ACHIEVING HIGH-PERFORMANCE THE *Functional* WAY

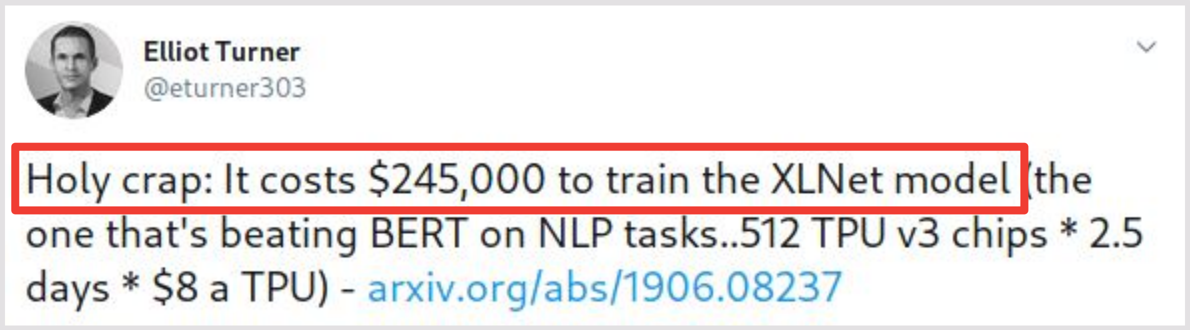
A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies

WWU MÜNSTER | THE UNIVERSITY of EDINBURGH | University of Glasgow

Bastian Hagedorn | Johannes Lenfers | Thomas Koehler | Xueying Qin | Sergei Gorlatch | Michel Steuwer

HIGH-PERFORMANCE

Why do we care ?


Elliot Turner
@eturner303

Holy crap: It costs $245,000 to train the XLNet model (the one that's beating BERT on NLP tasks..512 TPU v3 chips * 2.5 days * $8 a TPU) - arxiv.org/abs/1906.08237

HIGH-PERFORMANCE

Why do we care ?

Elliot Turner
@eturner303

Holy crap: It costs $245,000 to train the XLNet model (the one that's beating BERT on NLP tasks..512 TPU v3 chips * 2.5 days * $8 a TPU) - arxiv.org/abs/1906.08237

Elliot Turner
@eturner303

Another way (using carbon as opposed to $$) of thinking about this experiment: Training XLNet to convergence releases around 4.9 metric tons of CO_2 into the atmosphere (equivalent to driving a car around 11,000 miles)

GH-PERFORMANCE

Why do we care ?

Turner
rner303

It costs $245,000 to train the XLNet model (the
beating BERT on NLP tasks..512 TPU v3 chips * 2.5
a TPU) - arxiv.org/abs/1906.08237

Turner
rner303

ay (using carbon as opposed to $$) of thinking
experiment: Training XLNet to convergence
round 4.9 metric tons of CO2 into the atmosphere
nt to driving a car around 11,000 miles)

HIGH-PERFORMANCE

Achieving High-Performance the ~~Functional~~ Manual Way

```
__global__ void matmul(
        float *A, float *B, float *C,
        int K, int M, int N) {

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    float acc = 0.0;

    for (int k = 0; k < K; k++) {
        acc += A[y * M + k] * B[k * N + x];
    }

    C[y * N + x] = acc;
}
```

Naive Matrix Multiplication in NVIDIA CUDA

HIGH-PERFO

Achieving High-Performance

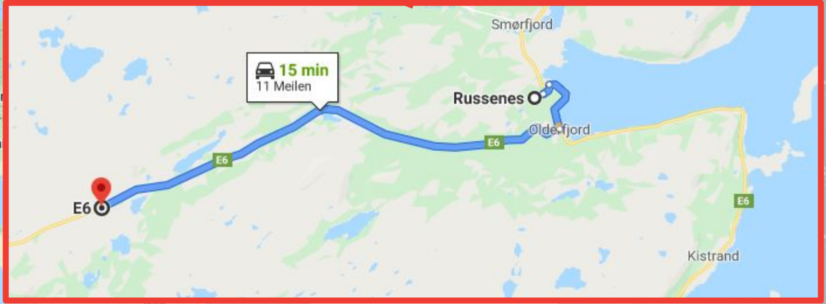
```
__global__ void matmul(
        float *A, float *B, float *C,
        int K, int M, int N) {

    int x = blockIdx.x * blockDim.x + threadIdx.x;
    int y = blockIdx.y * blockDim.y + threadIdx.y;
    float acc = 0.0;

    for (int k = 0; k < K; k++) {
        acc += A[y * M + k] * B[k * N + x];
    }

    C[y * N + x] = acc;
}
```

