



University  
of Glasgow

Beta

# ELEVATE *a language to write composable program optimisations*

Michel Steuwer – [michel.steuwer@glasgow.ac.uk](mailto:michel.steuwer@glasgow.ac.uk)

**INSPIRING  
PEOPLE**





Joined work with

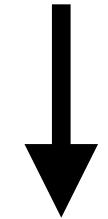
## **Bastian Hagedorn**

<https://bastianhagedorn.github.io>

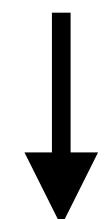
# How do we optimise programs today?



**Program**



**Compiler**



**Performance ?**

- Change the program manually
- Change compiler options

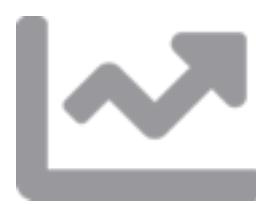
# How do we optimise programs today?



**Program**



**Compiler**



**Performance ?**

```
for (i = 0; i < N; ++i) {  
    for (j = 0; j < N; ++j){  
        C[i][j] = 0;  
        for (k = 0; k < N; ++k)  
            C[i][j] += A[i][k] * B[k][j]; } }
```

# How do we optimise programs today?



Program



Compiler



Performance ?

```
for (i = 0; i < N; ++i) {  
    for (j = 0; j < N; ++j){  
        C[i][j] = 0;  
        for (k = 0; k < N; ++k)  
            C[i][j] += A[i][k] * B[k][j]; } }
```

CPUs

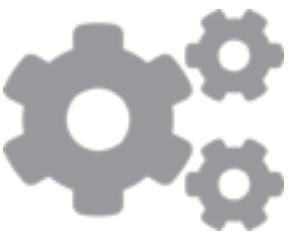
- Blocking / Tiling
- Exploit ILP
- Exploit locality

```
for (ii = 0; ii < N; ii += ib) {  
    for (kk = 0; kk < N; kk += kb) {  
        for (j=0; j < N; j += 2) {  
            for(i = ii; i < ii + ib; i += 2 ) {  
                if (kk == 0)  
                    acc00 = acc01 = acc10 = acc11 = 0;  
                else {  
                    acc00 = C[i + 0][j + 0];  
                    acc01 = C[i + 0][j + 1];  
                    acc10 = C[i + 1][j + 0];  
                    acc11 = C[i + 1][j + 1]; }  
                for (k = kk; k < kk + kb; k++) {  
                    acc00 += A[k][j + 0] * B[i + 0][k];  
                    acc01 += A[k][j + 1] * B[i + 0][k];  
                    acc10 += A[k][j + 0] * B[i + 1][k];  
                    acc11 += A[k][j + 1] * B[i + 1][k];  
                }  
                C[i + 0][j + 0] = acc00;  
                C[i + 0][j + 1] = acc01;  
                C[i + 1][j + 0] = acc10;  
                C[i + 1][j + 1] = acc11; } } } }
```

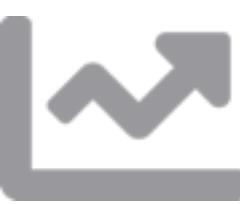
# How do we optimise programs today?



Program



Compiler



Performance ?

```
for (i = 0; i < N; ++i) {  
    for (j = 0; j < N; ++j){  
        C[i][j] = 0;  
        for (k = 0; k < N; ++k)  
            C[i][j] += A[i][k] * B[k][j]; } }
```

GPUs ↓

```
1 kernel __attribute__((reqd_work_group_size(32, 8, 1)))  
2     int K, M, N {  
3         local float tileA[512]; tileB[512];  
4         local float tileC[512];  
5         private float acc_0; ...; acc_31;  
6         private float blockOfA_0; ...; blockOfA_3;  
7         private float blockOfB_0; ...; blockOfB_3;  
8         int lid0 = local_id(0); lid1 = local_id(1);  
9         int wid0 = group_id(0); wid1 = group_id(1);  
10        for (int w1=wid1; w1<M/64; w1+=num_grps(1)) {  
11            for (int w0=wid0; w0<N/64; w0+=num_grps(0)) {  
12                acc_0 = 0.0f; ...; acc_31 = 0.0f;  
13                for (int i=0; i<K/8; i++) {  
14                    vstore4(vload4(lid1*K/4+2*i*16+w1*lid0,A),  
15                     .16*lid1+lid0, tileA);  
16                    vstore4(vload4(lid1*K/4+2*i*16+w0*lid0,B),  
17                     .16*lid1+lid0, tileB);  
18                    barrier(...);  
19                    for (int j = 0; j<8; j++) {  
20                        blockOfA_0 = tileA[(9*64*j+lid1)*8];  
21                        ... 6 more statements  
22                        blockOfA_7 = tileA[(7*64*j+lid1)*8];  
23                        blockOfB_0 = tileB[(6*64*j+lid0)*8];  
24                        ... 2 more statements  
25                        blockOfB_3 = tileB[(4*64*j+lid0)*8];  
26                        acc_0 += blockOfA_0 * blockOfB_0;  
27                        acc_1 += blockOfA_0 * blockOfB_1;  
28                        acc_2 += blockOfA_0 * blockOfB_2;  
29                        acc_3 += blockOfA_0 * blockOfB_3;  
30                        ... 24 more statements  
31                        acc_28 += blockOfA_7 * blockOfB_0;  
32                        acc_29 += blockOfA_7 * blockOfB_1;  
33                        acc_30 += blockOfA_7 * blockOfB_2;  
34                        acc_31 += blockOfA_7 * blockOfB_3;  
35                        ...  
36                        barrier(...);  
37                        ...  
38                        CL [ 0~8*lid1*M~64~w0~64~w1~N~0~N~lid0]=acc_0;  
39                        CL [16~8*lid1*M~64~w0~64~w1~N~0~N~lid0]=acc_1;  
40                        CL [32~8*lid1*M~64~w0~64~w1~N~0~N~lid0]=acc_2;  
41                        CL [48~8*lid1*M~64~w0~64~w1~N~0~N~lid0]=acc_3;  
42                        ... 24 more statements  
43                        CL [ 0~8*lid1*M~64~w0~64~w1~N~7~N~lid0]=acc_18;  
44                        CL [16~8*lid1*M~64~w0~64~w1~N~7~N~lid0]=acc_19;  
45                        CL [32~8*lid1*M~64~w0~64~w1~N~7~N~lid0]=acc_20;  
46                        CL [48~8*lid1*M~64~w0~64~w1~N~7~N~lid0]=acc_21;  
47                        ... 31 more statements  
48                        CL [ 0~8*lid1*M~64~w0~64~w1~N~7~N~lid0]=acc_28;  
49                        CL [16~8*lid1*M~64~w0~64~w1~N~7~N~lid0]=acc_29;  
50                        CL [32~8*lid1*M~64~w0~64~w1~N~7~N~lid0]=acc_30;  
51                        CL [48~8*lid1*M~64~w0~64~w1~N~7~N~lid0]=acc_31;  
52                        ...  
53                        ...  
54                        ...
```

AMD

```
// kernel __attribute__((reqd_work_group_size(32, 8, 1)))  
void KERNEL(const global float *restrict A, const global float *restrict B,  
           const global float *restrict C, float alpha, float beta,  
           global float *out, int K, int M, int N) {  
    local float l_tmp_1[512];  
    local float l_tmp_2[1024];  
    float acc_1_1_425 = 0.0f;  
    // ... 31 more  
    float p_tmp_1_1_457;  
    // ... 107 more  
    int wg_id_1 = get_group_id(1);  
    int wg_id_0 = get_group_id(0);  
    for (int i = 0; i < (K / 8); i = (1 + i)) {  
        int l_id_0 = get_local_id(0);  
        for (int l_id_0 = get_local_id(0); (l_id_0 < 64); l_id_0 = (32 + l_id_0)) {  
            l_tmp_1[(l_id_0 + (64 * l_id_1))] =  
                (A[(l_id_0 + (8 * M * i) + (64 * wg_id_1) + (M * l_id_1))]);  
        }  
        barrier(CLK_LOCAL_MEM_FENCE);  
        for (int l_id_0 = get_local_id(0); (l_id_0 < 128); l_id_0 = (32 + l_id_0)) {  
            l_tmp_2[(l_id_0 + (128 * wg_id_0) + (128 * wg_id_1))] =  
                (B[(l_id_0 + (8 * N * i) + (128 * wg_id_0) + (N * l_id_1))]);  
        }  
        barrier(CLK_LOCAL_MEM_FENCE);  
        barrier(CLK_LOCAL_MEM_FENCE);  
        for (int j = 0; (j < 8); j = (1 + j)) {  
            p_tmp_1_1_457 = (l_tmp_1[(0 + (8 * get_local_id(1)) + (64 * j))]);  
            p_tmp_1_2_458 = (l_tmp_1[(1 + (8 * get_local_id(1)) + (64 * j))]);  
            // ... 6 more  
            p_tmp_2_1_465 = (l_tmp_2[(0 + (128 * j) + get_local_id(0))]);  
            p_tmp_2_2_466 = (l_tmp_2[(32 + (128 * j) + get_local_id(0))]);  
            // ... 2 more  
            p_tmp_3_1_469 = p_tmp_1_1_457 * p_tmp_2_1_465;  
            acc_1_1_425 = acc_1_1_425 + p_tmp_3_1_469;  
            // ... 31 more  
        }  
        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);  
        p_tmp_4_1_501 = acc_1_1_425 * alpha;  
        p_tmp_5_1_533 = C[((64 * N * wg_id_1) + (8 * N * get_local_id(1)) +  
                           (128 * wg_id_0) + get_local_id(0))] *  
                           beta;  
        out[((64 * N * wg_id_1) + (8 * N * get_local_id(1)) + (128 * wg_id_0) +  
              get_local_id(0))] = p_tmp_4_1_501 + p_tmp_5_1_533;  
        // ... 31 more  
    }  
}
```

Nvidia

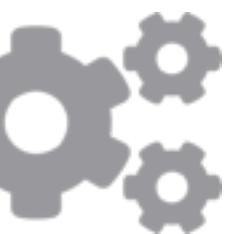
```
1 kernel void mm(global float4* const A,  
2     global float4* const B,  
3     global float2* C, uint n) {  
4     uint i = get_global_id(0);  
5     uint j = get_global_id(1);  
6     uint nv4 = n >> 2;  
7     float4 ab = (float4)(0.0f);  
8     for (uint k = 0; k < nv4; ++k) {  
9         float4 a0 = A[ 2*i ~ nv4+k];  
10        float4 a1 = A[(2*i+1)~nv4+k];  
11        float4 b0 = B[ 2*j ~ nv4+k];  
12        float4 b1 = B[(2*j+1)~nv4+k];  
13        ab += (float4)dot(a0, b0), dot(a1, b1));  
14        dot(a1, b0), dot(a1, b1));  
15        uint ix = 2*i~(n>>1) = j;  
16        C[ix] = ab.s01;  
17        C[ix + (n>>1)] = ab.s23; }  
18 }
```

ARM

# How do we optimise programs today?



# Program



# Compiler



# Performance ?

```
for (i = 0; i < N; ++i) {
    for (j = 0; j < N; ++j){
        C[i][j] = 0;
        for (k = 0; k < N; ++k)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

# GPUs

```

1 kernel void mm_and_opt(global float * A, B,
2                         int K, M, N) {
3     local float tileA[512]; tileB[512];
4
5     private float acc_B; ... i acc_B;
6     private float tileC[512]; tileD[512];
7
8     for (int k = 0; k < K; k++) {
9         for (int m = 0; m < M; m++) {
10             for (int n = 0; n < N; n++) {
11                 tileA[m * 16 + n] = A[m * K + k];
12                 tileB[n * 16 + k] = B[k * N + n];
13
14                 tileC[m * 16 + n] = tileA[m * 16 + n] *
15                     tileB[n * 16 + k];
16
17                 tileD[n * 16 + k] = tileC[m * 16 + n];
18             }
19         }
20     }
21
22     for (int k = 0; k < K; k++) {
23         for (int m = 0; m < M; m++) {
24             for (int n = 0; n < N; n++) {
25                 C[m * K + k] += tileD[n * 16 + k];
26             }
27         }
28     }
29 }

```

```
8
9 int lid9 = local_id(9); lid1 = local_id(1);
10 int lid2 = local_id(2); lid3 = local_id(3);
11
12 if
```

# mem accesses

## Vectorization

## Blocking / Tiling

```

33
34
35
36 acc_28 += blockOfIA_7 * blockOfB_0;
37 acc_29 += blockOfIA_7 * blockOfB_1;
38 acc_30 += blockOfIA_7 * blockOfB_2;
39 acc_31 += blockOfIA_7 * blockOfB_3;
40 }
41 barrier[...];
42
43 }
44
45 C[ 8-8*lid1*M+64*w6+64*w1*N+0*N+lid6]=acc;
46 C[16-8*lid1*M+64*w6+64*w1*N+0*N+lid6]=acc;
47 C[32-8*lid1*M+64*w6+64*w1*N+0*N+lid6]=acc;
48 C[48-8*lid1*M+64*w6+64*w1*N+0*N+1*ld6]=acc;
49 ...
50 C[ 8-8*lid1*M+64*w6+64*w1*N+7*N+lid6]=acc;
51 C[16-8*lid1*M+64*w6+64*w1*N+7*N+lid6]=acc;
52 C[32-8*lid1*M+64*w6+64*w1*N+7*N+lid6]=acc;
53 C[48-8*lid1*M+64*w6+64*w1*N+7*N+1*ld6]=acc;
54 } } }
```

```

// kernel __attribute__((reqd_work_group_size(32, 8, 1)))
void KERNEL(const global float *restrict A, const global float *restrict B,
            const global float *restrict C, float alpha, float beta,
            global float *out, int K, int M, int N) {
    local float l_tmp_1[512];
    local float l_tmp_2[1024];
    float acc_1_1_425 = 0.0f;
    // ... 31 more
    float p_tmp_1_1_457;
    // ... 107 more
    int wg_id_1 = get_group_id(1);
    int wg_id_0 = get_group_id(0);
    for (int i = 0; i < (K / 8); i = (1 + i)) {
        int l_id_1 = ((i * 8) % 32);
        for (int l_1 : l_id_1) {
            l_tmp_1[l_1] = (A[(l_1 * 32) + i]);
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        for (int l_2 : l_id_1) {
            l_tmp_2[l_2] = (B[(l_2 * 32) + i]);
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        barrier(CLK_GLOBAL_MEM_FENCE);
        for (int j : l_id_0) {
            p_tmp_1_1_457 = p_tmp_1_1_457 + l_tmp_2[l_2];
            p_tmp_1_2_457 = p_tmp_1_2_457 + l_tmp_2[l_2];
            // ... 6 i
            p_tmp_2_1_457 = p_tmp_2_1_457 + l_tmp_2[l_2];
            p_tmp_2_2_457 = p_tmp_2_2_457 + l_tmp_2[l_2];
            // ... 2 i
            p_tmp_3_1_469 = p_tmp_1_1_457 * p_tmp_2_1_457;
            acc_1_1_425 = acc_1_1_425 + p_tmp_3_1_469;
            // ... 31 more
        }
        barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
    }
    p_tmp_4_1_501 = acc_1_1_425 * alpha;
    p_tmp_5_1_533 = C[((64 * N * wg_id_1) + (8 * N * get_local_id(1)) +
                        (128 * wg_id_0) + get_local_id(0))] * beta;
    out[((64 * N * wg_id_1) + (8 * N * get_local_id(1)) + (128 * wg_id_0) +
          get_local_id(0))] = p_tmp_4_1_501 + p_tmp_5_1_533;
    // ... 31 more
}

```

# Coalesced mem accesses

# Blocking / Tiling

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17

# Vectorization

# Builtin math functions

# Blocking / Tiling

...

AMD

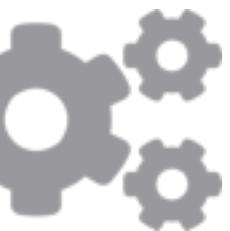
**Nvidia**

ARM

# How do we optimise programs today?



# Program



**Unsustainable to re-optimize for every new architecture ⇒ No performance portability**



# Performance ?

