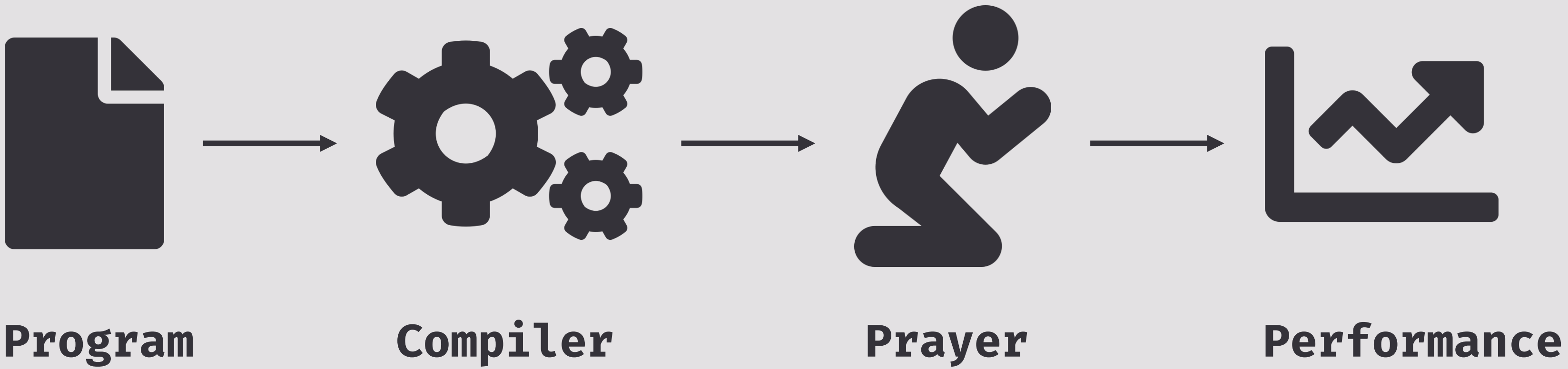
ELEVATE

A Language for Expressing Optimization Strategies

Bastian Hagedorn (University of Münster) | Michel Steuwer (University of Glasgow)

MOTIVATION

How do we optimize programs today?



MOTIVATION

How do we optimize programs today?

General Purpose Compilers

MOTIVATION

How do we optimize programs today?

General Purpose Compilers

C++

LLVM

Code Generation Options

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

Specify which optimization level to use:

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

-O1 Somewhere between -O0 and -O2.

-O2 Moderate level of optimization which enables most optimizations.

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

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

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

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

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

-O Equivalent to -O2.

-O4 and higher

Currently equivalent to -O3

Code Generation Options

MOTIVATION

How do we optimize programs today?

General Purpose Compilers

C++

LLVM

Code Generation Options

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

Specify which optimization level to use:

- -O0 Means “no optimization”: this level compiles the fastest and generates the most debuggable code.
- -O1 Somewhere between -O0 and -O2.
- -O2 Moderate level of optimization which enables most optimizations.
- -O3 Like -O2, except that it enables optimizations that take longer to perform or that may generate larger code (in an attempt to make the program run faster).
- -Ofast Enables all the optimizations from -O3 along with other aggressive optimizations that may violate strict compliance with language standards.
- -Os Like -O2 with extra optimizations to reduce code size.
- -Oz Like -Os (and thus -O2), but reduces code size further.
- -Og Like -O1. In future versions, this option might disable different optimizations in order to improve debuggability.
- -O Equivalent to -O2.
- -O4 and higher

  Currently equivalent to -O3

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

MOTIVATION

How do we optimize programs today?

General Purpose Compilers

C++

LLVM

```
-targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-info -forceattrs -inferattrs -callsite-splitting -i
psccp -called-value-propagation -globalopt -domtree -mem2reg -deadargelim -domtree -basicaa -aa -loops -lazy-branch-prob -lazy-block-fre
q -opt-remark-emitter -instcombine -simplifycfg -basiccg -globals-aa -prune-eh -inline -functionattrs -argpromotion -domtree -sroa -basi
caa -aa -memoryssa -early-cse-memssa -speculative-execution -basicaa -aa -lazy-value-info -jump-threading -correlated-propagation -simpl
ifycfg -domtree -aggressive-instcombine -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -libcall
s-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-veri
fication -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-simpli
fy -lcssa-verification -lcssa -basicaa -aa -scalar-evolution -licm -postdomtree -adce -simplifycfg -domtree -basicaa -aa -loops -lazy-br
anch-prob -lazy-block-freq -opt-remark-emitter -instcombine -barrier -elim-avail-extern -basiccg -rpo-functionattrs -globalopt -globaldc
e -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 -basic
aa -aa -loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-vectorize -loop-simplify -scalar-evolu
tion -aa -loop-accesses -loop-load-elim -basicaa -aa -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -d
omtree -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 -alignment-from-assumptions -strip-de
ad-prototypes -globaldce -constmerge -domtree -loops -branch-prob -block-freq -loop-simplify -lcssa-verification -lcssa -basicaa -aa -sc
alar-evolution -branch-prob -block-freq -loop-sink -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instsimplify -div-rem-pairs -
simplifycfg -verify
```

-O3

MOTIVATION

How do we optimize programs today?

General Purpose Compilers

C++

LLVM

```
-targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary
psccp -called-value-propagation -globalopt -domtree -mem2reg -deadargelim -domtree -
q -opt-remark-emitter -instcombine -simplifycfg -basiccg -globals-aa -prune-eh -inli
caa -aa -memoryssa -early-cse-memssa -speculative-execution -basicaa -aa -lazy-value
ifycfg -domtree -aggressive-instcombine -basicaa -aa -loops -lazy-branch-prob -lazy-
s-shrinkwrap -loops -branch-prob -block-freq -lazy-branch-prob -lazy-block-freq -opt
lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim -simplifycfg -re
fication -lcssa -basicaa -aa -scalar-evolution -loop-rotate -licm -loop-unswitch -si
prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa-verific
-loop-deletion -loop-unroll -mldst-motion -phi-values -basicaa -aa -memdep -lazy-bra
phi-values -basicaa -aa -memdep -memcpyopt -sccp -demanded-bits -bdce -basicaa -aa -
-emitter -instcombine -lazy-value-info -jump-threading -correlated-propagation -basi
fy -lcssa-verification -lcssa -basicaa -aa -scalar-evolution -licm -postdomtree -adc
anch-prob -lazy-block-freq -opt-remark-emitter -instcombine -barrier -elim-avail-ext
e -basiccg -globals-aa -float2int -domtree -loops -loop-simplify -lcssa-verification
-loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-distribu
aa -aa -loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark-
tion -aa -loop-accesses -loop-load-elim -basicaa -aa -lazy-branch-prob -lazy-block-
omtree -loops -scalar-evolution -basicaa -aa -demanded-bits -lazy-branch-prob -lazy-
remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolut
-opt-remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa -scalar-
ad-prototypes -globaldce -constmerge -domtree -loops -branch-prob -block-freq -loop-
alar-evolution -branch-prob -block-freq -loop-sink -lazy-branch-prob -lazy-block-fre
simplifycfg -verify
```

```
te-splitting -i
-lazy-block-fre
ree -sroa -basi
pagation -simpl
ombine -libcall
aa -aa -loops -
ify -lcssa-veri
s -lazy-branch-
rs -loop-idiom
-emitter -gvn -
req -opt-remark
ps -loop-simpli
-loops -lazy-br
alopt -globaldc
n -loop-rotate
volution -basic
y -scalar-evolu
-simplifycfg -d
ectorizer -opt-
azy-block-freq
tions -strip-de
basicaa -aa -so
div-rem-pairs -
```

-O3

Compiler Passes

MOTIVATION

How do we optimize programs today?