Naive Matrix Multiplication in NVIDIA CUDA

Optimized Matrix Multiplication

PERFO

High Performance t



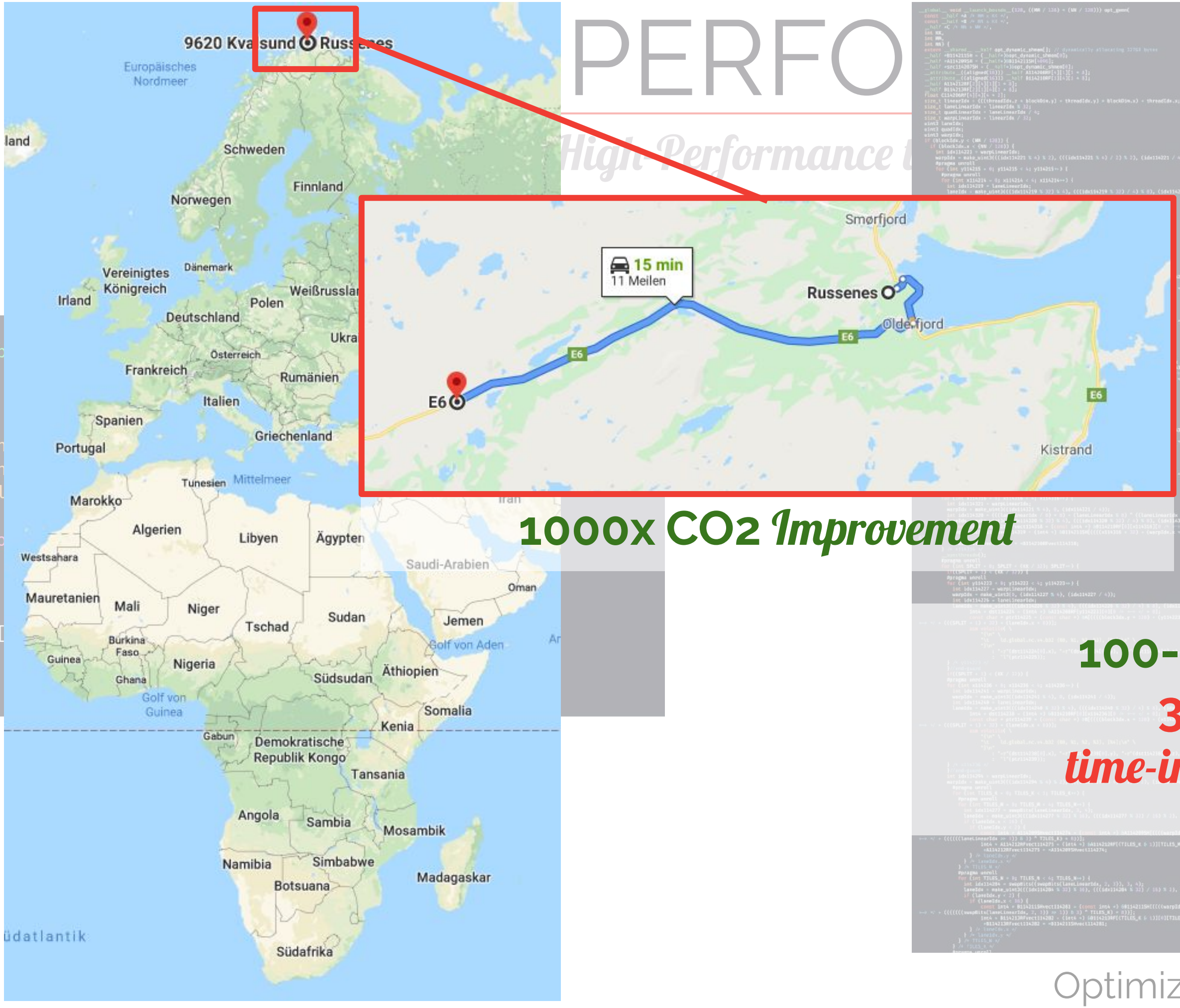
1000x CO2 *Improvement*

100-1000x *performance*

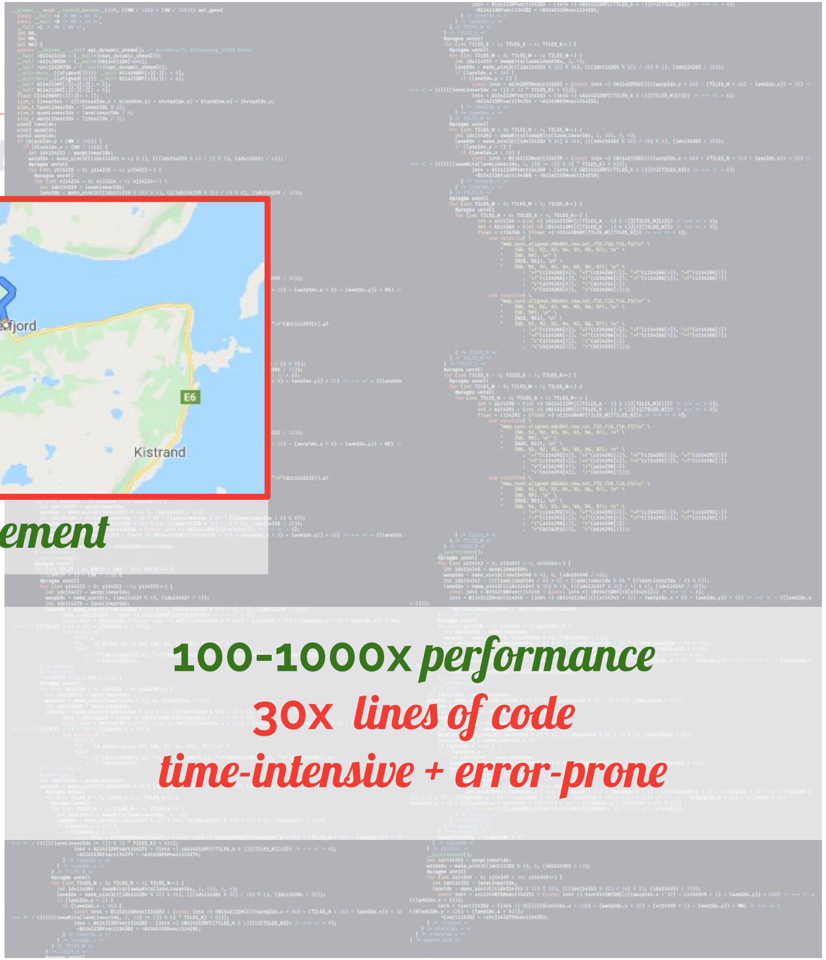
Optimized Matrix Multiplication

PERFO

High Performance



1000x CO2 *Improvement*



100-1000x *performance*

30x *lines of code*

time-intensive + error-prone

Optimized Matrix Multiplication

HIGH-PERFORMANCE

Achieving High-Performance the ~~Functional~~ Decoupled Way

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')
```

What to compute

Performance Engineer → Schedule