```
for (i = 0; i < N; ++i) {
    for (j = 0; j < N; ++j){
        C[i][j] = 0;
        for (k = 0; k < N; ++k)
            C[i][j] += A[i][k] * B[k][j];
    }
}
```

# GPUs

# Coalesced mem accesses

# Vectorization

# Blocking / Tiling

33  
34

```
36     acc_28 += blockOfA_7 * blockOfB_0;  
37     acc_29 += blockOfA_7 * blockOfB_1;  
38     acc_30 += blockOfA_7 * blockOfB_2;  
39     acc_31 += blockOfA_7 * blockOfB_3;  
40 }
```

```

42
43 }
44
45 C[ 0-8*lidi*M+64*w0-64*w1*N+0*N+lid6]=ac
46 C[16-8*lidi*M+64*w0-64*w1*N+0*N+lid6]=ac
47 C[32-8*lidi*M+64*w0-64*w1*N+0*N+lid6]=ac
48 C[48-8*lidi*M+64*w0-64*w1*N+0*N+lid6]=ac
49 ...
50 ... 24 more statements
51 C[ 0-8*lidi*M+64*w0-64*w1*N+7*N+lid6]=ac
52 C[16-8*lidi*M+64*w0-64*w1*N+7*N+lid6]=ac
53 C[32-8*lidi*M+64*w0-64*w1*N+7*N+lid6]=ac
54 C[48-8*lidi*M+64*w0-64*w1*N+7*N+lid6]=ac
55 }
56 }
```

```

global float *out, int K, int M, int N) {
local float l_tmp_1[512];
local float l_tmp_2[1024];
float acc_1_1_425 = 0.0f;
// ... 31 more
float p_tmp_1_1_457;
// ... 107 more
int wg_id_1 = get_group_id(1);
int wg_id_0 = get_group_id(0);
for (int i = 0; i < (K / 8); i = (1 + i)) {
    int l_id_1
    for (int l
        l_tmp_1[(A[(l
            (B[(l
        )} barrier(CLK
barrier(CLK
for (int l_
    l_tmp_2[((
        (B[(l_
    )} barrier(CLK
barrier(CLK
for (int j
    p_tmp_1_1_
    p_tmp_1_2_
    // ... 6
    p_tmp_2_1_
    p_tmp_2_2_
    // ... 2
    p_tmp_3_1_469 = p_tmp_1_1_457 * p_tmp_2_1_465;
    acc_1_1_425 = acc_1_1_425 + p_tmp_3_1_469;
    // ... 31 more
}
barrier(CLK_LOCAL_MEM_FENCE | CLK_GLOBAL_MEM_FENCE);
}
p_tmp_4_1_501 = acc_1_1_425 * alpha;
p_tmp_5_1_533 = C[((64 * N * wg_id_1) + (8 * N * get_local_id(1)) +
                    (128 * wg_id_0) + get_local_id(0))] *
beta;
out[((64 * N * wg_id_1) + (8 * N * get_local_id(1)) + (128 * wg_id_0) +
      get_local_id(0))] = p_tmp_4_1_501 + p_tmp_5_1_533;
// ... 31 more

```

# Coalesced mem accesses

# Blocking / Tiling

```
barrier(CLK);
barrier(CLK);
for (int j : Blocking / Tiling
```

## Vectorization

## Builtin math functions

# Blocking / Tiling

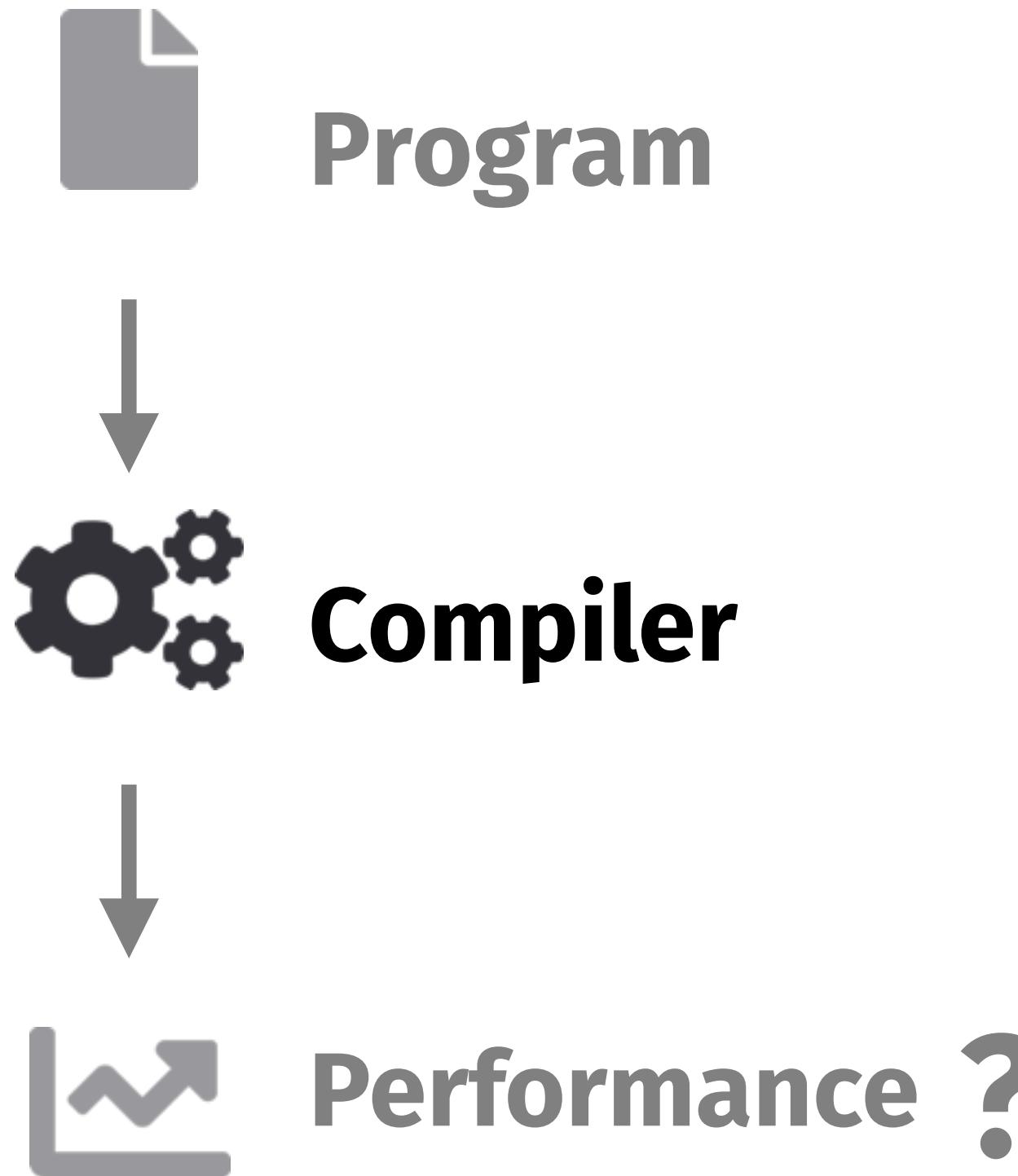
100

AMD

Nvidia

ARM

# How do we optimise programs today?



From the LLVM manual:

#### Code Generation Options

`-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -Og, -O, -O4`

Specify which optimization level to use:

`-O0` Means "no optimization": this level compiles the fastest and generates the most debuggable code.

`-O1` Somewhere between `-O0` and `-O2`.

`-O2` Moderate level of optimization which enables most optimizations.

`-O3` Like `-O2`, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).

`-Ofast` Enables all the optimizations from `-O3` along with other aggressive optimizations that may violate strict compliance with language standards.

`-Os` Like `-O2` with extra optimizations to reduce code size.

`-Oz` Like `-Os` (and thus `-O2`), but reduces code size further.

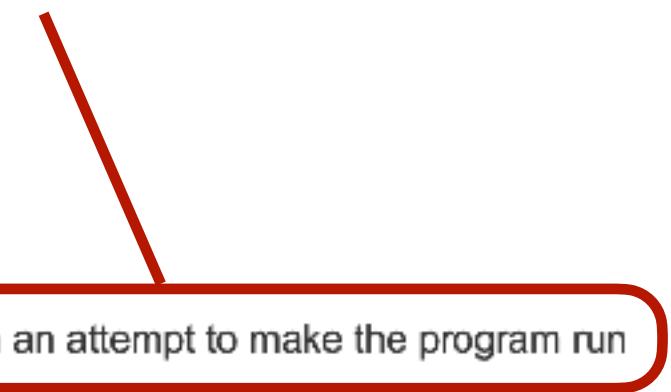
`-Og` Like `-O1`. In future versions, this option might disable different optimizations in order to improve debuggability.

`-O` Equivalent to `-O2`.

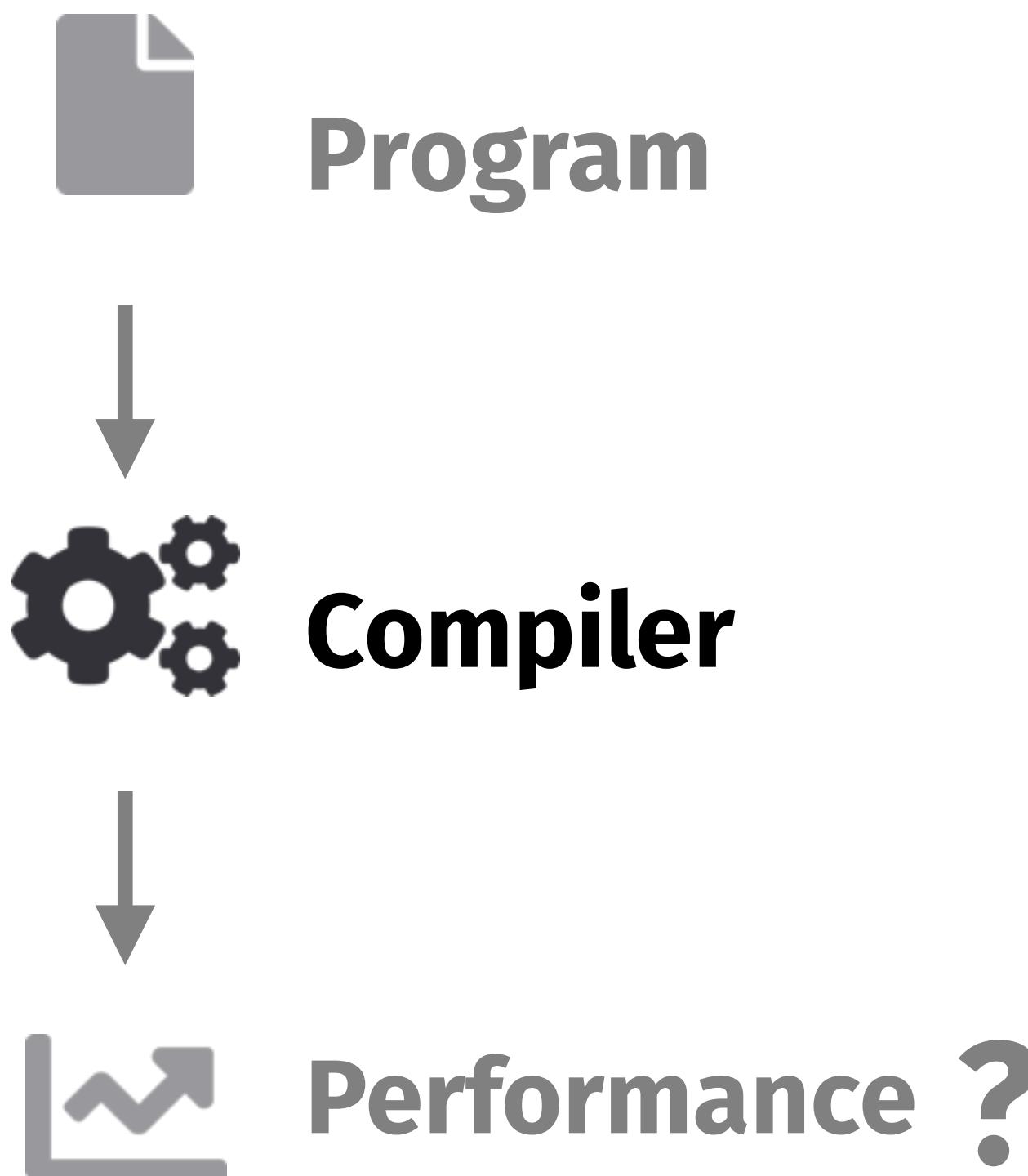
`-O4` and higher

Currently equivalent to `-O3`

"... in an attempt to make the program run faster"



# How do we optimise programs today?



From the LLVM manual:

## Code Generation Options

**-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -Og, -O, -O4**

Specify which optimization level to use:

**-O0** Means "no optimization": this level compiles the fastest and generates the most debuggable code.

**-O1** Somewhere between **-O0** and **-O2**.

**-O2** Moderate level of optimization which enables most optimizations.

**-O3** Like **-O2**, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).

**-Ofast** Enables all the optimizations from **-O3** along with other aggressive optimizations that may violate strict compliance with language standards.

**-Os** Like **-O2** with extra optimizations to reduce code size.

**-Oz** Like **-Os** (and thus **-O2**), but reduces code size further.

**-Og** Like **-O1**. In future versions, this option might disable different optimizations in order to improve debuggability.

**-O** Equivalent to **-O2**.

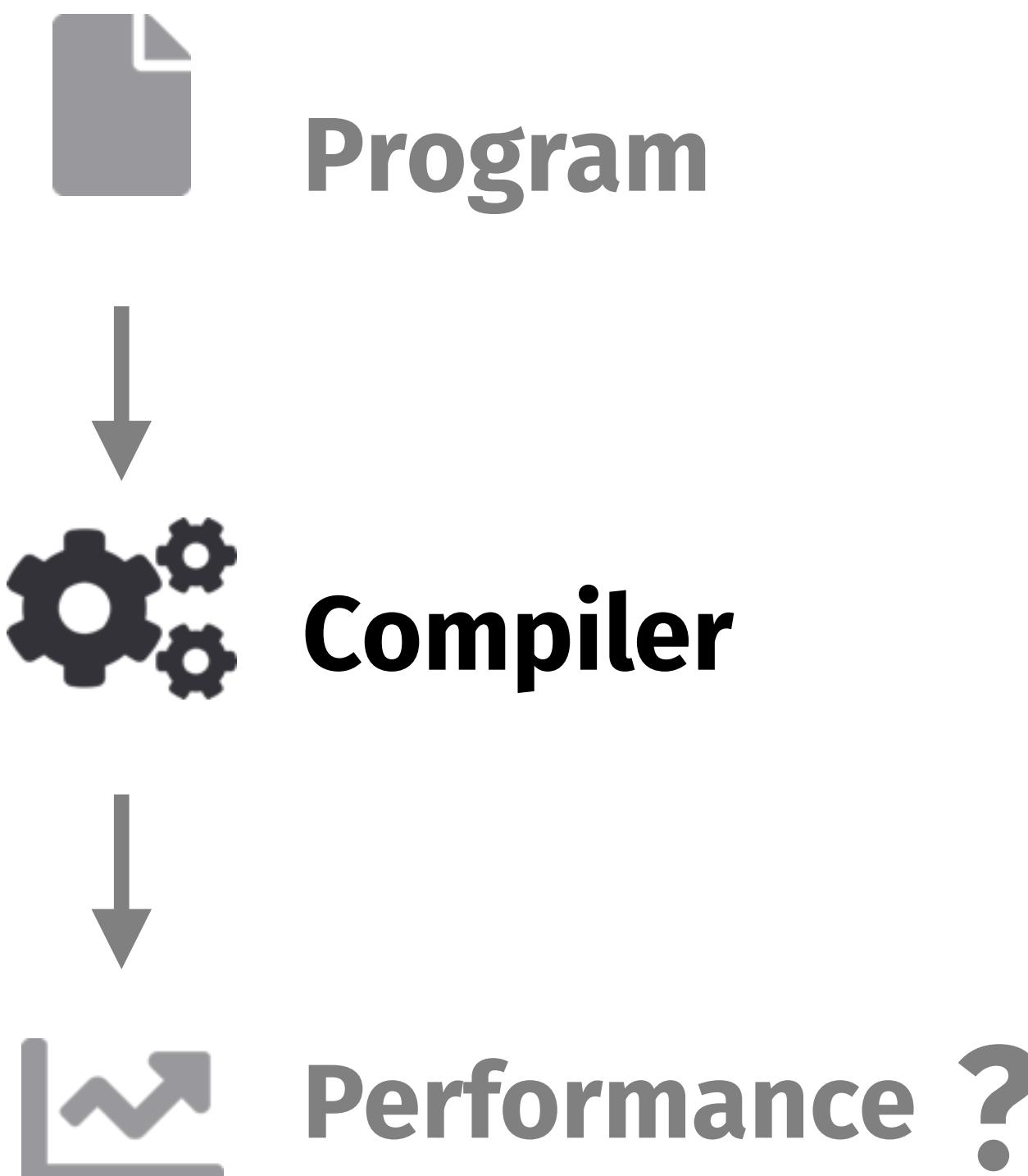
**-O4** and higher

Currently equivalent to **-O3**

"... in an **attempt** to make the program run faster"

**-O3**

# How do we optimise programs today?



From the LLVM manual:

## Code Generation Options

**-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -Og, -O, -O4**

Specify which optimization level to use:

**-O0** Means "no optimization": this level compiles the fastest and generates the most debuggable code.

**-O1** Somewhere between **-O0** and **-O2**.

**-O2** Moderate level of optimization which enables most optimizations.

**-O3** Like **-O2**, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).

**-Ofast** Enables all the optimizations from **-O3** along with other aggressive optimizations that may violate strict compliance with language standards.

**-Os** Like **-O2** with extra optimizations to reduce code size.

**-Oz** Like **-Os** (and thus **-O2**), but reduces code size further.

**-Og** Like **-O1**. In future versions, this option might disable different optimizations in order to improve debuggability.

**-O** Equivalent to **-O2**.

**-O4** and higher

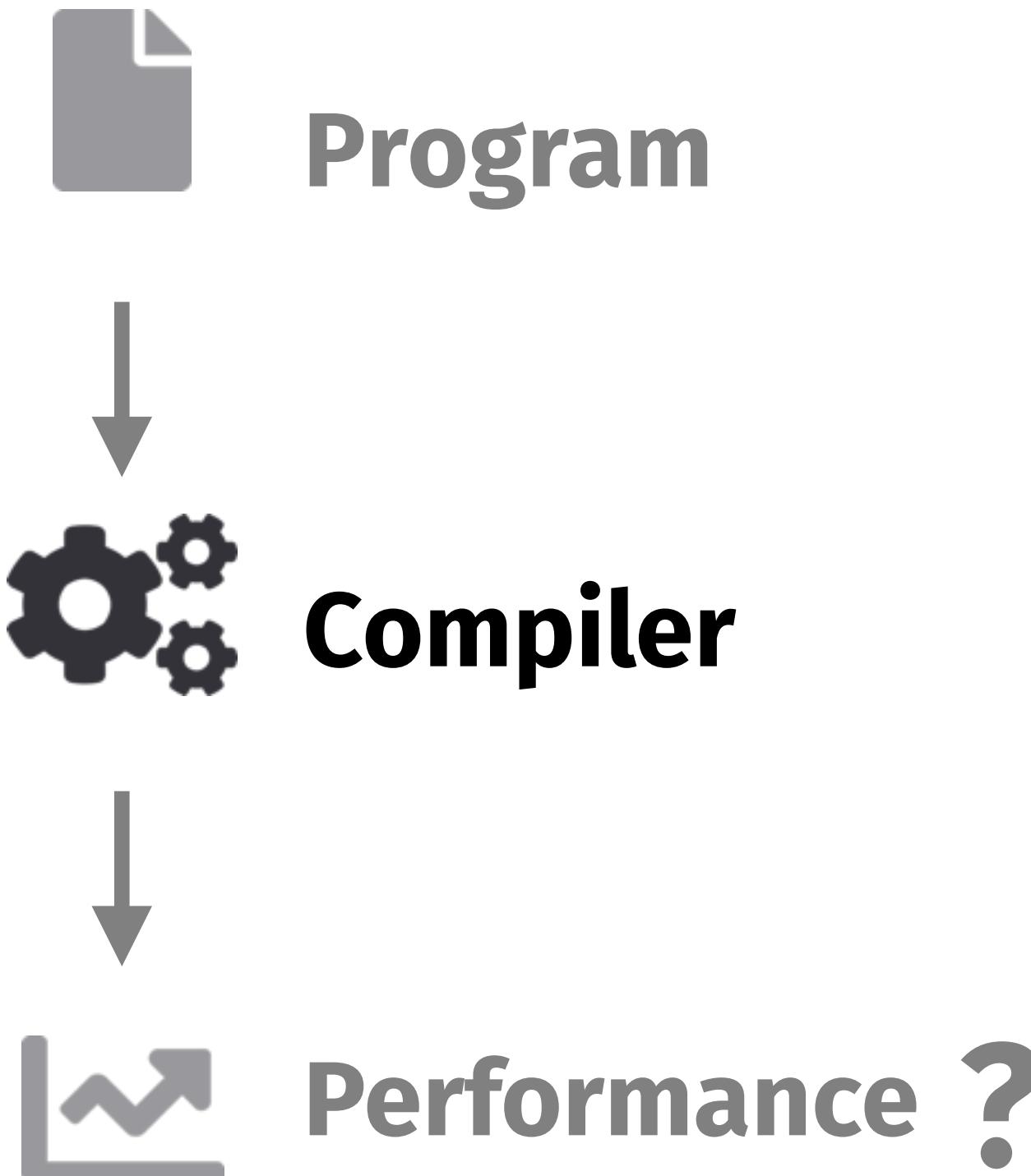
Currently equivalent to **-O3**

"... in an **attempt** to make the program run faster"

**-O3**

