HIGH-PERFORMANCE DOMAIN-SPECIFIC COMPILATION WITHOUT DOMAIN-SPECIFIC COMPILERS

BASTIAN HAGEDORN

Disputation | 24.08.2020 | Westfälische Wilhelms-Universität Münster

WHAT IS HIGH-PERFORMANCE DOMAIN-SPECIFIC COMPILATION ?

AND WHY DO WE WANT IT?

WITHOUT
DOMAIN-SPECIFIC COMPILERS

BASTIAN HAGEDORN

A SOLVED PROBLEM

HOW TO MAKE HIGH PERFORMANCE ACCESSIBLE TO DOMAIN SCIENTISTS?

Domain-Scientist

REQUIRES HIGH PERFORMANCE
FOR SCIENTIFIC APPLICATIONS

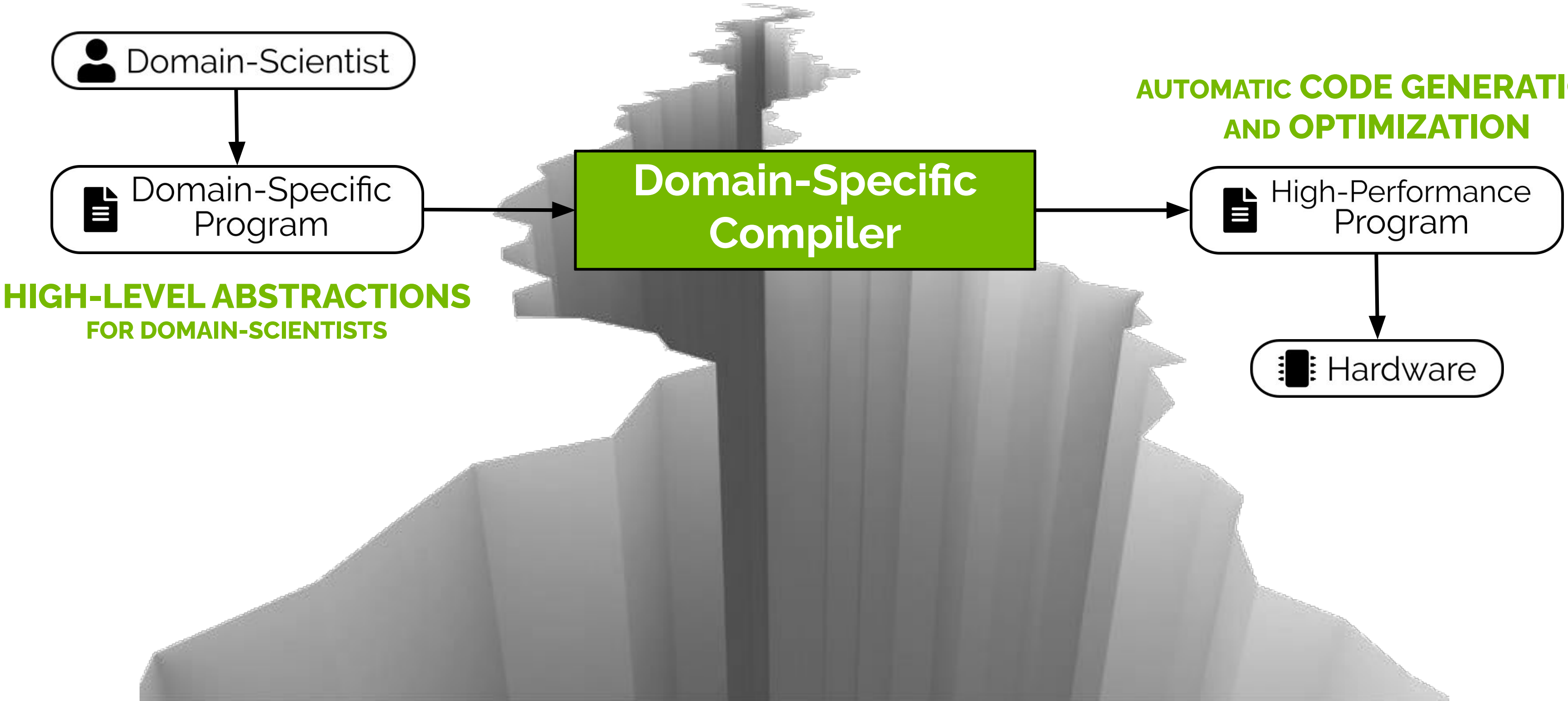
MODERN PARALLEL

Hardware

PROVIDES HIGH PERFORMANCE
BUT IS HARD TO PROGRAM

A SOLVED PROBLEM

HOW TO MAKE HIGH PERFORMANCE ACCESSIBLE TO DOMAIN SCIENTISTS?



HIGH-PERFORMANCE DOMAIN-SPECIFIC COMPILATION

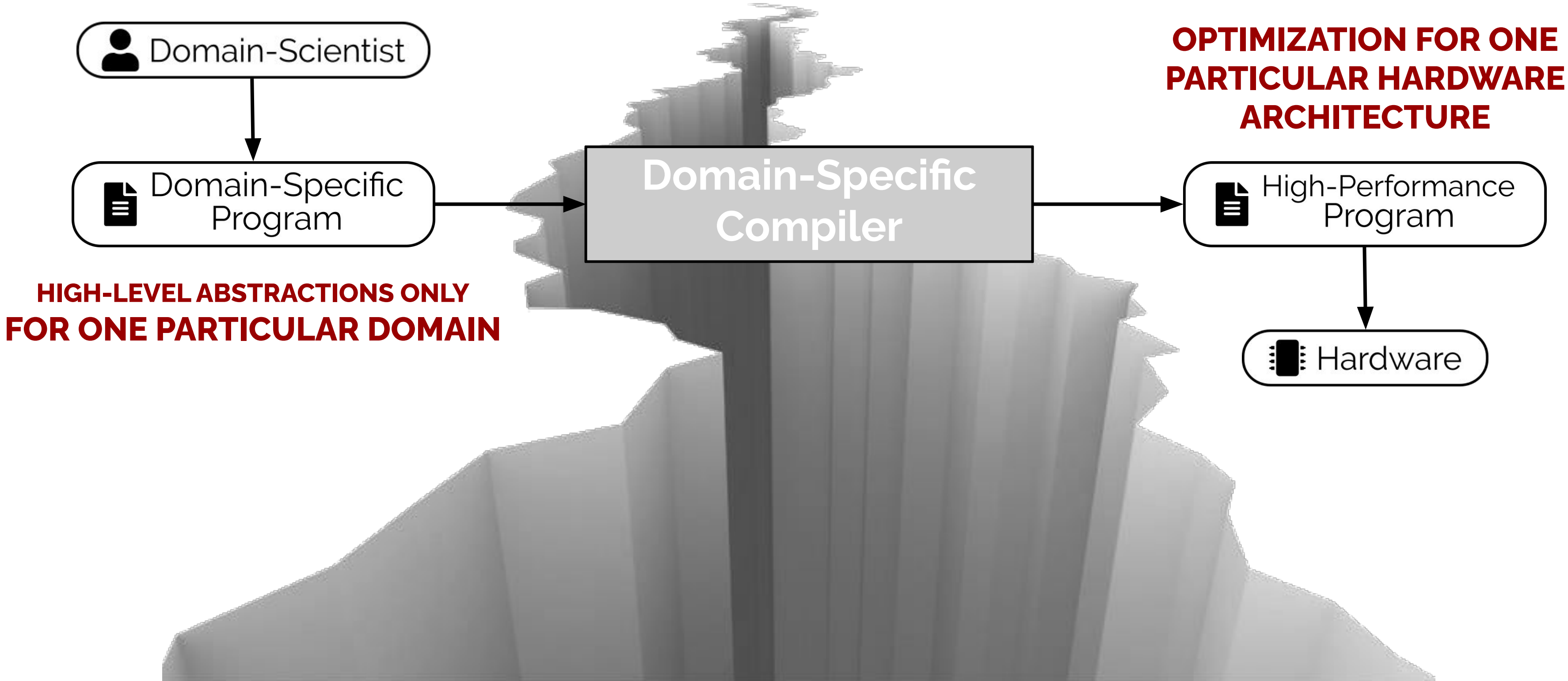
WHY TRYING TO ACHIEVE THIS WITHOUT DOMAIN-SPECIFIC COMPILERS?

WHAT IS WRONG WITH THEM?

BASTIAN HAGEDORN

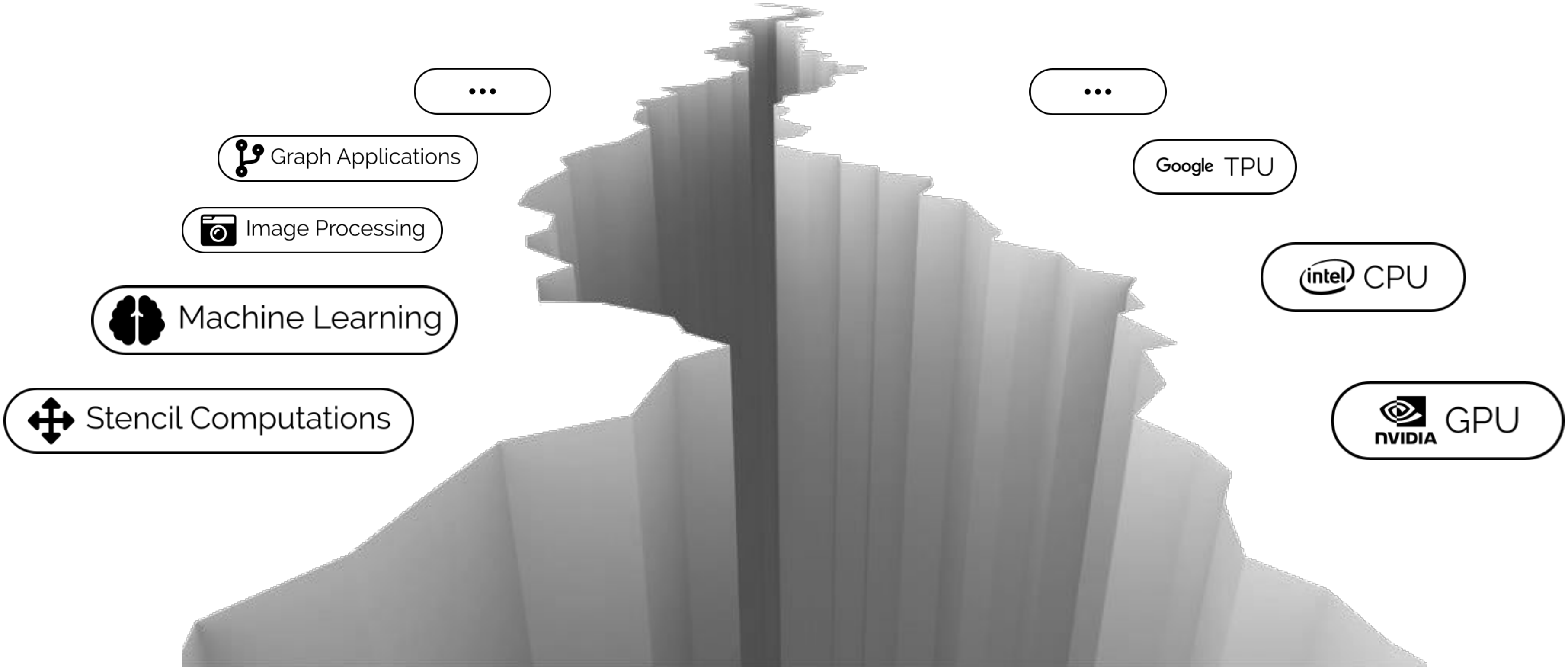
THE NEW CHALLENGE

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

GraphIt-DSL / **graphit**

halide / **Halide**

tvm

PolyMage

TensorFlow

Tensor Comprehensions

The Pochoir Stencil Compiler

...

Graph Applications

Image Processing

Machine Learning

Stencil Computations

...

Google TPU

intel CPU

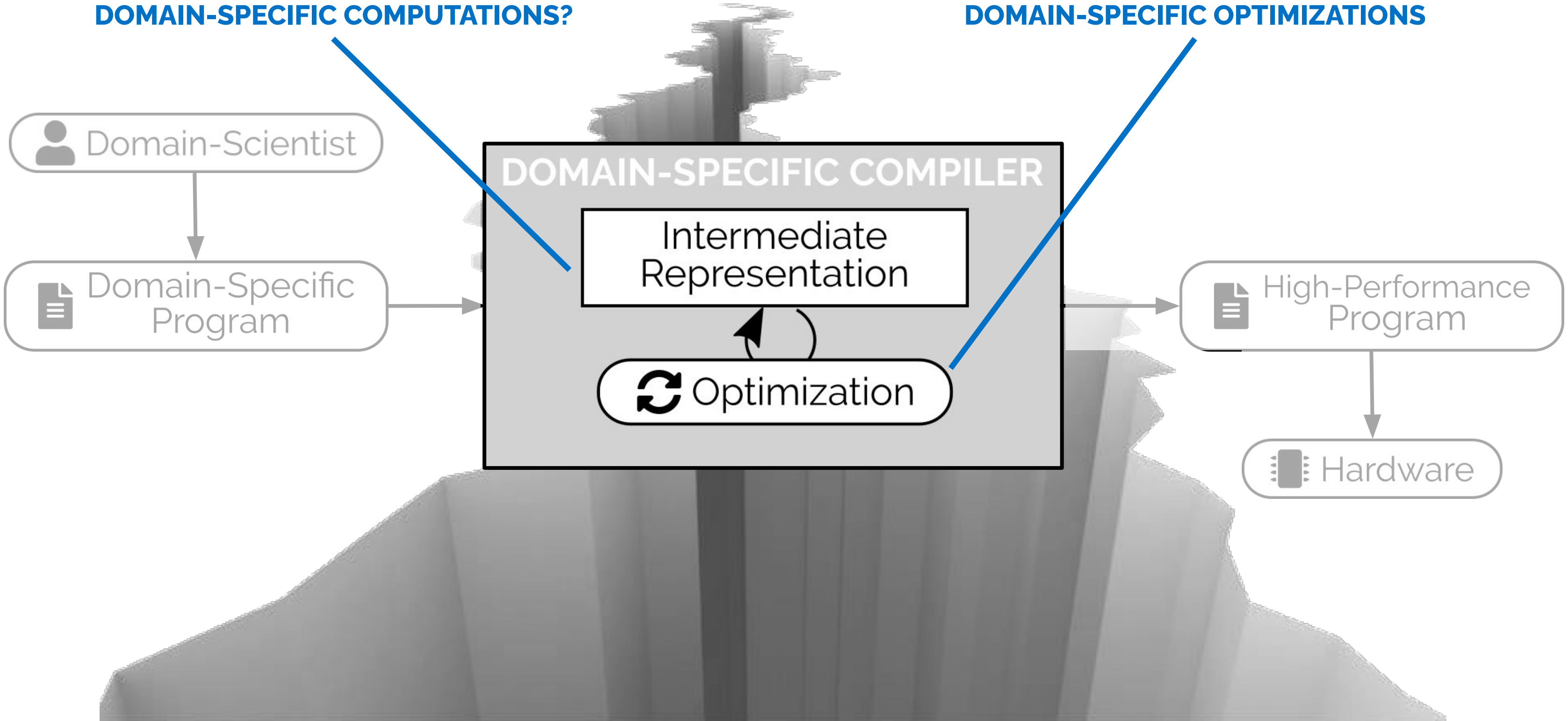
NVIDIA GPU

THE NEW CHALLENGE

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

QUESTION 1: **HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?**

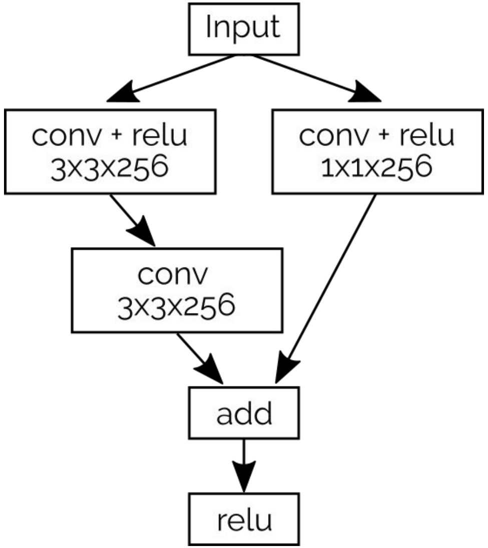
QUESTION 2 : **HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS**



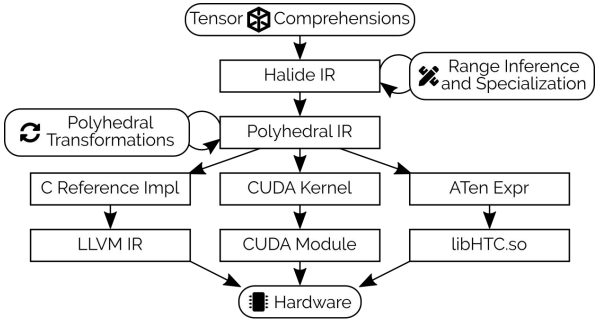
THE IR CHALLENGE

HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?

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




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



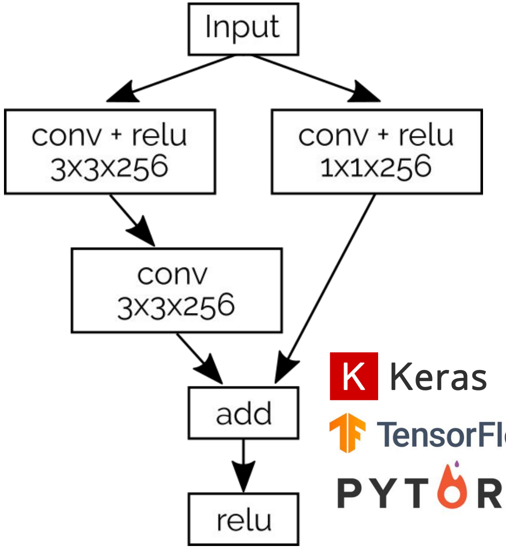
HIGH-LEVEL INTERMEDIATE REPRESENTATIONS

LOW-LEVEL INTERMEDIATE REPRESENTATIONS

HIERARCHICAL INTERMEDIATE REPRESENTATIONS

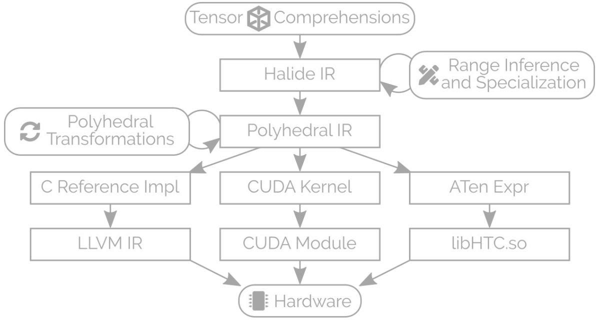
THE IR CHALLENGE

HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?



Keras
TensorFlow
PYTORCH

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



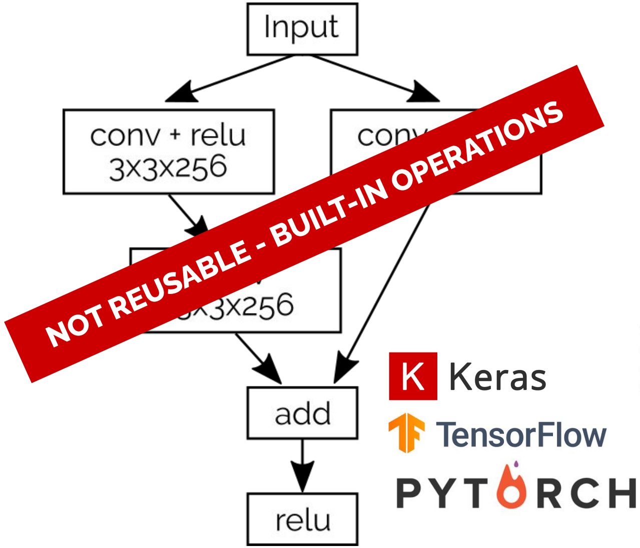
HIGH-LEVEL INTERMEDIATE REPRESENTATIONS

LOW-LEVEL INTERMEDIATE REPRESENTATIONS

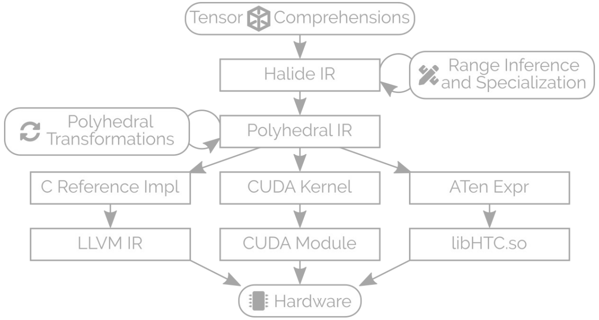
HIERARCHICAL INTERMEDIATE REPRESENTATIONS

THE IR CHALLENGE

HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?




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



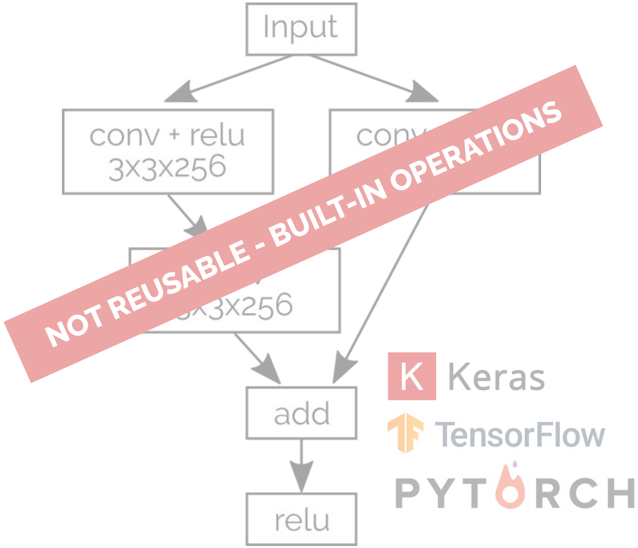
HIGH-LEVEL INTERMEDIATE REPRESENTATIONS

LOW-LEVEL INTERMEDIATE REPRESENTATIONS

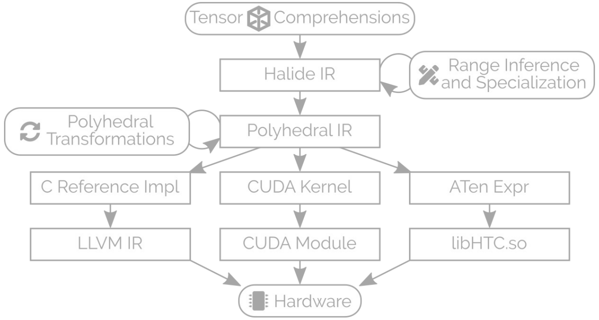
HIERARCHICAL INTERMEDIATE REPRESENTATIONS

THE IR CHALLENGE

HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?