```
# blocking version
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, ki, xi, yi)
```

How to optimize

Halide

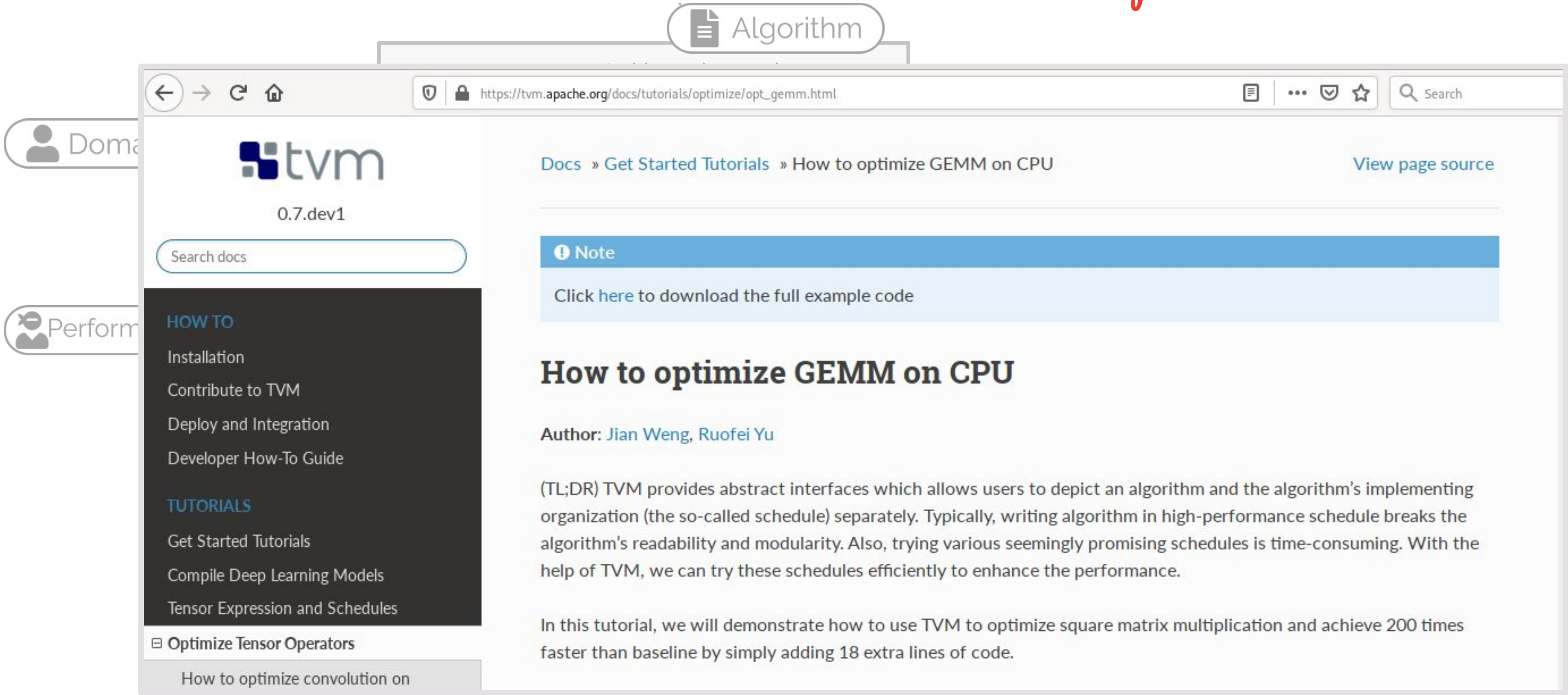
tvm

Tiramisu-Compiler / **tiramisu**

Fireiron NVIDIA

HIGH-PERFORMANCE

Achieving High-Performance the ~~Functional~~ Decoupled Way



https://tvm.apache.org/docs/tutorials/optimize/opt_gemm.html

tvm

0.7.dev1

Search docs

HOW TO

Installation

Contribute to TVM

Deploy and Integration

Developer How-To Guide

TUTORIALS

Get Started Tutorials

Compile Deep Learning Models

Tensor Expression and Schedules

Optimize Tensor Operators

How to optimize convolution on

Docs » Get Started Tutorials » How to optimize GEMM on CPU

View page source

Note

Click here to download the full example code

How to optimize GEMM on CPU

Author: Jian Weng, Ruofei Yu

(TL;DR) TVM provides abstract interfaces which allows users to depict an algorithm and the algorithm's implementing organization (the so-called schedule) separately. Typically, writing algorithm in high-performance schedule breaks the algorithm's readability and modularity. Also, trying various seemingly promising schedules is time-consuming. With the help of TVM, we can try these schedules efficiently to enhance the performance.

In this tutorial, we will demonstrate how to use TVM to optimize square matrix multiplication and achieve 200 times faster than baseline by simply adding 18 extra lines of code.

HIGH-PERFORMANCE

Achieving High-Performance the ~~Functional~~ Decoupled Way

Domain-Scientist

Algorithm

