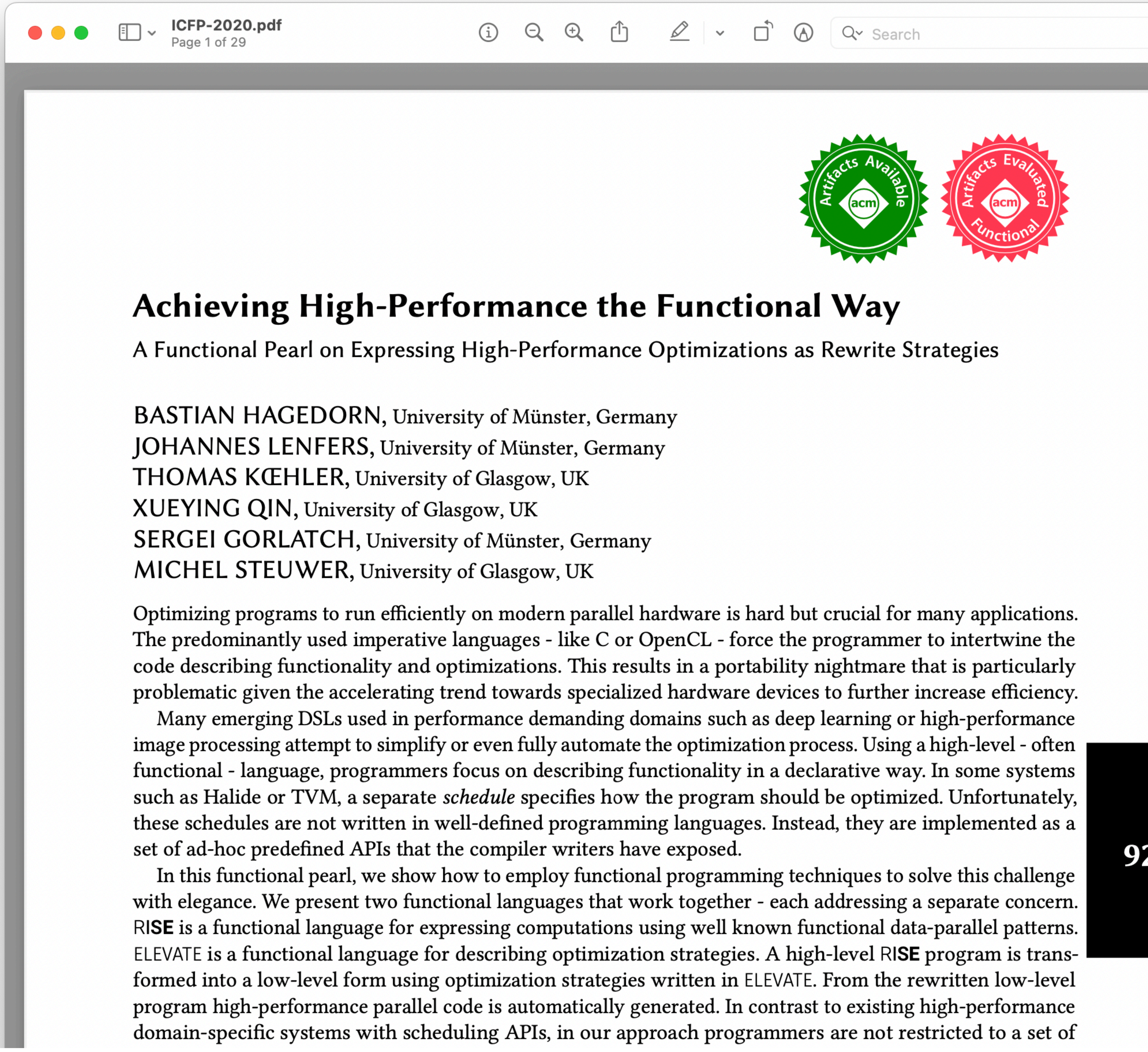
Achieving High-Performance the Functional Way

Expressing High-Performance Optimisations as Rewrite Strategies

Bastian Hagedorn, Johannes Lenfers, Thomas Kœhler, Xueying Qin, Sergei Gorlatch
and **Michel Steuwer**

WWU MÜNSTER · THE UNIVERSITY of EDINBURGH · University of Glasgow

- rejected from PLDI 2020
- published at [ICFP 2020]
- selected as 1 of 4 **ACM SIGPLAN Research Highlights** from 2020
- selected for publication as **Research Highlight** in an upcoming issue of the **Communications of the ACM**



