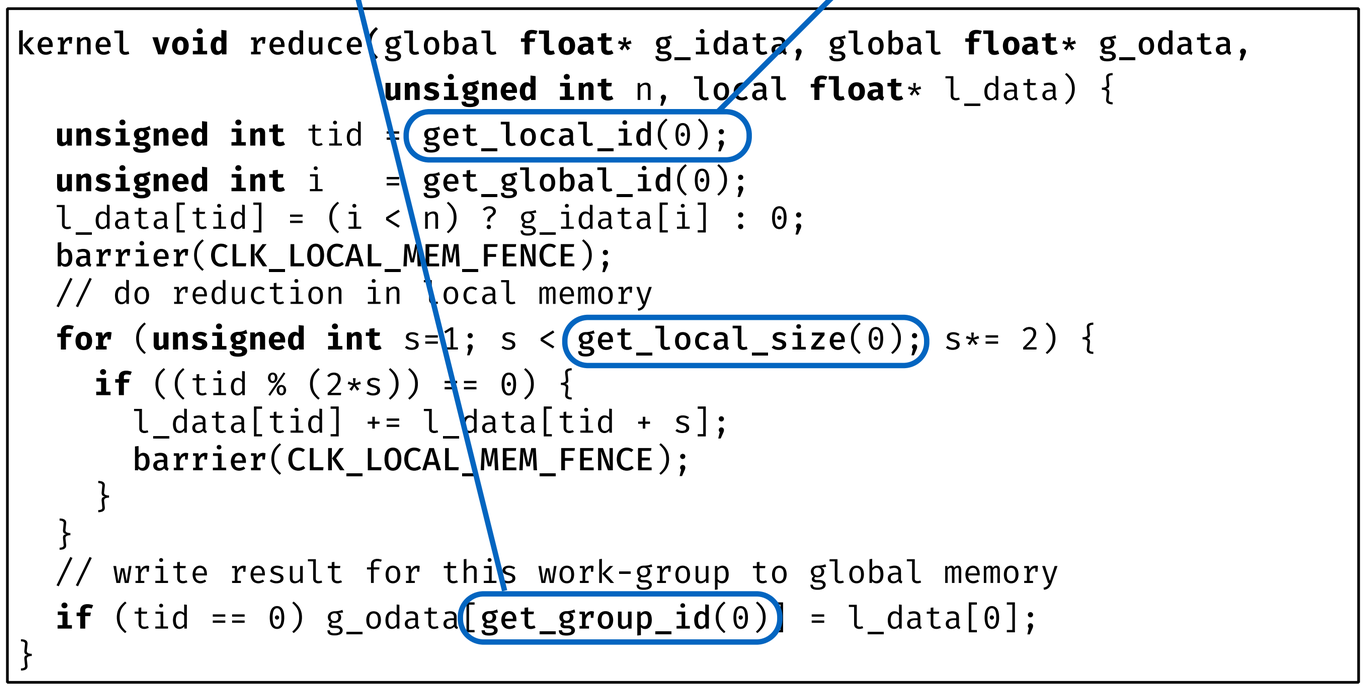
```
kernel void reduce(global float* g_idata, global float* g_odata,
                   unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
      barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

Work-items are identified by a unique **global id**

Parallel reduction with OpenCL

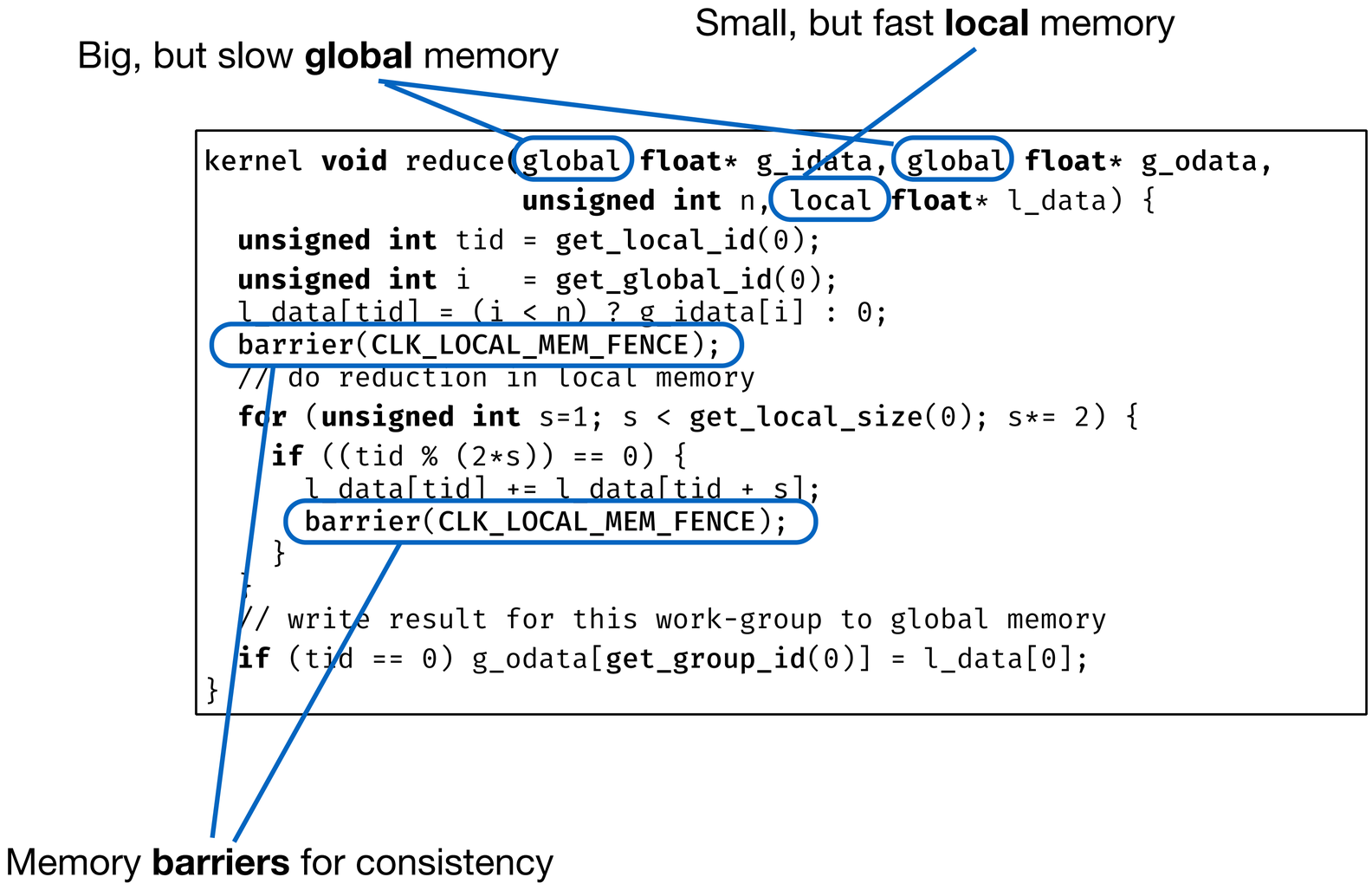
Work-items are grouped into **work-groups**

Local id within work-group



```
kernel void reduce(global float* g_idata, global float* g_odata,
                   unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
      barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

Parallel reduction with OpenCL




```
kernel void reduce(global float* g_idata, global float* g_odata,
                   unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
      barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

Parallel reduction with OpenCL

Potential **Deadlock**!

```
kernel void reduce(global float* g_idata, global float* g_odata,
                   unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

Functionally correct implementations in OpenCL are hard!

1. Version: Unoptimised Implementation Parallel Reduction

```
kernel void reduce0(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);
  // do reduction in local memory
  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
    barrier(CLK_LOCAL_MEM_FENCE);
    }
  }
  // write result for this work-group to global memory
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

2. Version: Avoid Divergent Branching

```
kernel void reduce1(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

  for (unsigned int s=1; s < get_local_size(0); s*= 2) {
    // continuous work-items remain active
    int index = 2 * s * tid;
    if (index < get_local_size(0)) {
      l_data[index] += l_data[index + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

3. Version: Avoid Interleaved Addressing

```
kernel void reduce2(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

  // process elements in different order
  // requires commutativity
  for (unsigned int s=get_local_size(0)/2; s>0; s>>=1) {
    if (tid < s) {
      l_data[tid] += l_data[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

4. Version: Increase Computational Intensity per Work-Item

```
kernel void reduce3(global float* g_idata, global float* g_odata,
                    unsigned int n, local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                   + get_local_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  // performs first addition during loading
  if (i + get_local_size(0) < n)
    l_data[tid] += g_idata[i+get_local_size(0)];
  barrier(CLK_LOCAL_MEM_FENCE);

  for (unsigned int s=get_local_size(0)/2; s>0; s>>=1) {
    if (tid < s) {
      l_data[tid] += l_data[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0];
}
```

5. Version: Avoid Synchronisation inside a Warp

```
kernel void reduce4(global float* g_idata, global float* g_odata,
                    unsigned int n, local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                   + get_local_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  if (i + get_local_size(0) < n)
    l_data[tid] += g_idata[i+get_local_size(0)];
  barrier(CLK_LOCAL_MEM_FENCE);

  # pragma unroll 1
  for (unsigned int s=get_local_size(0)/2; s>32; s>>=1) {
    if (tid < s) { l_data[tid] += l_data[tid + s]; }
    barrier(CLK_LOCAL_MEM_FENCE); }

  // this is not portable OpenCL code!
  if (tid < 32) {
    if (WG_SIZE >= 64) { l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) { l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) { l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) { l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) { l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) { l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

6. Version: Complete Loop Unrolling

```
kernel void reduce5(global float* g_idata, global float* g_odata,
                    unsigned int n, local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                   + get_local_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  if (i + get_local_size(0) < n)
    l_data[tid] += g_idata[i+get_local_size(0)];
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) { l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) { l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) { l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) { l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) { l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) { l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) { l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) { l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

7. Version: Fully Optimised Implementation

```
kernel void reduce6(global float* g_idata, global float* g_odata,
                    unsigned int n, local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i = get_group_id(0) * (get_local_size(0)*2)
                                   + get_local_id(0);
  unsigned int gridSize = WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) { l_data[tid] += g_idata[i];
                  if (i + WG_SIZE < n)
                    l_data[tid] += g_idata[i+WG_SIZE];
                  i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) { l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) { l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) { l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) { l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) { l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) { l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) { l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) { l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0) g_odata[get_group_id(0)] = l_data[0]; }
```

Reduction Case Study Conclusions

- Optimising OpenCL is complex
  - Understanding of target hardware required
- Program changes not obvious
- Is it worth it? …

```
kernel
void reduce0(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i   = get_global_id(0);
  l_data[tid] = (i < n) ? g_idata[i] : 0;
  barrier(CLK_LOCAL_MEM_FENCE);

  for (unsigned int s=1;
       s < get_local_size(0); s*= 2) {
    if ((tid % (2*s)) == 0) {
      l_data[tid] += l_data[tid + s];
    }
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

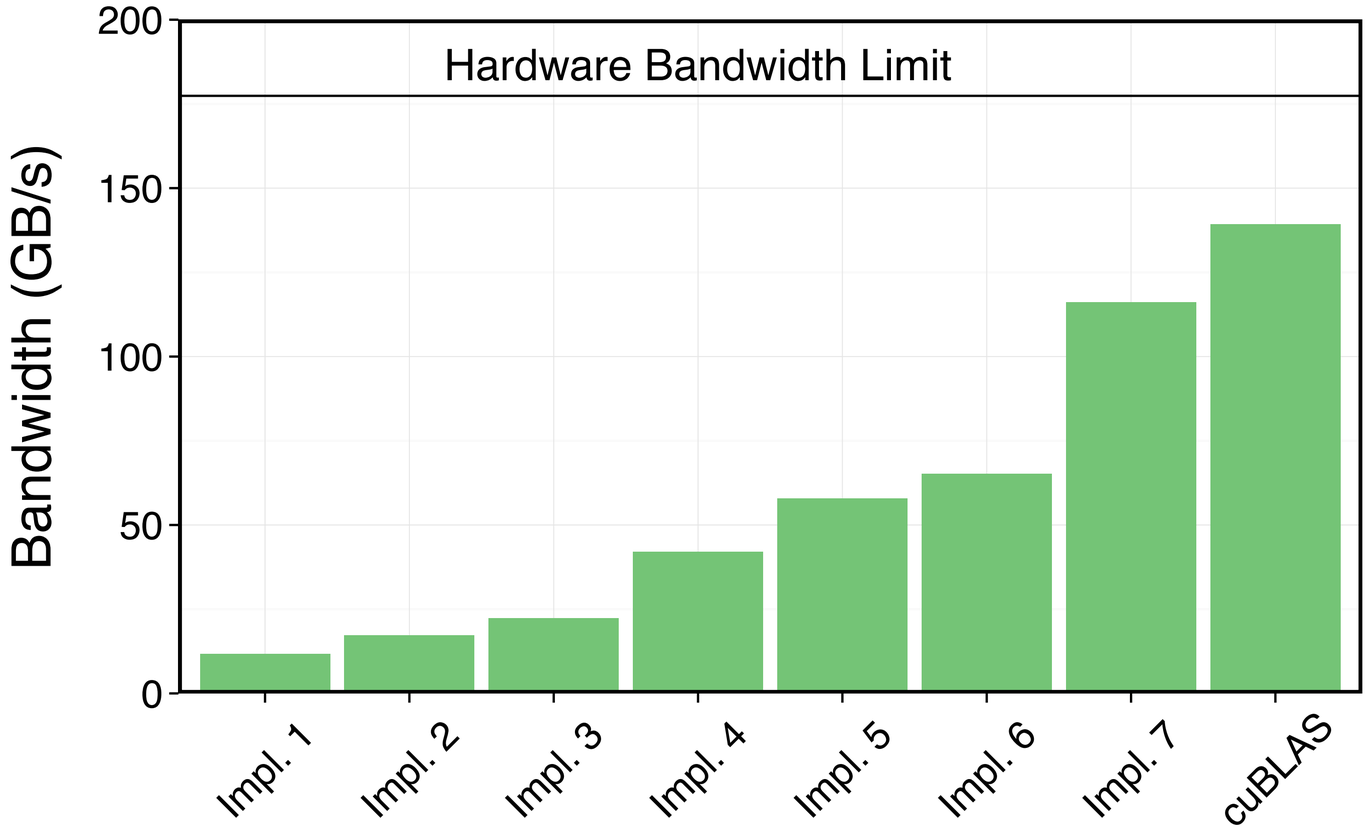
Unoptimized Implementation

```
kernel
void reduce6(global float* g_idata,
             global float* g_odata,
             unsigned int n,
             local volatile float* l_data) {
  unsigned int tid = get_local_id(0);
  unsigned int i =
    get_group_id(0) * (get_local_size(0)*2)
                    + get_local_id(0);
  unsigned int gridSize =
    WG_SIZE * get_num_groups(0);
  l_data[tid] = 0;
  while (i < n) {
    l_data[tid] += g_idata[i];
    if (i + WG_SIZE < n)
      l_data[tid] += g_idata[i+WG_SIZE];
    i += gridSize; }
  barrier(CLK_LOCAL_MEM_FENCE);

  if (WG_SIZE >= 256) {
    if (tid < 128) {
      l_data[tid] += l_data[tid+128]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (WG_SIZE >= 128) {
    if (tid <  64) {
      l_data[tid] += l_data[tid+ 64]; }
    barrier(CLK_LOCAL_MEM_FENCE); }
  if (tid < 32) {
    if (WG_SIZE >= 64) {
      l_data[tid] += l_data[tid+32]; }
    if (WG_SIZE >= 32) {
      l_data[tid] += l_data[tid+16]; }
    if (WG_SIZE >= 16) {
      l_data[tid] += l_data[tid+ 8]; }
    if (WG_SIZE >=  8) {
      l_data[tid] += l_data[tid+ 4]; }
    if (WG_SIZE >=  4) {
      l_data[tid] += l_data[tid+ 2]; }
    if (WG_SIZE >=  2) {
      l_data[tid] += l_data[tid+ 1]; } }
  if (tid == 0)
    g_odata[get_group_id(0)] = l_data[0];
}
```

Fully Optimized Implementation

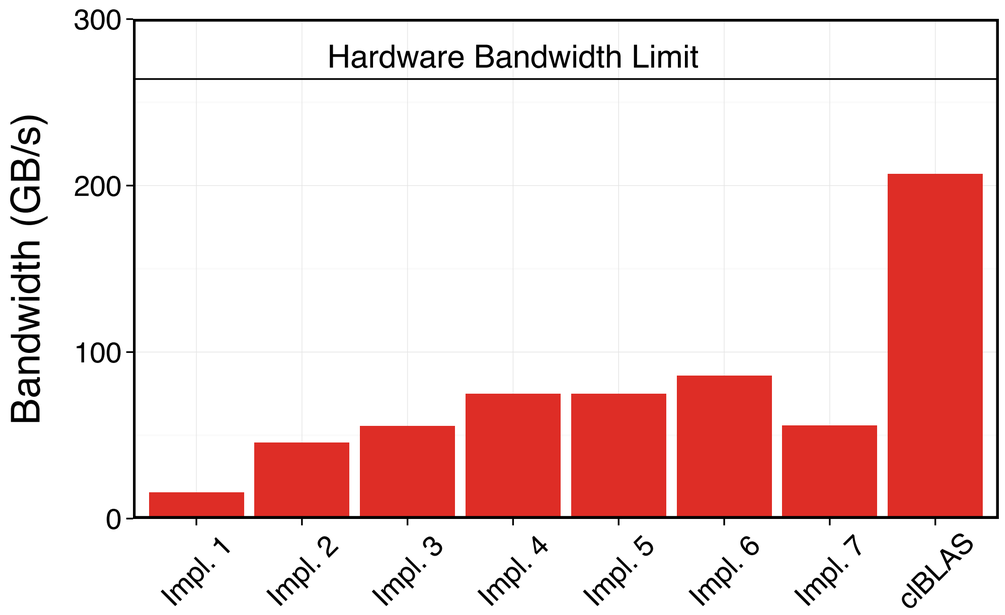
Performance Results Nvidia



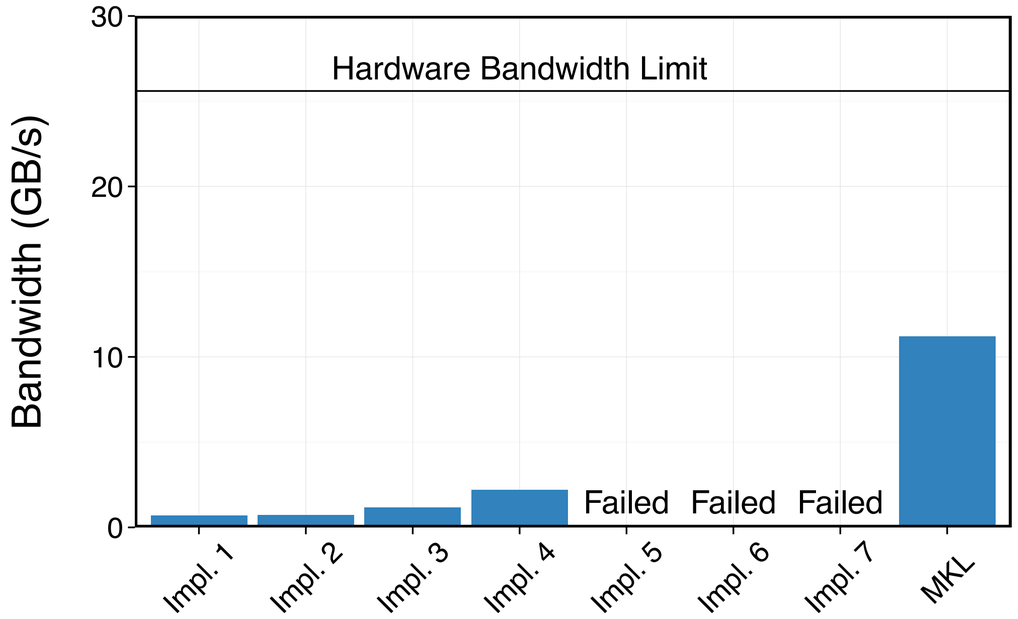
(a) Nvidia's GTX 480 GPU.