```python
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')
```

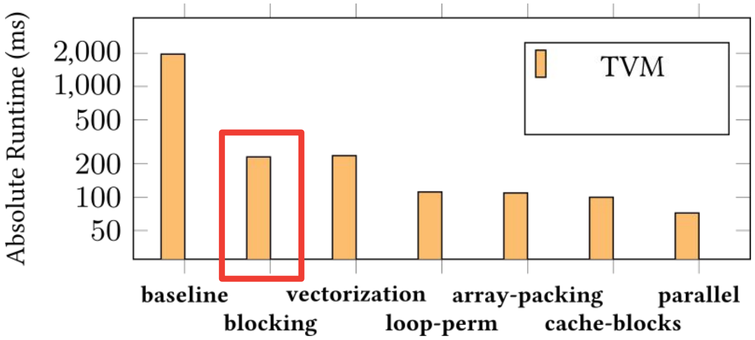
Performance Engineer

```python
# blocking version
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, ki, xi, yi)
```

Schedule

tvm

Compiler



HIGH-PERFORMANCE

Achieving High-Performance the ~~Functional~~ Way
Decoupled

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')
```

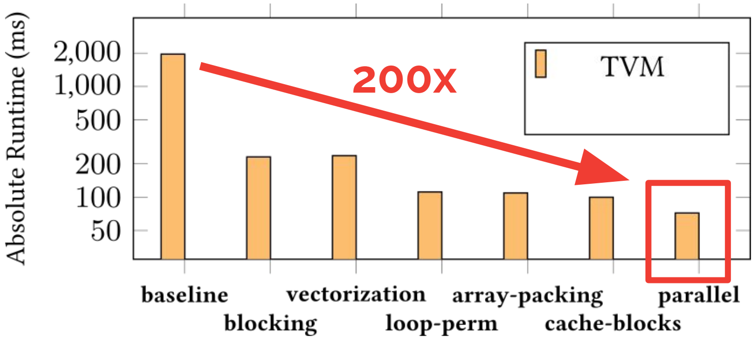
Performance Engineer

```
# "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

Compiler



HIGH-PERFORMANCE

Achieving High-Performance the ~~Functional~~ Decoupled Way

Algorithm

Domain-Scientist

```
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')
```

No clear separation

hinders reuse!

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

HIGH-PERFORMANCE

Achieving High-Performance the ~~Functional~~ Decoupled Way

Algorithm

Domain-Scientist

```
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)
```

cache_write(*tensor, scope*)

?

Create a cache write of original tensor, before storing into tensor.

```
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!

HIGH-PERFORMANCE

Achieving High-Performance the ~~Functional~~ Decoupled Way

Algorithm

Domain-Scientist

```
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!

HIGH-PERFORMANCE

Achieving High-Performance the ~~Functional~~ Way
Decoupled

Algorithm

Domain-Scientist

```
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')
```

No clear separation

hinders reuse!

We aim for a ***more principled*** way to ***describe and apply optimizations***

```
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

Optimizations are built-in

no extensibility!