Achieving High-Performance the Functional Way

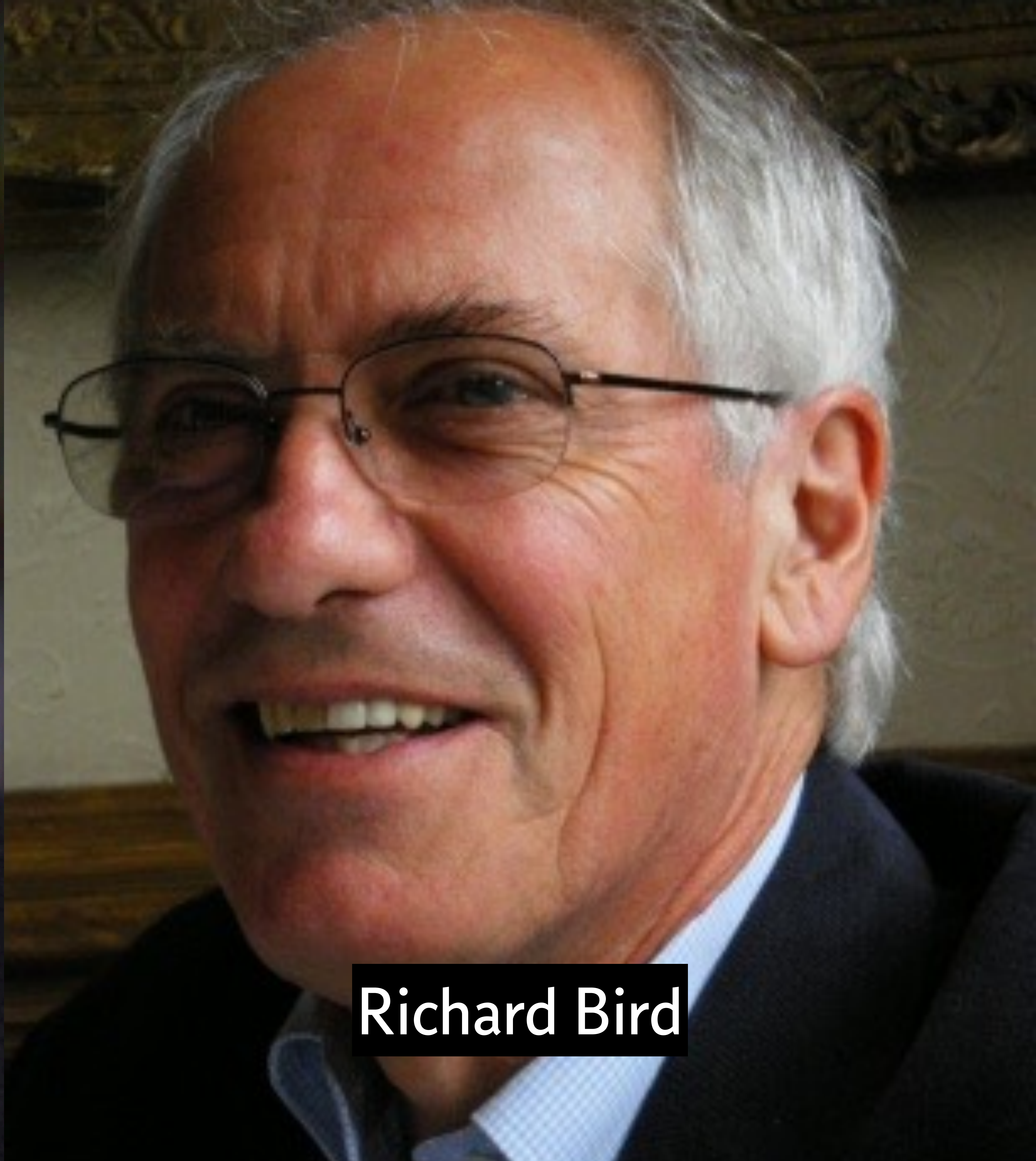
A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies

BASTIAN HAGEDORN, University of Münster, Germany
JOHANNES LENFERS, University of Münster, Germany
THOMAS KŒHLER, University of Glasgow, UK
XUEYING QIN, University of Glasgow, UK
SERGEI GORLATCH, University of Münster, Germany
MICHEL STEUWER, University of Glasgow, UK

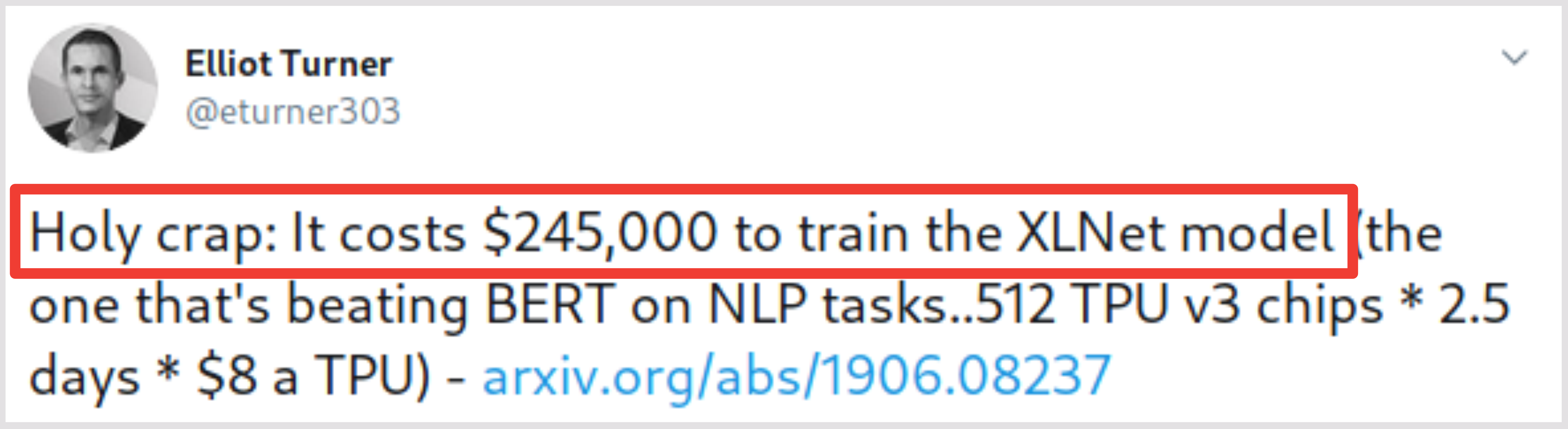
Optimizing programs to run efficiently on modern parallel hardware is hard but crucial for many applications. The predominantly used imperative languages - like C or OpenCL - force the programmer to intertwine the code describing functionality and optimizations. This results in a portability nightmare that is particularly problematic given the accelerating trend towards specialized hardware devices to further increase efficiency.