- … Yes! Optimising improves performance by a factor of **10**!
- Optimising is important, but …

Performance Results AMD and Intel

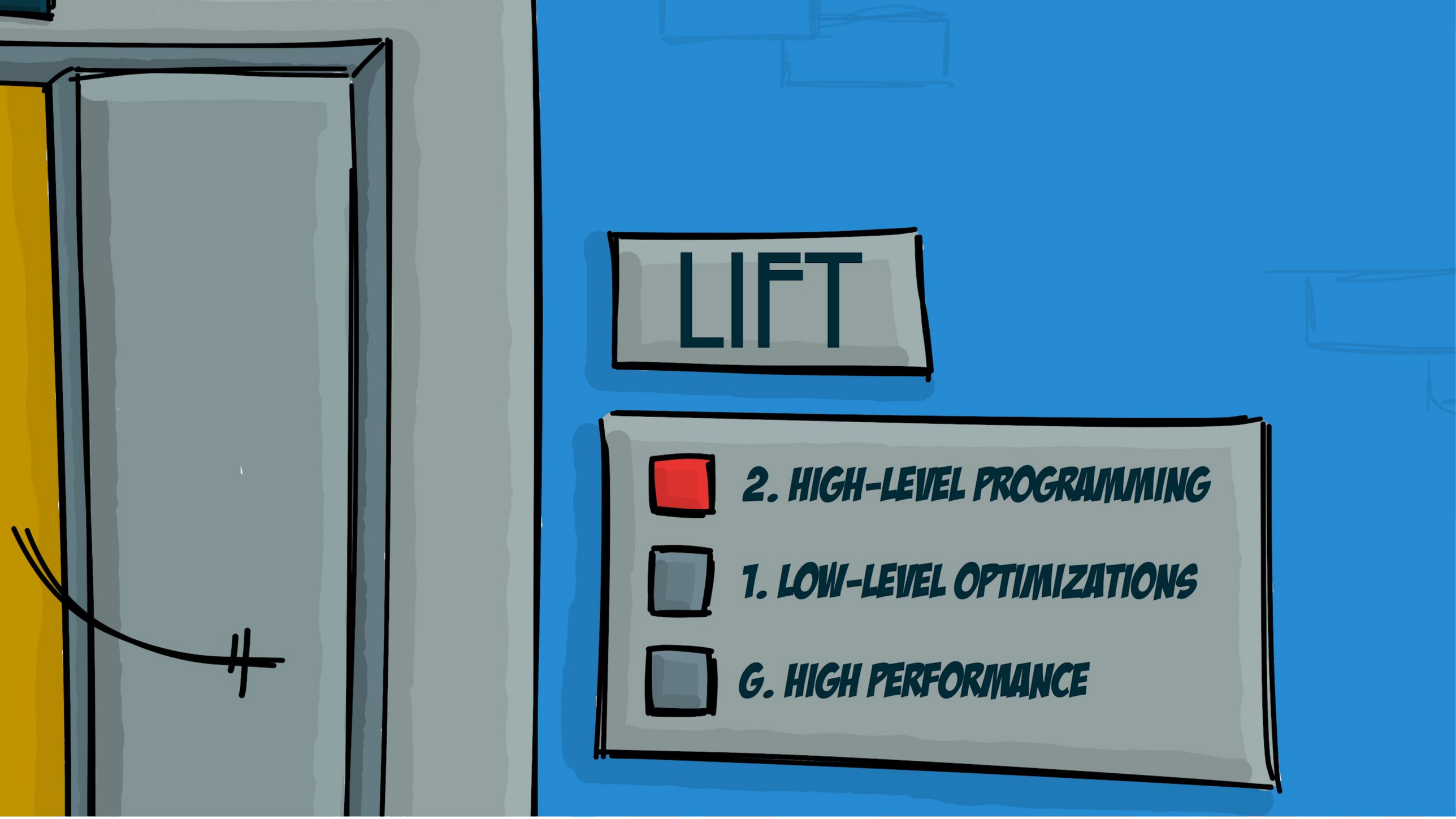


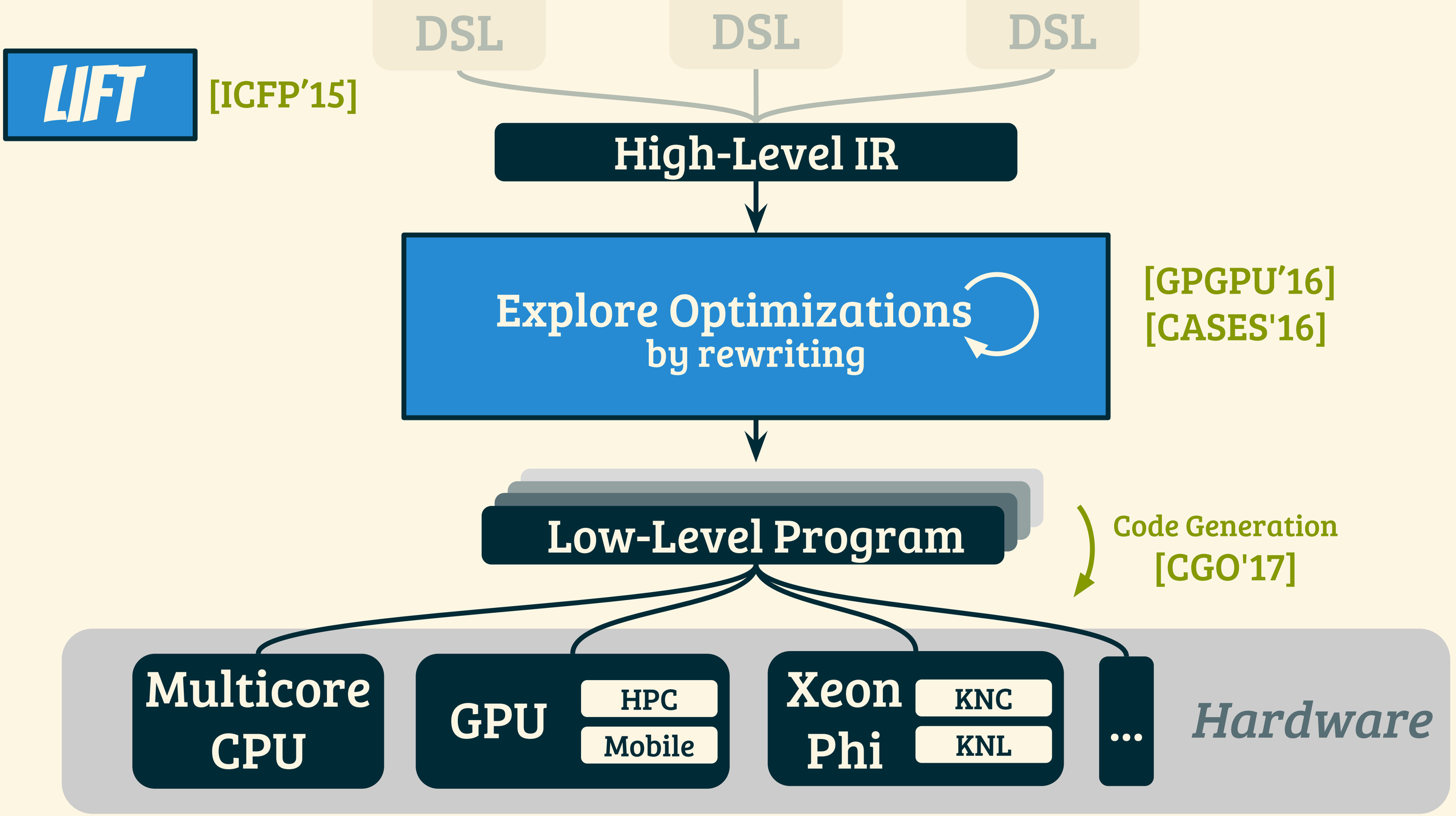
(b) AMD's HD 7970 GPU.



(c) Intel's E5530 dual-socket CPU.

- … unfortunately, optimisations in OpenCL are not portable!
- **Challenge**: how to achieving portable performance?

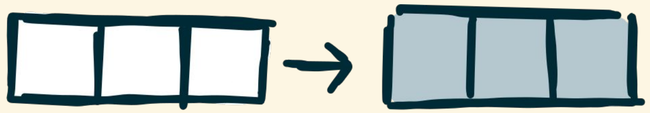

LIFT
2. HIGH-LEVEL PROGRAMMING
1. LOW-LEVEL OPTIMIZATIONS
G. HIGH PERFORMANCE



Collaboration with **Christophe Dubach** (University of Edinburgh) and the LIFT team

LIFT'S HIGH-LEVEL PRIMITIVES

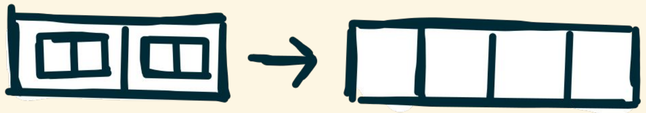
map(□→■)



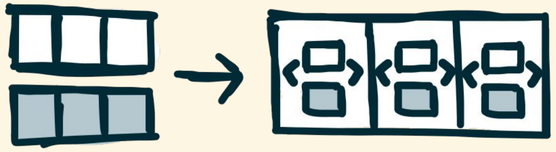
reduce(⊕)

split(n)