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



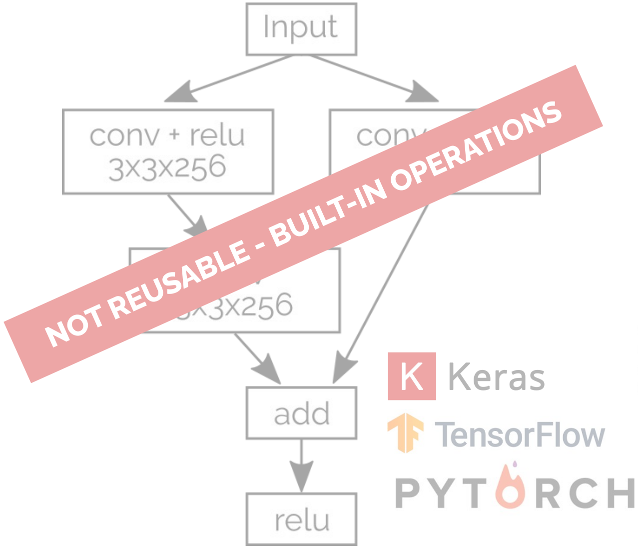
HIGH-LEVEL INTERMEDIATE REPRESENTATIONS

LOW-LEVEL INTERMEDIATE REPRESENTATIONS

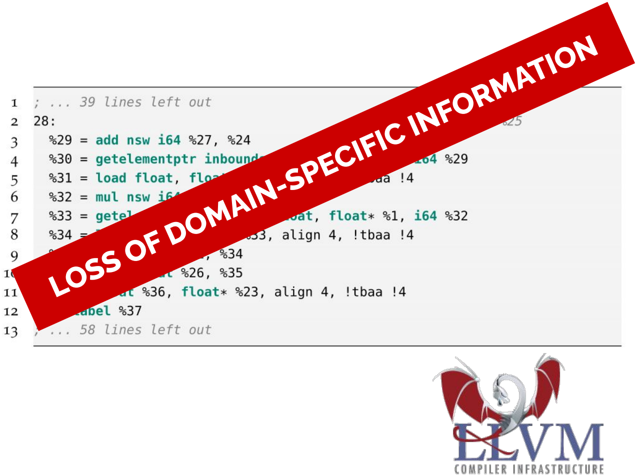
HIERARCHICAL INTERMEDIATE REPRESENTATIONS

THE IR CHALLENGE

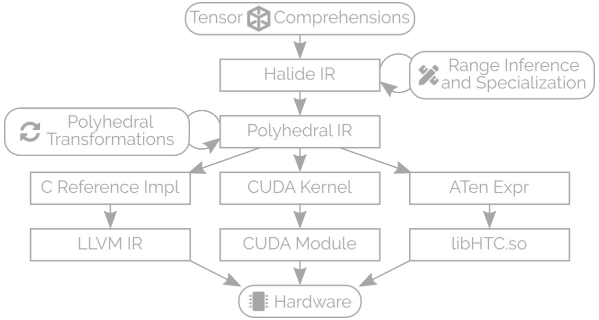
HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?



HIGH-LEVEL INTERMEDIATE REPRESENTATIONS



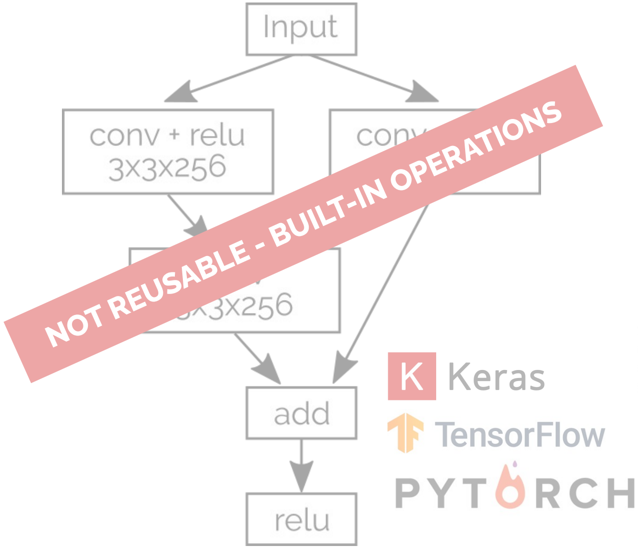
LOW-LEVEL INTERMEDIATE REPRESENTATIONS



HIERARCHICAL INTERMEDIATE REPRESENTATIONS

THE IR CHALLENGE

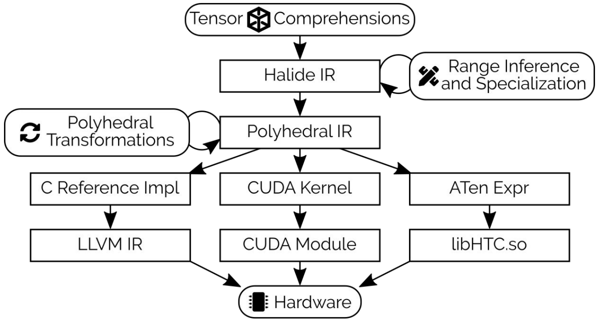
HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?



HIGH-LEVEL INTERMEDIATE REPRESENTATIONS

LLVM COMPILER INFRASTRUCTURE

LOW-LEVEL INTERMEDIATE REPRESENTATIONS

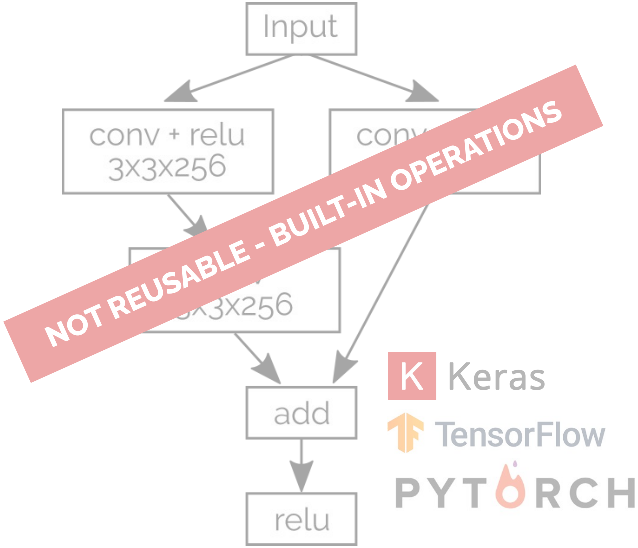


MLIR Tensor Comprehensions

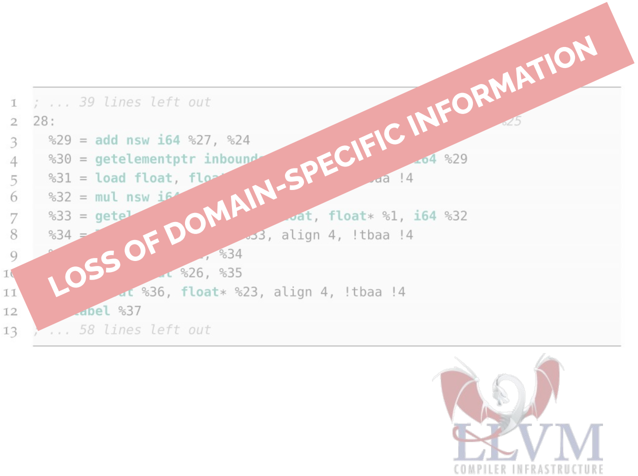
HIERARCHICAL INTERMEDIATE REPRESENTATIONS

THE IR CHALLENGE

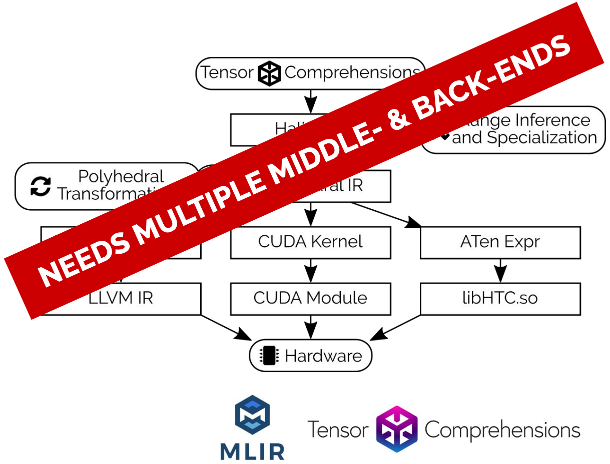
HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?



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

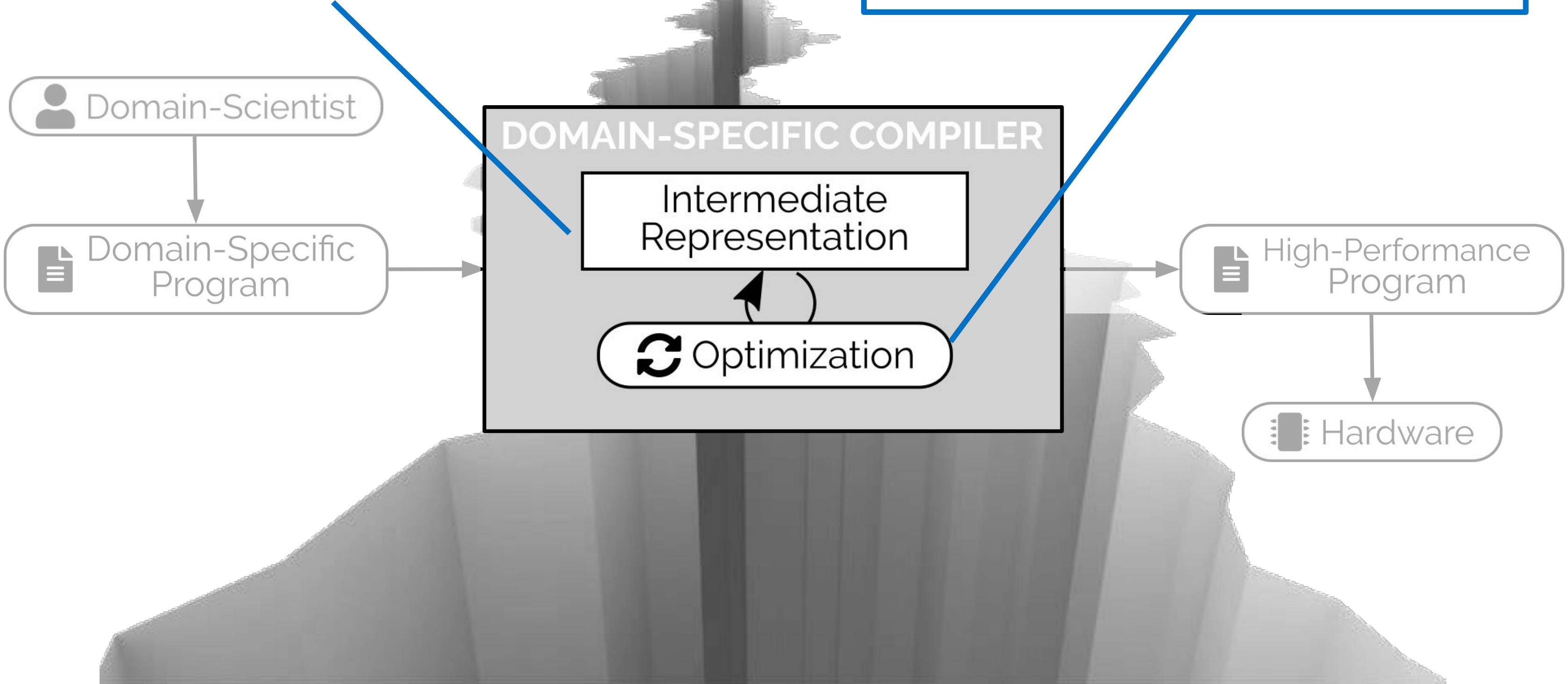
HIERARCHICAL INTERMEDIATE REPRESENTATIONS

THE NEW CHALLENGE

HOW TO DESIGN A DOMAIN-SPECIFIC COMPILER?

QUESTION 1: **HOW TO REPRESENT DOMAIN-SPECIFIC COMPUTATIONS?**

QUESTION 2 : **HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS**



THE OPTIMIZATION CHALLENGE

HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS?

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

```
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 -deadargelim -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 -memoryssa -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-verification -lcssa -basicaa -aa -scalar-
       evolution -loop-rotate -licm -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 -memcpyopt -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 -licm -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-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 -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-
       verification -lcssa -scalar-evolution -licm -lazy-branch-prob -lazy-block-freq -opt-remark-
       emitter -transform-warning -alignment-from-assumptions -strip-dead-prototypes -globaldce -
       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
```

```
// the algorithm: functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
      .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
      .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
      .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
      .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
      .unroll(xo).unroll(Ay);
```

RELYING ON LIBRARIES

HEURISTIC-BASED OPTIMIZATION

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 -deadargelim -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 -memoryssa -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-verification -lcssa -basicaa -aa -scalar-
      evolution -loop-rotate -licm -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 -memcpyopt -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 -licm -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-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 -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-
      verification -lcssa -scalar-evolution -licm -lazy-branch-prob -lazy-block-freq -opt-remark-
      emitter -transform-warning -alignment-from-assumptions -strip-dead-prototypes -globaldce -
      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

```
// the algorithm: functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidia GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
    .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
    .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
    .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
    .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
    .unroll(xo).unroll(Ay);
```

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 -deadargelim -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 -memoryssa -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-verification -lcssa -basicaa -aa -scalar-
       evolution -loop-rotate -licm -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 -memcpyopt -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 -licm -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-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 -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-
       verification -lcssa -scalar-evolution -licm -lazy-branch-prob -lazy-block-freq -opt-remark-
       emitter -transform-warning -alignment-from-assumptions -strip-dead-prototypes -globaldce -
       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

```
// the algorithm: functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
      .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
      .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
      .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
      .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
      .unroll(xo).unroll(Ay);
```

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 -deadargelim -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 -memoryssa -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-verification -lcssa -basicaa -aa -scalar-
        evolution -loop-rotate -licm -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 -memcpyopt -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 -licm -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-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 -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-
        verification -lcssa -scalar-evolution -licm -lazy-branch-prob -lazy-block-freq -opt-remark-
        emitter -transform-warning -alignment-from-assumptions -strip-dead-prototypes -globaldce -
        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

```
// the algorithm: functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
    .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
    .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
    .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
    .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
    .unroll(xo).unroll(Ay);
```

SCHEDULE-BASED OPTIMIZATION

THE OPTIMIZATION CHALLENGE

HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS?

RELYING ON LIBRARIES

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

```
Pass Arguments:  -targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-
      info -forceattrs -inferattrs -domtree -callsite-splitting -ipsccp -called-value-propaga
      attributor -globalopt -domtree -mem2reg -deadargelim -domtree -basicaa -aa -loops
      prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -basiccg
      -eh -inline -functionattrs -argpromotion -domtree -sroa -basicaa -aa -
      memssa -speculative-execution -basicaa -aa -lazy-value-info -jump
      propagation -simplifycfg -domtree -aggressive-instcombine -b                    nch-prob
       -lazy-block-freq -opt-remark-emitter -instcombine -li                         anch-prob -
      block-freq -lazy-branch-prob -lazy-block-freq -opt                             opt -basicaa -aa -
      loops -lazy-branch-prob -lazy-block-freq -op                               im -simplifycfg -
      reassociate -domtree -loops -loop-simpl                               ssa -basicaa -aa -scalar-
      evolution -loop-rotate -licm -loo                               tree -basicaa -aa -loops -lazy-
      branch-prob -lazy-block-freq                               mbine -loop-simplify -lcssa-
      verification -lcssa -s                               op-idiom -loop-deletion -loop-unroll -mldst-
      motion -phi-value                               -branch-prob -lazy-block-freq -opt-remark-emitter
      -gvn -phi-val                               memcpyopt -sccp -demanded-bits -bdce -basicaa -aa -loops
       -lazy                               q -opt-remark-emitter -instcombine -lazy-value-info -jump-
      t                               ation -basicaa -aa -phi-values -memdep -dse -loops -loop-simplify -
                                   sa -basicaa -aa -scalar-evolution -licm -postdomtree -adce -simplifycfg
                               -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -
                            barrier -elim-avail-extern -basiccg -rpo-functionattrs -globalopt -globaldce -
                         -globals-aa -float2int -domtree -loops -loop-simplify -lcssa-verification -lcssa -
      asicaa -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 -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-
      verification -lcssa -scalar-evolution -licm -lazy-branch-prob -lazy-block-freq -opt-remark-
      emitter -transform-warning -alignment-from-assumptions -strip-dead-prototypes -globaldce -
      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

```
// the algorithm: functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
      .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
      .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
      .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
      .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
      .unroll(xo).unroll(Ay);
```

SCHEDULE-BASED OPTIMIZATION

THE OPTIMIZATION CHALLENGE

HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS?

RELYING ON LIBRARIES

HEURISTIC-BASED OPTIMIZATION

tvm

halide / Halide

```
// the algorithm: functional description of matrix multiplication
Var x("x"), y("y"); Func prod("prod"); RDom r(0, size);
prod(x, y) += A(x, r) * B(r, y);
 out(x, y)  = prod(x, y);