General Purpose Compilers

C++

LLVM


-targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary
psccp -called-value-propagation -globalopt -domtree -mem2reg -deadargelim -domtree
q -opt-remark-emitter -instcombine -simplifycfg -basiccg -globals-aa -prune-eh -inli
caa -aa -memoryssa -early-cse-memssa -speculative-execution -basicaa -aa -lazy-value
ifycfg -domtree -aggressive-instcombine -basicaa -aa -loops -lazy-branch-prob -lazy
s-shrinkwrap -loops -branch-prob -block-freq -lazy-branch-prob -lazy-block-freq -opt
lazy-branch-prob -lazy-block-freq -opt-remark-emitter -tailcallelim -simplifycfg -re
fication -lcssa -basicaa -aa -scalar-evolution -loop-rotate -licm -loop-unswitch -si
prob -lazy-block-freq -opt-remark-emitter -instcombine -loop-simplify -lcssa-verifi
-loop-deletion -loop-unroll -mldst-motion -phi-values -basicaa -aa -memdep -lazy-bra
phi-values -basicaa -aa -memdep -memcpyopt -sccp -demanded-bits -bdce -basicaa -aa
-emitter -instcombine -lazy-value-info -jump-threading -correlated-propagation -bas
fy -lcssa-verification -lcssa -basicaa -aa -scalar-evolution -licm -postdomtree -ad
anch-prob -lazy-block-freq -opt-remark-emitter -instcombine -barrier -elim-avail-ext
e -basiccg -globals-aa -float2int -domtree -loops -loop-simplify -lcssa-verification
-loop-accesses -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -loop-distrib
aa -aa -loop-accesses -demanded-bits -lazy-branch-prob -lazy-block-freq -opt-remark
tion -aa -loop-accesses -loop-load-elim -basicaa -aa -lazy-branch-prob -lazy-block
omtree -loops -scalar-evolution -basicaa -aa -demanded-bits -lazy-branch-prob -lazy
remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa -scalar-evolu
-opt-remark-emitter -instcombine -loop-simplify -lcssa-verification -lcssa -scalar
ad-prototypes -globaldce -constmerge -domtree -loops -branch-prob -block-freq -loop
alar-evolution -branch-prob -block-freq -loop-sink -lazy-branch-prob -lazy-block-fr
simplifycfg -verify

-O3

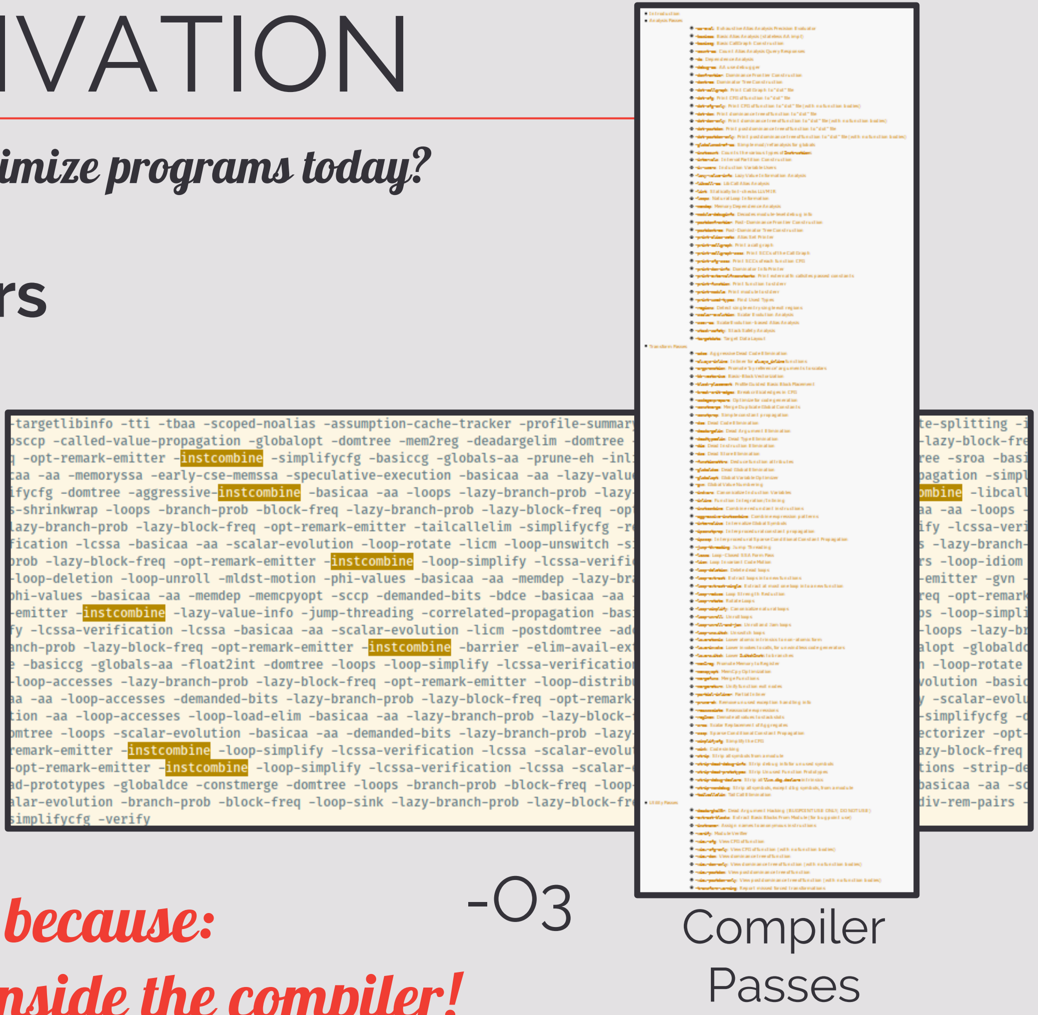
Compiler Passes

MOTIVATION

How do we optimize programs today?

General Purpose Compilers

C++

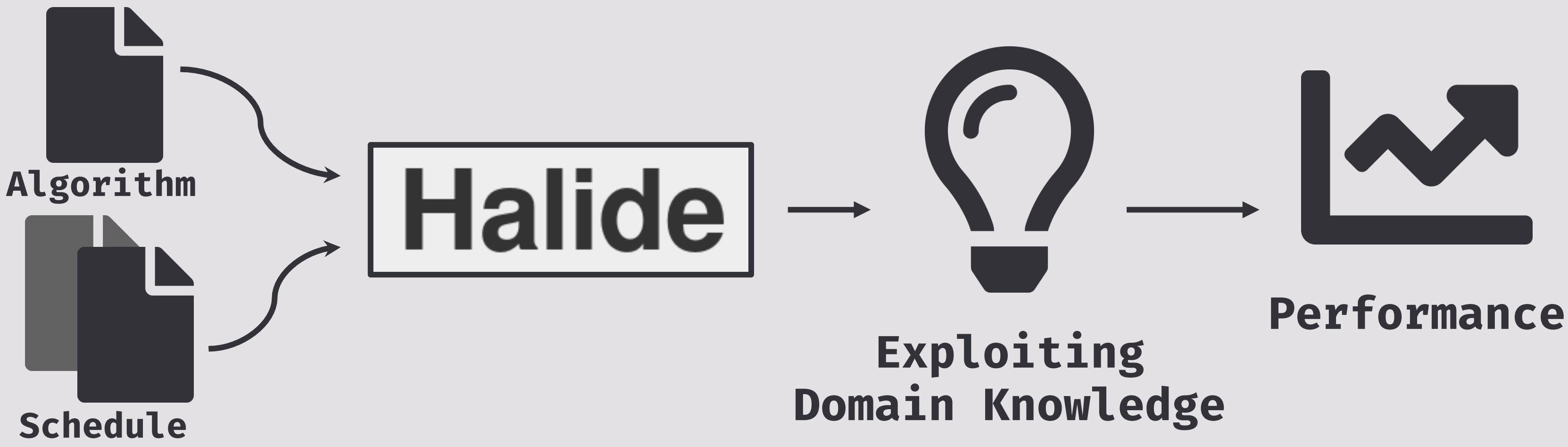


Controlling optimisations is hard, because:
We have no clue what's going on inside the compiler!

MOTIVATION

How do we optimize programs today?