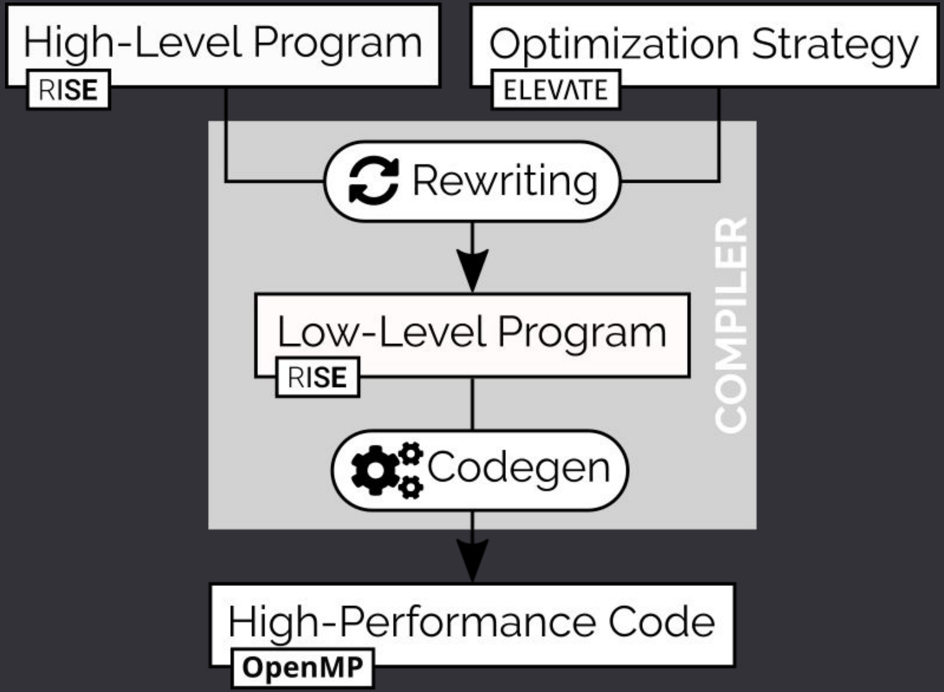
OUR GOALS

A Principled Way to Separate, Describe, and Apply Optimizations

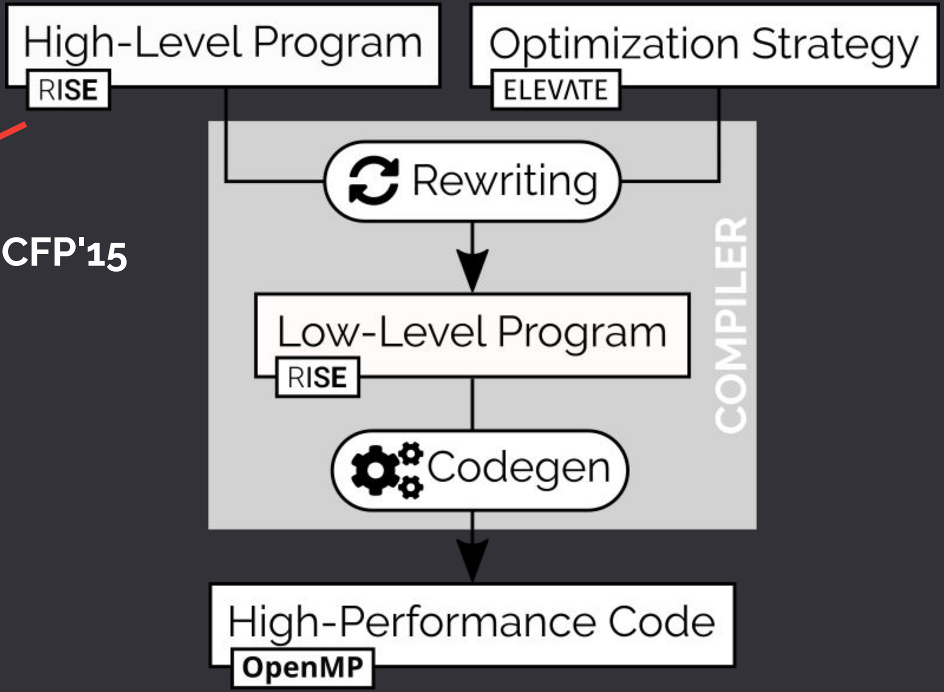
(1) *Separate concerns*: Computations should be expressed at a high abstraction level only. They should not be changed to express optimizations;
(2) *Facilitate reuse*: Optimization strategies should be defined clearly separated from the computational program facilitating reusability of computational programs and strategies;
(3) *Enable composability*: Computations *and* strategies should be written as compositions of user-defined building blocks (possibly domain-specific ones); *both languages* should facilitate the creation of higher-level abstractions;
(4) *Allow reasoning*: Computational patterns, but also especially strategies, should have a precise, well-defined semantics allowing reasoning about them;
(5) *Be explicit*: Implicit default behavior should be avoided to empower users to be in control.

Fundamentally we argue that a more principled high-performance code generation approach should be holistic by considering computation and optimization strategies equally important. As a consequence, a strategy language should be built with the same standards as a language describing computation.