```
-targetlibm= -tts -tbaa -scoped-matrices -assumption-cache-tracker -profile-summary-info -forceattrs -callsite-splitting -  
bscp -called-value-propagation -globalopt -domtree -mem2reg -deadargelim -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-free  
-phi-remover instcombine -simplifycfg -basiccg -globals-aa -prune-eh -inline -functionattrs -argumentattr -domtree -sra -basicaa  
-lcssa -early-cse-unsafe -speculative-execution -basicaa -aa -lazy-value-info -jump-threading -correlated-propagation -simpl  
ifycfg -domtree -aggressive instcombine -basicaa -aa -loops -lazy-branch-prob -lazy-block-free -opt-remark-emitter instcombine -liscall  
-shrinkmap -loops -branch-prob -block-freq -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -ppc-memop-opt -basicaa -aa -loops  
-Lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa-veri  
fication -lcssa -basicaa -aa -scalar-evolution -loop-rotate -lcssa -loop-unswitch -simplifycfg -domtree -basicaa -aa -loops -lazy-branch  
-prob -lazy-block-free -opt-remark-emitter instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolution -indvar -loop-ition  
-loop-deletion -loop-unroll -midst-motion -phi-values -basicaa -aa -loops -lazy-branch-prob -lazy-block-free -opt-remark-emitter -gen  
-phi-values -basicaa -aa -memdep -memgcrypt -sccp -demanded-bits -ddce -basicaa -aa -loops -lazy-branch-prob -lazy-block-free -opt-remark  
-emitter instcombine -lazy-value-info -jump-threading -correlated-propagation -basicaa -aa -phi-values -memdep -dss -loops -loop-simpl  
y -lcssa-verification -lcssa -basicaa -aa -scalar-evolution -lcssa -postdomtree -adce -simplifycfg -domtree -basicaa -aa -loops -lazy-branch  
-prob -lazy-block-free -opt-remark-emitter instcombine -barrier -dlm-avail-extern -basicaa -ppc-functionattr -globalopt -globale  
-basiccg -globals-aa -floatlist -domtree -loops -loop-simplify -lcssa-verification -lcssa -basicaa -aa -scalar-evolution -loop-rotate  
-loop-accesses -lazy-branch-prob -lazy-block-free -opt-remark-emitter -loop-distribute -branch-prob -block-freq -scalar-evolution -bas  
icaa -aa -loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -scalar-evo  
lution -aa -loop-accesses -loop-tail-elim -basicaa -aa -lazy-branch-prob -lazy-block-freq -opt-remark-emitter instcombine -simplifycfg -  
domtree -loops -scalar-evolution -basicaa -aa -demanded-bits -lazy-branch-prob -lazy-block-free -opt-remark-emitter -slp-vectorizer -opt  
-remark-emitter instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolution -loop-unroll -lazy-branch-prob -lazy-block-free  
-opt-remark-emitter instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolution -lcssa -alignment-from-assumptions -strip-  
d-prototypes -globalde -constmerge -domtree -loops -branch-prob -block-freq -loop-simplify -lcssa-verification -lcssa -basicaa -aa -  
lcssa-evolution -branch-prob -block-freq -loop-unroll -lazy-branch-prob -lazy-block-free -opt-remark-emitter -instsimplify -div-rem-pairs  
-loopify -verify
```

# How do we optimise programs today?



From the LLVM manual:

#### Code Generation Options

**-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -Og, -O, -O4**

Specify which optimization level to use:

**-O0** Means "no optimization": this level compiles the fastest and generates the most debuggable code.

**-O1** Somewhere between **-O0** and **-O2**.

**-O2** Moderate level of optimization which enables most optimizations.

**-O3** Like **-O2**, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).

**-Ofast** Enables all the optimizations from **-O3** along with other aggressive optimizations that may violate strict compliance with language standards.

**-Os** Like **-O2** with extra optimizations to reduce code size.

**-Oz** Like **-Os** (and thus **-O2**), but reduces code size further.

**-Og** Like **-O1**. In future versions, this option might disable different optimizations in order to improve debuggability.

**-O** Equivalent to **-O2**.

**-O4** and higher

Currently equivalent to **-O3**

"... in an attempt to make the program run faster"

**-O3**

Intel compiler:

**-opt-matmul**

Options **-opt-matmul** and **/Qopt-matmul** tell the compiler to **identify matrix multiplication loop nests (if any) and replace them with a matmul library call** for improved performance. The resulting executable **may get additional performance gain on Intel® microprocessors** than on non-Intel microprocessors.

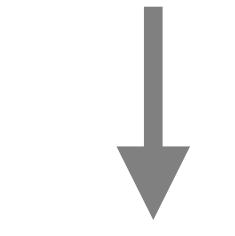
# How do we optimise programs today?



Program



Impossible to understand what is going on in the compiler ⇒ Hard to control optimisations



Performance ?

From the LLVM manual:

#### Code Generation Options

**-O0, -O1, -O2, -O3, -Ofast, -Os, -Oz, -Og, -O, -O4**

Specify which optimization level to use:

**-O0** Means "no optimization": this level compiles the fastest and generates the most debuggable code.

**-O1** Somewhere between **-O0** and **-O2**.

**-O2** Moderate level of optimization which enables most optimizations.

**-O3** Like **-O2**, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).

"... in an attempt to make the program run faster"

**-Os** Like **-O2** with extra optimizations to reduce code size.

**-Oz** Like **-Os** (and thus **-O2**), but reduces code size further.

**-Og** Like **-O1**. In future versions, this option might disable different optimizations in order to improve debuggability.

**-O** Equivalent to **-O2**.

**-O4** and higher

Currently equivalent to **-O3**

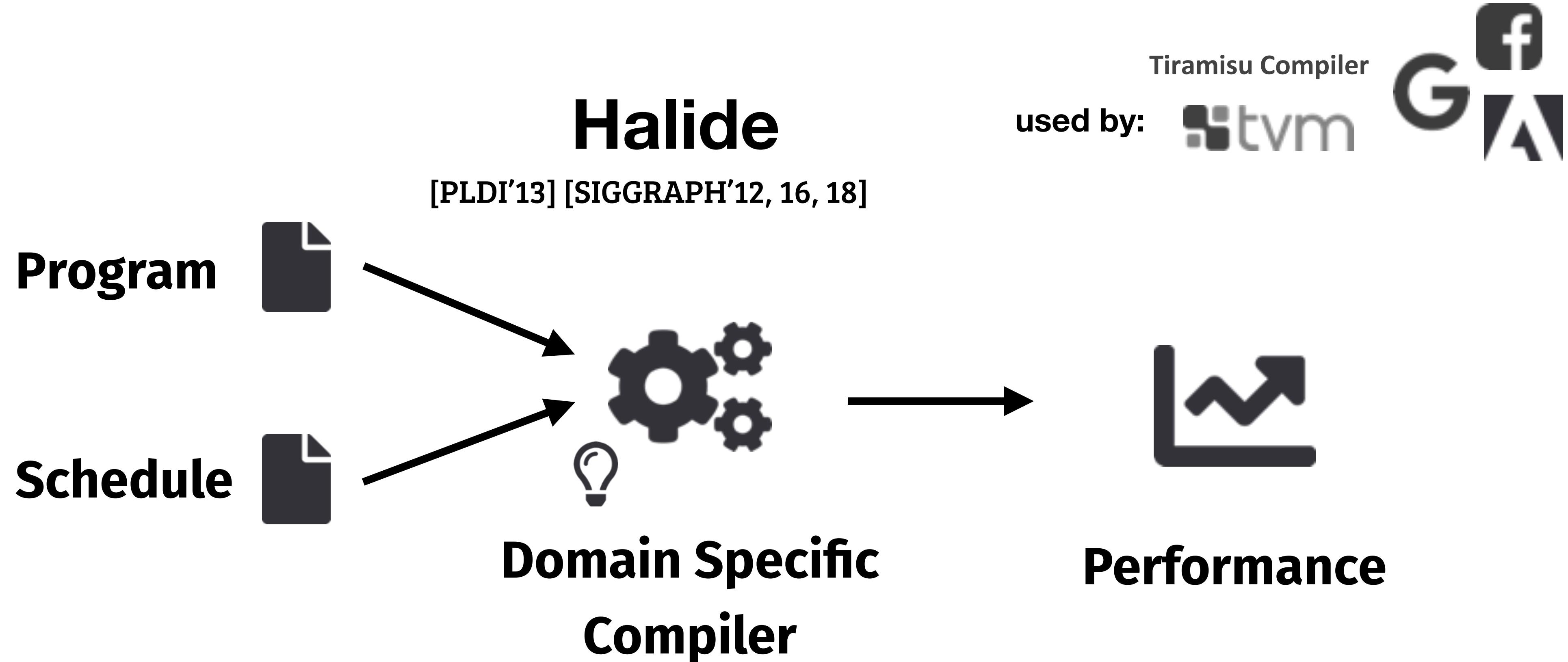
**-O3**

Intel compiler:

**-opt-matmul**

Options **-opt-matmul** and **/Qopt-matmul** tell the compiler to **identify matrix multiplication loop nests (if any) and replace them with a matmul library call** for improved performance. The resulting executable **may get additional performance gain on Intel® microprocessors** than on non-Intel microprocessors.

# Separate Program from Optimisations



Separation in Program and Schedule allows for portable performance

# Halide - Program vs. Schedule

## Program



Domain Specific Language  
embedded in C++

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

## Schedule



C++ API for selecting  
optimisation options

```
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, xio, 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 than functional program!**

# Halide - Program vs. Schedule

## Program



```
Domain  
emb...  
  
Func prod("...")  
RDom r(0, size);  
prod(x, y)  
out(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, xio, 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)
```

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

```
...  
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(x1, 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**

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

# Halide - Program vs. Schedule

Program



**Unintuitive semantics:** Why are these lines repeated

Schedule



Domain Specific Language

C++ API for selecting

```
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);
```

**Unintuitive semantics:** “Update: Get a handle on an update step for the purposes of scheduling it”

```
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, xio, 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)
```

```
.compute_at(prod, rxi)
.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);
```

# Halide - Program vs. Schedule

## Program



**Domain Specific Language  
embedded in C++**

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

## Schedule



**C++ API for selecting  
optimisation options**

```
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, xio, 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);
```

**Schedules are second class citizens.**

**We should write schedules in a proper programming language!**

# ELEVATE

## A programming language for program optimizations

ELEVATE is a functional language that allows to compose individual *program transformations* into larger *optimisation strategies*.

ELEVATE programs are composed of (possibly recursive) functions:

```
def transform(p: Program): RewriteResult[Program] = implementation
```

*Program transformations* are expressed as functions with a particular type:

```
Program → RewriteResult[Program]
```

*Optimisation strategies* are composed functions with the same type

A **RewriteResult** can either be **Success** or **Failure**

A successfully applied transformation contains the transformed program.

A unsuccessfully applied transformation is indicated as failure.

# ELEVATE for optimising LIFT programs

ELEVATE can be used to optimise programs written in different languages

In this talk I focus on programs written in two functional languages:

- the data parallel LIFT programming language
- the **FSmooth** language used for automatic differentiation

We intend to use ELEVATE for additional high-level languages like TensorFlow

LIFT: More info at <http://www.lift-project.org> and papers at: [ICFP 2015] [CASES 2016] [CGO 2017 & 2018]

**FSmooth**: [ICFP 2019]



[ICFP'15]

DSL

DSL

DSL

High-Level IR

Explore Optimizations  
by rewriting

[GPGPU'16]  
[CASES'16]

Low-Level Program

Code Generation  
[CGO'17]

Multicore  
CPU

GPU

HPC  
Mobile

Xeon  
Phi

KNC  
KNL

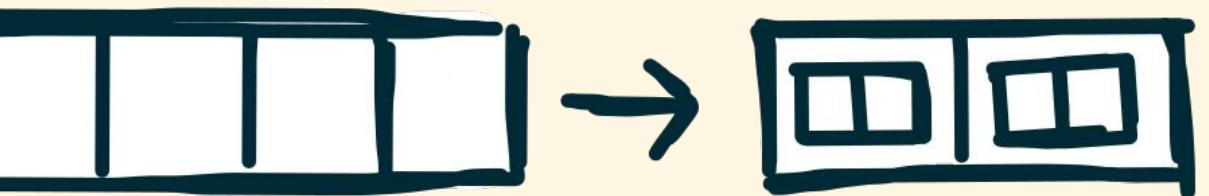
...

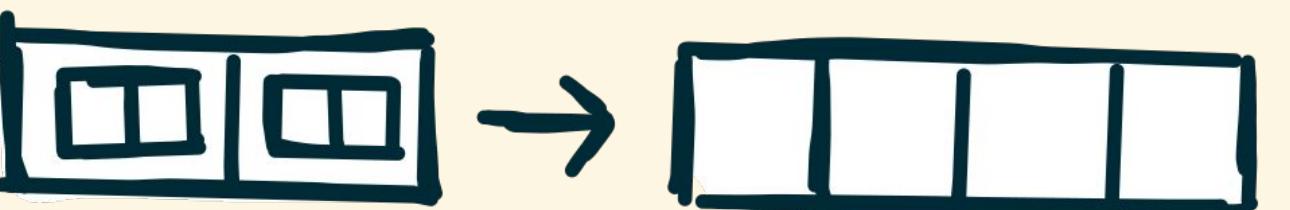
Hardware

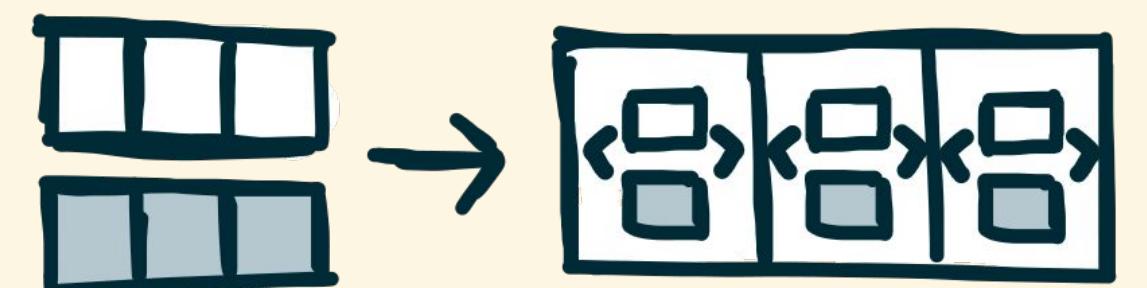