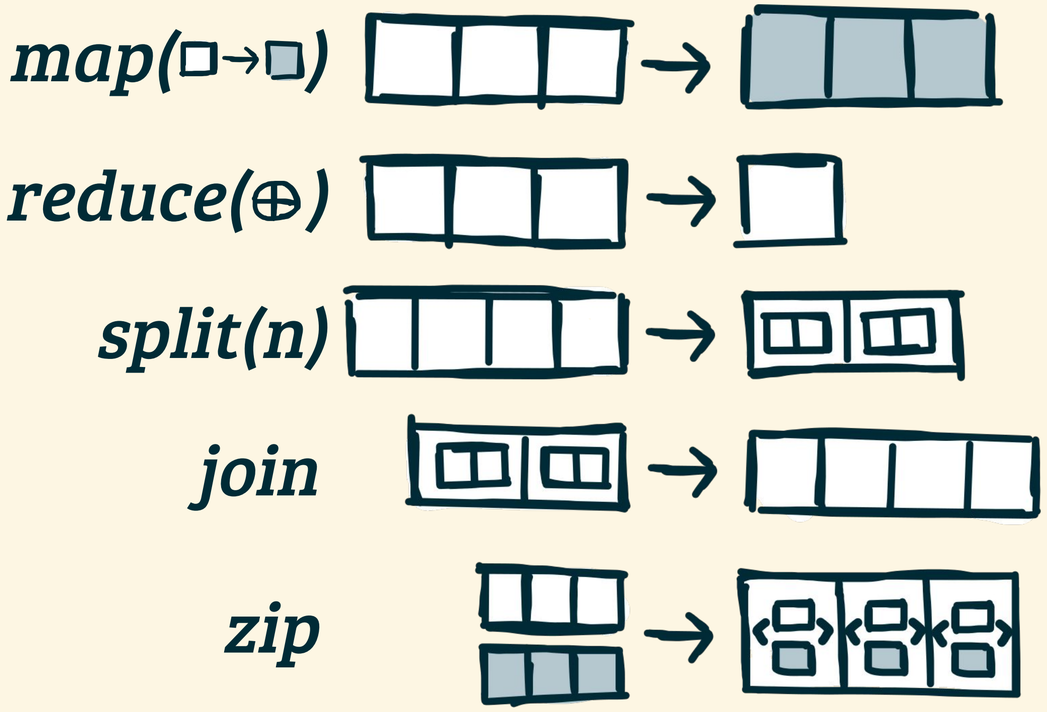
join



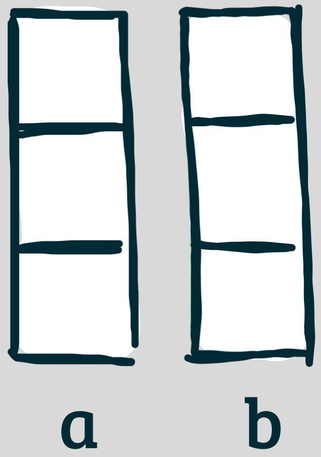
zip



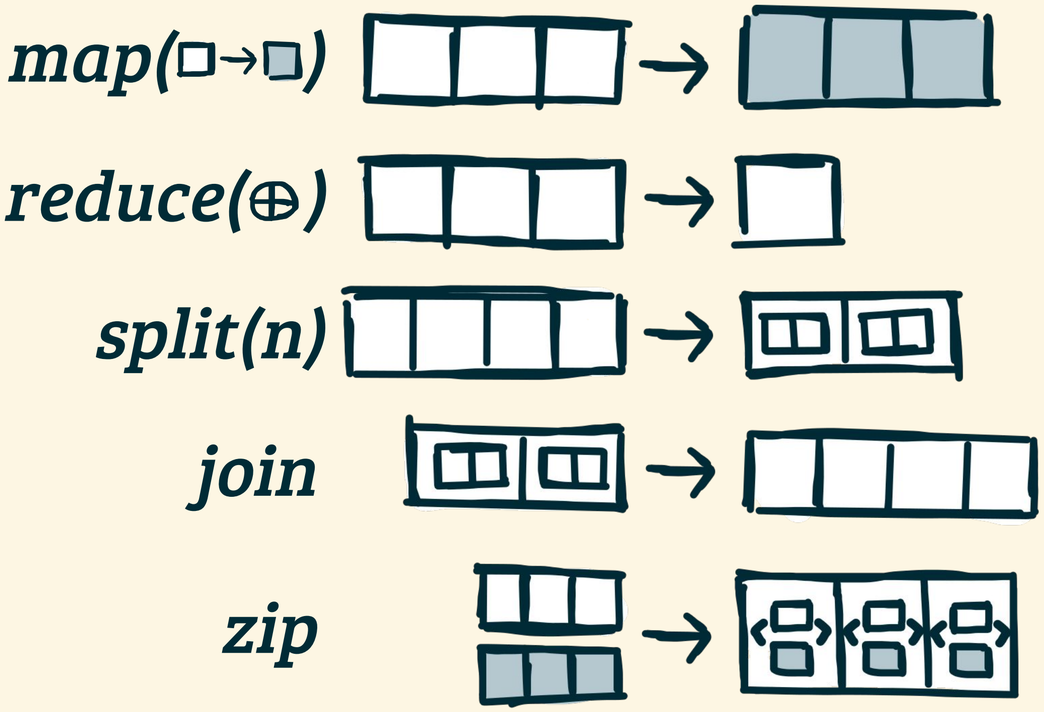
LIFT'S HIGH-LEVEL PRIMITIVES



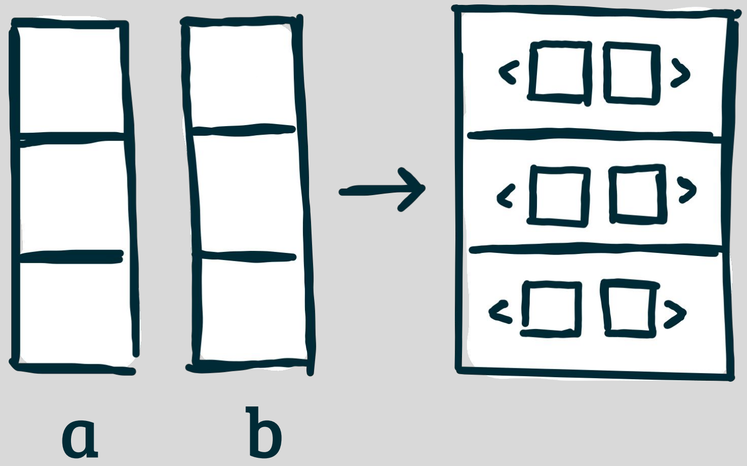
dotproduct.lift



LIFT'S HIGH-LEVEL PRIMITIVES

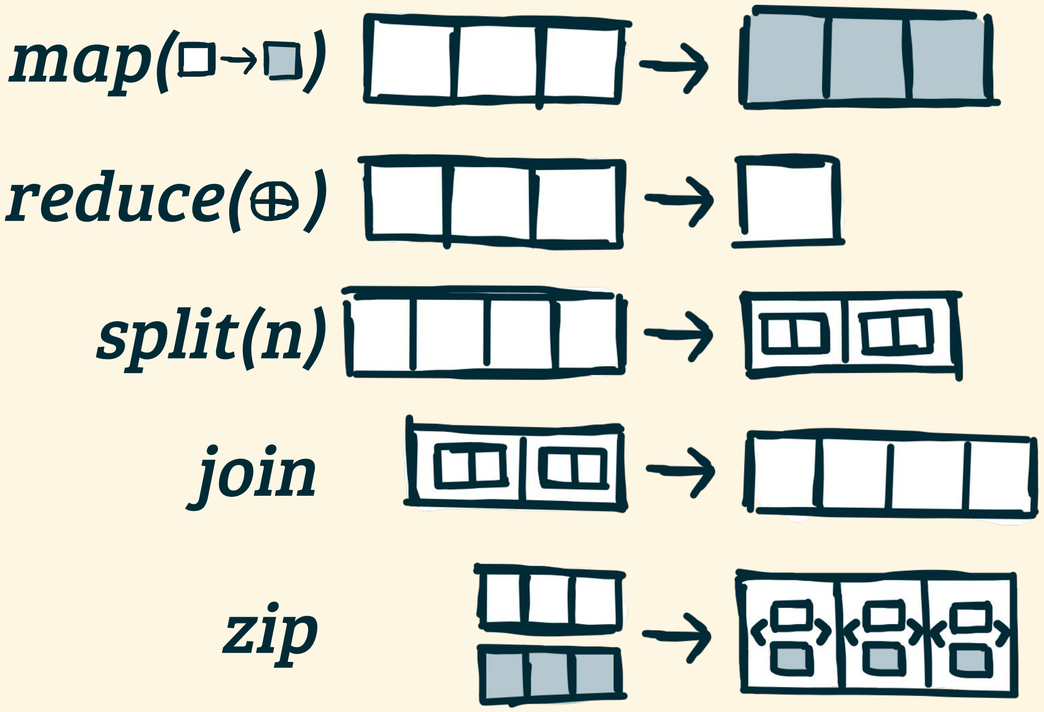


dotproduct.lift

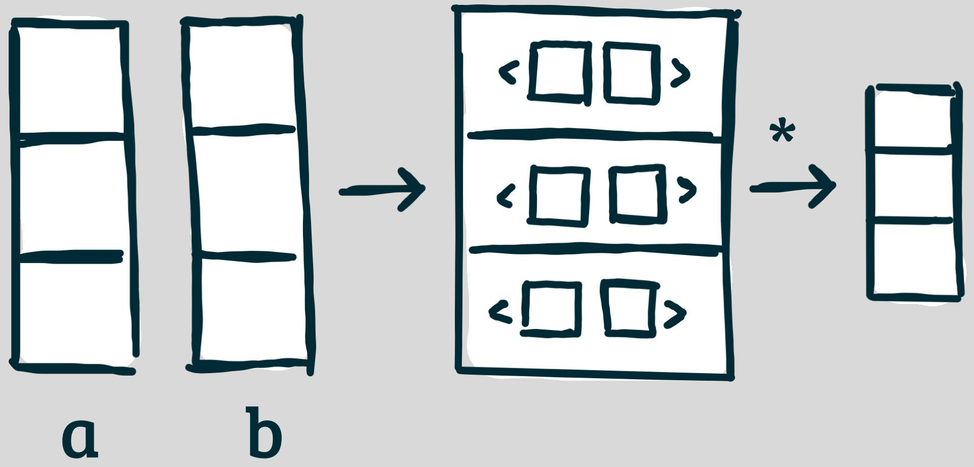



```
zip(a,b)
```

LIFT'S HIGH-LEVEL PRIMITIVES

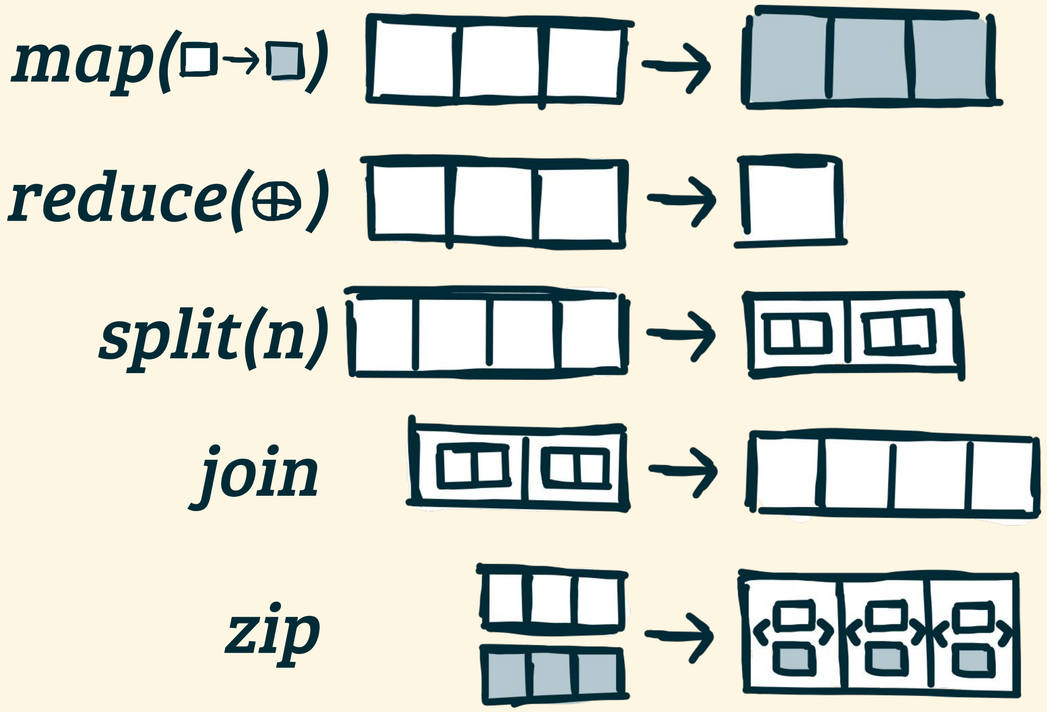


dotproduct.lift

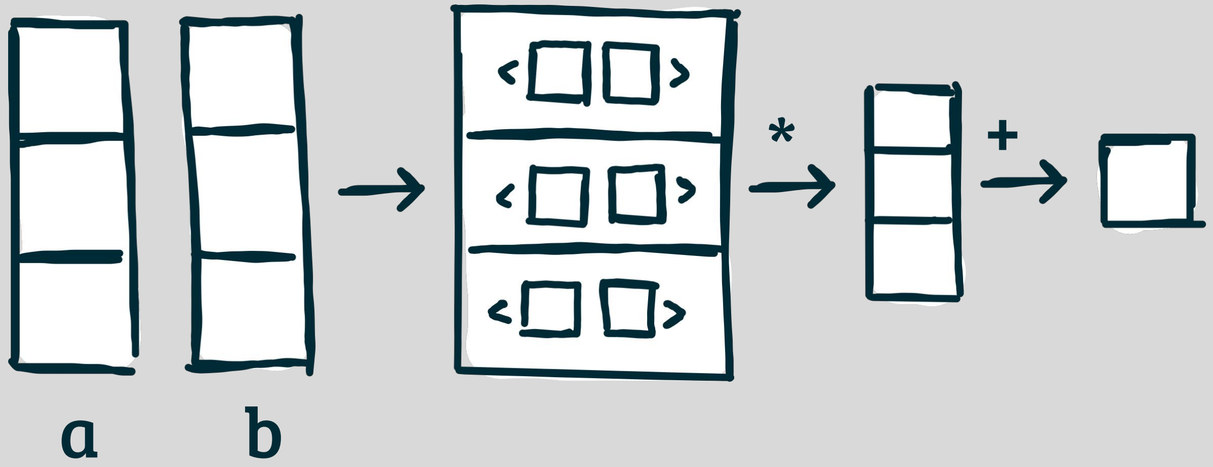



```
map(*, zip(a,b))
```

LIFT'S HIGH-LEVEL PRIMITIVES

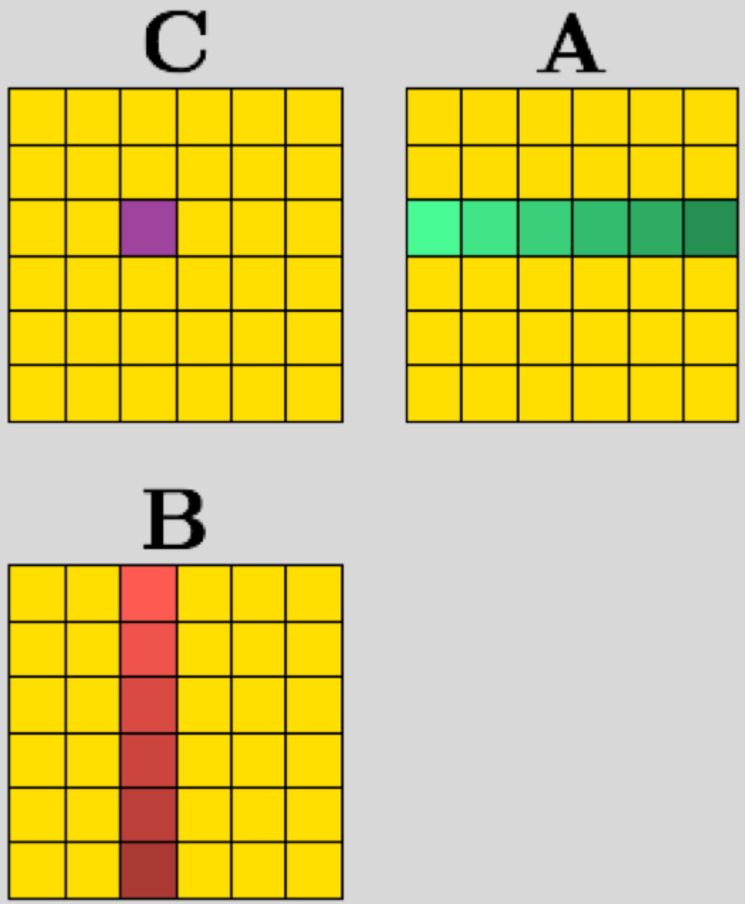


dotproduct.lift



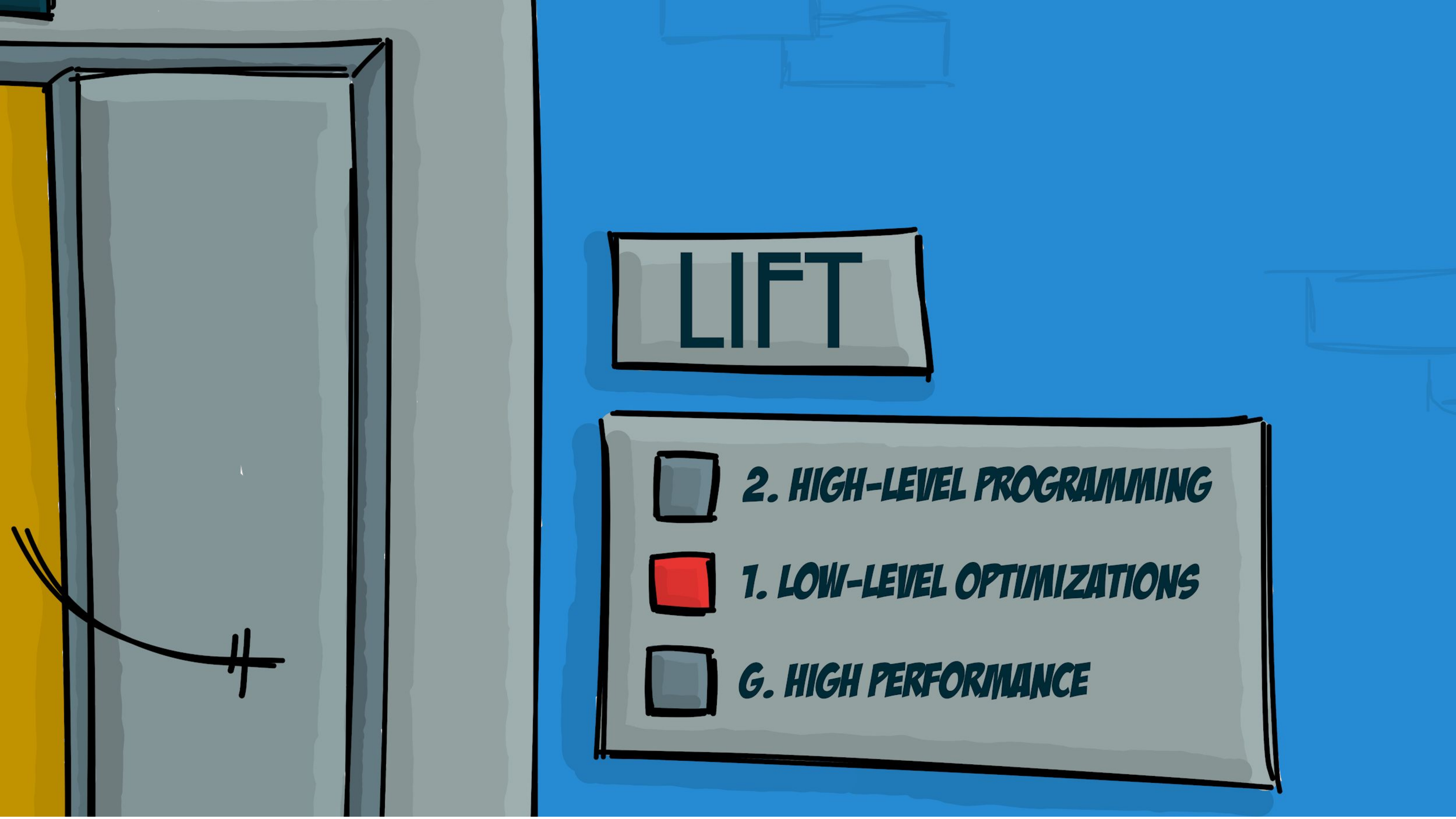

```
reduce(+,0, map(*, zip(a,b)))
```

LIFT'S HIGH-LEVEL PRIMITIVES



matrixMult.lift

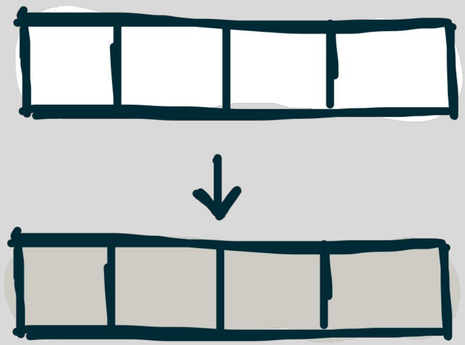
```
map(λ rowA ↦
    map(λ colB ↦
        dotProduct(rowA, colB)
    , transpose(B))
, A)
```


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

IMPLEMENTATION CHOICES AS REWRITE RULES

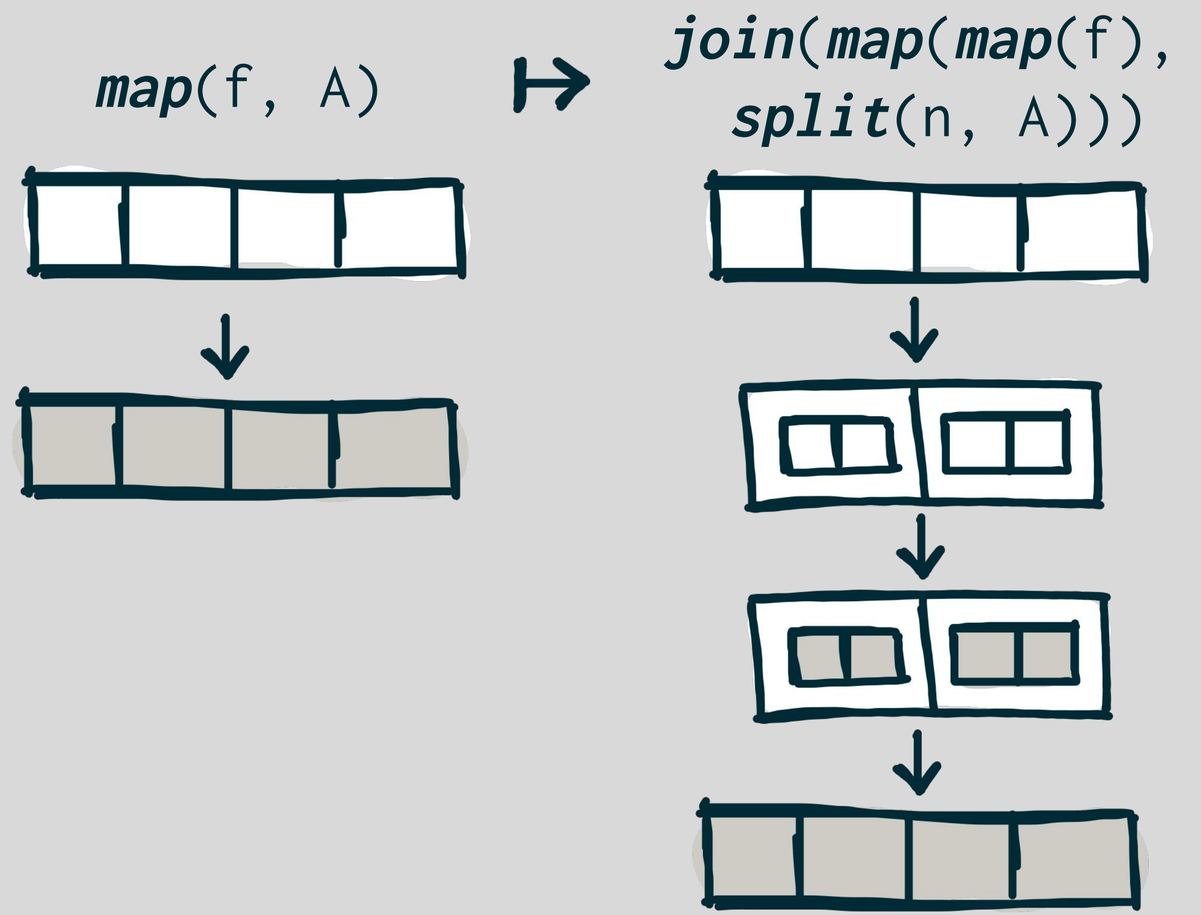
Divide & Conquer

map(f, A)



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, 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 VIA REWRITE RULES