The *Functional* Way

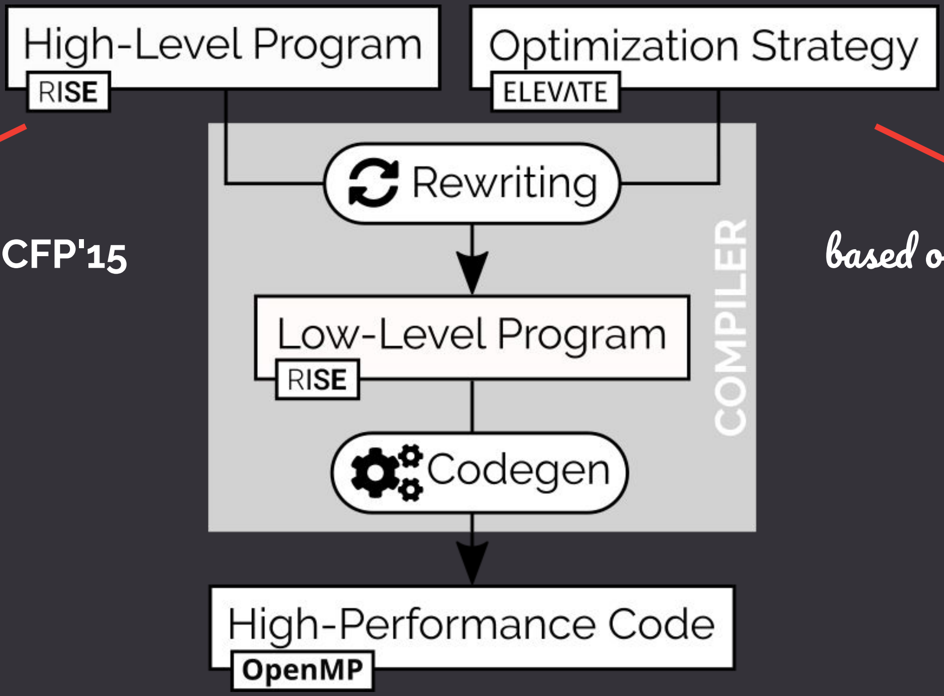


The *Functional* Way



based on Steuwer et. al. **ICFP'15**

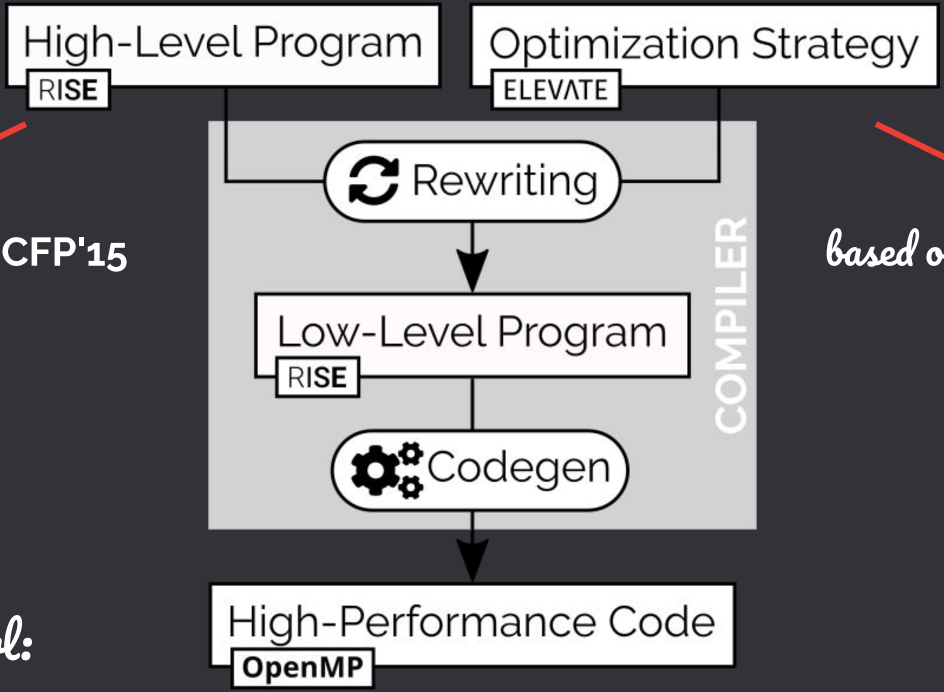
The *Functional* Way



based on Steuwer et. al. **ICFP'15**

based on Visser et. al. **ICFP'98**

The *Functional* Way



based on Steuwer et. al. **ICFP'15**

based on Visser et. al. **ICFP'98**

This Functional Pearl:

We apply established *functional programming techniques* for elegantly expressing *high-performance program optimizations* as *composable rewrite strategies*

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:

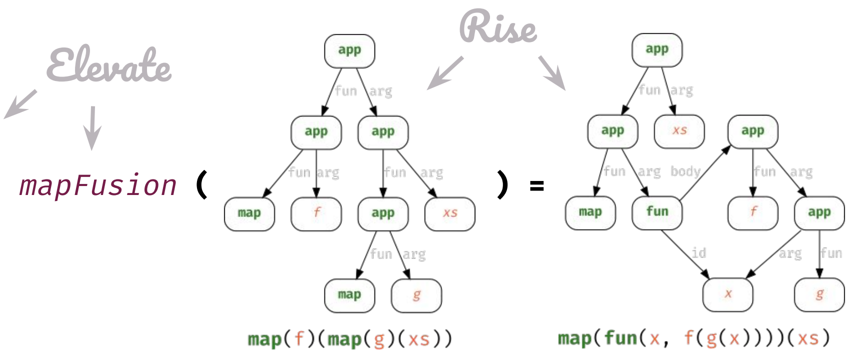
```
type Strategy[P] = P => RewriteResult[P]
```

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

```
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 )
  }
```



based on Visser et. al. ICFP'98

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)
```

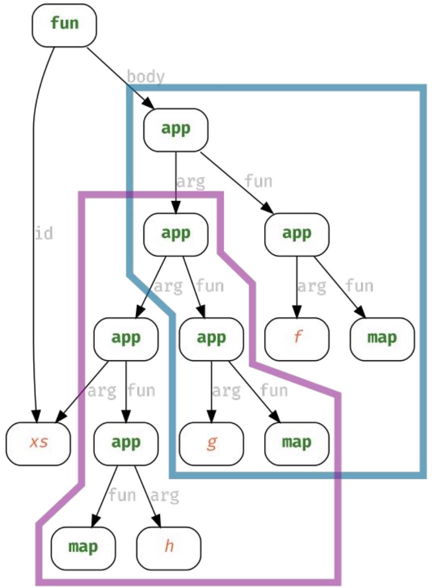
Repeat

```
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))))
```

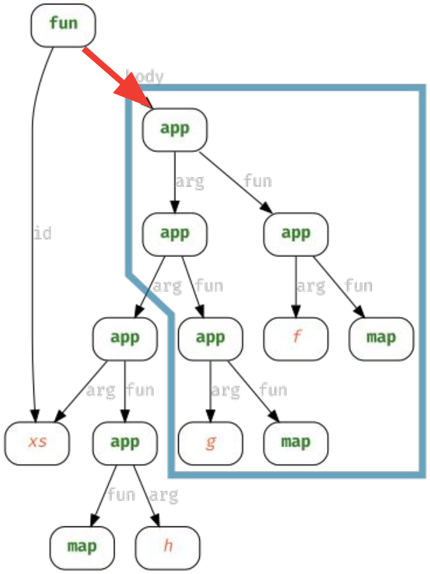
There are ***two possible locations*** for successfully applying the rule

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))))

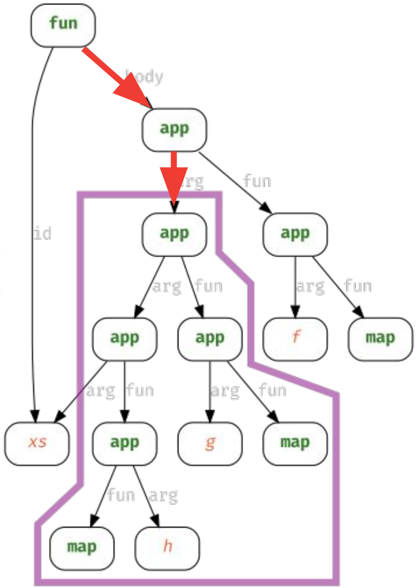
There are ***two possible locations*** for successfully applying the rule

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))))
```