Many emerging DSLs used in performance demanding domains such as deep learning or high-performance image processing attempt to simplify or even fully automate the optimization process. Using a high-level - often functional - language, programmers focus on describing functionality in a declarative way. In some systems such as Halide or TVM, a separate *schedule* specifies how the program should be optimized. Unfortunately, these schedules are not written in well-defined programming languages. Instead, they are implemented as a set of ad-hoc predefined APIs that the compiler writers have exposed.

In this functional pearl, we show how to employ functional programming techniques to solve this challenge with elegance. We present two functional languages that work together - each addressing a separate concern. RISE is a functional language for expressing computations using well known functional data-parallel patterns. ELEVATE is a functional language for describing optimization strategies. A high-level RISE program is transformed into a low-level form using optimization strategies written in ELEVATE. From the rewritten low-level program high-performance parallel code is automatically generated. In contrast to existing high-performance domain-specific systems with scheduling APIs, in our approach programmers are not restricted to a set of


Eelco Visser

Richard Bird

HIGH-PERFORMANCE: *Why do we care?*



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 CO2 into the atmosphere (equivalent to driving a car around 11,000 miles)

RMANCE: *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

vay (using carbon as opposed to $$) of thinking
experiment: Training XLNet to convergence
round 4.9 metric tons of CO_2 into the atmosphere
t to driving a car around 11,000 miles)

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

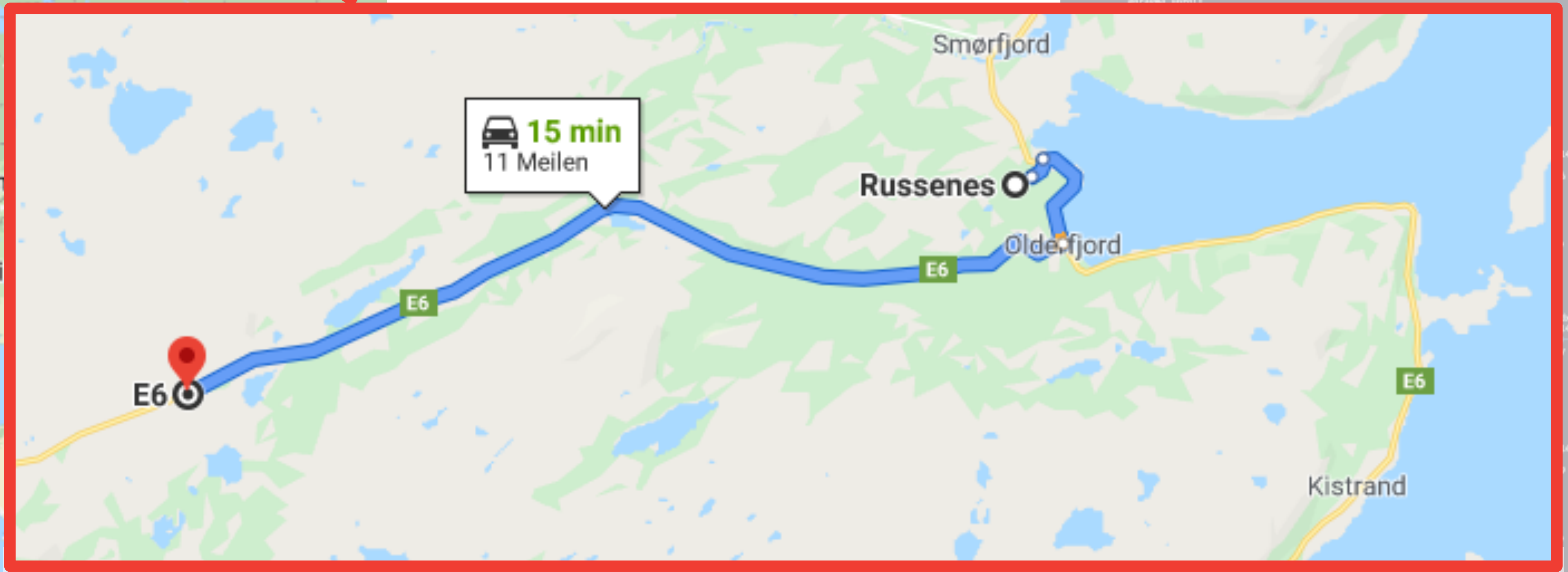
Achieving High-Performanc

```
__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

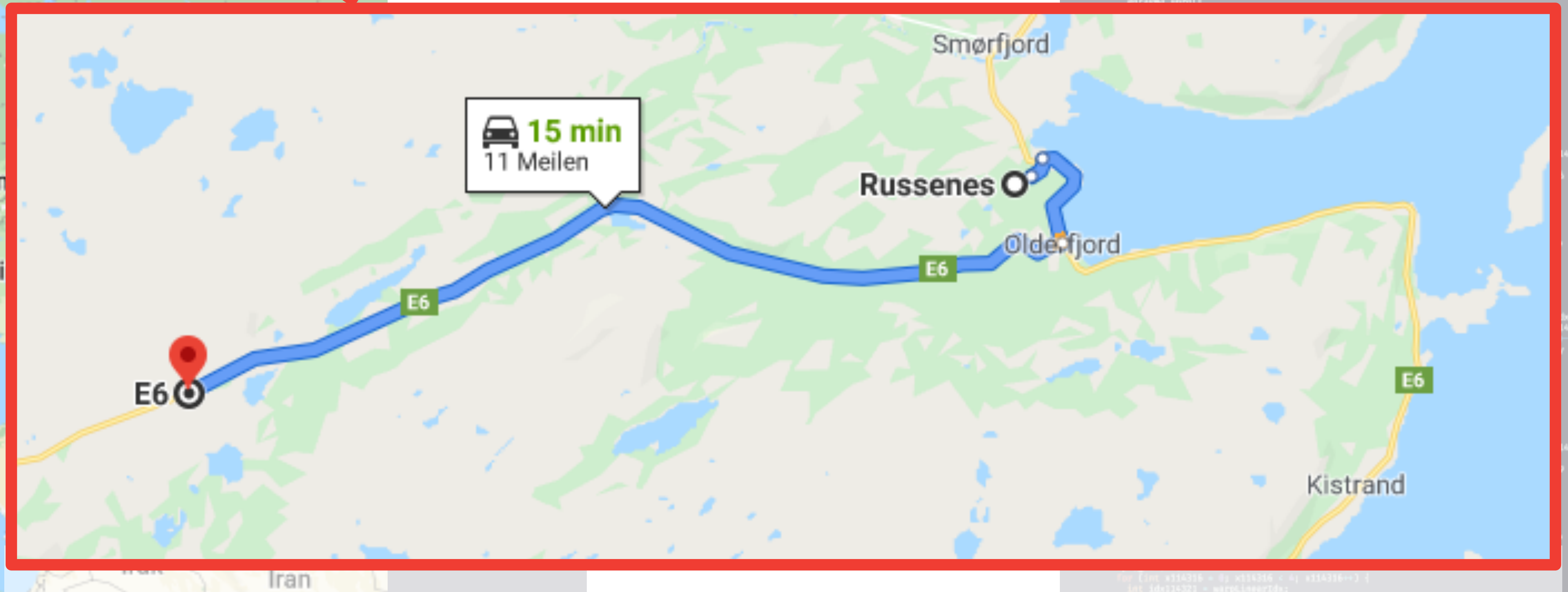
A… Performanc…



1000x CO2 Improvement

100-1000x performance

Optimized Matrix Multiplication



1000x CO2 Improvement

100-1000x performance

30x lines of code

time-intensive + error-prone

Optimized Matrix Multiplication

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')
```