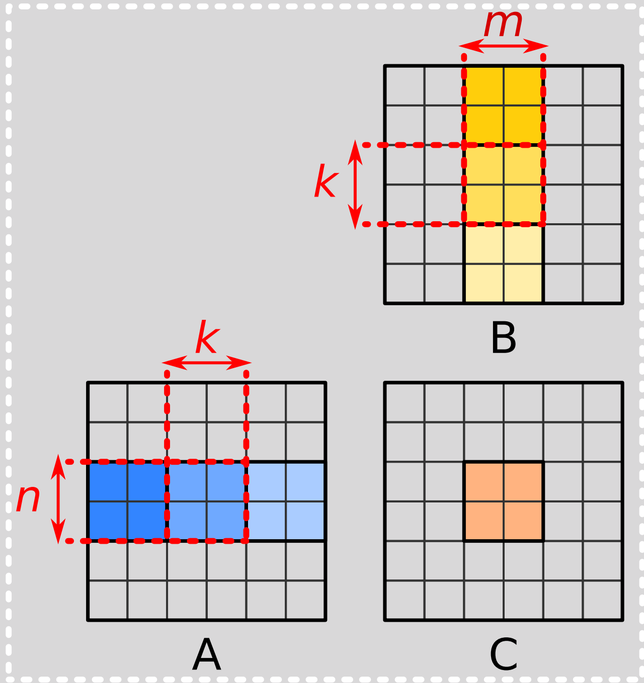
2D Tiling

Naïve matrix multiplication

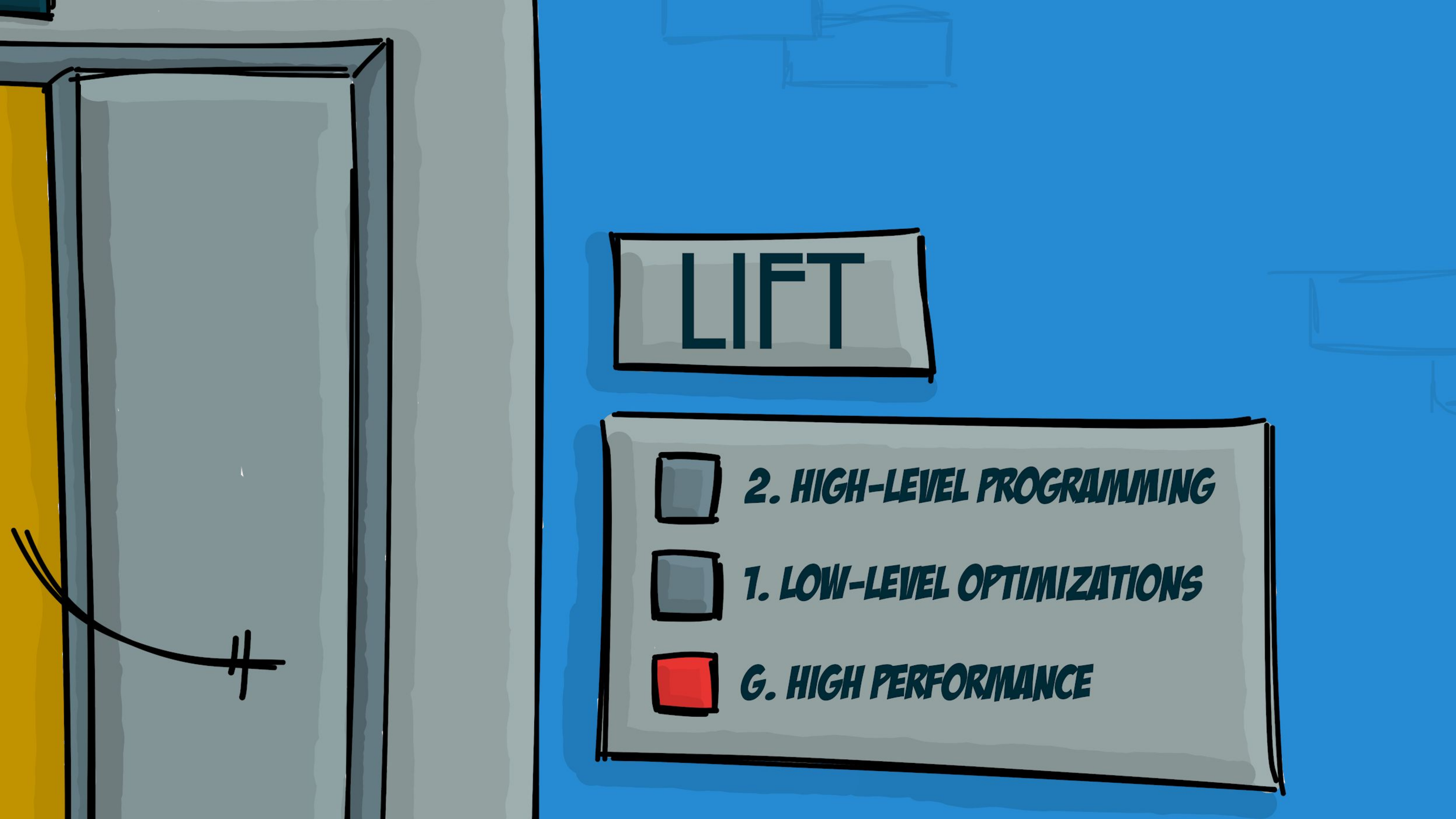
```
1  map(λ arow .
2    map(λ bcol .
3      reduce(+, 0) ∘ map(×) ∘ zip(arow, bcol)
4    , transpose(B))
5  , A)
```

↓ Apply tiling rules

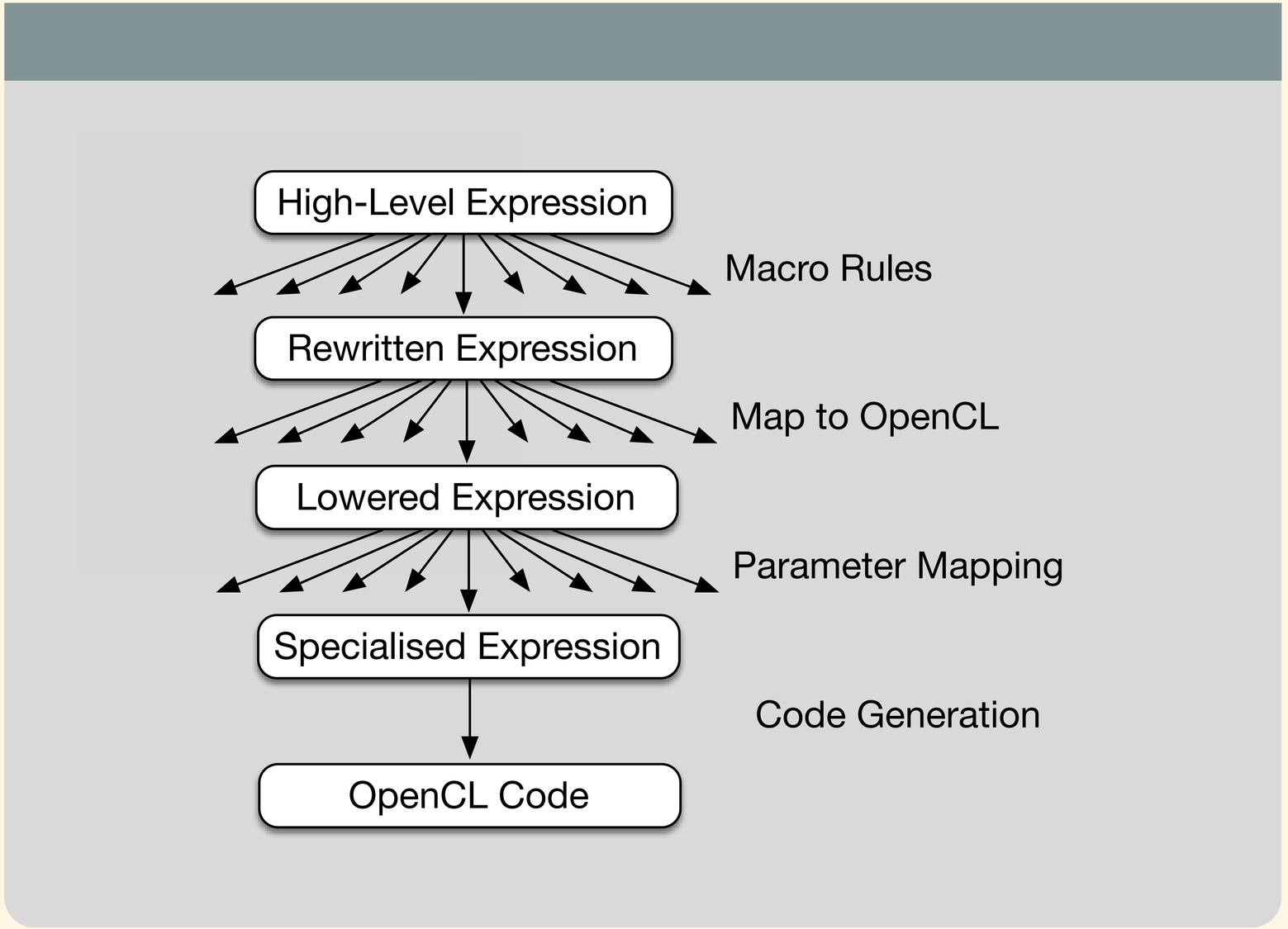
```
1  untile ∘ map(λ rowOfTilesA .
2    map(λ colOfTilesB .
3      toGlobal(copy2D) ∘
4      reduce(λ (tileAcc, (tileA, tileB)) .
5        map(map(+)) ∘ zip(tileAcc) ∘
6        map(λ as .
7          map(λ bs .
8            reduce(+, 0) ∘ map(×) ∘ zip(as, bs)
9          , toLocal(copy2D(tileB)))
10       , toLocal(copy2D(tileA)))
11     ,0, zip(rowOfTilesA, colOfTilesB))
12   ) ∘ tile(m, k, transpose(B))
13 ) ∘ tile(n, k, A)
```



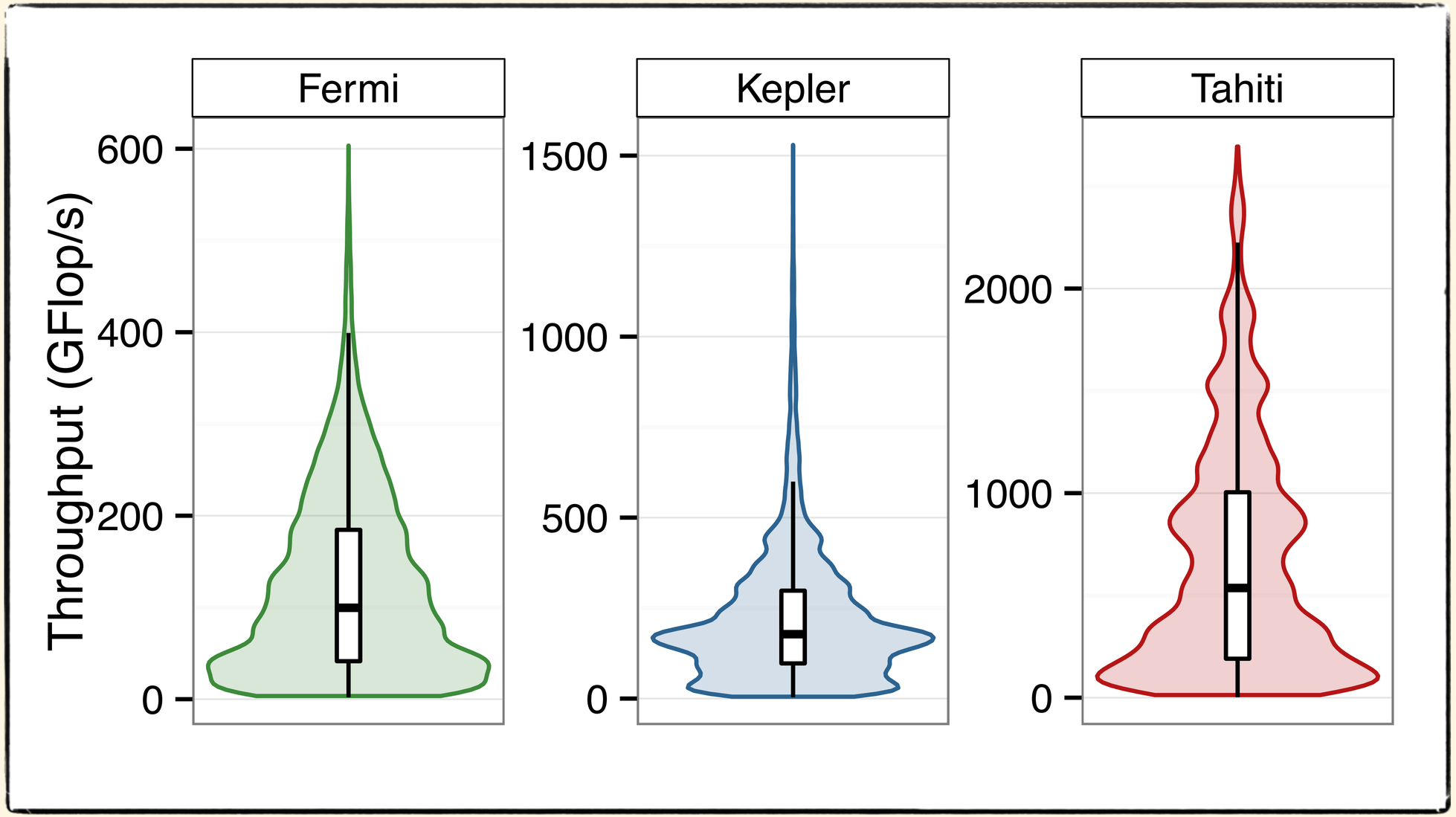
[GPGPU'16]


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

EXPLORATION BY REWRITING



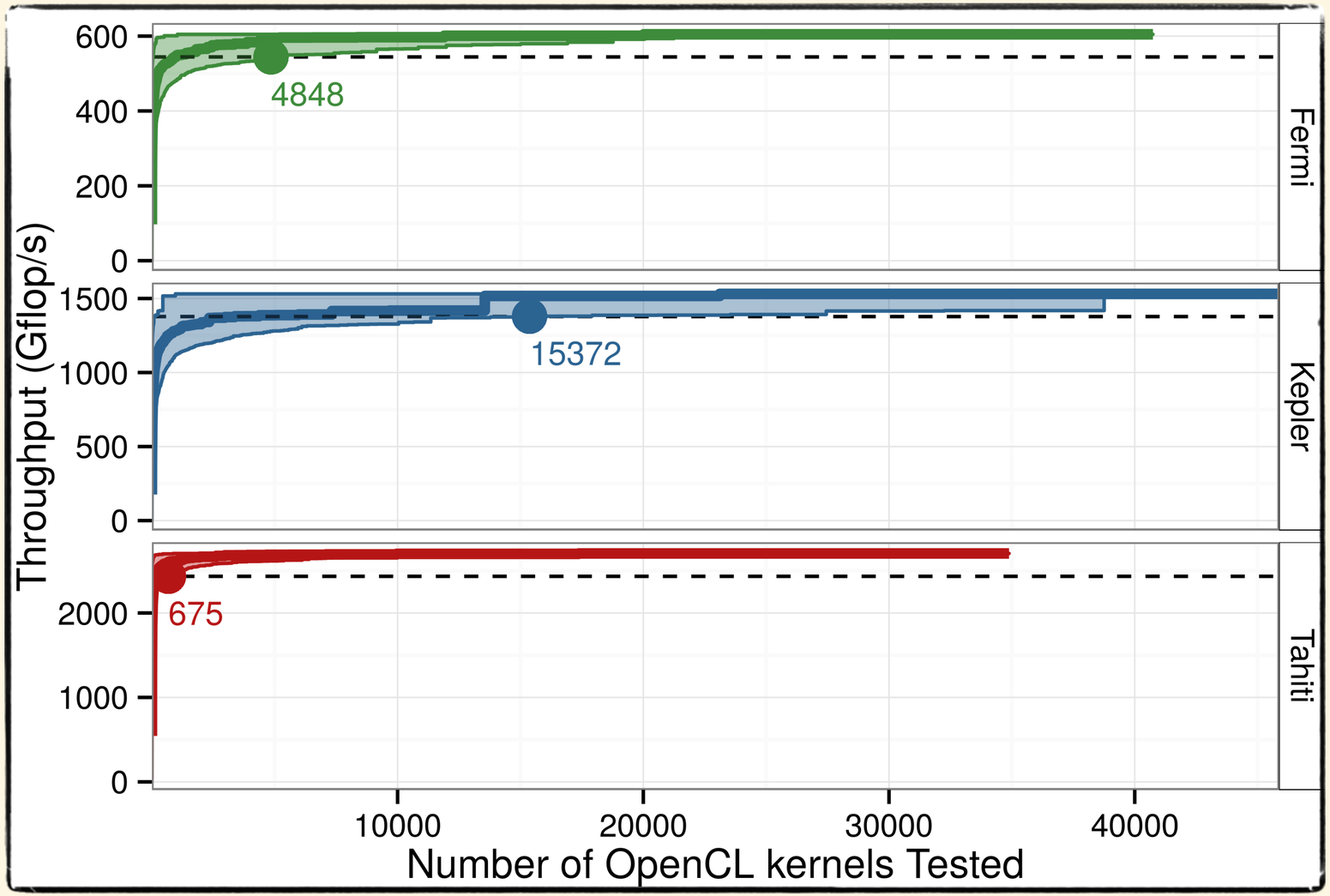
EXPLORATION SPACE MATRIX MULTIPLICATION



Only few generated code with very good performance

[GPGPU'16]

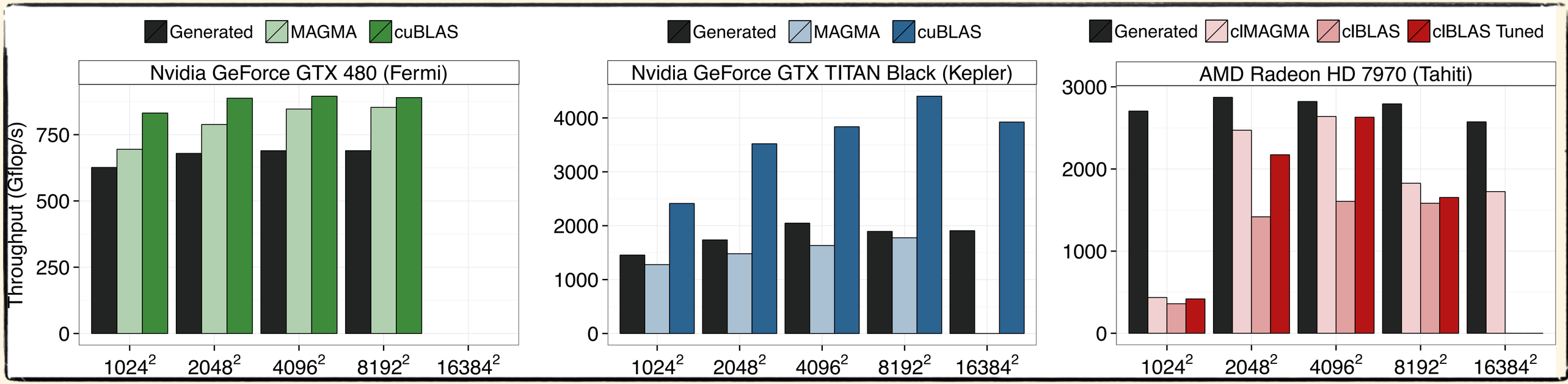
EVEN RANDOMISED SEARCH WORKS WELL!