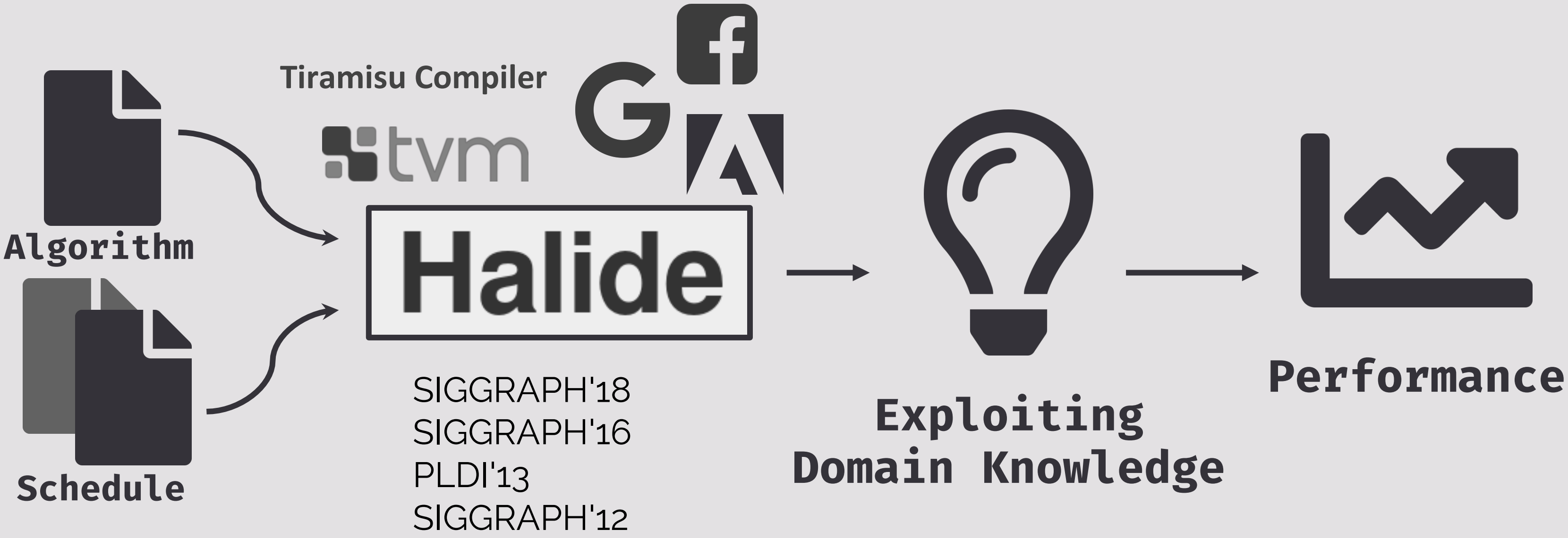
Domain-Specific Compilers



MOTIVATION

How do we optimize programs today?

Domain-Specific Compilers



MOTIVATION

How do we optimize programs today?

Domain-Specific Compilers

Matrix Multiplication

Algorithm

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

Schedule

MOTIVATION

How do we optimize programs today?

Domain-Specific Compilers

Matrix Multiplication

Algorithm

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

Schedule

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

MOTIVATION

How do we optimize programs today?

Domain-Specific Compilers

Matrix Multiplication

Algorithm

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

Schedule

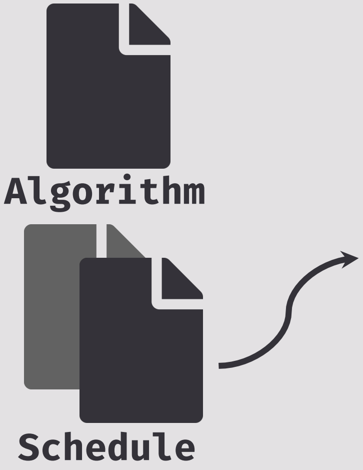
```
const int warp_size = 32;
const int vec_size = 2;
const int x_tile = 3;
const int y_tile = 4;
const int y_unroll = 8;
const int r_unroll = 1;
Var xi, yi, xio, xii, yii, xo, yo, x_pair, xiio, ty;
RVar rxo, rxi;
out.bound(x, 0, size)
   .bound(y, 0, size)
   .tile(x, y, xi, yi, x_tile * vec_size *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

Schedules are much harder to write than algorithms

MOTIVATION

How do we optimize programs today?

Domain-Specific Compilers




```
...
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);
...
```

MOTIVATION

How do we optimize programs today?

Domain-Specific Compilers

fixed set of optimizations ⇒ lack of extensibility

Algorithm

Schedule

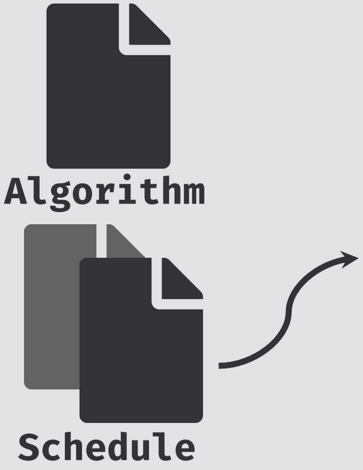
```
...
out.bound(x, 0, size)
.bound(y, 0, size)
.tile(x, y, xi, yi, x_tile * vec_size * warp_size, y_tile * y_unroll)
.split(yi, ty, yi, y_unroll)
.vectorize(xi, vec_size)
.split(xi, xio, xii, warp_size)
.reorder(xio, yi, xii, ty, x, y)
.unroll(xio)
.unroll(yi)
.gpu_blocks(x, y)
.gpu_threads(ty)
.gpu_lanes(xii);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

MOTIVATION

How do we optimize programs today?

Domain-Specific Compilers

fixed set of optimizations ⇒ lack of extensibility




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

what happens if the order of these is swapped ?

⇒ unclear semantics

⇒ unclear how to automatically generate schedules

MOTIVATION

How do we optimize programs today?

Not always a clear separation of algorithm and schedule

Domain-Specific Compilers

Algorithm

Schedule

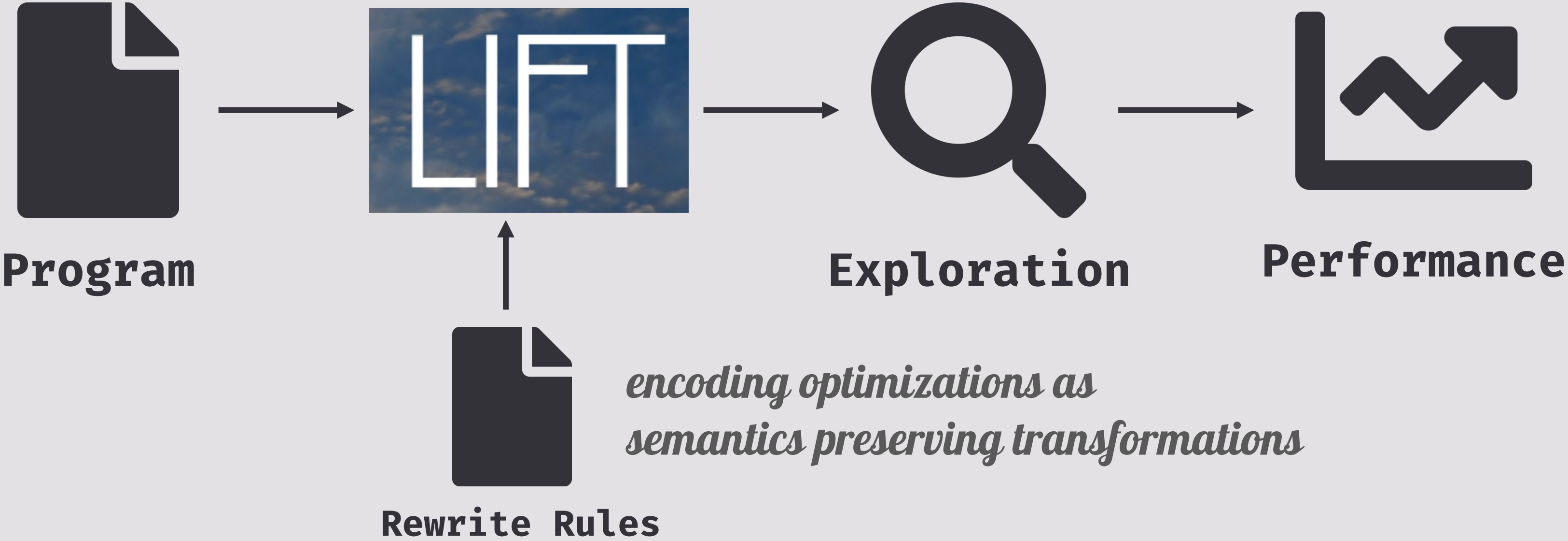
```
Func prod("prod");
prod(j, i) = A_(j, i) * x_(j);

RDom k(0, sum_size_vecs, "k");
Func accum_vecs("accum_vecs");
accum_vecs(j, i) += prod(k * vec_size + j, i);

Func accum_vecs_transpose("accum_vecs_transpose");
accum_vecs_transpose(i, j) = accum_vecs(j, i);

RDom lanes(0, vec_size);
Func sum_lanes("sum_lanes");
sum_lanes(i) += accum_vecs_transpose(i, lanes);

RDom tail(sum_size_vecs * vec_size, sum_size - sum_size_vecs * vec_size);
Func sum_tail("sum_tail");
sum_tail(i) = sum_lanes(i);
sum_tail(i) += prod(tail, i);

Func Ax("Ax");
Ax(i) = sum_tail(i);
result(i) = b_ * y_(i) + a_ * Ax(i);
```

Matrix Vector Multiplication

MOTIVATION

How do we optimize programs today?