// schedule for Nvidida GPUs
const int warp_size = 32; const int vec_size = 2;
const int x_tile    =  3; const int y_tile   = 4;
const int y_unroll  =  8; const int r_unroll = 1;
Var xi,yi,xio,xii,yii,xo,yo,x_pair,xiio,ty; RVar rxo,rxi;
out.bound(x, 0, size).bound(y, 0, size)
    .tile(x, y, xi, yi, x_tile * vec_size * warp_size,
          y_tile * y_unroll)
    .split(yi, ty, yi, y_unroll)
    .vectorize(xi, vec_size)
    .split(xi, xio, xii, warp_size)
    .reorder(xio, yi, xii, ty, x, y)
    .unroll(xio).unroll(yi)
    .gpu_blocks(x, y).gpu_threads(ty).gpu_lanes(xii);
prod.store_in(MemoryType::Register).compute_at(out, x)
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .unroll(xo).unroll(y).update()
    .split(x, xo, xi, warp_size * vec_size, RoundUp)
    .split(y, ty, y, y_unroll)
    .gpu_threads(ty).unroll(xi, vec_size).gpu_lanes(xi)
    .split(r.x, rxo, rxi, warp_size)
    .unroll(rxi, r_unroll).reorder(xi, xo, y, rxi, ty, rxo)
    .unroll(xo).unroll(y);
Var Bx = B.in().args()[0], By = B.in().args()[1];
Var Ax = A.in().args()[0], Ay = A.in().args()[1];
B.in().compute_at(prod, ty).split(Bx, xo, xi, warp_size)
    .gpu_lanes(xi).unroll(xo).unroll(By);
A.in().compute_at(prod, rxo).vectorize(Ax, vec_size)
    .split(Ax,xo,xi,warp_size).gpu_lanes(xi).unroll(xo)
    .split(Ay,yo,yi,y_tile).gpu_threads(yi).unroll(yo);
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
    .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
    .unroll(xo).unroll(Ay);
```

SCHEDULE-BASED OPTIMIZATION

THE OPTIMIZATION CHALLENGE

HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS?

RELYING ON LIBRARIES

HEURISTIC-BASED OPTIMIZATION

SCHEDULE-BASED OPTIMIZATION

THE OPTIMIZATION CHALLENGE:

HOW TO ENCODE AND APPLY DOMAIN-SPECIFIC OPTIMIZATIONS?

THE OPTIMIZATION CHALLENGE:

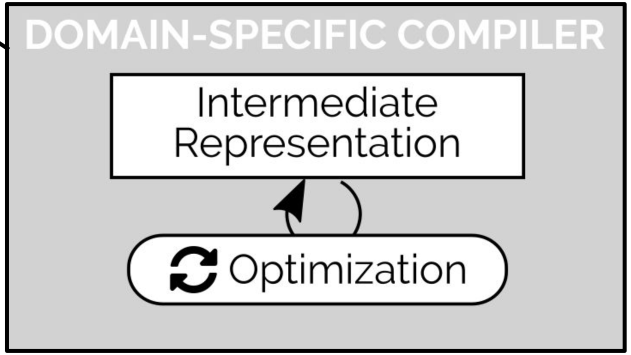
How can we encode and apply domain-specific optimizations for high-performance code generation while ***providing precise control*** *and the ability to* ***define custom optimizations,*** *thus achieving* ***a reusable optimization approach*** *across application domains and hardware architectures?*

RELYING ON LIBRARIES | *HEURISTIC-BASED OPTIMIZATION* | *SCHEDULE-BASED OPTIMIZATION*

HIGH PERFORMANCE DOMAIN-SPECIFIC COMPILATION WITHOUT DOMAIN-SPECIFIC COMPILERS

PACT'20

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



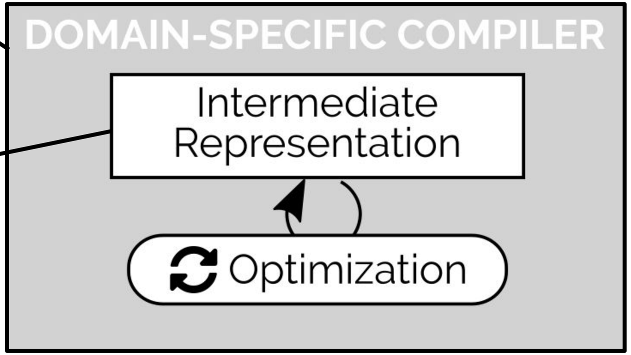
HIGH PERFORMANCE DOMAIN-SPECIFIC COMPILATION WITHOUT DOMAIN-SPECIFIC COMPILERS

PACT'20

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

CGO'18

PART II: ADDRESSING THE IR CHALLENGE
High-performance stencil computations with Lift, a generic IR for domain-specific computations



HIGH PERFORMANCE DOMAIN-SPECIFIC COMPILATION WITHOUT DOMAIN-SPECIFIC COMPILERS

PACT'20

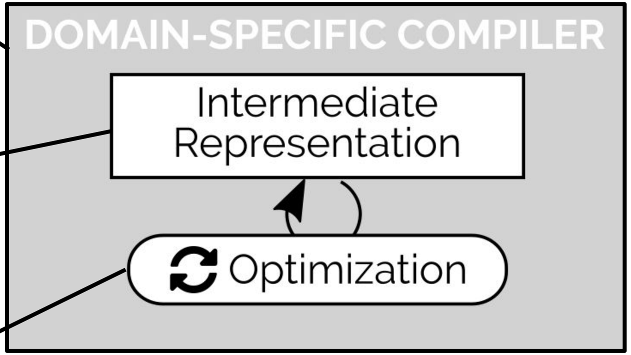
PART I: A CASE STUDY
Fireiron: A novel domain-specific compiler for GPUs that outperforms manually tuned high-performance libraries

CGO'18

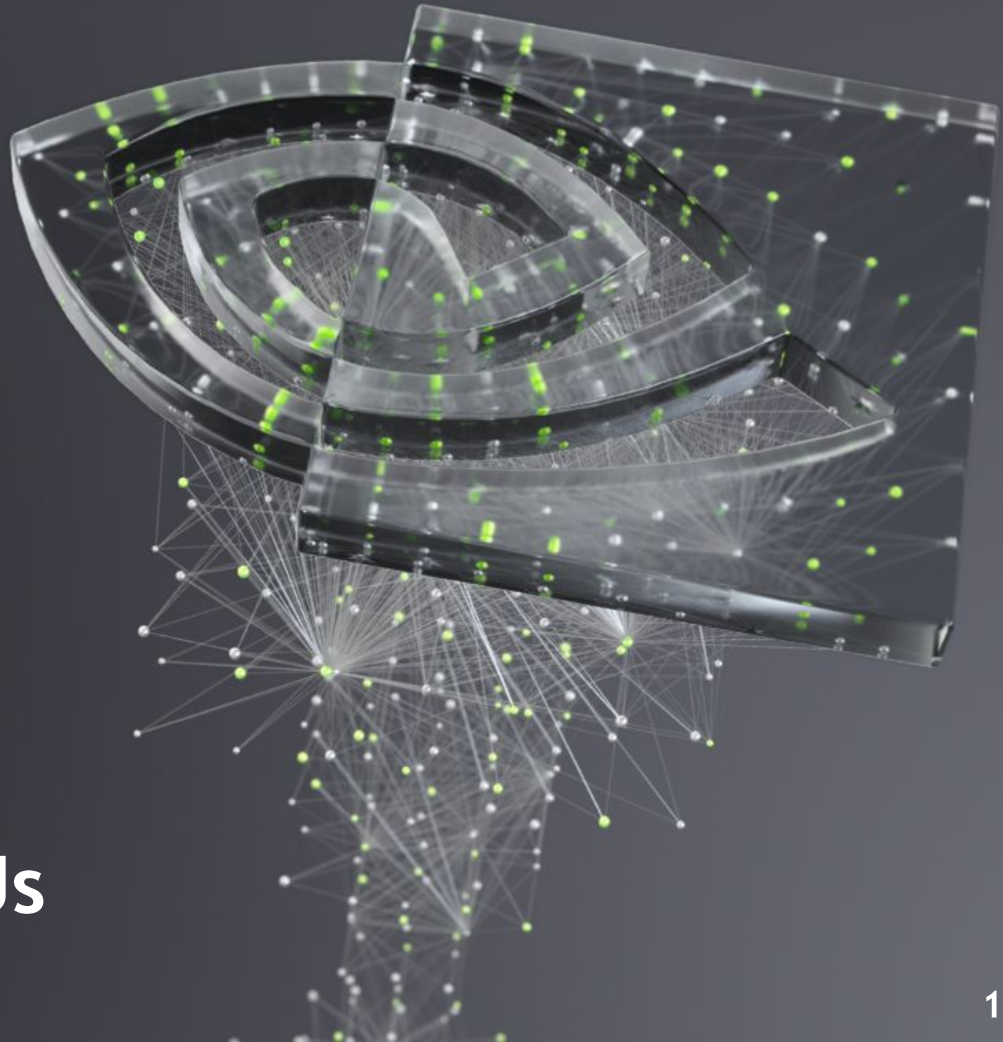
PART II: ADDRESSING THE IR CHALLENGE
High-performance stencil computations with Lift, a generic IR for domain-specific computations

ICFP'20

PART III: ADDRESSING THE OPTIMIZATION CHALLENGE
Elevate: A language for expressing optimization strategies as compositions of generic building blocks



PART I: A CASE STUDY



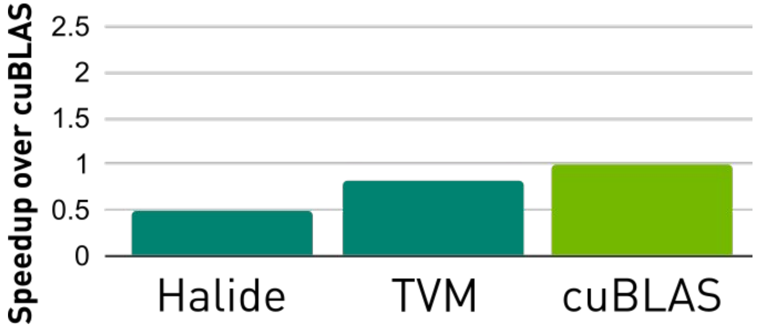
FIREIRON:
Domain-Specific Compilation for GPUs

PART I: A CASE STUDY

NVIDIA

FIREIRON: Matrix-Multiplication-Specific Compilation for GPUs

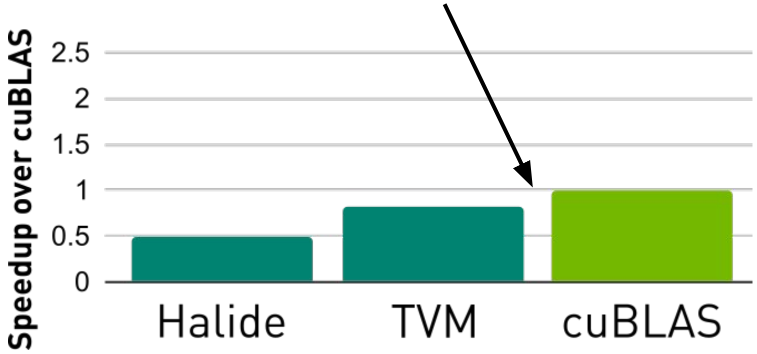
WHY YET ANOTHER DOMAIN-SPECIFIC COMPILER?



Computing Matrix Multiplication

WHY YET ANOTHER DOMAIN-SPECIFIC COMPILER?

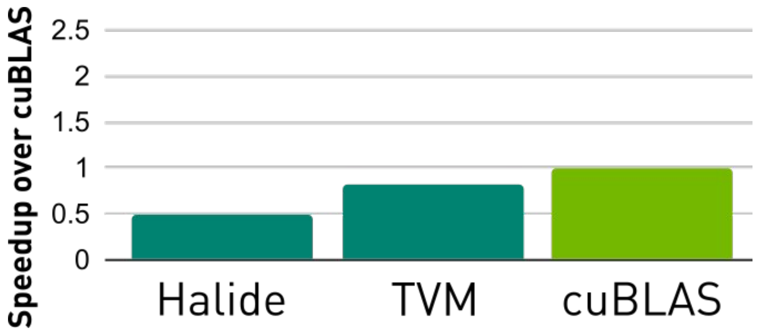
GAP IN PERFORMANCE



State-of-the-art compilers

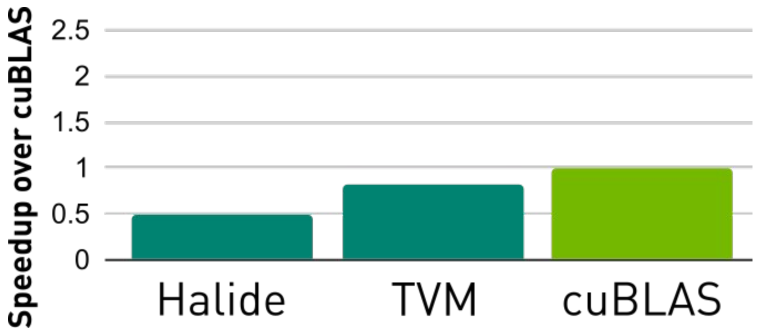
Optimized by human experts -
Best* implementation available today

WHY YET ANOTHER DOMAIN-SPECIFIC COMPILER?



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

WHY YET ANOTHER DOMAIN-SPECIFIC COMPILER?



tvm **Matrix Multiplication:**

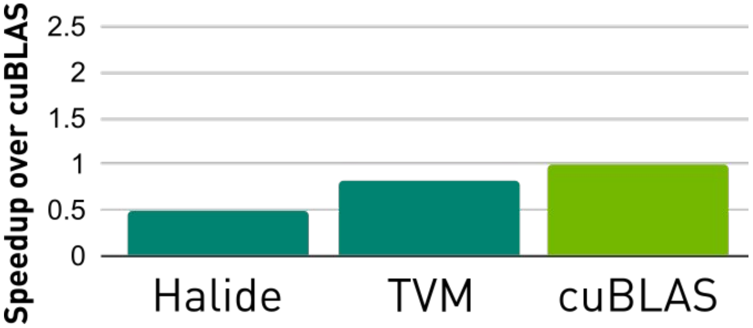
```
# Naive Algorithm
k = te.reduce_axis((0, K), 'k')
A = te.placeholder((M, K), name='A')
B = te.placeholder((K, N), name='B')
C = te.compute((M, N),
 lambda x,y: te.sum(A[x,k]*B[k,y], axis=k),name='C')
```

- `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 [illegible] split into vectors of size *k*, and each vector should be executed using [illegible]
- `im.unroll(x, k)` [illegible] that the evaluation of [illegible] should be unrolled across the dimension `x` by a factor of *k*.
- `im.split(x, xo, xi, k)` subdivides the dimension `x` into outer and inner dimensions `xo` and `xi`, where `xi` ranges from zero to `k`. `xo` and `xi` 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

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

WHY YET ANOTHER DOMAIN-SPECIFIC COMPILER?



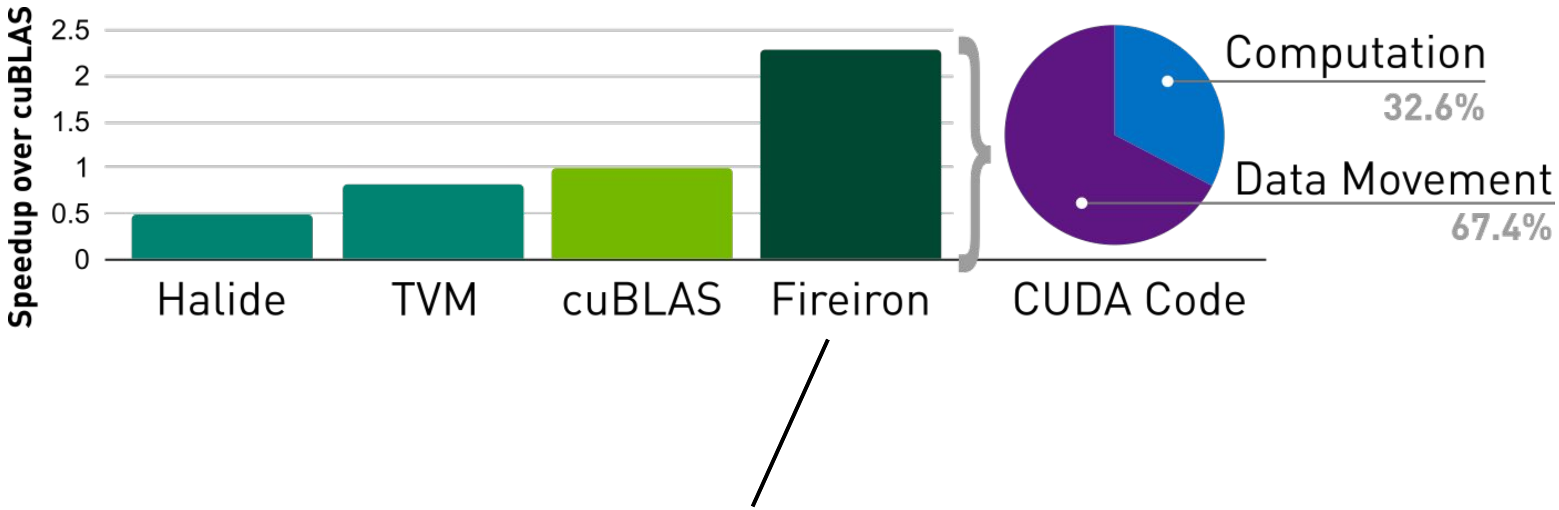
tvm **Matrix Multiplication:**

```
# 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')

# 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

```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
  __shared__ float ASH[128][8], BSH[8][128];
  float ARF[8][1], BRF[1][8], CRF[8][8];
  iBlock ← 128 * blockIdx.x;
    jBlock ← 128 * blockIdx.y;
    CRF ← 0;
    for (k ← 0; k < K / 8; k++) {
      GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))
      GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))
      __syncthreads();
      iWarp ← iBlock + warpIdx.x * 64;
        jWarp ← jBlock + warpIdx.y * 32;
          iThread ← iWarp + threadIdx.x * 8;
            jThread ← jWarp + threadIdx.y * 8
              for (kk ← 0; kk < 8; kk++)
                ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))
                ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))
                for (i ← 0; i < 8; i ++)
                  for (j ← 0; j < 8; j++)
                    CRF[i][j] += ARF[i][0] * BRF[0][j];
                  endfor
                endfor
              endfor
    endfor
    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
} // end kernel
```

Matrix Multiplication code written in (pseudo) CUDA

GPU CODE IS HIERARCHICALLY STRUCTURED

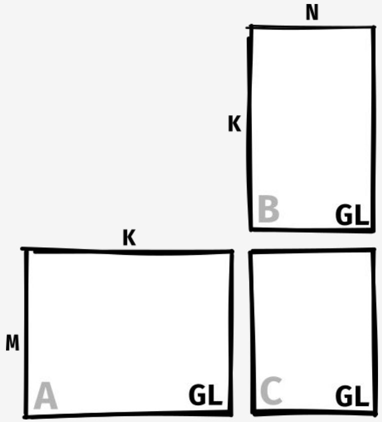
```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {

  __shared__ float ASH[128][8], BSH[8][128];
  float ARF[8][1], BRF[1][8], CRF[8][8];

  iBlock ← 128 * blockIdx.x;
    jBlock ← 128 * blockIdx.y;

    CRF ← 0;

    for (k ← 0; k < K / 8; k++) {

      GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))

      GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))

      __syncthreads();

      iWarp ← iBlock + warpIdx.x * 64;
        jWarp ← jBlock + warpIdx.y * 32;

          iThread ← iWarp + threadIdx.x * 8;
            jThread ← jWarp + threadIdx.y * 8

              for (kk ← 0; kk < 8; kk++)

                ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))

                ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))

                for (i ← 0; i < 8; i ++)
                  for (j ← 0; j < 8; j++)

                    CRF[i][j] += ARF[i][0] * BRF[0][j];

                  endfor
                endfor

              endfor


    endfor

    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)

} // end kernel
```

implements

sizes of matrices | *location in memory hierarchy* | *responsible level of compute hierarchy*

A: *B:* *C:*

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



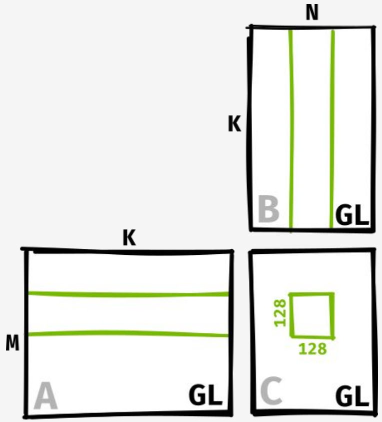
GPU CODE IS HIERARCHICALLY STRUCTURED

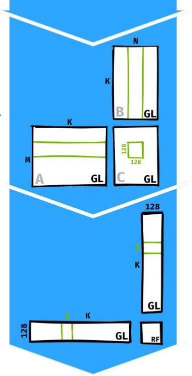
```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {

  __shared__ float ASH[128][8], BSH[8][128];
  float ARF[8][1], BRF[1][8], CRF[8][8];

  iBlock ← 128 * blockIdx.x;
    jBlock ← 128 * blockIdx.y;

    CRF ← 0;

    for (k ← 0; k < K / 8; k++) {

      GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))

      GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))

      __syncthreads();

      iWarp ← iBlock + warpIdx.x * 64;
        jWarp ← jBlock + warpIdx.y * 32;

          iThread ← iWarp + threadIdx.x * 8;
            jThread ← jWarp + threadIdx.y * 8

            for (kk ← 0; kk < 8; kk++)

              ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))

              ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))

              for (i ← 0; i < 8; i ++)
                for (j ← 0; j < 8; j++)

                  CRF[i][j] += ARF[i][0] * BRF[0][j];

                endfor
              endfor

            endfor


    endfor

    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)

} // end kernel
```

implements

sizes of matrices	*location in memory hierarchy* (A: B: C:)	*responsible level of compute hierarchy*

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





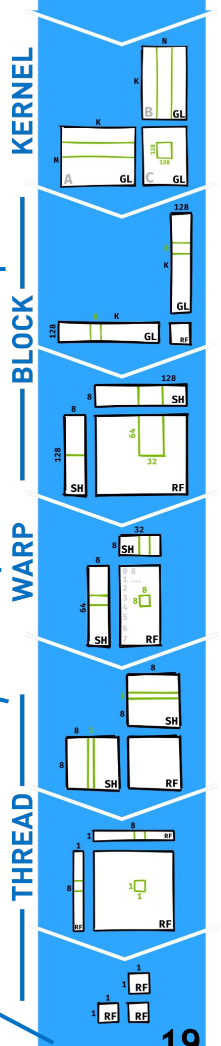
GPU CODE IS HIERARCHICALLY STRUCTURED

```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {

  __shared__ float ASH[128][8], BSH[8][128];
  float ARF[8][1], BRF[1][8], CRF[8][8];

 iBlock ← 128 * blockIdx.x;
   jBlock ← 128 * blockIdx.y;

   CRF ← 0;

   for (k ← 0; k < K / 8; k++) {

     GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))

     GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))

     __syncthreads();

     iWarp ← iBlock + warpIdx.x * 64;
       jWarp ← jBlock + warpIdx.y * 32;

         iThread ← iWarp + threadIdx.x * 8;
           jThread ← jWarp + threadIdx.y * 8

             for (kk ← 0; kk < 8; kk++)

               ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))

               ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))

               for (i ← 0; i < 8; i ++)
                 for (j ← 0; j < 8; j++)

                     CRF[i][j] += ARF[i][0] * BRF[0][j];

                 endfor
               endfor

             endfor

   endfor

   PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)

} // end kernel
```

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

A: B: C:

```
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 )
```



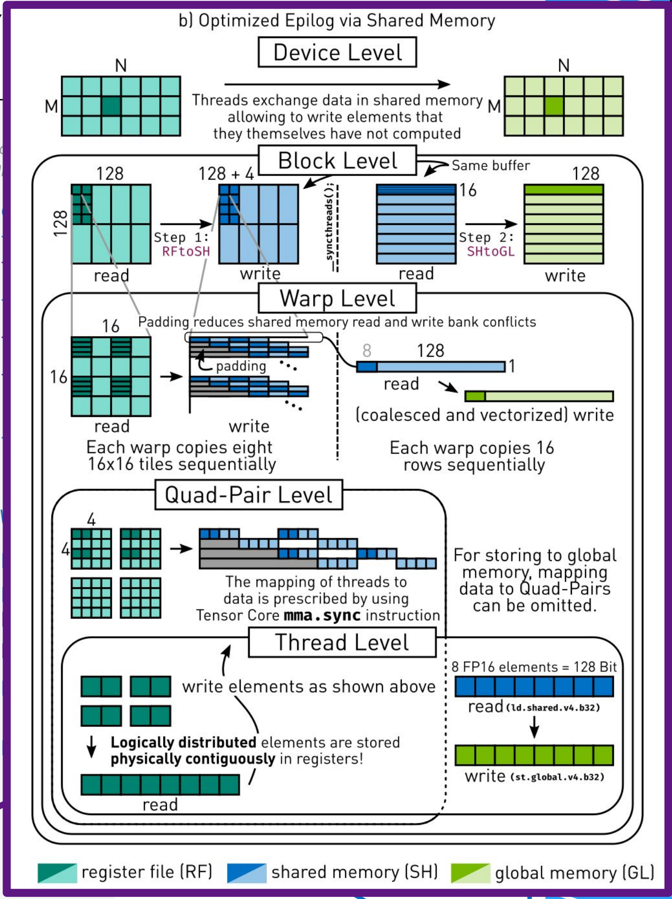
GPU CODE IS HIERARCHICALLY

```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {

  __shared__ float ASH[128][8], BSH[8][128];
  float ARF[8][1], BRF[1][8], CRF[8][8];

  iBlock ← 128 * blockIdx.x;
    jBlock ← 128 * blockIdx.y;

    CRF ← 0;

    for (k ← 0; k < K / 8; k++) {

      GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))

      GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))

      __syncthreads();

      iWarp ← iBlock + warpIdx.x * 64;
        jWarp ← jBlock + warpIdx.y * 32;

          iThread ← iWarp + threadIdx.x * 8;
            jThread ← jWarp + threadIdx.y * 8

            for (kk ← 0; kk < 8; kk++)

              ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))

              ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))

              for (i ← 0; i < 8; i ++)
                for (j ← 0; j < 8; j++)

                  CRF[i][j] += ARF[i][0] * BRF[0][j];

                endfor
              endfor

            endfor

    endfor

    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)

} // end kernel
```

