

# HIGH-PERFORMANCE DOMAIN-SPECIFIC COMPIRATION WITHOUT DOMAIN-SPECIFIC COMPILERS

BASTIAN HAGEDORN

WHAT IS HIGH-PERFORMANCE  
DOMAIN-SPECIFIC COMPIRATION ?

WITHOUT AND WHY DO WE WANT IT?  
DOMAIN-SPECIFIC COMPILERS

BASTIAN HAGEDORN

# A SOLVED PROBLEM

HOW TO MAKE HIGH PERFORMANCE  
ACCESSIBLE TO DOMAIN SCIENTISTS?



REQUIRES HIGH PERFORMANCE  
FOR SCIENTIFIC APPLICATIONS



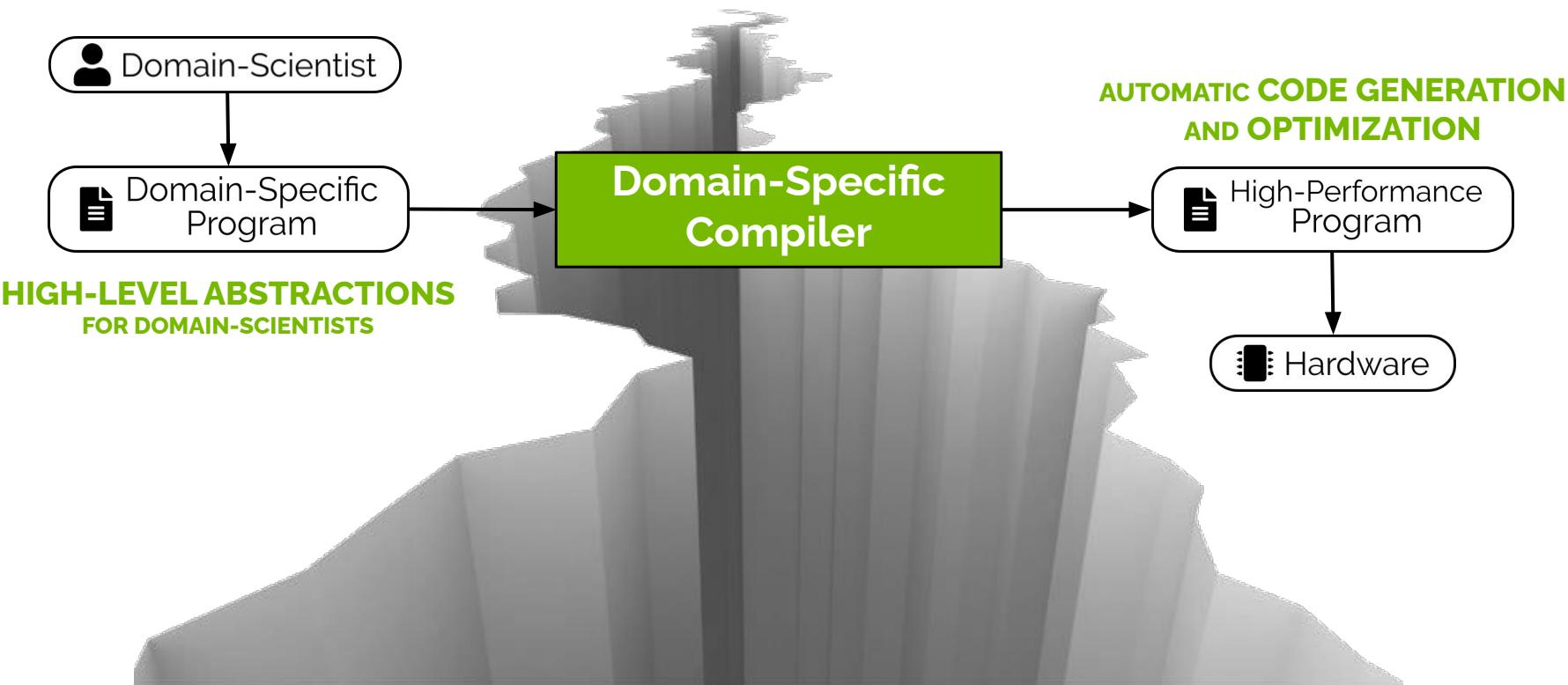
MODERN PARALLEL



PROVIDES HIGH PERFORMANCE  
BUT IS HARD TO PROGRAM

# A SOLVED PROBLEM

HOW TO MAKE HIGH PERFORMANCE  
ACCESSIBLE TO DOMAIN SCIENTISTS?



# HIGH-PERFORMANCE DOMAIN-SPECIFIC COMPIRATION

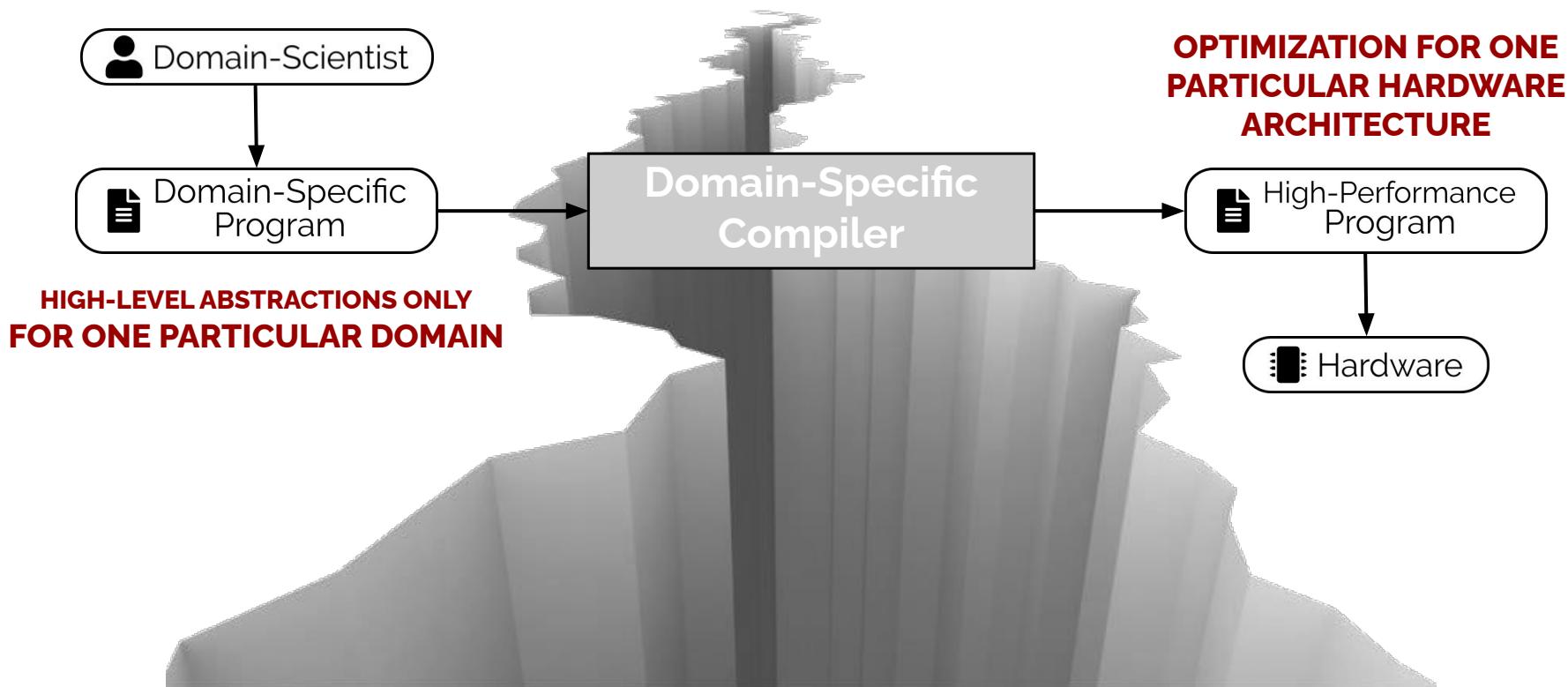
WHY TRYING TO ACHIEVE THIS WITHOUT  
DOMAIN-SPECIFIC COMPILERS?

WHAT IS WRONG WITH THEM?

BASTIAN HAGEDORN

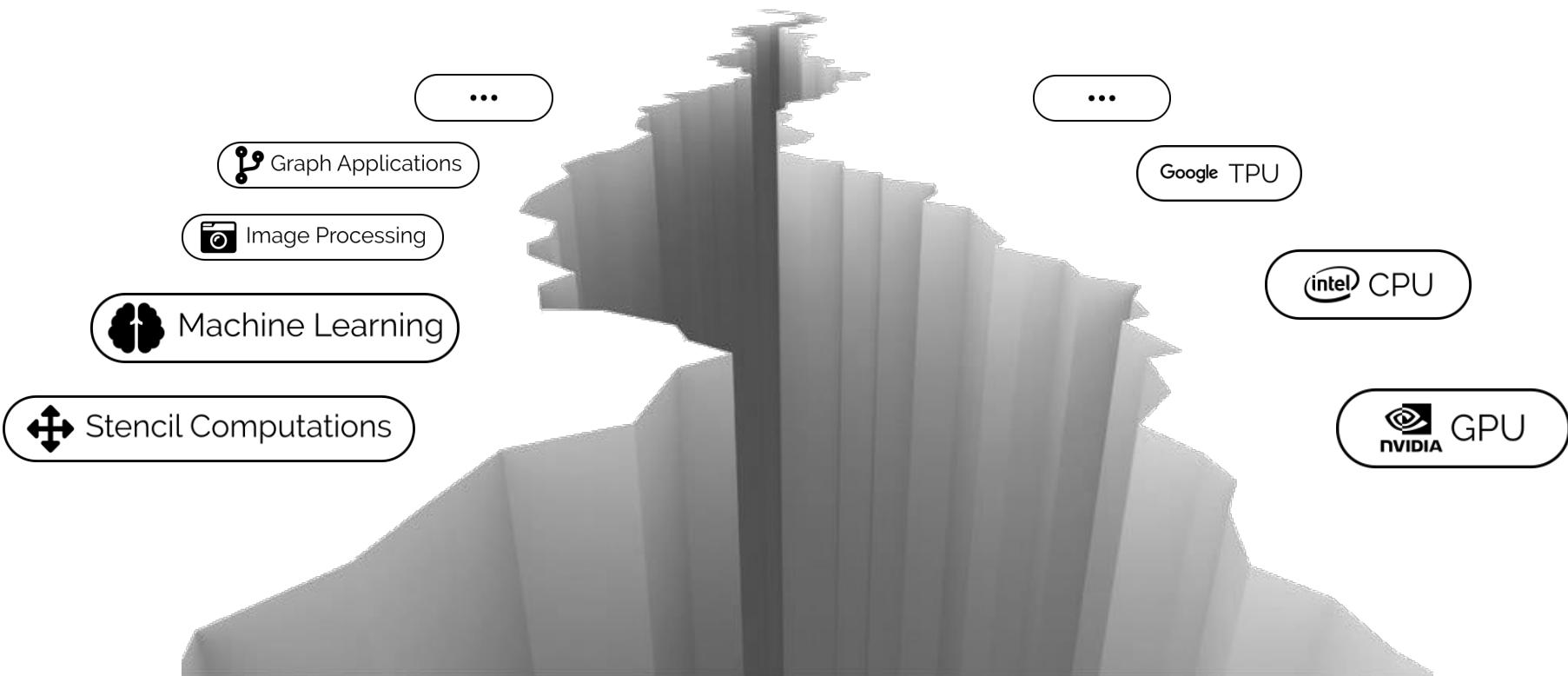
# THE NEW CHALLENGE

DOMAIN-SPECIFIC COMPILERS ARE  
*NOT REUSABLE ALMOST BY DEFINITION*



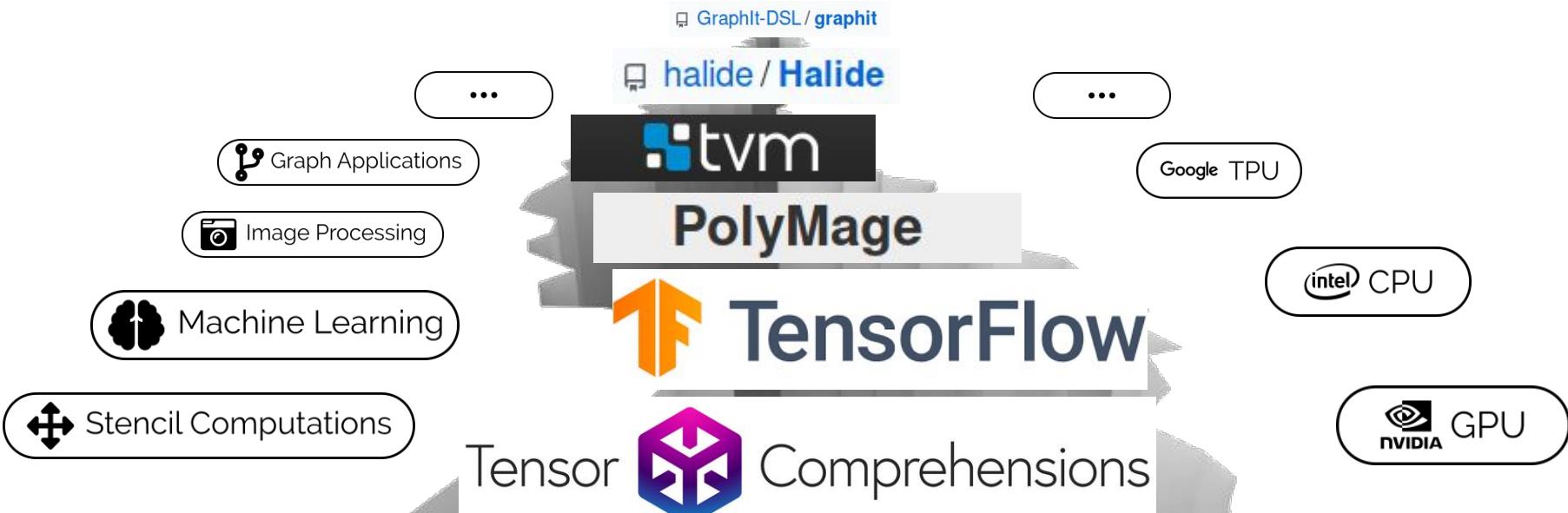
# THE NEW CHALLENGE

DOMAIN-SPECIFIC COMPILERS ARE  
*NOT REUSABLE ALMOST BY DEFINITION*



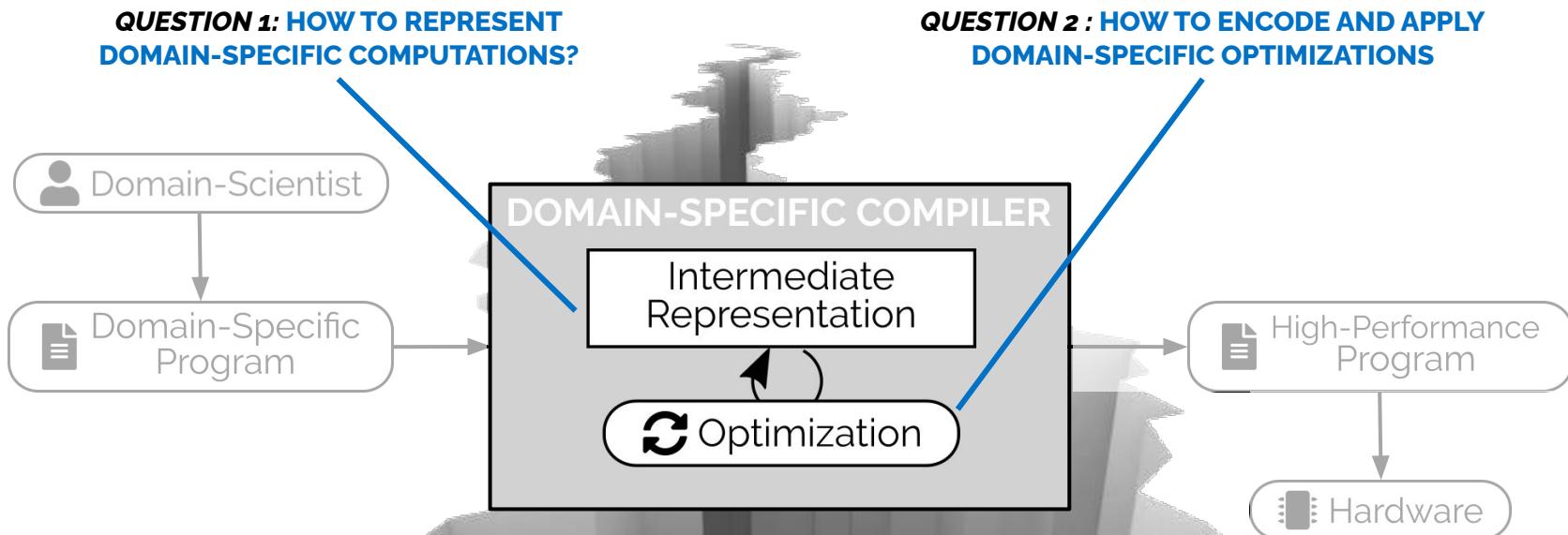
# THE NEW CHALLENGE

DOMAIN-SPECIFIC COMPILERS ARE  
*NOT REUSABLE ALMOST BY DEFINITION*



# THE NEW CHALLENGE

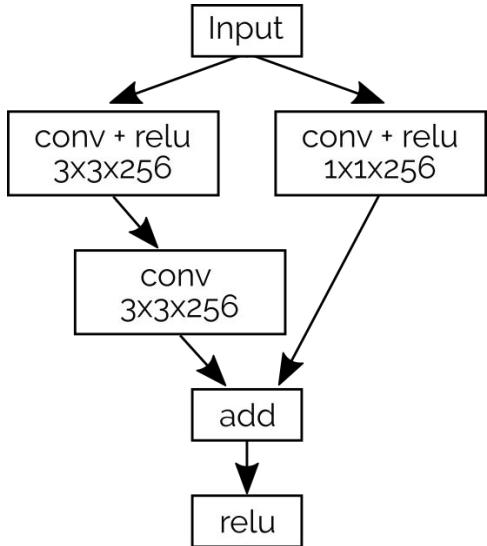
HOW TO DESIGN A *REUSABLE*  
DOMAIN-SPECIFIC COMPILER?



# THE IR CHALLENGE

## HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?

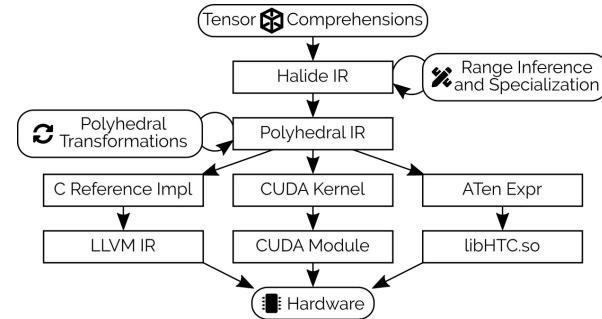
Three approaches used in existing state-of-the-art compilers today:



```
1 ; ... 39 lines left out
2 28: ; preds = %25
3 %29 = add nsw i64 %27, %24
4 %30 = getelementptr inbounds float, float* %0, i64 %29
5 %31 = load float, float* %30, align 4, !tbaa !4
6 %32 = mul nsw i64 %27, %12
7 %33 = getelementptr inbounds float, float* %1, i64 %32
8 %34 = load float, float* %33, align 4, !tbaa !4
9 %35 = fmul float %31, %34
10 %36 = fadd float %26, %35
11 store float %36, float* %23, align 4, !tbaa !4
12 br label %37
13 ; ... 58 lines left out
```

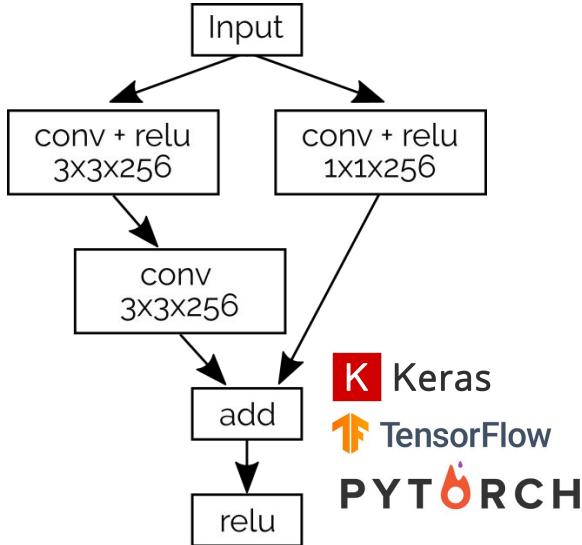
**HIGH-LEVEL INTERMEDIATE  
REPRESENTATIONS**

**LOW-LEVEL INTERMEDIATE  
REPRESENTATIONS**



**HIERARCHICAL INTERMEDIATE  
REPRESENTATIONS**

# THE IR CHALLENGE

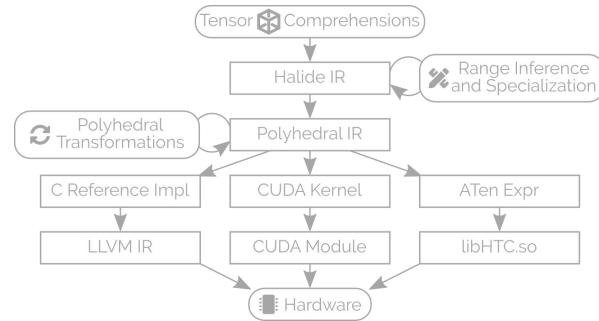


HIGH-LEVEL INTERMEDIATE  
REPRESENTATIONS

```
1 ; ... 39 lines left out
2 28: ; preds = %25
3 %29 = add nsw i64 %27, %24
4 %30 = getelementptr inbounds float, float* %0, i64 %29
5 %31 = load float, float* %30, align 4, !tbaa !4
6 %32 = mul nsw i64 %27, %12
7 %33 = getelementptr inbounds float, float* %1, i64 %32
8 %34 = load float, float* %33, align 4, !tbaa !4
9 %35 = fmul float %31, %34
10 %36 = fadd float %26, %35
11 store float %36, float* %23, align 4, !tbaa !4
12 br label %37
13 ; ... 58 lines left out
```

LOW-LEVEL INTERMEDIATE  
REPRESENTATIONS

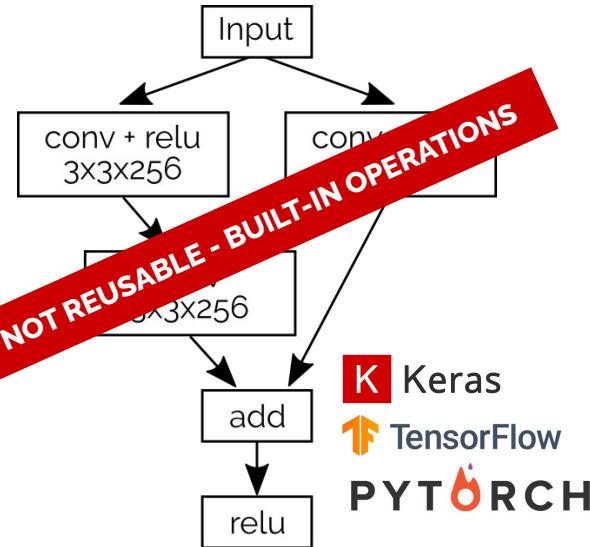
## HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?



HIERARCHICAL INTERMEDIATE  
REPRESENTATIONS

# THE IR CHALLENGE

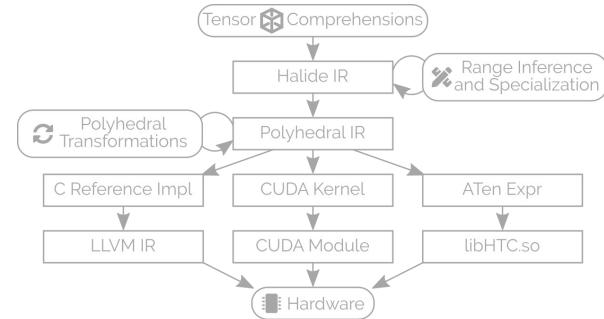
## HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?



**HIGH-LEVEL INTERMEDIATE  
REPRESENTATIONS**

```
1 ; ... 39 lines left out
2 28: ; preds = %25
3 %29 = add nsw i64 %27, %24
4 %30 = getelementptr inbounds float, float* %0, i64 %29
5 %31 = load float, float* %30, align 4, !tbaa !4
6 %32 = mul nsw i64 %27, %12
7 %33 = getelementptr inbounds float, float* %1, i64 %32
8 %34 = load float, float* %33, align 4, !tbaa !4
9 %35 = fmul float %31, %34
10 %36 = fadd float %26, %35
11 store float %36, float* %23, align 4, !tbaa !4
12 br label %37
13 ; ... 58 lines left out
```

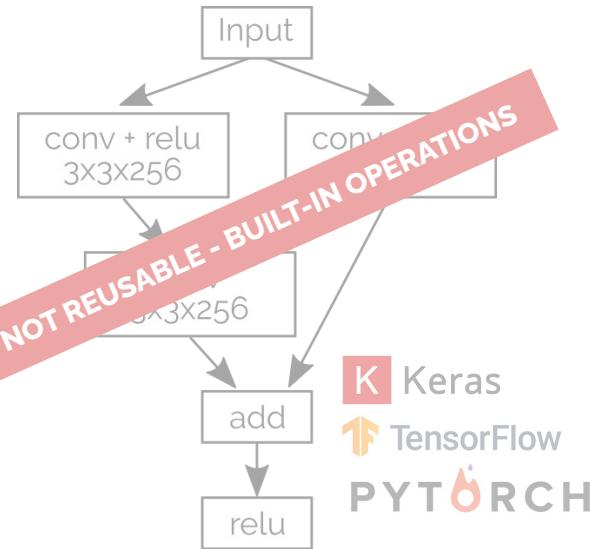
**LOW-LEVEL INTERMEDIATE  
REPRESENTATIONS**



**HIERARCHICAL INTERMEDIATE  
REPRESENTATIONS**

# THE IR CHALLENGE

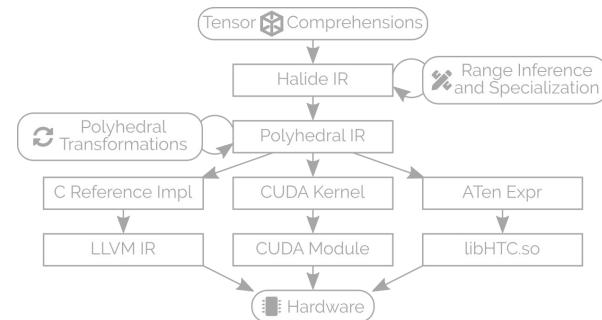
## HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?



HIGH-LEVEL INTERMEDIATE  
REPRESENTATIONS

```
1 ; ... 39 lines left out
2 28: ; preds = %25
3 %29 = add nsw i64 %27, %24
4 %30 = getelementptr inbounds float, float* %0, i64 %29
5 %31 = load float, float* %30, align 4, !tbaa !4
6 %32 = mul nsw i64 %27, %12
7 %33 = getelementptr inbounds float, float* %1, i64 %32
8 %34 = load float, float* %33, align 4, !tbaa !4
9 %35 = fmul float %31, %34
10 %36 = fadd float %26, %35
11 store float %36, float* %23, align 4, !tbaa !4
12 br label %37
13 ; ... 58 lines left out
```

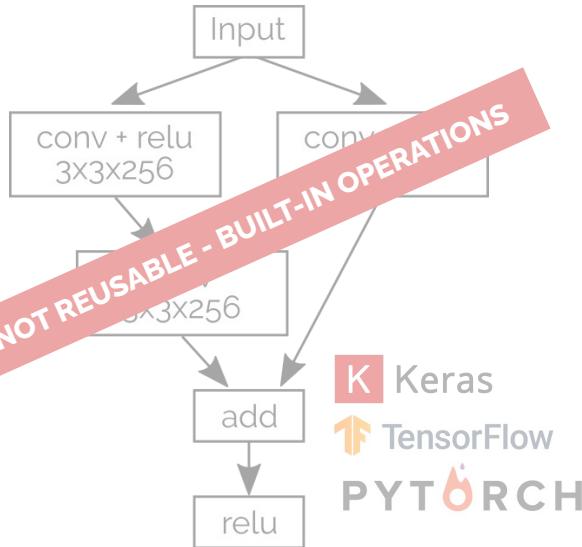
LOW-LEVEL INTERMEDIATE  
REPRESENTATIONS



HIERARCHICAL INTERMEDIATE  
REPRESENTATIONS

# THE IR CHALLENGE

## HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?



A snippet of LLVM assembly code representing the computation graph:

```
1 ; ... 39 lines left out
2 28:
3 %29 = add nsw i64 %27, %24
4 %30 = getelementptr inbounds [float*] %25, float* %25, i64 %29
5 %31 = load float, float* %30, align 4, !tbaa !4
6 %32 = mul nsw i64 %31, %26, %35
7 %33 = getelementptr inbounds [float*] %23, float* %1, i64 %32
8 %34 = add nsw i64 %33, align 4, !tbaa !4
9 %35 = mul nsw i64 %34, %26, %35
10 %36 = add nsw i64 %35, align 4, !tbaa !4
11 %37 = label %37
12 ; ... 58 lines left out
13
```

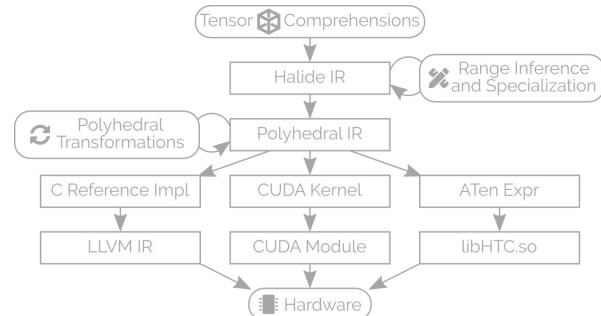
**LOSS OF DOMAIN-SPECIFIC INFORMATION** is written diagonally across the code snippet.