# LIFT'S HIGH-LEVEL PRIMITIVES

*map*( $\square \rightarrow \square$ ) 

*reduce*( $\oplus$ ) 

*split(n)* 

*join* 

*zip* 

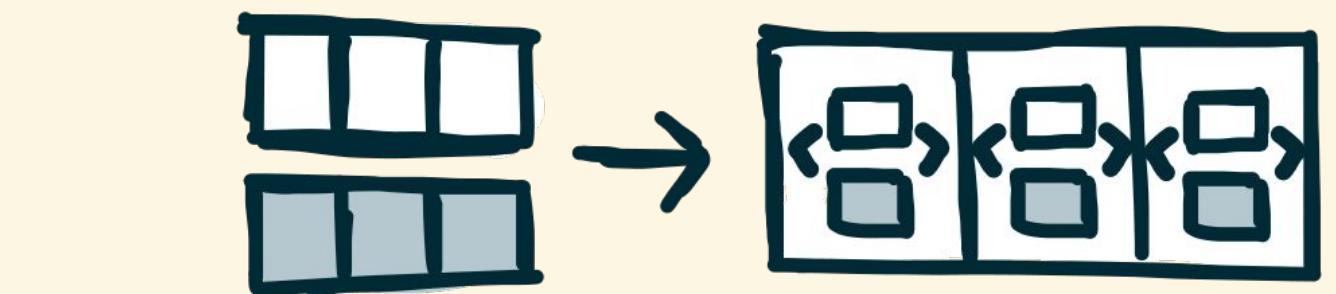
# LIFT'S HIGH-LEVEL PRIMITIVES

*map*( $\square \rightarrow \square$ ) 

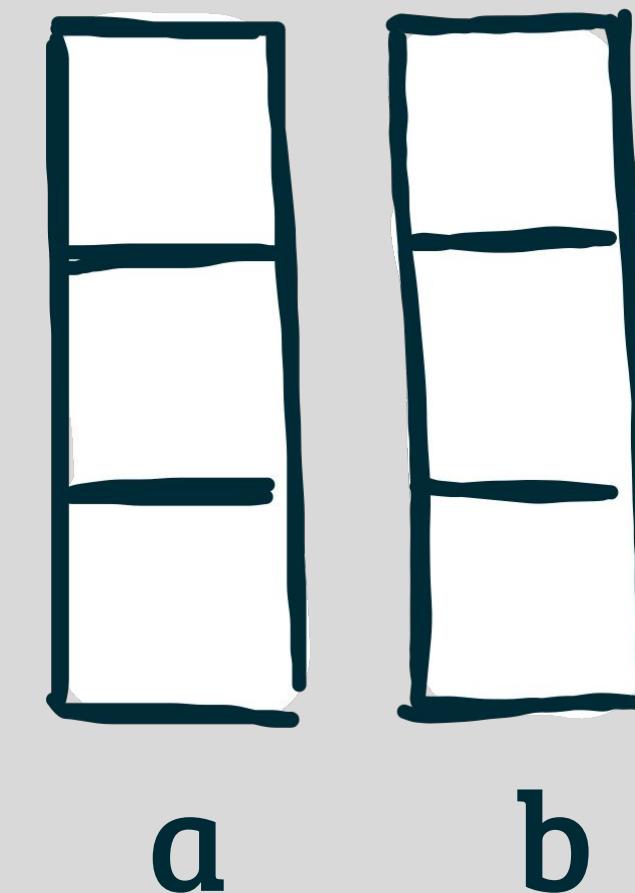
*reduce*( $\oplus$ ) 

*split(n)* 

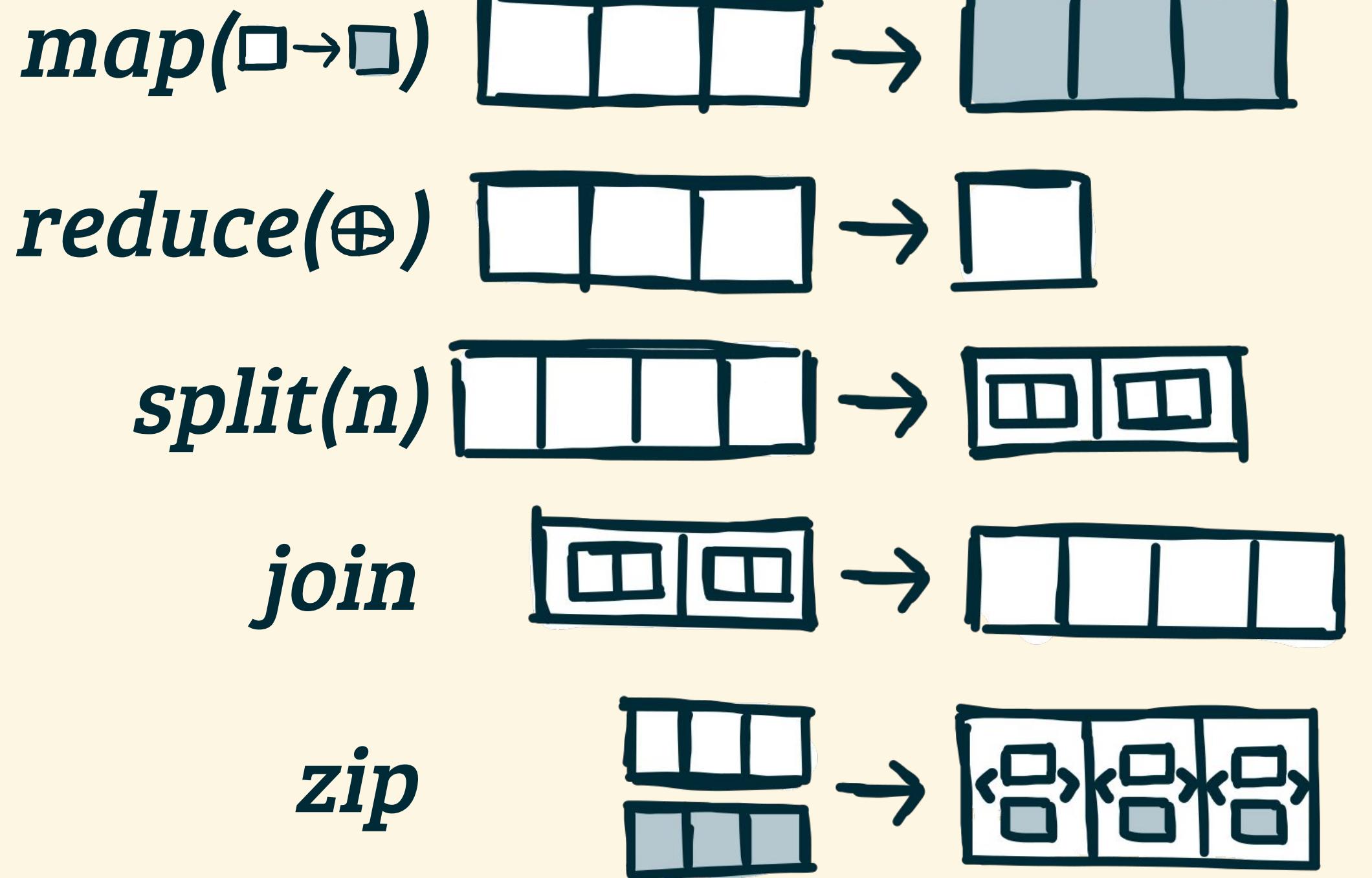
*join* 

*zip* 

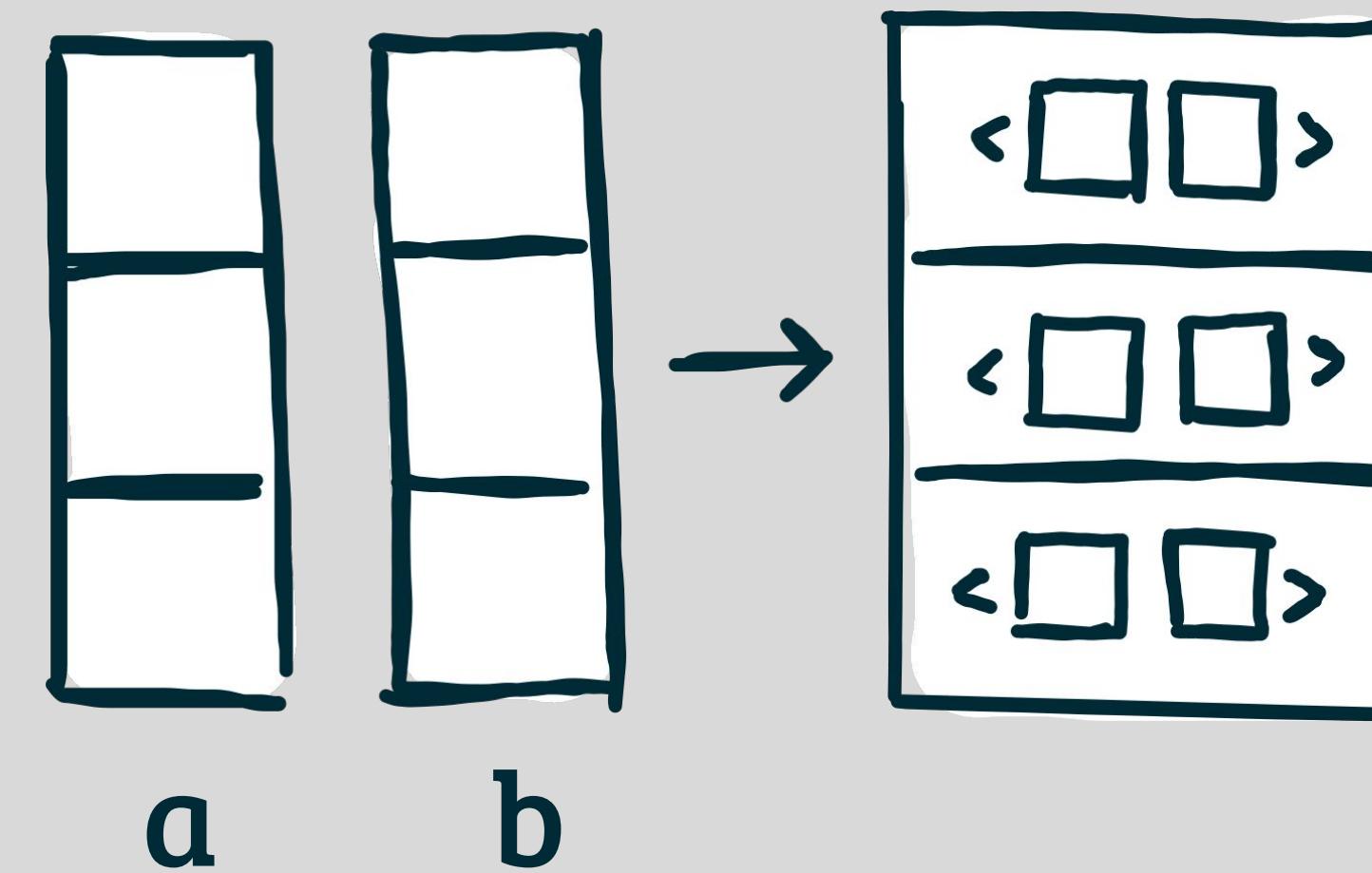
dotproduct.lift



# LIFT'S HIGH-LEVEL PRIMITIVES



dotproduct.lift



*zip(a, b)*

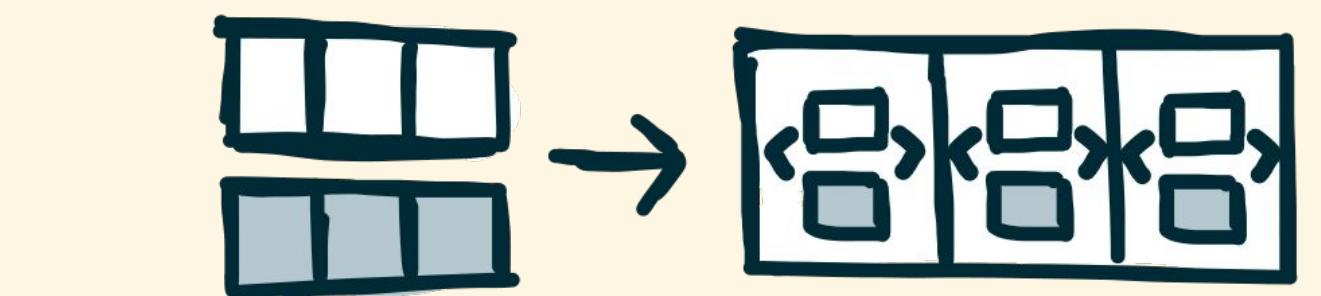
# LIFT'S HIGH-LEVEL PRIMITIVES

*map*( $\square \rightarrow \square$ ) 

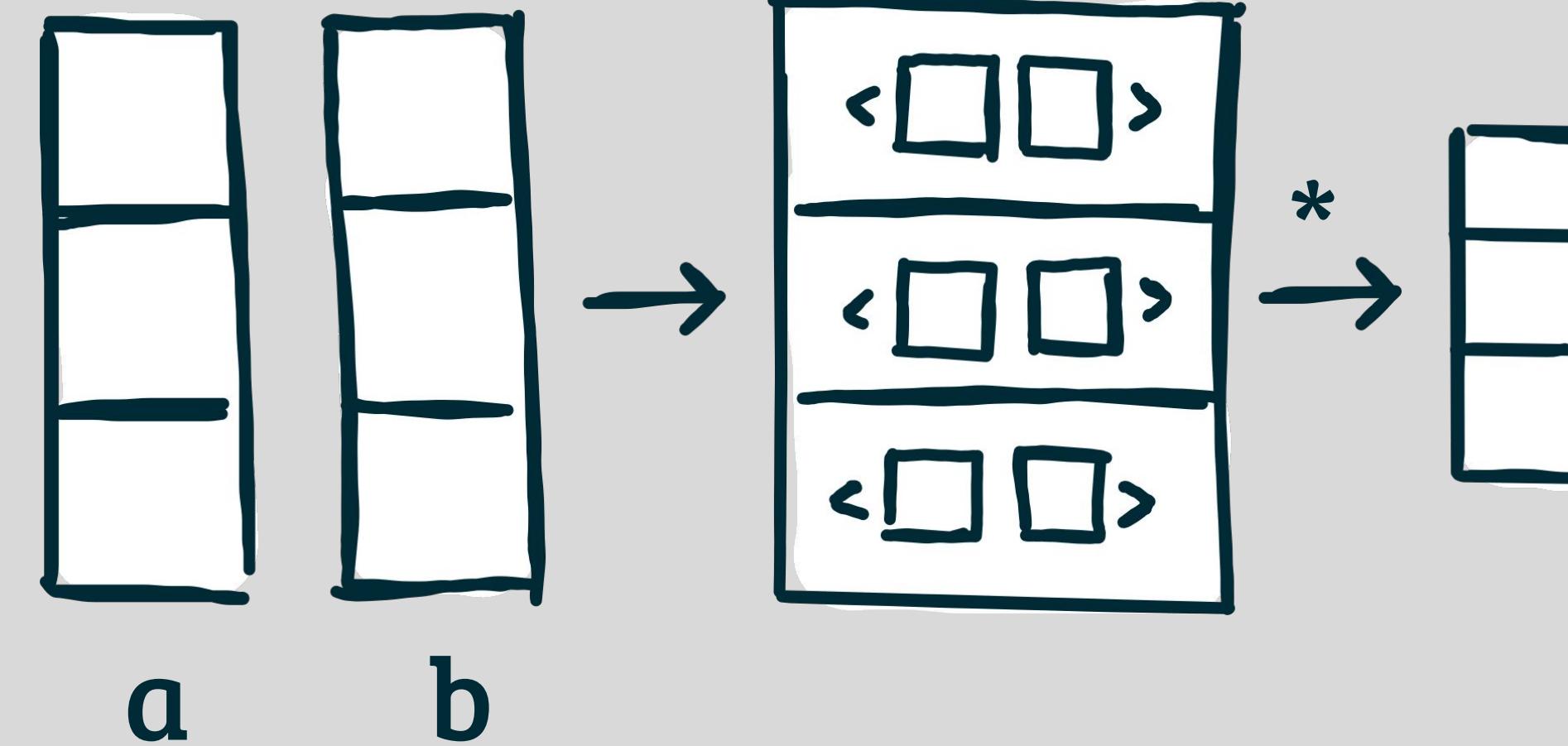
*reduce*( $\oplus$ ) 

*split(n)* 

*join* 

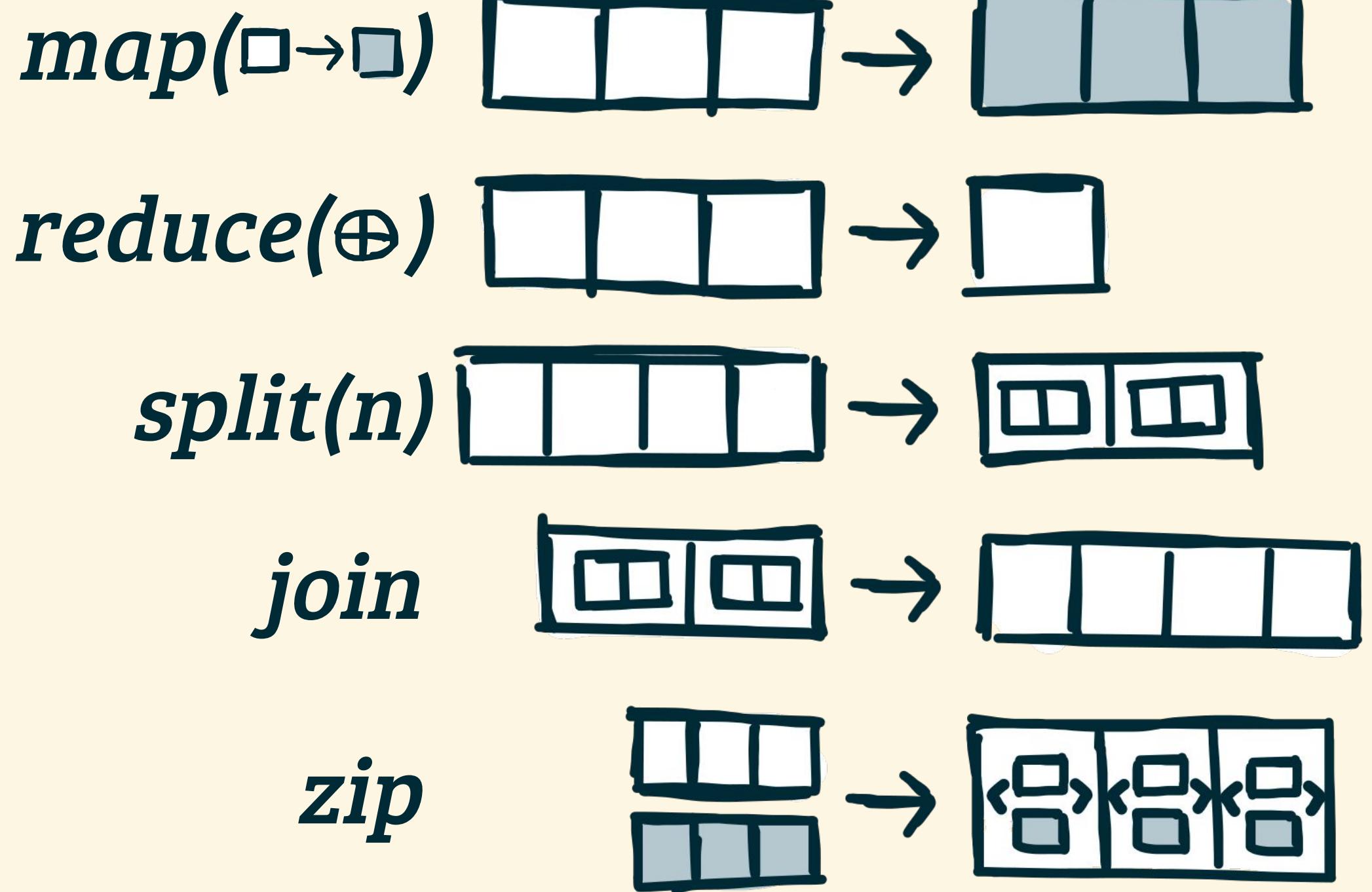
*zip* 

dotproduct.lift

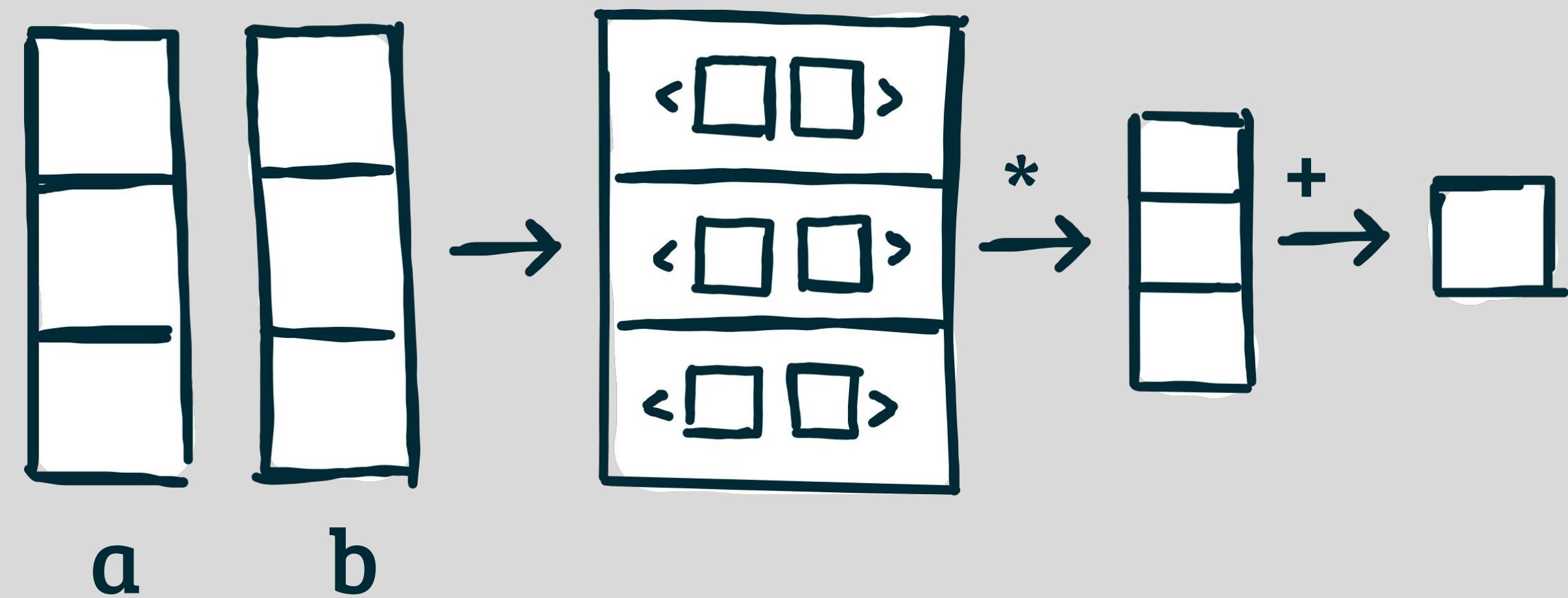


*map*(\* , *zip*( $a, b$ ))

# LIFT'S HIGH-LEVEL PRIMITIVES

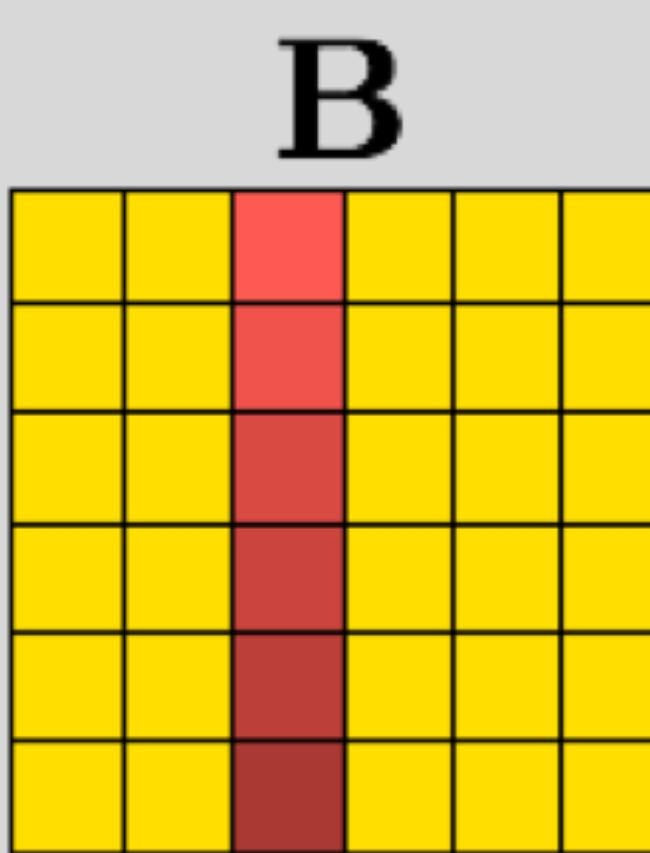
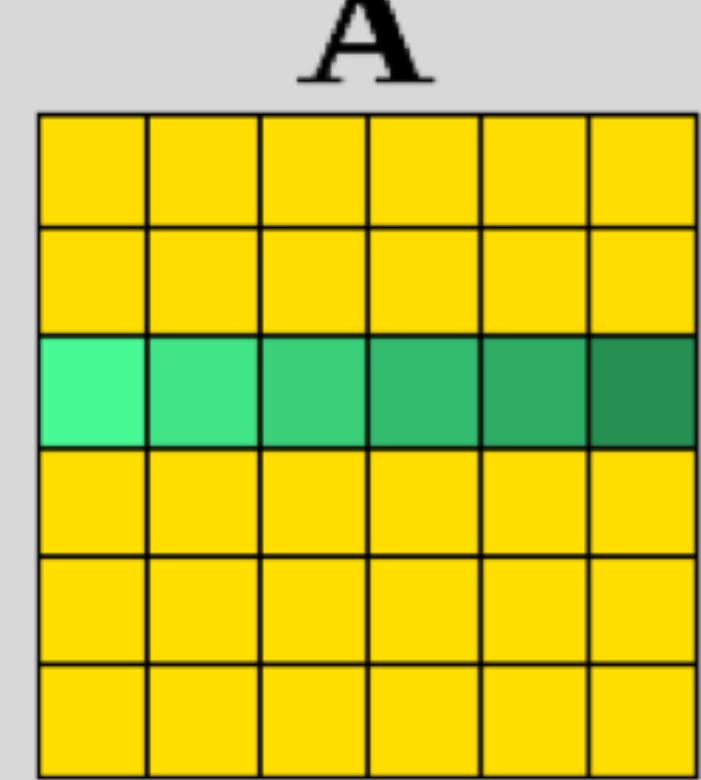
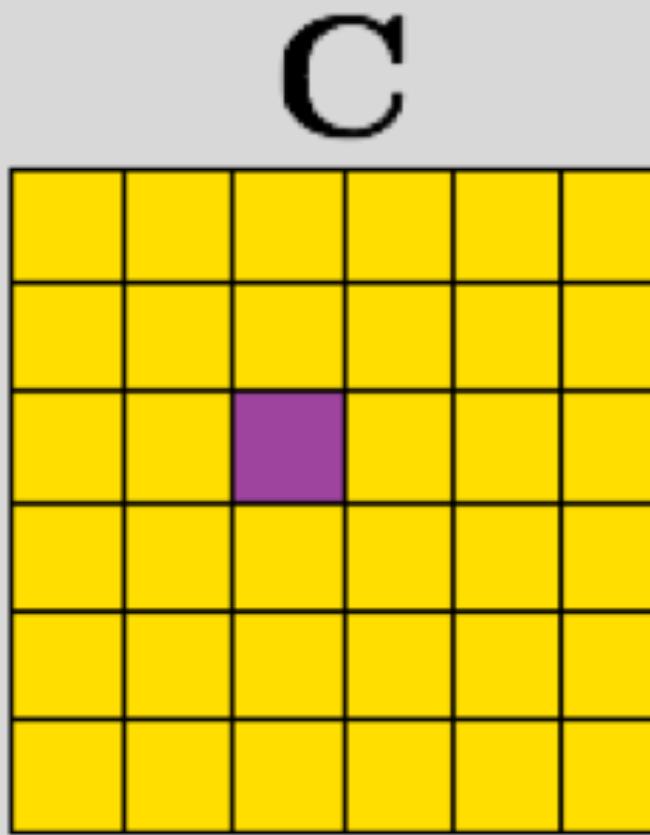


dotproduct.lift



*reduce*( $+$ ,  $0$ , *map*(\* , *zip*(a,b)))

# LIFT'S HIGH-LEVEL PRIMITIVES

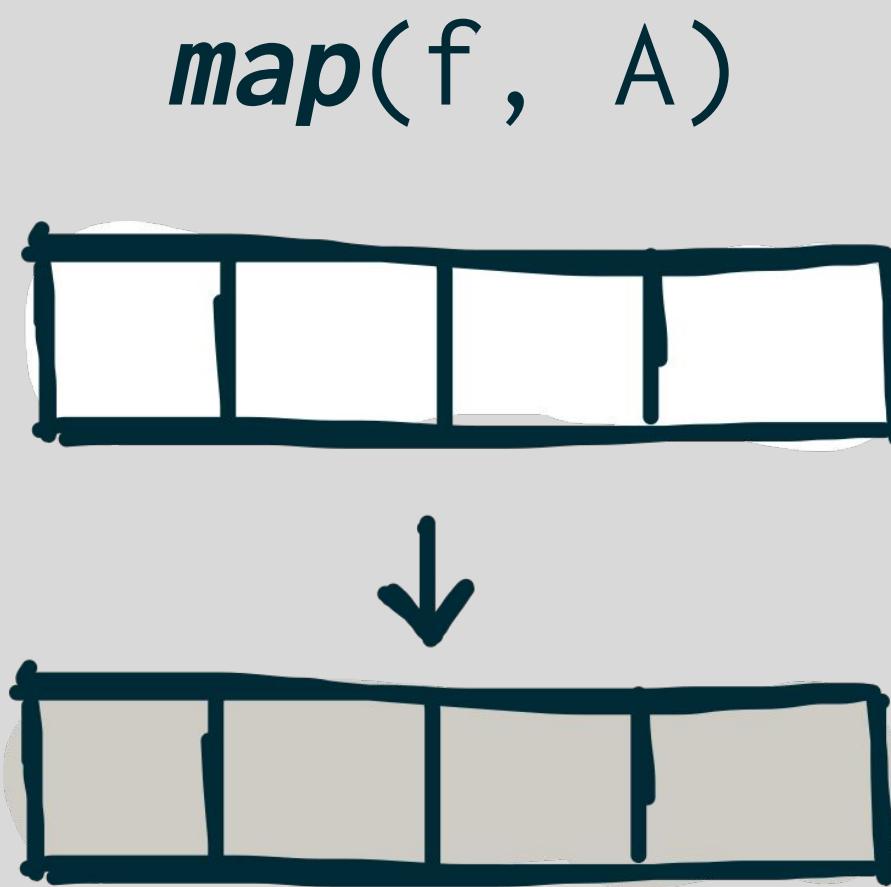


**matrixMult.lift**