Functional Domain-Specific Compilers



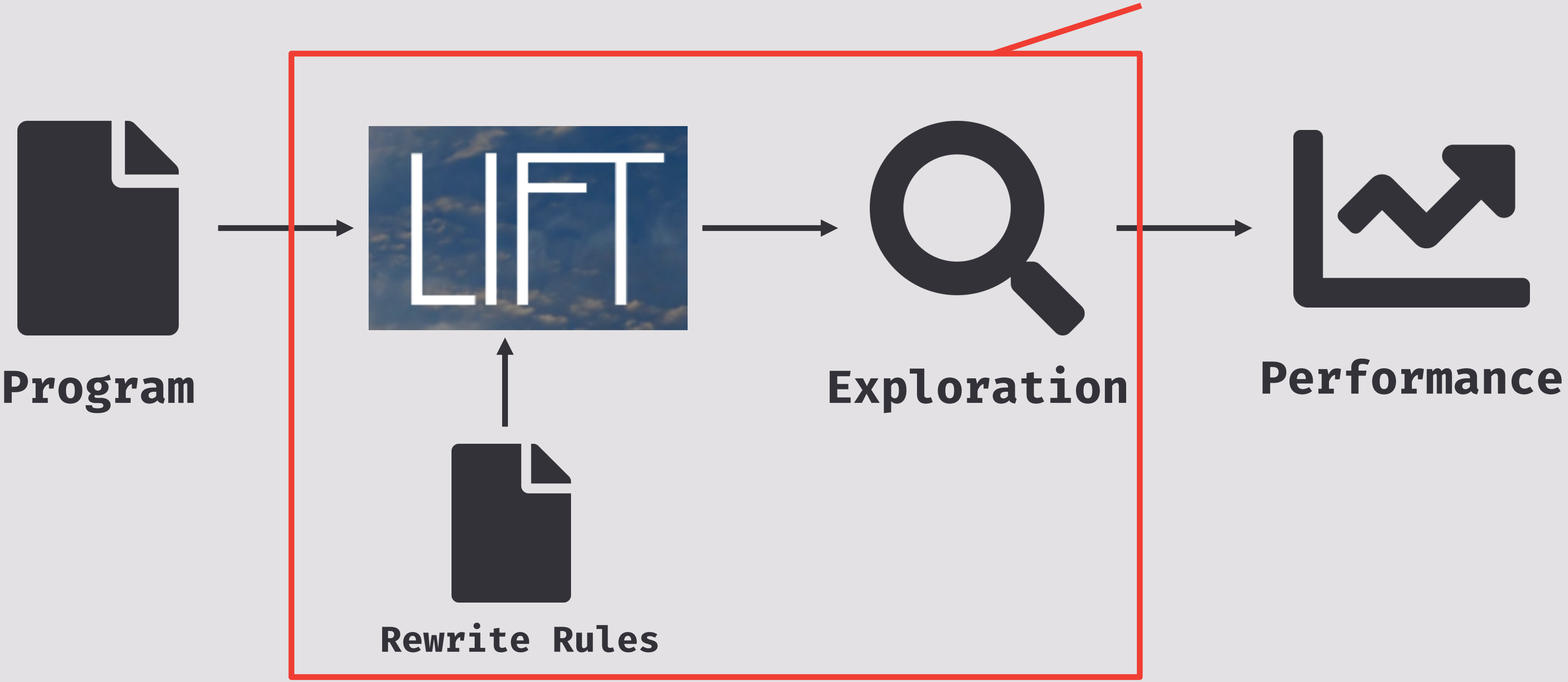
encoding optimizations as
semantics preserving transformations

MOTIVATION

How do we optimize programs today?

Functional Domain-Specific Compilers

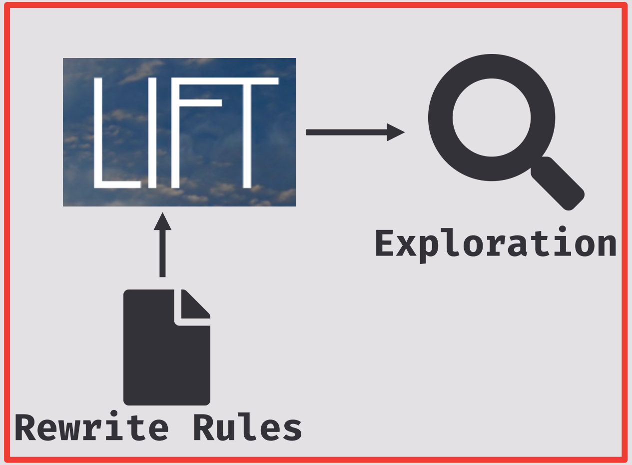
How does this actually work?



MOTIVATION

How do we optimize programs today?

Functional Domain-Specific Compilers

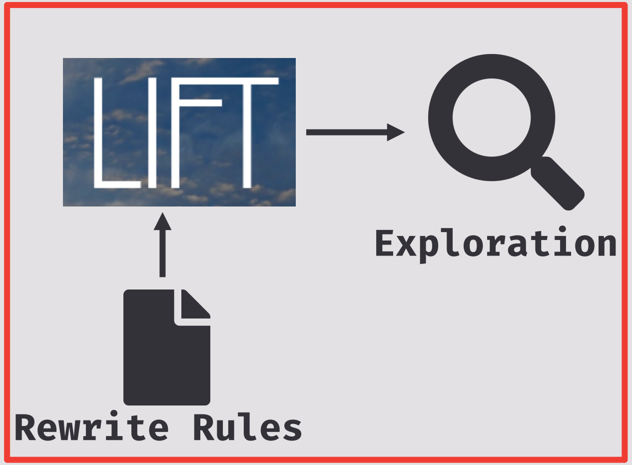


CGO'18
CGO'17
CASES'16
GPGPU'16
ICFP'15

MOTIVATION

How do we optimize programs today?

Functional Domain-Specific Compilers



~~CGO'18~~ No explanation

CGO'17

CASES'16

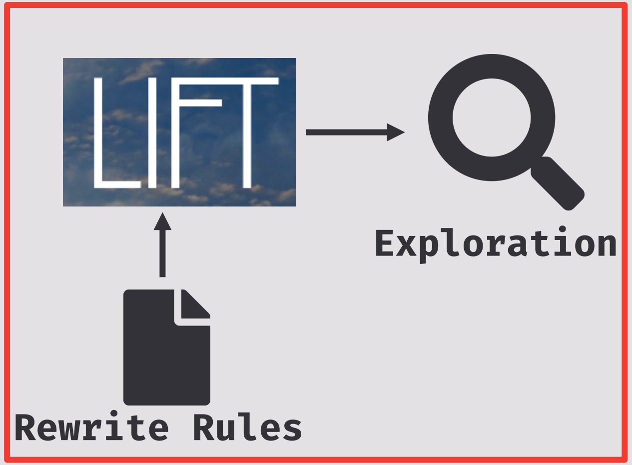
GPGPU'16

ICFP'15

MOTIVATION

How do we optimize programs today?

Functional Domain-Specific Compilers

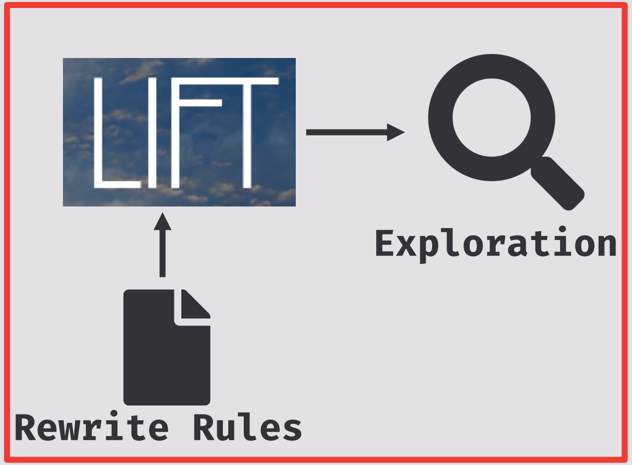


~~CGO'18~~
~~CGO'17~~ No explanation
CASES'16
GPGPU'16
ICFP'15

MOTIVATION

How do we optimize programs today?