HIGH-LEVEL INTERMEDIATE  
REPRESENTATIONS

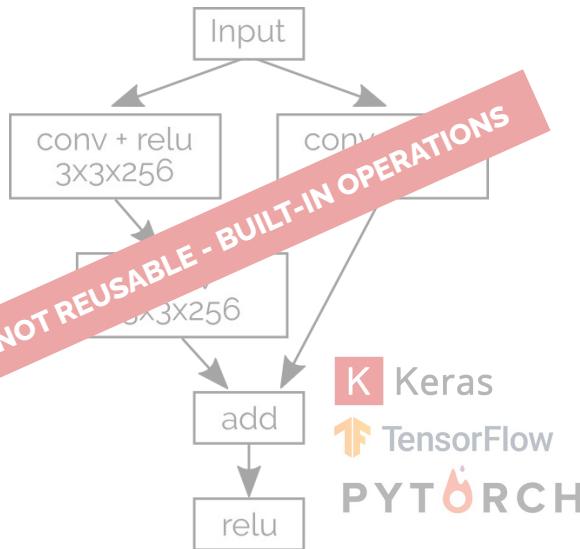
LOW-LEVEL INTERMEDIATE  
REPRESENTATIONS

HIERARCHICAL INTERMEDIATE  
REPRESENTATIONS



# THE IR CHALLENGE

## HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?



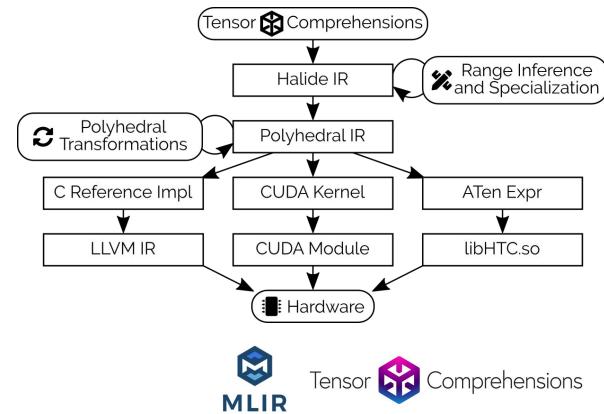
1 ; ... 39 lines left out  
2 28:  
3 %29 = add nsw i64 %27, %24  
4 %30 = getElementptr inbounds i64 %26, i64 %29  
5 %31 = load float, float\* %30, align 4, !tbaa !4  
6 %32 = mul nsw i64 %31, float\* %1, i64 %32  
7 %33 = getelementptr inbounds float, float\* %1, i64 %32  
8 %34 = mul nsw i64 %33, align 4, !tbaa !4  
9 %35 = add nsw i64 %34, %35  
10 %36 = add nsw i64 %36, float\* %23, align 4, !tbaa !4  
11 %37 = label %37  
12 ; ... 58 lines left out

**LOSS OF DOMAIN-SPECIFIC INFORMATION** is written diagonally across the code snippet.



**HIGH-LEVEL INTERMEDIATE REPRESENTATIONS**

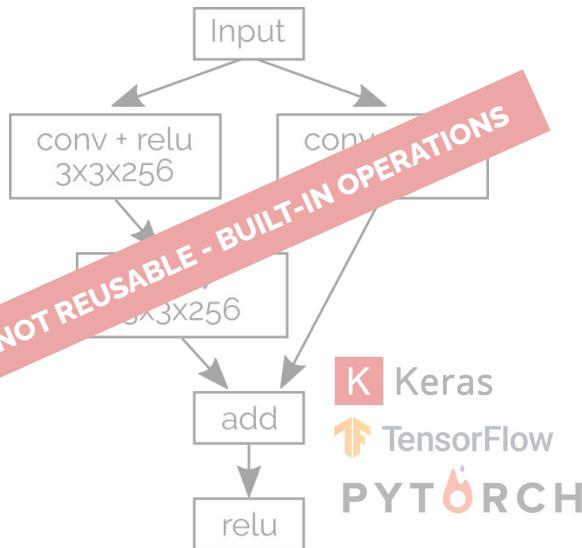
**LOW-LEVEL INTERMEDIATE REPRESENTATIONS**



**HIERARCHICAL INTERMEDIATE REPRESENTATIONS**

# THE IR CHALLENGE

## HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?

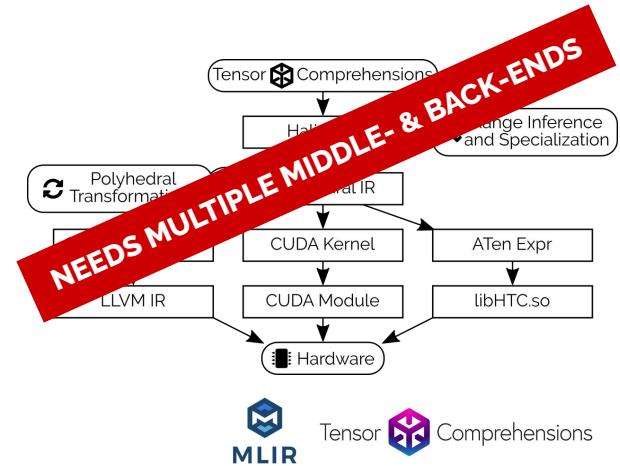


**LOSS OF DOMAIN-SPECIFIC INFORMATION**

```
1 ; ... 39 lines left out
2 28:
3 %29 = add nsw i64 %27, %24
4 %30 = getElementptr inbounds i64* %26, i64 %29
5 %31 = load float, float* %30, align 4, !tbaa !4
6 %32 = mul nsw i64 %31, float* %1, i64 %32
7 %33 = getelementptr i64, float* %1, i64 %32
8 %34 = add nsw i64 %33, align 4, !tbaa !4
9 %35 = mul nsw i64 %34, float* %23, align 4, !tbaa !4
10 %36 = add nsw i64 %35, float* %23, align 4, !tbaa !4
11 %37 = label %37
12 , ... 58 lines left out
13 ; ... 39 lines left out
```

**HIGH-LEVEL INTERMEDIATE  
REPRESENTATIONS**

**LOW-LEVEL INTERMEDIATE  
REPRESENTATIONS**



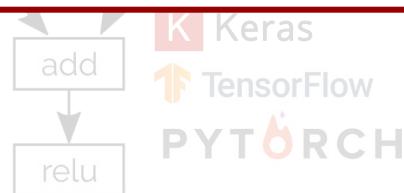
**HIERARCHICAL INTERMEDIATE  
REPRESENTATIONS**

# THE IR CHALLENGE

HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?

## THE IR CHALLENGE:

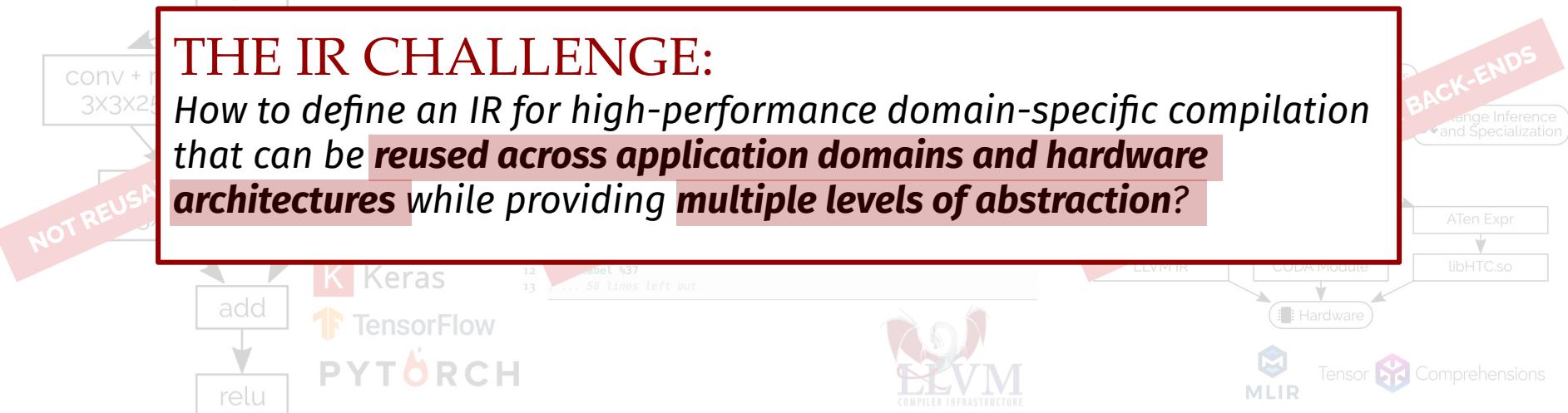
*How to define an IR for high-performance domain-specific compilation that can be reused across application domains and hardware architectures while providing multiple levels of abstraction?*



HIGH-LEVEL INTERMEDIATE  
REPRESENTATIONS

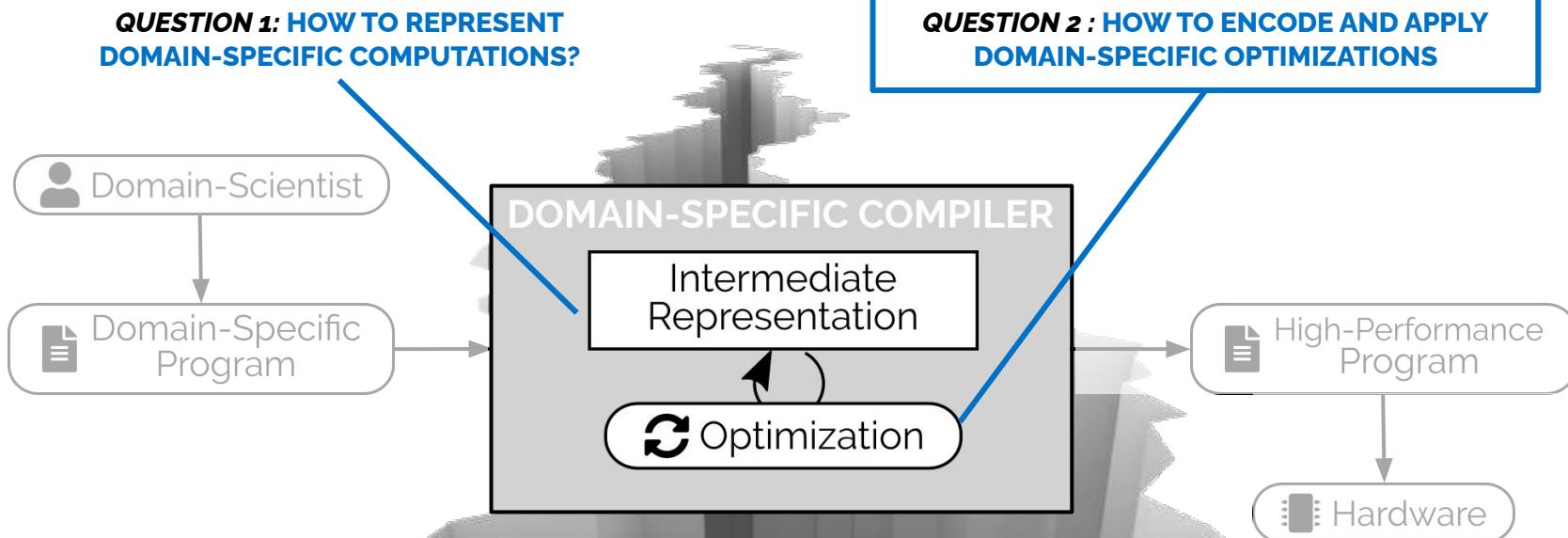
LOW-LEVEL INTERMEDIATE  
REPRESENTATIONS

HIERARCHICAL INTERMEDIATE  
REPRESENTATIONS



# THE NEW CHALLENGE

HOW TO DESIGN A  
DOMAIN-SPECIFIC COMPILER?



# THE OPTIMIZATION CHALLENGE

## HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS?

Three approaches used in existing state-of-the-art compilers today:



RELYING ON LIBRARIES

```
Pass Arguments: -targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary
info -forceattrs -inferattrs -domtree -callsite-splitting -ipscpp -called-value-propagation
-attribute -globalopt -domtree -mem2reg -deadgslim -domtree -basicaa -aa -loops -lazy-branch-
prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -basiccg -globals-aa -prune
-eh -inline -functionattrs -argpromotion -domtree -sra -basicaa -aa -memoryss -early-cse-
memssa -speculative-execution -basicaa -aa -lazy-value-info -jump-threading -correlated-
propagation -simplifycfg -domtree -aggressive -instcombine -basicaa -aa -loops -lazy-branch-prob
-lazy-block-freq -opt-remark-emitter -instcombine -libcalls-shrinkwrap -loops -branch-prob -
block-freq -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -pgo-memop-opt -basicaa -aa -
loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim -simplifycfg -
reassociate -domtree -loops -loop-simplify -lcssa -basicaa -aa -scalar-
evolution -loop-rotate -lcm -loop-unswitch -simplifycfg -domtree -basicaa -aa -loops -lazy-
branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -lcssa -scalar-evolution -loop-simplify -lcssa-
verification -lcssa -scalar-evolution -indvars -loop-idiom -loop-deletion -loop-unroll -mldst-
motion -phi-values -basicaa -aa -memdep -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -
gvn -phi-values -basicaa -aa -memdep -nemcpops -scpp -demanded-bits -bdce -basicaa -aa -loops -
lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -lazy-value-info -jump-
threading -correlated-propagation -basicaa -aa -phi-values -memdep -dse -loops -loop-freq -loop-
lcssa-verification -lcssa -basicaa -aa -scalar-evolution -lcls -postdomtree -adce -simplifycfg -
domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -
instcombine -barrier -elim-avail-extern -basiccg -rpo-functionattrs -globalopt -globaladce -
basicaa -aa -scalar-evolution -loop-rotate -loop-accesses -lazy-branch-prob -lazy-block-freq -
opt-remark-emitter -loop-distribute -branch-prob -block-freq -scalar-evolution -basicaa -aa -
loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-
vectorize -loop-simplify -scalar-evolution -aa -loop-accesses -lazy-branch-prob -lazy-block-
freq -loop-load-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-freq -opt-remark-emitter -slip-vectorizer -opt-remark-emitter -
instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolution -loop-unroll -lazy-
branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa -
verification -lcssa -scalar-evolution -lcls -lazy-branch-prob -lazy-block-freq -opt-remark-
emitter -transform-warning -alignment-from-assumptions -strip-dead-prototypes -globaladce -
constmerge -domtree -loops -branch-prob -block-freq -loop-simplify -lcssa-verification -lcssa -
basicaa -aa -scalar-evolution -block-freq -loop-sink -lazy-branch-prob -lazy-block-freq -opt-
remark-emitter -instsimplify -div-rem-pairs -simplifycfg -verify
```

HEURISTIC-BASED OPTIMIZATION

```
1 // the algorithm: functional description of matrix multiplication
2 Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
3 prod(x, y) += A(x, r) * B(r, y);
4 out(x, y) = prod(x, y);
5
6 // schedule for Nvidia GPU
7 const int warp_size = 32; const int vec_size = 2;
8 const int x_tile = 3; const int y_tile = 4;
9 const int y_unroll = 8; const int r.unroll = 1;
10 Var xi,yi,xio,xii,yi,xo,yo,x_pair,xiyo,ty; RVar rxo,rxi;
11 out.bound(x, 0, size).bound(y, 0, size)
12   .tile(x, y, xi, yi, x.tile * vec_size * warp_size,
13         y.tile * y.unroll)
14   .split(xi, ty, yi, y.unroll)
15   .vectorize(xi, vec_size)
16   .split(xio, xii, xii, warp_size)
17   .reorder(xi, yi, xii, ty, x, y)
18   .unroll(xi).unroll(yi)
19   .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
20 prod.store(inMemoryType:Register).compute_at(out, x)
21   .split(x, xo, xi, warp_size * vec_size, RoundUp)
22   .split(y, ty, y.unroll)
23   .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
24   .unroll(xo).update()
25   .split(x, xo, warp_size * vec_size, RoundUp)
26   .split(y, ty, y.unroll)
27   .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
28   .split(r_xo, rxi, warp_size)
29   .unroll(rxo, r.unroll).reorder(xi, xo, y, rxi, ty, rxo)
30   .unroll(xo).unroll(yi);
31 Var Bx = B.in().args()[0], By = B.in().args()[1];
32 Var Ax = A.in().args()[0], Ay = A.in().args()[1];
33 B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
34   .gpu_lanes(xi).unroll(xo).unroll(By);
35 A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
36   .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(yo);
37   .split(Ay,yi,yi,yi).gpu_threads(yi).unroll(yo);
38 A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
39   .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
40   .unroll(xo).unroll(Ay);
```

SCHEDULE-BASED OPTIMIZATION

# THE OPTIMIZATION CHALLENGE



**RELYING ON LIBRARIES**

**HEURISTIC-BASED OPTIMIZATION**

**HOW TO ENCODE AND APPLY  
DOMAIN-SPECIFIC OPTIMIZATIONS?**

Pass Arguments: -targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-info -forceattrs -inferattrs -domtree -callsite-splitting -ipsccp -called-value-propagation -attributor -globalopt -domtree -mem2reg -deadgelim -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -basiccg -globals -aa -prune -eh -inline -functionattrs -argpromotion -domtree -sroa -basicaa -aa -memoryss -early-cse -memssa -speculative-execution -basicaa -aa -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -domtree -aggressive-instcombine -basicaa -aa -loops -lazy-branch-prob -block-free -lazy-block-freq -opt-remark-emitter -instcombine -libcalls-shrinkwrap -loops -branch-prob -block-free -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -basicaa -aa -scalar-evolution -loop-rotate -lcm -loop-unswitch -simplifycfg -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -lcssa -scalar-evolution -indvars -loop-idiom -loop-deletion -loop-unroll -mldst-motion -phi-values -basicaa -aa -memdep -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -gvn -phi-values -basicaa -aa -memdep -memcpops -sccp -demanded-bits -bdce -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -lazy-value-info -jump-threading -correlated-propagation -basicaa -aa -phi-values -memdep -dse -loops -loop-simplify -lcssa -verification -lcssa -basicaa -aa -scalar-evolution -lcls -postdomtree -adce -simplifycfg -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -barrier -elim-avail-extern -basiccg -rpo-functionattrs -globalopt -globaldce -basiccg -globals -aa -float2int -domtree -loops -loop-simplify -lcssa -verification -lcssa -basicaa -aa -scalar-evolution -loop-rotate -loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-distribute -branch-prob -block-free -scalar-evolution -basicaa -aa -loops -accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -basicaa -aa -loop-accesses -lazy-branch-prob -lazy-block-freq -instcombine -barrier -elim-avail-extern -basiccg -rpo-functionattrs -globalopt -globaldce -basiccg -globals -aa -float2int -domtree -loops -loop-simplify -lcssa -scalar-evolution -loop-unroll -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa -scalar-evolution -lcls -loop-simplify -lcssa -verification -lcssa -scalar-evolution -lcls -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -transform-warning -alignment-from-assumptions -strip-dead-prototypes -globalde -constmerge -domtree -loops -branch-prob -block-freq -loop-simplify -lcssa -verification -lcssa -basicaa -aa -scalar-evolution -block-free -loop-simpl -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instsimplify -div-rem-pairs -simplifycfg -verify

```
1 // the algorithm: functional description of matrix multiplication
2 Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
3 prod(x, y) += A(x, r) * B(r, y);
4 out(x, y) = prod(x, y);
5
6 // schedule for Nvidia GPUs
7 const int warp_size = 32; const int vec_size = 2;
8 const int x_file = 3; const int y_file = 4;
9 const int y_unroll = 8; const int r.unroll = 1;
10 Var xi,yi,xio,yii,xo,yo,xo.y.pair,xiyo,ty; RVar rxo,rxi;
11 out.bound(x, 0, size).bound(y, 0, size)
12 .tile(x, y, xi, yi, x_file * vec_size * warp_size,
13 .y_file * y_unroll)
14 .split(x, ty, yi, y_file)
15 .vectorize(xi, vec_size)
16 .split(x, xio, xii, warp_size)
17 .reorder(xio, yi, xii, ty, x, y)
18 .unroll(xio).unroll(yi)
19 .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
20 prod.store(inMemoryType:Register).compute_at(out, x)
21 .split(x, xo, xi, warp_size + vec_size, RoundUp)
22 .split(y, ty, y, y.unroll())
23 .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
24 .unroll(xo).unroll(yi).update()
25 .split(x, xo, xi, warp_size + vec_size, RoundUp)
26 .split(y, ty, y, y.unroll())
27 .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
28 .split(r.xo, rxi, warp_size)
29 .unroll(xi, r.unroll).reorder(xi, xo, y, rxi, ty, rxo)
30 .unroll(xo).unroll(yi);
31 Var Bx = B.in().args()[0], By = B.in().args()[1];
32 Var Ax = A.in().args()[0], Ay = A.in().args()[1];
33 B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
34 .gpu_lanes(xi).unroll(xo).unroll(yi);
35 A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
36 .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
37 .split(Ay,yo,yi,y.tile).gpu_threads(yi).unroll(yo);
38 A.in().in().compute_at(prod, rxo).vectorize(Ax, vec_size)
39 .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
40 .unroll(xo).unroll(Ay);
```

**SCHEDULE-BASED OPTIMIZATION**

# THE OPTIMIZATION CHALLENGE



RELYING ON LIBRARIES



Pass Arguments: -targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-info -forceattrs -inferattrs -domtree -callsite-splitting -ipsccp -called-value-propagation -attributor -globalopt -domtree -mem2reg -deadrgelim -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -basicgg -globals -aa -prune -eh -inline -functionattrs -argpromotion -domtree -srop -basicaa -aa -memoryss -early-cse -memssa -speculative-execution -basicaa -aa -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -domtree -aggressive -instcombine -basicaa -aa -loops -lazy-branch-prob -block-free -lazy-block-freq -opt-remark-emitter -instcombine -libcalls-shrinkwrap -loops -branch-prob -block-free -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -pgo-memop-opt -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -basicaa -aa -scalar-evolution -loop-rotate -lcm -loop-unswitch -simplifycfg -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa -scalar-evolution -indvars -loop-idiom -loop-deletion -loop-unroll -mldst-motion -phi-values -basicaa -aa -memdep -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -gvn -phi-values -basicaa -aa -memdep -memcpops -sccp -demanded-bits -bdce -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -lazy-value-info -jump-threading -correlated-propagation -basicaa -aa -phi-values -memdep -dse -loops -loop-simplify -lcssa -verification -lcssa -basicaa -aa -scalar-evolution -lcls -postdomtree -adce -simplifycfg -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -barrier -elim-avail-extern -basicgg -rpo-functionattrs -globalopt -globaldce -basicgg -globals -aa -float2int -domtree -loops -loop-simplify -lcssa -verification -lcssa -basicaa -aa -scalar-evolution -loop-rotate -loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-distribute -branch-prob -block-free -scalar-evolution -basicaa -aa -loops -accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -scalar-evolution -aa -loop-accesses -lazy-branch-prob -lazy-block-freq -instcombine -barrier -elim-avail-extern -basicgg -rpo-functionattrs -globalopt -globaldce -basicgg -globals -aa -float2int -domtree -loops -loop-simplify -lcssa -scalar-evolution -loop-unroll -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa -verification -lcssa -scalar-evolution -lcls -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -transform-warning -alignment-from-assumptions -strip-dead-prototypes -globalde -constmerge -domtree -loops -branch-prob -block-freq -loop-simplify -lcssa -verification -lcssa -basicaa -aa -scalar-evolution -block-free -loop -loop -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instsimplify -div -rem-pairs -simplifycfg -verify

HEURISTIC-BASED OPTIMIZATION

## HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS?

```
1 // the algorithm: functional description of matrix multiplication
2 Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
3 prod(x, y) += A(x, r) * B(r, y);
4 out(x, y) = prod(x, y);
5
6 // schedule for Nvidia GPUs
7 const int warp_size = 32; const int vec_size = 2;
8 const int x_file = 3; const int y_file = 4;
9 const int y_unroll = 8; const int r.unroll = 1;
10 Var xi,yi,xio,yii,xo,yo,x.y.pair,xiyo,ty; RVar rxo,rxi;
11 out.bound(x, 0, size).bound(y, 0, size)
12 .tile(x, y, xi, yi, x.tile * vec.size * warp.size,
13 .y.tile * y.unroll)
14 .split(x, ty, yi, y.unroll)
15 .vectorize(xi, vec.size)
16 .split(x, xio, xii, warp.size)
17 .reorder(xio, yi, xii, ty, x, y)
18 .unroll(xio).unroll(yi)
19 .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
20 prod.store(inMemoryType:Register).compute_at(out, x)
21 .split(x, xo, xi, warp.size + vec.size, RoundUp)
22 .split(y, ty, y, y.unroll)
23 .gpu_threads(ty).unroll(xi, vec.size).gpu_lanes(xi)
24 .unroll(xo).unroll(y).update()
25 .split(x, xo, warp.size + vec.size, RoundUp)
26 .split(y, ty, y, y.unroll)
27 .gpu_threads(ty).unroll(xi, vec.size).gpu_lanes(xi)
28 .split(r.x, rxo, rxi, warp.size)
29 .unroll(rxo, r.unroll).reorder(xi, xo, y, rxi, ty, rxo)
30 .unroll(xo).unroll(y);
31 Var Bx = B.in().args()[0], By = B.in().args()[1];
32 Var Ax = A.in().args()[0], Ay = A.in().args()[1];
33 B.in().compute_at(prod, ty).split(Bx, xo, xi, warp.size)
34 .gpu_lanes(xi).unroll(xo).unroll(By);
35 A.in().compute_at(prod, rxo).vectorize(Ax, vec.size)
36 .split(Ax,xo,xi,warp.size).gpu_lanes(xi).unroll(xo)
37 .split(Ay,yo,yi,y.tile).gpu_threads(yi).unroll(yo);
38 A.in().in().compute_at(prod, rxo).vectorize(Ax, vec.size)
39 .split(Ax, xo, xi, warp.size).gpu_lanes(xi)
40 .unroll(xo).unroll(Ay);
```

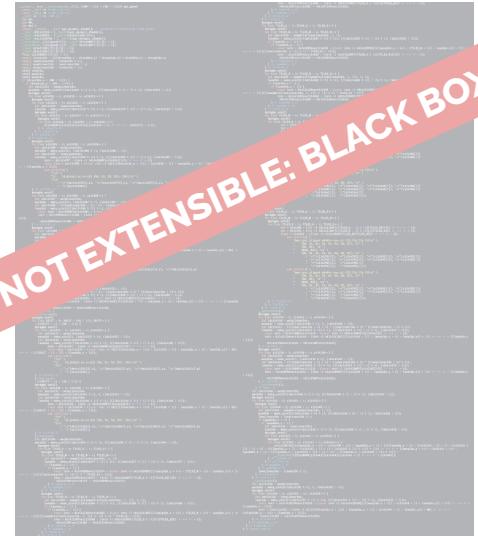
SCHEDULE-BASED OPTIMIZATION

# THE OPTIMIZATION CHALLENGE

## HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS?



RELYING ON LIBRARIES



Pass Arguments: -targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-info -forceattrs -inferattrs -domtree -callsite-splitting -ipsccp -called-value-propagation -attributor -globalopt -domtree -mem2reg -deadagelim -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -basiccc -globals -aa -prune -eh -inline -functionattrs -argpromotion -domtree -sra -basicaa -aa -memoryss -early-cse -memssa -speculative-execution -basicaa -aa -lazy-value-info -jump-threading -correlated-propagation -simplifycfg -domtree -lazy -aggressive -instcombine -basicaa -aa -loops -lazy-branch-prob -block-freq -opt-remark-emitter -instcombine -libcalls-shrinkwrap -loops -branch-prob -block-freq -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -pgo -memop -opt -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcall -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa -basicaa -aa -scalar-evolution -loop-rotate -lcm -loop-unswitch -simplifycfg -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa -verification -lcssa -scalar-evolution -indvars -loop-idiom -loop-deletion -loop-unroll -mldst-motion -phi-values -basicaa -aa -memdep -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -gvn -phi-values -basicaa -aa -memdep -memcpys -scpp -demanded-bits -bdce -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -lazy-value-info -jump-threading -correlated-propagation -basicaa -aa -phi-values -memdep -dse -loops -loop-simplify -lcssa -verification -lcssa -basicaa -aa -scalar-evolution -llic -postdomtree -adce -simplifycfg -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -barrier -elim-avail-extern -basicaa -rpo -functionattrs -globalopt -globaladce -basiccc -globals -aa -float2int -domtree -loops -loop-simplify -lcssa -verification -lcssa -basicaa -aa -scalar-evolution -loop -rotate -loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-distribute -branch-prob -block-freq -scalar-evolution -basicaa -aa -loops -accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -lcssa -scalar-evolution -aa -loop-accesses -lazy-branch-prob -lazy-block-freq -instcombine -load-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-freq -opt-remark-emitter -slip-vectorizer -opt-remark-emitter -instcombine -loop-simplify -lcssa -verification -lcssa -scalar-evolution -loop -unroll -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa -scalar-evolution -llic -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -transform-warning -alignment-from-assumptions -strip-dead-prototypes -globaladce -constmerge -domtree -loops -branch-prob -block-freq -loop-simplify -lcssa -verification -lcssa -basicaa -aa -scalar-evolution -block-freq -loop-sink -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instsimplify -div -rem-pairs -simplifycfg -verify

HEURISTIC-BASED OPTIMIZATION

```
1 // the algorithm: functional description of matrix multiplication
2 Var x(*x), y(*y); Func prod("prod"); RDom r(0, size);
3 prod(x, y) += A(x, r) * B(r, y);
4 out(x, y) = prod(x, y);
5
6 // schedule for Nvidia GPU
7 const int warp_size = 32; const int vec_size = 2;
8 const int x_tile = 3; const int y_tile = 4;
9 const int y_unroll = 8; const int r.unroll = 1;
10 Var xi,yi,xio,yii,xo,yo,xo.y.pair,xiyo,ty; RVar rxo,rxi;
11 out.bound(x, 0, size).bound(y, 0, size);
12 .tile(x, y, xi, yi, x.tile * vec.size * warp.size,
13 .y.tile * y.unroll)
14 .split(x, ty, yi, y.tile * y.unroll)
15 .vectorize(xi, vec.size)
16 .split(x, xio, xii, warp.size)
17 .reorder(xio, yi, xii, ty, x, y)
18 .unroll(xio).unroll(yi)
19 .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
20 prod.store(inMemoryType:Register).compute_at(out, x)
21 .split(x, xo, xi, warp.size + vec.size, RoundUp)
22 .split(y, ty, y, y.unroll)
23 .gpu_threads(ty).unroll(xi, vec.size).gpu_lanes(xi)
24 .unroll(xo).unroll(yi).update()
25 .split(x, xo, warp.size + vec.size, RoundUp)
26 .split(y, ty, y, y.unroll)
27 .gpu_threads(ty).unroll(xi, vec.size).gpu_lanes(xi)
28 .split(r.x, rxo, rxi, warp.size)
29 .unroll(rxo, r.unroll).reorder(xi, xo, y, rxi, ty, rxo)
30 .unroll(xo).unroll(yi);
31 Var Bx = B.in().args()[0], By = B.in().args()[1];
32 Var Ax = A.in().args()[0], Ay = A.in().args()[1];
33 B.in().compute_at(prod, ty).split(Bx, xo, xi, warp.size)
34 .gpu_lanes(xi).unroll(xo).unroll(yi);
35 A.in().compute_at(prod, rxo).vectorize(Ax, vec.size)
36 .split(Ax,xo,xi,warp.size).gpu_lanes(xi).unroll(xo)
37 .split(Ay,yo,yi,y.tile).gpu_threads(yi).unroll(yo);
38 A.in().in().compute_at(prod, rxo).vectorize(Ax, vec.size)
39 .split(Ax, xo, xi, warp.size).gpu_lanes(xi)
40 .unroll(xo).unroll(Ay);
```

SCHEDULE-BASED OPTIMIZATION

# THE OPTIMIZATION CHALLENGE

# HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS?



A black and white background showing a dense grid of binary code characters (0s and 1s) across the entire screen. The text "NOT EXTENSIBLE: BLACK BOX" is overlaid diagonally from the bottom-left corner to the top-right corner in a large, bold, sans-serif font.