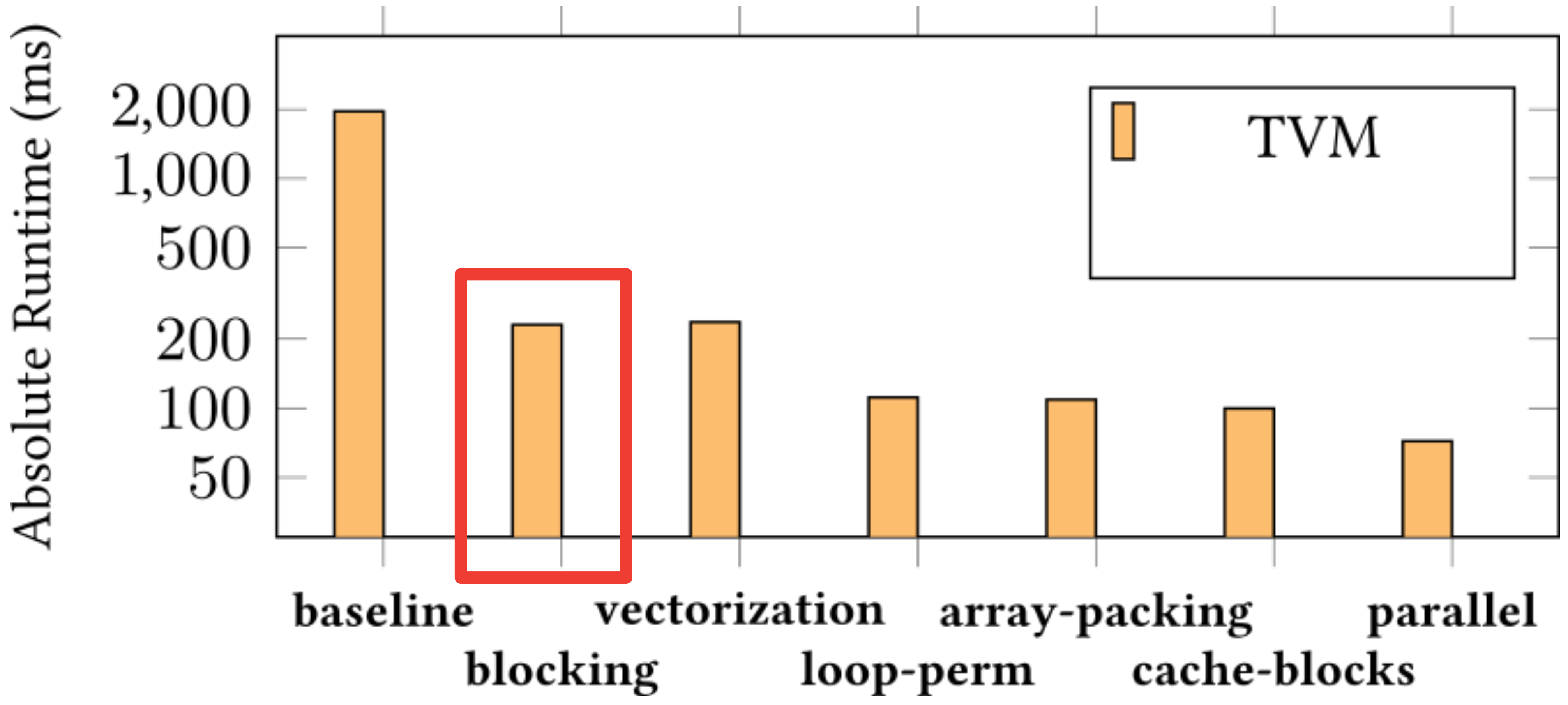
Performance Engineer

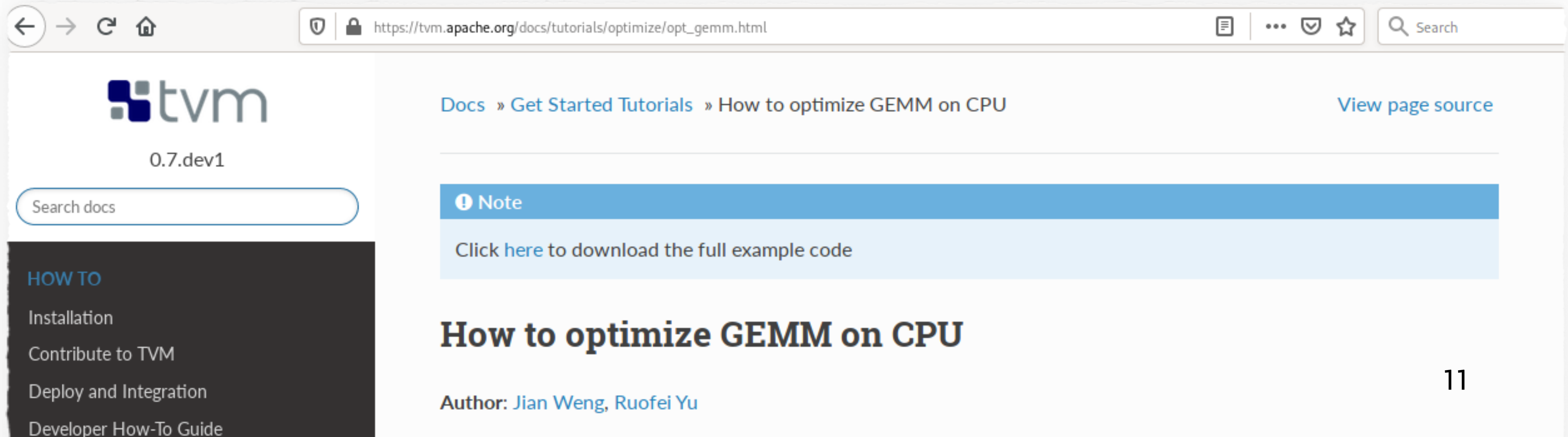
Schedule

```
# 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)
```