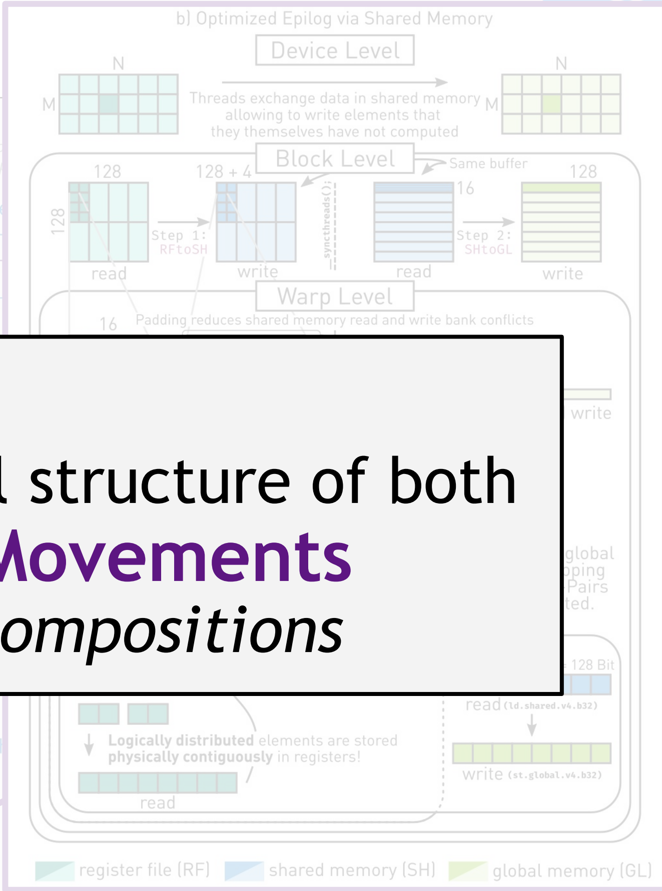
implements | *sizes of matrices* | *location in memory hierarchy* | *resp… com…*

A: B: C:

```
MatMul(M,   N,   K)(GL,GL,GL)(K
MatMul(128,128,K)(GL,GL,GL)(B
  Init(C:128×128)( dst:RF )(B
MatMul(128,128,8)(GL,GL,RF)(B
  Move( A:128×8 )(GL → SH)(B
  Move( B:8×128 )(GL → SH)(B
MatMul(128,128,8)(SH,SH,RF)(B
MatMul(64, 32, 8)(SH,SH,RF)(
MatMul(8,  8,  8)(SH,SH,RF)(T
MatMul(8,  8,  1)(SH,SH,RF)(T
  Move(  A:8×1  )(SH → RF)(T
  Move(  B:1×8  )(SH → RF)(T
MatMul(8,  8,  1)(RF,RF,RF)(T
MatMul(1,  1,  1)(RF,RF,RF)(T
  Move(C:128×128)(RF → GL)(Block )
```



GPU CODE IS HIERARCHICALLY



FIREIRON:

Programmers describe hierarchical structure of both

Computations and **Data Movements**

using *Specifications* and *Decompositions*

DECOMPOSING HIGH-PERFORMANCE KERNELS

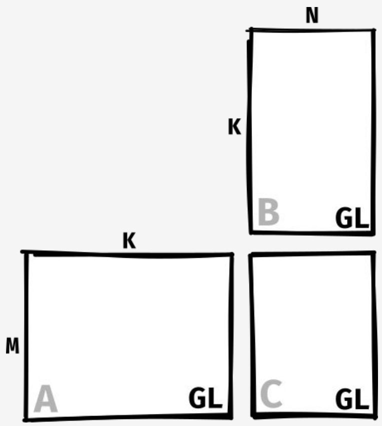
```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {
  __shared__ float ASH[128][8], BSH[8][128];
  float ARF[8][1], BRF[1][8], CRF[8][8];
  iBlock ← 128 * blockIdx.x;
   jBlock ← 128 * blockIdx.y;
   CRF ← 0;
   for (k ← 0; k < K / 8; k++) {
     GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))
     GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))
     __syncthreads();
     iWarp ← iBlock + warpIdx.x * 64;
       jWarp ← jBlock + warpIdx.y * 32;
         iThread ← iWarp + threadIdx.x * 8;
           jThread ← jWarp + threadIdx.y * 8
             for (kk ← 0; kk < 8; kk++)
               ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))
               ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))
               for (i ← 0; i < 8; i ++)
                 for (j ← 0; j < 8; j++)
                   CRF[i][j] += ARF[i][0] * BRF[0][j];
                 endfor
               endfor
             endfor
   endfor
   PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)
} // end kernel
```

implements

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

A: B: C:

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

```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {

  __shared__ float ASH[128][8], BSH[8][128];
  float ARF[8][1], BRF[1][8], CRF[8][8];

  iBlock ← 128 * blockIdx.x;
    jBlock ← 128 * blockIdx.y;

    CRF ← 0;

    for (k ← 0; k < K / 8; k++) {

      GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))

      GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))

      __syncthreads();

      iWarp ← iBlock + warpIdx.x * 64;
        jWarp ← jBlock + warpIdx.y * 32;

          iThread ← iWarp + threadIdx.x * 8;
            jThread ← jWarp + threadIdx.y * 8

            for (kk ← 0; kk < 8; kk++)

              ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))

              ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))

              for (i ← 0; i < 8; i ++)
                for (j ← 0; j < 8; j++)

                  CRF[i][j] += ARF[i][0] * BRF[0][j];

                endfor
              endfor

            endfor


    endfor

    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)

} // end kernel
```

implements

sizes of matrices | location in memory hierarchy (A: B: C:) | 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

```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {

  __shared__ float ASH[128][8], BSH[8][128];
  float ARF[8][1], BRF[1][8], CRF[8][8];

  iBlock ← 128 * blockIdx.x;
    jBlock ← 128 * blockIdx.y;

    CRF ← 0;

    for (k ← 0; k < K / 8; k++) {

      GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))

      GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))

      __syncthreads();

      iWarp ← iBlock + warpIdx.x * 64;
        jWarp ← jBlock + warpIdx.y * 32;

          iThread ← iWarp + threadIdx.x * 8;
            jThread ← jWarp + threadIdx.y * 8

              for (kk ← 0; kk < 8; kk++)

                ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))

                ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))

                for (i ← 0; i < 8; i ++)
                  for (j ← 0; j < 8; j++)

                    CRF[i][j] += ARF[i][0] * BRF[0][j];

                  endfor
                endfor

              endfor


    endfor

    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)

} // end kernel
```

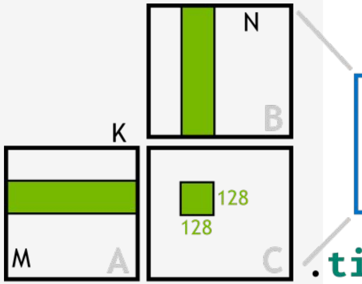
implements

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 )
```
A: B: C:

Decompositions:

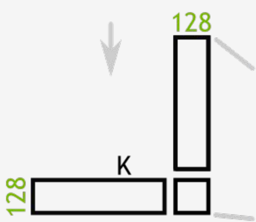
How to implement the current spec



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;
 jBlock ← 128 * blockIdx.y;      // (see Sec. 4.1)
```

New Spec

```
MatMul(Block,
  A:Matrix((128xK   ),FP32,GL,ColMajor),
  B:Matrix((K   x128),FP32,GL,ColMajor),
  C:Matrix((128x128),FP32,GL,ColMajor))
```

DECOMPOSING HIGH-PERFORMANCE KERNELS

```
__global__ void MatMul(const float A[M * K], const float B[K * N], float C[M * N]) {

  __shared__ float ASH[128][8], BSH[8][128];
  float ARF[8][1], BRF[1][8], CRF[8][8];

  iBlock ← 128 * blockIdx.x;
    jBlock ← 128 * blockIdx.y;

    CRF ← 0;

    for (k ← 0; k < K / 8; k++) {

      GlbToSh(A → ASH (128×8), start at (iBlock, jBlock))

      GlbToSh(B → BSH (8×128), start at (iBlock, jBlock))

      __syncthreads();

      iWarp ← iBlock + warpIdx.x * 64;
        jWarp ← jBlock + warpIdx.y * 32;

          iThread ← iWarp + threadIdx.x * 8;
            jThread ← jWarp + threadIdx.y * 8

              for (kk ← 0; kk < 8; kk++)

                ShToPvt(ASH → ARF (8×1), start at (iThread,jThread))

                ShToPvt(BSH → BRF (1×8), start at (iThread,jThread))

                for (i ← 0; i < 8; i ++)
                  for (j ← 0; j < 8; j++)

                    CRF[i][j] += ARF[i][0] * BRF[0][j];

                  endfor
                endfor

              endfor


    endfor

    PvtToGlb(CRF → C (128×128), start at iBlock, jBlock)

} // end kernel
```

implements

sizes of matrices | *location in memory hierarchy* | *responsible level of compute hierarchy*

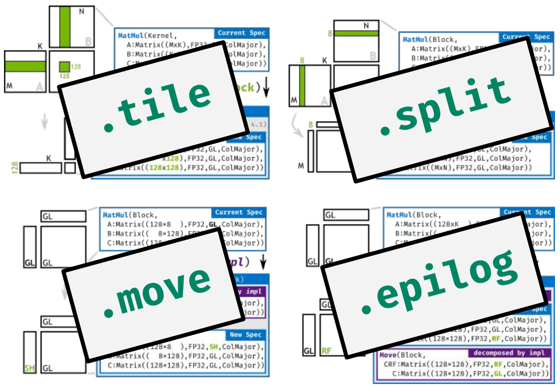
A: B: C:

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

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

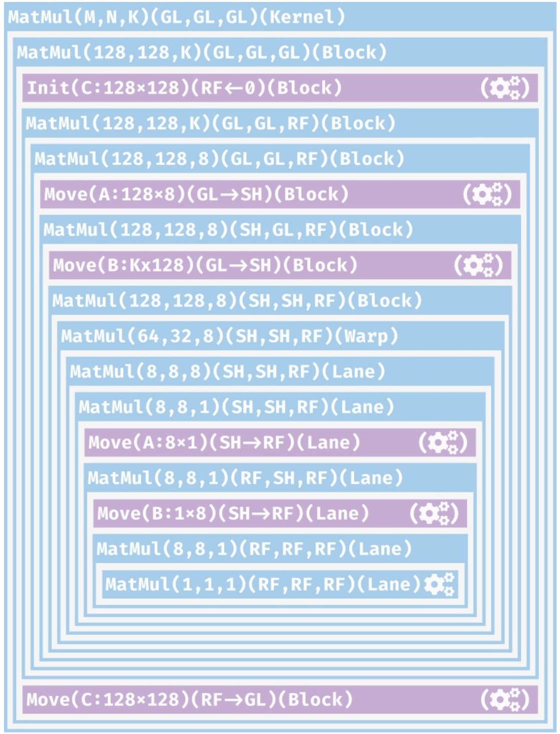
Decompositions:

How to implement the current spec



DECOMPOSING HIGH-PERFORMANCE KERNELS

Describing the implementation strategy in Fireiron:

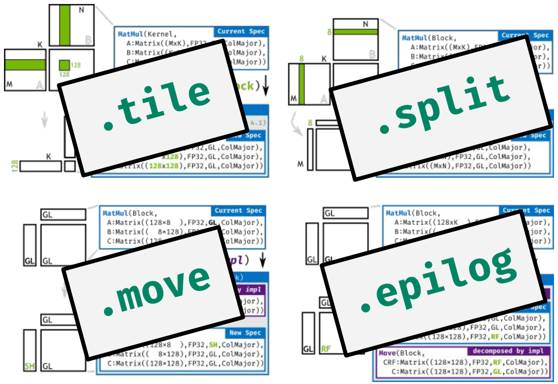



```
.tile(128,128).to(Block)
.epilog(RF,init,store)
  .apply(init)
.split(8)
.move(MatMul.A,SH,AtoSH)
  .apply(AtoSH)
.move(MatMul.B,SH,BtoSH)
  .apply(BtoSH)
.tile(64,32).to(Warp)
.tile(8,8).to(Lane)
.split(1)
.move(MatMul.A,RF,AtoRF)
  .apply(AtoRF)
.move(MatMul.B,RF,BtoRF)
  .apply(BtoRF)
.tile(1,1)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

  .apply(store)
```

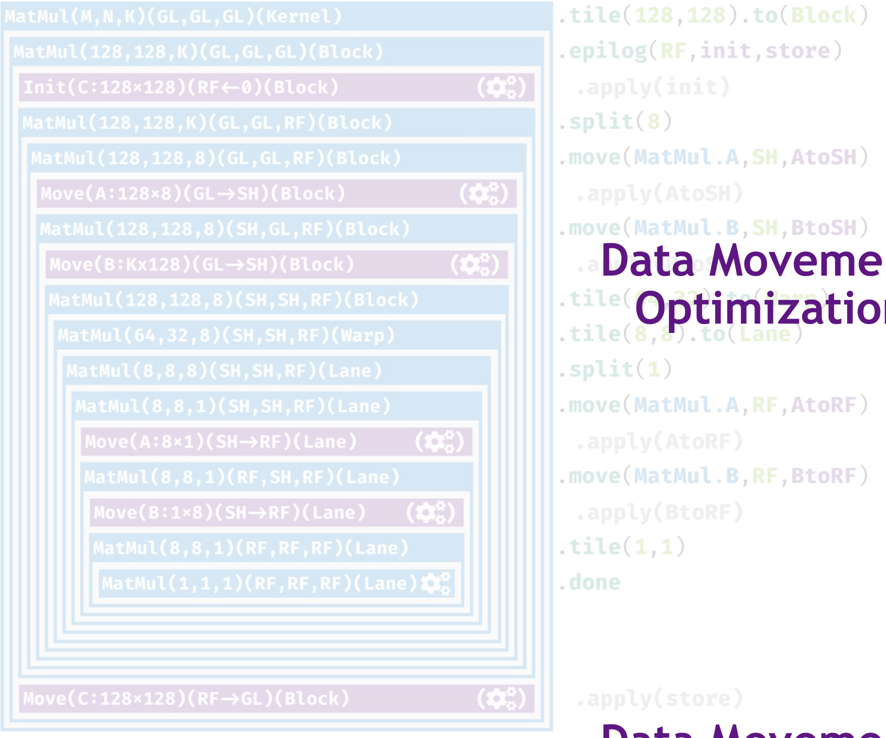
Fireiron IR created by **Fireiron Strategy**

Decompositions:

How to implement the current spec



DECOMPOSING HIGH-PERFORMANCE KERNELS



Fireiron IR created by Fireiron Strategy

Data Movement Optimizations

Data Movement Optimizations

```
val swizz: Swizzle = id => // permutation of thread-ids
  ((id >> 1) & 0x07) | (id & 0x30) | ((id & 0x01) << 3)
val storeCUDA: String = //* CUDA Epilog Micro Kernel */ /
// MATMUL-KERNEL ////////////////////////////////////////
val maxwellOptimized = MatMul(M,N,K)(GL,GL,GL)(Kernel)
///// BLOCK-LEVEL ///////////////////////////////////////
  .tile(128,128).to(Block).layout(ColMajor)
//--- epilog: store results RF => GL ------------------//
  .epilog(RF, Init// accumulate in registers
     .tile(64,32).to(Warp)
     .tile(8,  8).to(Thread) // alloc 64 reg per thread
     .tile(1,  1).unroll.done,
    Move.done(storeCUDA) /* use microkernel (18 LoC) */ )
  .split(8).sync
//--- move A to SH ------------------------------------//
  .move(MatMul.A, SH, Move(A:128x8)(GL->SH)(Block)
    .tile(128, 1).to(Warp)
    .tile(64,  1).unroll // copy in two steps
    .tile(2,   1).to(Thread).layout(ColMajor)
    .done).storageLayout(ColMajor).noSync
//--- move B to SH ------------------------------------//
  .move(MatMul.B, SH, Move(B:8x128)(GL->SH)(Block)
    .tile(8, 16).to(Warp)
    .tile(8,  4).unroll
    .tile(1,  1).to(Thread).layout(ColMajor)
    .done).storageLayout(RowMajor).pad(4)
///// WARP-LEVEL ////////////////////////////////////////
  .tile(64,32).to(Warp)
///// THREAD-LEVEL //////////////////////////////////////
  .tile((4,32),(4,16)).to(Thread)
    .layout(ColMajor).swizzle(swizz)
  .split(1).unroll
// move A and B to RF--(omit Move details for brevity)-//
  .move(MatMul.A, RF, Move.tile(4,1).unroll.done)
  .move(MatMul.B, RF, Move.tile(1,4).unroll.done)
//--- perform computation using FMA -------------------//
  .tile(1,1).unroll.done//MatMul(1,1,1)(RF,RF,RF)(Thread)
```

EVALUATION

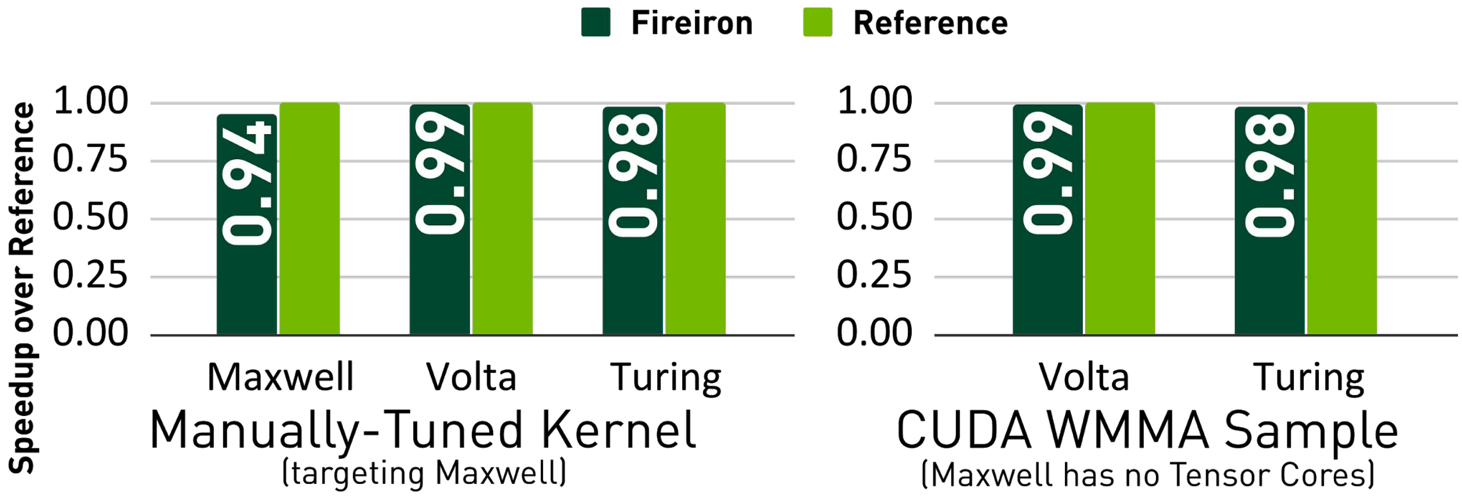
***Hypothesis A**: Code related to data movements makes up a significant fraction in high-performance kernels.*

	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)
cuBLAS		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.*

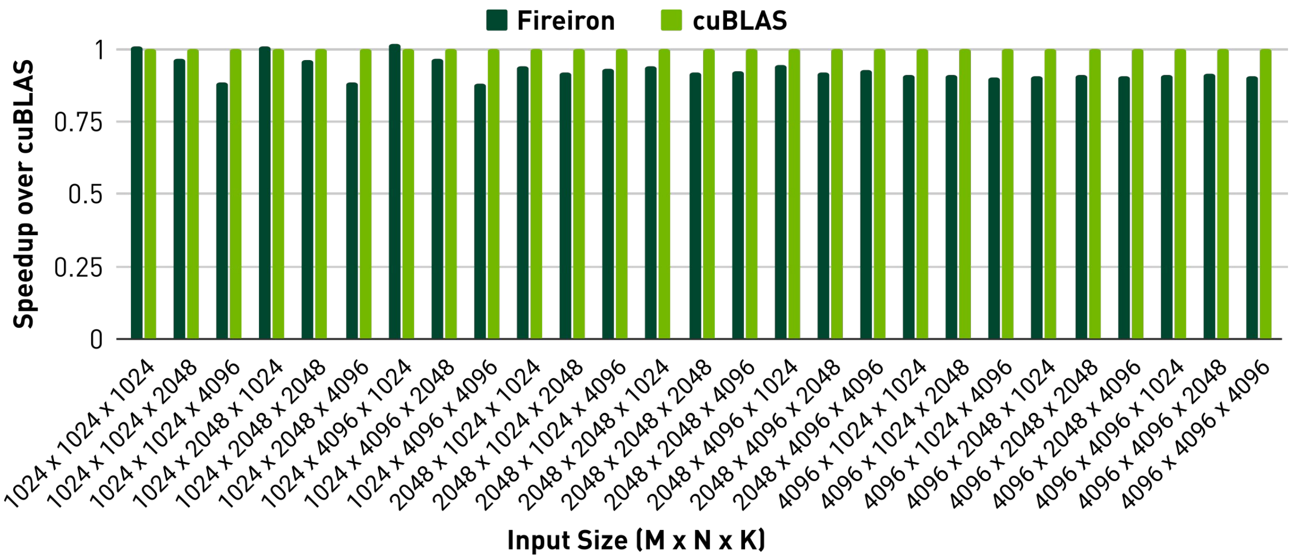


EVALUATION

***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*



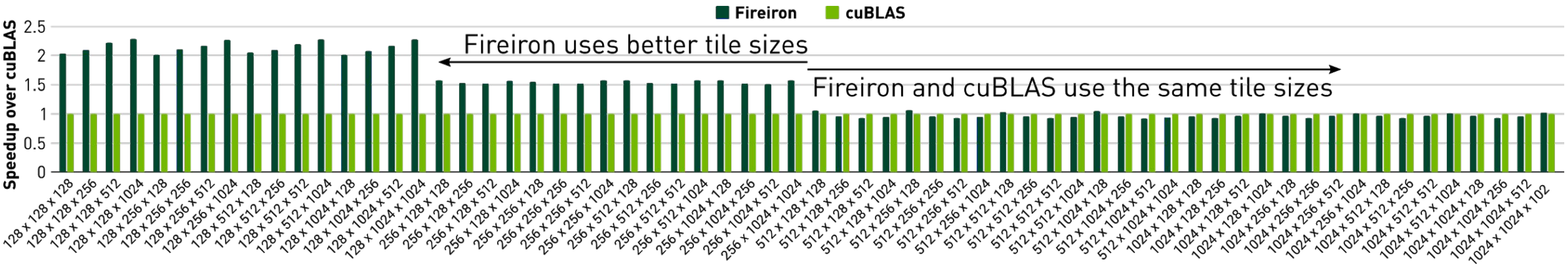
EVALUATION

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

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

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

***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*

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

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

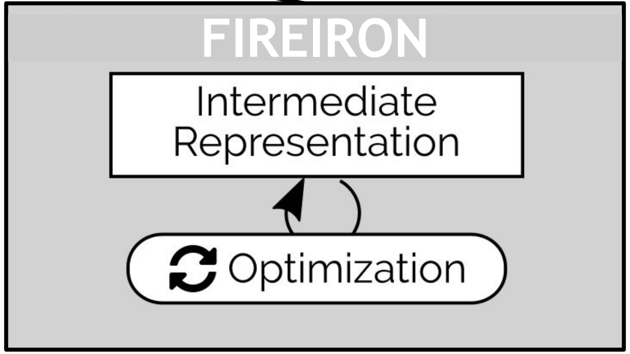
Rastislav Bodik*
University of Washington
bodik@cs.washington.edu
9 Months