## **RELYING ON LIBRARIES**

## **HEURISTIC-BASED OPTIMIZATION**

## SCHEDULE-BASED OPTIMIZATION

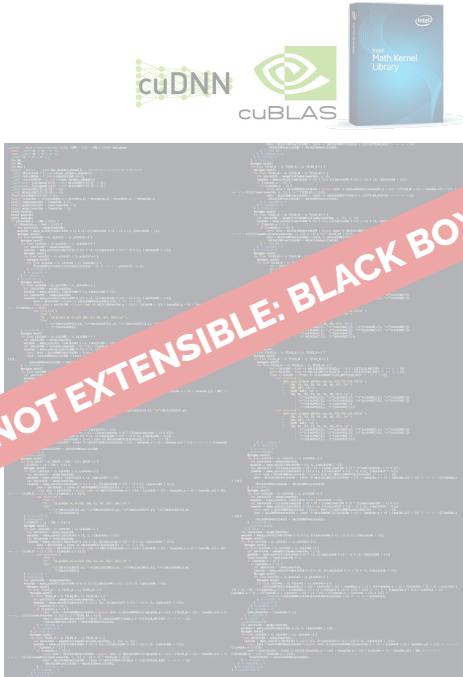
```

1 // the algorithm: functional description of matrix multiplication
2 Var x(x), y(y); Func prod("prod"); RDom r(0, size);
3 prod(x, y) += Ax(r) * Bt(r, y);
4 out(x, y) = prod(x, y);
5
6 // schedule for Nvidia GPUs
7 const int warp_size = 32; const int vec_size = 2;
8 const int x_tile = 3; const int y_tile = 4;
9 const int y_unroll = 8; const int r_unroll = 1;
10 Var xi,yi,xii,yii,xi,yo,xi,yo,xi,yo,xi,yo;
11 out.bound(x, 0, size).bound(y, 0, size)
12 .title(x, y, xi, yi, x_tile * vec_size * warp_size,
13 y_tile * y_unroll);
14 .split(yi, ty, yi, y.unroll());
15 .vectorize(xi, vec_size);
16 .split(xi, xio, xii, warp_size)
17 .reorder(xio, yi, xii, ty, x, y)
18 .unroll(xio).unroll(yi);
19 .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
20 prod.store_in(MemoryType::Register).compute_at(out, x);
21 .split(x, xo, xi, warp_size * vec_size, RoundUp);
22 .split(y, ty, y, y.unroll);
23 .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi);
24 .unroll(xo).unroll(yi).update();
25 .split(x, xo, xi, warp_size * vec_size, RoundUp);
26 .split(y, ty, y, y.unroll);
27 .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi);
28 .split(r, rxo, rxi, warp_size);
29 .unroll(rxo, r.unroll()).reorder(xi, xo, y, rxi, ty, rxo);
30 .unroll(xo).unroll(yi);
31
32 Var B = B.in().args()[10], By = B.in().args()[11];
33 Var Ax = A.in().args()[8], Ay = A.in().args()[11];
34 B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
35 .gpu_lanes(xi).unroll(xo).unroll(By);
36 A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
37 .split(Ax, xo, xi, warp_size).gpu_lanes(xi).unroll(xo)
38 .split(yo, yo, yi, ty, file).gpu_threads(ti).unroll(yo);
39 A.in().in().compute_at(prod, rxii).vectorize(Ac, vec_size)
40 .split(Ac, xo, xi, warp_size).gpu_lanes(xi)
41 .unroll(xo).unroll(Ay);

```

# THE OPTIMIZATION CHALLENGE

# HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS?



A black rectangular background featuring a faint, grayscale watermark of a circuit board pattern. A large, solid red diagonal banner runs from the bottom-left corner to the top-right corner. The banner contains the text "NOT EXTENSIBLE: BLACK BOX" in a white, sans-serif font.

**NO CONTROL: "ONE-SIZE-FITS-ALL"**

```
prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -basicic  
-eh -inline -functionattrs -argpromotion -dmdtree -sroa -basicaaa -  
messia -speculative-execution -basicaaa -aa -lazy-value-info -jumo  
propagation -simplifyfcg -dmdtree -aggressive -instcombine -b  
-lazy-block-freq -opt-remark-emitter -instcombine -lhc -branch-prob  
-block-freq -lazy-branch-prob -lazy-block-freq -opt-  
loops -lazy-branch-prob -lazy-block-freq -opt-  
reassociate -dmdtree -loops -loop-simplif  
-evolution -loop-rotate -lhc -loop -basicaaa -aa -loops -lazy  
-branch-prob -branch-freq -branch-prob -branch -loop-simplif -lcssa  
-verification -lcssa -  
-motion -phi-value -  
-gvn -phi-value -  
-lazy-block-freq -opt-remark-emitter -instcombine -lazy-value-info -jumo  
-basicic -aa -scalar-evolution -basicaaa -aa -phi-value -memdep -loops -loop-simplif -  
-aa -loops -lazy-branch-prob -lazy-block-freq -postdmdtree -adce -simplifycfg  
-barrier -elim -allow-extrem -basicaaa -rpo-functionattrs -globalobj -globalbde -  
-globals -aa -float2int -dmdtree -loops -loop-simplif -lcssa -verification -lcssa  
-basicaaa -aa -scalar-evolution -loop -rotate -loop-accesses -lazy -branch -prob -lazy -block -freq  
-opt-remark-emitter -loop -distribution -branch -prob -block -freq -scalar -evolution -basicaaa -aa  
-loop -accesses -demanded -bits -lazy -branch -prob -lazy -block -freq -opt-remark -emitter -loop  
-vectorize -loop -simplif -scalar -evolution -ad -loop -accesses -lazy -branch -prob -lazy -block -freq  
-loop -load -elide -basicaaa -aa -lazy -branch -prob -lazy -block -freq -opt-remark -emitter -  
-instcombine -simplifycfg -dmdtree -loops -scalar -evolution -basicaaa -aa -demanded -bits -lazy  
-branch -prob -lazy -block -freq -opt-remark -emitter -slp -vectorizer -opt -remark -emitter -  
-instcombine -loop -simplif -lcssa -verification -lcssa -scalar -evolution -loop -unroll -lazy  
-branch -prob -lazy -block -freq -opt -remark -emitter -instcombine -loop -simplif -lcssa -  
-verification -lcssa -scalar -evolution -lhc -lazy -branch -prob -lazy -block -freq -opt -remark  
-emitter -transform -warning -alignment -from -assumption -strip -dead -prototypes -globalbde  
-consmere -dmdtree -loops -branch -prob -block -freq -loop -simplif -lcssa -verification -lcssa  
-basicaaa -aa -scalar -evolution -block -freq -loop -simplif -lazy -branch -prob -lazy -block -freq -opt -  
-remark -emitter -instcombine -div -rem -pairs -simplifyfcg -verify
```

## **RELYING ON LIBRARIES**

# HEURISTIC-BASED OPTIMIZATION



 tvm halide / Halide

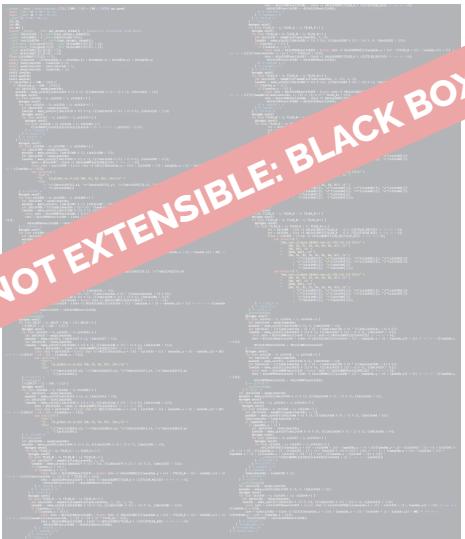
```

1 // the algorithm: functional description of matrix multiplication
2 Var x("x", y("y")); Func prod("prod"); RDom r(0, size);
3 prod(x, y) := A(x, r) * B(r, y);
4 out(x, y) = prod(x, y);
5
6 // schedule for Nvidida GPUs
7 const int warp_size = 32; const int vec_size = 2;
8 const int x_tile_ = 3; const int y_tile_ = 4;
9 const int y_unroll_ = 8; const int r_unroll_ = 1;
10 Var xi,yi,xo,yi,yo,y,xo,pair,xio,ty; RVar rxo,rxi;
11 out.bound(x, 0, size).bound(y, 0, size)
12 .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
13       y_tile * y_unroll);
14 .split(yi, ty, yi * y_unroll)
15   .vectorize(xi, vec_size)
16   .split(xi, xio, xii, warp_size)
17   .reorder(xio, yi, xii, ty, x, y)
18   .unroll(xio).unroll(yi)
19   .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
20 prod.state_in(MemoryType::Register).compute_at(out, x)
21 .split(x, xo, xi, warp_size + vec_size, RoundUp)
22 .split(y, ty, y, y.unroll)
23 .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
24 .unroll(xo).unroll(yi).update()
25 .split(x, xo, xi, warp_size + vec_size, RoundUp)
26 .split(y, ty, y, y.unroll)
27 .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
28 .split(x, rxo, rxi, warp_size)
29 .unroll(rxo).unroll(rxi).reorder(xi, xo, y, rxi, ty, rxo)
30 .unroll(xo).unroll(yi);
31 Var Bx = B.in().args()[0], By = B.in().args()[1];
32 Var Ax = A.in().args()[0], Ay = A.in().args()[1];
33 B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
34   .gpu_lanes(xi).unroll(xo).unroll(By);
35 A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
36   .split(Ax, xo, xi, warp_size).gpu_lanes(xi).unroll(xo)
37   .split(Ay, yo, yi, y.tile).gpu_threads(yi).unroll(yo);
38 A.in().in().compute_at(prod, rxj).vectorize(Ay, vec_size)
39   .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
40   .unroll(xo).unroll(Ay);

```

## SCHEDULE-BASED OPTIMIZATION

# THE OPTIMIZATION CHALLENGE



NOT EXTENSIBLE: BLACK BOX

Pass Arguments: -targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-info -forceattrs -inferattrs -domtree -callsite-splitting -ipsccp -called-value-propagation -attribute -globalopt -domtree -memreg -deadgelim -domtree -basicaa -aa -loops -lazy-block-prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -basicaa -aa -eh -inline -functionattrs -argpromotion -domtree -sroa -basicaa -aa -memssa -speculative-execution -basicaa -aa -lazy-value-info -jump-propagation -simplifycfg -domtree -aggressive-instcombine -basicaa -aa -loops -lazy-block-prob -lazy-block-freq -opt-remark-emitter -instcombine -lcssa -scalar-evolution -loop-unroll -mdst-motion -phi-values -loop-simplify -memcprop -sccp -demanded-bits -bdce -basicaa -aa -loops -lazy-block-prob -lazy-block-freq -opt-remark-emitter -instcombine -lcssa -basicaa -aa -scalar-evolution -loop-rotate -lcm -loop -loop-simplify -memcprop -sccp -domtree -basicaa -aa -loops -lazy-block-prob -lazy-block-freq -opt-remark-emitter -instcombine -lcssa -basicaa -aa -scalar-evolution -loop-rotate -lcm -loop -loop-simplify -memcprop -sccp -demanded-bits -bdce -basicaa -aa -loops -lazy-block-prob -lazy-block-freq -opt-remark-emitter -instcombine -lcssa -basicaa -aa -phi-values -memdep -dse -loops -loop-simplify -lcssa -basicaa -aa -scalar-evolution -lisc -postdomtree -adce -simplifycfg -basicaa -aa -loops -lazy-block-prob -lazy-block-freq -opt-remark-emitter -barrier -elim-avail-extern -basiccg -rpo-functionattrs -globalopt -globalde -lcssa -aa -globals -aa -float2int -domtree -loops -loop-simplify -lcssa -verification -lcssa -lcssa -aa -scalar-evolution -loop-rotate -loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-distribute -branch-prob -block-freq -scalar-evolution -basicaa -aa -loops -accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -scalar-evolution -aa -loops -accesses -lazy-branch-prob -lazy-block-freq -loop-load -elim -basicaa -aa -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -domtree -loops -scalar-evolution -basicaa -aa -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -slip-vectorizer -opt-remark-emitter -instcombine -loop-simplify -lcssa -verification -lcssa -scalar-evolution -loop-unroll -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa -verification -lcssa -scalar-evolution -lisc -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -transform-warning -alignment-from-assumptions -strip-dead-prototypes -globalde -constmerge -domtree -loops -branch-prob -block-freq -loop-simplify -lcssa -verification -lcssa -basicaa -aa -scalar-evolution -block-freq -loop-sink -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instsimplify -div-rem-pairs -simplifycfg -verify

RELYING ON LIBRARIES

HEURISTIC-BASED OPTIMIZATION

## HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS?



NO CONTROL: "ONE-SIZE-FITS-ALL"



```
1 // the algorithm: functional description of matrix multiplication
2 Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
3 prod(x, y) += A(x, r) * B(r, y);
4 out(x, y) = prod(x, y);
5
6 // schedule for Nvidia GPU
7 const int warp.size = 32; const int vec.size = 2;
8 const int x.tile = 3; const int y.tile = 4;
9 const int y.unroll = 8; const int r.unroll = 1;
10 Var xi,yi,xio,xii,yi,xi,yo,yo,x.pair,xiyo,ty; RV;
11 out.bound(x, 0, size).bound(y, 0, size)
12   .tile(x, y, xi, yi, x.tile * vec.size)
13   .tile(y, ty, yi, y.tile * y.unroll);
14 .split(ty, yi, ty);
15 .vectorize(xi, xi);
16 .split(xi, xi);
17 .reorder(xi, xi);
18 .unroll(xi, xi);
19 .unroll(yo, yo);
20 .unroll(x, x);
21 .unroll(y, y);
22 .unroll(x, x).gpu.lanes(xi);
23 .unroll(y, y).gpu.lanes(yi);
24 .type(Register).compute_at(out, x)
25   .x(xo, xi, warp.size * vec.size, RoundUp)
26   .y(yo, yi, y.unroll);
27 .gpu.threads(ty).unroll(xi, vec.size).gpu.lanes(xi);
28 .split(x, xo, r.xi, warp.size)
29 .split(r, xi, r.unroll).reorder(xi, xo, y, rxi, ty, rxo)
30 .unroll(xo).unroll(yo);
31 Var Bx = B.in().args()[0], By = B.in().args()[1];
32 Var Ax = A.in().args()[0], Ay = A.in().args()[1];
33 B.in().compute_at(prod, ty).split(Bx, xo, xi, warp.size)
34   .gpu.lanes(xi).unroll(xo).unroll(By);
35 A.in().compute_at(prod, rxo).vectorize(Ax, vec.size)
36   .split(Ax, xo, xi, warp.size).gpu.lanes(xi).unroll(yo);
37 .split(Ay, yo, yi, y.tile).gpu.threads(yi).unroll(yo);
38 A.in().in().compute_at(prod, rxo).vectorize(Ax, vec.size)
39   .split(Ax, xo, xi, warp.size).gpu.lanes(xi)
40 .unroll(xo).unroll(Ay);
```

NO REUSE: BUILT-IN OPTIMIZATIONS

SCHEDULE-BASED OPTIMIZATION

# THE OPTIMIZATION CHALLENGE:

HOW TO ENCODE AND APPLY  
DOMAIN-SPECIFIC OPTIMIZATIONS?



ation

ATIONS

## THE OPTIMIZATION CHALLENGE:

*How can we encode and apply domain-specific optimizations for high-performance code generation while **providing precise control** and the ability to **define custom optimizations**, thus achieving **a reusable optimization approach** across application domains and hardware architectures?*

NOTES

```
loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -scalar-evolution -aa -loop-accesses -lazy-branch-prob -lazy-block-freq -loop-loop-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-freq -opt-remark-emitter -slp-vectorizer -opt-remark-emitter -instcombine -loop-simplify -lcssa -verification -lcssa -scalar-evolution -loop-unroll -lazy-branch-prob -Lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa -scalar-evolution -lcls -lazy-branch-prob -Lazy-block-freq -opt-remark-emitter -transform-warning alignment-from-assumptions -strip-dead-prototypes -globaldead -constmerge -domtree loops -branch-prob -block-freq -loop-simplify -lcssa -verification -lcssa -basicaa -aa -scalar-evolution -block-freq -loop-sink -lazy-branch-prob -Lazy-block-freq -opt-remark-emitter -instsimplify -div-rem-pairs -simplifycfg -verify
```

```
26 .split(y, ty, y.unroll());
27 .gpu_threads(ty).unroll(x1, vec_size).gpu_lanes(x1);
28 .split(r.x, rxo, rx1, warp_size)
29 .unroll(rx1, r.unroll).reorder(x1, xo, y, rxi, ty, rxo)
30 .unroll(xo).unroll(ty);
31 Var Bx = B.in().args()[0], By = B.in().args()[1];
32 Var Ax = A.in().args()[0], Ay = A.in().args()[1];
33 B.in().compute_at(rod, ty).split(Bx, xo, xi, warp_size)
34 .gpu_lanes(xi).unroll(xo).unroll(By);
35 A.in().compute_at(prod, rso).vectorize(Ax, vec_size)
36 .split(Ax, xo, xi, warp_size).gpu_lanes(xi).unroll(xo)
37 .split(Ay, yo, yi, y, tile).gpu_threads(y1).unroll(yo);
38 A.in().compute_at(prod, rx1).vectorize(Ay, vec_size)
39 .split(Ay, xo, xi, warp_size).gpu_lanes(xi)
40 .unroll(xo).unroll(yo);
```

RELYING ON LIBRARIES

HEURISTIC-BASED OPTIMIZATION

SCHEDULE-BASED OPTIMIZATION

# HIGH PERFORMANCE DOMAIN-SPECIFIC COMPIRATION WITHOUT DOMAIN-SPECIFIC COMPILERS

## PART I: A CASE STUDY

Fireiron: A novel domain-specific compiler for GPUs that outperforms manually tuned high-performance libraries

PACT'20

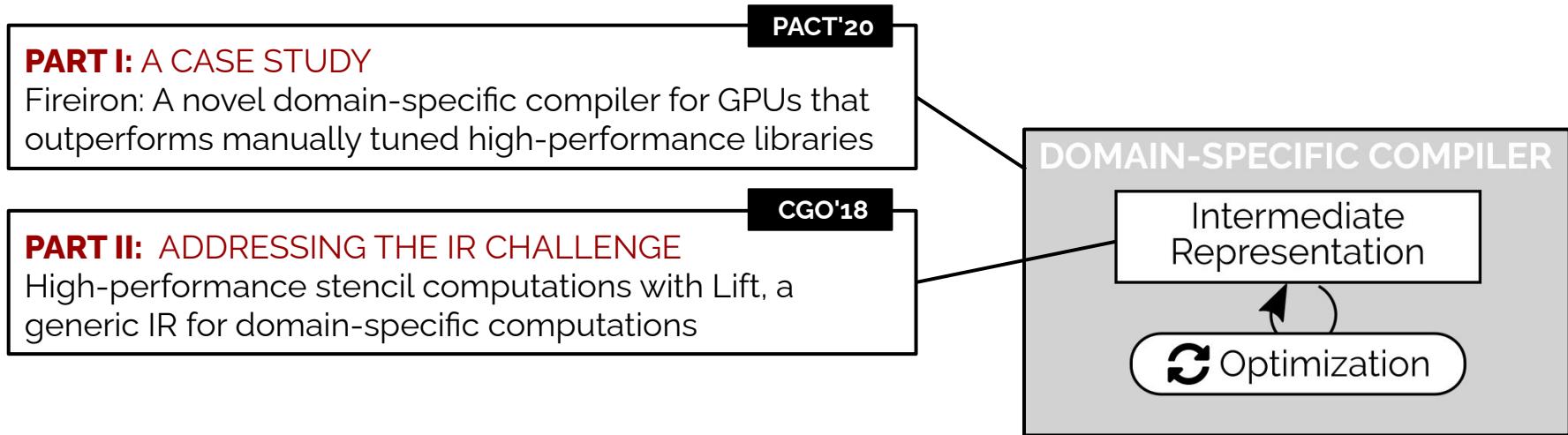
DOMAIN-SPECIFIC COMPILER

Intermediate Representation

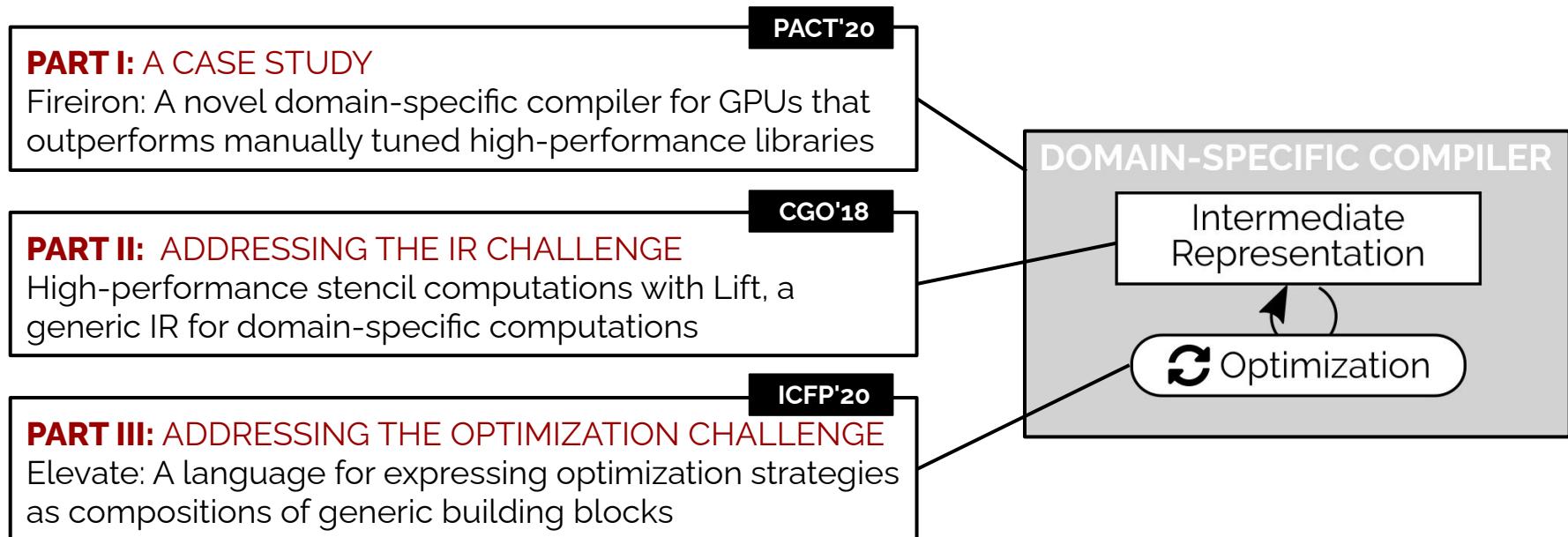
Optimization



# HIGH PERFORMANCE DOMAIN-SPECIFIC COMPIRATION WITHOUT DOMAIN-SPECIFIC COMPILERS



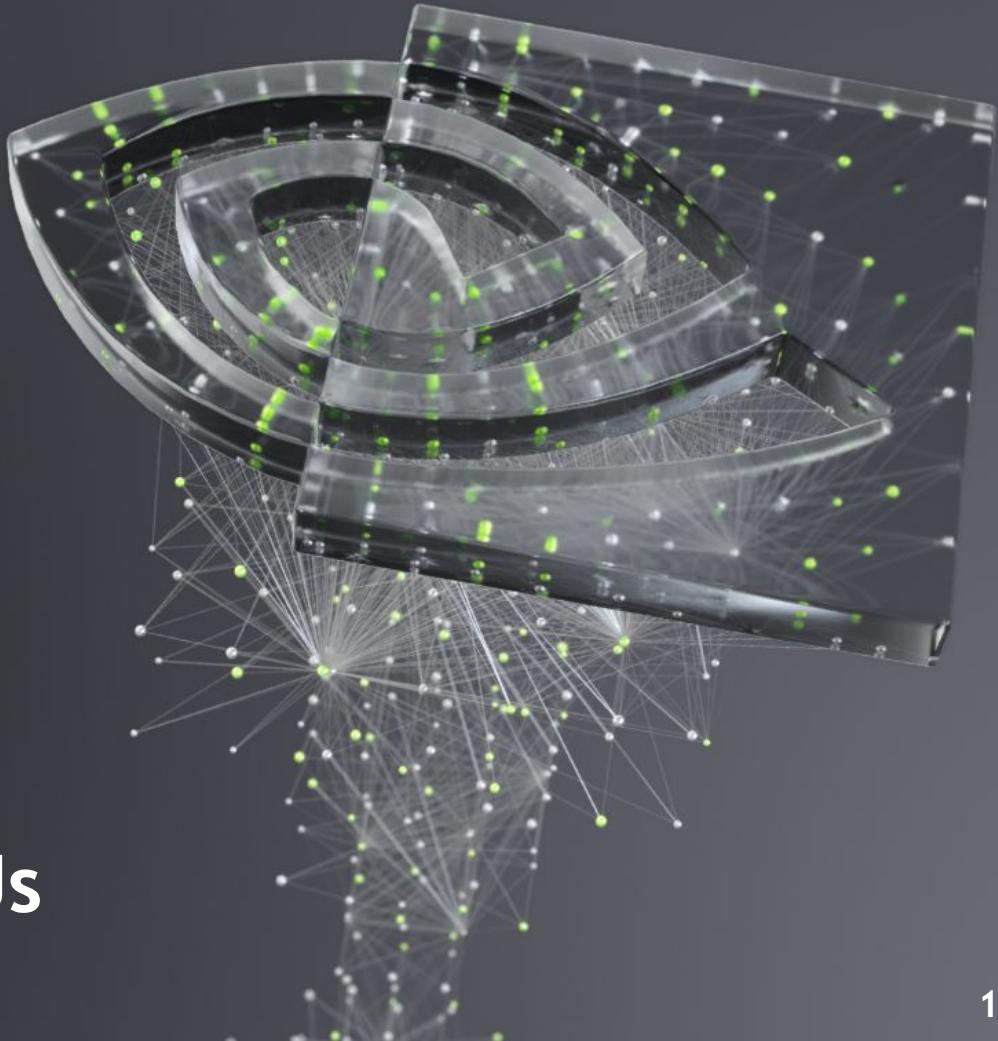
# HIGH PERFORMANCE DOMAIN-SPECIFIC COMPIRATION WITHOUT DOMAIN-SPECIFIC COMPILERS



# PART I: A CASE STUDY



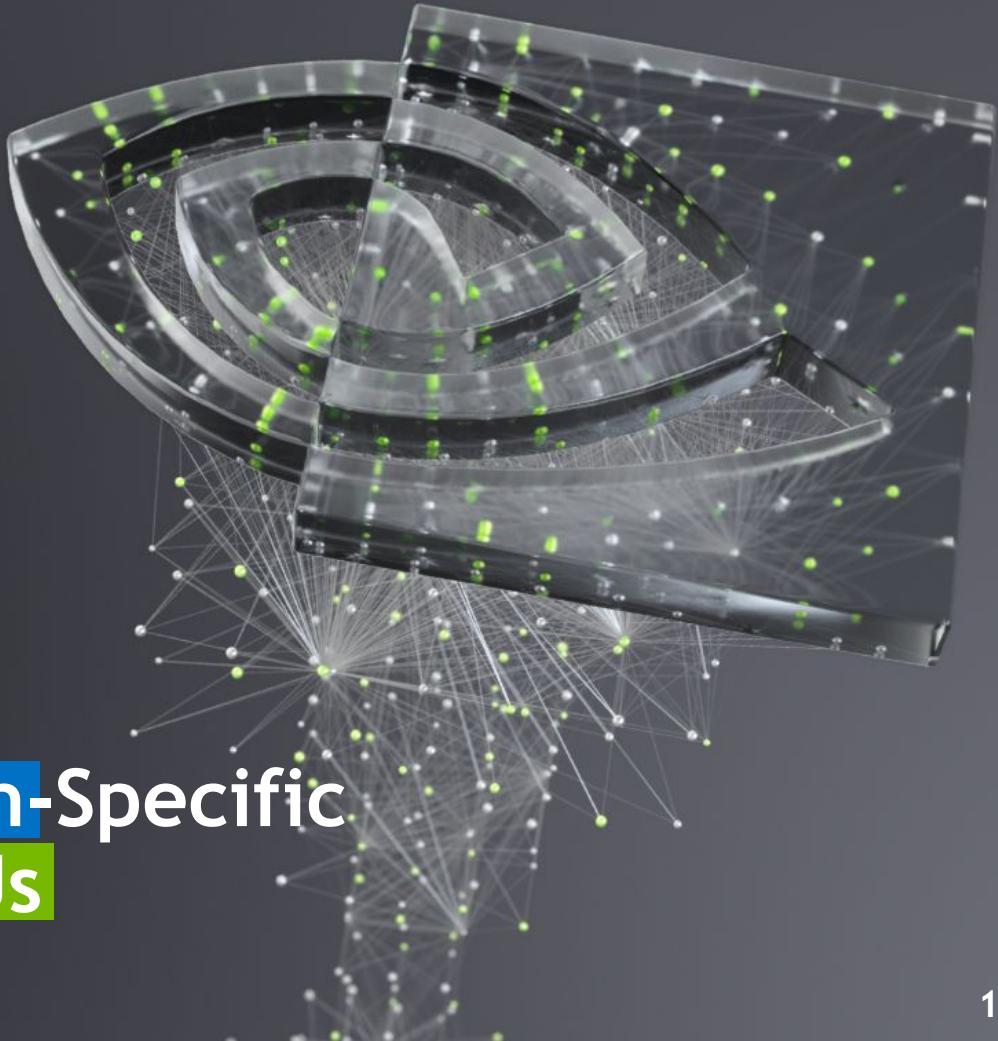
**FIREIRON:**  
Domain-Specific  
Compilation for GPUs