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]

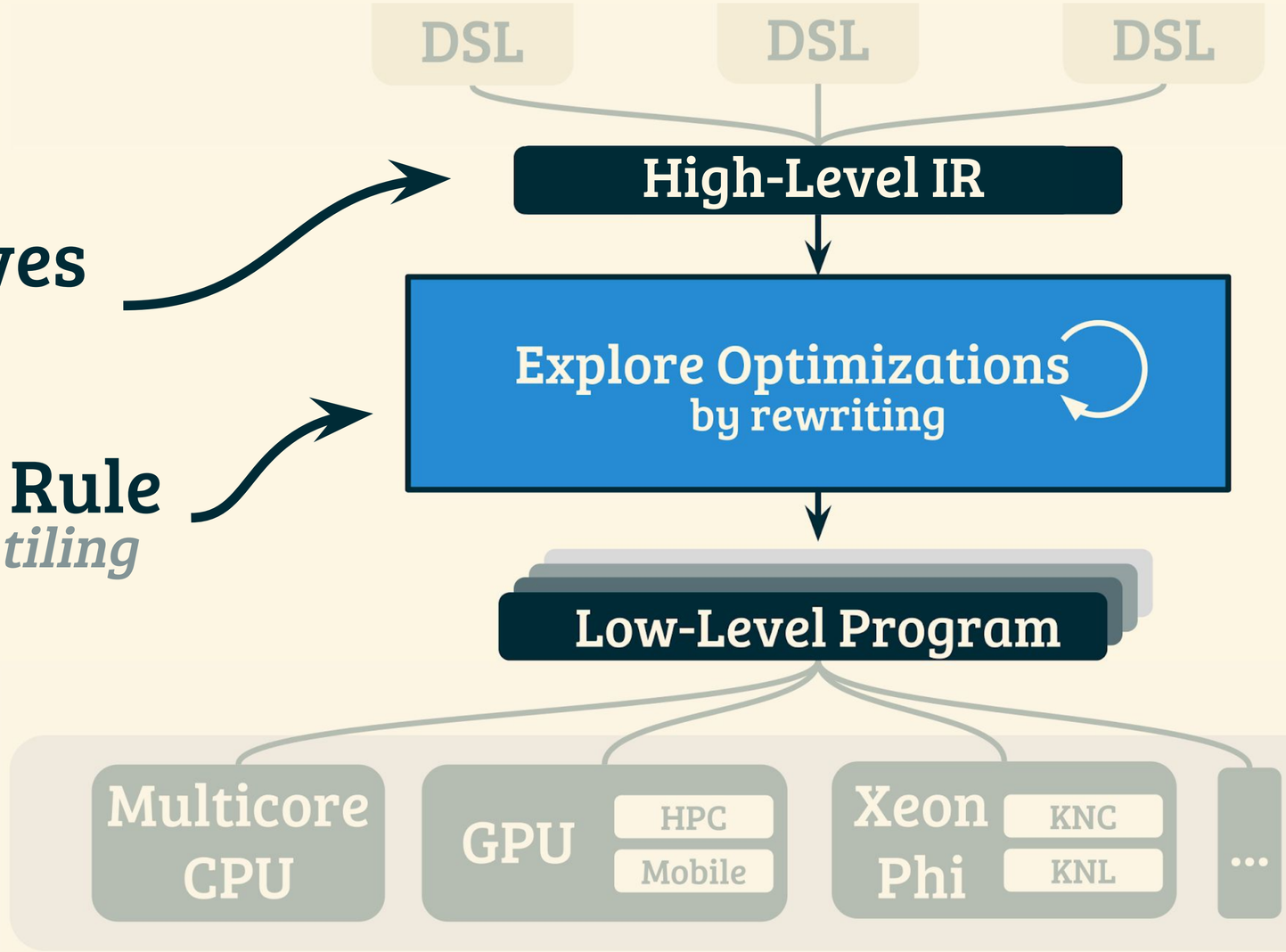
STENCIL COMPUTATIONS IN LIFT

[CGO'18] Best Paper Award

We added:

2 Primitives
pad, *slide*

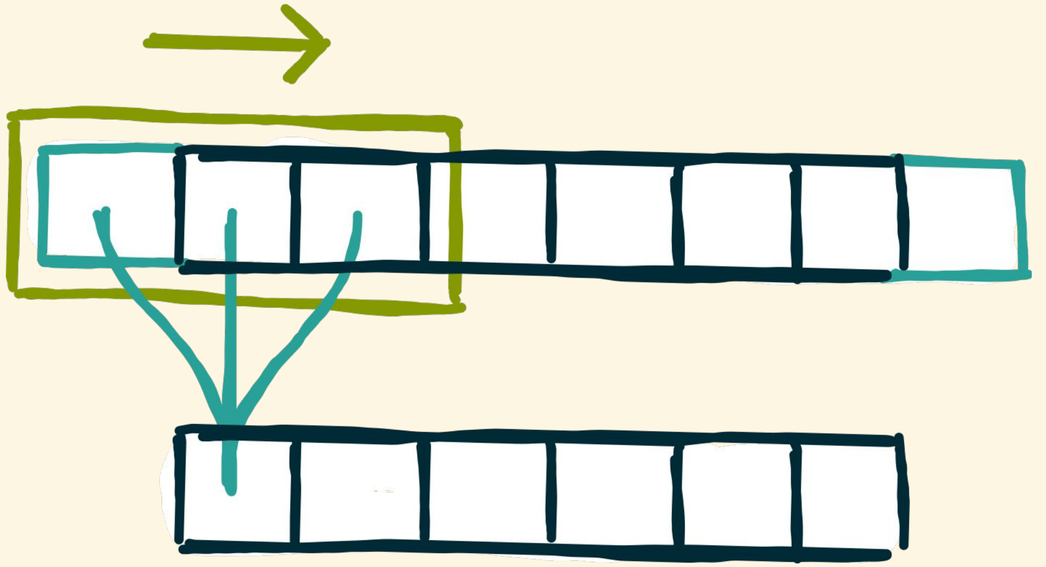
1 Rewrite Rule
overlapped tiling



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 ; }
```

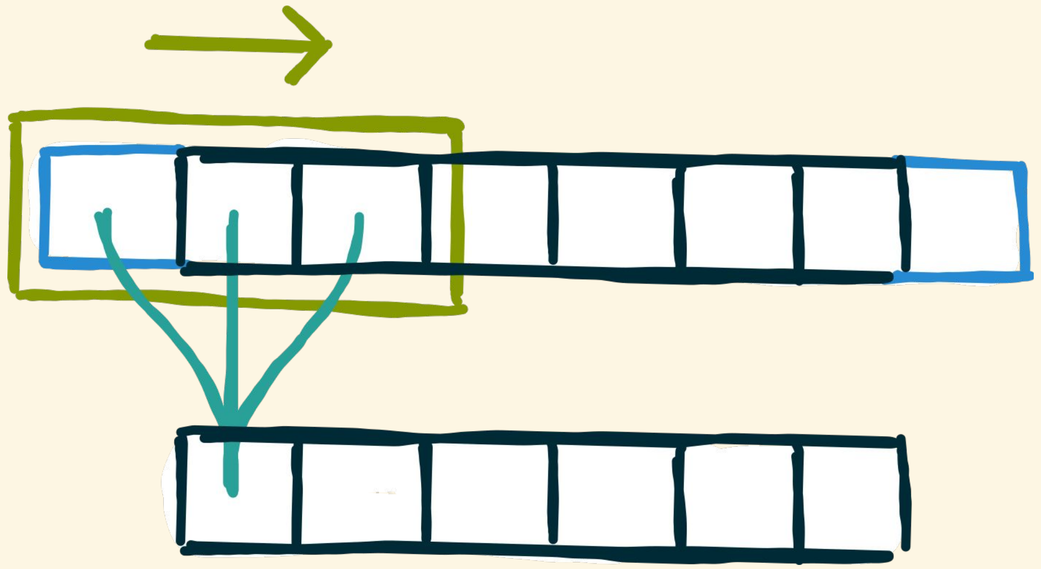


(a) access neighborhoods for every element

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 ; }
```



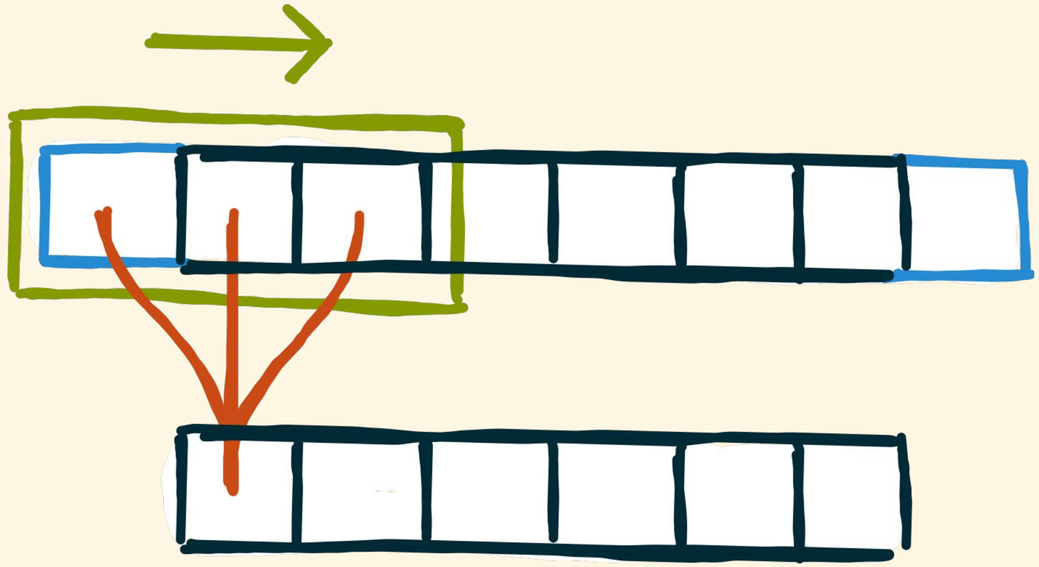
(a) access neighborhoods for every element

(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

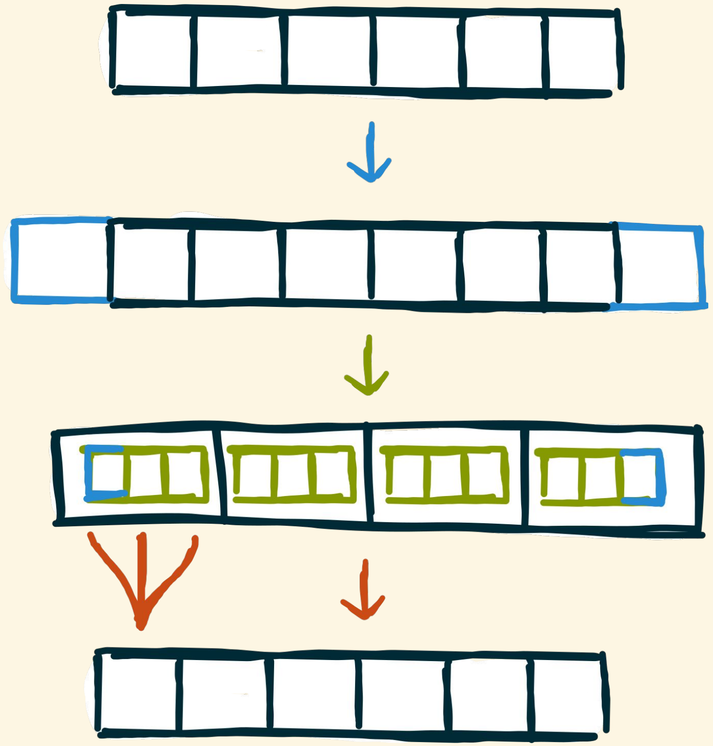
(b) specify boundary handling

(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 ; }
```



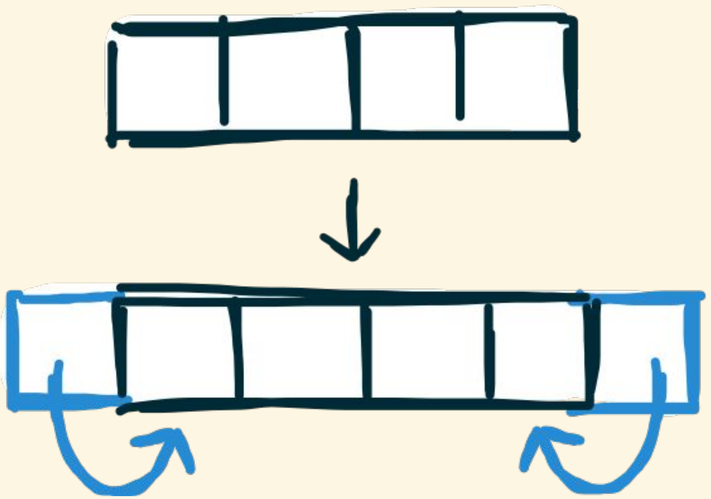
(a) access neighborhoods for every element

(b) specify boundary handling

(c) apply stencil function to neighborhoods

BOUNDARY HANDLING USING PAD

pad (reindexing)

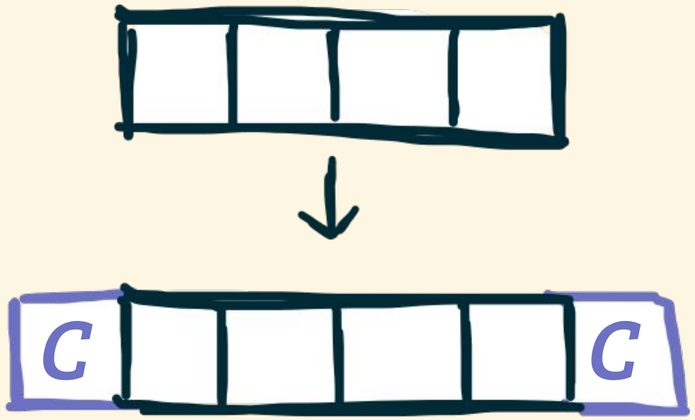


pad-reindexing.lift

```
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)