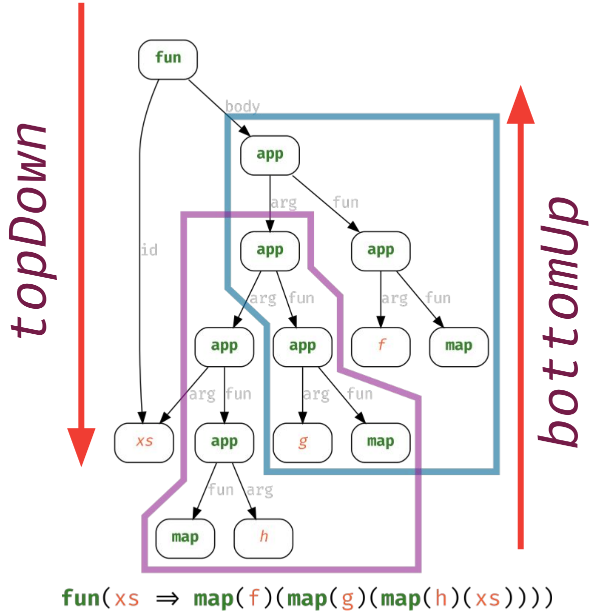
There are ***two possible locations*** for successfully applying the rule

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...

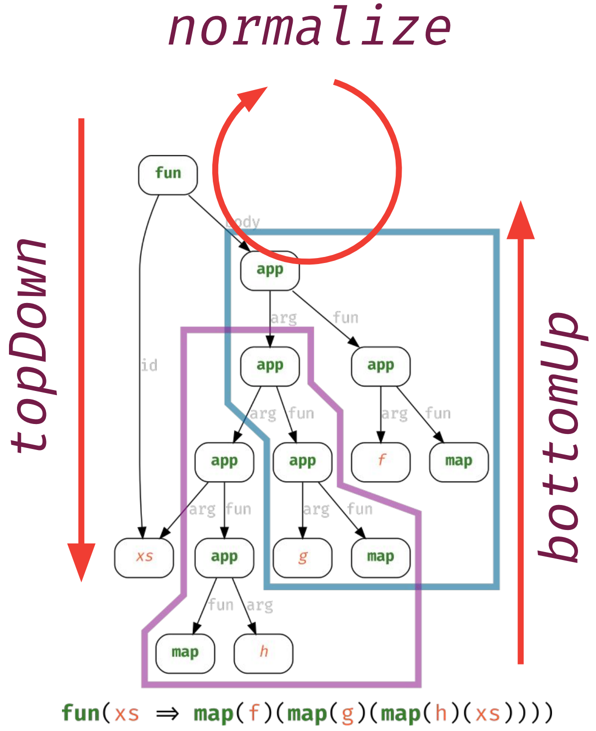
```
def topDown:  Traversal[Rise] = s => p => (s <+ one(topDown(s)))(p)
def bottomUp: Traversal[Rise] = s => p => (one(topDown(s)) <+ s)(p)
...
```

... and a strategy for normalization

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

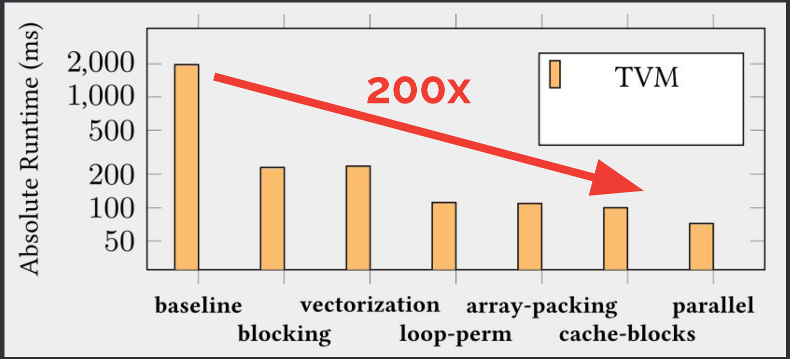
With these, we define normal-forms like βη-normal-form

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



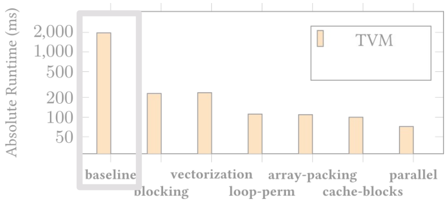
CASE STUDY

Implementing TVM's Scheduling Language



CASE STUDY

Optimizing Matrix Multiplication - Baseline



What to compute

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

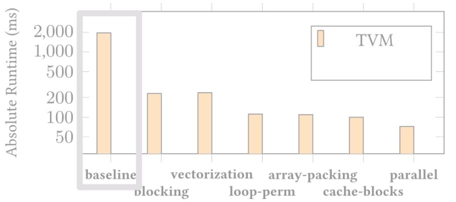
tvm

```
# 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)
```

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

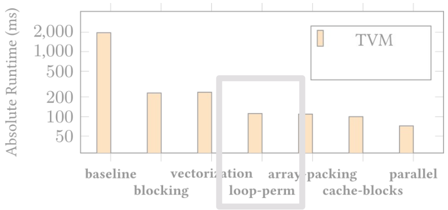
tvm

```
# 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)
```

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)
```

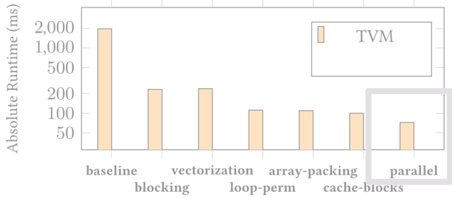
ELEVATE