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

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

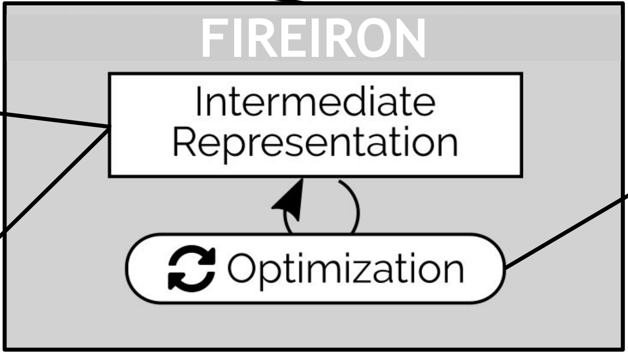
***Hypothesis D**: Experts can write Fireiron strategies that generate code which outperforms the state-of-the-art*

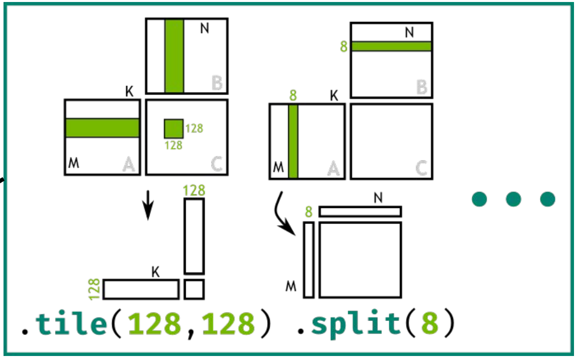
***Problem**: Time-intensive: Developing Fireiron required about nine months of full-time work*

***Problem**: Not easily reusable: IR & Optimizations specialized for matrix multiplications and GPUs*

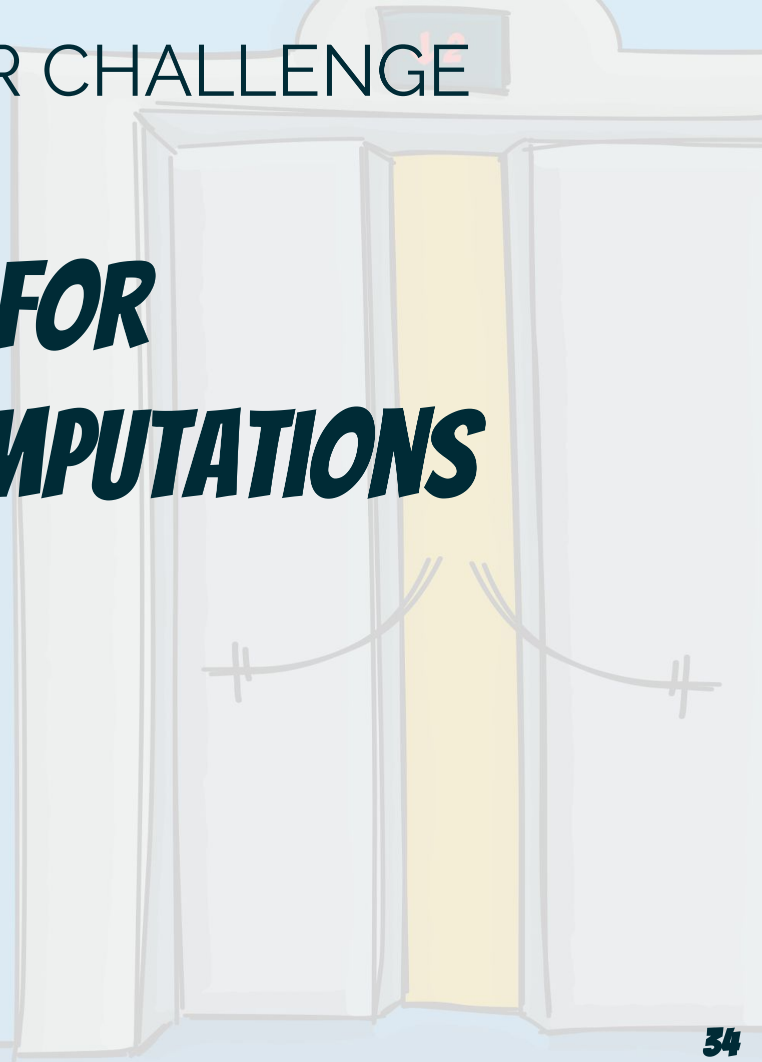
```
MatMul(ComputeHierarchy: Kernel,
  A:Matrix((M x K),FP32,GL,ColMajor),
  B:Matrix((K x N),FP32,GL,ColMajor),
  C:Matrix((M x N),FP32,GL,ColMajor))
```

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

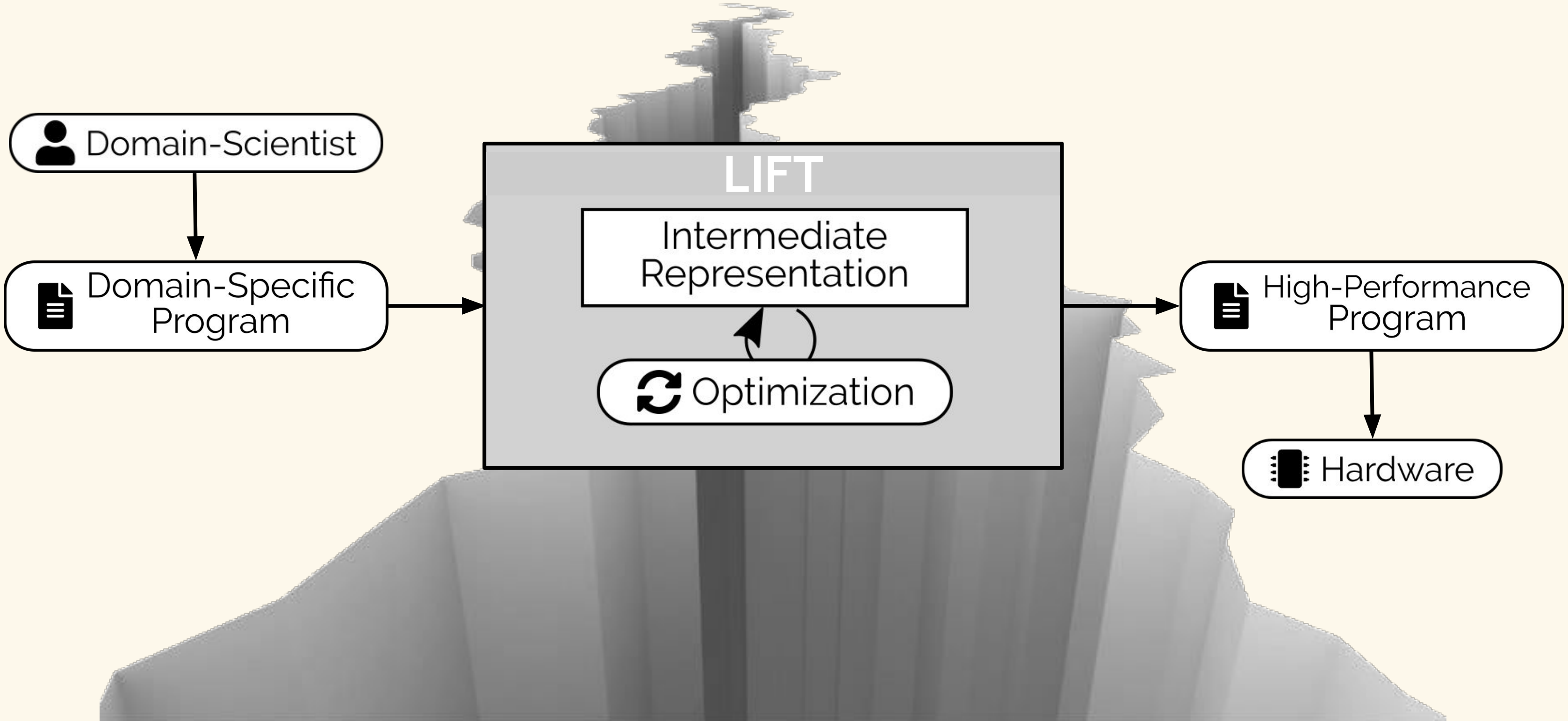




A GENERIC IR FOR DOMAIN-SPECIFIC COMPUTATIONS



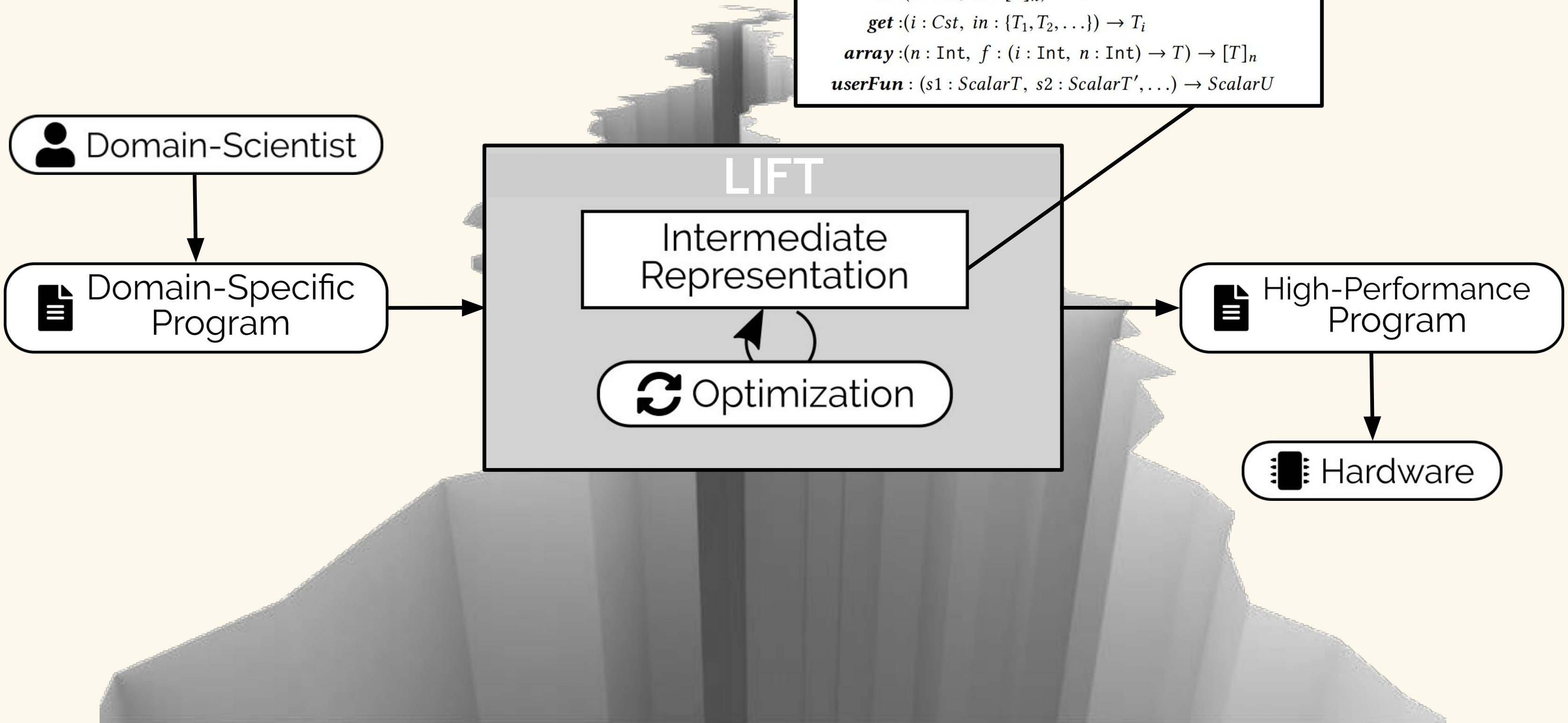
THE LIFT APPROACH (EST. 2015)



THE LIFT APPROACH (EST. 2015)

ALGORITHMIC PATTERNS

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



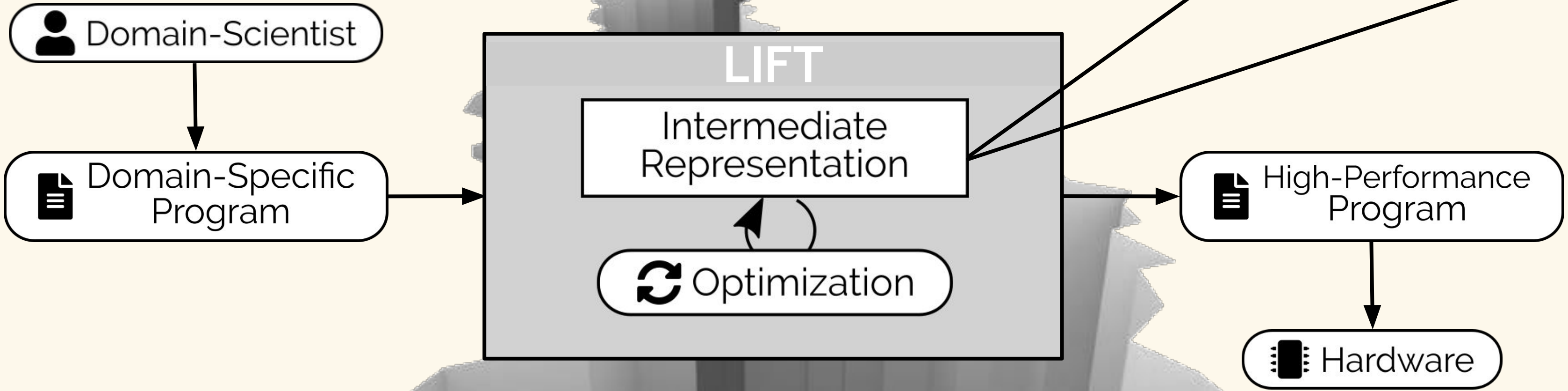
THE LIFT APPROACH (EST. 2015)

ALGORITHMIC PATTERNS

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

OPENCL-PATTERNS

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



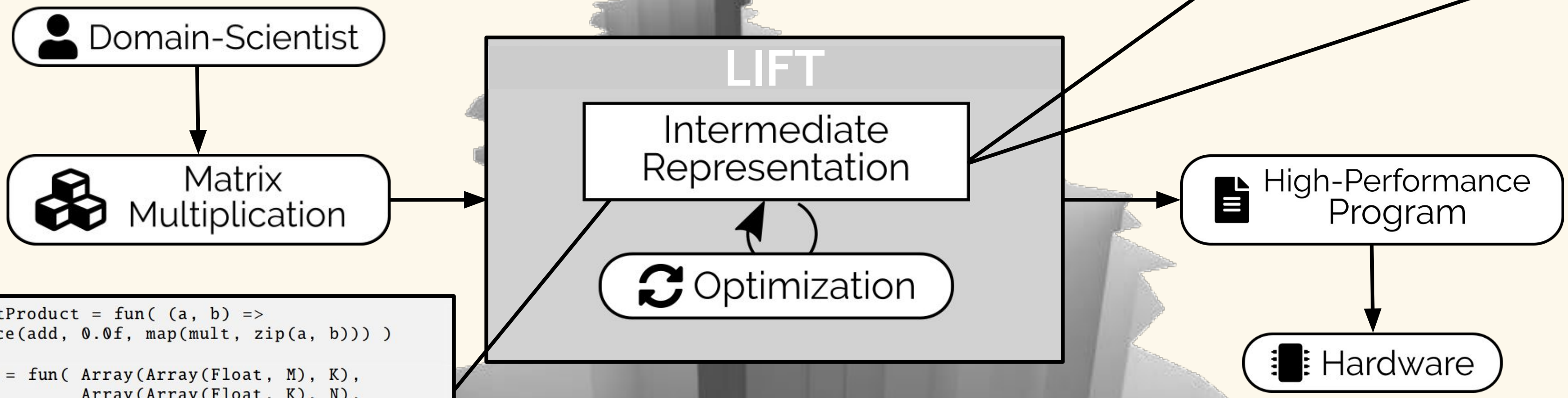
THE LIFT APPROACH (EST. 2015)

ALGORITHMIC PATTERNS

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

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

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

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

ALGORITHMIC PATTERNS

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

OPENCL-PATTERNS

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

Domain-Scientist

Matrix Multiplication

LIFT

Intermediate Representation

Optimization

High-Performance Program

Hardware

```
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

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

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

Domain-Scientist

$untile \circ map(\lambda\ rowOfTilesA\ .$
$\quad map(\lambda\ colOfTilesB\ .$
$\quad\quad toGlobal(copy2D) \circ$
$\quad\quad reduce(\lambda\ (tileAcc, (tileA, tileB))\ .$
$\quad\quad\quad map(map(+)) \circ zip(tileAcc) \circ$
$\quad\quad\quad map(\lambda\ \mathbf{aBlocks}\ .$
$\quad\quad\quad\quad map(\lambda\ bs\ .$
$\quad\quad\quad\quad\quad reduce(+, 0) \circ$
$\quad\quad\quad\quad\quad \mathbf{map(\lambda\ (aBlock, b)\ .}$
$\quad\quad\quad\quad\quad\quad \mathbf{map(\lambda\ (a,bp)\ .\ a \times bp}$
$\quad\quad\quad\quad\quad\quad , \mathbf{zip}(aBlock, \mathbf{toPrivate(id(b))}))$
$\quad\quad\quad\quad) \circ zip(\mathbf{transpose(aBlocks)}, bs)$
$\quad\quad\quad , toLocal(copy2D(tileB)))$
$\quad\quad , \mathbf{split(l,}\ toLocal(copy2D(tileA))))$
$\quad ,0, zip(rowOfTilesA, colOfTilesB))$
$\ ) \circ tile(m, k, transpose(B))$
$) \circ tile(n, k, A)$

MATRIX MULTIPLICATION

LIFT

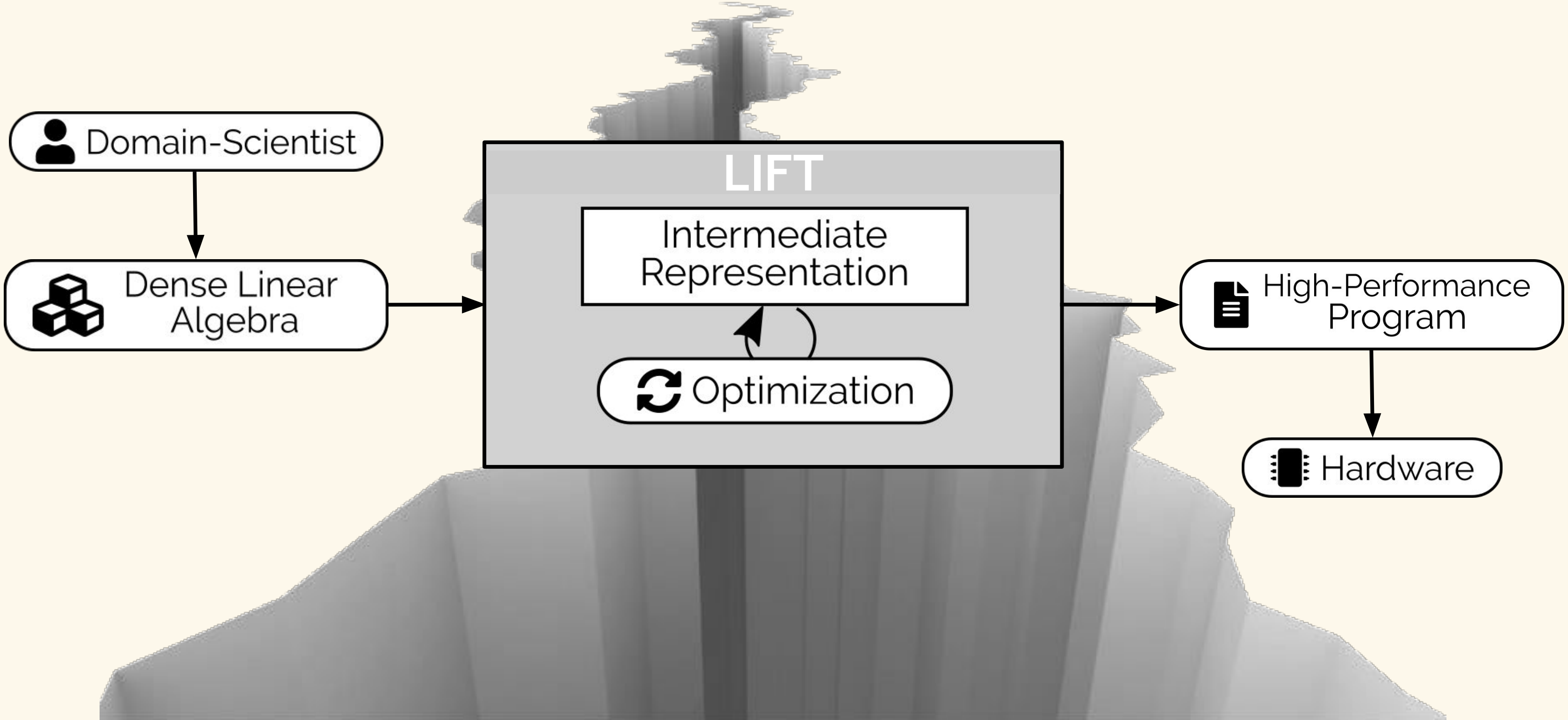
Intermediate Representation

Optimization