tvm

Compiler





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')
```

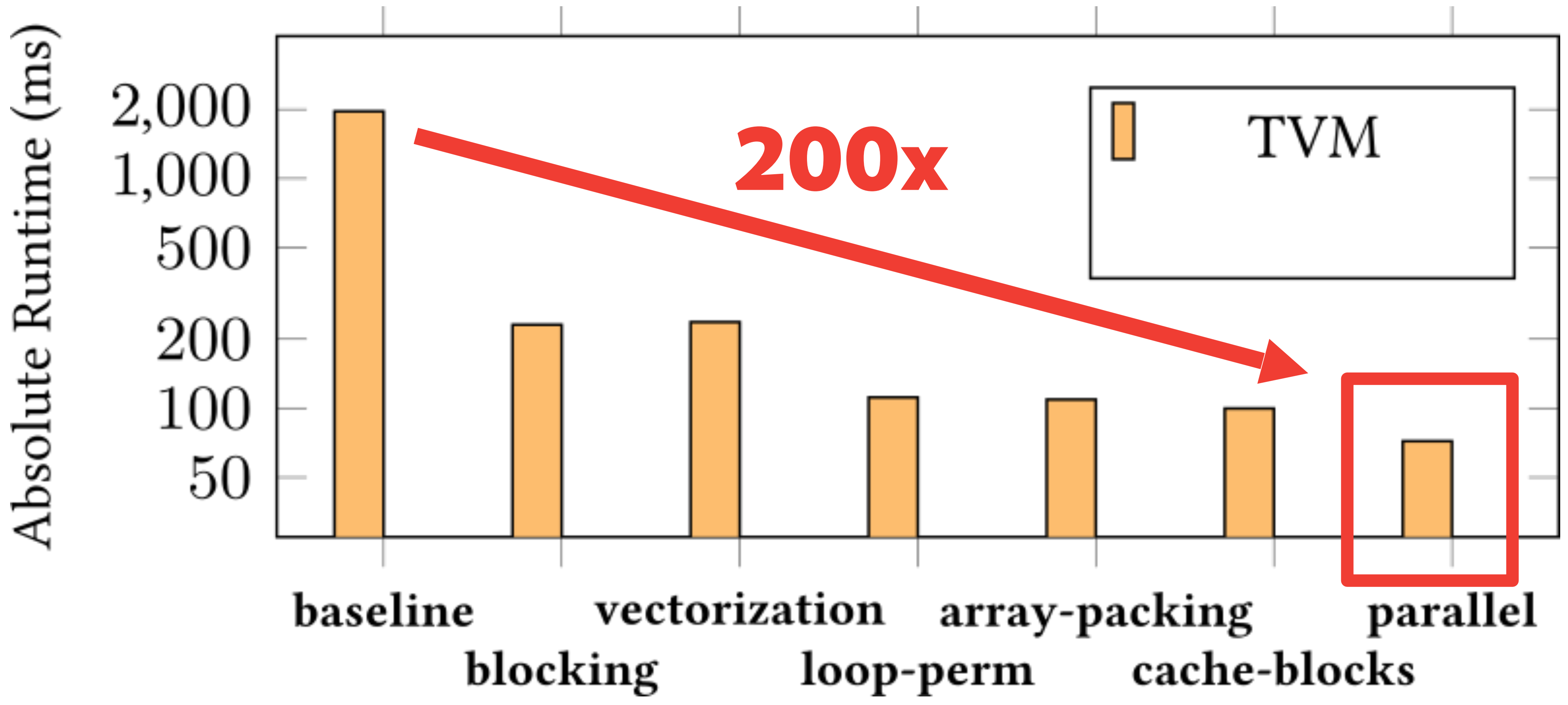
Performance Engineer

```
# "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



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

Compilers with scheduling APIs

Halide

tvm

Tiramisu-Compiler / **tiramisu**

Fireiron NVIDIA

Program

Algorithm

Domain-Scientist

```
// 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);
```

Performance Engineer

```
// 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

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

Program

```
// 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;
```

```
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);
```

```
    .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);
```

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

No clear separation

Program

```
// 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;
```

```
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);
```

```
    .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);
```

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

Hinders reuse

Program

```
// 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;
```

```
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);
```

```
    .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);
```

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

Program

```
// 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);
```

Hinders reuse

Not well defined semantics

```
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);
```

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

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

Hinders reuse

Hinders understanding

Program

```
// 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;
```

```
.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);
```

```
    .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);
```

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

Hinders reuse

Hinders understanding

Not well-d... ...ntics

Only fixed built-in optimisations

Program

```
// 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;
```

```
.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);
```

```
   .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);
```

Optimisation Schedule

Halide compiler

Optimised Code

Problems with Scheduling APIs

N... ...ation

Hinders reuse

Not well-d... ...ntics

Hinders understanding

Only fix... ...isations

No extensibility

Program

```
// 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;
```

```
.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);
```

```
   .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);
```

Optimisation Schedule

Halide compiler

`Optimised Code`

Problems with Scheduling APIs

Program

```
// 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;
```

Hinders reuse

N... ...ation

Hinders understanding

Not well d... ...ntics

No extensibility

Only fix... ...isations