```
map( $\lambda$  rowA  $\mapsto$ 
    map( $\lambda$  colB  $\mapsto$ 
        dotProduct(rowA, colB)
        , transpose(B))
    , A)
```

# **IMPLEMENTATION CHOICES AS REWRITE RULES**

## Divide & Conquer



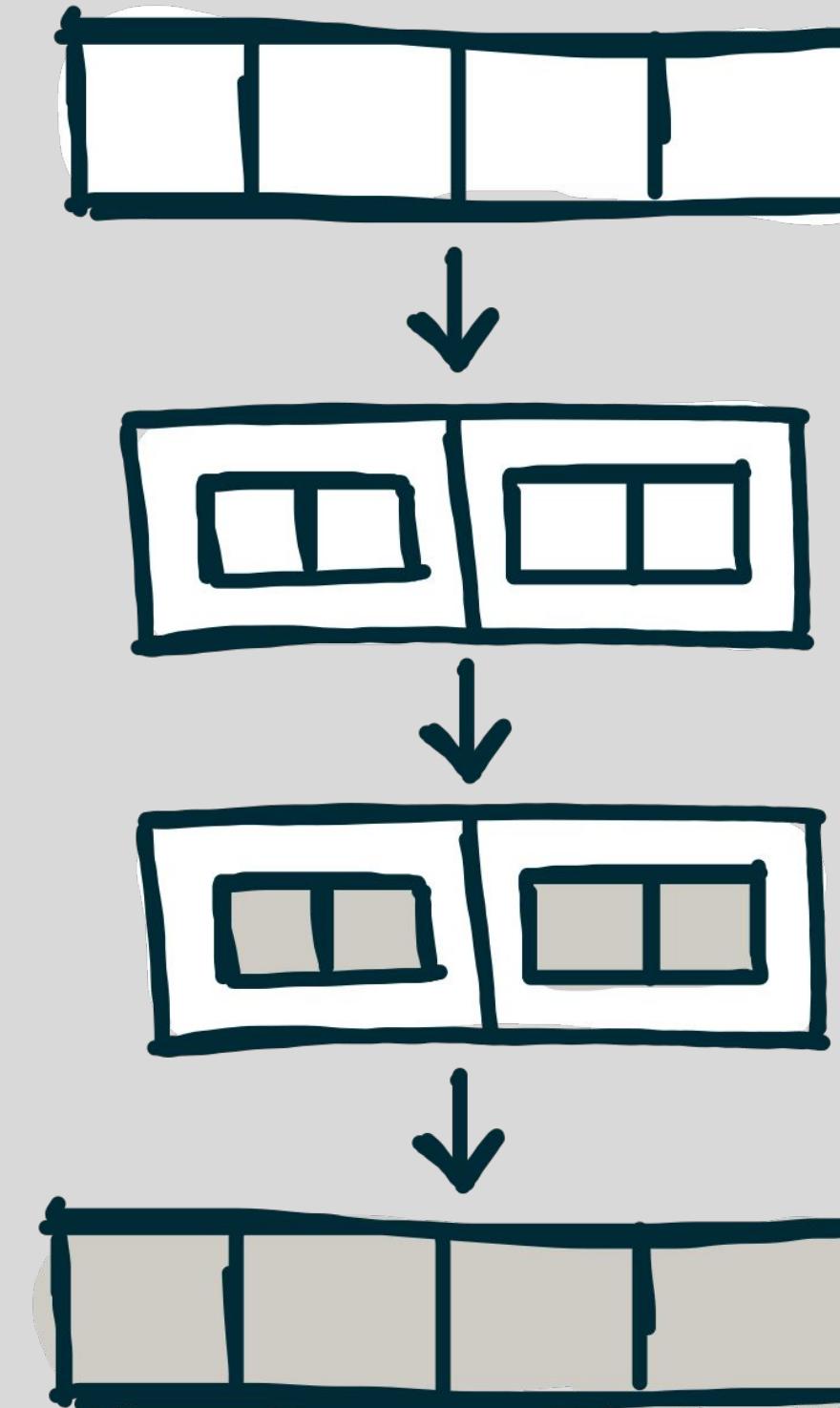
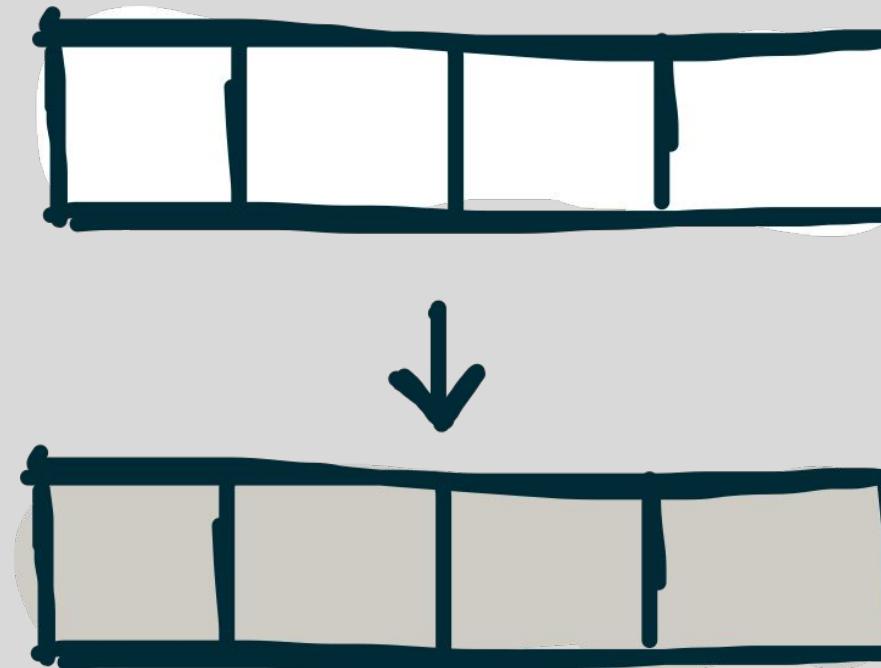
# IMPLEMENTATION CHOICES AS REWRITE RULES

## Divide & Conquer

*map(f, A)*



*join(map(map(f),  
split(n, A)))*



# OPTIMIZATIONS AS MACRO RULES

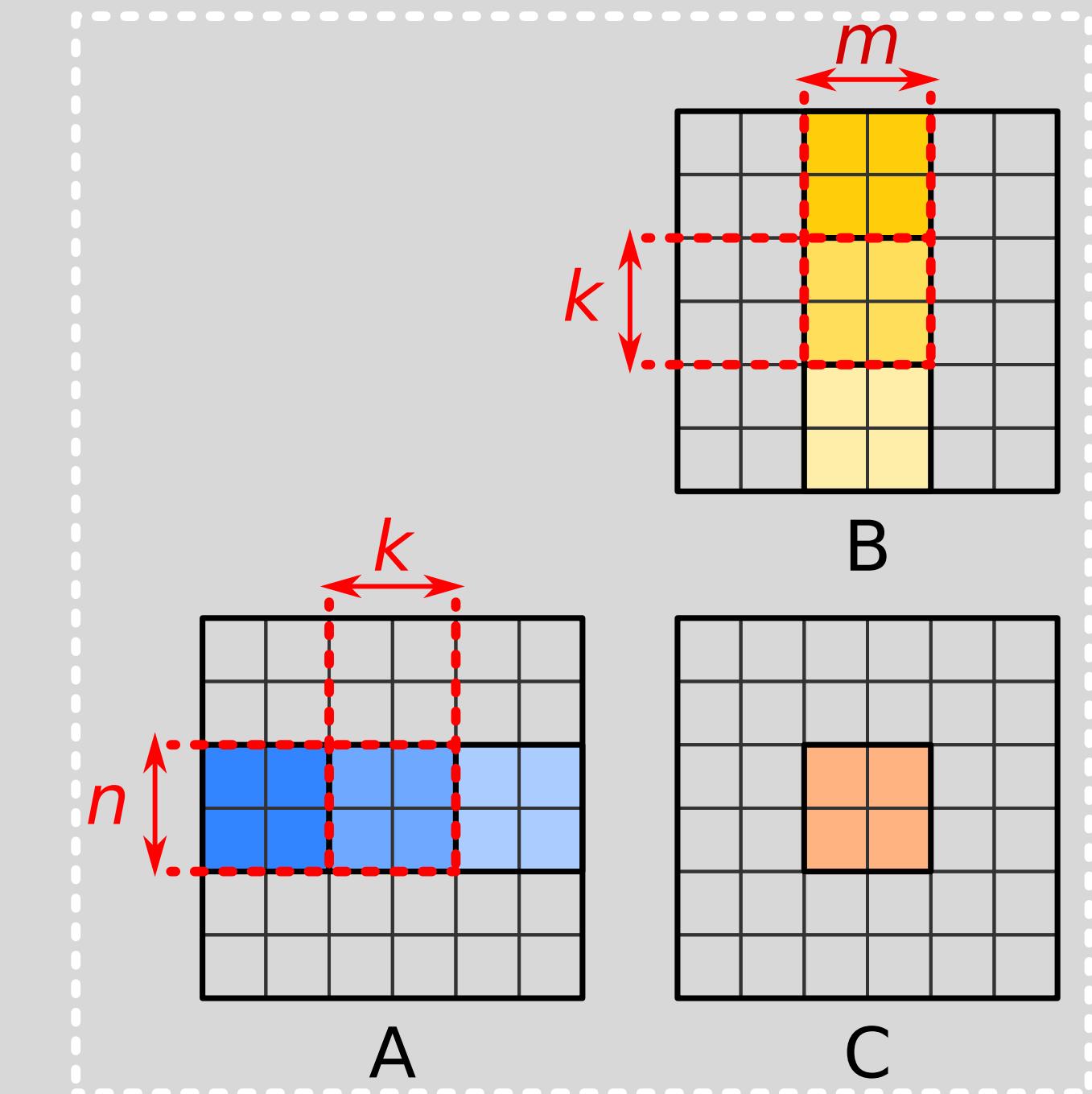
## 2D Tiling

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]

# OPTIMIZATIONS AS MACRO RULES

## 2D Tiling

Naïve matrix multiplication

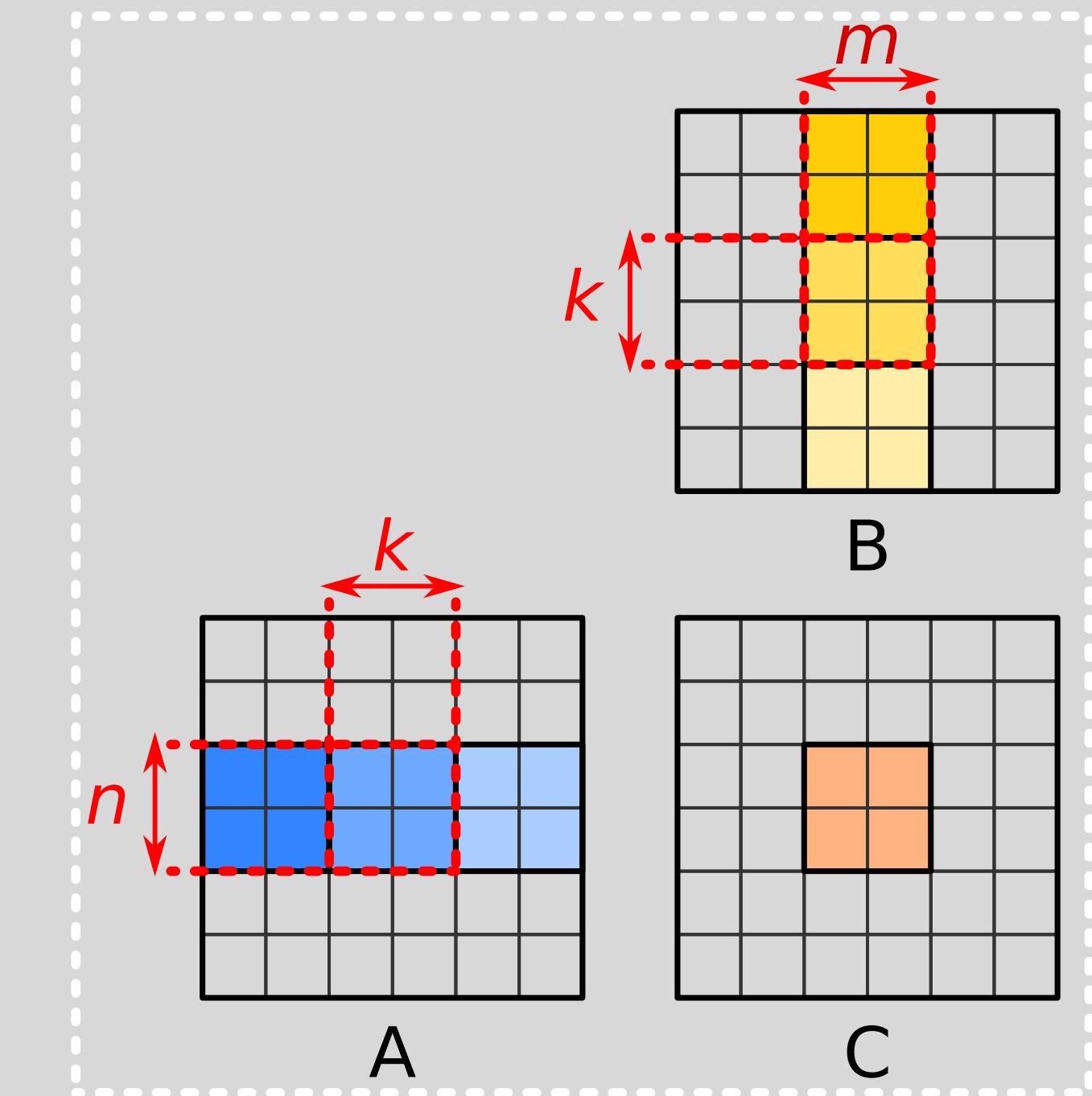
```
1 map(λ arow .  
2   map(λ bcol .  
3     reduce(+, 0) ○ map(×) ○ zip(arow, bcol)  
4     , transpose(B))  
5   , A)
```

Many rewrite rules applied here



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]

# [GPGPU'16] Presentation Slides

# Register Blocking

The diagram illustrates the transformation of a MapReduce pipeline from a flat structure to a block-structured one. It consists of two main parts connected by a large grey arrow pointing right.

**Left Side (Flat Structure):**

- `Join() ∘ Map(rowsA →`
- `Map(rowA →` (highlighted in yellow)
- `Map(colB →` (highlighted in yellow))
- `Reduce(+) ∘ Map(*)`
- `\$ Zip(rowA, colB)`
- `) ∘ Transpose() \$ B`
- `) \$ rowsA`
- `) ∘ Split(blockFactor) \$ A`

**Right Side (Block-Structured Structure):**

- `Join() ∘ Map(rowsA →`
- `Transpose() ∘ Map(colB →` (highlighted in yellow))
- `Map(rowA →` (highlighted in yellow))
- `Reduce(+) ∘ Map(*)`
- `\$ Zip(rowA, colB)`
- `) \$ rowsA` (highlighted in yellow)
- `) ∘ Transpose() \$ B` (highlighted in yellow)
- `) ∘ Split(blockFactor) \$ A`

## Register Blocking