```
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

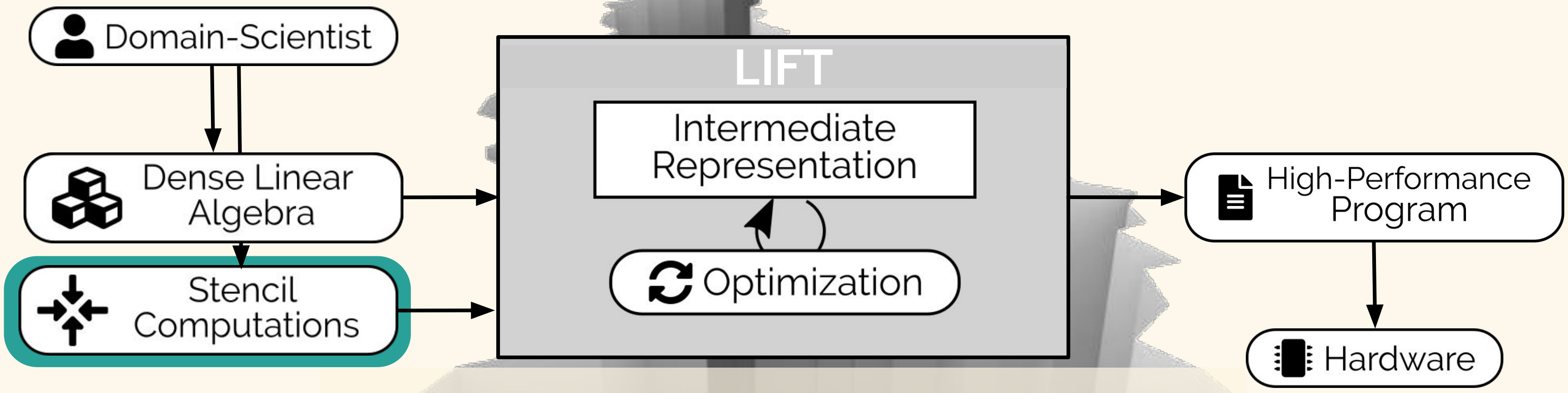
The LIFT approach
works well for dense linear algebra:



THIS WORK: DEMONSTRATING THAT THE

LIFT IR IS EASILY EXTENSIBLE, REUSABLE ACROSS DOMAINS AND PROVIDES MULTIPLE LEVELS OF ABSTRACTION

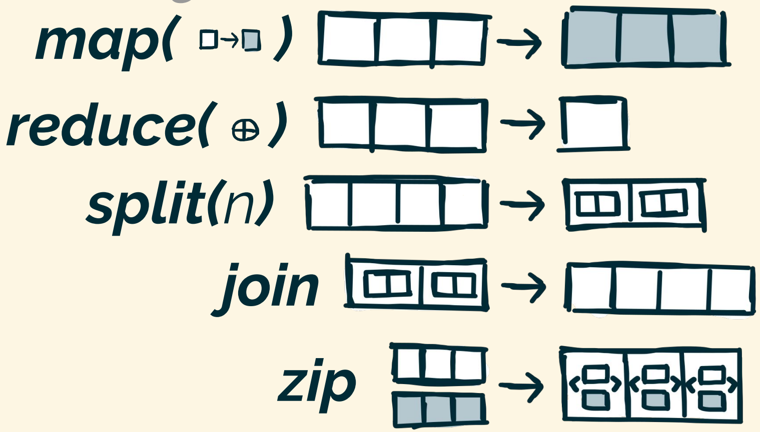
(ADDRESSING THE IR CHALLENGE)



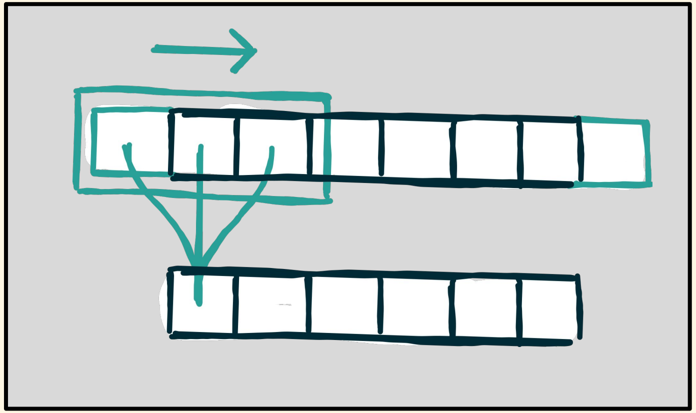
BY ADDING SUPPORT FOR STENCIL COMPUTATIONS

STENCIL COMPUTATIONS IN LIFT?

Existing Patterns:



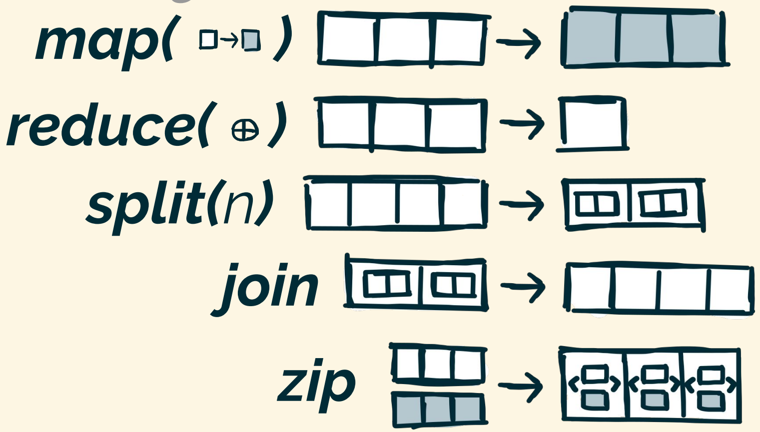
1D STENCIL COMPUTATION



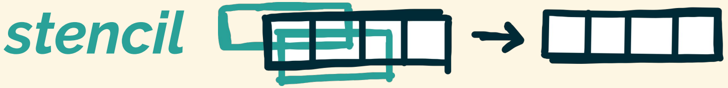
HOW TO EXPRESS THIS IN LIFT?

STENCIL COMPUTATIONS IN LIFT? NO PROBLEM . . .

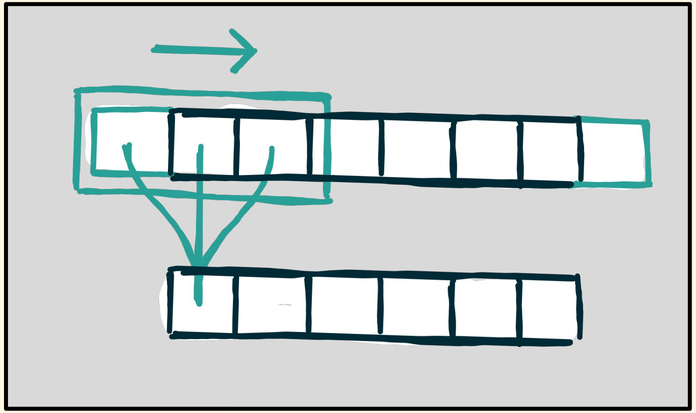
Existing Patterns:



New Pattern?



1D STENCIL COMPUTATION



⊘ **DOMAIN-SPECIFIC**
rather than generic

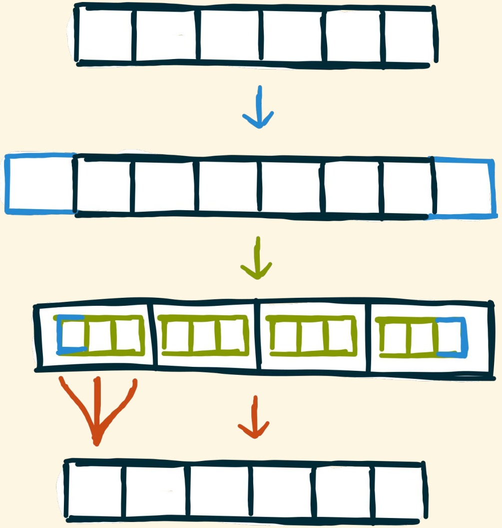
⊘ **NO REUSE**
of existing patterns and rewrites

⊘ **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 ; }
```

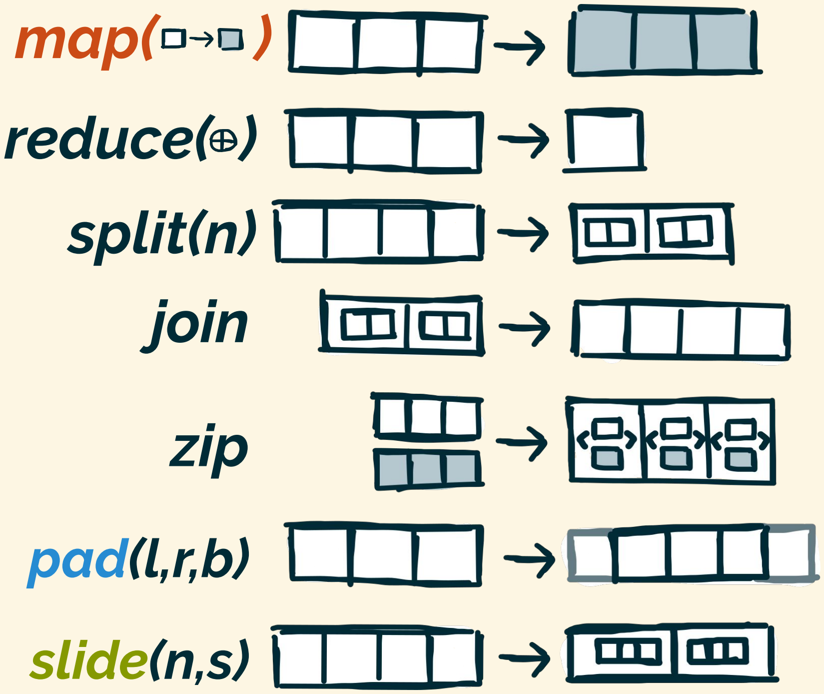


(a) access **neighborhoods** for every element

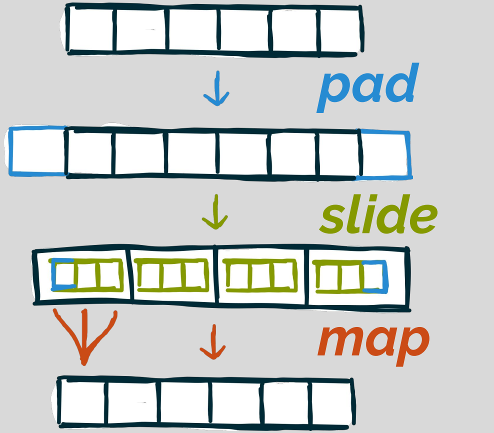
(b) specify **boundary handling**

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

Expressing Stencil Computations



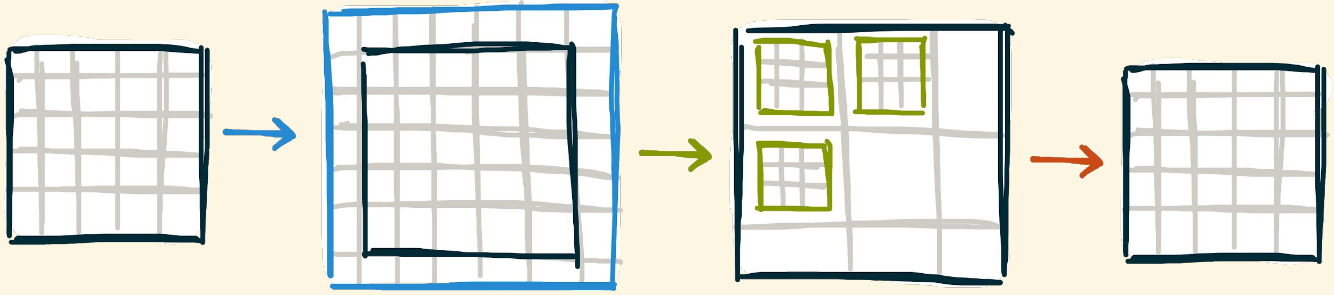
stencil1D.lift




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

Multidimensional Stencil Computations

DECOMPOSE TO RE-COMPOSE



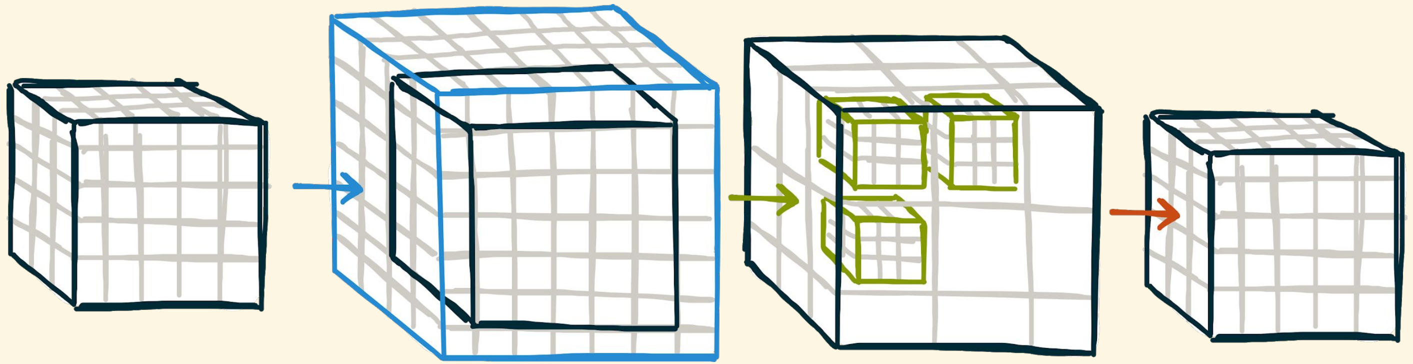
$map_2(sum,\ slide_2(3,1,\ pad_2(1,1,clamp,input)))$

$pad_2 = map(pad(1,1,clamp,pad(1,1,clamp,input)))$

✔ **Multidimensional domain-specific abstractions as compositions of one-dimensional generic patterns**

Multidimensional Stencil Computations

Decompose to Re-compose



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

$$pad_3 = map(map(pad(1,1,clamp(map(pad(1,1,clamp,pad(1,1,clamp,input)))))))$$

✔ Multidimensional domain-specific abstractions as compositions of one-dimensional generic patterns

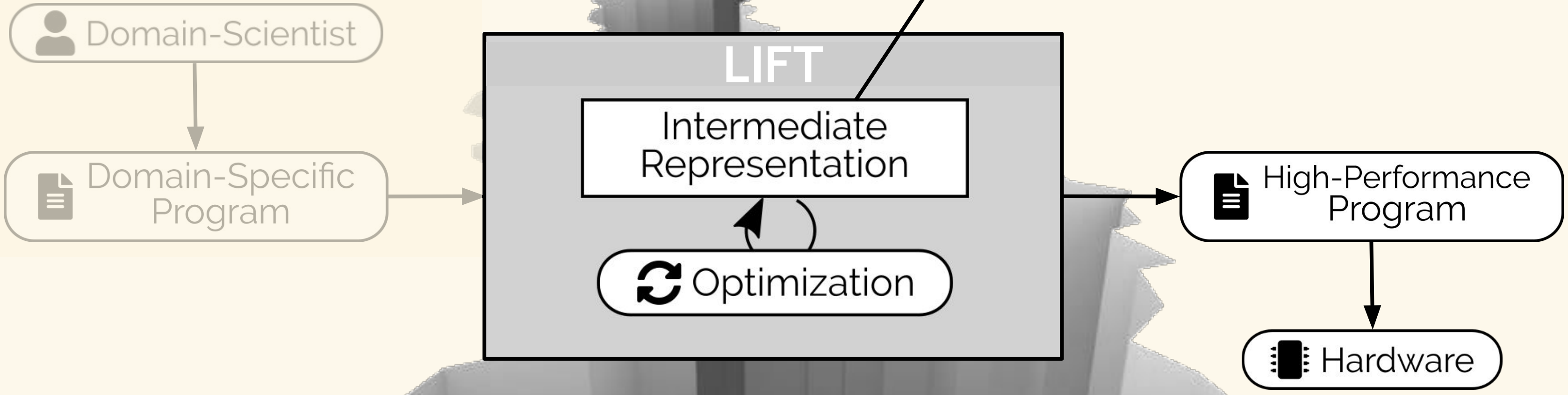
SUPPORTING STENCIL COMPUTATIONS

We added:

2 Patterns
pad, ***slide***

ALGORITHMIC PATTERNS

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



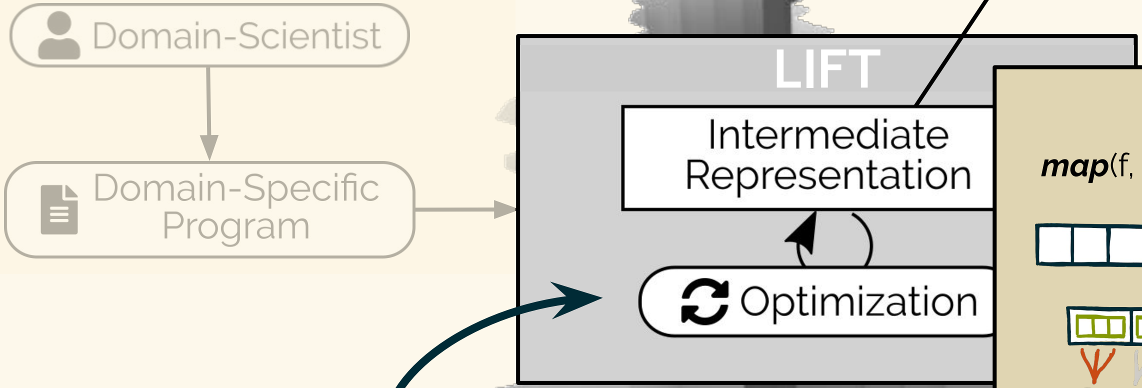
SUPPORTING STENCIL COMPUTATIONS

We added:

2 Patterns
pad, ***slide***

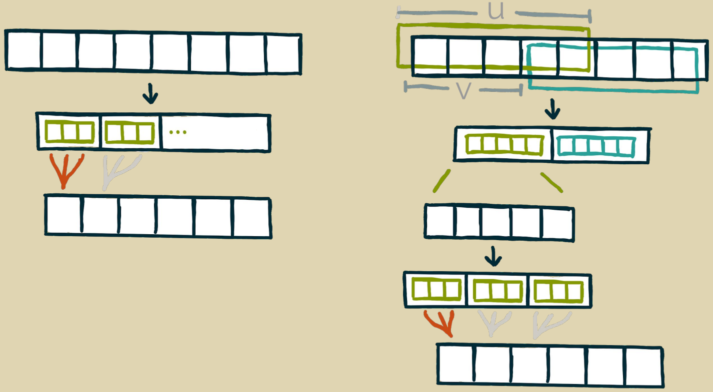
ALGORITHMIC PATTERNS

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



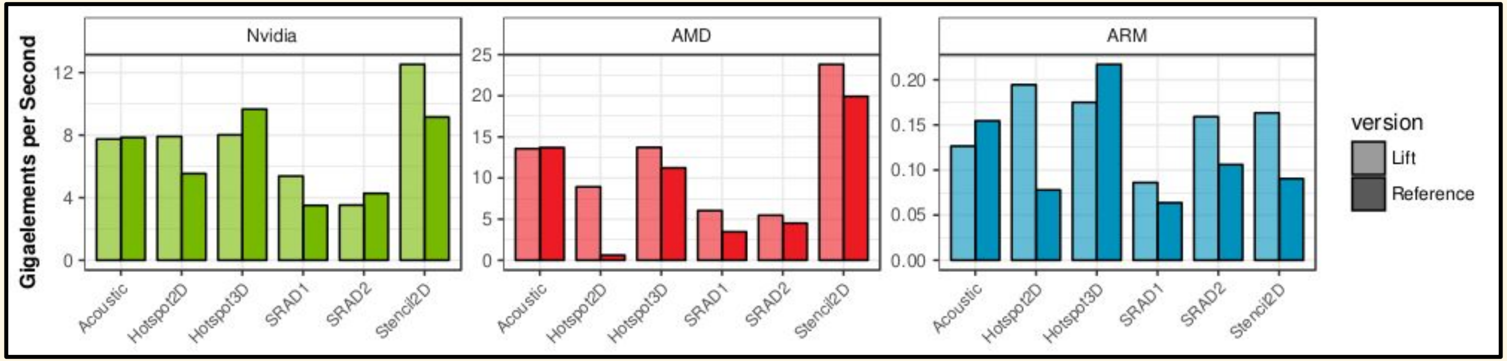
1 Rewrite Rule
overlapped tiling

map(f, ***slide***(3,1,input)) $\mapsto$ ***join***(***map***(tile ⇒ ***map***(f, ***slide***(3,1,tile)), ***slide***(u,v,input)))



COMPARISON WITH HAND-OPTIMIZED CODES

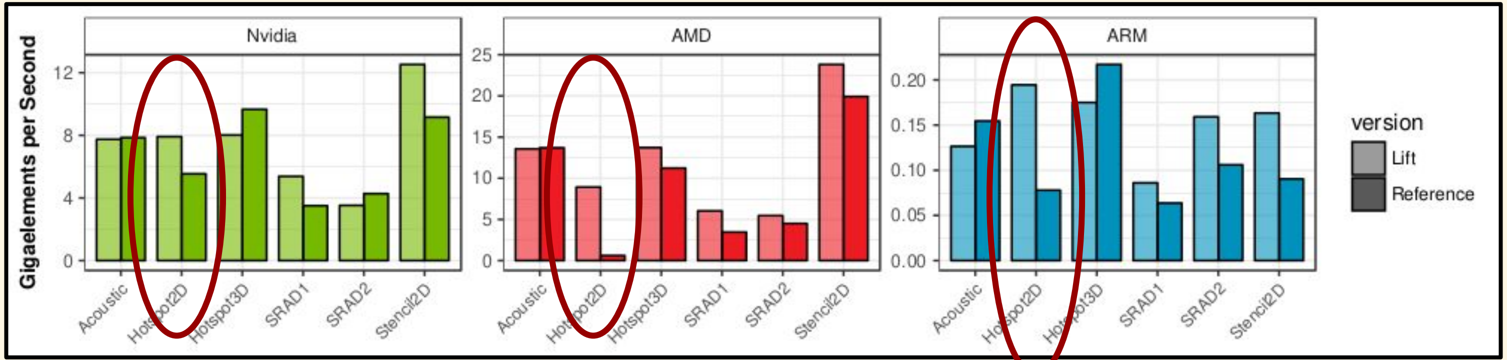
HIGHER IS BETTER



LIFT ACHIEVES PERFORMANCE COMPETITIVE TO HAND OPTIMIZED CODE

Comparison with Hand-Optimized Codes

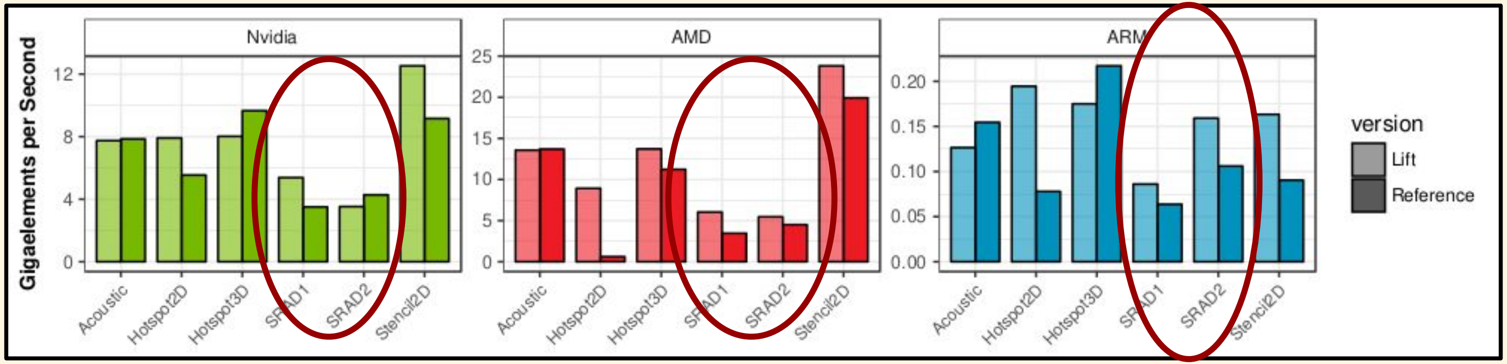
HIGHER IS BETTER



LIFT ACHIEVES PERFORMANCE COMPETITIVE TO HAND OPTIMIZED CODE

Comparison with hand-optimized codes

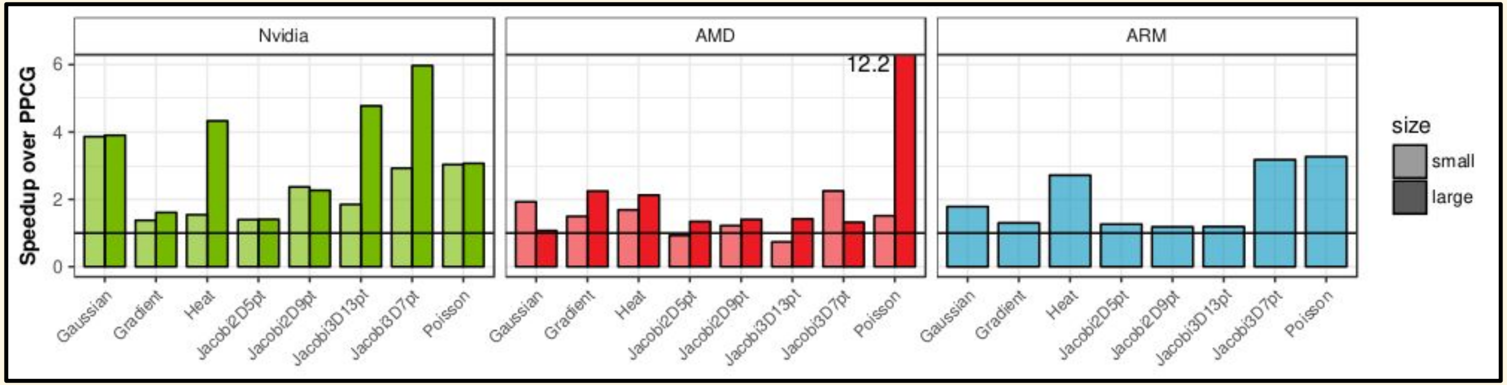
HIGHER IS BETTER



LIFT ACHIEVES PERFORMANCE COMPETITIVE TO HAND OPTIMIZED CODE

COMPARISON WITH POLYHEDRAL COMPILATION

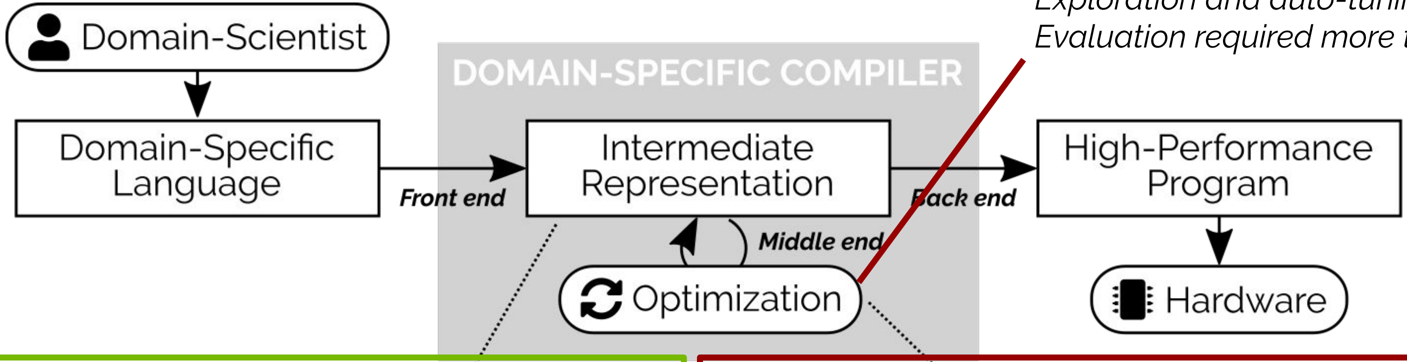
HIGHER IS BETTER



LIFT OUTPERFORMS STATE-OF-THE-ART OPTIMIZING COMPILERS

HIGH PERFORMANCE DOMAIN-SPECIFIC COMPILATION WITH DOMAIN-SPECIFIC COMPILERS

Lift provides no solution here.
Exploration and auto-tuning does not scale.
Evaluation required more than 2 days!



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?"

ADDRESSED

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?"

ELEVATE

A Language for Describing Optimization Strategies

PART III: ADDRESSING THE OPTIMIZATION CHALLENGE

SCHEDULE-BASED COMPILATION

Decoupling Computations and Optimizations

Domain-Scientist

Algorithm

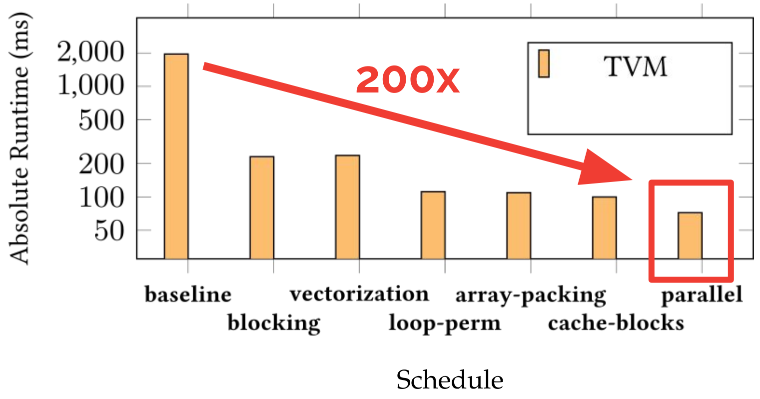
```
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute((M, N), lambda x, y:
 tvm.sum(A[x, k] * B[k, y], axis=k),
 name='C')
```

Performance Engineer