```
.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);
```

```
    .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];
...
A.in().in().compute_at(prod, rxi).vectorize(Ax, vec_size)
      .split(Ax, xo, xi, warp_size).gpu_lanes(xi)
      .unroll(xo).unroll(Ay);
```

Halide compiler

We should aim for more principled ways to describe and apply optimisations

...Code

Optimisation Schedule

The Need for a Principled Way to Separate, Describe and Apply Optimizations

Our goals:

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.

The Need for a Principled Way to Separate, Describe and Apply Optimizations

Our goals:

1. *Separate concerns*

   Computations should be expressed at a high abstraction level only.

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.

Computational patterns, but also especially strategies, should have a precise, well-defined semantics allowing reasoning about them;

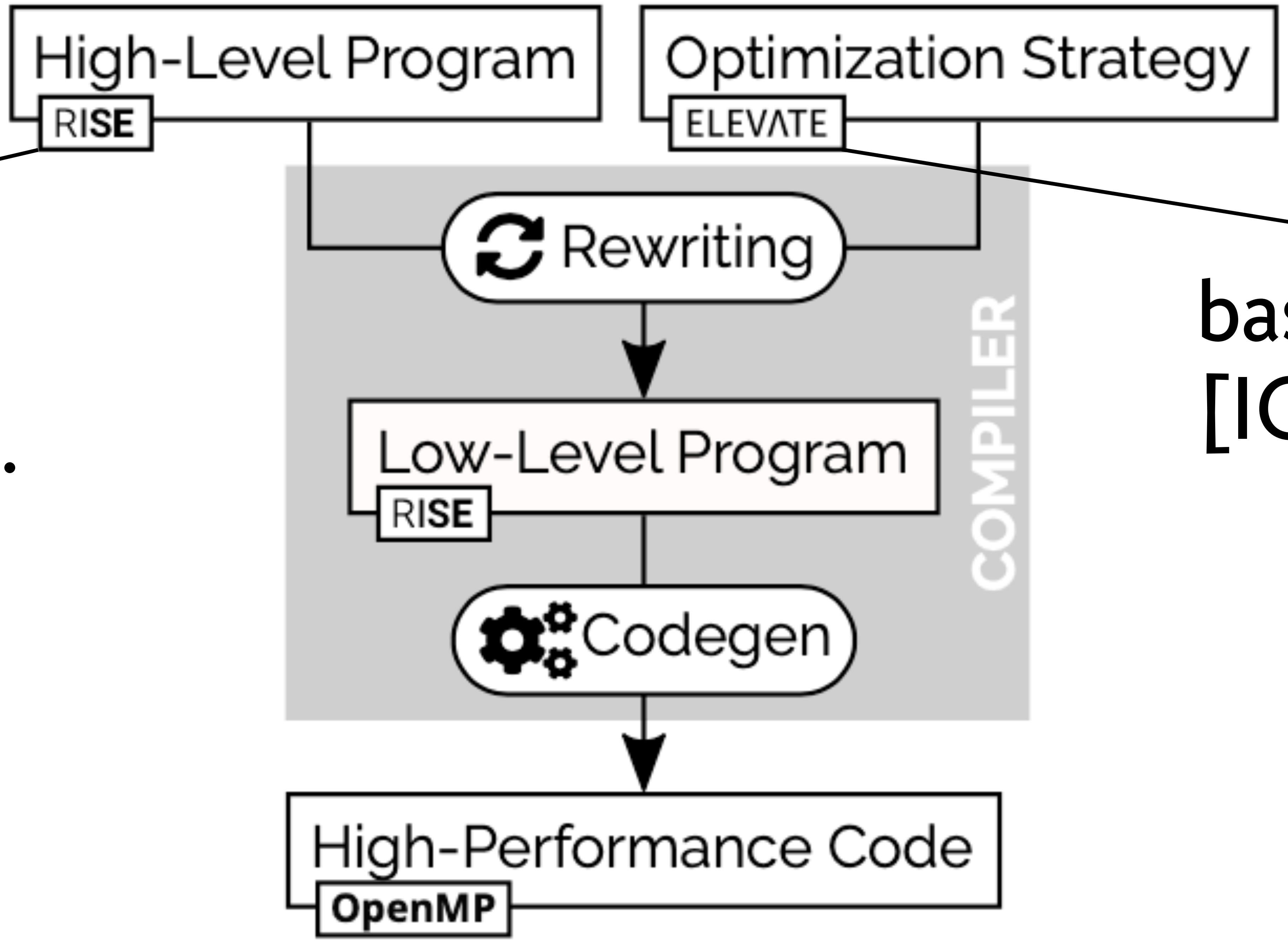
5. *Be explicit*

   Implicit default behavior should be avoided to empower users to be in control.

Achieving High-Performance the Functional Way

```
// Matrix 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 : M.K.float, fun(b : K.N.float,
  a |> map(fun(arow,                          // iterating over M
    transpose(b) |> map(fun(bcol,             // iterating over N
      dot(arow)(bcol) )))) ) )                // iterating over K
```