```

Iap(rowsA →
ose() ∘ Map( $\overrightarrow{\text{col } B}$ ) →
 $\overrightarrow{\text{row } A}$  →
duce(+) ∘ Map(*)
3 Zip( $\overrightarrow{\text{row } A}$ ,  $\overrightarrow{\text{col } B}$ )
wsA
nspose() § B
lockFactor) § A

```

# gister Blocking

The diagram illustrates the transformation of a MapReduce pipeline from a sequential style to a functional style using F# combinators. The left side shows the sequential code, and the right side shows the functional code with combinators highlighted in yellow.

**Sequential Code (Left):**

- $wsA \mapsto$
- $\text{Map}(\overrightarrow{\text{col } B}) \mapsto$
- $\circ \text{Reduce}((\overrightarrow{\text{acc}}, \text{next}) \mapsto$
- $\circ \text{Zip}(\overrightarrow{\text{acc}}, \text{next})$
- $\text{sel}() \circ \text{Map}(\overrightarrow{\text{row } A} \mapsto$
- $\text{Zip}(\overrightarrow{\text{row } A}, \overrightarrow{\text{col } B})$
- $(\$ \mathbf{B}$
- $\text{actor}) \$ \mathbf{A}$
- $\text{Map}(\text{Map}(f)) \Rightarrow$
- $\text{Transpose}() \circ \text{Map}(\text{Map}(f)) \circ \text{Transpose}()$

**Functional Code (Right):**

- $Join() \circ \text{Map}(\text{rows } A \mapsto$
- $\text{Transpose}() \circ \text{Map}(\overrightarrow{\text{col } B} \mapsto$
- $\text{Transpose}() \circ \text{Reduce}((\overrightarrow{\text{acc}}, \text{next}) \mapsto$
- $\text{Map}(+) \$ \text{Zip}(\overrightarrow{\text{acc}}, \text{next})$
- $) \circ \text{Transpose}()$
- $\circ \text{Transpose}() \circ \text{Map}(\text{pair} \mapsto$
- $\text{Map}(x \mapsto x * \text{pair}.1) \$ \text{pair}.0$
- $) \$ \text{Zip}(\text{Transpose}() \$ \text{rows } A, \overrightarrow{\text{col } B})$
- $) \circ \text{Transpose}() \$ \mathbf{B}$
- $\circ \text{Split}(\text{blockFactor}) \$ \mathbf{A}$

# register Blocking

```

 $\overrightarrow{colB} \mapsto$ 
 $uce((\overrightarrow{acc}, \overrightarrow{next}) \mapsto$ 
 $\overrightarrow{acc}, \overrightarrow{next}))$ 

 $ap(pair \mapsto$ 
 $ir..1) \$ pair..0$ 
 $) \$ rowsA, \overrightarrow{colB})$ 

```

# Register Blocking

The diagram illustrates the transformation of a MapReduce pipeline from a sequential style to a functional style using the `Map` class.

**Left Side (Sequential Style):**

```

Join() o Map(rowsA →
Transpose() o Map( $\vec{colB}$ ) →
Map( $\vec{rowA}$ ) →
Reduce(+) o Map(*) →
\$ Zip( $\vec{rowA}, \vec{colB}$ ) →
) \$ rowsA →
) o Transpose() \$ B →
) o Split(blockFactor) \$ A
  
```

**Right Side (Functional Style):**

```

Join() o Map(rowsA →
Transpose() o Map( $\vec{colB}$ ) →
Map( $\vec{rowA}$ ) →
Reduce(+) →
) o Map( $\vec{rowA}$ ) →
Map(*) \$ Zip( $\vec{rowA}, \vec{colB}$ ) →
) \$ rowsA →
) o Transpose() \$ B →
) o Split(blockFactor) \$ A
  
```

A large grey arrow points from left to right, indicating the transformation. The `Map` class is highlighted in yellow in both the original and transformed code snippets.

**Bottom:**

$$Map(f \circ g) \Rightarrow Map(f) \circ Map(g)$$

## Register Blocking

```

Map(rowsA ↪
  ↮ Map(colB ↪
    ↮ Map(transpose() $ Zip(rowA, colB))
    ↮ Map(reduce(+))
    ↮ Map(blockFactor) $ A
  )
)

```

# gister Blocking

The diagram illustrates the simplification of a complex sequence of operations into a more concise form. On the left, a sequence of operations is shown:

- $\text{rowsA} \mapsto$
- $\circ \text{Map}(\overrightarrow{\text{colB}} \mapsto$
- $\circ () \circ \text{Reduce}((\overleftarrow{\text{acc}}, \overrightarrow{\text{next}}) \mapsto$
- $\circ \text{Zip}(\overleftarrow{\text{acc}}, \overrightarrow{\text{next}})$
- $\circ \text{ose}()$
- $\circ \text{use}() \circ \text{Map}(\text{pair} \mapsto$
- $\rightarrow x \times \text{pair}..1) \$ \text{pair}..0$
- $\circ \text{Transpose}() \$ \text{rowsA}, \overrightarrow{\text{colB}}$
- $\circ (\$ \mathbf{B}$
- $\circ \text{factor}) \$ \mathbf{A}$

An arrow points from this sequence to the right, where it is transformed into a simplified sequence:

- $\text{Join}() \circ \text{Map}(\text{rowsA} \mapsto$
- $\circ \text{Transpose}() \circ \text{Map}(\overrightarrow{\text{colB}} \mapsto$
- $\circ \text{Transpose}() \circ \text{Reduce}((\overleftarrow{\text{acc}}, \overrightarrow{\text{next}}) \mapsto$
- $\circ \text{Map}(+) \$ \text{Zip}(\overleftarrow{\text{acc}}, \overrightarrow{\text{next}})$
- $\circ \text{Map}(\text{pair} \mapsto$
- $\circ \text{Map}(x \mapsto x \times \text{pair}..1) \$ \text{pair}..0$
- $\circ \$ \text{Zip}(\text{Transpose}() \$ \text{rowsA}, \overrightarrow{\text{colB}})$
- $\circ \text{Transpose}() \$ \mathbf{B}$
- $\circ \text{Split}(\text{blockFactor}) \$ \mathbf{A}$

# register Blocking

```

 $\overrightarrow{colB} \mapsto$ 
 $uce((\overrightarrow{acc}, \overrightarrow{next}) \mapsto$ 
 $\overrightarrow{c}, \overrightarrow{next})$ 

ir..1) $ pair..0
() $ rowsA,  $\overrightarrow{colB}$ )

```

# Register Blocking

The diagram illustrates the transformation of a MapReduce pipeline. On the left, a sequential pipeline is shown:

$$\begin{aligned}
 &Join() \circ Map(rowsA \mapsto \\
 &\quad Transpose() \circ Map(\overrightarrow{\text{col } B} \mapsto \\
 &\quad \quad Map( \\
 &\quad \quad \quad Reduce(+)) \\
 &\quad ) \circ Map(\overrightarrow{\text{row } A} \mapsto \\
 &\quad \quad Map(*) \$ Zip(\overrightarrow{\text{row } A}, \overrightarrow{\text{col } B}) \\
 &\quad ) \$ rowsA \\
 &\quad ) \circ Transpose() \$ \mathbf{B} \\
 &\quad ) \circ Split(blockFactor) \$ \mathbf{A}
 \end{aligned}$$

On the right, the transformed parallel pipeline is shown, separated by a large arrow pointing from left to right:

$$\begin{aligned}
 &Join() \circ Map(rowsA \mapsto \\
 &\quad Transpose() \circ Map(\overrightarrow{\text{col } B} \mapsto \\
 &\quad \quad Map( \\
 &\quad \quad \quad Reduce((\overrightarrow{\text{ac}} \circ \\
 &\quad \quad \quad Map(+) \$ Zip(\overrightarrow{\text{ac}}, \overrightarrow{\text{ex}}))) \\
 &\quad ) \circ Transpose() \circ Map(\overrightarrow{\text{row } A} \mapsto \\
 &\quad \quad Map(*) \$ Zip(\overrightarrow{\text{row } A}, \overrightarrow{\text{col } B}) \\
 &\quad ) \$ rowsA \\
 &\quad ) \circ Transpose() \$ \mathbf{B} \\
 &\quad ) \circ Split(blockFactor) \$ \mathbf{A}
 \end{aligned}$$

Below the parallel stages, the resulting output is shown:

$$\begin{aligned}
 &Map(Reduce(f)) \Rightarrow \\
 &Transpose() \circ Reduce(Map(f) \circ Zip())
 \end{aligned}$$

# Register Blocking

```
(rowsA ↠
  ) o Map( $\overrightarrow{colB}$ ) ↠
  se() o Reduce(( $\overrightarrow{acc}, \overrightarrow{next}$ ) ↠
    -) \$ Zip( $\overrightarrow{acc}, \overrightarrow{next}$ )
  pose() o Map( $\overrightarrow{rowA}$ ) ↠
  \$ Zip( $\overrightarrow{rowA}, \overrightarrow{colB}$ )
  1
  pose() \$ B
  :Factor) \$ A

  Map(Map(f)) ⇒
  Transpose() o Map(Map(f)) o Transpose()
```

# gister Blocking

# register Blocking

```

    >
    \overrightarrow{colB} \mapsto
    \overleftarrow{uce}((\overrightarrow{acc}, \overrightarrow{pair}) \mapsto
    \overrightarrow{cc},
    pair..1) \$ pair..0)
    () \$ rowsA, \overrightarrow{colB})

```

# 80 rewrite steps!

# Combining Optimisations

```

Join() ∘ Map((p179 ↣
Transpose() ∘ Join() ∘ Map((p70 ↣
Transpose() ∘ Join() ∘ Map((p20 ↣
Transpose() ∘ Map((p65 ↣
Transpose()(p65)
)) ∘ Transpose()(p20)
)) ∘ Transpose() ∘ Reduce((p75, p0 ↣
Map((p164 ↣
Join() ∘ Map((p81 ↣
Reduce((p136, p90 ↣
Map((p163 ↣
Get(0)(p163) + Get(1)(p163) * Get(1)(p90)
)) ∘ Zip(2)(p136, Get(0)(p90))
))(Get(0)(p81), Zip(2)(Transpose() ∘ Get(1)(p164), Get(1)(p81)))
)) ∘ Zip(2)(Get(0)(p164), Get(1)(p0))
)) ∘ Zip(2)(p75, Split(blockFactor) ∘ Transpose() ∘ Get(0)(p0))
))(Zip(2)(Split(sizeK) ∘ Transpose()(p179), p70))
)) ∘ Transpose() ∘ Map((p4 ↣
Split(sizeN) ∘ Transpose()(p4)
)) ∘ Split(sizeK)(p36)
)) ∘ Split(sizeM)(p239)

```

# ELEVATE for optimising LIFT programs

LIFT Program 

Domain Specific Language  
embedded in Scala

```
val scale = fun(a => fun(xs =>  
  xs ▷ map(fun(x => a * x) )))
```

ELEVATE Program 

Domain Specific Language  
embedded in Scala

# LIFT rewrite rule in ELEVATE

LIFT Program 

```
val scale = fun(a => fun(xs =>
  xs ▷ map(fun(x => a * x))) )
```

ELEVATE Program 

```
def splitJoin(n: Nat)(e: Lift): RewriteResult[Lift] = e match {
  case Apply(`map`, f) => Success(split(n) ▷ map(map(f)) ▷ join)
  case _ => Failure(splitJoin(n))
}
```

# LIFT rewrite rule in ELEVATE

LIFT Program

```
val scale = fun(a => fun(xs =>  
  xs ▷ map(fun(x => a * x) )))
```

ELEVATE Program

```
def splitJoin(n: Nat)(e: Lift): RewriteResult[Lift] = e match {  
  case Apply(`map`, f) => Success(split(n) ▷ map(map(f)) ▷ join)  
  case _ => Failure(splitJoin(n))  
}
```

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

# LIFT rewrite rule in ELEVATE

LIFT Program

```
val scale = fun(a => fun(xs =>  
  xs ▷ map(fun(x => a * x) )))
```

ELEVATE Program

```
def splitJoin(n: Nat)(e: Lift): RewriteResult[Lift] = e match {  
  case Apply(`map`, f) => Success(split(n) ▷ map(map(f)) ▷ join)  
  case _ => Failure(splitJoin(n))  
}
```

Apply transformation:

```
splitJoin(n)(scale)
```

# LIFT rewrite rule in ELEVATE

LIFT Program

```
val scale = fun(a => fun(xs =>  
  xs ▷ map(fun(x => a * x) )))
```

ELEVATE Program

```
def splitJoin(n: Nat)(e: Lift): RewriteResult[Lift] = e match {  
  case Apply(`map`, f) => Success(split(n) ▷ map(map(f)) ▷ join)  
  case _ => Failure(splitJoin(n))  
}
```

Apply transformation:

```
splitJoin(n)(scale)
```

Failure!

# LIFT rewrite rule in ELEVATE

LIFT Program

```
val scale = fun(a => fun(xs =>  
  xs ▷ map(fun(x => a * x) )))
```

ELEVATE Program

```
def splitJoin(n: Nat)(e: Lift): RewriteResult[Lift] = e match {  
  case Apply(`map`, f) => Success(split(n) ▷ map(map(f)) ▷ join)  
  case _ => Failure(splitJoin(n))  
}
```

Apply transformation:

```
splitJoin(n)(scale)
```

Failure!

The transformation is applied at the wrong location

# Traversal LIFT programs

ELEVATE Program 

```
def body(s: Lift ⇒ RewriteResult[Lift])
        (e: Lift): RewriteResult[Lift] = e match {
    case Lambda(f, x) ⇒ s(x).mapSuccess(y ⇒ Lambda(f, y))
    case _ ⇒ Failure(s)
}

def function(s: Lift ⇒ RewriteResult[Lift])
        (e: Lift): RewriteResult[Lift] = e match {
    case Apply(f, e) ⇒ s(f).mapSuccess(g ⇒ Apply(g, e))
    case _ ⇒ Failure(s)
}
```

# Traversal LIFT programs

ELEVATE Program 

```
def body(s: Lift ⇒ RewriteResult[Lift])
        (e: Lift): RewriteResult[Lift] = e match {
    case Lambda(f, x) ⇒ s(x).mapSuccess(y ⇒ Lambda(f, y))
    case _ ⇒ Failure(s)
}

def function(s: Lift ⇒ RewriteResult[Lift])
        (e: Lift): RewriteResult[Lift] = e match {
    case Apply(f, e) ⇒ s(f).mapSuccess(g ⇒ Apply(g, e))
    case _ ⇒ Failure(s)
}
```

Apply transformation:

```
body(body(function(splitJoin(n)))){
    fun(a ⇒ fun(xs ⇒
        xs ▷ map(fun(x ⇒ a * x) ))
    )
}
```

Success!

# Traversal LIFT programs

ELEVATE Program 

Compose existing strategies