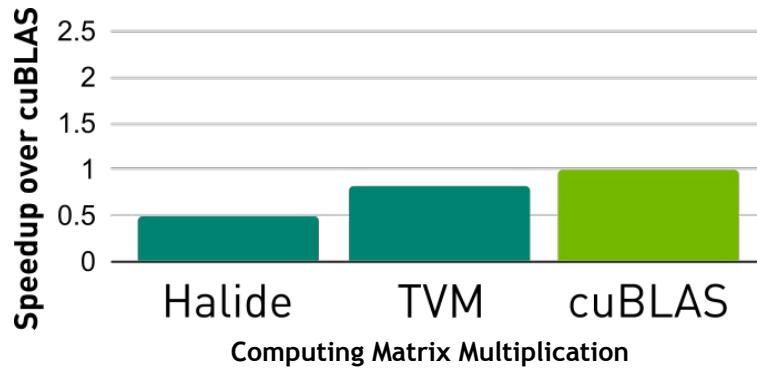
# PART I: A CASE STUDY



**FIREIRON:**  
**Matrix-Multiplication-Specific**  
**Compilation for GPUs**

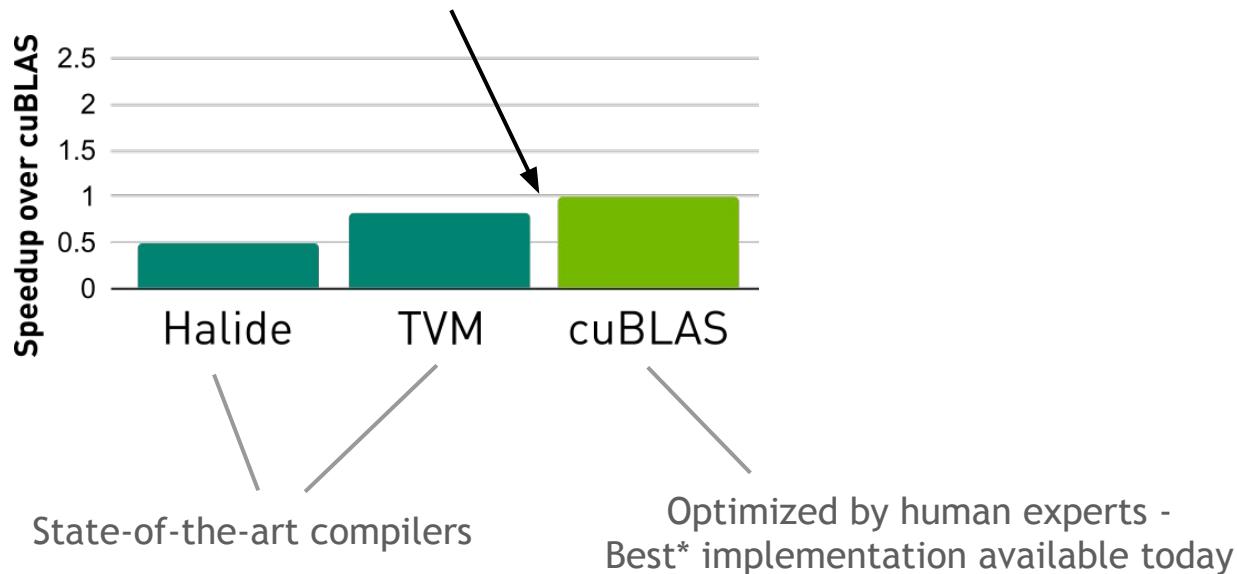


# WHY YET ANOTHER DOMAIN-SPECIFIC COMPILER?

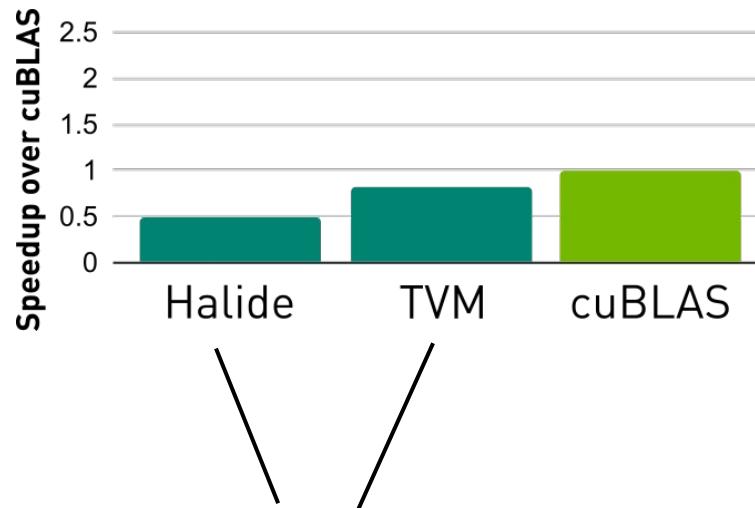


# WHY YET ANOTHER DOMAIN-SPECIFIC COMPILER?

## GAP IN PERFORMANCE

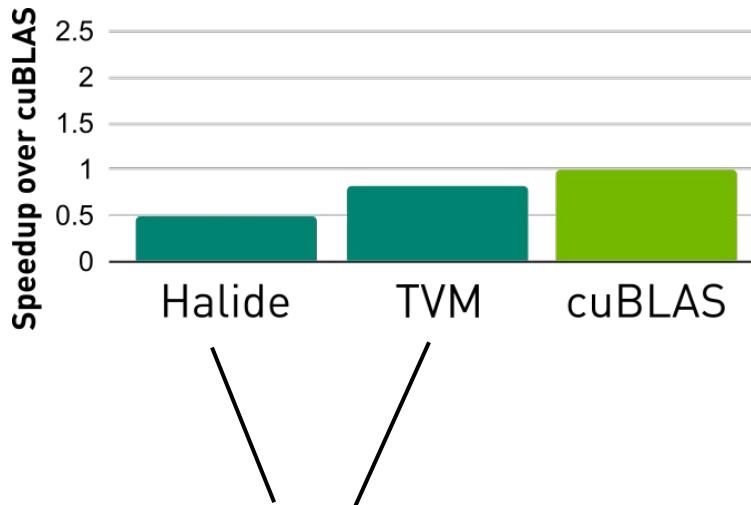


# WHY YET ANOTHER DOMAIN-SPECIFIC COMPILER?



**Data Movements** are treated as second-class concepts!

# WHY YET ANOTHER DOMAIN-SPECIFIC COMPILER?



**Data Movements** are treated as second-class concepts!

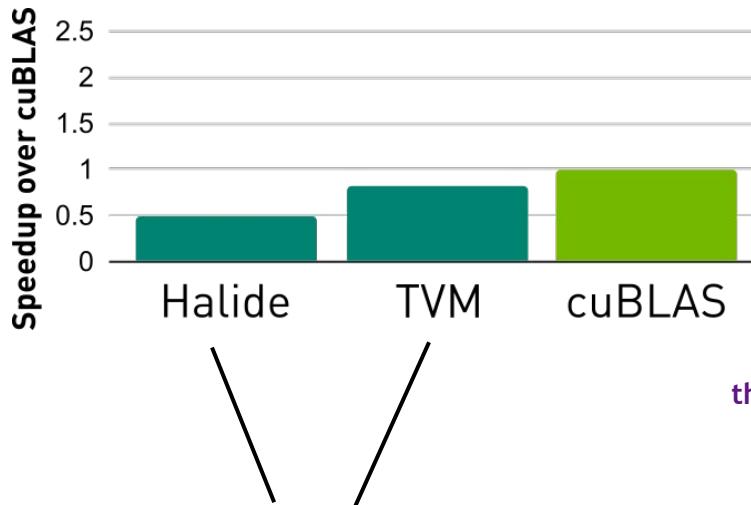
 **tvm** **Matrix Multiplication:**

```
# Naive Algorithm
k = te.reduce_axis((0, K), 'k')
A = te.placeholder((M, K), name='A')
B = te.placeholder((K, N), name='B')
C = te.compute((M, N),
    lambda x,y: te.sum(A[x,k]*B[k,y], axis=k), name='C')
```

- `im.transpose(x, y)` moves iteration over `x` outside of `y` in the traversal order (*i.e.*, this switches from row-major to column-major traversal).
- `im.parallel(y)` indicates that each row of `im` should be computed in parallel across `y`.
- `im.vectorized(x, k)` indicates that `x` should be split into vectors of size `k`, and each vector should be executed using SIMD.
- `im.unroll(k)` indicates that the entire dimension `x` should be unrolled, and the dimension `x` will be of `k`.
- `im.split(x, xi, xi1)` divides the dimension `x` into outer and inner dimensions `xi` and `xi1`, where `xi` ranges from zero to `k`. `xi1` can then be independently marked as parallel, serial, vectorized, or even recursively split.
- `im.tile(x, y, xi, yi, tw, th)` is a convenience method that splits `x` by a factor of `tw`, and `y` by a factor of `th`, then transposes the inner dimension of `y` with the outer dimension of `x` to effect traversal over tiles.

**Schedule Language**  
for Expressing Optimizations

# WHY YET ANOTHER DOMAIN-SPECIFIC COMPILER?



## Matrix Multiplication:

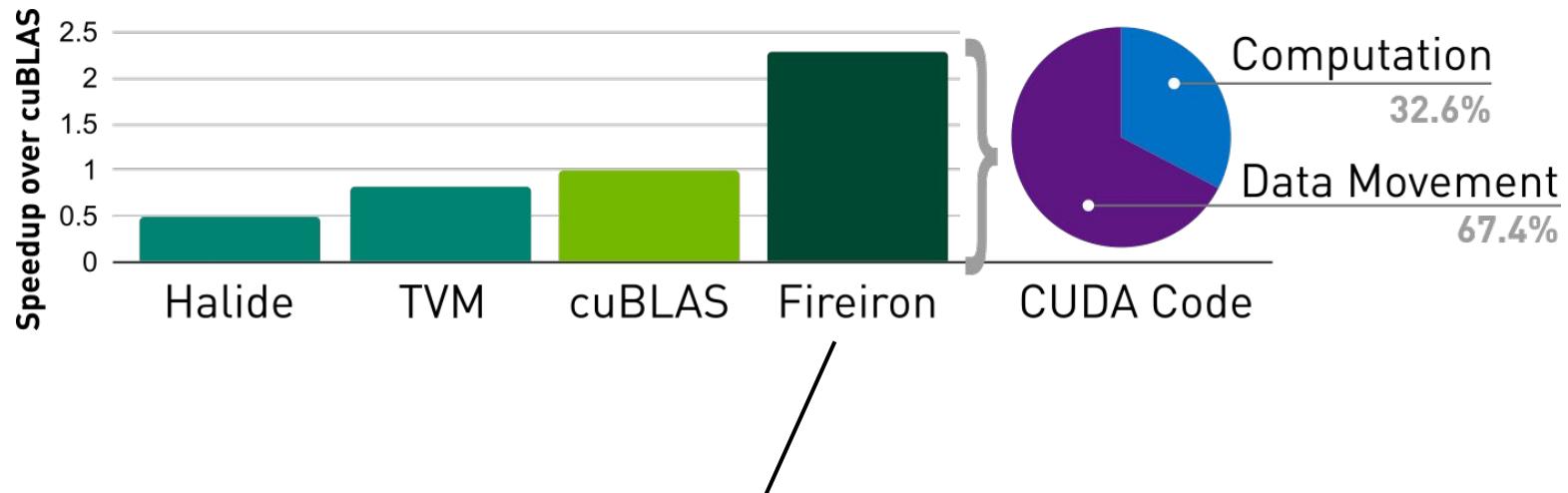
```
# Naive Algorithm
k = te.reduce_axis((0, K), 'k')
A = te.placeholder((M, K), name='A')
B = te.placeholder((K, N), name='B')
C = te.compute((M, N),
    lambda x,y: te.sum(A[x,k]*B[k,y], axis=k), name='C')

# Optimized Algorithm
packedB = te.compute(
    (N/bn,K/bn), lambda x,y,z:B[y,x*bn+z], name='packedB')
C_opt = te.compute((M, N),
    lambda x, y: te.sum(A[x, k] *
        packedB[y // bn, k, tvm.tir.indexmod(y, bn)], axis=k), name = 'C_opt')
```

there is no schedule for expressing data movement optimizations!

**Data Movements** are treated as **second-class** concepts!

# WHY YET ANOTHER DOMAIN-SPECIFIC COMPILER?



**Data Movements** are treated as **first-class concepts!**

by explicitly representing them in our IR and optimizations

# GPU CODE IS HIERARCHICALLY STRUCTURED

```
1 _global_ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
2
3     _shared_ float ASH[128][8], BSH[8][128];
4     float ARF[8][1], BRF[1][8], CRF[8][8];
5
6     iBlock ← 128 * blockIdx.x;
7     jBlock ← 128 * blockIdx.y;
8
9     CRF ← 0;
10
11    for (k ← 0; k < K / 8; k++) {
12
13        GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))
14
15        GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))
16
17        syncthreads();
18
19        iWarp ← iBlock + warpIdx.x * 64;
20        jWarp ← jBlock + warpIdx.y * 32;
21
22        iThread ← iWarp + threadIdx.x * 8;
23        jThread ← jWarp + threadIdx.y * 8
24
25        for (kk ← 0; kk < 8; kk++)
26
27            ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))
28
29            ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))
30
31            for (i ← 0; i < 8; i++)
32                for (j ← 0; j < 8; j++)
33
34                    CRF[i][j] += ARF[i][0] * BRF[0][j];
35
36            endfor
37        endfor
38
39    endfor
40
41
42    endfor
43
44    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
45
46 } // end kernel
```

Matrix Multiplication code  
written in (pseudo) CUDA



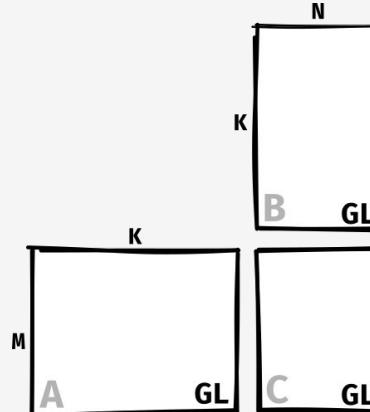
# GPU CODE IS HIERARCHICALLY STRUCTURED

```
1 __global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {  
2  
3     shared__ float ASH[128][8], BSH[8][128];  
4     float ARF[8][1], BRF[1][8], CRF[8][8];  
5  
6     iBlock ← 128 * blockIdx.x;           implements  
7     jBlock ← 128 * blockIdx.y;           sizes of  
8  
9     CRF ← 0;  
10  
11    for (k ← 0; k < K / 8; k++) {  
12  
13        GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))  
14  
15        GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))  
16  
17        syncthreads();  
18  
19        iWarp ← iBlock + warpIdx.x * 64;  
20        jWarp ← jBlock + warpIdx.y * 32;  
21  
22        iThread ← iWarp + threadIdx.x * 8;  
23        jThread ← jWarp + threadIdx.y * 8  
24  
25        for (kk ← 0; kk < 8; kk++)  
26  
27            ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))  
28  
29            ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))  
30  
31            for (i ← 0; i < 8; i ++)  
32                for (j ← 0; j < 8; j++)  
33  
34                CRF[i][j] += ARF[i][0] * BRF[0][j];  
35  
36            endfor  
37            endfor  
38  
39        endfor  
40  
41    endfor  
42  
43    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)  
44  
45 } // end kernel  
46 }
```

MatMul(M, N, K)(GL,GL,GL)(Kernel)

location in memory hierarchy      responsible level of compute hierarchy

$A^{\text{GL}}$      $B^{\text{GL}}$      $C^{\text{GL}}$



# GPU CODE IS HIERARCHICALLY STRUCTURED

```

1 __global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
2
3     _shared__ float ASH[128][8], BSH[8][128];
4     float ARF[8][1], BRF[1][8], CRF[8][8];
5
6     iBlock <- 128 * blockIdx.x;
7     jBlock <- 128 * blockIdx.y;
8
9     CRF <- 0;
10
11    for (k <- 0; k < K / 8; k++) {
12
13        GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))
14
15        GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))
16
17        syncthreads();
18
19        iWarp <- iBlock + warpIdx.x * 64;
20        jWarp <- jBlock + warpIdx.y * 32;
21
22        iThread <- iWarp + threadIdx.x * 8;
23        jThread <- jWarp + threadIdx.y * 8
24
25        for (kk <- 0; kk < 8; kk++)
26
27            ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))
28
29            ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))
30
31            for (i <- 0; i < 8; i++)
32                for (j <- 0; j < 8; j++)
33
34                    CRF[i][j] += ARF[i][0] * BRF[0][j];
35
36                endfor
37            endfor
38
39        endfor
40
41    endfor
42
43    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
44
45 } // end kernel

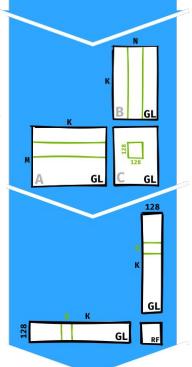
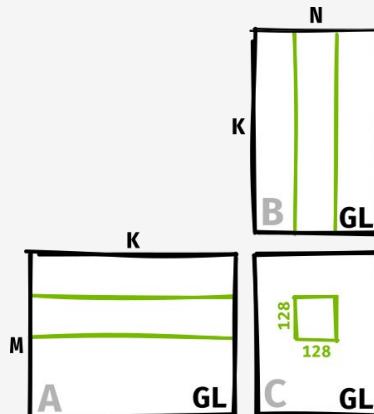
```

implements

sizes of matrices	location in memory hierarchy	responsible level of compute hierarchy
A: 128 × 8	GL	
B: 8 × 128	GL	
C: 128 × 128	GL	

**MatMul(M, N, K)(GL,GL,GL)(Kernel)**

**MatMul(128,128,K)(GL,GL,GL)(Block )**



# GPU CODE IS HIERARCHICALLY STRUCTURED

```

1 __global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
2
3     shared__ float ASH[128][8], BSH[8][128];
4     float ARF[8][1], BRF[1][8], CRF[8][8];
5
6     iBlock ← 128 * blockIdx.x;
7     jBlock ← 128 * blockIdx.y;
8
9     CRF ← 0;
10
11    for (k ← 0; k < K / 8; k++) {
12
13        GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))
14
15        GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))
16
17        syncthreads();
18
19        iWarp ← iBlock + warpIdx.x * 64;
20        jWarp ← jBlock + warpIdx.y * 32;
21
22        iThread ← iWarp + threadIdx.x * 8;
23        jThread ← jWarp + threadIdx.y * 8
24
25        for (kk ← 0; kk < 8; kk++) {
26
27            ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))
28
29            ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))
30
31            for (i ← 0; i < 8; i++)
32                for (j ← 0; j < 8; j++)
33
34                C[i][j] += ARF[i][0] * BRF[0][j];
35
36                endfor
37                endfor
38
39            endfor
40
41        endfor
42
43        PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
44
45    } // end kernel
46 }
```

implements

sizes of matrices	location in memory hierarchy	responsible level of compute hierarchy
A: 128 × 8 B: 8 × 128 C: 128 × 128	GL, GL, GL	Kernel

**MatMul(M, N, K)(GL,GL,GL)(Kernel)**

**MatMul(128,128,K)(GL,GL,GL)(Block )**  
**Init(C:128×128)( dst:RF )(Block )**

**MatMul(128,128,8)(GL,GL,RF)(Block )**

**Move( A:128×8 )(GL → SH)(Block )**

**Move( B:8×128 )(GL → SH)(Block )**

**MatMul(128,128,8)(SH,SH,RF)(Block )**

**MatMul(64, 32, 8)(SH,SH,RF)( Warp )**

**MatMul(8, 8, 8)(SH,SH,RF)(Thread)**

**MatMul(8, 8, 1)(SH,SH,RF)(Thread)**

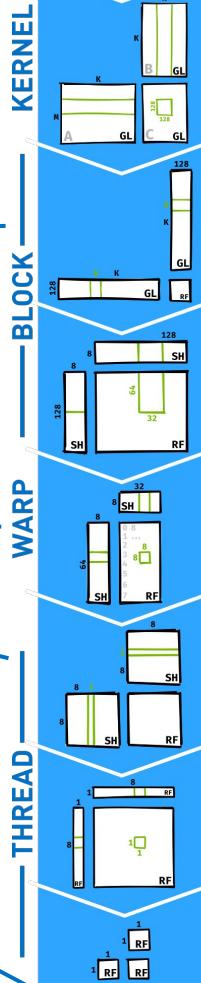
**Move( A:8×1 )(SH → RF)(Thread)**

**Move( B:1×8 )(SH → RF)(Thread)**

**MatMul(8, 8, 1)(RF,RF,RF)(Thread)**

**MatMul(1, 1, 1)(RF,RF,RF)(Thread)**

**Move(C:128×128)(RF → GL)(Block )**



# GPU CODE IS HIERARCHICALLY

```

1 __global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
2
3     shared__ float ASH[128][8], BSH[8][128];
4     float ARF[8][1], BRF[1][8], CRF[8][8];
5
6     iBlock <- 128 * blockIdx.x;
7     jBlock <- 128 * blockIdx.y;
8
9     CRF <- 0;
10
11    for (k < 0; k < K / 8; k++) {
12
13        GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))
14
15        GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))
16
17        syncthreads();
18
19        iWarp <- iBlock + warpIdx.x * 64;
20        jWarp <- jBlock + warpIdx.y * 32;
21
22        iThread <- iWarp + threadIdx.x * 8;
23        jThread <- jWarp + threadIdx.y * 8
24
25        for (kk < 0; kk < 8; kk++)
26
27            ShToPvt(ASH → ARF (8×1), start at (iThread, jThread))
28
29            ShToPvt(BSH → BRF (1×8), start at (iThread, jThread))
30
31            for (i < 0; i < 8; i++)
32                for (j < 0; j < 8; j++)
33
34                CRF[i][j] += ARF[i][0] * BRF[0][j];
35
36                endfor
37                endfor
38
39            endfor
40
41
42            PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
43
44    Move(C:128×128)(RF → GL)(Block )
45
46 } // end kernel

```

implements

sizes of  
matrices

location in  
memory hierarchy

respon-

**MatMul**(M, N, K)(GL,GL,GL)(K)
A: 128
B: 128
C: 128

**MatMul**(128,128,K)(GL,GL,GL)(B)
**Init**(C:128×128)( dst:RF )(B)
A: 128
B: 128
C: 128

**MatMul**(128,128,8)(GL,GL,RF)(B)
**Move**( A:128×8 )(GL → SH)(B)
A: 128
B: 128
C: 128

**MatMul**(128,128,8)(SH,SH,RF)(B)
**Move**( B:8×128 )(SH → RF)(B)
A: 128
B: 128
C: 128

**MatMul**(128,128,8)(SH,SH,RF)(B)
**Move**( C:128×128 )(RF → GL)(B)
A: 128
B: 128
C: 128

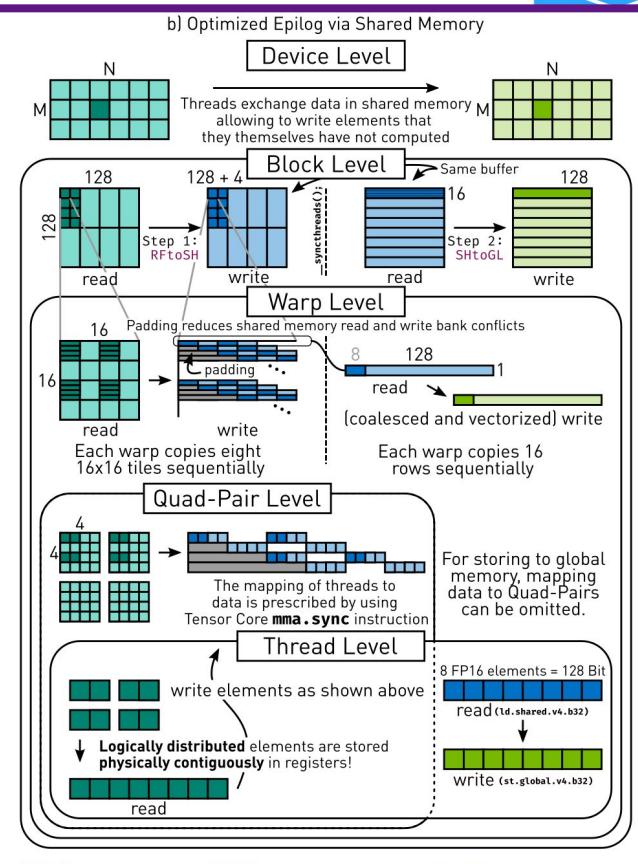
**MatMul**(64, 32, 8)(SH,SH,RF)(T)
A: 64
B: 32
C: 8

**MatMul**(8, 8, 8)(SH,SH,RF)(T)
**Move**( A:8×1 )(SH → RF)(T)
A: 8
B: 8
C: 8

**MatMul**(8, 8, 1)(SH,SH,RF)(T)
**Move**( B:1×8 )(SH → RF)(T)
A: 8
B: 1
C: 8

**MatMul**(8, 8, 1)(RF,RF,RF)(T)
A: 8
B: 8
C: 1

**MatMul**(1, 1, 1)(RF,RF,RF)(T)
A: 1
B: 1
C: 1



# GPU CODE IS HIERARCHICALLY

```

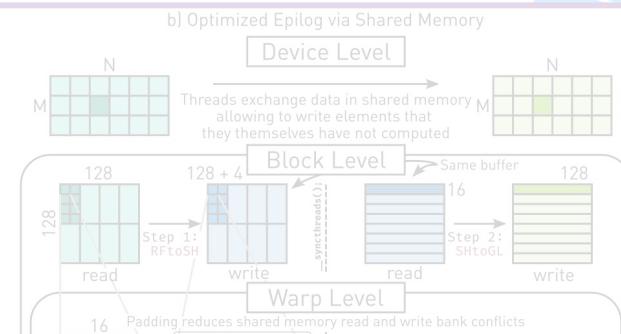
1 __global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
2
3     __shared__ float ASH[128][8], BSH[8][128];
4     float ARF[8][1], BRF[1][8], CRF[8][8];
5
6     iBlock ← 128 * blockIdx.x;
7     jBlock ← 128 * blockIdx.y;
8
9     CRF ← 0;
10
11    for (k ← 0; k < K / 8; k++) {
12
13        ...
14
15        ...
16
17        ...
18
19
20        ...
21
22
23
24
25
26
27
28
29
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46 } // end kernel

```

implements

sizes of matrices	location in memory hierarchy	compute
$A: 128 \times 128$	$B: 128 \times 128$	$C: 128 \times 128$
$M: 128 \times 128$	$GL: 128 \times 128$	$GL: 128 \times 128$

$MatMul(M, N, K)(GL,GL,GL)(K)$   
 $MatMul(128,128,K)(GL,GL,GL)(B)$   
 $Init(C:128\times128)(\text{dst:RF})(B)$   
 $MatMul(128,128,8)(GL,GL,RF)(B)$



## FIREIRON:

Programmers describe hierarchical structure of both **Computations and Data Movements** using *Specifications and Decompositions*

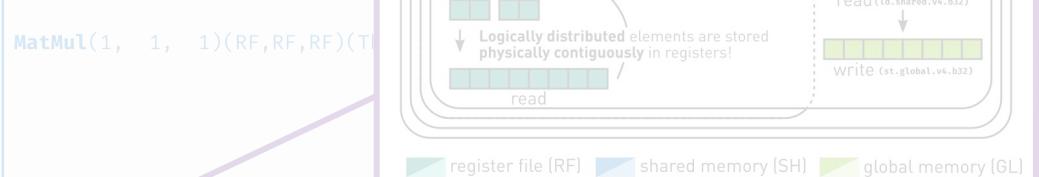
```

for (j ← 0; j < 8; j++)
    CRF[i][j] += ARF[i][0] * BRF[0][j];
endfor
endfor
endfor
PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
} // end kernel

```

$MatMul(1, 1, 1)(RF,RF,RF)(T)$

$Move(C:128\times128)(RF \rightarrow GL)(Block)$

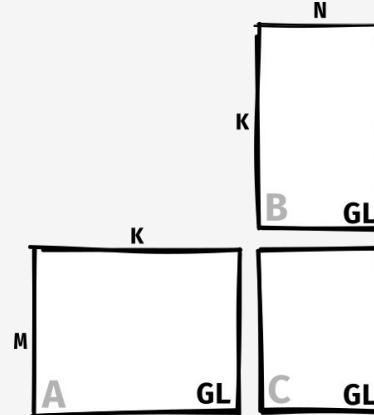


# DECOMPOSING HIGH-PERFORMANCE KERNELS

```
1 _global_ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
2
3     _shared_ float ASH[128][8], BSH[8][128];
4     float ARF[8][1], BRF[1][8], CRF[8][8];
5
6     iBlock ← 128 * blockIdx.x;
7     jBlock ← 128 * blockIdx.y;
8
9     CRF ← 0;
10
11    for (k ← 0; k < K / 8; k++) {
12
13        GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))
14
15        GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))
16
17        syncthreads();
18
19        iWarp ← iBlock + warpIdx.x * 64;
20        jWarp ← jBlock + warpIdx.y * 32;
21
22        iThread ← iWarp + threadIdx.x * 8;
23        jThread ← jWarp + threadIdx.y * 8
24
25        for (kk ← 0; kk < 8; kk++)
26
27            ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))
28
29            ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))
30
31            for (i ← 0; i < 8; i++)
32                for (j ← 0; j < 8; j++)
33
34                CRF[i][j] += ARF[i][0] * BRF[0][j];
35
36            endfor
37        endfor
38
39    endfor
40
41
42    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
43
44
45
46 } // end kernel
```

implements  
sizes of matrices      location in memory hierarchy      responsible level of compute hierarchy

MatMul(M, N, K)(GL,GL,GL)(Kernel)



## Specifications:

Data-Structure describing the task performed in a specific region of code

Example MatMul Spec:

```
MatMul(ComputeHierarchy: Kernel,
       A:Matrix((M x K),FP32,GL,ColMajor),
       B:Matrix((K x N),FP32,GL,ColMajor),
       C:Matrix((M x N),FP32,GL,ColMajor))
```

# DECOMPOSING HIGH-PERFORMANCE KERNELS

```

1 _global_ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
2
3     _shared_ float ASH[128][8], BSH[8][128];
4     float ARF[8][1], BRF[1][8], CRF[8][8];
5
6     iBlock ← 128 * blockIdx.x;
7     jBlock ← 128 * blockIdx.y;
8
9     CRF ← 0;
10
11    for (k ← 0; k < K / 8; k++) {
12
13        GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))
14
15        GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))
16
17        syncthreads();
18
19        iWarp ← iBlock + warpIdx.x * 64;
20        jWarp ← jBlock + warpIdx.y * 32;
21
22        iThread ← iWarp + threadIdx.x * 8;
23        jThread ← jWarp + threadIdx.y * 8
24
25        for (kk ← 0; kk < 8; kk++) {
26
27            ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))
28
29            ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))
30
31            for (i ← 0; i < 8; i++)
32                for (j ← 0; j < 8; j++)
33
34                    CRF[i][j] += ARF[i][0] * BRF[0][j];
35
36                endfor
37            endfor
38
39        endfor
40
41    endfor
42
43    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
44
45 } // end kernel

```

implements

sizes of matrices	location in memory hierarchy	responsible level of compute hierarchy
-------------------	------------------------------	--

**MatMul(M, N, K)(GL,GL,GL)(Kernel)**

**MatMul(128,128,K)(GL,GL,GL)(Block )**  
**Init(C:128×128)( dst:RF )(Block )**

**MatMul(128,128,8)(GL,GL,RF)(Block )**

**Move( A:128×8 )(GL → SH)(Block )**

**Move( B:8×128 )(GL → SH)(Block )**

**MatMul(128,128,8)(SH,SH,RF)(Block )**

**MatMul(64, 32, 8)(SH,SH,RF)( Warp )**

**MatMul(8, 8, 8)(SH,SH,RF)(Thread)**

**MatMul(8, 8, 1)(SH,SH,RF)(Thread)**  
**Move( A:8×1 )(SH → RF)(Thread)**

**Move( B:1×8 )(SH → RF)(Thread)**

**MatMul(8, 8, 1)(RF,RF,RF)(Thread)**

**MatMul(1, 1, 1)(RF,RF,RF)(Thread)**

**Move(C:128×128)(RF → GL)(Block )**

## Specifications:

Data-Structure describing the task performed in a specific region of code

Example **MatMul** Spec:

**MatMul**(ComputeHierarchy: **Kernel**,  
A:**Matrix**((M x K),**FP32**,**GL**,**ColMajor**),  
B:**Matrix**((K x N),**FP32**,**GL**,**ColMajor**),  
C:**Matrix**((M x N),**FP32**,**GL**,**ColMajor**))

Example **Move** Spec:

**Move**(ComputeHierarchy: **Block**,  
src:**Matrix**((128×8),**FP32**,**GL**,**ColMajor**),  
dst:**Matrix**((128×8),**FP32**,**SH**,**RowMajor**))

**MatMul(1,1,1)(RF,RF,RF)(Thread)**   
C[ ... ] = **\_hfma**(A[ ... ],B[ ... ],C[ ... ]);

# DECOMPOSING HIGH-PERFORMANCE KERNELS

```

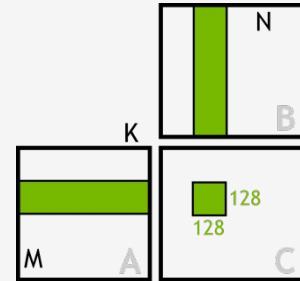
1 _global_ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
2
3     _shared_ float ASH[128][8], BSH[8][128];
4     float ARF[8][1], BRF[1][8], CRF[8][8];
5
6     iBlock ← 128 * blockIdx.x;
7     jBlock ← 128 * blockIdx.y;
8
9     CRF ← 0;
10
11    for (k ← 0; k < K / 8; k++) {
12
13        GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))
14
15        GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))
16
17        syncthreads();
18
19        iWarp ← iBlock + warpIdx.x * 64;
20        jWarp ← jBlock + warpIdx.y * 32;
21
22        iThread ← iWarp + threadIdx.x * 8;
23        jThread ← jWarp + threadIdx.y * 8
24
25        for (kk ← 0; kk < 8; kk++)
26
27            ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))
28
29            ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))
30
31            for (i ← 0; i < 8; i++)
32                for (j ← 0; j < 8; j++)
33
34                CRF[i][j] += ARF[i][0] * BRF[0][j];
35
36            endfor
37        endfor
38
39    endfor
40
41
42    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
43
44 } // end kernel
45

```

implements

sizes of matrices	location in memory hierarchy	responsible level of compute hierarchy
$A: 128 \times 8$	$B: 8 \times 128$	$C: 128 \times 128$

**MatMul(M, N, K)(GL,GL,GL)(Kernel)**  
**MatMul(128,128,K)(GL,GL,GL)(Block )**

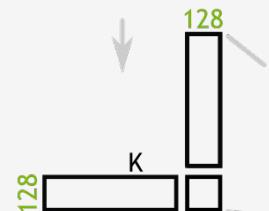


## Decompositions:

How to implement the current spec

**MatMul(Kernel,**  
**A:Matrix((MxK),FP32,GL,ColMajor),**  
**B:Matrix((KxN),FP32,GL,ColMajor),**  
**C:Matrix((MxN),FP32,GL,ColMajor))**