Functional Domain-Specific Compilers



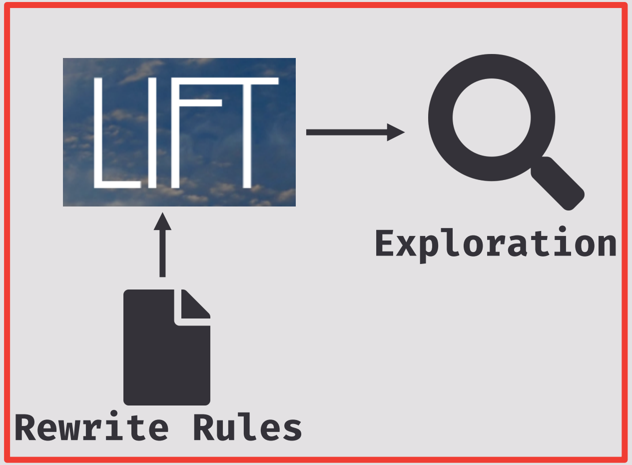
~~CGO'18~~
~~CGO'17~~
CASES'16
GPGPU'16
ICFP'15

"…the resulting space is extremely large, even potentially unbounded,which opens up a ***new research challenge***.
…
We present here a first, ***simple and heuristic-based*** pruning strategy to tackle the space complexity problem. ***Future research*** will investigate more advanced techniques to fully automate the pruning process.
…"

MOTIVATION

How do we optimize programs today?

Functional Domain-Specific Compilers



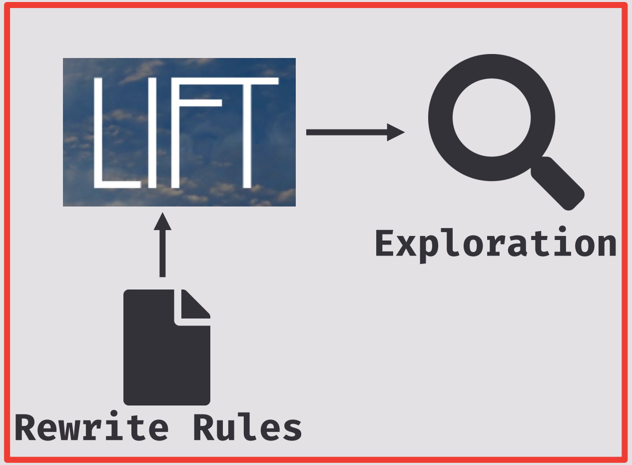
~~CGO'18~~
~~CGO'17~~
~~CASES'16~~
GPGPU'16
ICFP'15

> "…guide the automatic rewrite process by grouping rewrite rules together into ***macro rules***
>
> …
>
> These macro rules are more flexible than the simple rules. They ***try to apply different sequences*** of rewrites to achieve their optimization goal."

MOTIVATION

How do we optimize programs today?

Functional Domain-Specific Compilers



~~CGO'18~~
~~CGO'17~~
~~CASES'16~~
GPGPU'16
ICFP'15

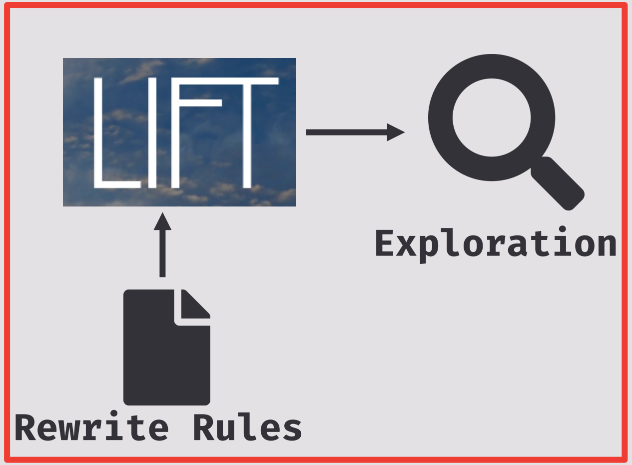
> "...guide the automatic rewrite process by grouping rewrite rules together into ***macro rules***
>
> ...
>
> These macro rules are more flexible than the simple rules. They ***try to apply different sequences*** of rewrites to achieve their optimization goal."

unclear semantics of macro rules
unclear how to define macro rules

MOTIVATION

How do we optimize programs today?

Functional Domain-Specific Compilers



~~CGO'18~~
~~CGO'17~~
~~CASES'16~~
~~GPGPU'16~~
ICFP'15

"...we have developed a ***simple automatic search*** strategy

...

Our current search strategy is ***rather basic*** and just designed to prove that it is possible to find good implementations automatically.

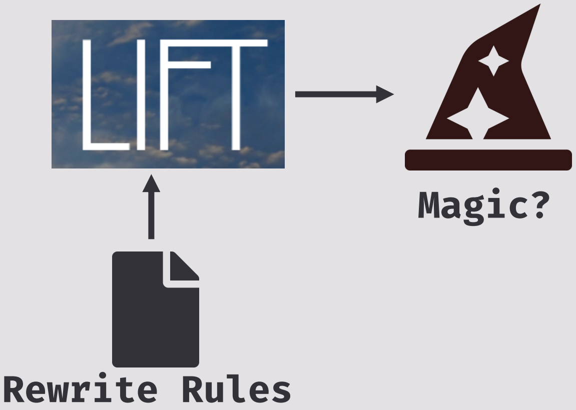
...

We ***envision replacing this*** exploration strategy in the future"

MOTIVATION

How do we optimize programs today?

Functional Domain-Specific Compilers

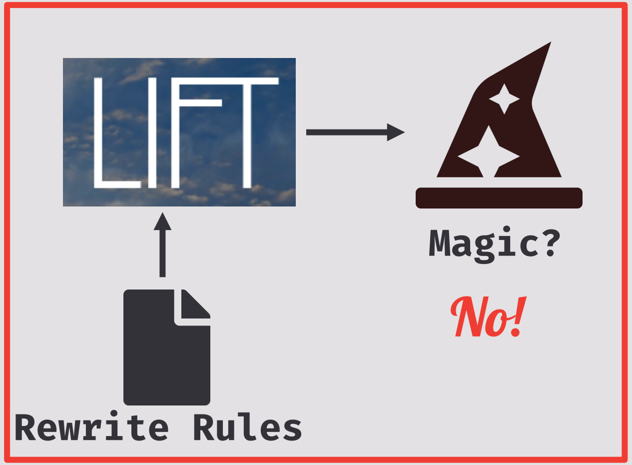


~~CGO'18~~
~~CGO'17~~
~~CASES'16~~
~~GPGPU'16~~
~~ICFP'15~~

MOTIVATION

How do we optimize programs today?

Functional Domain-Specific Compilers



~~CGO'18~~
~~CGO'17~~
~~CASES'16~~
~~GPGPU'16~~
~~ICFP'15~~

The future research is here...

STRATEGIES

Optimizing Programs like it's ~~1998~~ 2019*

* Elevate's design is inspired by Eelco Visser and others work,
e.g. the 1998 ICFP paper *"Building program optimizers with rewriting strategies"*

ELEVATE

lessons from the past

- A ***Strategy*** is a function: ***Program*** → ***Program***
- A ***Transformation*** is the simplest strategy:

map(f) → join ∘ map(map(f)) ∘ split(n)

- ***isDefined*** tests if a strategy is defined for a given program

isDefined : Strategy → Program → Bool

- ***apply*** applies a strategy at a particular location in the program *(and might fail)*

apply : Strategy → Location → Strategy

- A ***Location*** and ***Traversal*** are ADTs:

```
data Location = Position(Traversal, Int) | FindFirst(Traversal, Program → Bool)
data Traversal = BFS | DFS
```

COMPOSING SRATEGIES

defining simple building blocks

id: Strategy
id = λp . p

seq: Strategy → Strategy -> Strategy
seq = λf . λs . λp . s (f p)

leftChoice: Strategy → Strategy → Strategy
leftChoice = λf . λs . λp . `try` (f p) `catch` (s p)