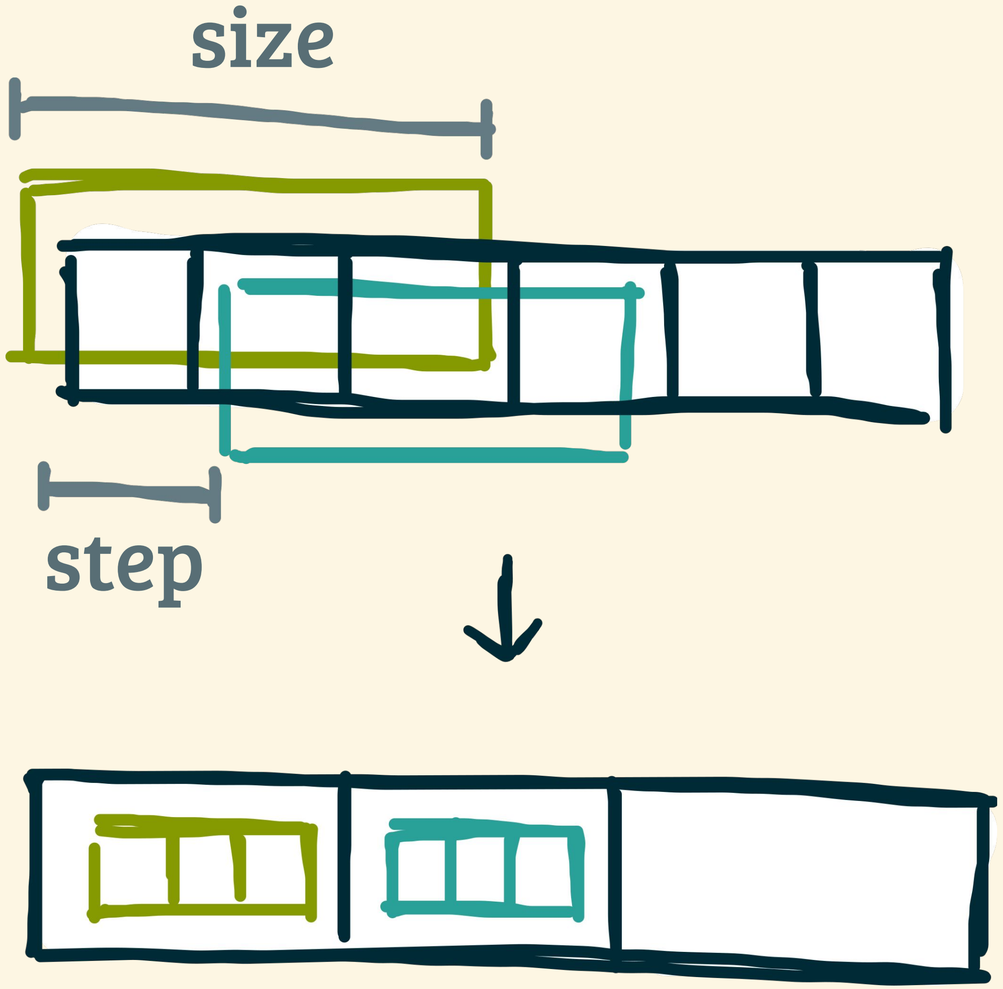


pad-constant.lift

```
constant(i, n) = C


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

NEIGHBORHOOD CREATION USING SLIDE

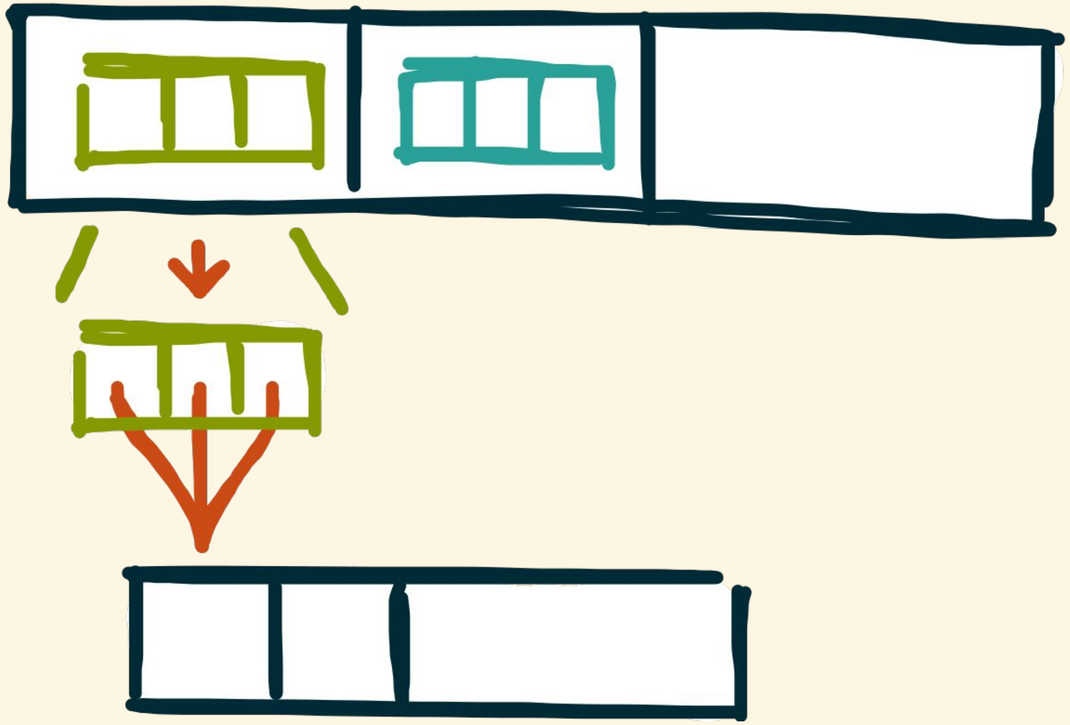


slide-example.lift

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

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

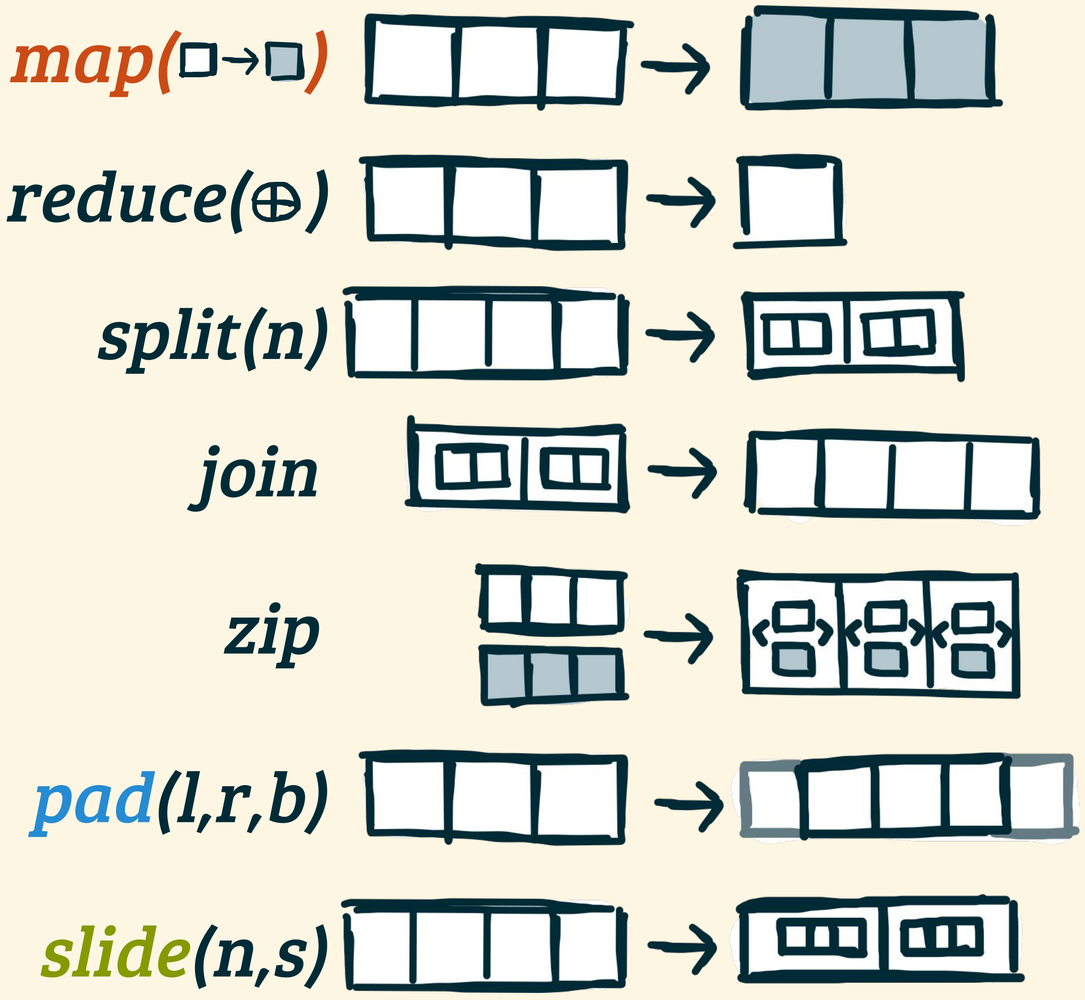
APPLYING STENCIL FUNCTION USING MAP



sum-neighborhoods.lift

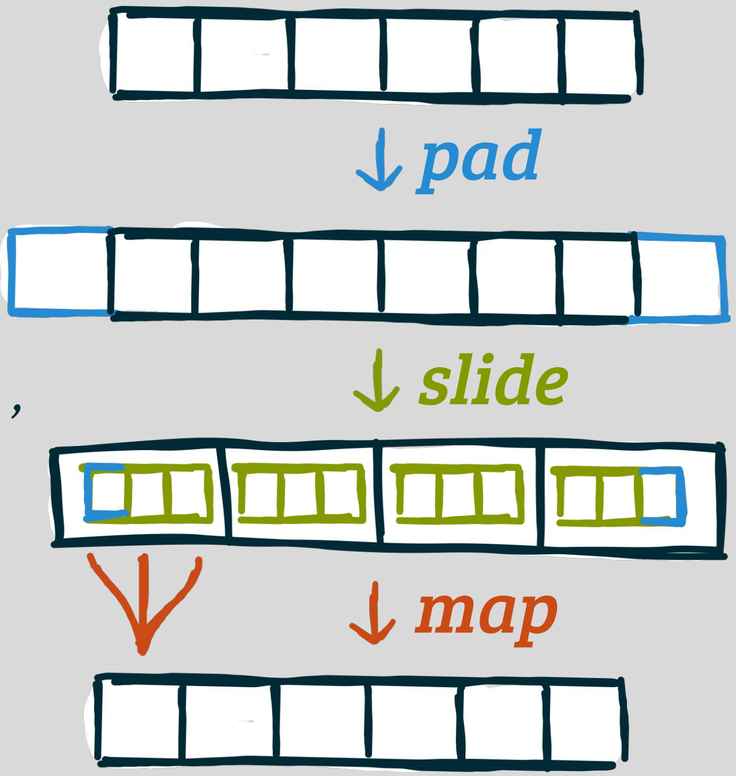
```
map(nbh =>
  reduce(add, 0.0f, nbh))
```

PUTTING IT TOGETHER



stencil1D.lift

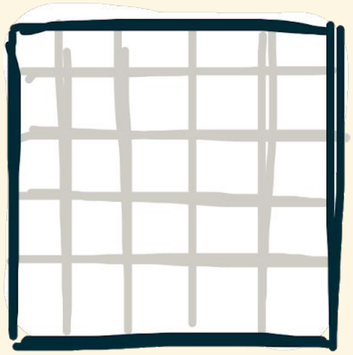
```
def stencil1D =
 fun(A =>
  map(reduce(add, 0.0f),
   slide(3,1,
    pad(1,1,clamp,A))))
```



MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

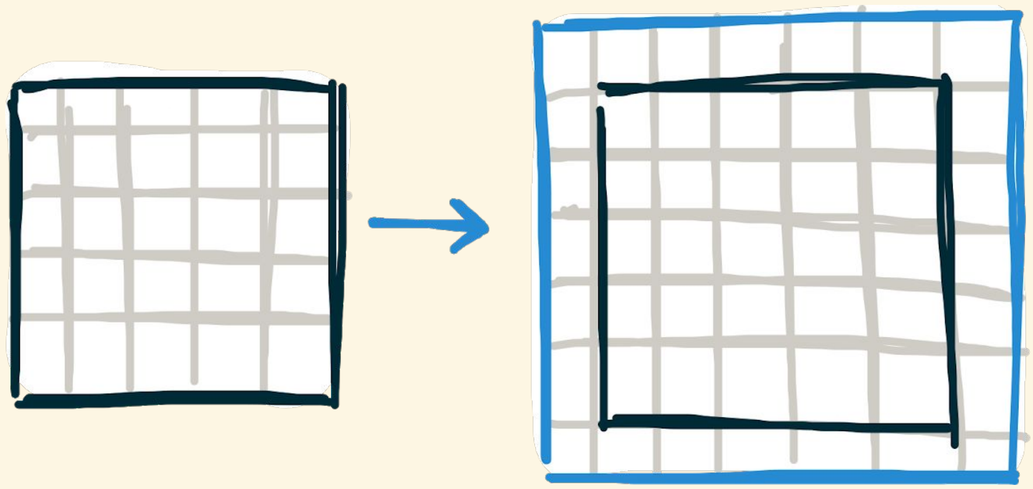
Decompose to Re-Compose



MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

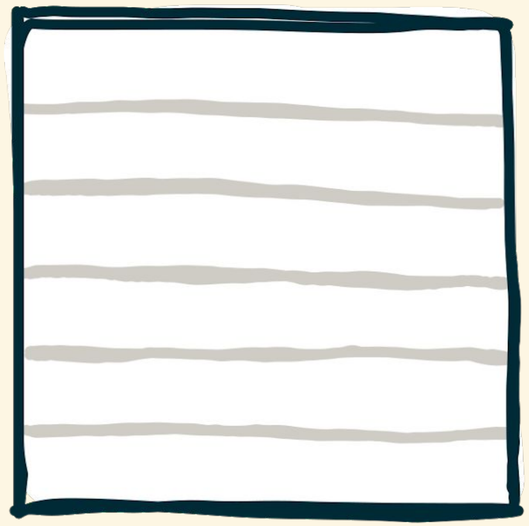
Decompose to Re-Compose



$pad_2(1, 1, clamp, \text{input})$

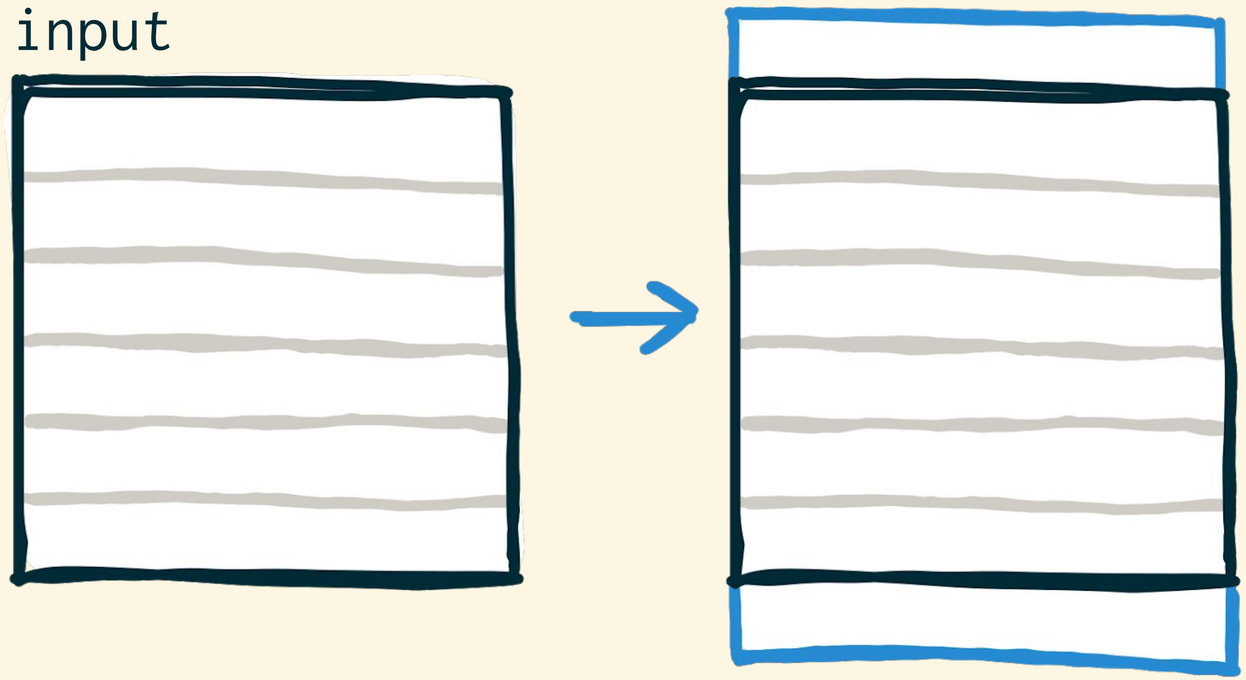
MULTIDIMENSIONAL BOUNDARY HANDLING USING pad_2

input



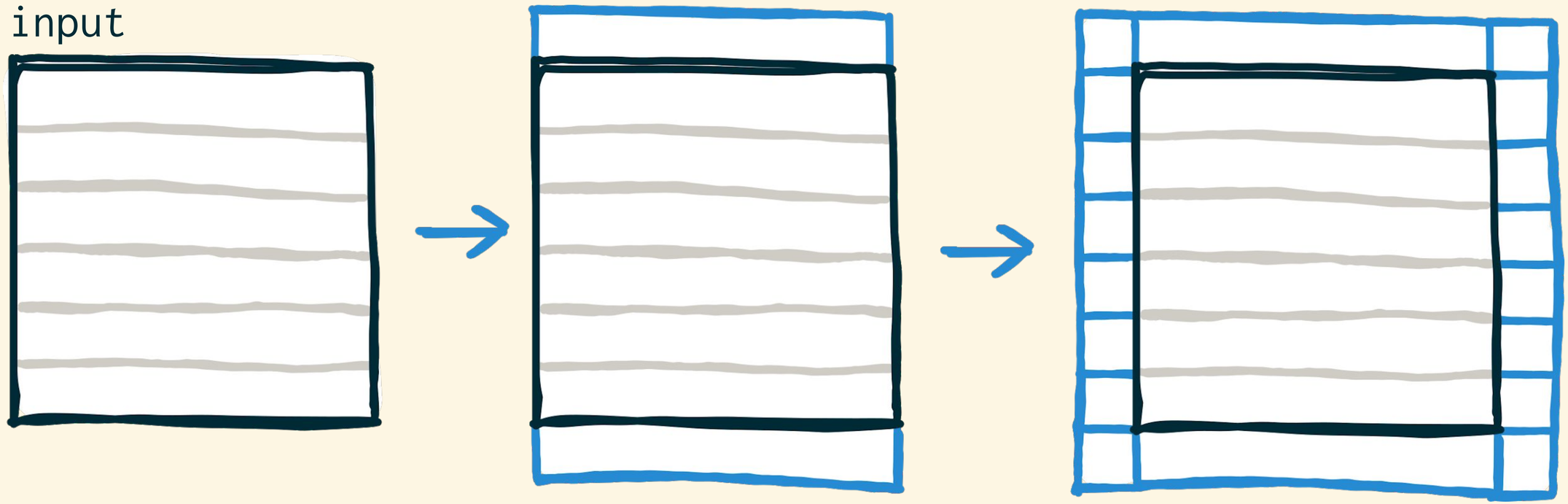
pad_2 =

MULTIDIMENSIONAL BOUNDARY HANDLING USING pad_2



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

MULTIDIMENSIONAL BOUNDARY HANDLING USING pad_2

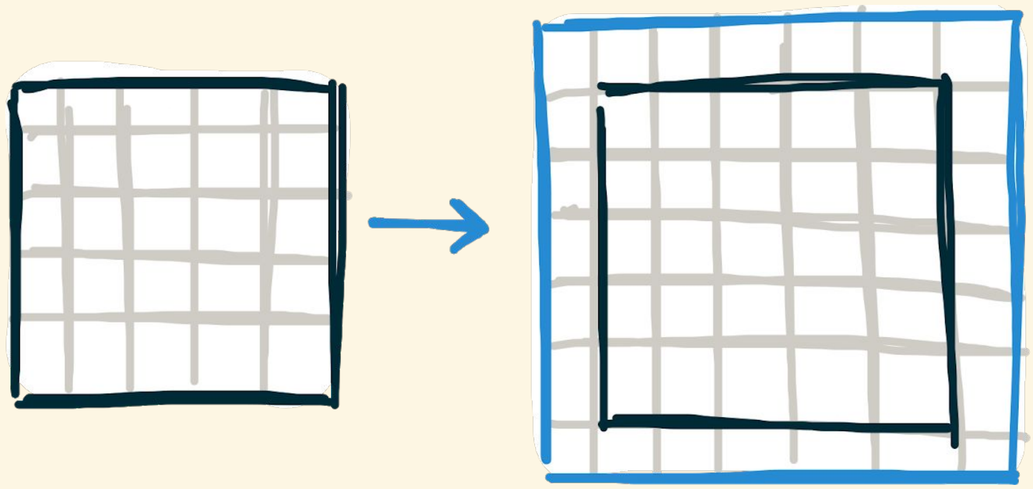


$$pad_2 = map(pad(l,r,b,pad(l,r,b,input)))$$

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

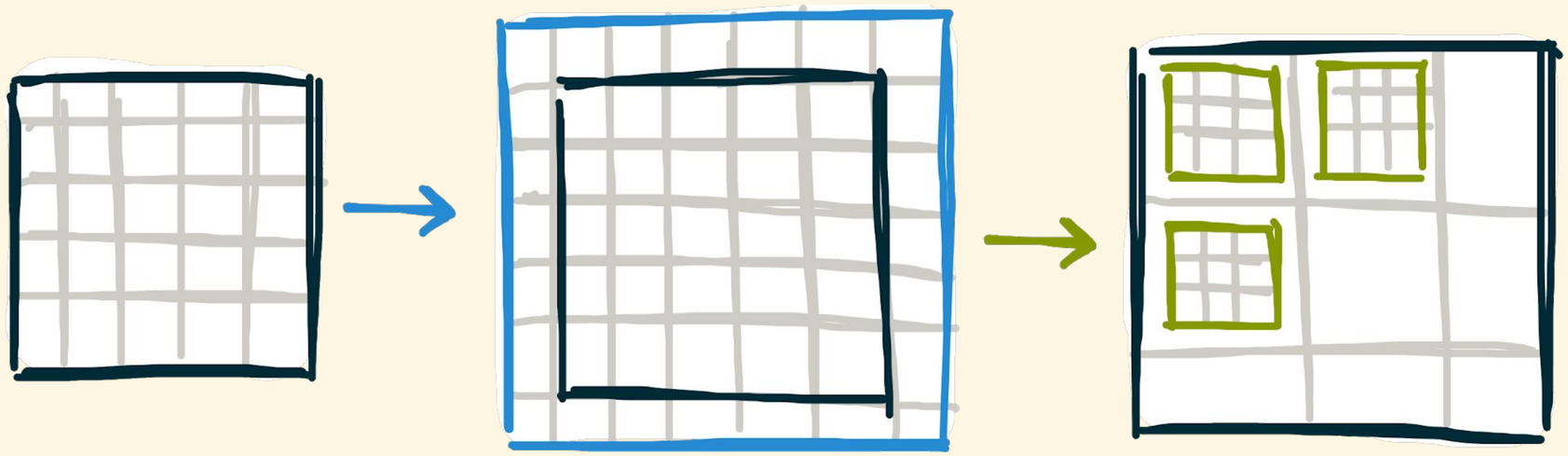


$pad_2(1, 1, clamp, input)$

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

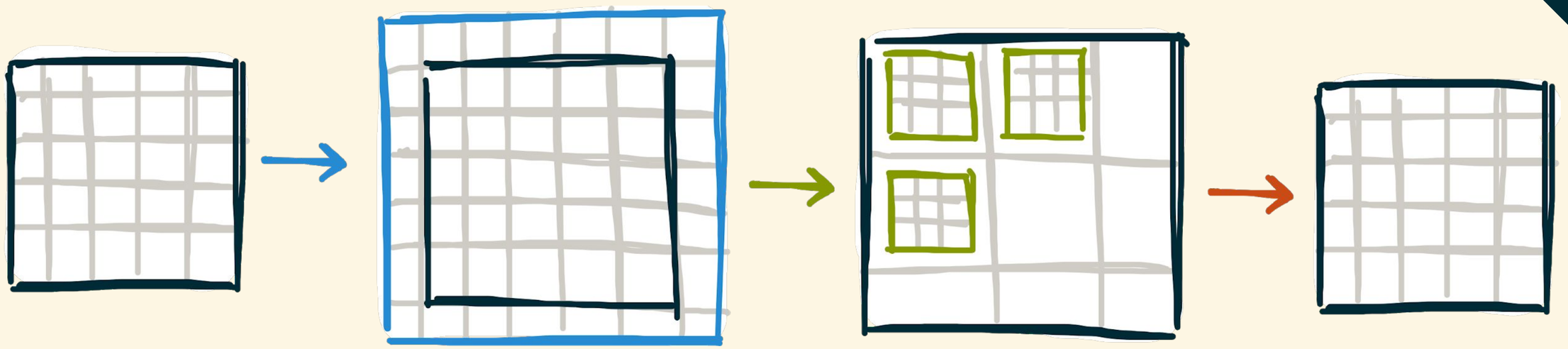


$$slide_2(3,1,\ pad_2(1,1,clamp,input))$$

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

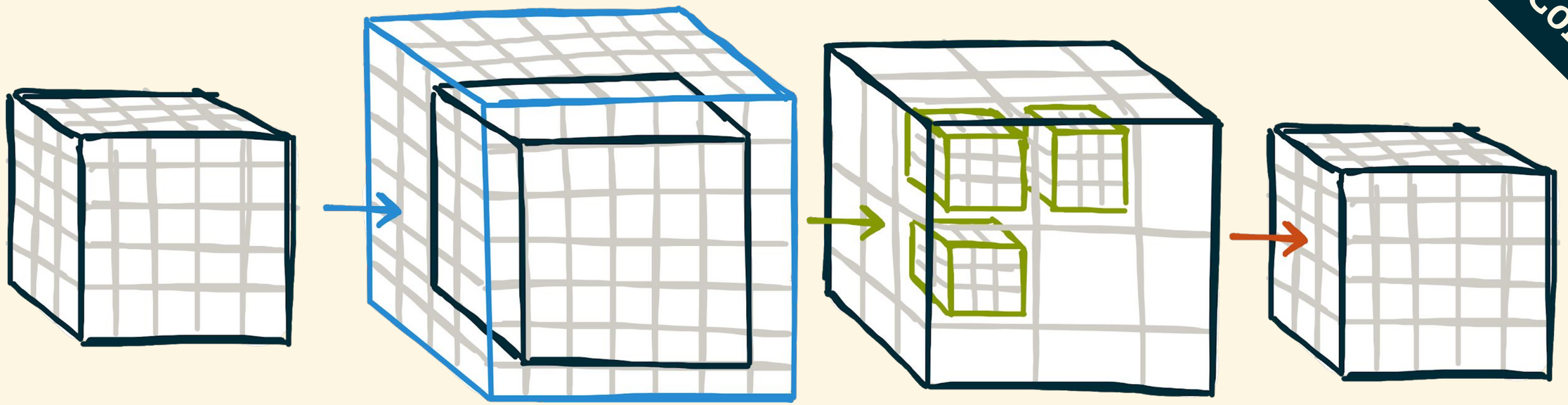


$$map_2(sum,\ slide_2(3,1,\ pad_2(1,1,clamp,input)))$$

MULTIDIMENSIONAL STENCIL COMPUTATIONS

are expressed as compositions of intuitive, generic 1D primitives

Decompose to Re-Compose

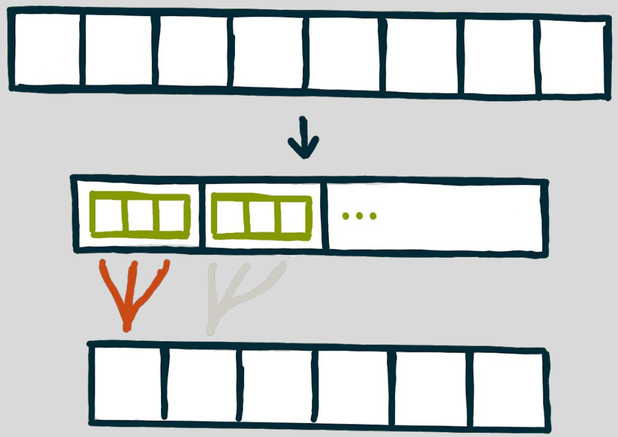


$$map_3(sum,\ slide_3(3,1,\ pad_3(1,1,clamp,input)))$$

OVERLAPPED TILING AS A REWRITE RULE

overlapped tiling rule

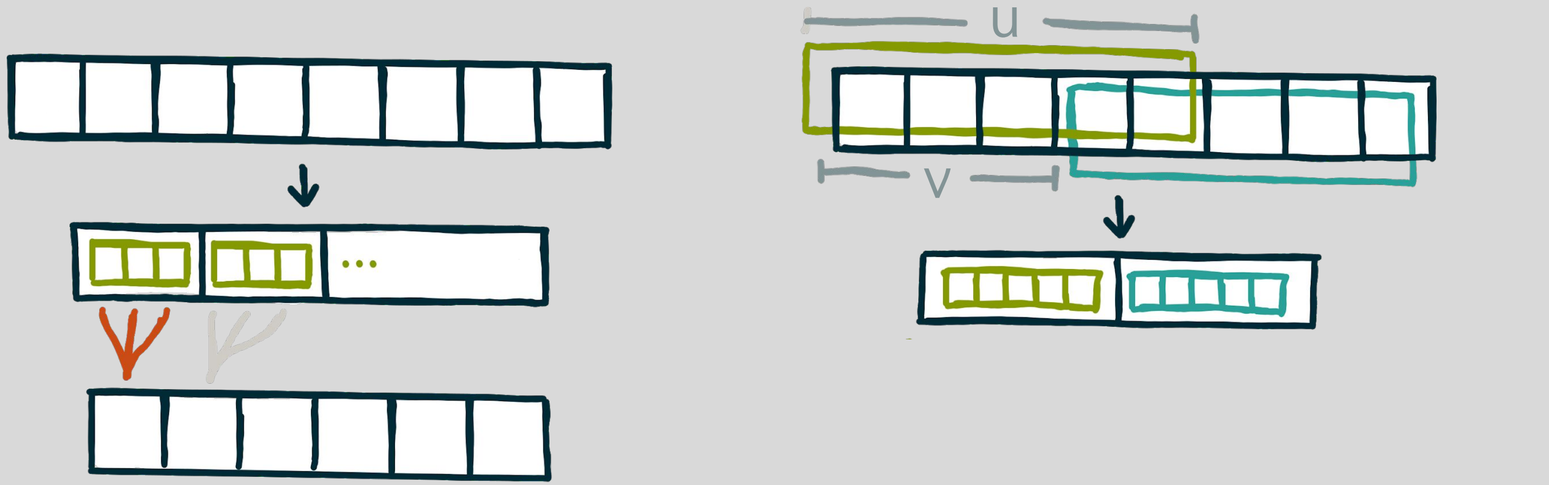
```
map(f, slide(3,1,input))
```



OVERLAPPED TILING AS A REWRITE RULE

overlapped tiling rule

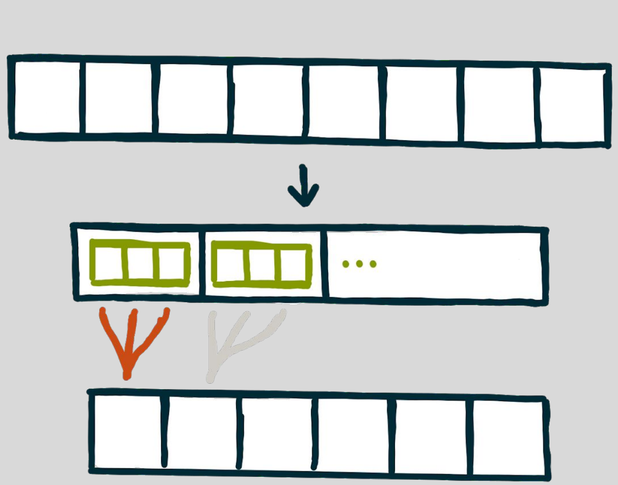
map(f, *slide*(3,1,input)) ↦ *slide*(u,v,input)

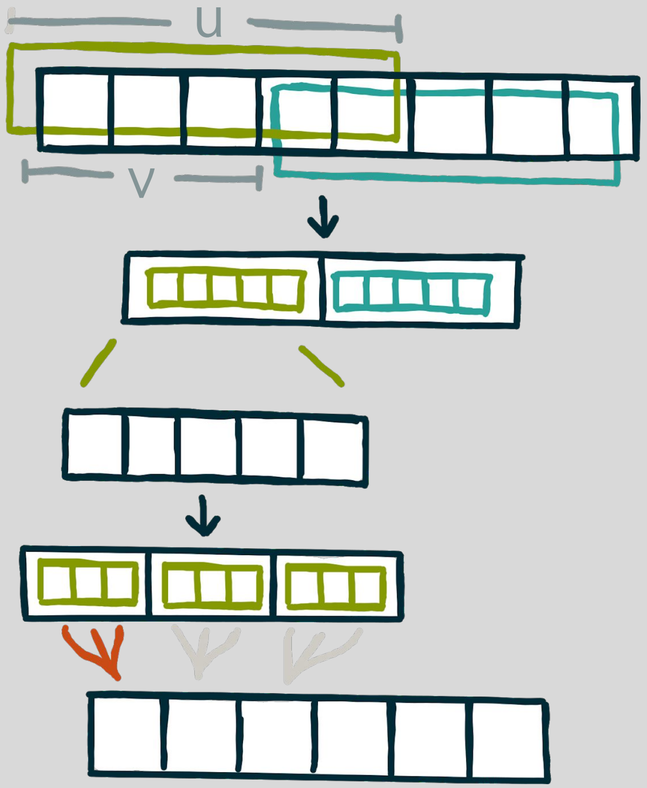


OVERLAPPED TILING AS A REWRITE RULE