**.tile(128,128).to(Block)**



**MatMul(M,N,K)(GL,GL,GL)(Kernel)**  
*iBlock ← 128 \* blockIdx.x;*    *// (see Sec. 4.1)*  
*jBlock ← 128 \* blockIdx.y;*

**MatMul(Block,**  
**A:Matrix((**128**xK ),FP32,GL,ColMajor),**  
**B:Matrix((K x**128**),FP32,GL,ColMajor),**  
**C:Matrix((**128**x**128**),FP32,GL,ColMajor))**

# DECOMPOSING HIGH-PERFORMANCE KERNELS

```

1 __global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
2
3     _shared__ float ASH[128][8], BSH[8][128];
4     float ARF[8][1], BRF[1][8], CRF[8][8];
5
6     iBlock ← 128 * blockIdx.x;
7     jBlock ← 128 * blockIdx.y;
8
9     CRF ← 0;
10
11    for (k ← 0; k < K / 8; k++) {
12
13        GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))
14
15        GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))
16
17        syncthreads();
18
19        iWarp ← iBlock + warpIdx.x * 64;
20        jWarp ← jBlock + warpIdx.y * 32;
21
22        iThread ← iWarp + threadIdx.x * 8;
23        jThread ← jWarp + threadIdx.y * 8
24
25        for (kk ← 0; kk < 8; kk++)
26
27            ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))
28
29            ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))
30
31            for (i ← 0; i < 8; i++)
32                for (j ← 0; j < 8; j++)
33
34                CRF[i][j] += ARF[i][0] * BRF[0][j];
35
36            endfor
37
38        endfor
39
40    endfor
41
42
43    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
44
45 } // end kernel

```

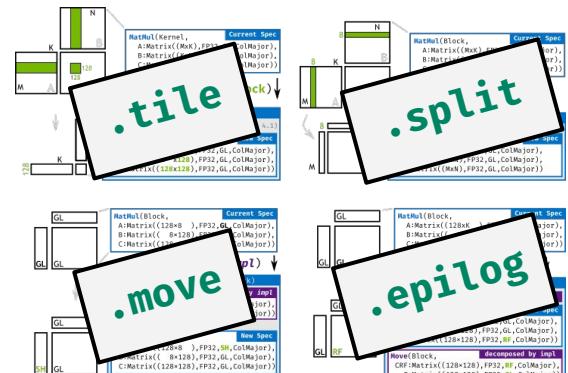
implements

sizes of matrices	location in memory hierarchy	responsible level of compute hierarchy
$A: 128 \times 128$	$B: 128 \times 8$	$C: 128 \times 128$

**MatMul(M, N, K)(GL,GL,GL)(Kernel)**  
**MatMul(128,128,K)(GL,GL,GL)(Block )**

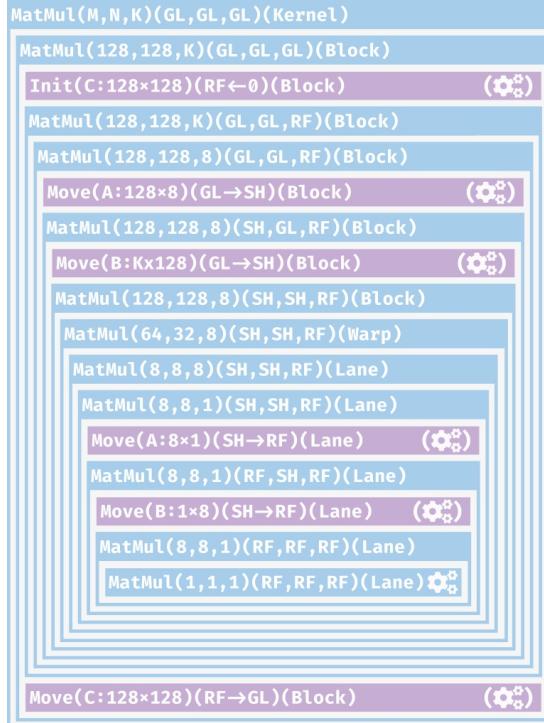
## Decompositions:

How to implement the current spec



# DECOMPOSING HIGH-PERFORMANCE KERNELS

Describing the implementation strategy in Fireiron:



Fireiron IR

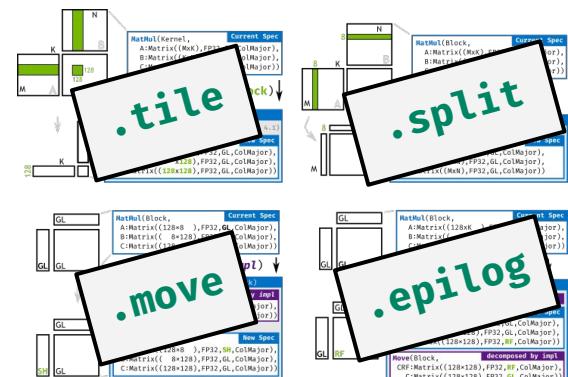
created by

Fireiron Strategy

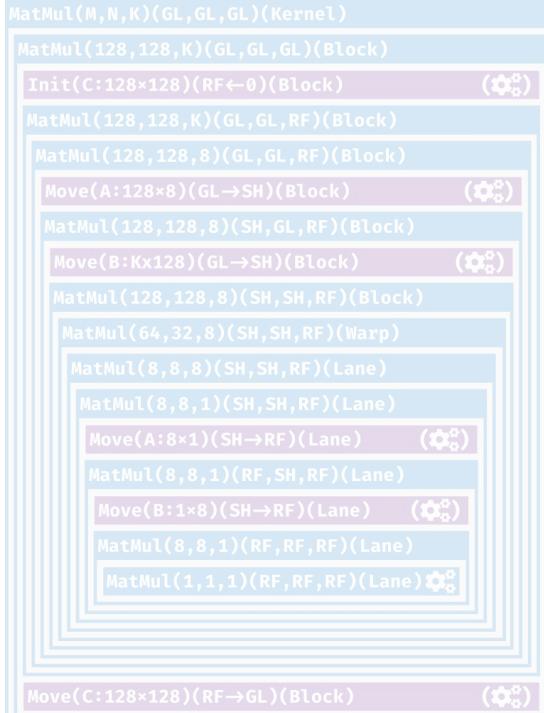
```
.tile(128, 128).to(Block)
.epilog(RF, init, store)
.apply(init)
.split(8)
.move(MatMul.A, SH, AtoSH)
.apply(AtoSH)
.move(MatMul.B, SH, BtoSH)
.apply(BtoSH)
.tile(64, 32).to(Warp)
.tile(8, 8).to(Lane)
.split(1)
.move(MatMul.A, RF, AtoRF)
.apply(AtoRF)
.move(MatMul.B, RF, BtoRF)
.apply(BtoRF)
.tile(1, 1)
.done
.apply(store)
```

## Decompositions:

How to implement the current spec



# DECOMPOSING HIGH-PERFORMANCE KERNELS



Fireiron IR

created by  
fireiron strategy

```
.tile(128, 128).to(Block)
.epilog(RF, init, store)
.apply(init)
.split(8)
.move(MatMul.A, SH, AtoSH)
.apply(AtoSH)
.move(MatMul.B, SH, BtoSH)
.tile(128, 128).to(Block)
.tile(8, 8).to(Thread)
.tile(1, 1).unroll.done,
done
.apply(store)
```

## Data Movement Optimizations

```
1 val swizz: Swizzle = id => // permutation of thread-ids
2   ((id >> 1) & 0x07) | (id & 0x30) | ((id & 0x01) << 3)
3 val storeCUDA: String = /* CUDA Epilog Micro Kernel */
4 // MATMUL -KERNEL ///////////////////////////////////////////////////////////////////
5 val maxwellOptimized = MatMul(M,N,K)(GL,GL,GL)(Kernel)
6 ///////////////////////////////////////////////////////////////////
7 .tile(128, 128).to(Block).layout(ColMajor)
8 //--- epilog: store results RF => GL -----
9 .epilog(RF, Init// accumulate in registers
10 .tile(64, 32).to(Warp)
11 .tile(8, 8).to(Thread) // alloc 64 reg per thread
12 .tile(1, 1).unroll.done,
13 Move.done(storeCUDA) /* use microkernel (18 LoC) */ )
14 .split(8).sync
15 //--- move A to SH -----
16 .move(MatMul.A, SH, Move(A:128x8)(GL→SH)(Block)
17 .tile(128, 1).to(Warp)
18 .tile(64, 1).unroll // copy in two steps
19 .tile(2, 1).to(Thread).layout(ColMajor)
20 .done).storageLayout(ColMajor).noSync
21 //--- move B to SH -----
22 .move(MatMul.B, SH, Move(B:8x128)(GL→SH)(Block)
23 .tile(8, 16).to(Warp)
24 .tile(8, 4).unroll
25 .tile(1, 1).to(Thread).layout(ColMajor)
26 .done).storageLayout(RowMajor).pad(4)
27 ///////////////////////////////////////////////////////////////////
28 .tile(64, 32).to(Warp)
29 ///////////////////////////////////////////////////////////////////
30 .tile((4, 32),(4, 16)).to(Thread)
31 .layout(ColMajor).swizzle(swizz)
32 .split(1).unroll
33 // move A and B to RF--(omit Move details for brevity)---
34 .move(MatMul.A, RF, Move.tile(4,1).unroll.done)
35 .move(MatMul.B, RF, Move.tile(1,4).unroll.done)
36 //--- perform computation using FMA -----
37 .tile(1,1).unroll.done//MatMul(1,1,1)(RF,RF,RF)(Thread)
```

# EVALUATION

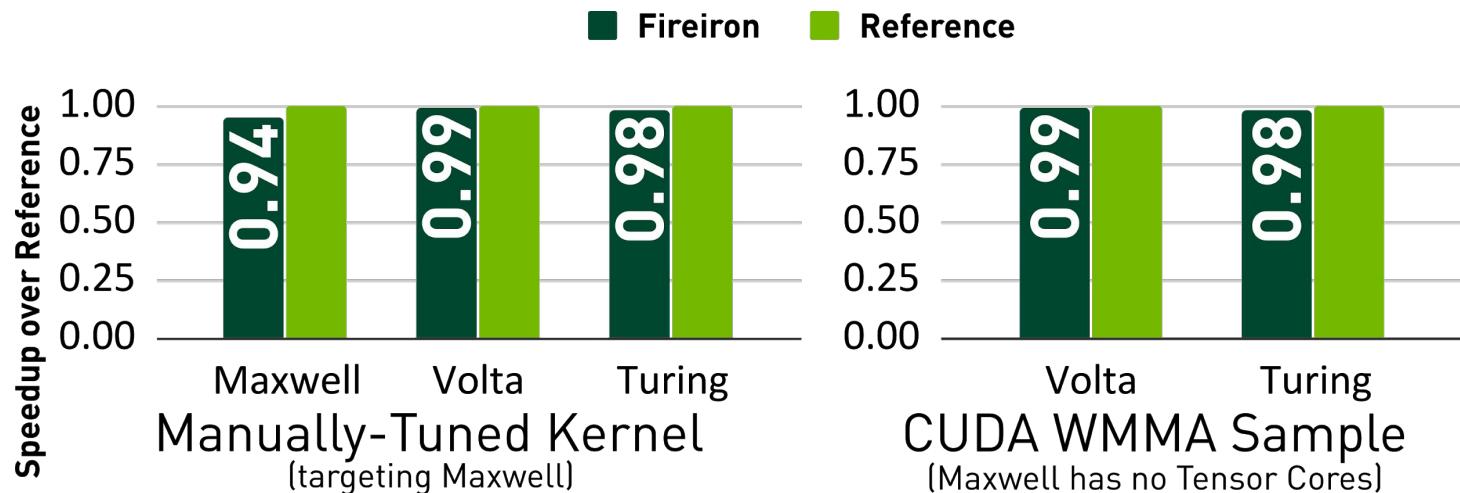
**Hypothesis A:** Code related to data movements makes up a significant fraction in high-performance kernels.

	Reference Code	Fireiron Strategy	Fireiron Generated Code
maxwell	72 (68.1%)	44 (81.8%)	94 (67.0%)
wmma	122 (41.0%)	26 (76.9%)	113 (65.4%)
cuBLAS	closed source	49 (83.7%)	260 (60.4%) (small)
		46 (84.8%)	309 (72.2%) (large)

# EVALUATION

**Hypothesis A:** Code related to data movements makes up a significant fraction in high-performance kernels.

**Hypothesis B:** Fireiron can express optimizations that are applied by experts in manually-tuned code.

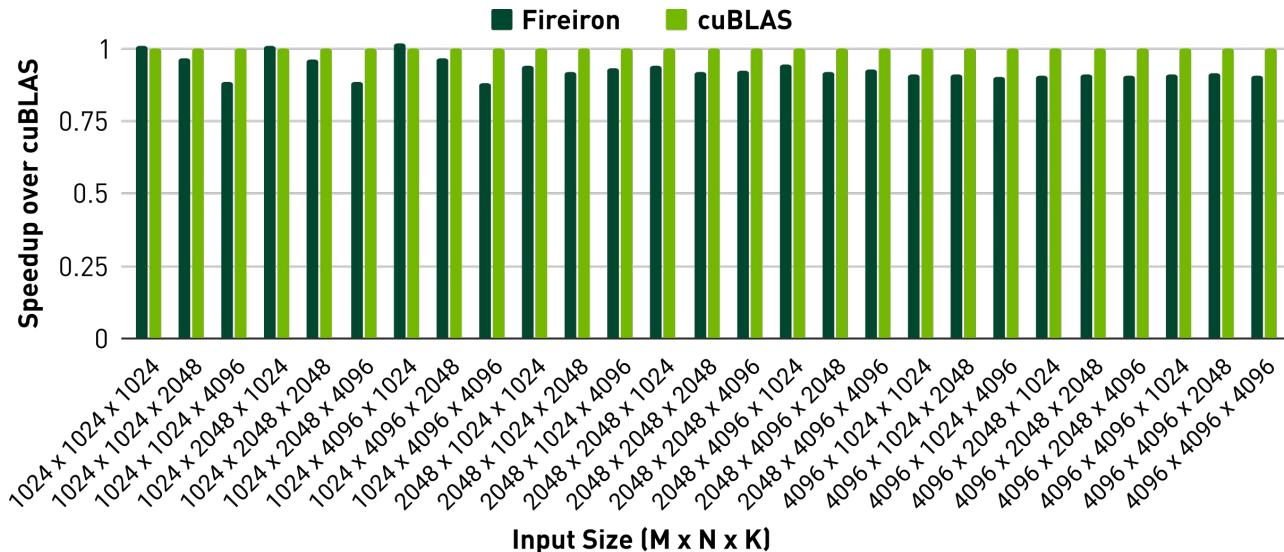


# EVALUATION

**Hypothesis A:** Code related to data movements makes up a significant fraction in high-performance kernels.

**Hypothesis B:** Fireiron can express optimizations that are applied by experts in manually-tuned code.

**Hypothesis C:** Fireiron-generated code achieves performance close to expert-tuned code



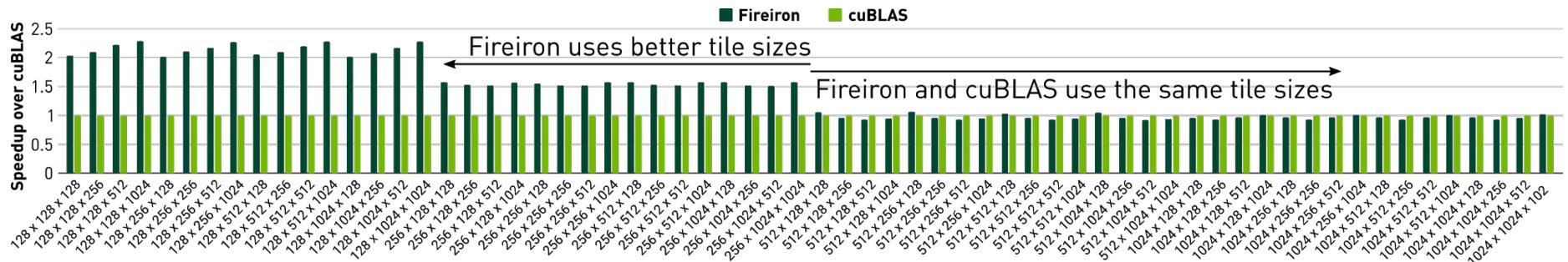
# EVALUATION

**Hypothesis A:** Code related to data movements makes up a significant fraction in high-performance kernels.

**Hypothesis B:** Fireiron can express optimizations that are applied by experts in manually-tuned code.

**Hypothesis C:** Fireiron-generated code achieves performance close to expert-tuned code

**Hypothesis D:** Experts can write Fireiron strategies that generate code which outperforms the state-of-the-art



# EVALUATION

**Hypothesis A:** Code related to data movements makes up a significant fraction in high-performance kernels.

**Hypothesis B:** Fireiron can express optimizations that are applied by experts in manually-tuned code.

**Problem:** Time-intensive: Developing Fireiron required about nine months of full-time work

**Hypothesis C:** Fireiron-generated code achieves performance close to expert-tuned code

**Hypothesis D:** Experts can write Fireiron strategies that generate code which outperforms the state-of-the-art

Bastian Hagedorn\*  
University of Münster  
b.hagedorn@wwu.de  
**9 Months**

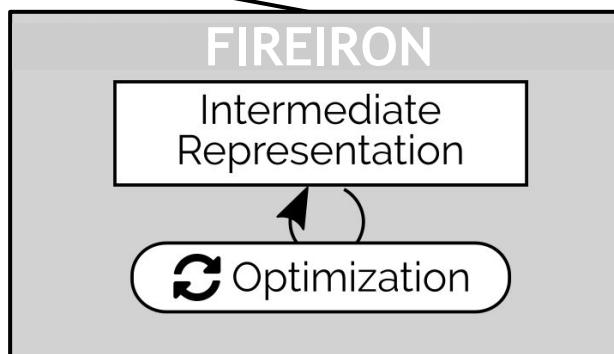
Archibald Samuel Elliott\*  
lowRISC  
sam@lenary.co.uk  
**3 Months**

Henrik Barthels\*  
AICES, RWTH Aachen University  
barthels@aices.rwth-aachen.de  
**3 Months**

Rastislav Bodik\*  
University of Washington  
bodik@cs.washington.edu  
**9 Months**

Vinod Grover  
NVIDIA  
vgrover@nvidia.com  
**9 Months**

Auto-Scheduling



v1. Codegen

Supervisors

# EVALUATION

**Hypothesis A:** Code related to data movements makes up a significant fraction in high-performance kernels.

**Hypothesis B:** Fireiron can express optimizations that are applied by experts in manually-tuned code.

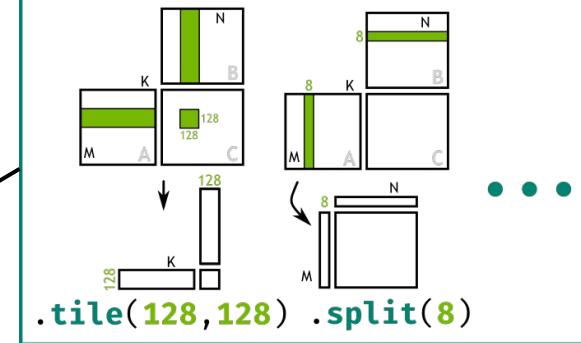
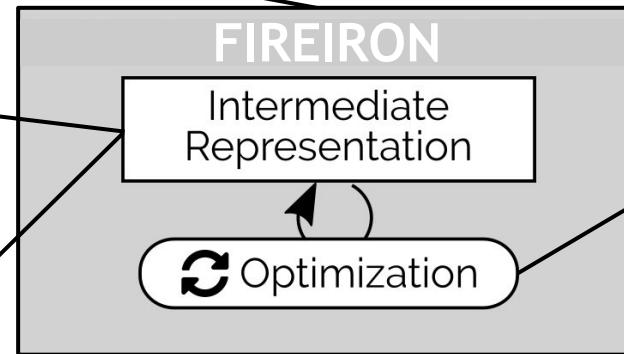
**Problem:** Time-intensive: Developing Fireiron required about nine months of full-time work

**Hypothesis C:** Fireiron-generated code achieves performance close to expert-tuned code

**Hypothesis D:** Experts can write Fireiron strategies that generate code which outperforms the state-of-the-art

```
MatMul(ComputeHierarchy: Kernel,  
       A:Matrix((M x K), FP32, GL, ColMajor),  
       B:Matrix((K x N), FP32, GL, ColMajor),  
       C:Matrix((M x N), FP32, GL, ColMajor))
```

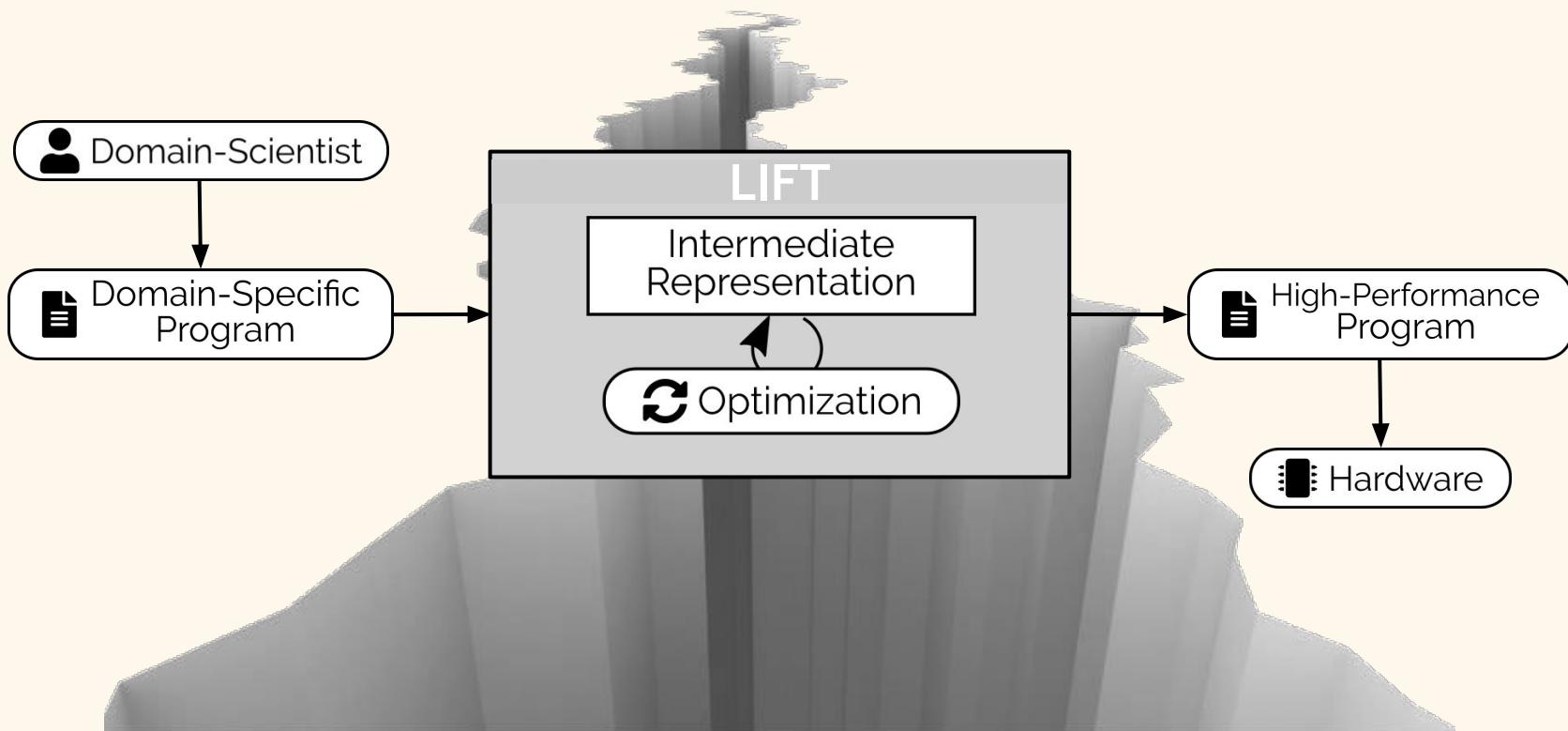
```
Move(ComputeHierarchy: Block,  
      src:Matrix((128x8), FP32, GL, ColMajor),  
      dst:Matrix((128x8), FP32, SH, RowMajor))
```



## PART II: ADDRESSING THE IR CHALLENGE

# *A GENERIC IR FOR DOMAIN-SPECIFIC COMPUTATIONS*

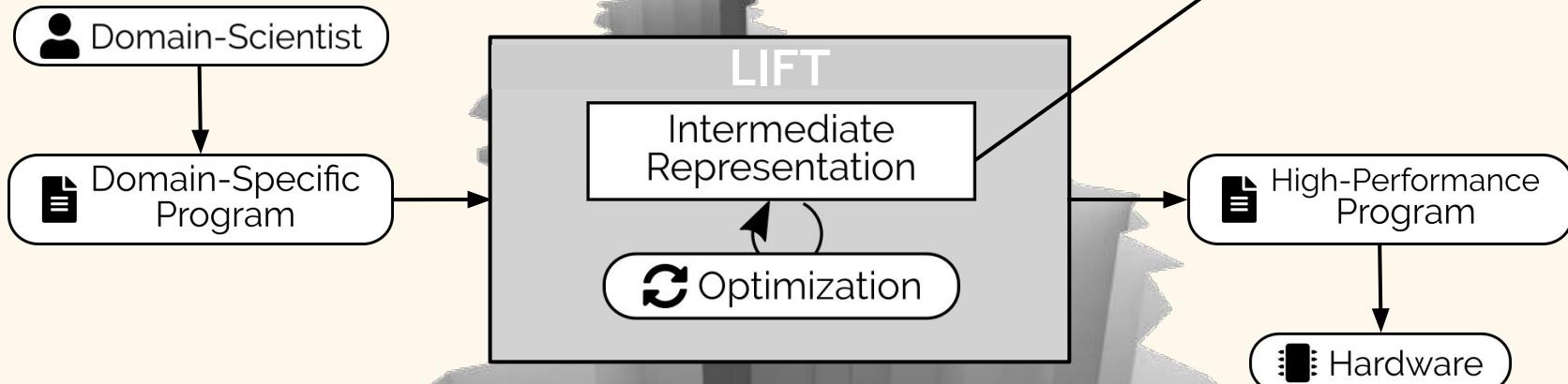
# THE LIFT APPROACH (EST. 2015)



# THE LIFT APPROACH (EST. 2015)

## ALGORITHMIC PATTERNS

*map :  $(f : T \rightarrow U, in : [T]_n) \rightarrow [U]_n$*   
*reduce :  $(init : U, f : (U, T) \rightarrow U, in : [T]_n) \rightarrow [U]_1$*   
*zip :  $(in1 : [T]_n, in2 : [U]_n) \rightarrow [(T, U)]_n$*   
*iterate :  $(in : [T]_n, f : [T]_n \rightarrow [T]_n, m : \text{Int}) \rightarrow [T]_n$*   
*split :  $(m : \text{Int}, in : [T]_n) \rightarrow [[T]]_m|_{n/m}$*   
*join :  $(in : [[T]]_m)_n \rightarrow [T]_{m \times n}$*   
*at :  $(i : \text{Cst}, in : [T]_n) \rightarrow T$*   
*get :  $(i : \text{Cst}, in : \{T_1, T_2, \dots\}) \rightarrow T_i$*   
*array :  $(n : \text{Int}, f : (i : \text{Int}, n : \text{Int}) \rightarrow T) \rightarrow [T]_n$*   
*userFun :  $(s1 : \text{ScalarT}, s2 : \text{ScalarT}', \dots) \rightarrow \text{ScalarU}$*



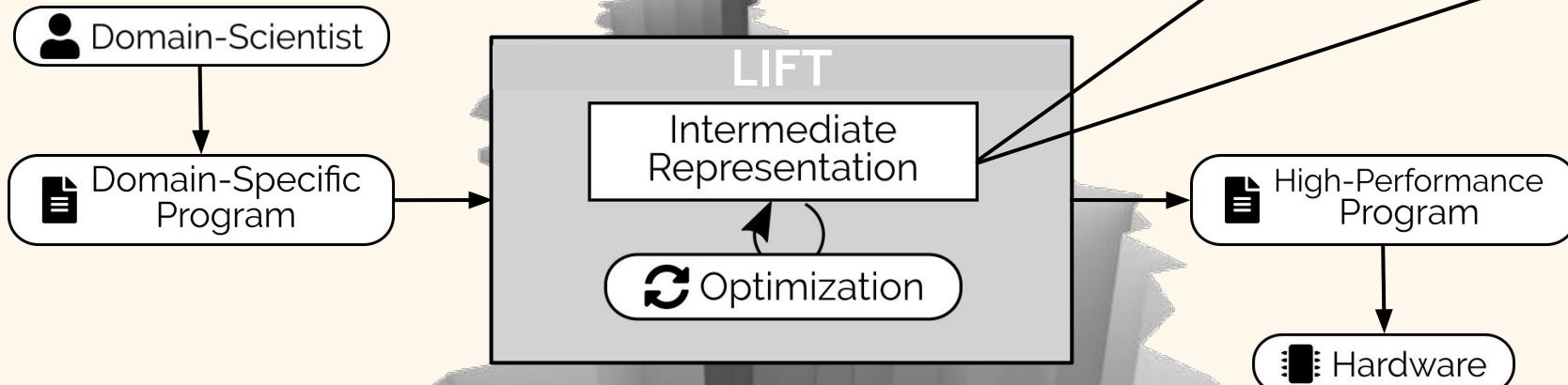
# THE LIFT APPROACH (EST. 2015)

## ALGORITHMIC PATTERNS

*map :  $(f : T \rightarrow U, in : [T]_n) \rightarrow [U]_n$*   
*reduce :  $(init : U, f : (U, T) \rightarrow U, in : [T]_n) \rightarrow [U]_1$*   
*zip :  $(in1 : [T]_n, in2 : [U]_n) \rightarrow [[T, U]]_n$*   
*iterate :  $(in : [T]_n, f : [T]_n \rightarrow [T]_n, m : \text{Int}) \rightarrow [T]_n$*   
*split :  $(m : \text{Int}, in : [T]_n) \rightarrow [[T]]_m \text{ } n/m$*   
*join :  $(in : [[T]]_m)_n \rightarrow [T]_{m \times n}$*   
*at :  $(i : \text{Cst}, in : [T]_n) \rightarrow T_i$*   
*get :  $(i : \text{Cst}, in : \{T_1, T_2, \dots\}) \rightarrow T_i$*   
*array :  $(n : \text{Int}, f : (i : \text{Int}, n : \text{Int}) \rightarrow T) \rightarrow [T]_n$*   
*userFun :  $(s1 : \text{ScalarT}, s2 : \text{ScalarT}', \dots) \rightarrow \text{ScalarU}$*

## OPENCL-PATTERNS

<i>mapWorkgroup</i>	<i>toGlobal</i>
<i>mapLocal</i>	<i>toLocal</i>
<i>mapSeq</i>	<i>toPrivate</i>
<i>vectorize</i>	<i>gather</i>
<i>toVector</i>	<i>scatter</i>
<i>toScalar</i>	