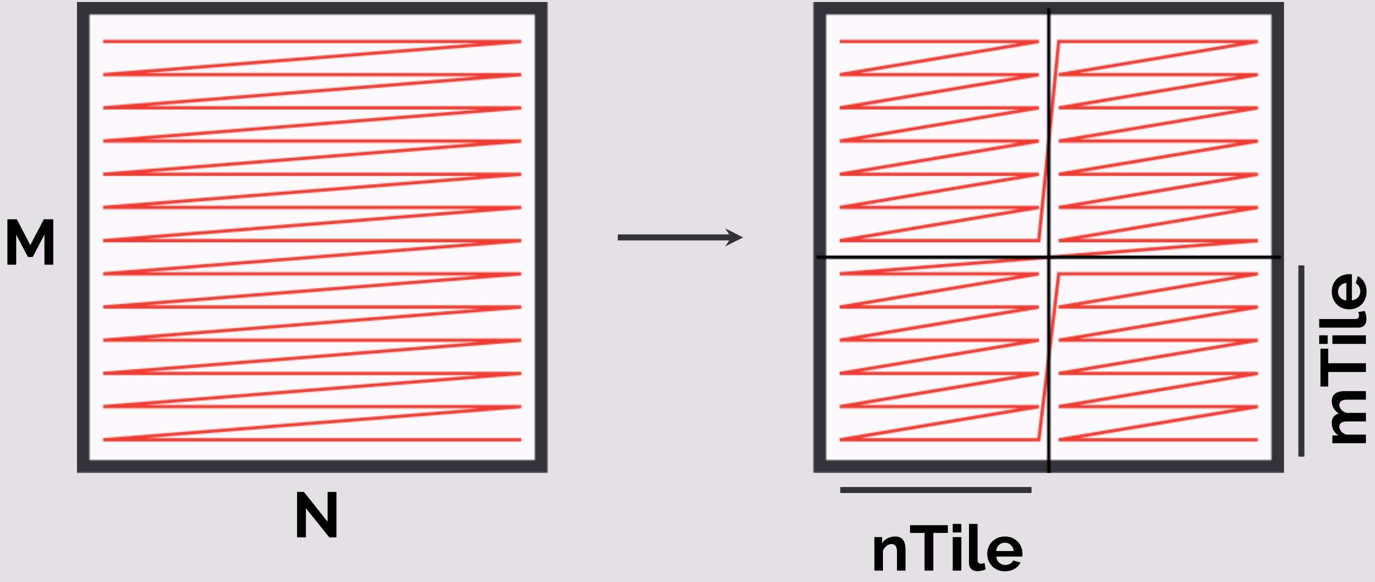
try: Strategy → Strategy
try = λs . *leftChoice* s id

repeat: Strategy → Strategy
repeat = λs . *try* (s ; (*repeat* s))

normalize: Strategy → Strategy
normalize: λs . *repeat*(*apply* s *FindFirst*(BFS, isDefined s))

DATA LAYOUT STRATEGY

what is tiling and why is it important for performance



Benefits of tiling:

- Exposes more ***parallelism***
- Enables to exploit ***locality***

→ **performance improvements of orders of magnitudes**

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` ***in Elevate***

```
tiling: Int → Strategy
tiling =  λn . λp .
  ((tileEveryDimension n) ;        // step 1
    rewriteNormalForm ;            // step 2
      rearrangeDimensions) p   // step 3
```

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` *in Elevate*

Short form for ***seq***

```
tiling: Int → Strategy
tiling =  λn . λp .
  ((tileEveryDimension n) ;       // step 1
    rewriteNormalForm ;          // step 2
     rearrangeDimensions) p    // step 3
```

We have *decomposed* the tiling Strategy into three *conceptual steps* that we define also as Strategies

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` *in Elevate*

Elevate:

tiling: Int → Strategy
tiling = λn . λp .
 ((tileEveryDimension n) ;
  rewriteNormalForm ;
   rearrangeDimensions) p

Lift:

$**f$

OpenCL:

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

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` *in Elevate*

Elevate:

step 1

tileEveryDimension: Int → Strategy
tileEveryDimension = λn . λp .
fold (λ (p, l). **try** (**apply** (splitJoin n) l p)
p
(**findAll** (**isDefined** (*splitJoin* n)) p))

(*splitJoin* size) = $*f \rightarrow$ **J** $\circ\ **f \circ$ **S**(size)
fold: (List L) → (P → L → P) → P → P
findAll: (Program → Bool) → Program → (List Location)

Lift:

$**f$

OpenCL:

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

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` *in Elevate*

Elevate:

step 1

tileEveryDimension: Int → Strategy
tileEveryDimension = λn . λp .
fold (λ (p, l). **try** (**apply** (splitJoin n) l p)
p
(**findAll** (**isDefined** (*splitJoin* n)) p))

(*splitJoin* size) = $*f \rightarrow$ **J** $\circ$ $**f$ $\circ$ **S**(size)
fold: (List L) → (P → L → P) → P → P
findAll: (Program → Bool) → Program → (List Location)

Lift:

$**f$

2 locations where splitJoin rule is defined

OpenCL:

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

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` *in Elevate*

Elevate:

step 1

tileEveryDimension: Int → Strategy
tileEveryDimension = λn . λp .
fold (λ (p, l). **try** (**apply** (splitJoin n) l p)
p
(**findAll** (**isDefined** (*splitJoin* n)) p))

(*splitJoin* size) = $^{*}f \rightarrow \mathbf{J} \circ {}^{**}f \circ \mathbf{S}$(size)
fold: (List L) → (P → L → P) → P → P
findAll: (Program → Bool) → Program → (List Location)

Lift:

$\mathbf{J} \circ$
$^{**}(\mathbf{J} \circ {}^{**}f \circ \mathbf{S}) \circ$
$\mathbf{S}$

OpenCL:

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

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` *in Elevate*

Elevate:

step 1

tileEveryDimension: Int → Strategy
tileEveryDimension = λn . λp .
fold (λ (p, l). **try** (**apply** (splitJoin n) l p)
p
(**findAll** (**isDefined** (*splitJoin* n)) p))

(*splitJoin* size) = **f* → **J** ∘ ***f* ∘ **S**(size)
fold: (List L) → (P → L → P) → P → P
findAll: (Program → Bool) → Program → (List Location)

Lift:

J ∘
(J** ∘ ***f* ∘ **S**) ∘
S

OpenCL:

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

tiling the M & N dimension

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` *in Elevate*

Elevate:

step 2

rewriteNormalForm: Strategy
rewriteNormalForm = λp .
(**normalize** *mapFission*) p

mapFission = $^*(f \circ g) \rightarrow {}^*f \circ {}^*g$

Lift:

$\mathbf{J} \circ$
$^{**}(\mathbf{J} \circ {}^{**}f \circ \mathbf{S}) \circ$
$\mathbf{S}$

OpenCL:

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

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` *in Elevate*

Elevate:

step 2

rewriteNormalForm: Strategy
rewriteNormalForm = λp . **repeat**
(**apply** *mapFission*
findFirst (*isDefined mapFission*))
p

mapFission = $*(f \circ g) \rightarrow *f \circ *g$

Lift:

$\mathbf{J} \circ$
$**(\mathbf{J} \circ **f \circ \mathbf{S}) \circ$
$\mathbf{S}$

OpenCL:

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

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` *in Elevate*

Elevate:

step 2

rewriteNormalForm: Strategy
rewriteNormalForm = λp . **repeat**
(**apply** *mapFission*
findFirst (*isDefined mapFission*))
p

mapFission = *(*f* ∘ *g*) → **f* ∘ **g*

Lift:

$J \circ {}^{**}J \circ$
$^{****}f \circ$
$^{**}S \circ S$

OpenCL:

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

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` ***in Elevate***

Elevate:

step 3

rearrangeDimensions: Int ⟶ Strategy
rearrangeDimensions = λd . λp .
 d *match*
  case <2 : p
  case 2 : (*shuffleDimension* d) *p*
  case _ : (rearrangeDimension (d-1) ;
            (*shuffleDimension d*)) *p*

Lift:

J ∘ ****J** ∘
****f ∘
****S** ∘ **S**

OpenCL:

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

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` *in Elevate*

Elevate:

step 3

rearrangeDimensions: Int → Strategy
rearrangeDimensions = λd . λp .
 d *match*
  case <2 : p
  case 2 : (*shuffleDimension* d) *p*
  case _ : (rearrangeDimension (d-1) ;
            (*shuffleDimension d*)) *p*

Lift:

J ∘ **J ∘
*T ∘ ****f ∘ *T ∘
**S ∘ S

OpenCL:

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

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

DATA LAYOUT STRATEGY

let's define Halide's `.tiling` *in Elevate*

Elevate:

step 3

rearrangeDimensions: Int → Strategy
rearrangeDimensions = λd . λp .
 d *match*
 case <2 : p
 case 2 : (*shuffleDimension* d) *p*
 case _ : (rearrangeDimension (d-1) ;
 (*shuffleDimension d*)) *p*

Lift:

J ∘ ***J** ∘
*__T__ ∘ ****f ∘ *__T__ ∘
****S** ∘ **S**

OpenCL:

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

swapped loop-order

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

DATA LAYOUT STRATEGY

Why is defining tiling as a strategy a good idea?