```
# "parallel schedule
s = tvm.create_schedule(C.op)
CC = s.cache_write(C, 'global')
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0], C.op.axis[1], bn, bn)

s[CC].compute_at(s[C], yo)
xc, yc = s[CC].op.axis
k, = s[CC].op.reduce_axis
ko, ki = s[CC].split(k, factor=4)
s[CC].reorder(ko, xc, ki, yc)
s[CC].unroll(ki)
s[CC].vectorize(yc)
s[C].parallel(xo)
x, y, z = s[packedB].op.axis
s[packedB].vectorize(z)
s[packedB].parallel(x)
```

Schedule

tvm



SCHEDULE-BASED COMPILATION

Decoupling Computations and Optimizations

Domain-Scientist

Algorithm

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

Performance Engineer

```
# "parallel schedule
s = tvm.create_schedule(C.op)
CC = s.cache_write(C, 'global')
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0], C.op.axis[1], bn, bn)

s[CC].compute_at(s[C], yo)
xc, yc = s[CC].op.axis
k, = s[CC].op.reduce_axis
ko, ki = s[CC].split(k, factor=4)
s[CC].reorder(ko, xc, ki, yc)
s[CC].unroll(ki)
s[CC].vectorize(yc)
s[C].parallel(xo)
x, y, z = s[packedB].op.axis
s[packedB].vectorize(z)
s[packedB].parallel(x)
```

Schedule

No clear separation
hinders reuse!

No well-defined semantics
hinders understanding!

Optimizations are built-in
no extensibility!

OUR GOALS

A Principled Way to Separate, Describe, and Apply Optimizations

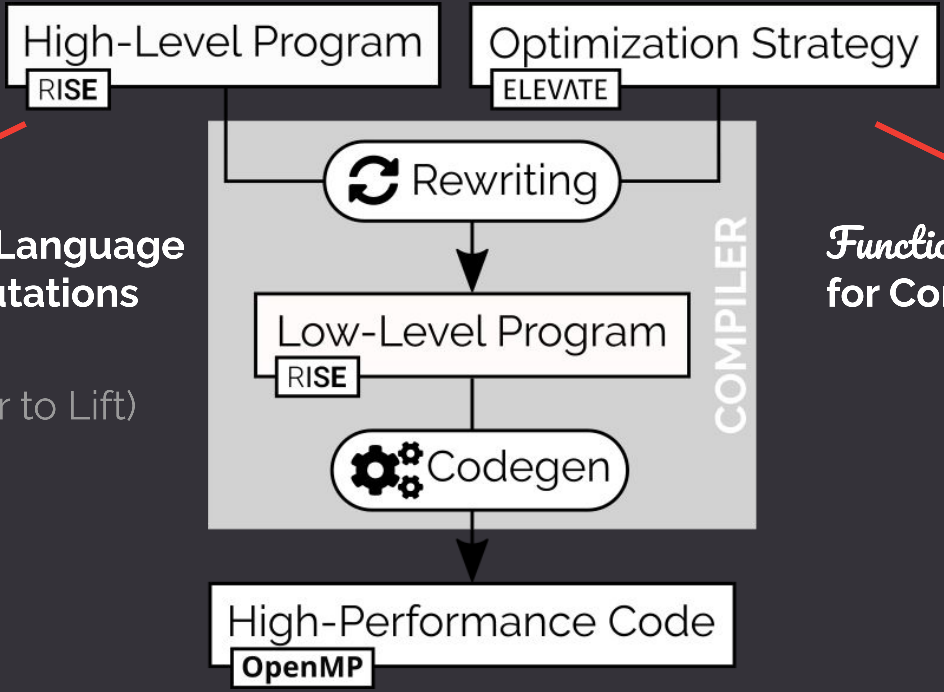
1. **Separate concerns:** Computations should not be changed for expressing optimizations
2. **Facilitate reuse:** Clear separation between computations and optimizations
3. **Enable composability:** Allow user-defined abstractions composed of simple building blocks
4. **Allow reasoning:** Well-defined 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:

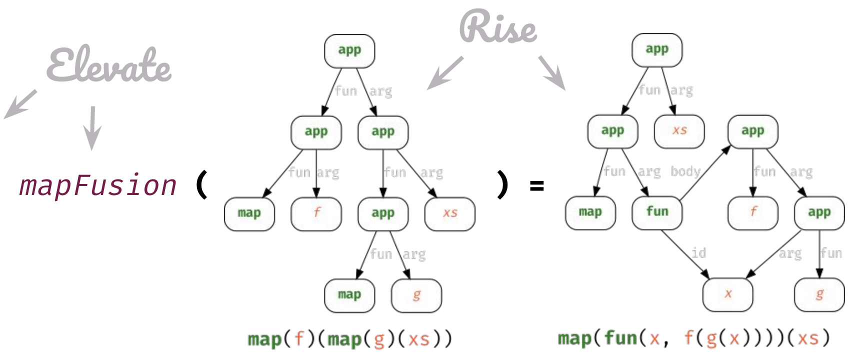
```
type Strategy[P] = P => RewriteResult[P]
```

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: *map(f)* $\circ$ *map(g)* = *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

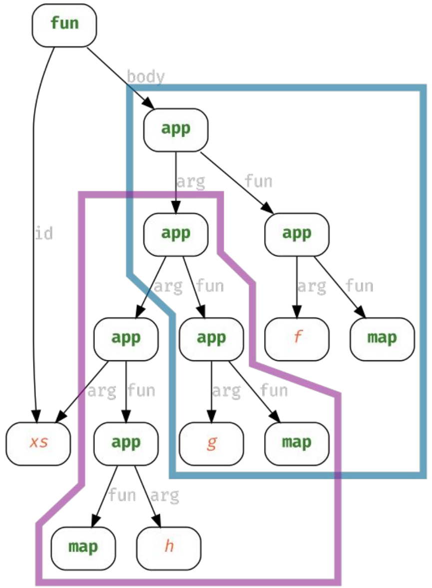
```
def try[P]: Strategy[P] => Strategy[P] =
  s => p => (s <+ id)(p)
```

Repeat

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

TRAVERSALS

Describing Precise Locations

mapFusion () ?


```
threemaps = fun(xs ⇒ map(f)(map(g)(map(h)(xs))))
```

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

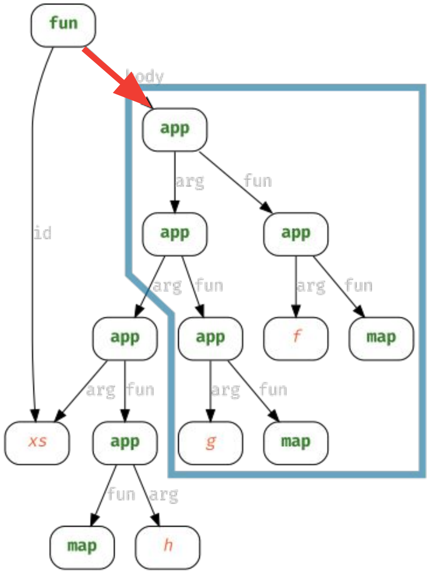
TRAVERSALS

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) )                }
```

apply s at body of function abstraction

body(mapFusion) (



)

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

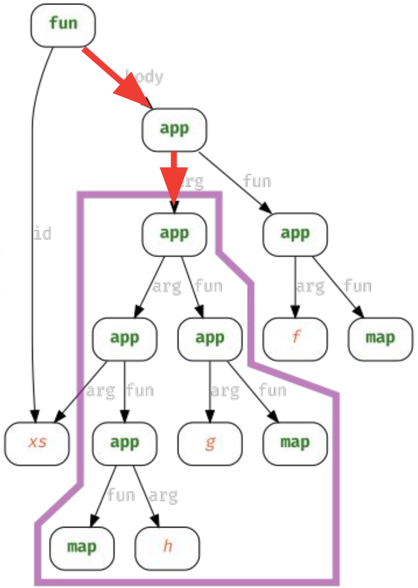
There are ***two possible locations*** for successfully applying the rule

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))))
```

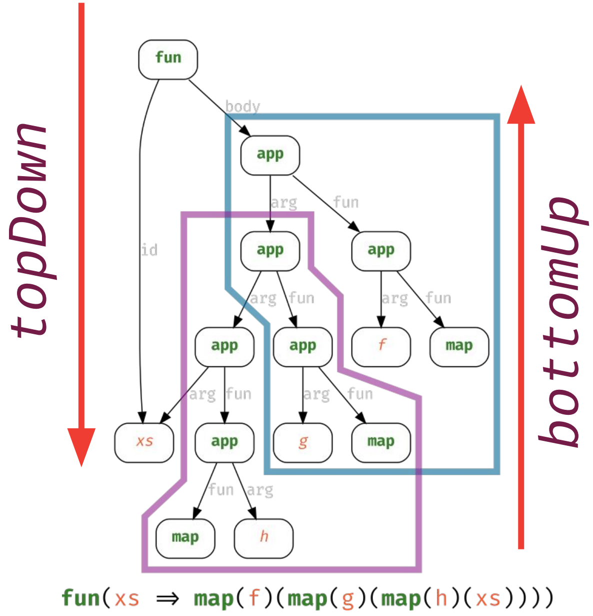
There are ***two possible locations*** for successfully applying the rule

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(topDown(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(topDown(s)) <+ s)(p)
...
```

... and a strategy for normalization

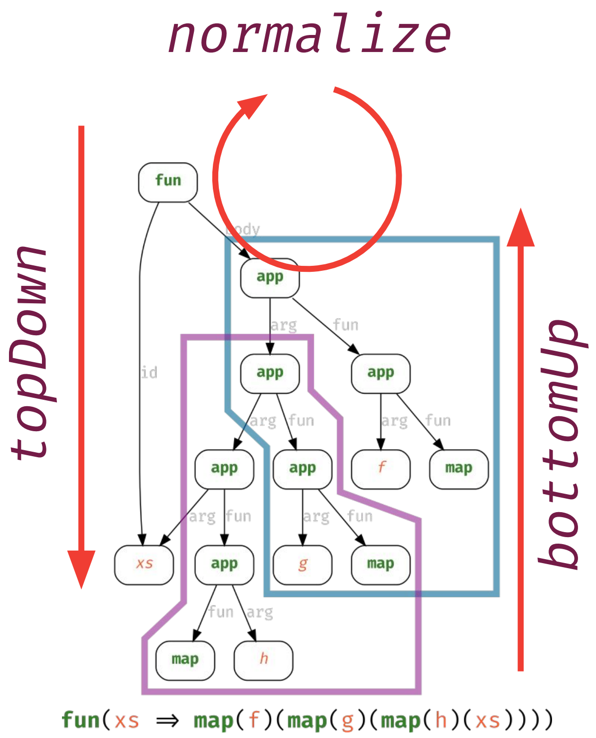
```
def normalize: Traversal[Rise] = s => p => repeat(topDown(s))(p)
```

With these, we define normal-forms like βη-normal-form

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

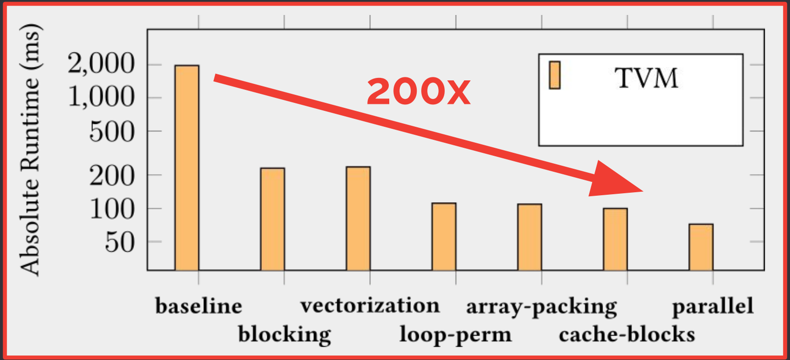
$(\lambda x. t)s \rightarrow t[x := s]$

η-reduction converts between λx.*f x* and *f* whenever *x* does not appear free in *f*.