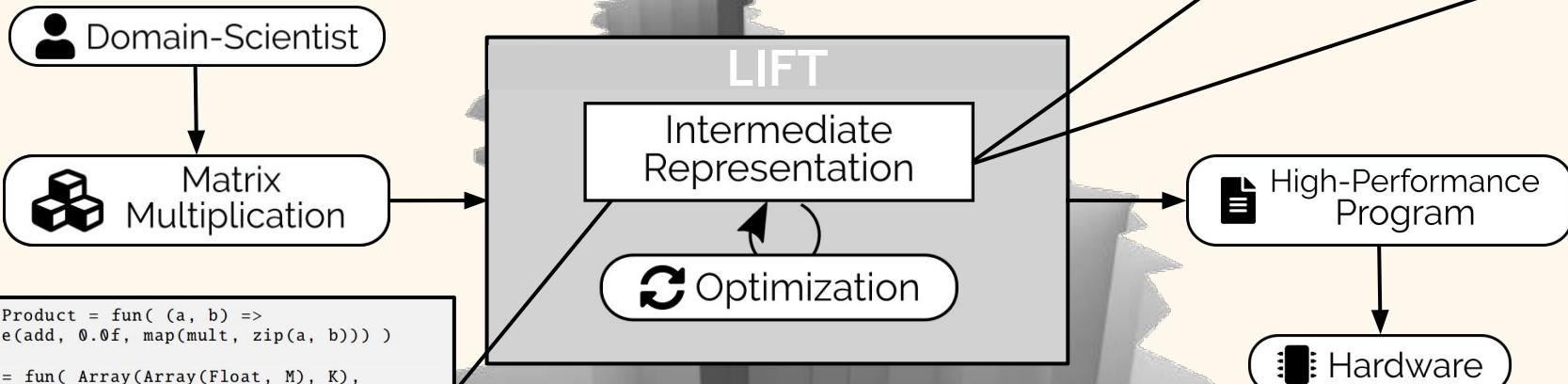
# THE LIFT APPROACH (EST. 2015)

## ALGORITHMIC PATTERNS

```
map : (f : T → U, in : [T]n) → [U]n
reduce : (init : U, f : (U, T) → U, in : [T]n) → [U]1
zip : (in1 : [T]n, in2 : [U]n) → [{T, U}]n
iterate : (in : [T]n, f : [T]n → [T]n, m : Int) → [T]n
split : (m : Int, in : [T]n) → [[T]m]n/m
join : (in : [[T]m]n) → [T]m×n
at : (i : Cst, in : [T]n) → T
get : (i : Cst, in : {T1, T2, ...}) → Ti
array : (n : Int, f : (i : Int, n : Int) → T) → [T]n
userFun : (s1 : ScalarT, s2 : ScalarT', ...) → ScalarU
```

## OPENCL-PATTERNS

mapWorkgroup	toGlobal
mapLocal	toLocal
mapSeq	toPrivate
vectorize	gather
toVector	scatter
toScalar	



```
val dotProduct = fun( (a, b) =>
    reduce(add, 0.0f, map(mult, zip(a, b))) )

val mm = fun( Array(Array(Float, M), K),
              Array(Array(Float, K), N),
              (A, B) =>
      map(fun(aRow =>
              map(fun(bCol =>
                      dotProduct(aRow, bCol)), transpose(B))), A))
```

## MATRIX MULTIPLICATION

# THE LIFT APPROACH (EST. 2015)

## REWRITE RULES

$map(f) \circ map(g) \rightarrow map(f \circ g)$

$map(f) \rightarrow join \circ map(map(f)) \circ split(n)$



```
val dotProduct = fun( (a, b) =>
    reduce(add, 0.0f, map(mult, zip(a, b))) )

val mm = fun( Array(Array(Float, M), K),
             Array(Array(Float, K), N),
             (A, B) =>
               map(fun(aRow =>
                   map(fun(bCol =>
                       dotProduct(aRow, bCol)), transpose(B))), A))
```

## MATRIX MULTIPLICATION

## ALGORITHMIC PATTERNS

$map : (f : T \rightarrow U, in : [T]_n) \rightarrow [U]_n$   
 $reduce : (init : U, f : (U, T) \rightarrow U, in : [T]_n) \rightarrow [U]_1$   
 $zip : (in1 : [T]_n, in2 : [U]_n) \rightarrow [(T, U)]_n$   
 $iterate : (in : [T]_n, f : [T]_n \rightarrow [T]_n, m : Int) \rightarrow [T]_n$   
 $split : (m : Int, in : [T]_n) \rightarrow [[T]]_m/n$   
 $join : (in : [[T]]_m/n) \rightarrow [T]_{m \times n}$   
 $at : (i : Cst, in : [T]_n) \rightarrow T_i$   
 $get : (i : Cst, in : \{T_1, T_2, \dots\}) \rightarrow T_i$   
 $array : (n : Int, f : (i : Int, n : Int) \rightarrow T) \rightarrow [T]_n$   
 $userFun : (s1 : ScalarT, s2 : ScalarT', \dots) \rightarrow ScalarU$

## OPENCL-PATTERNS

$mapWorkgroup$	$toGlobal$
$mapLocal$	$toLocal$
$mapSeq$	$toPrivate$
$vectorize$	$gather$
$toVector$	$scatter$
$toScalar$	

## LIFT

Intermediate Representation

Optimization



High-Performance Program



Hardware

# THE LIFT APPROACH (EST. 2015)

## REWRITE RULES

$map(f) \circ map(g) \rightarrow map(f \circ g)$

$map(f) \rightarrow join \circ map(map(f)) \circ split(n)$

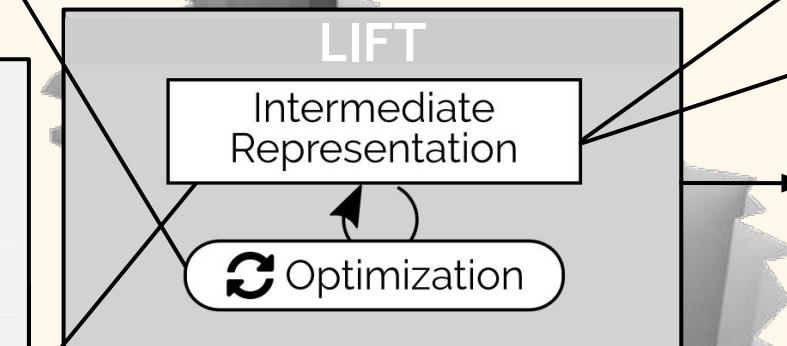


Domain-Scientist

```
until o map(λ rowOfTilesA .
  map(λ colOfTilesB .
    toGlobal(copy2D) o
    reduce(λ (tileAcc, (tileA, tileB)) .
      map(map(+)) o zip(tileAcc) o
      map(λ aBlocks .
        map(λ bs .
          reduce(+, 0) o
          map(λ (aBlock, b) .
            map(λ (a,bp) . a × bp
              , zip(aBlock, toPrivate(id(b)))))
          ) o zip(transpose(aBlocks), bs)
          , toLocal(copy2D(tileB)))
          , split(l, toLocal(copy2D(tileA))))
          ,0, zip(rowOfTilesA, colOfTilesB)
        ) o tile(m, k, transpose(B))
      ) o tile(n, k, A)
```

## ALGORITHMIC PATTERN

```
map : (f : T → U, in : [T]n) → [U]n
reduce : (init : U, f : (U, T) → U, in : [T]n) → U
zip : (in1 : [T]n, in2 : [U]n) → [(T, U)]n
iterate : (in : [T]n, f : [T]n → [T]n, m : M) → M
split : (m : Int, in : [T]n) → [[T]]mn
join : (in : [[T]]mn) → [T]m×n
at : (i : Cst, in : [T]n) → Ti
get : (i : Cst, in : {T1, T2, ...}) → Ti
array : (n : Int, f : (i : Int, n : Int) → Ti) → [T]n
userFun : (s1 : ScalarT, s2 : ScalarT', ...)
```



## MATRIX MULTIPLICATION

```
kernel mm_amd_opt(global float * A, B, C,
                   int K, M, N) {
  local float tileA[512]; tileB[512];
  private float acc_0; ...; acc_31;
  private float blockOfB_0; ...; blockOfB_3;
  private float blockOfA_0; ...; blockOfA_7;

  int lid0 = local_id(0); lid1 = local_id(1);
  int wid0 = group_id(0); wid1 = group_id(1);

  for (int w1=wid1; w1<M/64; w1+=num_grps(1)) {
    for (int w0=wid0; w0<N/64; w0+=num_grps(0)) {

      acc_0 = 0.0f; ...; acc_31 = 0.0f;
      for (int i=0; i<K/8; i++) {
        vstore4(vload4(lid1*M/4+2*i*M+16*w1+lid0,A),
                ,16*lid1+lid0, tileA);
        vstore4(vload4(lid1*N/4+2*i*N+16*w0+lid0,B),
                ,16*lid1+lid0, tileB);
        barrier(...);

      for (int j = 0; j<8; j++) {
        blockOfA_0 = tileA[0+64*j+lid1*8];
        ... 6 more statements
        blockOfA_7 = tileA[7+64*j+lid1*8];
        blockOfB_0 = tileB[0 + 64*j+lid0];
        ... 2 more statements
        blockOfB_3 = tileB[48+64*j+lid0];

        acc_0 += blockOfA_0 * blockOfB_0;
        acc_1 += blockOfA_0 * blockOfB_1;
        acc_2 += blockOfA_0 * blockOfB_2;
        acc_3 += blockOfA_0 * blockOfB_3;
        ... 24 more statements
        acc_28 += blockOfA_7 * blockOfB_0;
        acc_29 += blockOfA_7 * blockOfB_1;
        acc_30 += blockOfA_7 * blockOfB_2;
        acc_31 += blockOfA_7 * blockOfB_3;
      }
      barrier(...);

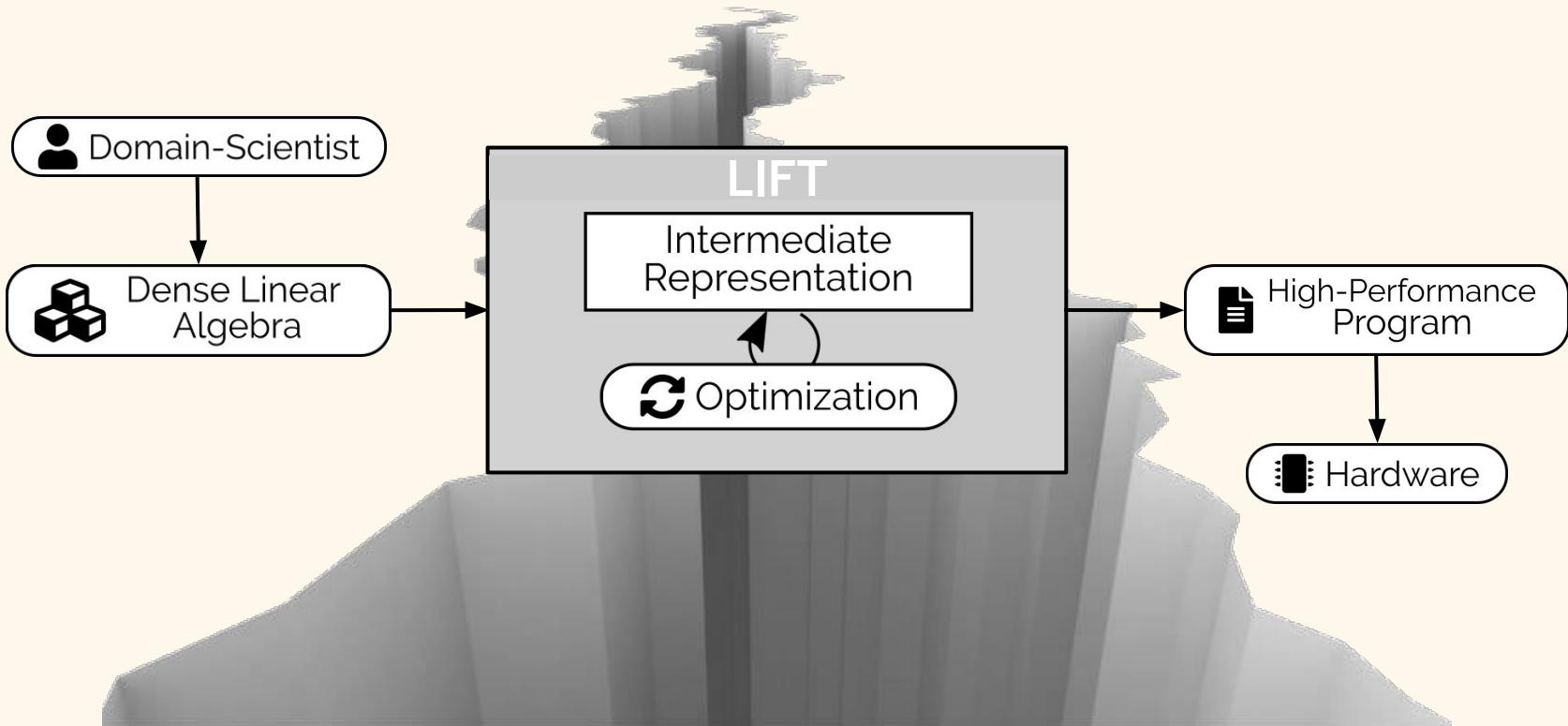
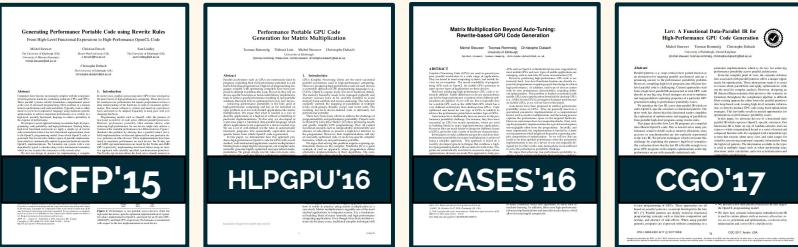
    }

    C[ 0+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_0;
    C[16+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_1;
    C[32+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_2;
    C[48+8*lid1*N+64*w0+64*w1*N+0*N+lid0]=acc_3;
    ... 24 more statements
    C[ 0+8*lid1*N+64*w0+64*w1*N+7*N-lid0]=acc_28;
    C[16+8*lid1*N+64*w0+64*w1*N+7*N-lid0]=acc_29;
    C[32+8*lid1*N+64*w0+64*w1*N+7*N-lid0]=acc_30;
    C[48+8*lid1*N+64*w0+64*w1*N+7*N-lid0]=acc_31;
  } } }
```



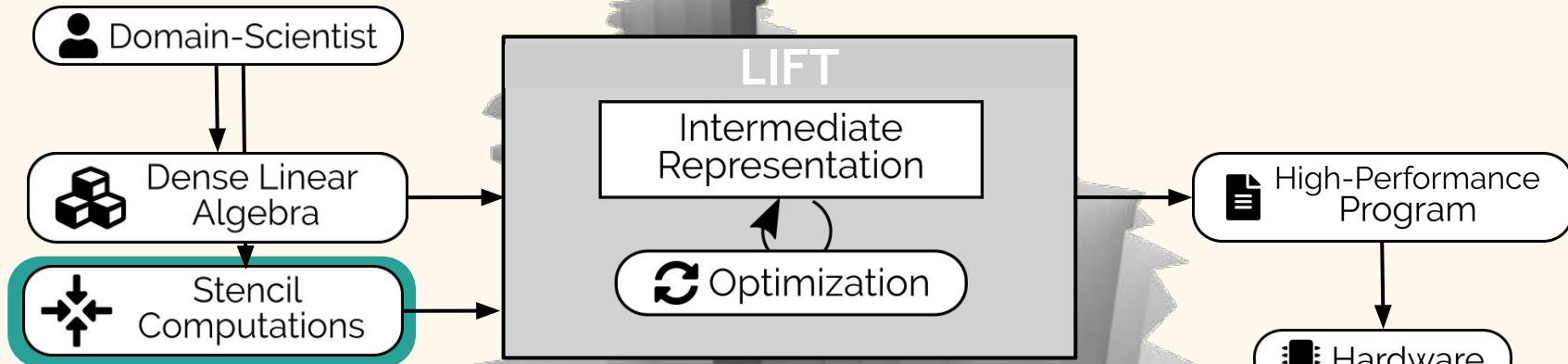
Hardware

# THE LIFT APPROACH WORKS WELL FOR DENSE LINEAR ALGEBRA:



THIS WORK: DEMONSTRATING THAT THE  
**LIFT IR IS EASILY EXTENSIBLE, REUSABLE ACROSS DOMAINS AND**  
**PROVIDES MULTIPLE LEVELS OF ABSTRACTION**

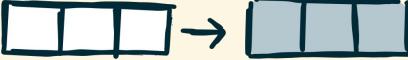
(ADDRESSING THE IR CHALLENGE)



BY ADDING SUPPORT FOR **STENCIL COMPUTATIONS**

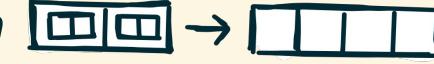
# STENCIL COMPUTATIONS IN LIFT?

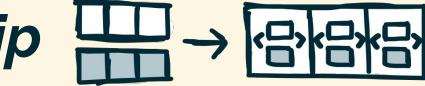
*Existing Patterns:*

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

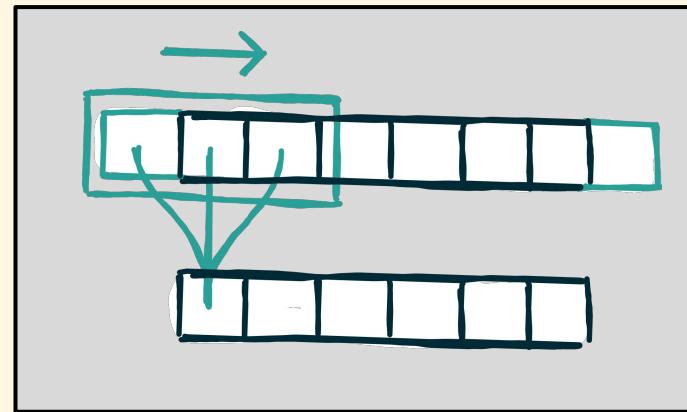
*reduce(  $\oplus$  )* 

*split( $n$ )* 

*join* 

*zip* 

**1D STENCIL COMPUTATION**



HOW TO EXPRESS THIS IN LIFT?

# STENCIL COMPUTATIONS IN LIFT? NO PROBLEM ...

Existing Patterns:

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

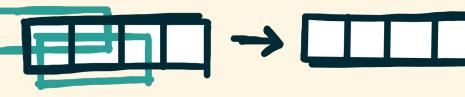
*reduce*(  $\oplus$  ) 

*split(n)* 

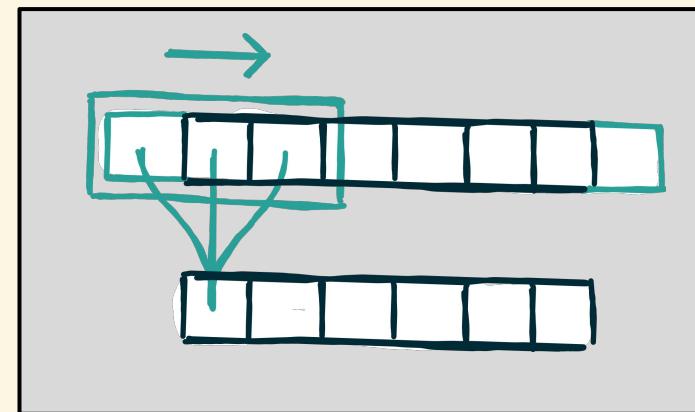
*join* 

*zip* 

New Pattern?

*stencil* 

## 1D STENCIL COMPUTATION

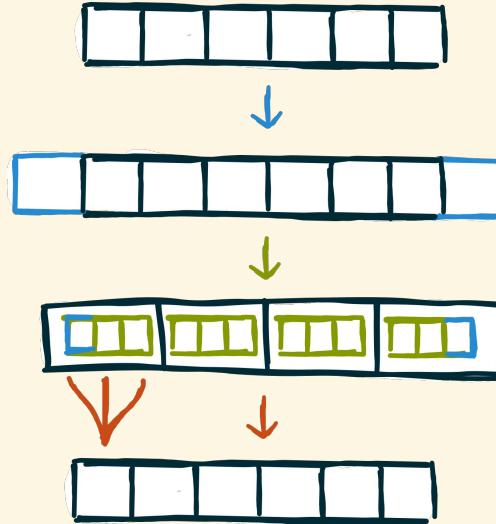


- ∅ **DOMAIN-SPECIFIC**  
*rather than generic*
- ∅ **NO REUSE**  
*of existing patterns and rewrites*
- ∅ **MULTIDIMENSIONAL?**  
*is it composable?*

# DECOMPOSING STENCIL COMPUTATIONS

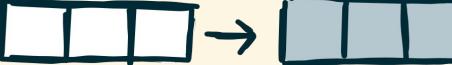
## 3-point-stencil.c

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



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

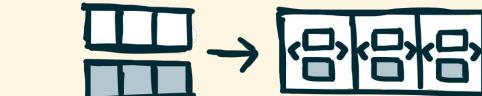
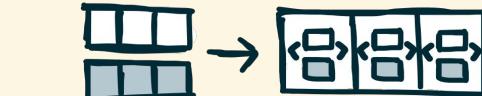
# EXPRESSING STENCIL COMPUTATIONS

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

*reduce*( $\oplus$ )   $\rightarrow$  

*split(n)*   $\rightarrow$  

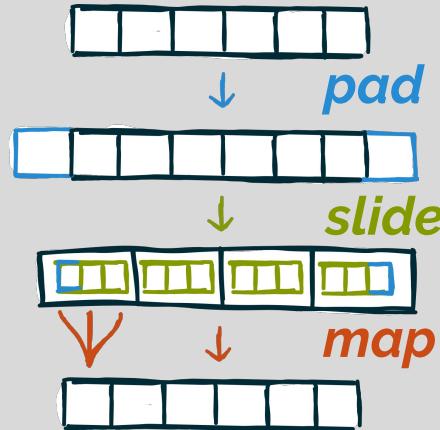
*join*   $\rightarrow$  

*zip*   $\rightarrow$  

*pad(l,r,b)*   $\rightarrow$  

*slide(n,s)*   $\rightarrow$  

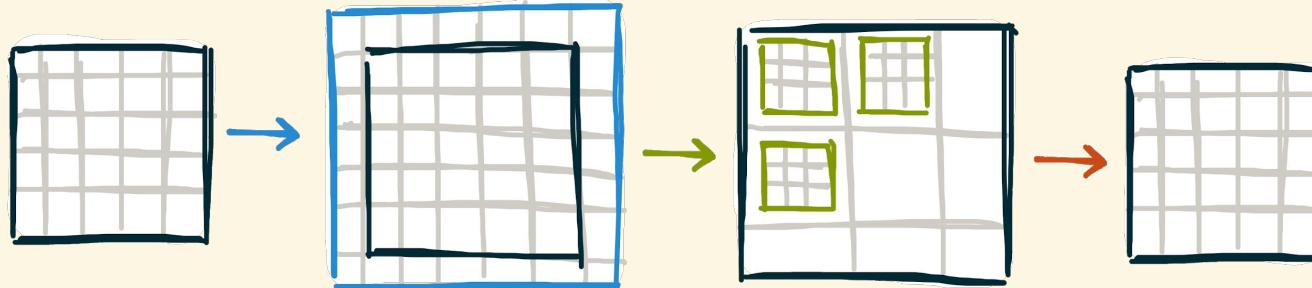
## stencil1D.lift



```
def stencil1D = fun(xs =>
  map(reduce(add, 0),
    slide(3,1,
      pad(1,1,clamp,xs))))
```

# MULTIDIMENSIONAL STENCIL COMPUTATIONS

DECOMPOSE TO RE-COMPOSE



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

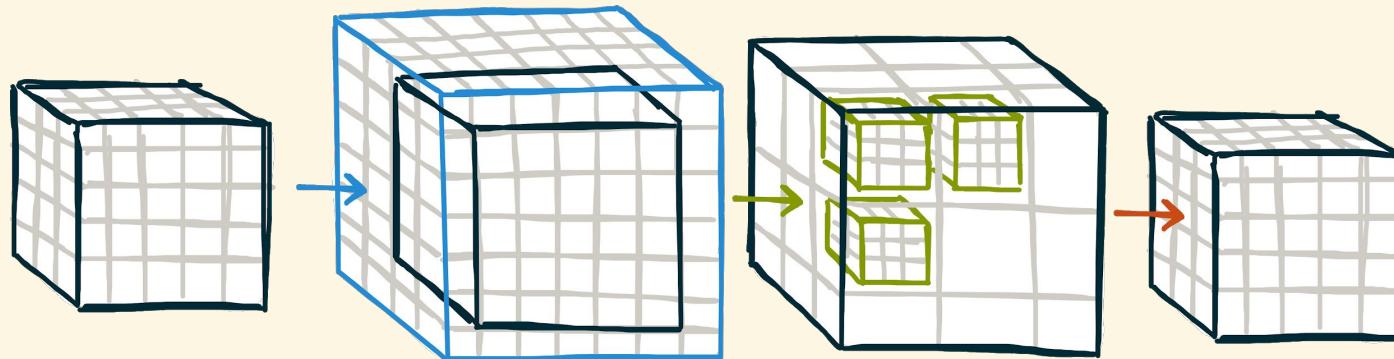
$\text{pad}_2 = \text{map}(\text{pad}(1,1,\text{clamp},\text{pad}(1,1,\text{clamp},\text{input})))$



MULTIDIMENSIONAL DOMAIN-SPECIFIC ABSTRACTIONS AS  
COMPOSITIONS OF ONE-DIMENSIONAL GENERIC PATTERNS

# MULTIDIMENSIONAL STENCIL COMPUTATIONS

DECOMPOSE TO RE-COMPOSE



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

$\text{pad}_3 = \text{map}(\text{map}(\text{pad}(1,1,\text{clamp}(\text{map}(\text{pad}(1,1,\text{clamp},\text{pad}(1,1,\text{clamp},\text{input}))))))))$



MULTIDIMENSIONAL DOMAIN-SPECIFIC ABSTRACTIONS AS  
COMPOSITIONS OF ONE-DIMENSIONAL GENERIC PATTERNS

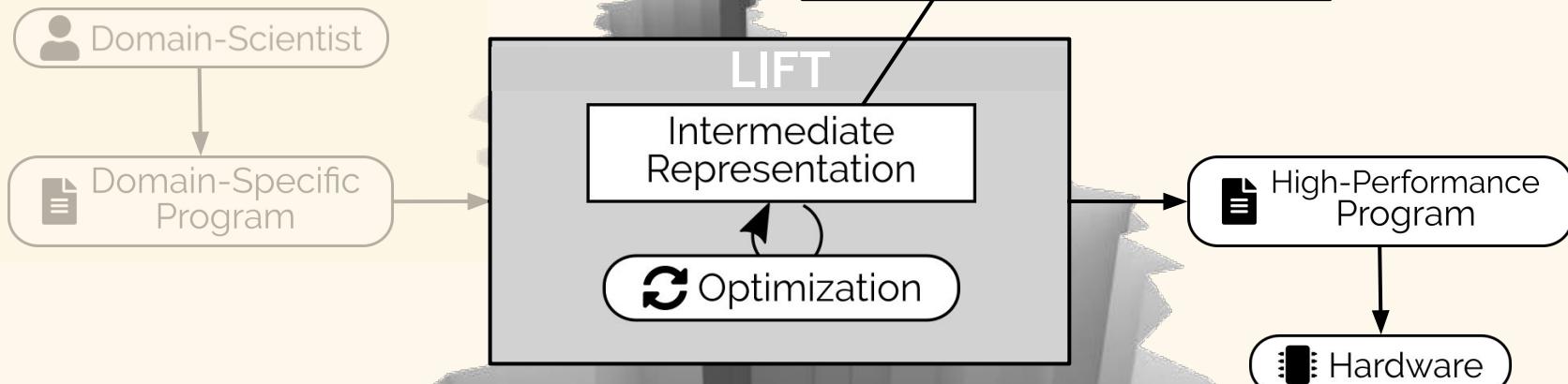
# SUPPORTING STENCIL COMPUTATIONS

We added:

 2 Patterns  
*pad, slide*

## ALGORITHMIC PATTERNS

*map :  $(f : T \rightarrow U, in : [T]_n) \rightarrow [U]_n$*   
*reduce :  $(init : U, f : (U, T) \rightarrow U, in : [T]_n) \rightarrow [U]_1$*   
*zip :  $(in1 : [T]_n, in2 : [U]_n) \rightarrow [[T, U]]_n$*   
*iterate :  $(in : [T]_n, f : [T]_n \rightarrow [T]_n, m : \text{Int}) \rightarrow [T]_n$*   
*split :  $(m : \text{Int}, in : [T]_n) \rightarrow [[T]]_m|_{n/m}$*   
*join :  $(in : [[T]]_m)_n \rightarrow [T]_{m \times n}$*   
*at :  $(i : \text{Cst}, in : [T]_n) \rightarrow T_i$*   
*get :  $(i : \text{Cst}, in : \{T_1, T_2, \dots\}) \rightarrow T_i$*   
*array :  $(n : \text{Int}, f : (i : \text{Int}, n : \text{Int}) \rightarrow T) \rightarrow [T]_n$*   
*userFun :  $(s1 : \text{ScalarT}, s2 : \text{ScalarT}', \dots) \rightarrow \text{ScalarU}$*



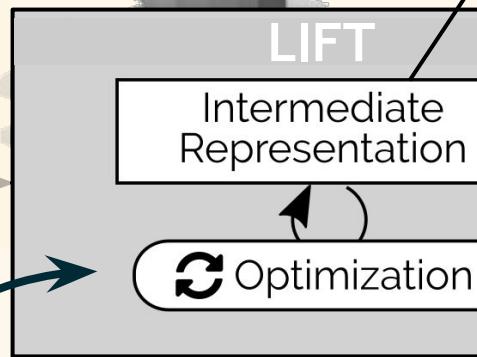
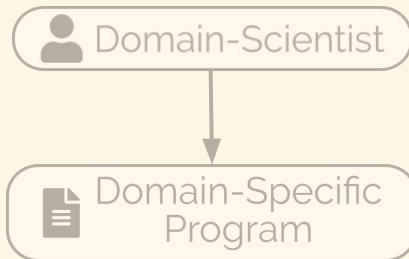
# SUPPORTING STENCIL COMPUTATIONS

We added:

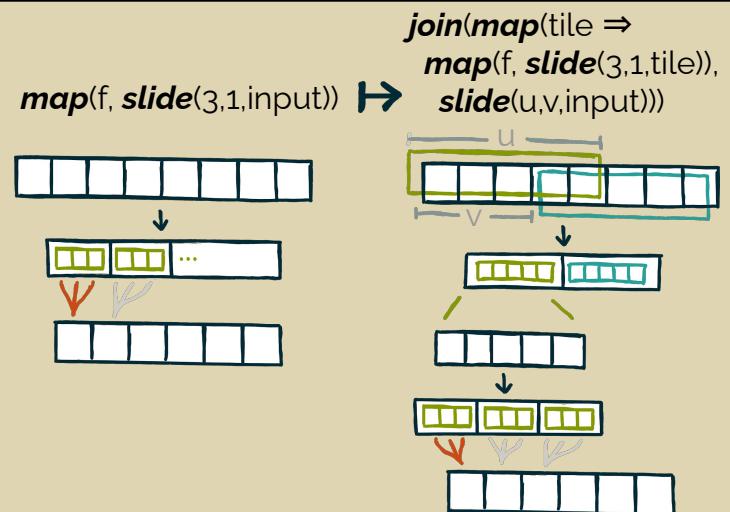
2 Patterns  
*pad, slide*

## ALGORITHMIC PATTERNS

*map : (f : T → U, in : [T]<sub>n</sub>) → [U]<sub>n</sub>*  
*reduce : (init : U, f : (U, T) → U, in : [T]<sub>n</sub>) → [U]<sub>1</sub>*  
*zip : (in1 : [T]<sub>n</sub>, in2 : [U]<sub>n</sub>) → [[T, U]]<sub>n</sub>*  
*iterate : (in : [T]<sub>n</sub>, f : [T]<sub>n</sub> → [T]<sub>n</sub>, m : Int) → [T]<sub>n</sub>*  
*split : (m : Int, in : [T]<sub>n</sub>) → [[T]]<sub>m</sub><sub>n/m</sub>*  
*join : (in : [[T]]<sub>m</sub><sub>n</sub>) → [T]<sub>m×n</sub>*  
*at : (i : Cst, in : [T]<sub>n</sub>) → T<sub>i</sub>*  
*get : (i : Cst, in : {T<sub>1</sub>, T<sub>2</sub>, ...}) → T<sub>i</sub>*  
*array : (n : Int, f : (i : Int, n : Int) → T) → [T]<sub>n</sub>*  
*userFun : (s1 : ScalarT, s2 : ScalarT', ...) → ScalarU*