ELEVATE

```
tiling: Int → Strategy
tiling =  λs . λp .
  ((tileEveryDimension s) ;       // step 1
    rewriteNormalForm ;           // step 2
     rearrangeDimensions) p       // step 3
```

vs

Halide

```
...
out.bound(x, 0, size)
.bound(y, 0, size)
.tile(x, y, xi, yi, x_tile * vec_size * …)
.split(yi, ty, yi, y_unroll)
.vectorize(xi, vec_size)
.split(xi, xio, xii, warp_size)
...
```

- Strategy not built-in in the compiler
- Works for arbitrary dimensions

2D specific

DATA LAYOUT STRATEGY

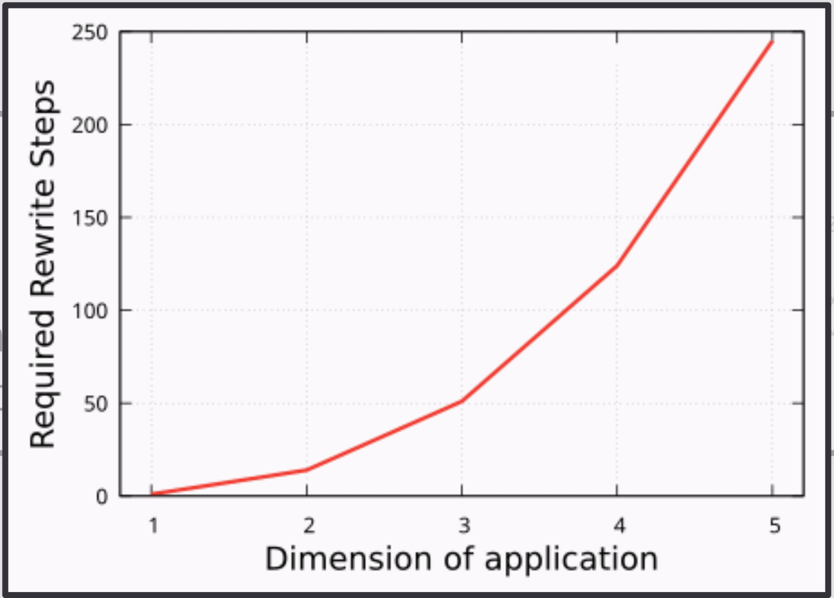
Why is defining tiling as a strategy a good idea?

ELEVATE

```
tiling: Int → Strategy
tiling = λs . λp .
  ((tileEveryDimension
    rewriteNormalForm
      rearrangeDimensio
```

Halide

```
size)
ze)
yi, x_tile * vec_size * …)
yi, y_unroll)
vec_size)
xii, warp_size)
```



- Strategy not built-in in the compiler

Strategies are crucial for tiling in higher dimensions

DATA LAYOUT STRATEGY

Why is defining tiling as a strategy a good idea?

ELEVATE

```
overlapTiling: Int → Strategy
overlapTiling =  λs . λp .
  ((tileEveryDimension s) ;        // step 1
    rewriteNormalForm ;            // step 2
     rearrangeDimensions) ;        // step 3
      (try overlapping) p          // step 4
```

vs

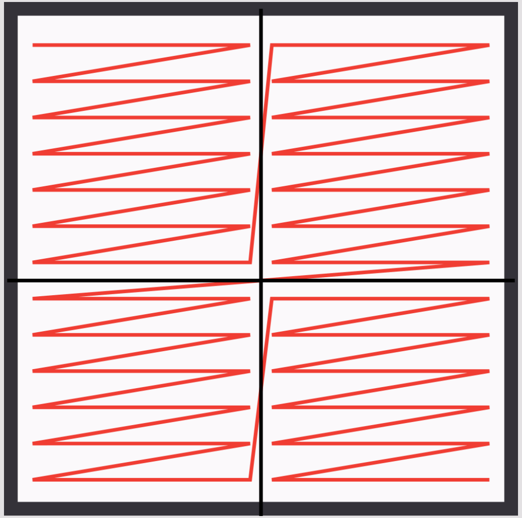
Halide

```
...
out.bound(x, 0, size)
.bound(y, 0, size)
.tile(x, y, xi, yi, x_tile * vec_size * …)
.split(yi, ty, yi, y_unroll)
.vectorize(xi, vec_size)
.split(xi, xio, xii, warp_size)
...
```

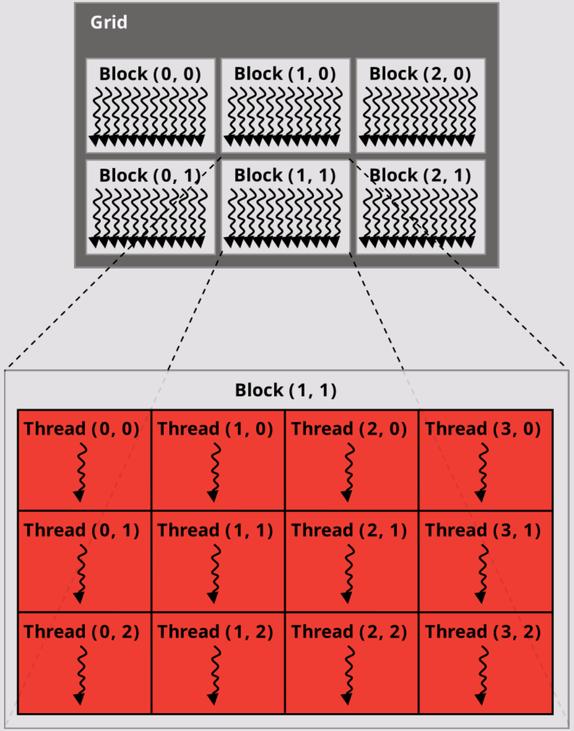
- Strategy not built-in in the compiler
- Works for arbitrary dimensions
- Easy to extend and reuse existing strategies

PARALLELISM STRATEGY

How to best exploit parallelism in the hardware



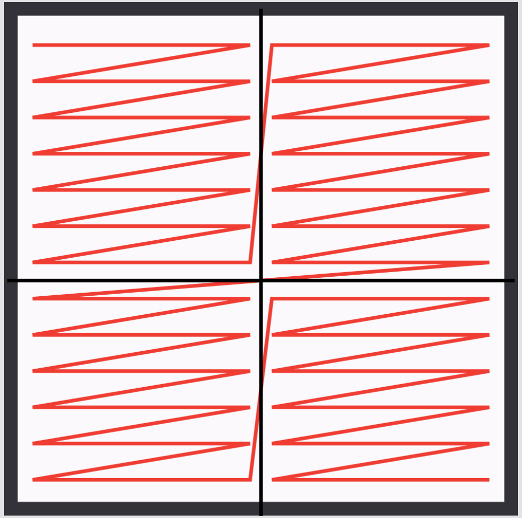
Application



GPU Parallelism Model

PARALLELISM STRATEGY

How to best exploit parallelism in the hardware



Application

Futhark @ PPoPP'19


Incremental Flattening for Nested Data Parallelism

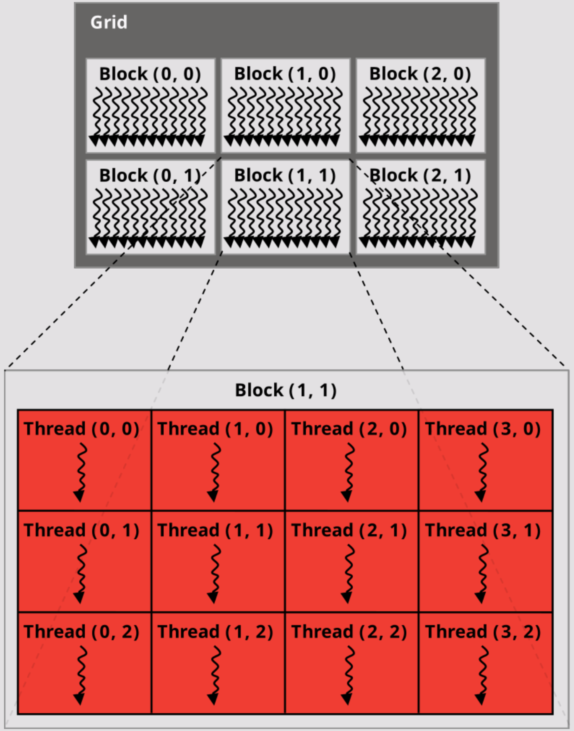
Troels Henriksen, University of Copenhagen

Frederik Thorøe, University of Copenhagen

Martin Elsman, University of Copenhagen

Cosmin Oancea, University of Copenhagen

Goal:
Minimize parallelism while still fully utilize hardware



GPU Parallelism Model

PARALLELISM STRATEGY

let's define Halide's .tiling

Elevate:

```
workgroupMapping: Strategy
workgroupMapping = λp . (
  (applyOutermost (mapWrg 1)) ;
  (applyOutermost (mapWrg 0)) ;
  (applyOutermost (mapLcl 1)) ;
  (applyOutermost (mapLcl 0)) ;
  (normalize mapSeq)) p
```

applyOutermost: Strategy → Strategy
applyOutermost = λs . λp .
apply *s* (FindFirst DFS (*isDefined s*)) p