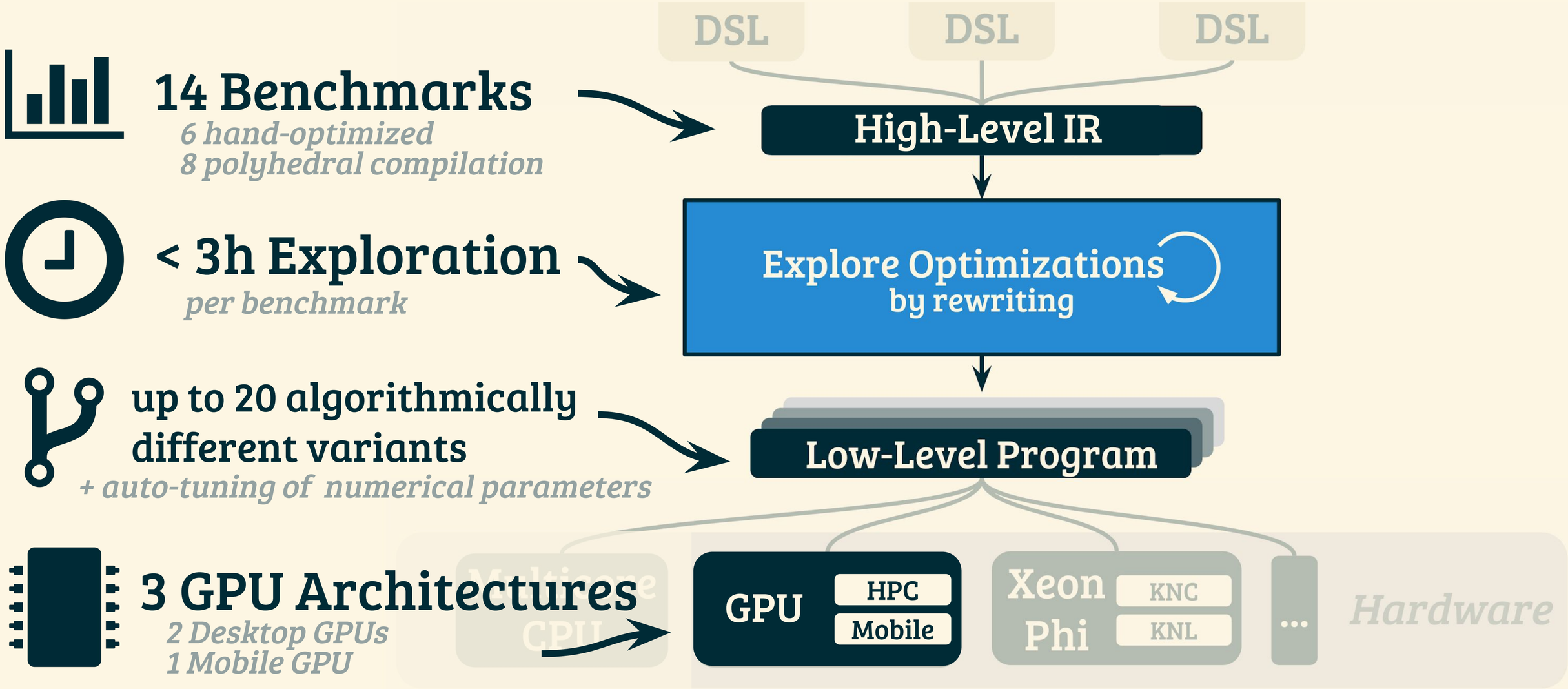
overlapped tiling rule

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



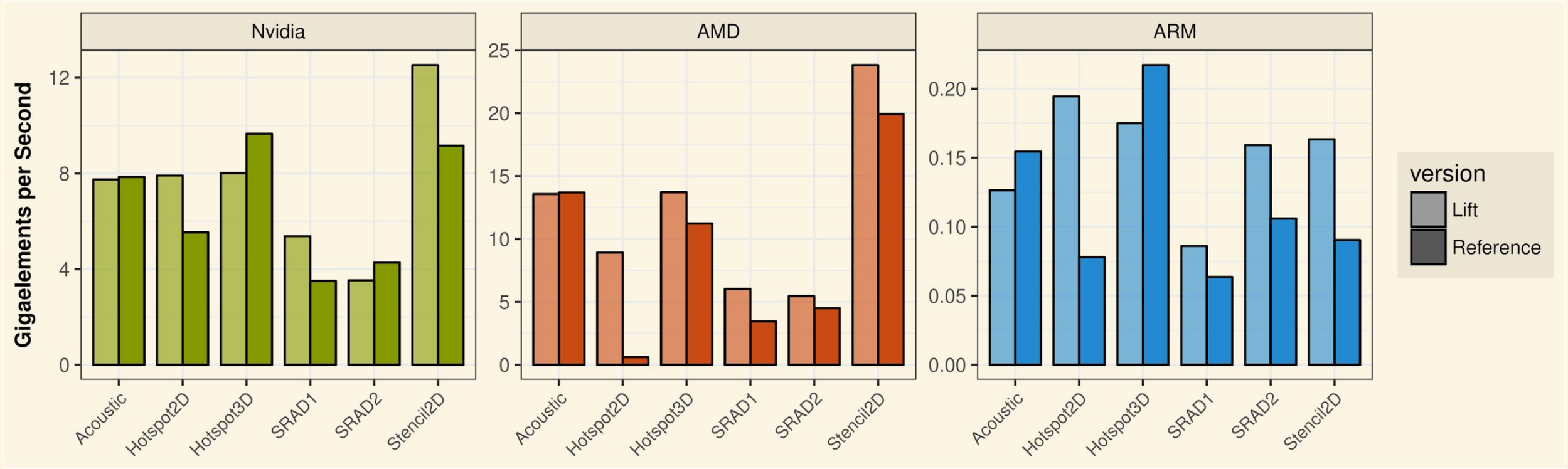


EXPERIMENTAL EVALUATION



COMPARISON WITH HAND-OPTIMIZED CODES

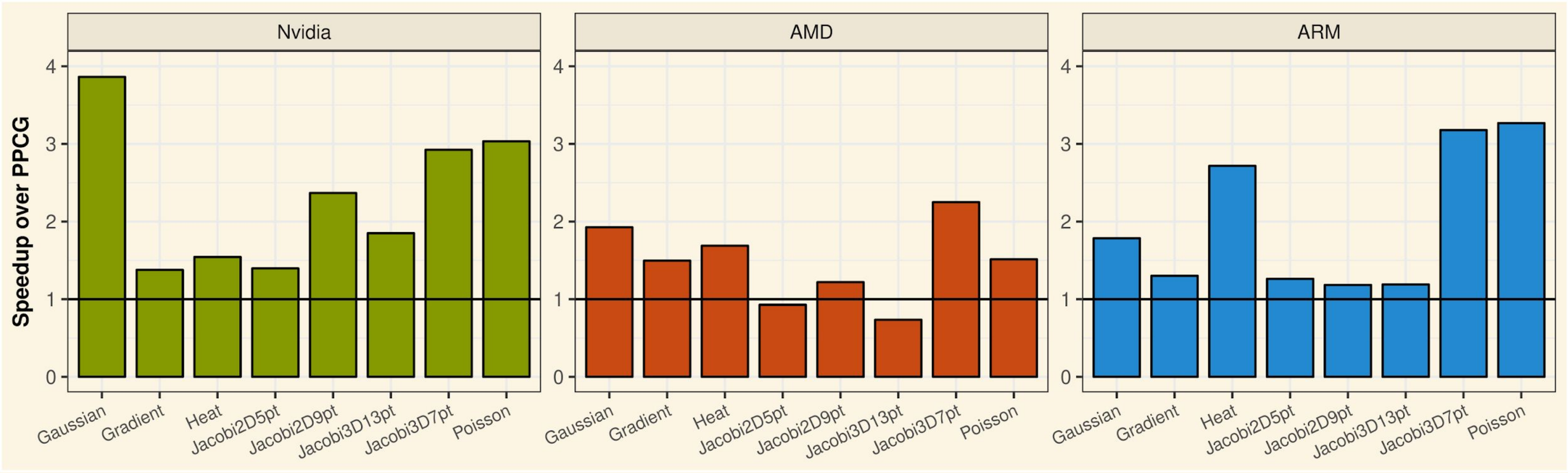
higher is better



Lift achieves the same performance as hand optimized code

COMPARISON WITH POLYHEDRAL COMPILATION

higher is better



Lift outperforms state-of-the-art optimizing compilers

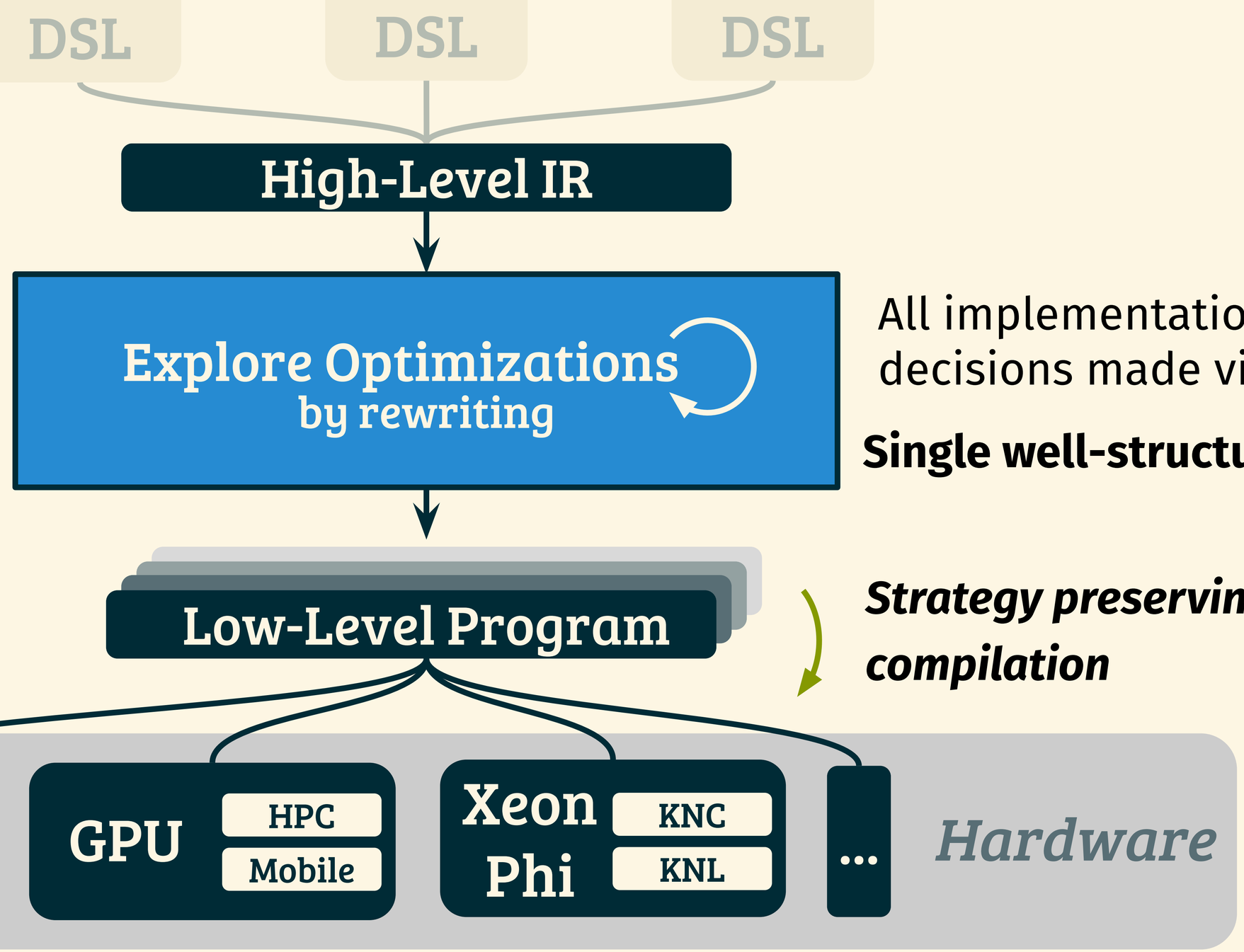
TOWARDS AN EXTENSIBLE AND SMART COMPILER

- We want compilers that are **easy to extend** and we want to **reuse** optimisations
  - LLVM has done this for low-level C-like languages

- We want the same for **higher-level languages**

- We want to define a **space of implementations and optimisations** that is automatic searchable
- We want a **generic** and hardware agnostic **optimisation process**
- We want to have a principled way to **inject domain and expert knowledge**

- Our approach:
  - **High-level primitives** describe algorithms & **Low-level primitives** describe hardware
  - **Rewrite rules** as the principled way to describe optimisations
  - **Strategies** as the principled way to inject domain and expert knowledge in the optimisation process

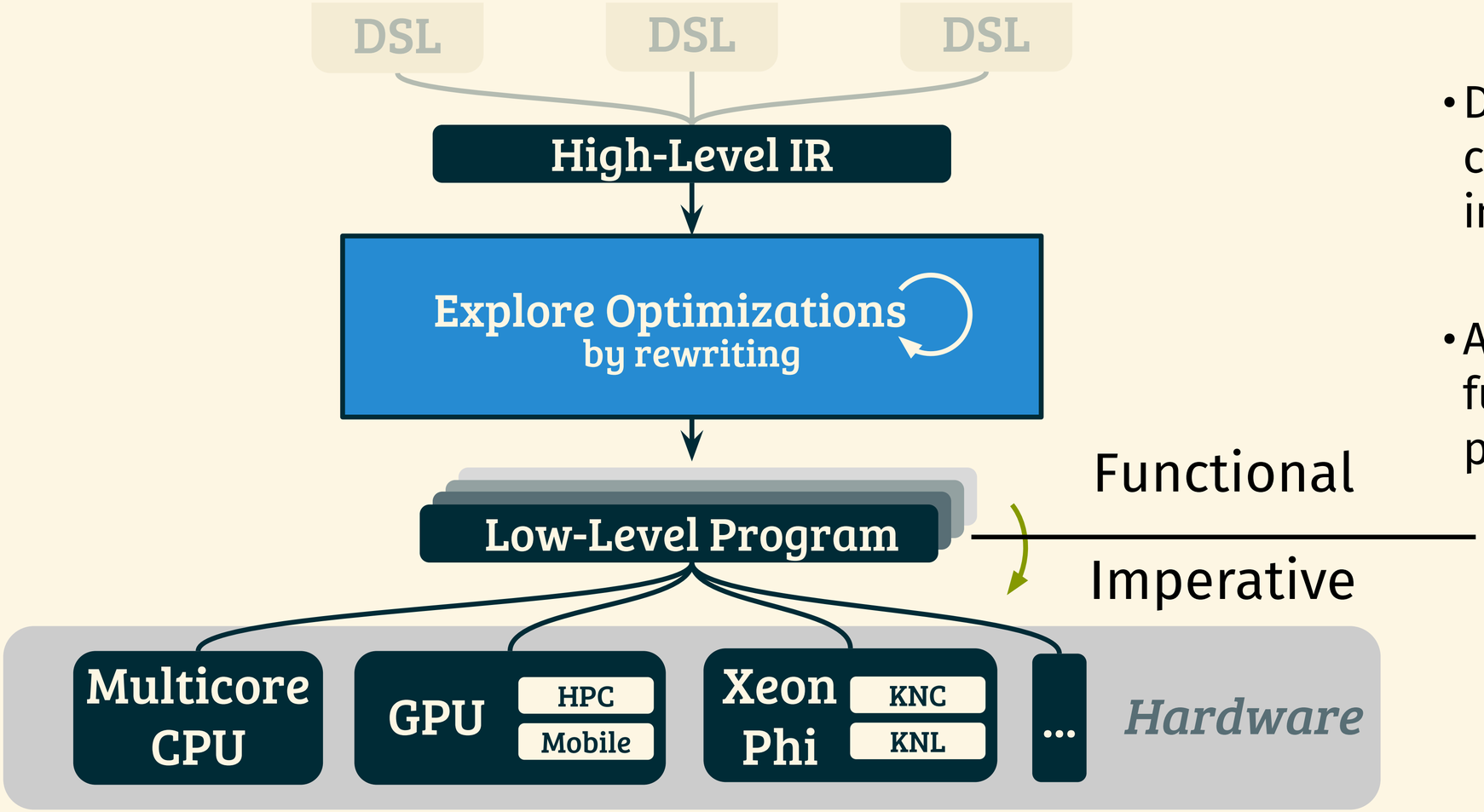


All implementation and optimization decisions made via rewriting

Single well-structured optimization space!

Strategy preserving compilation

STRATEGY PRESERVING COMPILATION WITH DPIA



- DPIA is a single language combining functional and imperative constructs
- A formal translation turns functional into imperative programs

Collaboration with **Bob Atkey** (University of Strathclyde), **Sam Lindley**, and **Christophe Dubach** (University of Edinburgh)

STRATEGY PRESERVING COMPILATION WITH DPIA

- Functional primitives are translated into imperative constructs, e.g. `map` is translated into `for`
- Translation guarantees deadlock and race freedom
- Translation is deterministic:
  - No decisions are made regarding parallelization, memory allocation, etc.

Functional

reduce (+) 0 (**map** (λx. **fst** x ∗ **snd** x) (**zip** xs ys))

↓

asScalar$_4$ (**join** (**mapWorkgroup** (λzs_1. **mapLocal** (λzs_2. **reduce** (λx a. (**fst** x ∗ **snd** x) + a) 0 (**split** 8192 zs_2)) zs_1)
(**split** 8192 (**zip** (**asVector**$_4$ xs) (**asVector**$_4$ ys))))))

↓

Imperative