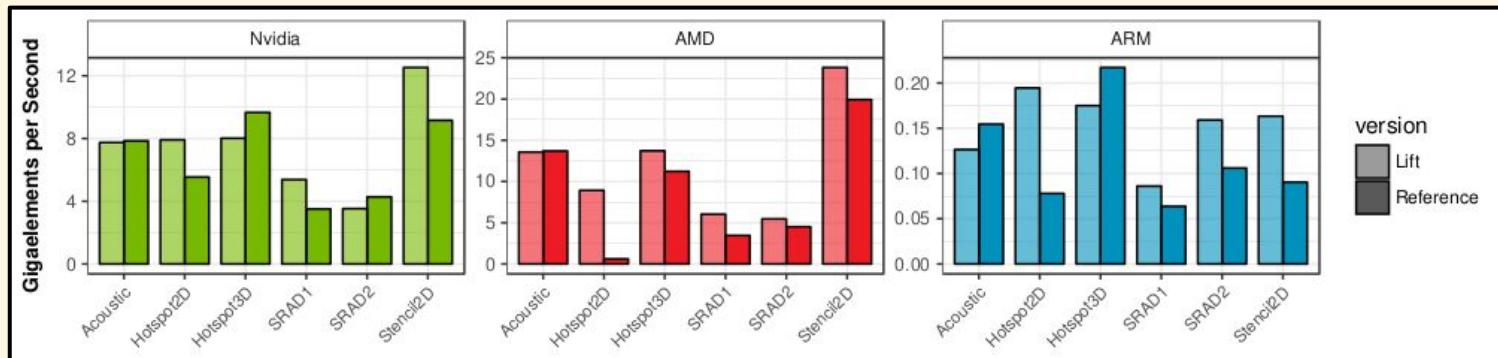


1 Rewrite Rule  
*overlapped tiling*



# COMPARISON WITH HAND-OPTIMIZED CODES

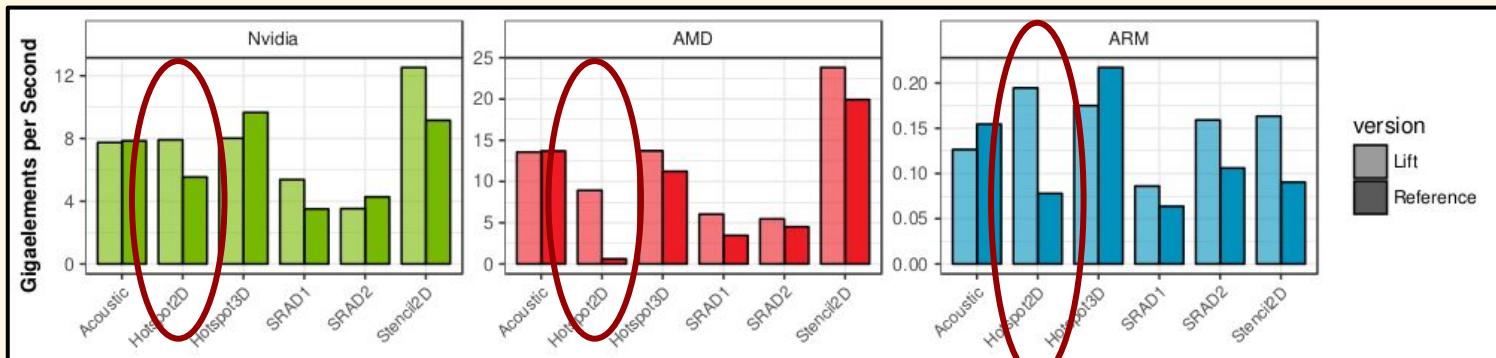
HIGHER IS BETTER



LIFT ACHIEVES PERFORMANCE COMPETITIVE TO HAND OPTIMIZED CODE

# COMPARISON WITH HAND-OPTIMIZED CODES

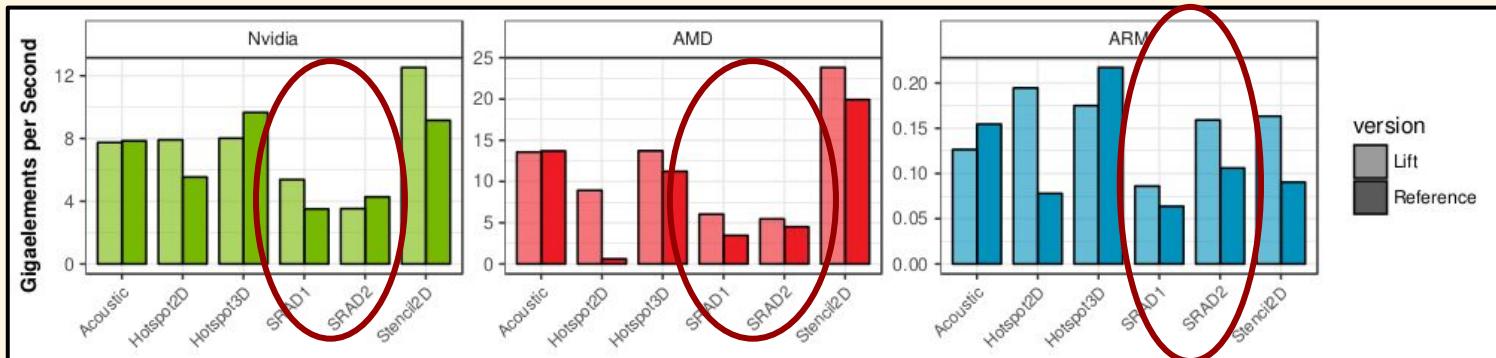
HIGHER IS BETTER



LIFT ACHIEVES PERFORMANCE COMPETITIVE TO HAND OPTIMIZED CODE

# COMPARISON WITH HAND-OPTIMIZED CODES

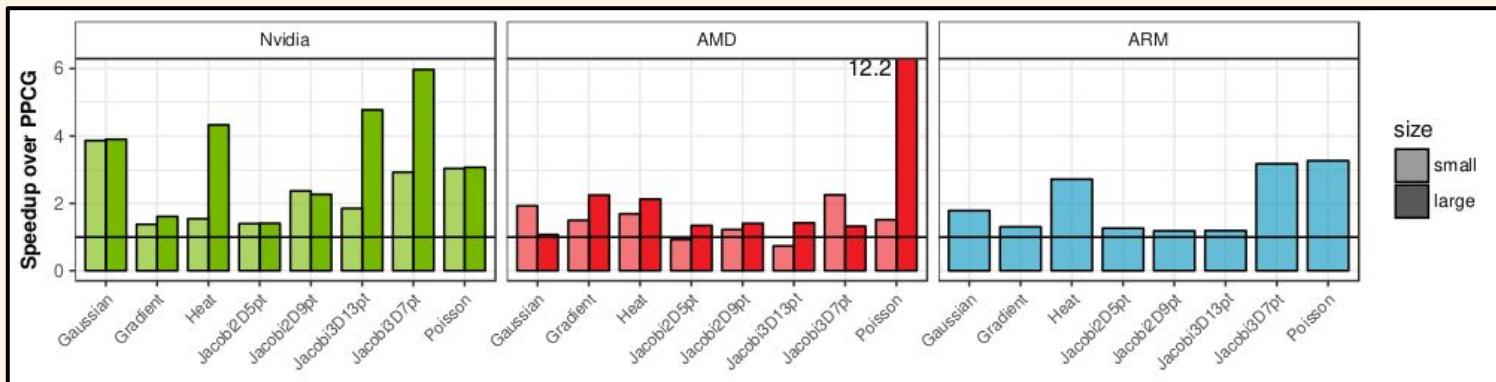
HIGHER IS BETTER



LIFT ACHIEVES PERFORMANCE COMPETITIVE TO HAND OPTIMIZED CODE

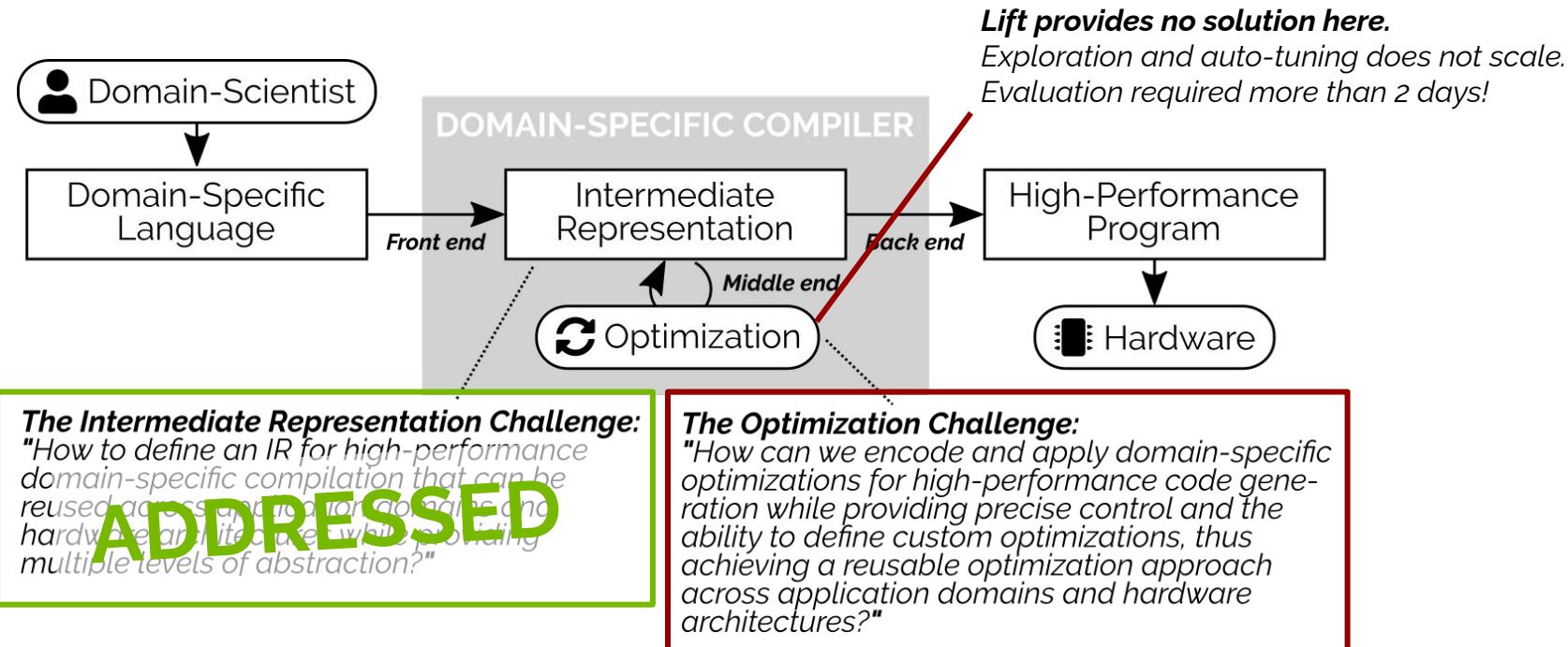
# COMPARISON WITH POLYHEDRAL COMPILATION

HIGHER IS BETTER



LIFT OUTPERFORMS STATE-OF-THE-ART OPTIMIZING COMPILERS

# HIGH PERFORMANCE DOMAIN-SPECIFIC COMPIRATION WITH DOMAIN-SPECIFIC COMPILERS



# ELEVATE

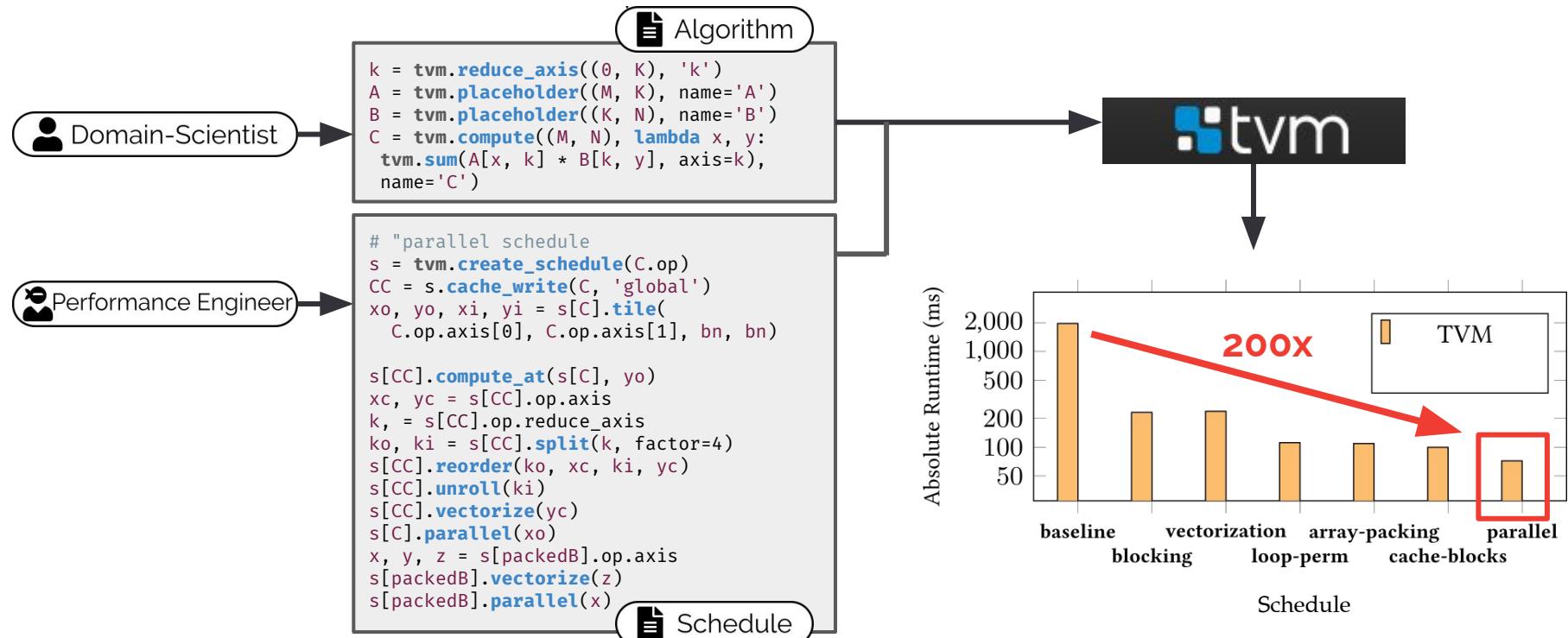
---

*A Language for Describing Optimization Strategies*

**PART III:** ADDRESSING THE OPTIMIZATION CHALLENGE

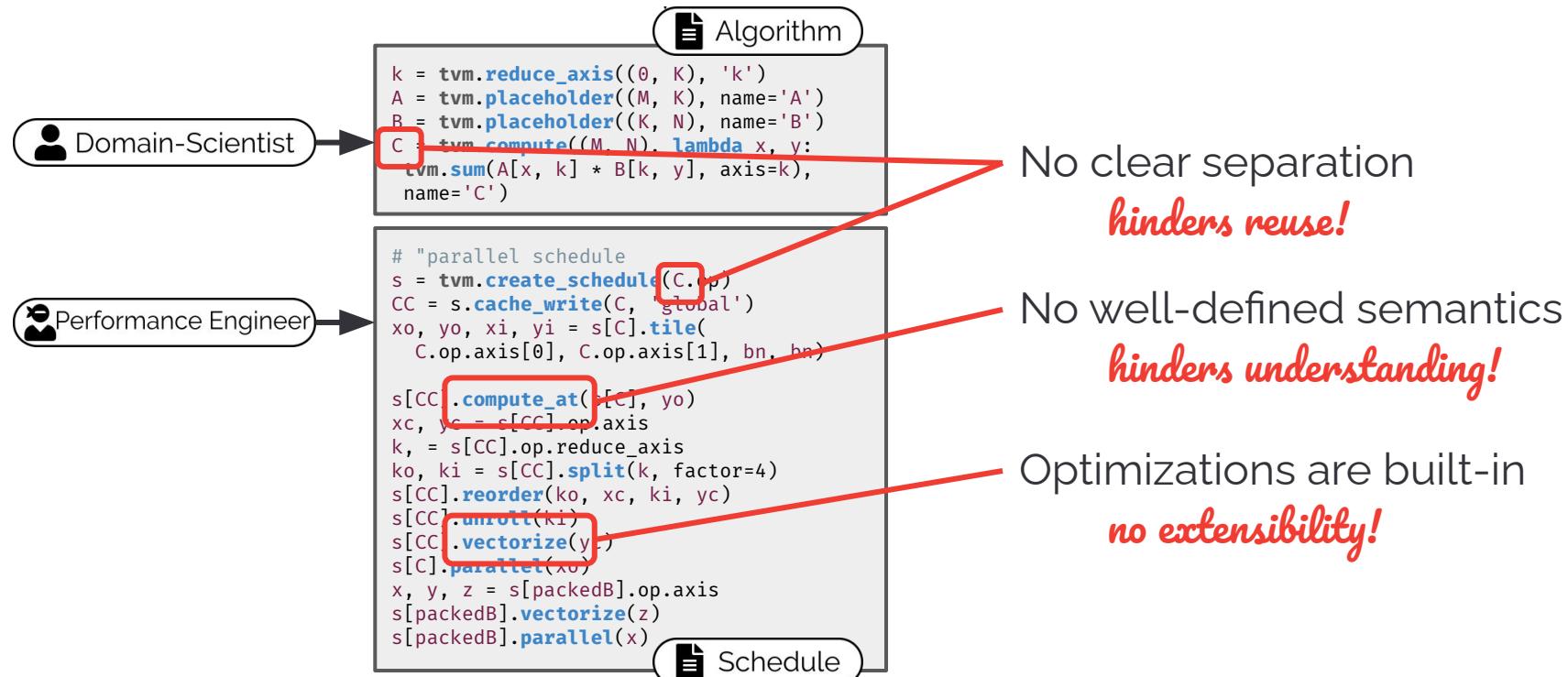
# SCHEDULE-BASED COMPIRATION

*Decoupling Computations and Optimizations*



# SCHEDULE-BASED COMPIRATION

*Decoupling Computations and Optimizations*



# OUR GOALS

---

*A Principled Way to Separate, Describe, and Apply Optimizations*

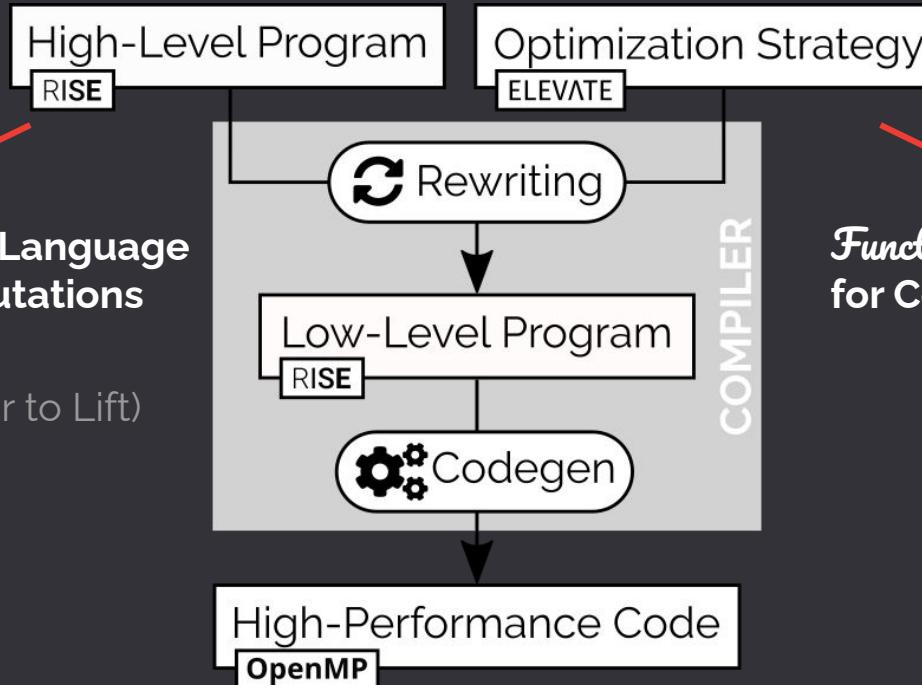
- 1 **Separate concerns:** Computations should not be changed for expressing optimizations
- 2 **Facilitate reuse:** Clear separation between computations and optimizations
- 3 **Enable composability:** Allow user-defined abstractions composed of simple building blocks
- 4 **Allow reasoning:** Well-defined semantics for all provided building blocks
- 5 **Be explicit:** Avoid all implicit behaviour during compilation

# The *Functional* Way

to high-performance domain-specific compilation

*Functional Data-parallel Language  
for Expressing Computations*

(the spiritual successor to Lift)



*Functional Strategy Language  
for Composing Rewrite Rules*

# ELEVATE

---

## *A Language for Describing Optimization Strategies*

A **Strategy** encodes a program transformation:

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

A **RewriteResult** encodes its success or failure:

```
RewriteResult[P] = Success[P](p: P)  
| Failure[P](s: Strategy[P])
```

# ELEVATE

## *A Language for Describing Optimization Strategies*

A **Strategy** encodes a program transformation:

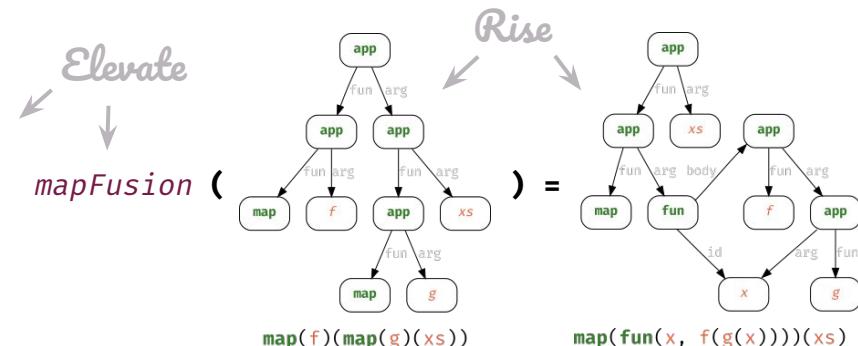
```
type Strategy[P] = P => RewriteResult[P]
```

A **RewriteResult** encodes its success or failure:

```
RewriteResult[P] = Success[P](p: P)  
| Failure[P](s: Strategy[P])
```

**Rewrite Rules** are examples for basic strategies:  $\text{map}(f) \circ \text{map}(g) = \text{map}(f \circ g)$

```
def mapFusion: Strategy[Rise] =  
(p:Rise) => p match {  
  case app(app(map, f),  
           app(app(map, g), xs)) =>  
    Success( map(fun(x => f(g(x))))(xs) )  
  case _ => Failure( mapFusion )  
}
```



# COMBINATORS

## *How to Build More Powerful Strategies*

Sequential Composition ( ; )

```
def seq[P]: Strategy[P] => Strategy[P] => Strategy[P] =
  fs => ss => p => fs(p) >>= ss
```

Left Choice (<+)

```
def lChoice[P]: Strategy[P] => Strategy[P] => Strategy[P] =
  fs => ss => p => fs(p) <|> ss(p)
```

Try

```
def try[P]: Strategy[P] => Strategy[P] =
  s => p => (s <+ id)(p)
```

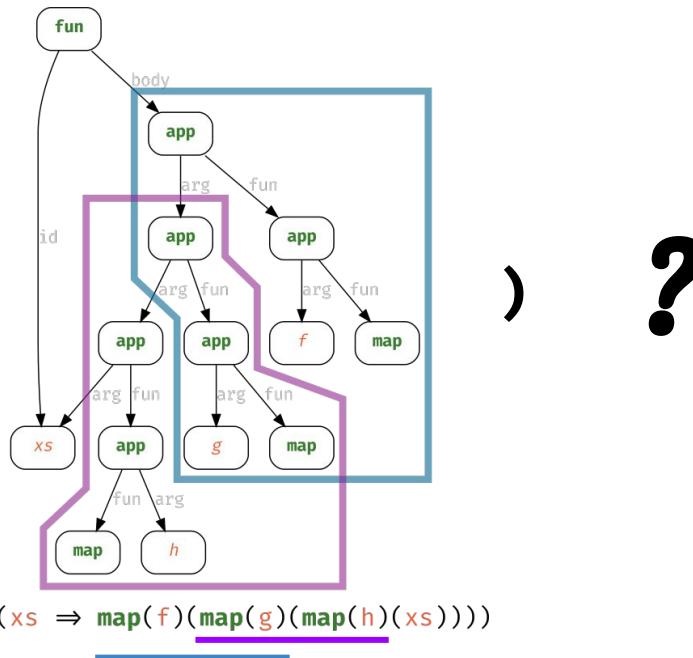
Repeat

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

# TRAVERSALS

*Describing Precise Locations*

*mapFusion* (



There are **two possible locations** for successfully applying the rule

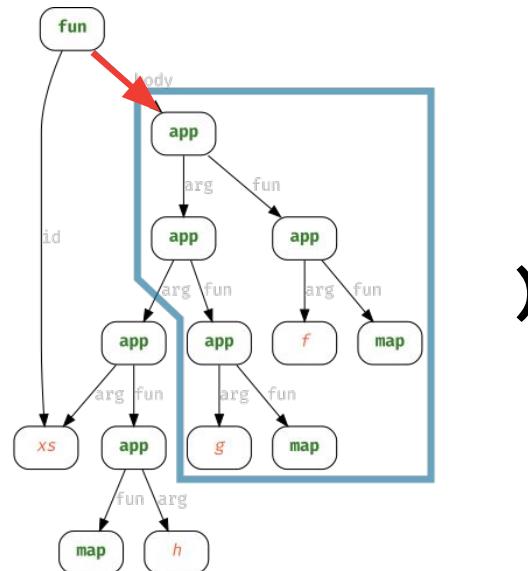
# TRAVERSALS

## *Describing Precise Locations*

```
def body: Traversal[Rise] = s => p => p match {  
  case fun(x,b) => (nb => fun(x,nb) <$> s(b))  
  case _ => Failure( body(s) )  
}
```

*apply s at body of function abstraction*

*body(mapFusion)* (



*threemaps* = fun(xs, map(f)(map(g)(map(h)(xs))))

There are ***two possible locations*** for successfully applying the rule

# TRAVERSALS

*Describing Precise Locations*

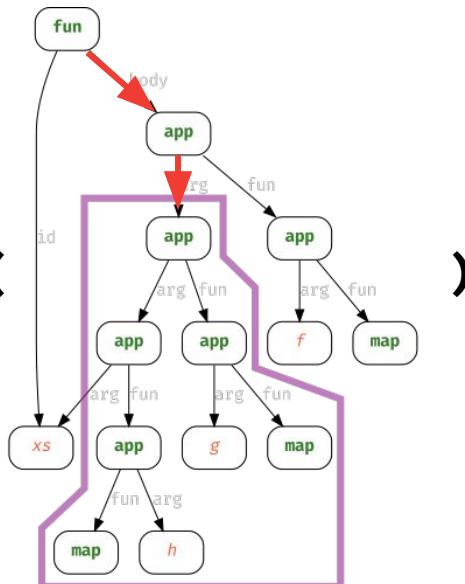
```
def body: Traversal[Rise] = s => p => p match {  
  case fun(x,b) => (nb => fun(x,nb) <$> s(b))  
  case _ => Failure( body(s) )  
}
```

*body(argument(mapFusion)) (*

```
def argument: Traversal[Rise] = s => p => p match {  
  case app(f,a) => (na => app(f,na) <$> s(a))  
  case _ => Failure( argument(s) )  
}
```

*apply s at argument of function application*

*threemaps = fun(xs, map(f)(map(g)(map(h)(xs))))*



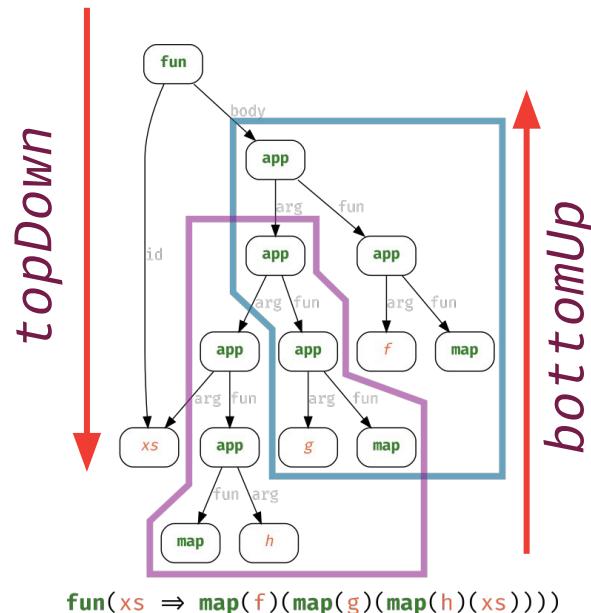
There are ***two possible locations*** for successfully applying the rule

# NORMALIZATION

## *More Complex Traversals*

Generic Tree Traversals...

```
def topDown: Traversal[Rise] = s => p => (s <+ one(topDown(s)))(p)
def bottomUp: Traversal[Rise] = s => p => (one(bottomUp(s)) <+ s)(p)
...
```



# NORMALIZATION

## *More Complex Traversals*

Generic Tree Traversals...

```
def topDown: Traversal[Rise] = s => p => (s <+ one(topDown(s)))(p)
def bottomUp: Traversal[Rise] = s => p => (one(bottomUp(s)) <+ s)(p)
...
```

... and a strategy for normalization

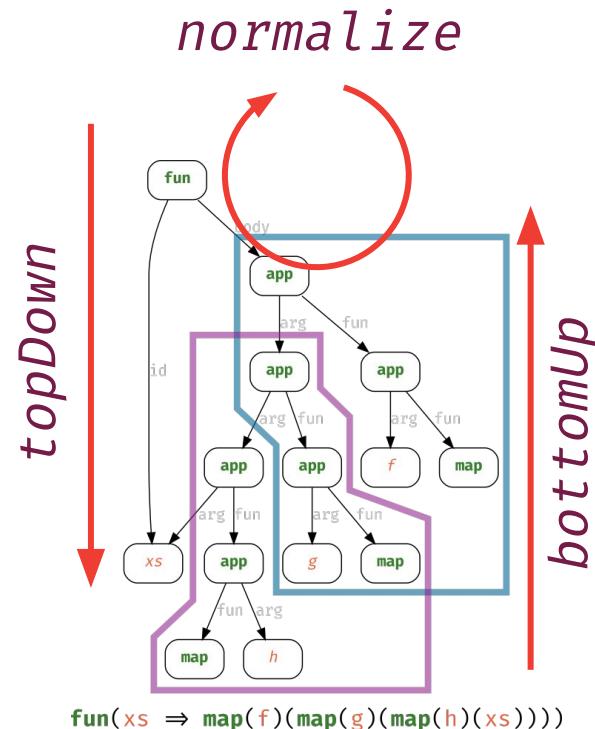
```
def normalize: Traversal[Rise] = s => p => repeat(topDown(s))(p)
```

With these, we define normal-forms like  $\beta\eta$ -normal-form

```
def BENF = normalize(betaReduction <+ etaReduction)
```

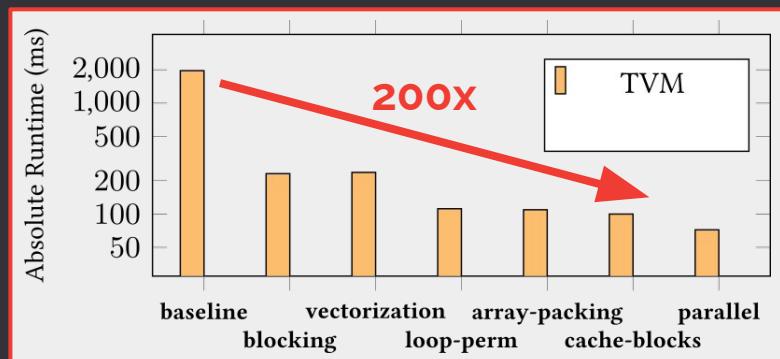
$$(\lambda x. t)s \rightarrow t[x := s]$$

$\eta$ -reduction converts between  $\lambda x. fx$  and  $f$  whenever  $x$  does not appear free in  $f$ .