```
def body(s: Lift ⇒ RewriteResult[Lift])
  (e: Lift): RewriteResult[Lift] = e match {
  case Lambda(f, x) ⇒ s(x).mapSuccess(y ⇒ Lambda(f, y))
  case _ ⇒ Failure(s)
}

def function(s: Lift ⇒ RewriteResult[Lift])
  (e: Lift): RewriteResult[Lift] = e match {
  case Apply(f, e) ⇒ s(f).mapSuccess(g ⇒ Apply(g, e))
  case _ ⇒ Failure(s)
}
```

Apply transformation:

```
body(body(function(splitJoin(n)))){
  fun(a ⇒ fun(xs ⇒
    xs ▷ map(fun(x ⇒ a * x) )))
}
```

Success!

# Traversal LIFT programs

ELEVATE Program 

These are domain specific abstractions  
that makes sense for optimising  
LIFT programs.

These are not backed into ELEVATE

Apply transformation:

```
def body(s: Lift ⇒ RewriteResult[Lift])
  (e: Lift): RewriteResult[Lift] = e match {
  case Lambda(f, x) ⇒ s(x).mapSuccess(y ⇒ Lambda(f, y))
  case _ ⇒ Failure(s)
}

def function(s: Lift ⇒ RewriteResult[Lift])
  (e: Lift): RewriteResult[Lift] = e match {
  case Apply(f, e) ⇒ s(f).mapSuccess(g ⇒ Apply(g, e))
  case _ ⇒ Failure(s)
}
```

```
body(body(function(splitJoin(n)))(  
  fun(a ⇒ fun(xs ⇒  
    xs ▷ map(fun(x ⇒ a * x) )))  
)
```

Compose existing strategies

Success!

# Complex compiler optimisations in ELEVATE

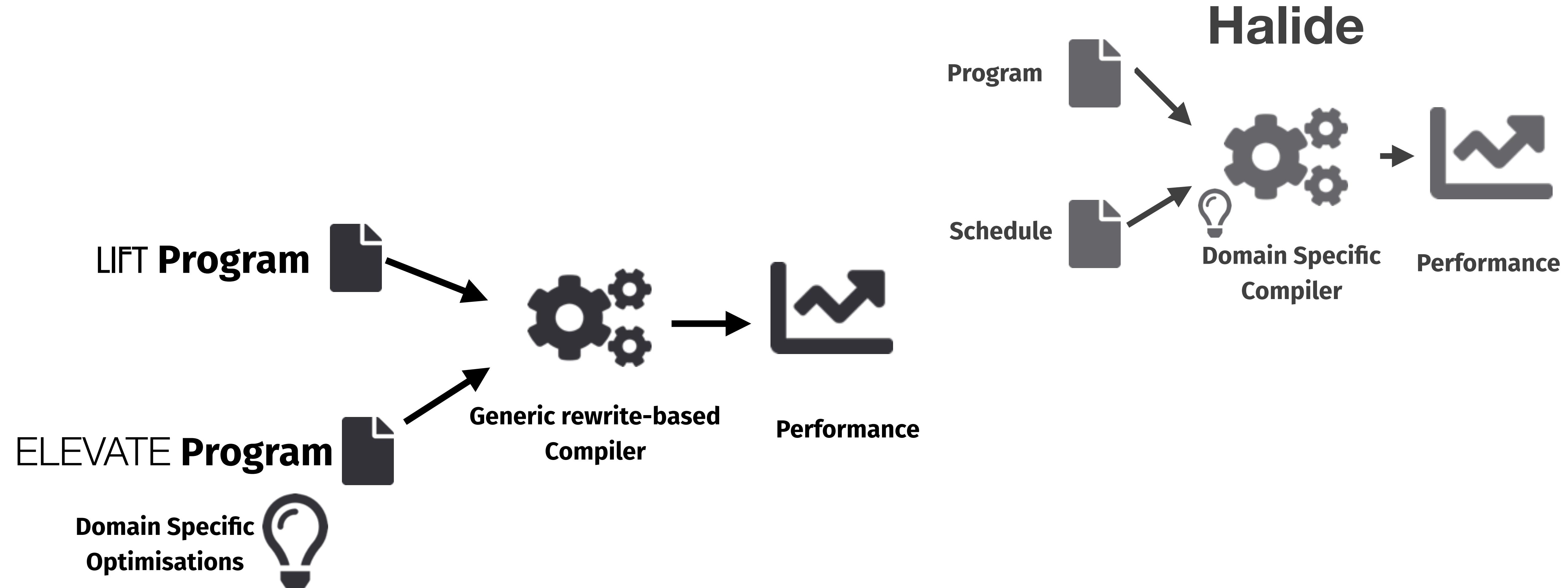
With ELEVATE we easily express traditional compiler optimisations, like tiling or loop reordering:

```
def tileNDRec: Int => Int => Strategy[Lift] = dim => n => dim match {
  case x if x <= 0 => id()
  case 1 => function(splitJoin(n))
  case 2 => fmap(function(splitJoin(n))) `;` function(splitJoin(n)) `;` shiftDim(2)
  case i => fmap(tileNDRec(dim-1)(n)) `;` tileNDRec(1)(n) `;` shiftDim(i)
}
```

Sequential composition of strategies

```
def reorder: Seq[Int] => Strategy[Lift] = perm => {
  if(perm.length = 1) return id
  (perm.head match {
    case 1 => fmap(reorder(perm.tail.map(_-1)))
    case x =>
      val transposes = x-1
      shiftDimension(transposes) `;`
      moveTowardsArgument(transposes)(fmap(reorder(perm.tail.map(y => if(y > x) y-1 else y ))))
  }) `;` RNF `;` LCNF
}
```

# ELEVATE for optimising LIFT programs



**Goal: Demonstrate same performance as Halide with a more extensible design**

# Generic ELEVATE combinators

ELEVATE defines generic combinators for programs written in an arbitrary language  $P$

```
type Strategy[P] = P => RewriteResult[P]

def id[P](p: P) = Success(p)

def seq[P](f: Strategy[P], s: Strategy[P])
          (p: P): RewriteResult[P] = f(p).flatMapSuccess(s)

def leftChoice[P](f: Strategy[P], s: Strategy[P])
                  (p: P): RewriteResult[P] = f(p).flatMapFailure(_ => s(p))

def try[P](s: Strategy[P])
           (p: P): RewriteResult[P] = leftChoice[P](s, id)(p)

def repeat[P](s: Strategy[P])
              (p: P): RewriteResult[P] = `try`[P](s `;` repeat[P](s))®

...
```

# Generic ELEVATE traversals

ELEVATE **defines generic traversals if three basic traversals are defined for P**

```
// applies strategy to all direct subexpressions
def all[P]: Strategy[P] ⇒ Strategy[P]

// applies strategy to one direct subexpression
def one[P]: Strategy[P] ⇒ Strategy[P]

// applies strategy to at least one direct subexpression
def some[P]: Strategy[P] ⇒ Strategy[P]

def oncetd[P](s: Strategy[P])
  (p: P): RewriteResult[P] = (s <+ one(oncetd(s)))) (p)

def tryAll[P](s: Strategy[P])
  (p: P): RewriteResult[P] = (all(tryAll(`try`(s))) `;` `try`'(s)) Ⓜ
...
...
```

# Generic ELEVATE normalisation

ELEVATE **defines a normalisation strategy based on the generic traversals**

```
def normalize[P]: Strategy[P] => Strategy[P] = s => repeat(oncetd(s))
```

This applies a given strategy until this is not applicable anymore

# ELEVATE for optimising FSmooth programs

[ICFP 2019]

97

## Efficient Differentiable Programming in a Functional Array-Processing Language

AMIR SHAIKHHA, University of Oxford, United Kingdom

ANDREW FITZGIBBON, Microsoft Research, United Kingdom

DIMITRIOS VYTINIOTIS, DeepMind, United Kingdom

SIMON PEYTON JONES, Microsoft Research, United Kingdom

We present a system for the automatic differentiation (AD) of a higher-order functional array-processing language. The core functional language underlying this system simultaneously supports both source-to-source forward-mode AD and global optimisations such as loop transformations. In combination, gradient computation with forward-mode AD can be as efficient as reverse mode, and that the Jacobian matrices required for numerical algorithms such as Gauss-Newton and Levenberg-Marquardt can be efficiently computed.

CCS Concepts: • Mathematics of computing → Automatic differentiation; • Software and its engineering → Functional languages; *Domain specific languages*.

Additional Key Words and Phrases: Linear Algebra, Differentiable Programming, Optimising Compilers, Loop Fusion, Code Motion.

### ACM Reference Format:

Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones. 2019. Efficient Differentiable Programming in a Functional Array-Processing Language. *Proc. ACM Program. Lang.* 3, ICFP, Article 97 (August 2019), 30 pages. <https://doi.org/10.1145/3341701>

## 5 EFFICIENT DIFFERENTIATION

...

**One of the key challenges for applying these rewrite rules is the order in which these rules should be applied.**

We apply these rules based on **heuristics** and **cost models for the size of the code** (which is used by many optimising compilers, especially the ones for just-in-time scenarios). Furthermore, based on heuristics, we ensure that certain rules are applied only when some specific other rules are applicable. For example, the loop fission rule (Figure 8g) is usually applicable only when it can be combined with tuple projection partial evaluation rules (Figure 8f). **We leave the use of search strategies for automated rewriting** (e.g., using Monte-Carlo tree search [De Mesmay et al. 2009]) **as future work**.

...

# ELEVATE for optimising FSmooth programs

97:14

Amir Shaikhha, Andrew Fitzgibbon, Dimitrios Vytiniotis, and Simon Peyton Jones

$(\text{fun } x \rightarrow e_0) e_1 \rightsquigarrow \text{let } x = e_1 \text{ in } e_0$	$e + 0 = 0 + e \rightsquigarrow e$
$\text{let } x = e_0 \text{ in } e_1 \rightsquigarrow e_1[x \mapsto e_0]$	$e * 1 = 1 * e \rightsquigarrow e$
$\text{let } x = e_0 \text{ in } e_1 \rightsquigarrow e_1(x \notin \text{fvs}(e_1))$	$e * 0 = 0 * e \rightsquigarrow 0$
$\text{let } x = \text{let } y = e_0 \text{ in } e_1 \text{ in } e_2 \rightsquigarrow \text{let } y = e_0 \text{ in } \text{let } x = e_1 \text{ in } e_2$	$e + -e = e - e \rightsquigarrow 0$
$\text{let } x = e_0 \text{ in } \text{let } y = e_0 \text{ in } e_1 \rightsquigarrow \text{let } x = e_0 \text{ in } \text{let } y = x \text{ in } e_1$	$e_0 * e_1 + e_0 * e_2 \rightsquigarrow e_0 * (e_1 + e_2)$

(b) Ring-Structure Rules

$f(\text{let } x = e_0 \text{ in } e_1) \rightsquigarrow \text{let } x = e_0 \text{ in } f(e_1)$	$(\text{build } e_0 e_1)[e_2] \rightsquigarrow e_1 e_2$
	$\text{length}(\text{build } e_0 e_1) \rightsquigarrow e_0$

(c) Loop Fusion Rules

$\text{if true then } e_1 \text{ else } e_2 \rightsquigarrow e_1$
$\text{if false then } e_1 \text{ else } e_2 \rightsquigarrow e_2$
$\text{if } e_0 \text{ then } e_1 \text{ else } e_1 \rightsquigarrow e_1$
$\text{if } e_0 \text{ then } e_1 \text{ else } e_2 \rightsquigarrow \text{if } e_0 \text{ then } e_1[e_0 \mapsto \text{true}] \text{ else } e_2[e_0 \mapsto \text{false}]$
$f(\text{if } e_0 \text{ then } e_1 \text{ else } e_2) \rightsquigarrow \text{if } e_0 \text{ then } f(e_1) \text{ else } f(e_2)$

(d) Conditional Rules

$\text{ifold } f z 0 \rightsquigarrow z$
$\text{ifold } f z n \rightsquigarrow \text{ifold } (\text{fun } a i \rightarrow f a (i+1))(f z 0)(n - 1)$
$\text{ifold } (\text{fun } a i \rightarrow a) z n \rightsquigarrow z$
$\text{ifold } (\text{fun } a i \rightarrow \text{if}(i = e_0) \text{ then } e_1 \text{ else } a) z n \rightsquigarrow \text{let } a = z \text{ in } \text{let } i = e_0 \text{ in } e_1 \text{ (if } e_0 \text{ does not mention } a \text{ or } i\text{)}$

(e) Loop Normalisation Rules

$\text{fst } (e_0, e_1) \rightsquigarrow e_0$	$\text{ifold } (\text{fun } a i \rightarrow (f_0(\text{fst } a)i, f_1(\text{snd } a)i) \rightsquigarrow (\text{ifold } f_0 z_0 n, \text{ifold } f_1 z_1 n)) (z_0, z_1) n$
---	---

(g) Loop Fission Rule

(f) Tuple Normalisation Rules

```

def funToLet(e: FSmooth): RewriteResult[FSmooth] = e match {
  case Application(Abstraction(Seq(x), e0, _, Seq(e1), _)) =>
    Success(Let(x, e1, e0))
  case _ => Failure(funToLet)
}

def additionZero(e: FSmooth): RewriteResult[FSmooth] = e match {
  case Application(`+`(_), Seq(e, ScalarValue(0)), _) =>
    Success(e)
  case Application(`+`(_), Seq(ScalarValue(0), e), _) =>
    Success(e)
  case _ => Failure(additionZero)
}

def trivialFold(e: FSmooth): RewriteResult[FSmooth] = e match {
  case Application(`ifold`(_), Seq(f, z, ScalarValue(0)), _) =>
    Success(z)
  case _ => Failure(trivialFold)
}

...

```

Fig. 8. Transformation Rules for  $\tilde{F}$ . Even though none of these rules are AD-specific, the rules of Figure 8f and Figure 8g are more useful in the AD context.

# ELEVATE for optimising FSmooth programs

**Example 5.** It is known that for a matrix  $M$ , the following equality holds  $(M^T)^T = M$ . We show how we can derive the same equality in dF. In other words, we show that:

```
matrixTranspose (matrixTranspose M) = M
```

```
let MT =
  build(length M[0]) (fun i ->
    build(length M) (fun j ->
      M[j][i] )) in
build(length MT[0]) (fun i ->
  build(length MT) (fun j ->
    MT[j][i] ))
```

Now, by applying the loop fusion rules (cf. Figure 8c) and performing further partial evaluation, the following expression is derived:

```
build(length M) (fun i ->
  build(length M[0]) (fun j ->
    M[i][j] ))
```

```
normalize()
buildGet <+
lengthBuild <+
letPartialEvaluation <+
conditionalPartialEvaluation <+
conditionApplication <+
letApplication <+
funToLet <+
letFission <+
letInitDuplication
).apply(
  fun(M => matrixTranspose(matrixTranspose(M)))
)
```

Left choice combinator

# ELEVATE

## A programming language for program optimizations

This is work in progress.

No evaluation yet, and some open questions and challenges:

- How do we evaluate ELEVATE?
- How do we design a programming interface friendly to systems programmers?
- Can we use ELEVATE to help model stochastic searches in a design space?
- Can we automatically find good ELEVATE programs,  
e.g. using machine learning or program synthesis techniques?

# LIFT IS OPEN SOURCE!



University  
of Glasgow



THE UNIVERSITY  
of EDINBURGH

WWU  
MÜNSTER

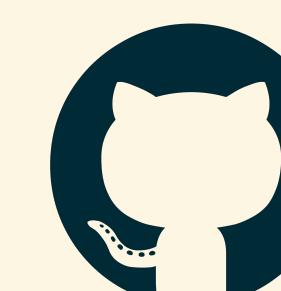
more info at:

# lift-project.org

„ Paper



Artifacts



Source Code

Naums Mogers



Christof Schlaak



Bastian Hagedorn



Toomas Remmelg



Larisa Stoltzfus



Federico Pizzuti



Bastian Köpcke



Christophe Dubach



Andrej Ivanis



Michel Steuwer



Thomas Koehler



Lu Li

[michel.steuwer.info](http://michel.steuwer.info)

# ELEVATE *a language to write composable program optimisations*

Michel Steuwer — [michel.steuwer@glasgow.ac.uk](mailto:michel.steuwer@glasgow.ac.uk)

[www.lift-project.org](http://www.lift-project.org)     @LIFTlang

**INSPIRING  
PEOPLE**

#UofGWorldChangers



@UofGlasgow