Lift:

$\mathbf{J} \circ {}^{**}\mathbf{J} \circ {}^{*}\mathbf{T} \circ$
${}^{****}f \circ$
${}^{*}\mathbf{T} \circ {}^{**}\mathbf{S} \circ \mathbf{S}$

OpenCL:

```
for(int i = 0; i < M; i++) {
 for(int j = 0; j < N; j++) {
  for(int ii = 0; ii < s; ii++) {
   for(int jj = 0; jj < s; jj++) {
       // f
}}}}
```

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

PARALLELISM STRATEGY

let's define Halide's .tiling

Elevate:

```
workgroupMapping: Strategy
workgroupMapping =  λp . (
  (applyOutermost (mapWrg 1)) ;
  (applyOutermost (mapWrg 0)) ;
  (applyOutermost (mapLcl 1)) ;
  (applyOutermost (mapLcl 0)) ;
  (normalize mapSeq)) p
```

applyOutermost: Strategy → Strategy
applyOutermost = λs . λp .
apply *s* (FindFirst DFS (*isDefined s*)) p

Lift:

$$\mathbf{J} \circ {}^{**}\mathbf{J} \circ {}^{*}\mathbf{T} \circ {}^{****}f \circ {}^{*}\mathbf{T} \circ {}^{**}\mathbf{S} \circ \mathbf{S}$$

(mapWrg i) = *map f* → *mapWrg$_i$ f*
(mapLcl i) = *map f* → *mapLcl$_i$ f*

OpenCL:

```
for(int i = 0; i < M; i++) {
 for(int j = 0; j < N; j++) {
  for(int ii = 0; ii < s; ii++) {
   for(int jj = 0; jj < s; jj++) {
       // f
}}}}
```

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

PARALLELISM STRATEGY

let's define Halide's .tiling

Elevate:

```
workgroupMapping: Strategy
workgroupMapping =  λp . (
  (applyOutermost (mapWrg 1)) ;
  (applyOutermost (mapWrg 0)) ;
  (applyOutermost (mapLcl 1)) ;
  (applyOutermost (mapLcl 0)) ;
  (normalize mapSeq)) p
```

applyOutermost: Strategy → Strategy
applyOutermost = λs . λp .
apply *s* (FindFirst DFS (*isDefined s*)) p

Lift:

$J \circ {**}J \circ {*}T \circ {****}f \circ {*}T \circ {**}S \circ S$

(mapWrg i) = map f → mapWrg$_i$ f
(mapLcl i) = map f → mapLcl$_i$ f

f must write to memory

OpenCL:

```
for(int i = 0; i < M; i++) {
 for(int j = 0; j < N; j++) {
  for(int ii = 0; ii < s; ii++) {
   for(int jj = 0; jj < s; jj++) {
       // f
}}}}
```

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

PARALLELISM STRATEGY

let's define Halide's .tiling

Elevate:

workgroupMapping: Strategy
workgroupMapping = λp . (
  (applyOutermost (mapWrg 1)) ;
  (applyOutermost (mapWrg 0)) ;
  (applyOutermost (mapLcl 1)) ;
  (applyOutermost (mapLcl 0)) ;
  (normalize mapSeq)) p

applyOutermost: Strategy → Strategy
applyOutermost = λs . λp .
apply *s* (FindFirst DFS (*isDefined s*)) p

Lift:

J ∘ ****J** ∘ ***T** ∘
map(map(
map(map *f*))) ∘
***T** ∘ ****S** ∘ **S**

(mapWrg i) = map f → $mapWrg_i\ f$
(mapLcl i) = map f → $mapLcl_i\ f$

OpenCL:

```
for(int i = 0; i < M; i++) {
 for(int j = 0; j < N; j++) {
  for(int ii = 0; ii < s; ii++) {
   for(int jj = 0; jj < s; jj++) {
         // f
}}}}
```

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

PARALLELISM STRATEGY

let's define Halide's .tiling

Elevate:

workgroupMapping: Strategy
workgroupMapping = λp . (
  (applyOutermost (mapWrg 1)) ;
  (applyOutermost (mapWrg 0)) ;
  (applyOutermost (mapLcl 1)) ;
  (applyOutermost (mapLcl 0)) ;
  (normalize mapSeq)) p

applyOutermost: Strategy → Strategy
applyOutermost = λs . λp .
apply *s* (FindFirst DFS (*isDefined s*)) p

Lift:

$J \circ {}^{**}J \circ {}^{*}T \circ$
$mapWrg_1(map($
$map(map\ f))) \circ$
${}^{*}T \circ {}^{**}S \circ S$

(mapWrg i) = map f → mapWrg$_i$ f
(mapLcl i) = map f → mapLcl$_i$ f

OpenCL:

```
int i = get_group_id(1);
 for(int j = 0; j < N; j++) {
  for(int ii = 0; ii < s; ii++) {
   for(int jj = 0; jj < s; jj++) {
          // f
}}}}
```

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

PARALLELISM STRATEGY

let's define Halide's .tiling

Elevate:

workgroupMapping: Strategy
workgroupMapping = λp . (
  (applyOutermost (mapWrg 1)) ;
  (applyOutermost (mapWrg 0)) ;
  (applyOutermost (mapLcl 1)) ;
  (applyOutermost (mapLcl 0)) ;
  (normalize mapSeq)) p

applyOutermost: Strategy → Strategy
applyOutermost = λs . λp .
apply *s* (FindFirst DFS (*isDefined s*)) p

Lift:

$J \circ {**J} \circ {*T} \circ$
$mapWrg_1(mapWrg_0($
$mapLcl_1(mapLcl_0\ f)))$
$\circ {*T} \circ {**S} \circ S$

(mapWrg i) = map f → mapWrg$_i$ *f*
(mapLcl i) = map f → mapLcl$_i$ *f*

OpenCL:

```
int i = get_group_id(1);
 int j = get_group_id(0);
  int ii = get_local_id(1);
   int jj = get_local_id(0);
          // f
}}}}
```

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

PARALLELISM STRATEGY

let's define Halide's .tiling

Elevate:

```
workgroupMapping: Strategy
workgroupMapping =  λp . (
  (applyOutermost (mapWrg 1)) ;
  (applyOutermost (mapWrg 0)) ;
  (applyOutermost (mapLcl 1)) ;
  (applyOutermost (mapLcl 0)) ;
  (normalize mapSeq)) p
```

Lift:

```
map(map f))
```

OpenCL:

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

What if we applied this strategy to our non-tiled Lit program?

PARALLELISM STRATEGY

let's define Halide's .tiling

Elevate:

workgroupMapping: Strategy
workgroupMapping = λp . (
  (applyOutermost (mapWrg 1)) ;
  (applyOutermost (mapWrg 0)) ;
  (applyOutermost (mapLcl 1)) ; **!**
  (applyOutermost (mapLcl 0)) ;
  (normalize mapSeq)) p

Lift:

map(map f))

OpenCL:

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

What if we applied this strategy to our non-tiled Lit program?

PARALLELISM STRATEGY

let's define Halide's .tiling

Elevate:

globalMapping: Strategy
globalMapping = λp . (
  (applyOutermost (mapGlb 1)) ;
  (applyOutermost (mapGlb 0)) ;
  (normalize mapSeq)) p

Lift:

$\mathrm{mapGlb}_1($
$\quad \mathrm{mapGlb}_0\ f))$

OpenCL:

```
int i = get_global_id(1);
 int j = get_global_id(0);
   out[i][j] = f(in[i][j]);
}}
```

What if we applied this strategy to our non-tiled Lift program?

Lift abbreviations: * = map | **S** = split(s) | **J** = join | **T** = transpose

PARALLELISM STRATEGY

exploiting intentional failing of strategies

Short form for ***leftChoice***

```
mapParallelism: Strategy
mapParallelism =  λp .
  (workgroup3D +> workgroup2D +> global2D +> sequential) p
```

Trying to ***exploit all available parallelism*** and ***gradually fall back*** to strategies which make use of less parallelism

Achieves the same goal as Futhark's incremental flattening

ELEVATE

Specifying Compiler Optimizations:

- ***Principled:***
  One principled way to understand, write and apply compiler optimizations as Strategies
- ***Extensible:***
  not be fix and built-in.
  let programmers define abstractions and inject domain & expert knowledge

One Language - Many Optimizations:

Controlling different categories of optimizations with the same language

Algorithmic	*Hardware-specific*
Data-Layout	Memory
Computation	Parallelism