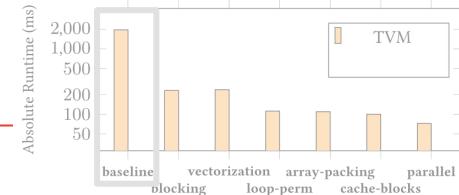
# CASE STUDY

## *Implementing TVM's Scheduling Language*



# CASE STUDY

## Optimizing Matrix Multiplication - Baseline



RISE

What to compute

```
1 // matrix multiplication in RISE
2 val dot = fun(as, fun(bs, zip(as)(bs) |>
3   map(fun(ab, mult(fst(ab))(snd(ab)))) |>
4   reduce(add)(@) ) )
5 val mm = fun(a, fun(b, a |>
6   map( fun(arow, transpose(b) |>
7     map( fun(bcol,
8       dot(arow)(bcol) ))))) ) )
```

```
1 // baseline strategy in ELEVATE
2 val baseline = ( DFNF `;`'
3   fuseReduceMap '@' topDown )
4 (baseline `;` lowerToC)(mm)
```

tvm

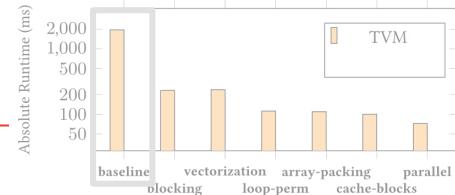
```
1 # Naive matrix multiplication algorithm
2 k = tvm.reduce_axis((0, K), 'k')
3 A = tvm.placeholder((M, K), name= 'A')
4 B = tvm.placeholder((K, N), name= 'B')
5 C = tvm.compute((M, N), lambda x, y:
6   tvm.sum(A[x, k] * B[k, y],
7   axis=k), name= 'C')
8
9
10
11
12 # TVM default schedule
13 s = tvm.create_schedule(C.op)
```

ELEVATE

How to optimize

# CASE STUDY

## Optimizing Matrix Multiplication - Baseline



clear separation

# RISE

```
1 // matrix multiplication in RISE
2 val dot = fun(as, fun(bs, zip(as)(bs) |>
3   map(fun(ab, mult(fst(ab))(snd(ab)))) |>
4   reduce(add)(@) ) )
5 val mm = fun(a, fun(b, a |>
6   map( fun(arow, transpose(b) |>
7     map( fun(bcol,
8       dot(arow)(bcol) ))))) ) )
```

```
1 // baseline strategy in ELEVATE
2 val baseline = ( DFNF `;` 
3   fuseReduceMap `@` topDown )
4 (baseline `;` lowerToC)(mm)
```

ELEVATE  
composable      explicit



```
1 # Naive matrix multiplication algorithm
2 k = tvm.reduce_axis((0, K), 'k')
3 A = tvm.placeholder((M, K), name='A')
4 B = tvm.placeholder((K, N), name='B')
5 C = tvm.compute((M, N), lambda x, y:
6   tvm.sum(A[x, k] * B[k, y],
7   axis=k), name='C')
```

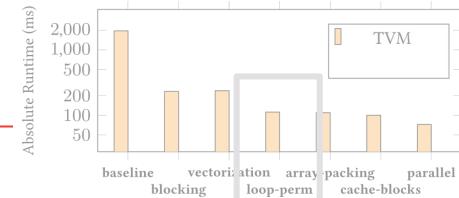
no separation

```
10
11
12 # TVM default schedule
13 s = tvm.create_schedule(C.op)
```

implicit

# CASE STUDY

## Optimizing Matrix Multiplication - Loop Permutation



facilitate reuse

user-defined vs. built-in

```
1 val loopPerm = (           user-defined
2   tile(32,32)    '@' outermost(mapNest(2))    ';;'
3   fissionReduceMap '@' outermost(appliedReduce) ';;'
4   split(4)        '@' innermost(appliedReduce) ';;'
5   reorder(Seq(1,2,5,3,6,4))
6   vectorize(32)   '@' innermost(isApp(isApp(isMap))))
7   (loopPerm ';' lowerToC)(mm)
```

```
1 xo, yo, xi, yi = s[C].tile(           built-in
2   C.op.axis[0], C.op.axis[1], 32, 32)
3 k,                                = s[C].op.reduce_axis
4 ko, ki                           = s[C].split(k, factor=4)
5 s[C].reorder(xo, yo, ko, xi, ki, yi)
6 s[C].vectorize(yi)
```

ELEVATE



no clear separation of concerns

# CASE STUDY

## Optimizing Matrix Multiplication - Parallel

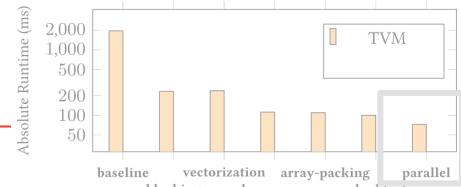
clear separation of concerns vs. no clear separation

facilitate reuse

```
1 val appliedMap = isApp(isApp(isMap))
2 val isTransposedB = isApp(isTranspose)
3
4 val packB = storeInMemory(isTransposedB,
5   permuteB `;`;
6   vectorize(32) `@` innermost(appliedMap) `;`;
7   parallel `@` outermost(isMap)
8 ) `@` inLambda
9
10 val par = (
11   packB `;` loopPerm `;`;
12   (parallel `@` outermost(isMap)),
13   `@` outermost(isToMem) `;`;
14   unroll `@` innermost(isReduce))
15
16 (par `;` lowerToC )(mm)
```

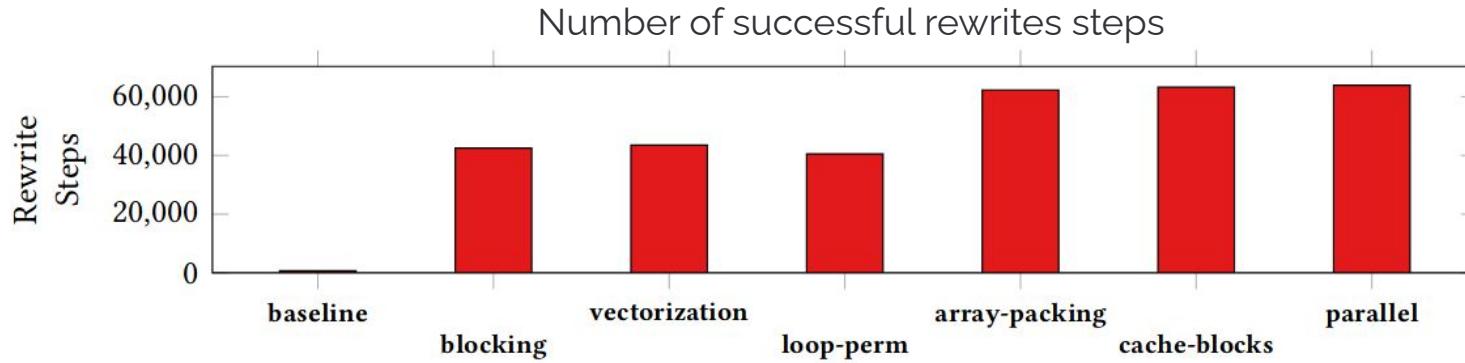
ELEVATE

```
1 # Modified algorithm
2 bn = 32
3 k = tvm.reduce_axis((0, K), 'k')
4 A = tvm.placeholder((M, K), name='A')
5 B = tvm.placeholder((K, N), name='B')
6 pB = tvm.compute((N / bn, K, bn),
7   lambda x, y, z: B[y, x * bn + z], name='pB')
8 C = tvm.compute((M, N), lambda x, y:
9   tvm.sum(A[x, k] * pB[y//bn, k,
10    tvm.indexmod(y, bn)], axis=k), name='C')
11 # Array packing schedule
12 s = tvm.create_schedule(C.op)
13 CC = s.cache_write(C, 'global')
14 xo, yo, xi, yi = s[C].tile(
15   C.op.axis[0], C.op.axis[1], bn, bn)
16 s[CC].compute_at(s[C], yo)
17 xc, yc = s[CC].op.axis
18 k, = s[CC].op.reduce_axis
19 ko, ki = s[CC].split(k, factor=4)
20 s[CC].reorder(ko, xc, ki, yc)
21 s[CC].unroll(ki)
22 s[CC].vectorize(yc)
23 s[C].parallel(xo)
24 x, y, z = s[pB].op.axis
25 s[pB].vectorize(z)
26 s[pB].parallel(x)
```



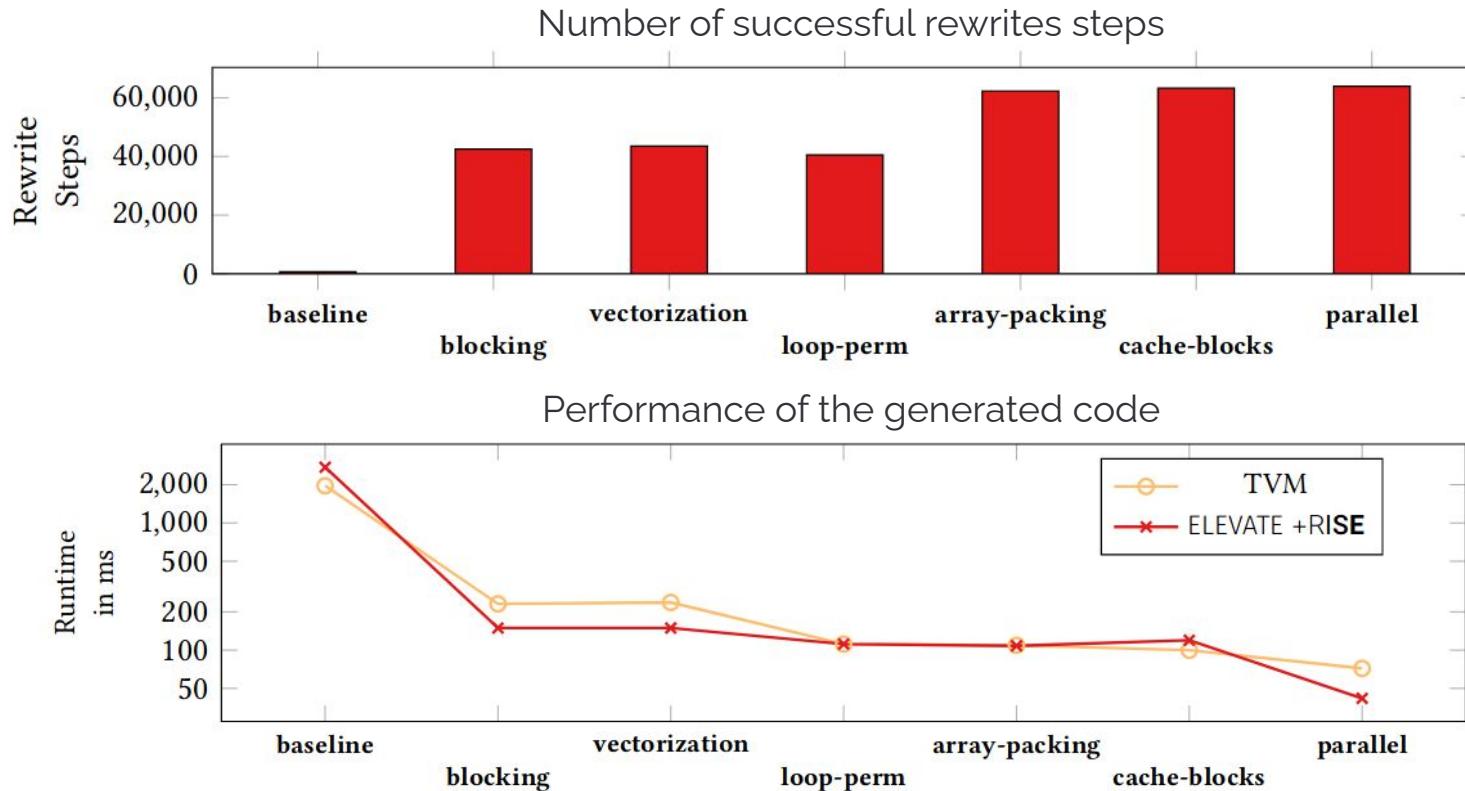
# CASE STUDY

## *Counting Rewrite Steps and Measuring Performance*

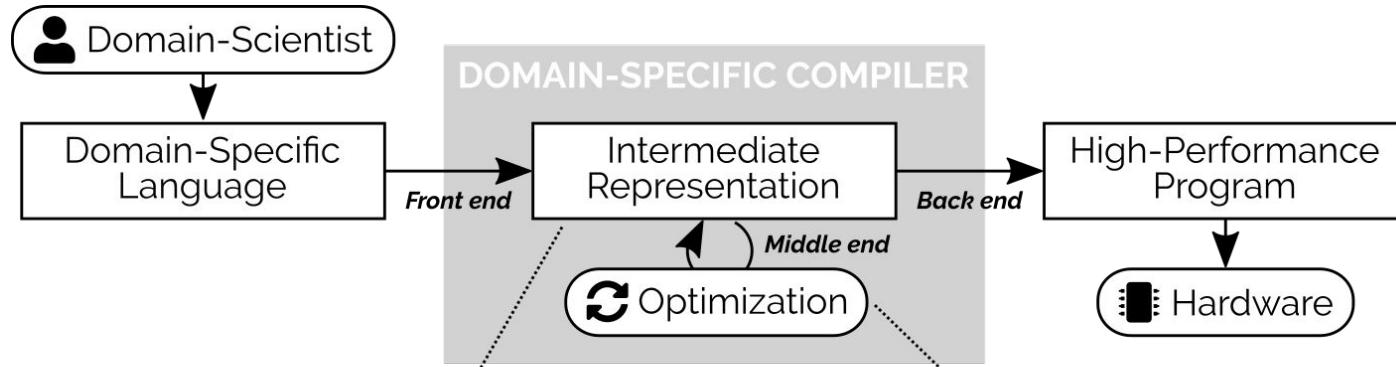


# CASE STUDY

## *Counting Rewrite Steps and Measuring Performance*



# HIGH PERFORMANCE DOMAIN-SPECIFIC COMPIRATION WITH DOMAIN-SPECIFIC COMPILERS



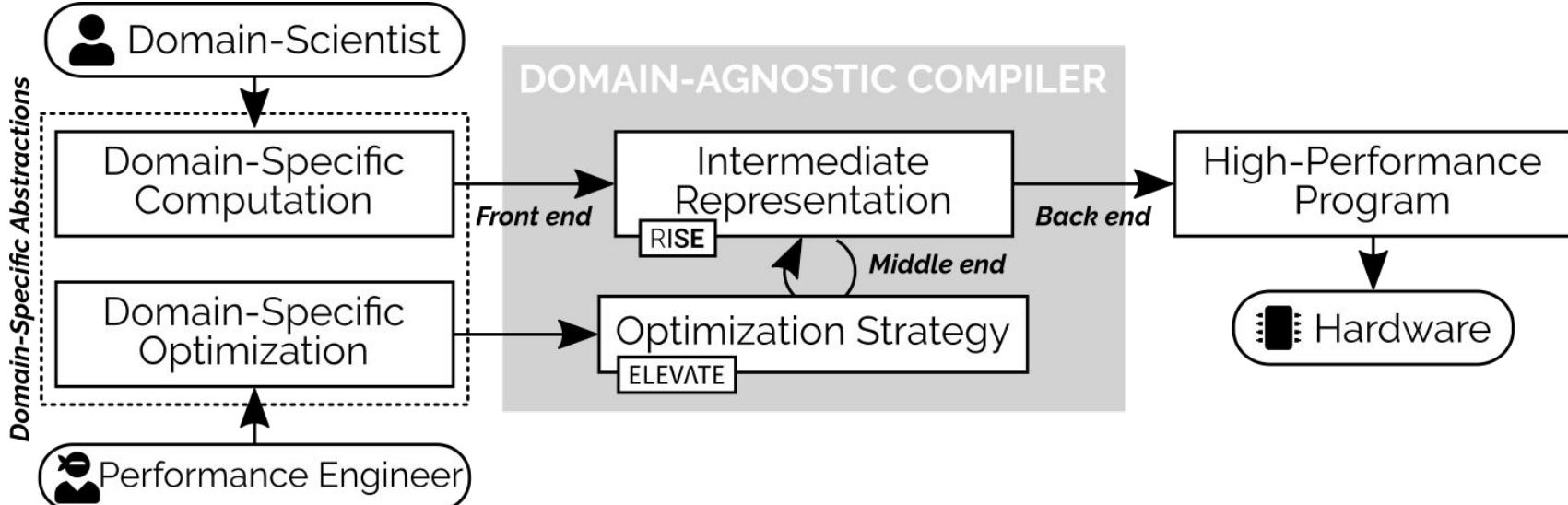
## ***The Intermediate Representation Challenge:***

"How to define an IR for high-performance domain-specific compilation that can be reused across application domains and hardware architectures while providing multiple levels of abstraction?"

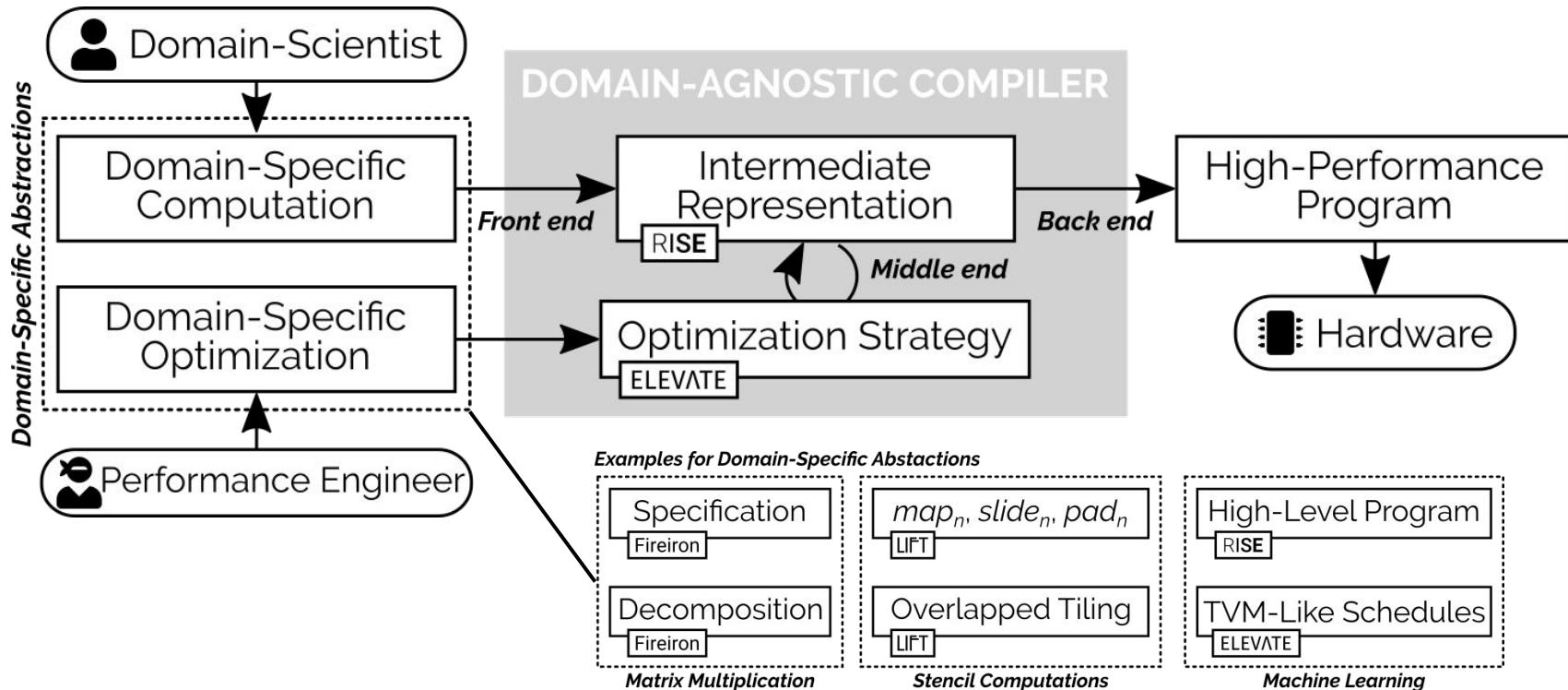
## ***The Optimization Challenge:***

"How can we encode and apply domain-specific optimizations for high-performance code generation while providing precise control and the ability to define custom optimizations, thus achieving a reusable optimization approach across application domains and hardware architectures?"

# HIGH PERFORMANCE DOMAIN-SPECIFIC COMPIRATION WITHOUT DOMAIN-SPECIFIC COMPILERS



# HIGH PERFORMANCE DOMAIN-SPECIFIC COMPIRATION WITHOUT DOMAIN-SPECIFIC COMPILERS



HIGH-PERFORMANCE  
DOMAIN-SPECIFIC COMPIRATION  
WITHOUT  
DOMAIN-SPECIFIC COMPILERS

thanks for your attention.

# BACKUP SLIDES

# PAD AND SLIDE ARE REUSABLE!

## Machine Learning - Strided Convolution

```

1 def partialConv(kernelsWeights : [[[float]]], inputChannels, kernelWidth, kernelHeight, numKernels,
2                 paddedInput : [[[float]]], inputChannels, paddedInputWidth, paddedInputHeight,
3                 kernelStride : (int, int),
4                 : [[[[float]]]]) {
5     val tiledInput4D = join(tiled2D(θ, tilingStride, paddedInput))
6     val tiledSlidedInput4D = map(join(tiled2D(kernelHeight, kernelWidth), kernelStride))
7     val windowSize = inputChannels * kernelWidth * kernelHeight
8     def coalesceChunkVectorizeWindow(window : [[[float]]]) : [[[float]]] = {
9         val flatWindowID = join(join(window))
10        val flatCoalescedWindowID = reorder(strideIndex(windowSize / w, flatWindowID))
11        val flatCoalescedWindowID = split(w, flatCoalescedWindowID)
12        asVector(v, flatCoalescedWindowID)
13    }
14    val tiledSlidedCoalescedChunkedVectorizedInput4D = map(tile4D -> split(σ, map(wi,
15        coalesceChunkVectorizeWindow(windowSize / tile4D), tiledSlidedInput5D))
16        groupedCoalescedChunkedVectorizedKernelsWeights4D = split(x, map(singleKernel,
17            coalesceChunkVectorizeWindow(singleKernelWeights), kernelsWeights))
18    mapWrg(i, inputTiled3D ->
19        mapWrg(θ, kernelsGroupWeights3D -> transpose(
20            mapL(1, inputWindows2D -> transpose(
21                mapL(θ, (inputWindowsChunk1D, kernelsGroupChunk2D) ->
22                    mapSeq(singleKernelReducedChunk -> toGlobal(singleKernelReducedChunk),
23                        join(
24                            reduceSeq(
25                                init = mapSeq(toPrivate(id(value(θ, [float]))),
26                                f = (acc, (inputsValue, kernelsGroupValue1D)) ->
27                                    let(inputsValuePrivate ->
28                                        mapSeq((accValue, singleKernelValue) ->
29                                            mapSeq((inputValuePrivate) ->
30                                                accValue + vectorize(v, dot(inputValuePrivate, single
31                                                inputsValuePrivate,
32                                                kernelsGroupValue1D),
33                                                private(vectorize(v, id(inputsValue))),
34                                                InputWindowsChunk1D), transpose(kernelsGroupChunk2D),
35                                                transpose(kernelsGroupWeights3D))),
36                                            orizedKernelsWeights4D),
37                                            storizedInput4D)
38                            )
39                        )
40                    )
41                )
42            )
43        )
44    )
45}

```



```

object PoolCPU3D {
  def apply(fc: FunCall, in: Expr) : Expr = {
    val p = fc.f.asInstanceOf[AveragePool]
    val kernel_shape = p.kernel_shape
    assert(kernel_shape.length == 3)

    val counts = kernel_shape.reduce(_*)

    val steps = p.strides
    assert(steps.length == 3)

    CPUFunc(MapSeq(MapSeq(MapSeq( MapSeq( MapSeq( Join() o MapSeq( MapSeq( Join() o MapSeq(
      fun(y =>
        ReduceSeq(add, 0.0f) o
        mapWrg(i => mapSeq(y -> add(0.0f) $ y)
      ) o
      a Slide3D.Rk(kernel_shape(i), steps(i), kernel_shape(1), steps(1), kernel_shape(2), steps(2)) ) $ in
      ), kernelGroupValue1D),
      private(vectorize(v, id(inputsValue))),
      InputWindowsChunk1D), transpose(kernelsGroupChunk2D),
      transpose(kernelsGroupWeights3D))),
      storizedInput4D)
    )
  )
}

```

REWRITE RULE 5. Dimensionality Change  
For  $x : [[T]_n]_m$  and  $y : [U]_n$

$map(map(f) \circ zip(y))(x) \rightarrow$   
 $split(n) \circ map(f) \circ zip(pad(0, (m-1)n, wrap y), join(x))$

This rule expresses the fact that instead of zipping the same values and applying a function to the results in parallel, we can concatenate the same values, zip function over all the values of  $x$  in a single

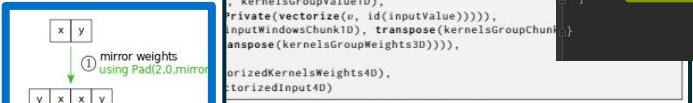


```

src: R.6
  > split(k)
  > mapBlock(chunk =>
    chunk |> split(j)
    > mapThread(threadChunk =>
      threadChunk
      |> reduceSeq(@, neutral(@))
      |> toPrivate
    )
  )
  |> split(32)
  > mapWarp(warpChunk =>
    warpChunk
    |> reduceWarp(@, neutral(@))
  )
  > toLocal
  > padCst(0) (32 - (k/j)) / 32 (neutral(@))

  ...
  |> mapWarp(warpChunk =>
    warpChunk
    |> mapThread(threadValue =>
      threadValue
      |> toPrivate
    )
    |> reduceWarp(@, neutral(@))
  )
  |> toGlobal
}

```



## Machine Learning - Pooling

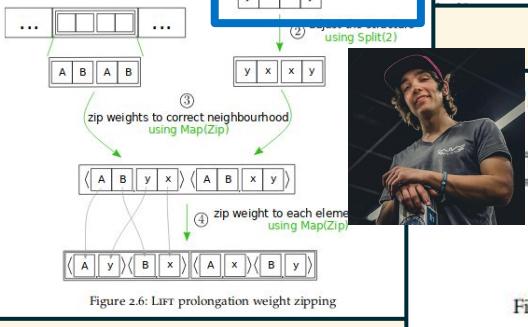


Figure 2.6: LIFT prolongation weight zipping

## Numerical Solvers - Multigrid Methods

$$p(l : 2, r : 1, in : [1, 2, 3, 4, 5, 6]) \rightarrow [3, 4, 5] \quad (2.3)$$



Figure 2.7: LIFT prolongation deleting unnecessary values

## GPUs - Efficient Reductions

# STRATEGO

*Comparison to Visser et. al.*

From ICFP'98:

```
signature
sorts TExp Vdec Fdec Se Exp
operations
  Funtype : List(TExp) * TExp    -> TExp   -- Type expressions
  Recordtype : List(TExp)        -> TExp
  Printtype : String            -> TExp
  Vdec   : TExp * String * Exp -> Vdec   -- Variable declarations
  Fdec   : TExp * String * Exp -> Fdec   -- Function declarations
  Const  : TExp * String       -> Se     -- Simple expressions
  Var   : String               -> Se
  Simple : Se                  -> Exp   -- Expressions
  Record : List(Se)            -> Exp
  Select : Int * Se            -> Exp
  Papp   : String * List(Se)   -> Exp
  App    : Se * List(Se)       -> Exp
  Let    : Vdec * Exp          -> Exp
  Letrec : List(Fdec) * Exp    -> Exp
rules
Hoist1 : Let(Vdec(t, x, Let(vdec, e1)), e2) -> Let(vdec, Let(Vdec(t, x, e1), e2))
Hoist2 : Let(Vdec(t, x, Letrec(fdecs, e1)), e2) -> Letrec(fdecs, Let(Vdec(t, x, e1), e2))
Dead1 : Let(Vdec(t, x, e1), e2) -> e2 where not(<in> (Var(x), e2)); <safe> e1
Dead2 : Letrec(fdecs, e1) -> e1 where <map>(f : match(Fdec(., f, ., .)); not(<in> (Var(f), e1))))> fdecs
Prop  : Let(Vdec(t, x, Simple(se)), e[Var(x)]) -> Let(Vdec(t, x, Simple(se)), e[se](sometd))
Inl1  : Letrec([Fdec(t, f, xs, e1)], e2[App(Var(f), ss)]) ->
        Letrec([Fdec(t, f, xs, e1)], e2[<rsubs; rrename> (xs, ss, e1)](sometd))
        where <small> e1
Inl2  : Letrec([Fdec(t, f, xs, e1)], e2[App(Var(f), ss)]) ->
        Letrec([Fdec(t, f, xs, e1)], e2[<rsubs; rrename> (xs, ss, e1)](onctd))
Sel   : Let(Vdec(t, x, Record(ss)), e[Select(i, Var(x))]) ->
        Let(Vdec(t, x, Record(ss)), e[Simple(<index> (i, ss))](sometd))
EtaExp : Let(Vdec(Funtype(ts, t), f1, e1), e2) ->
        Letrec([Fdec(Funtype(ts, t), f1, xs, Let(Vdec(Funtype(ts, t), f2, e1), App(Var(f2), ses)))], e2)
        where <safe> e1; new -> f2; <map(new)> ts -> xs; <map(MkVar)> xs -> ses
strategies
opt1 = innermost'(Hoist1 + Hoist2);
manydownup((Inl1 <-> (Inl2; Dead2) + Sel + Prop); repeat(Dead1 + Dead2) <-> repeat1(Dead1 + Dead2))
optimize1 = bottomup(try(EtaExp)); repeat(opt1)

opt2 = rec x(repeat(Hoist1); try(Hoist2);
             tryLet(id, x); try(Prop + Sel); try(Dead1; x)
             + Letrec(id, x); (Dead2 <-> try(Letrec(map(Fdec(id,id,id,x)),id),
                                         try((Inl1; try(Dead2) <-> Inl2; Dead2); x)))))

optimize2 = bottomup(try(EtaExp)); opt2
```

Target Language: RML (Reduced ML)

Rewrite Rules (e.g., Dead Code Elimination)

Our work:

- Focus on *high performance*
- Competitive to state-of-the-art optimizing compilers
- Traversals + Strategy Predicates
- Normal-forms (e.g., DNF)

2 Optimization Strategies

# LOG-SCALE SPEEDUP PLOT