```
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 - Array Packing

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

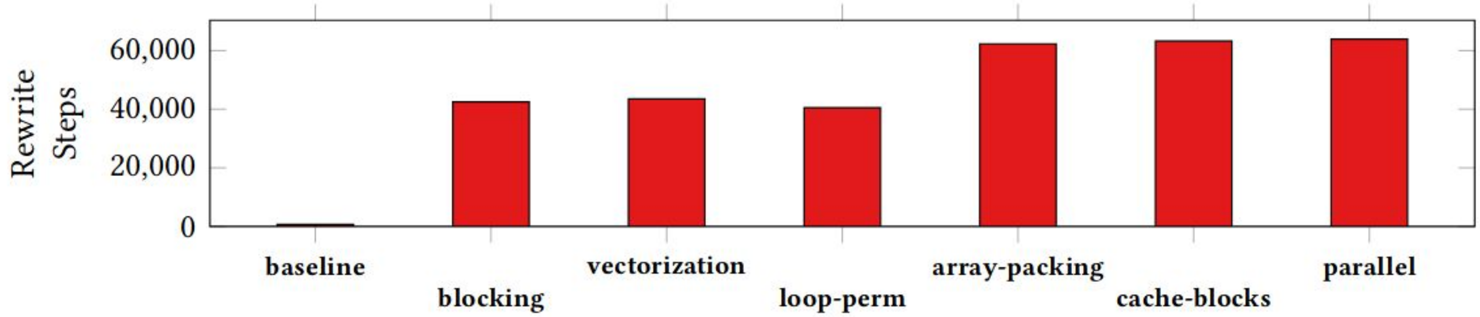
```
# Modified algorithm
bn = 32
k = tvm.reduce_axis((0, K), 'k')
A = tvm.placeholder((M, K), name='A')
B = tvm.placeholder((K, N), name='B')
pB = tvm.compute((N / bn, K, bn),
  lambda x, y, z: B[y, x * bn + z], name='pB')
C = tvm.compute((M,N), lambda x,y:
  tvm.sum(A[x,k] * pB[y//bn,k,
  tvm.indexmod(y,bn)], 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

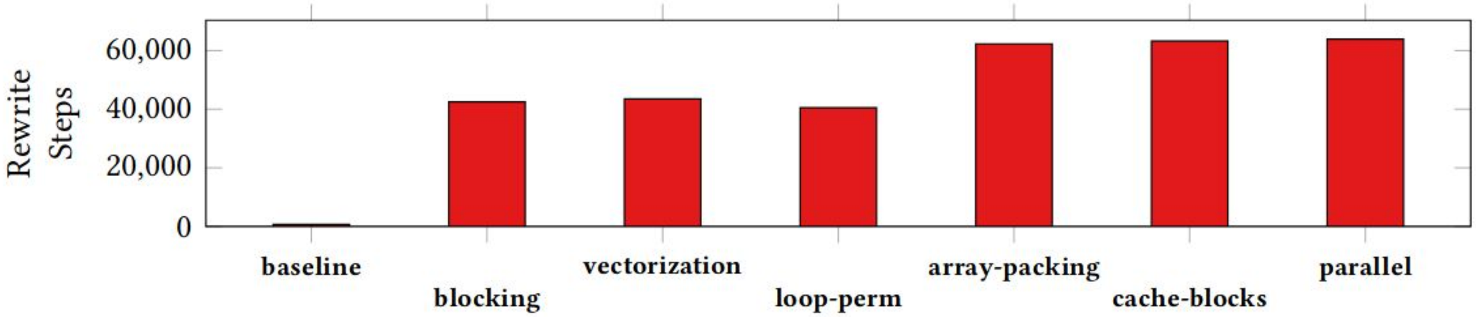
Number of successful rewrites steps



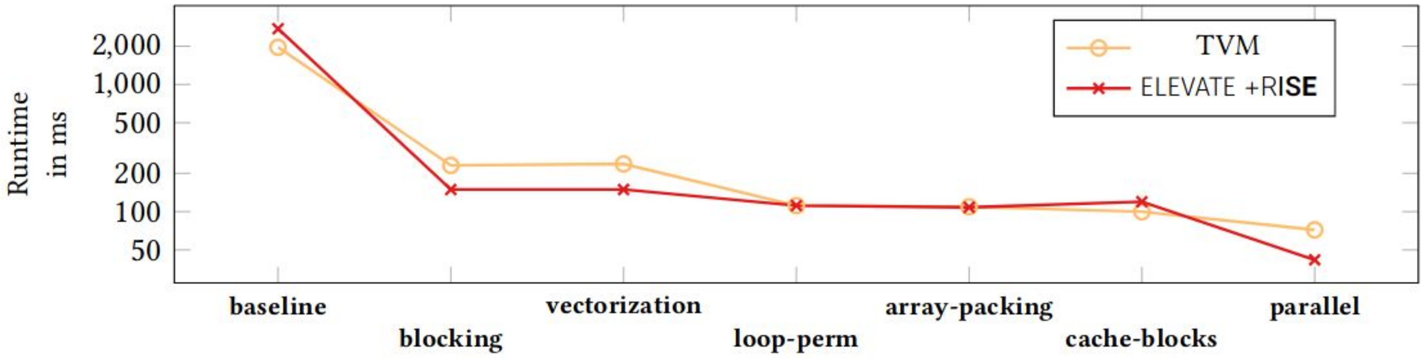
CASE STUDY

Counting Rewrite Steps and Measuring Performance

Number of successful rewrites steps



Performance of the generated code



ACHIEVING HIGH-PERFORMANCE THE *Functional* WAY

...is Open Source!

RISE
rise-lang.org/
github.com/rise-lang

ELEVATE
elevate-lang.org
github.com/elevate-lang

Artifacts Available
Artifacts Evaluated Functional

b.hagedorn@wwu.de
bastianhagedorn.github.io