```
parforWorkgroup (N/8192) (joinAcc (N/8192) 64 (asScalarAcc4 (N/128) out)) (λ gid o.
  parforLocal 64 o (λ lid o.
    newPrivate num⟨4⟩ accum.
      accum.1 := 0;
      for 2048 (λ i.
        accum.1 := accum.2 +
          (fst (idx (idx (split 2048 (idx (split (8192 ∗ 4) (zip (asVector4 xs) (asVector4 ys))) gid)) lid) i)) ∗
          (snd (idx (idx (split 2048 (idx (split (8192 ∗ 4) (zip (asVector4 xs) (asVector4 ys))) gid)) lid) i)) );
      out := accum.2 ))
```

↓

```
kernel void KERNEL(global float *out, const global float *restrict xs,
                   const global float *restrict ys, int N) {
  for (int g_id = get_group_id(0); g_id < N / 8192; g_id += get_num_groups(0)) {
    for (int l_id = get_local_id(0); l_id < 64; l_id += get_local_size(0)) {
      float4 accum;
      accum = (float4)(0.0, 0.0, 0.0, 0.0);
      for (int i = 0; i < 2048; i += 1) {
        accum = (accum +
                 (vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), xs) *
                  vload4(((2048 * l_id) + (8192 * 4 * g_id) + i), ys))); }
      vstore4(accum, ((64 * g_id) + l_id), out); } } }
```

https://michel.steuwer.info/publications/2017/arXiv/

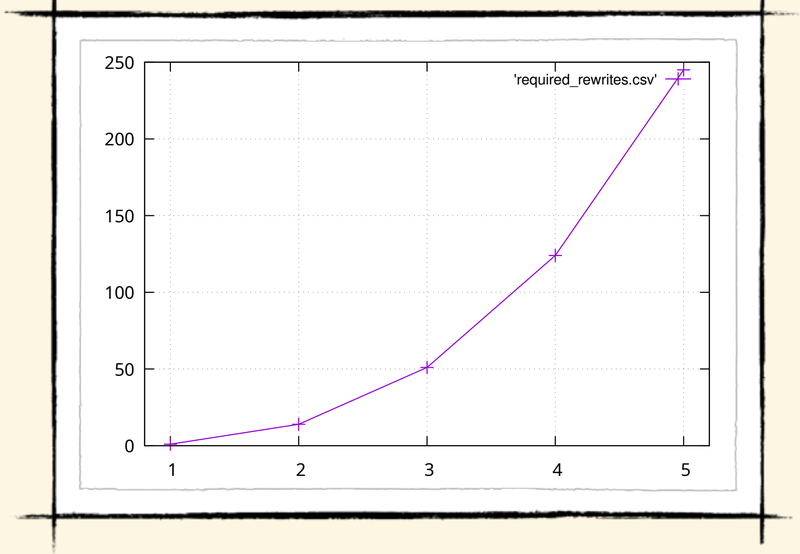
Collaboration with **Bob Atkey** (University of Strathclyde), **Sam Lindley**, and **Christophe Dubach** (University of Edinburgh)

ELEVATE: A LANGUAGE FOR EXPRESSING OPTIMISATION STRATEGIES

Beta

- **Motivation:** Let programmers inject domain or expert knowledge into the optimisation process

Number of Rewrites for tiling nD arrays



- Should be a proper **language** to allow programmers to express complex optimisation strategies
  - It should be impossible to build *illegal* strategies
  - Strategies should *not be fix and built-in, but extensible*
  - Strategies should *compose* to build larger out of smaller strategies

- Inspired by:
  - **Halide** which separates programs into a *functional description* and a *schedule*
  - Strategy languages like **Stratego** designed for formal term rewriting

Collaboration with **Bastian Hagedorn** (University of Münster)

PROBLEMS WITH HALIDE'S SCHEDULING "LANGUAGE"

Beta

- Example **MatMult:**

Functional description:

```
Func prod("prod");
RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
out(x, y) = prod(x, y);
```

Schedule:

```
const int warp_size = 32;
const int vec_size = 2;
const int x_tile = 3;
const int y_tile = 4;
const int y_unroll = 8;
const int r_unroll = 1;

Var xi, yi, xio, xii, yii, xo, yo, x_pair, xiio, ty;
RVar rxo, rxi;

out.bound(x, 0, size)
   .bound(y, 0, size)
   .tile(x, y, xi, yi, x_tile * vec_size * warp_size, y_tile * y_unroll)
   .split(yi, ty, yi, y_unroll)
   .vectorize(xi, vec_size)
   .split(xi, xio, xii, warp_size)
   .reorder(xio, yi, xii, ty, x, y)
   .unroll(xio)
   .unroll(yi)
   .gpu_blocks(x, y)
   .gpu_threads(ty)
   .gpu_lanes(xii);
prod.store_in(MemoryType::Register)
    .compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, TailStrategy::RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty)
    .unroll(xi, vec_size)
    .gpu_lanes(xi)
    .unroll(xo)
    .unroll(y)
    .update()
    .split(x, xo, xi, warp_size * vec_size, TailStrategy::RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty)
    .unroll(xi, vec_size)
    .gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll)
    .reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo)
    .unroll(y);

Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in()
    .compute_at(prod, ty)
    .split(Bx, xo, xi, warp_size)
    .gpu_lanes(xi)
    .unroll(xo).unroll(By);

A.in()
    .compute_at(prod, rxo)
    .vectorize(Ax, vec_size)
    .split(Ax, xo, xi, warp_size)
    .gpu_lanes(xi)
    .unroll(xo).split(Ay, yo, yi, y_tile)
    .gpu_threads(yi).unroll(yo);

A.in().in().compute_at(prod, rxi)
    .vectorize(Ax, vec_size)
    .split(Ax, xo, xi, warp_size)
    .gpu_lanes(xi)
    .unroll(xo).unroll(Ay);

set_alignment_and_bounds(A, size);
set_alignment_and_bounds(B, size);
set_alignment_and_bounds(out, size);
```

Schedule much harder to write and reason about then functional program!

Problems with Halide's scheduling "language"

Beta

Fixed set of optimisations ⇒ **lack of extensibility**

Schedule:

```
…
out.bound(x, 0, size)
    .bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size, y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio)
    .unroll(yi)
    .gpu_blocks(x, y)
    .gpu_threads(ty)
    .gpu_lanes(xii);
…
```

What happens if the order of these are swapped?

⇒ **unclear semantics** ⇒ **unclear how to automatically generate schedules**

Optimisation strategies from first principle

Beta

- A **Strategy** is a function: `LiftExpr → LiftExpr`
- A rewrite rule is the simples form of a strategy:

```
def split-join-rule = (n: Int) ⇒ (expr: LiftExpr) ⇒ expr match {
  case map(f, input) ⇒ join(map(map(f), split(n, input)))
  case _ ⇒ throw Exception()
}
```

- We have an operator **`applyAt`** to apply a strategy at a particular location inside of a `LiftExpr`

```
def applyAt(strategy: Strategy, location: Location): Strategy
```

- Locations can be specified absolute, relative, or finding the first Lift pattern satisfying a predicate

COMPOSING BASIC STRATEGIES

Beta

- Strategies compose into (slightly) larger strategies:

```
def id = (expr: LiftExpr) ⇒ expr

def leftChoice = (fst: Strategy, snd: Strategy) ⇒ {
  (expr: LiftExpr) ⇒ try { fst(expr) } catch { case _ ⇒ snd(expr) }
}

def try = (s: Strategy) ⇒ leftChoise(s, id)

def seq = (fst: Strategy, snd: Strategy) = snd(fst(expr))

def repeat = (s: Strategy) ⇒ try(seq(s, repeat(s)))

def normalize = (s: Strategy) ⇒ repeat(applyAt(s, findFirst(isDefined(s))))
```

BUILDING A TILING STRATEGY

Beta

- Using these building blocks and standard functional programming patterns like fold we define the *tiling optimisation strategy* that is built-in in Halide

```
def tile = (n: Int) ⇒ (expr: LiftExpr) ⇒ {
  seq(
    fold( listPotentialRewrites( split-join-rule(n), expr ), (e, (s, l)) ⇒
      try(applyAt(s, l)(e)) ),
    seq(
      normalize( map-fission-rule /* map(f o g) ⇒ map(f) o map(g) */ ),
      interleaveDimensions( numberOfDimensions(expr) )
    )
  )
}
```

TILING IN THREE STEPS

Beta

Lift Expression

```
map(map(f), input)
```

1. Tile in every dimension: **fold(…)**

```
join(map(map(
  join(map(map(f),
    split(jTile)))),
      split(iTile)))
```

2. Rewrite to normal form: **normalize(…)**

```
join(
  map(map(join),
    map(map(map(map(f))),
      map(map(split(jTile)),
        split(iTile)))))
```

Pseudo C Code

```
for (int i = 0; i < M; i++)
  for (int j = 0; j < N; j++)
    out[i][j] = f(in[i][j]);
```

```
for (int i = 0; i < M/iTile; i++)
  for (int ii = 0; ii < iTile; ii++)
    for (int j = 0; j < N/jTile; j++)
      for (int jj = 0; jj < jTile; jj++) {
        int posI = i * iTile + ii;
        int posJ = j * jTile + jj;
        out[posI][posJ] = f(in[posI][posJ]);
      }
```

TILING IN THREE STEPS

Beta

Lift Expression

```
join(
  map(map(join),
    map(map(map(map(f))),
      map(map(split(jTile)),
        split(iTile)))))
```

Pseudo C Code

```
for (int i = 0; i < M/iTile; i++)
  for (int ii = 0; ii < iTile; ii++)
    for (int j = 0; j < N/jTile; j++)
      for (int jj = 0; jj < jTile; jj++) {
        int posI = i * iTile + ii;
        int posJ = j * jTile + jj;
        out[posI][posJ] = f(in[posI][posJ]);
      }
```

3. Rearrange dimensions: **`interleaveDimensions(…)`**

```
join(
  map(map(join),
    map(transpose,
      map(map(map(map(f))),
        map(transpose,
          map(map(split(jTile)),
            split(iTile)))))))
```

```
for (int i = 0; i < M/iTile; i++)
  for (int j = 0; j < N/jTile; j++)
    for (int ii = 0; ii < iTile; ii++)
      for (int jj = 0; jj < jTile; jj++) {
        int posI = i * iTile + ii;
        int posJ = j * jTile + jj;
        out[posI][posJ] = f(in[posI][posJ]);
      }
```

WHY IS THIS USEFUL?

Beta

- We control different categories of rewrites with the same language
  - Algorithmic:
    - computational optimisations, like fusion or fission
    - data-layout transformations, like tiling
  - Hardware-specific:
    - memory optimisations, like usage of scratchpad/shared memory
    - parallelism mapping, like using workgroups vs. only global threads

  ⇒ **principled way to understand and write optimisations**

- No built-in strategies: *tiling* strategy is defined in a library

  ⇒ **easy to extend to other optimisations and domains**

- Easy to guarantee that strategies not fail (see `try` strategy)

  ⇒ **amenable for automatic exploration of strategies**

LIFT IS OPEN SOURCE!

University of Glasgow

THE UNIVERSITY of EDINBURGH

WWU MÜNSTER

more info at:

lift-project.org

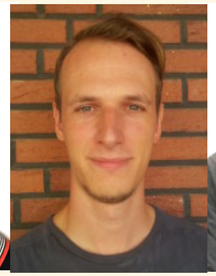
Paper

Artifacts

Source Code


Naums Mogers


Bastian Köpcke


Christof Schlaak

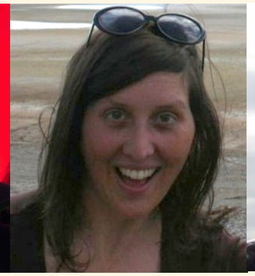

Christophe Dubach


Bastian Hagedorn


Andrej Ivanis


Toomas Remmelg


Michel Steuwer


Larisa Stoltzfus


Thomas Koehler


Federico Pizzuti


Lu Li

michel.steuwer.info