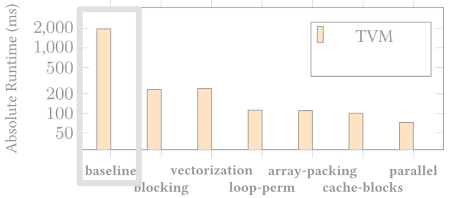


CASE STUDY

Implementing TVM's *Scheduling Language*



CASE STUDY



Optimizing Matrix Multiplication - Baseline

What to compute

RISE

tvm

```
// matrix multiplication in RISE
val dot = fun(as, fun(bs, zip(as)(bs) |>
  map(fun(ab, mult(fst(ab))(snd(ab)))) |>
  reduce(add)(0) ) )
val mm = fun(a, fun(b, a |>
  map( fun(arow, transpose(b) |>
    map( fun(bcol,
      dot(arow)(bcol) )))) ))
```

```
// baseline strategy in ELEVATE
val baseline = ( DFNF ‘;‘
  fuseReduceMap ‘@‘ topDown )
(baseline ‘;‘ lowerToC)(mm)
```

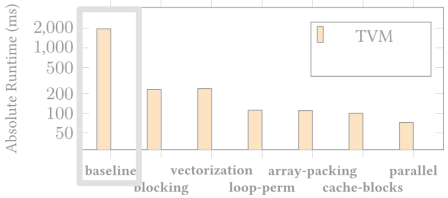
ELEVATE

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




# TVM default schedule
s = tvm.create_schedule(C.op)
```

How to optimize

CASE STUDY

Optimizing Matrix Multiplication - Baseline



clear separation

RISE

```
// matrix multiplication in RISE
val dot = fun(as, fun(bs, zip(as)(bs) |>
  map(fun(ab, mult(fst(ab))(snd(ab)))) |>
  reduce(add)(0) ) )
val mm = fun(a, fun(b, a |>
  map( fun(arow, transpose(b) |>
    map( fun(bcol,
      dot(arow)(bcol) )))) ))
```

```
// baseline strategy in ELEVATE
val baseline = ( DFNF ‘;‘
  fuseReduceMap ‘@‘ topDown )
(baseline ‘;‘ lowerToC)(mm)
```

ELEVATE

composable

explicit

tvm

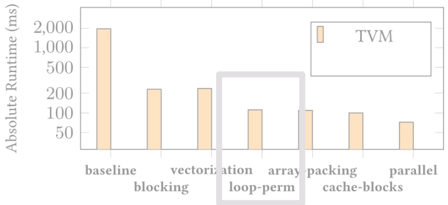
```
# Naive matrix multiplication algorithm
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
C = tvm.compute((M, N),lambda x, y:
    tvm.sum(A[x, k] * B[k, y],
    axis=k),name='C')





# TVM default schedule
s = tvm.create_schedule(C.op)
```

no separation

implicit

CASE STUDY

Optimizing Matrix Multiplication - Loop Permutation



facilitate reuse

user-defined vs. built-in

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

ELEVATE

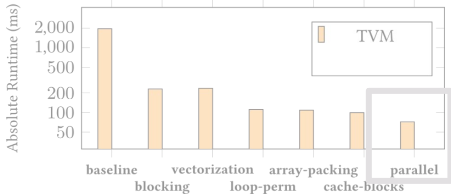
```
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0],C.op.axis[1],32,32)
k,               = s[C].op.reduce_axis
ko, ki           = s[C].split(k, factor=4)
s[C].reorder(xo, yo, ko, xi, ki, yi)
s[C].vectorize(yi)
```

tvm

no clear separation of concerns

CASE STUDY

Optimizing Matrix Multiplication - Parallel



clear separation of concerns vs. no clear separation

facilitate reuse

```
val appliedMap = isApp(isApp(isMap))
val isTransposedB = isApp(isTranspose)

val packB = storeInMemory(isTransposedB,
 permuteB ‘;;‘
 vectorize(32) ‘@‘ innermost(appliedMap) ‘;;‘
 parallel      ‘@‘ outermost(isMap)
) ‘@‘ inLambda

val par = (
  packB ‘;;‘ loopPerm ‘;;‘
  (parallel ‘@‘ outermost(isMap))
            ‘@‘ outermost(isToMem) ‘;;‘
   unroll   ‘@‘ innermost(isReduce))

(par ‘;‘ lowerToC )(mm)
```

ELEVATE

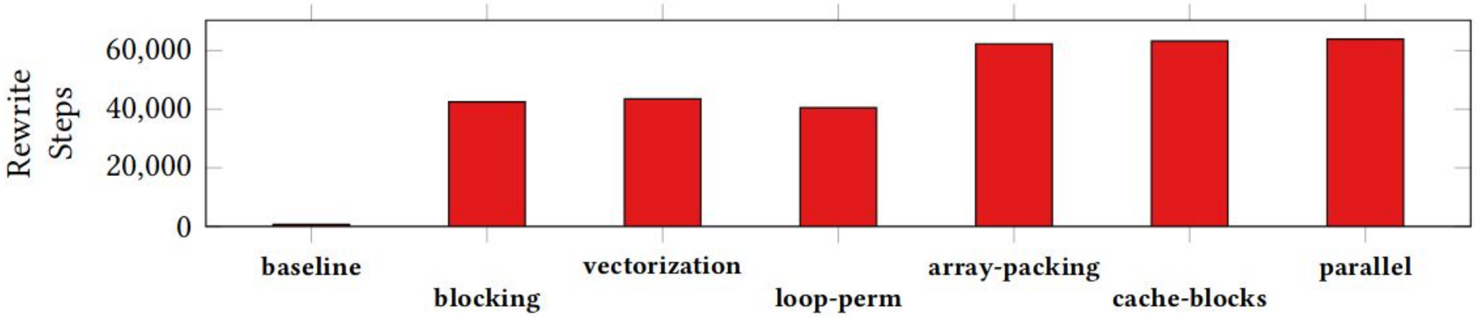
```
# Modified algorithm
bn = 32
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
pB = tvm.compute((N / bn, K, bn),
  lambda x, y, z: B[y, x * bn + z], name='pB')
C = tvm.compute((M,N), lambda x,y:
  tvm.sum(A[x,k] * pB[y//bn,k,
  tvm.indexmod(y,bn)], axis=k),name='C')
# Array packing schedule
s = tvm.create_schedule(C.op)
CC = s.cache_write(C, 'global')
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0], C.op.axis[1], bn, bn)
s[CC].compute_at(s[C], yo)
xc, yc = s[CC].op.axis
k, = s[CC].op.reduce_axis
ko, ki = s[CC].split(k, factor=4)
s[CC].reorder(ko, xc, ki, yc)
s[CC].unroll(ki)
s[CC].vectorize(yc)
s[C].parallel(xo)
x, y, z = s[pB].op.axis
s[pB].vectorize(z)
s[pB].parallel(x)
```

tvm

CASE STUDY

Counting Rewrite Steps and Measuring Performance

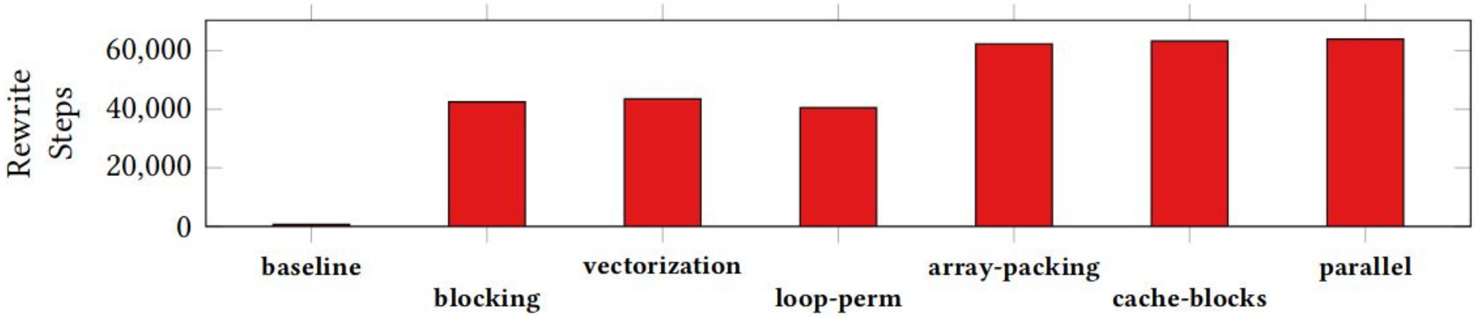
Number of successful rewrites steps



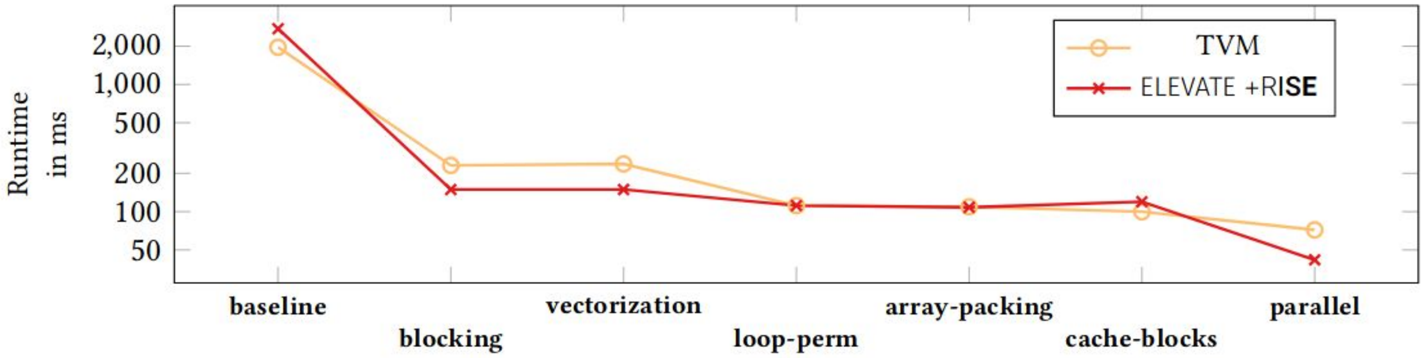
CASE STUDY

Counting Rewrite Steps and Measuring Performance

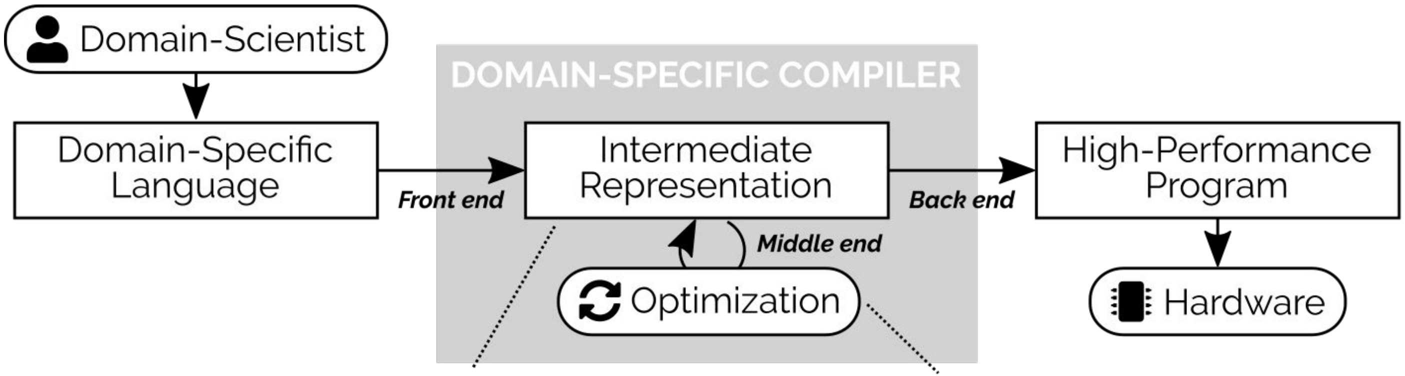
Number of successful rewrites steps



Performance of the generated code



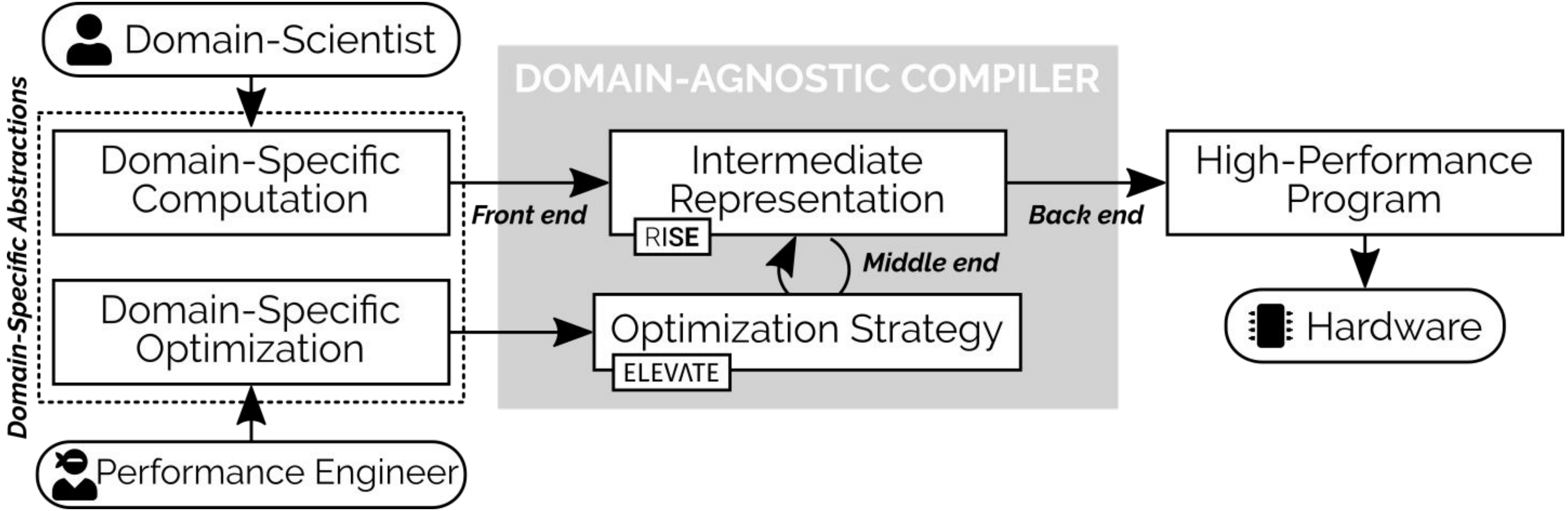
HIGH PERFORMANCE DOMAIN-SPECIFIC COMPILATION WITH DOMAIN-SPECIFIC COMPILERS



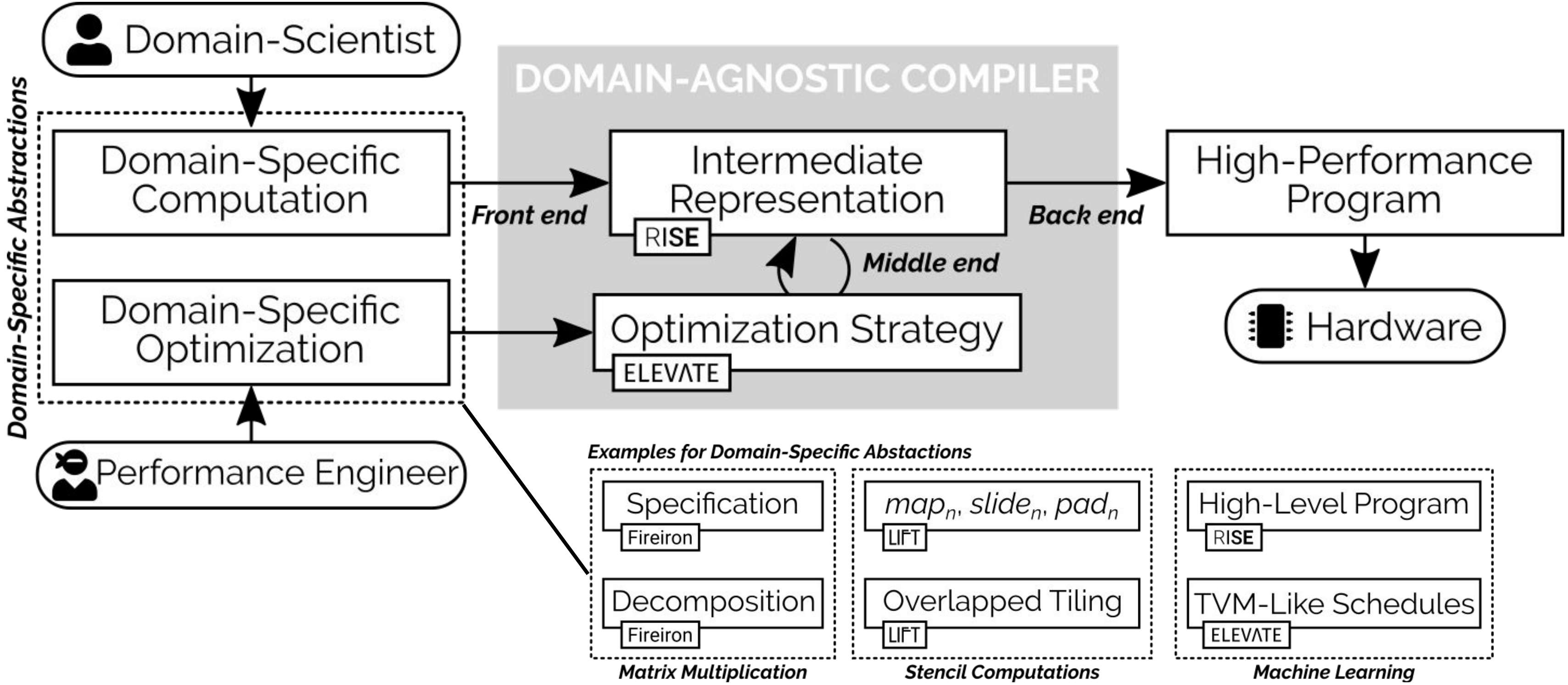
The Intermediate Representation Challenge:
"How to define an IR for high-performance domain-specific compilation that can be reused across application domains and hardware architectures while providing multiple levels of abstraction?"

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 COMPILATION WITHOUT DOMAIN-SPECIFIC COMPILERS



HIGH PERFORMANCE DOMAIN-SPECIFIC COMPILATION WITHOUT DOMAIN-SPECIFIC COMPILERS



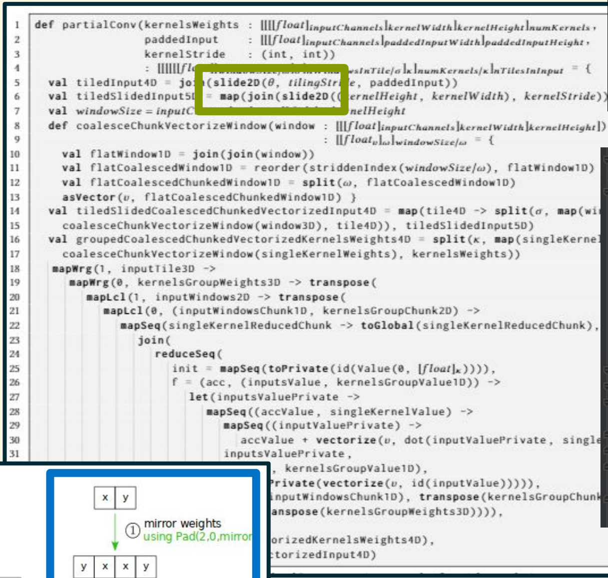
HIGH-PERFORMANCE
DOMAIN-SPECIFIC COMPILATION
WITHOUT
DOMAIN-SPECIFIC COMPILERS

thanks for your attention.

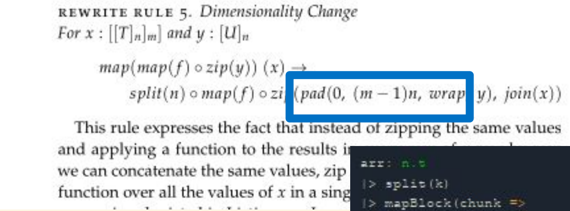
BACKUP SLIDES

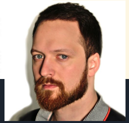
PAD AND SLIDE ARE REUSABLE!

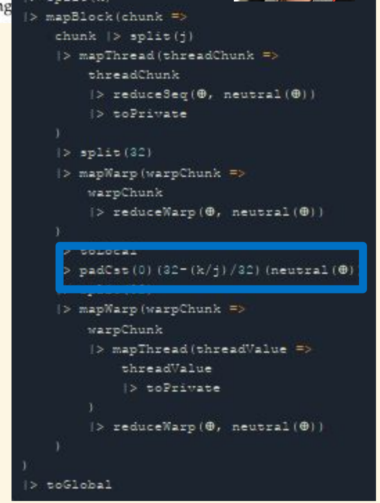
Machine Learning - Strided Convolution

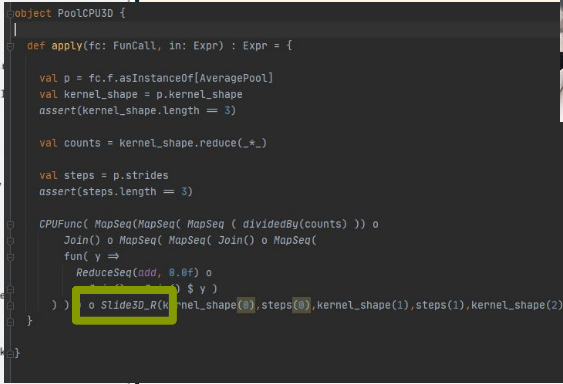


Signal Processing - Fast Fourier Transform









Machine Learning - Pooling

GPUs - Efficient Reductions

Numerical Solvers - Multigrid Methods

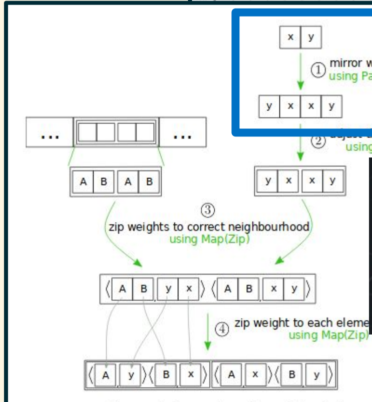


Figure 2.6: LIFT prolongation weight zipping

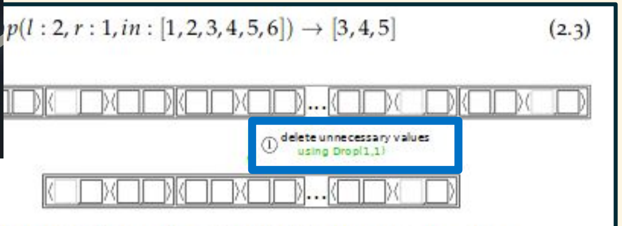


Figure 2.7: LIFT prolongation deleting unnecessary values

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
    Primtype   : String               -> TExp
    Vdec       : TExp * String * Exp  -> Vdec   -- Variable declarations
    Fdec       : TExp * String *
                 List(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)](oncetd))

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);
            try(Let(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)

2 Optimization Strategies

Our work:

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

LOG-SCALE SPEEDUP PLOT