```
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)
```



based on Lift
[ICFP 2015] by Steuwer et. al.

based on Stratego
[ICFP 1998] by Visser et. al.

ELEVATE **A Language for Describing Optimisation Strategies**

- A **Strategy** encodes a program transformation as a function:

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

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

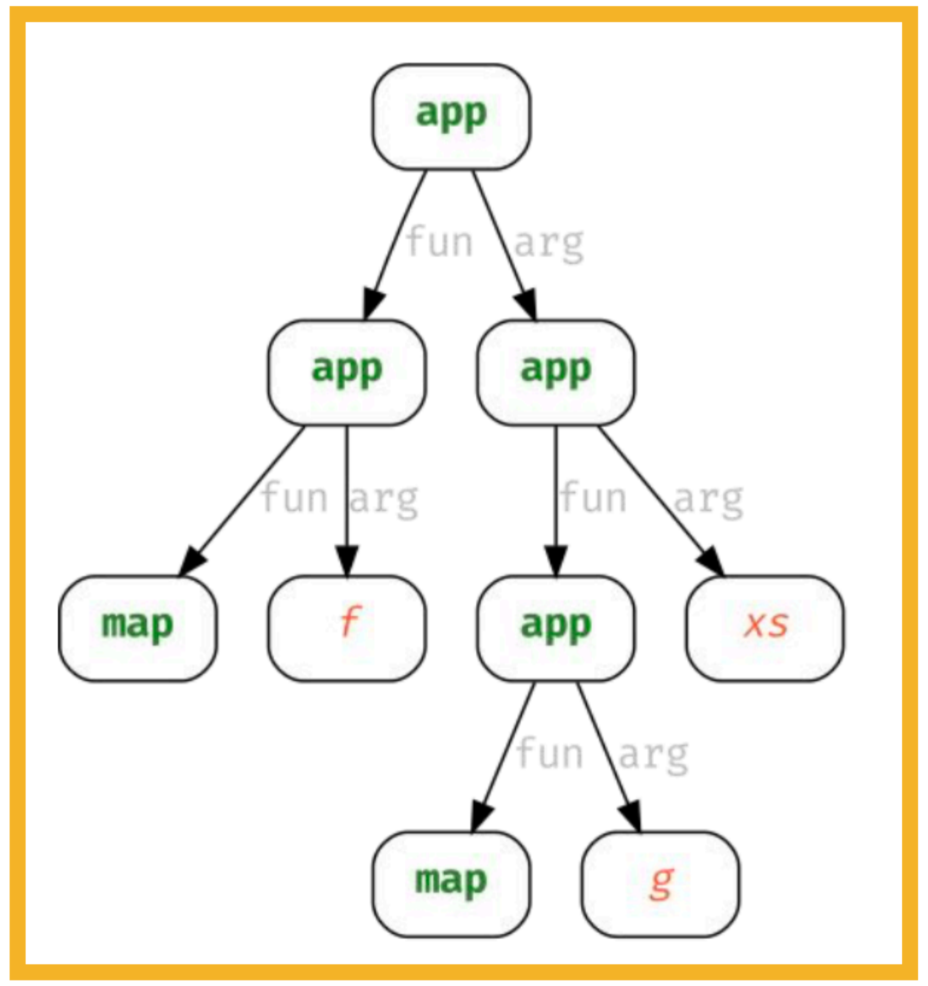
```
RewriteResult[P] = Success[P](p: P)
                 | Failure[P](s: Strategy[P])
```

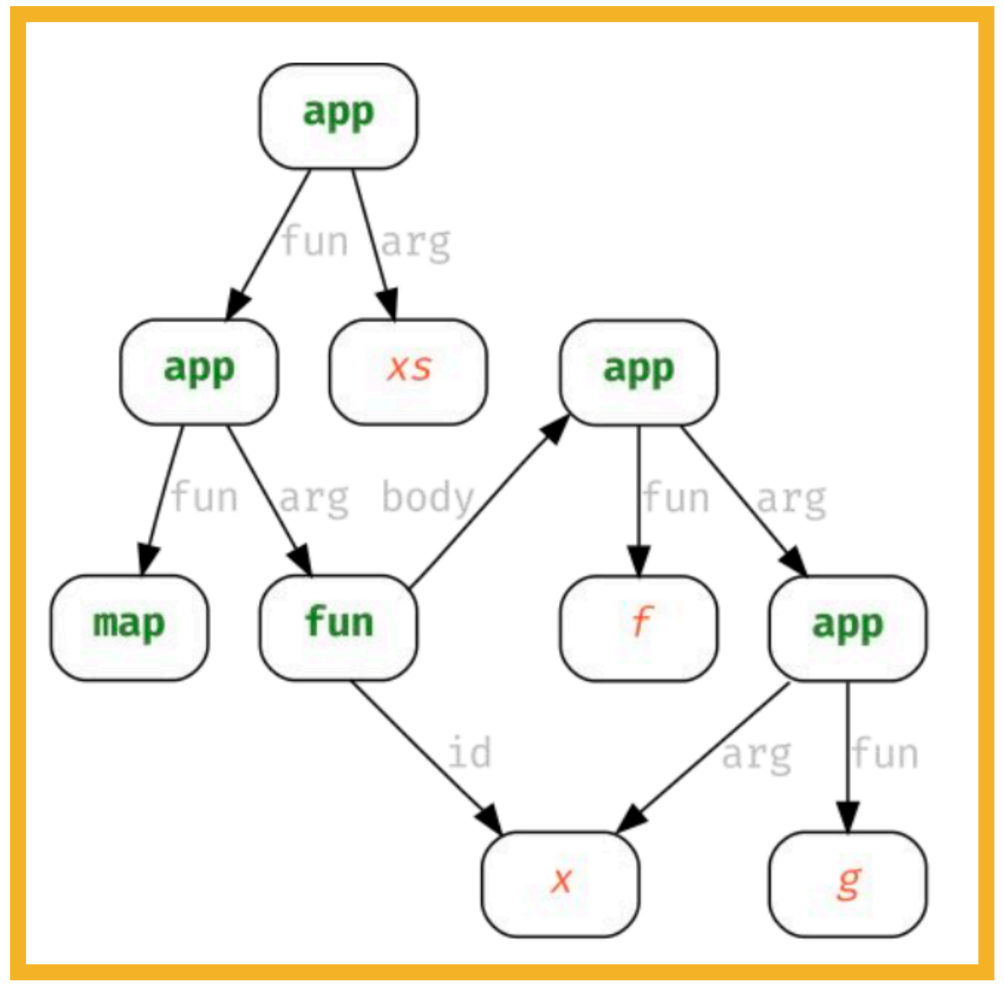
Rewrite Rules in ELEVATE

- *Rewrite rules* are basic strategies

$$\texttt{map(f) << map(g)} \rightsquigarrow \texttt{map(f << g)}$$

```
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)
}
```

mapFusion(

) = 

Combinators in ELEVATE

- Building more complex strategies from simpler once

- Sequential Composition (;)

```
def seq[P]: Strategy[P] ⇒ Strategy[P] ⇒ Strategy[P] =
      fs ⇒ ss ⇒ p ⇒ fs(p).flatMapSuccess(ss)
```

- Left Choice (<+)

```
def lChoice[P]: Strategy[P] ⇒ Strategy[P] ⇒ Strategy[P] =
      fs ⇒ ss ⇒ p ⇒ fs(p).flatMapFailure(_ ⇒ ss(p))
```

- Try

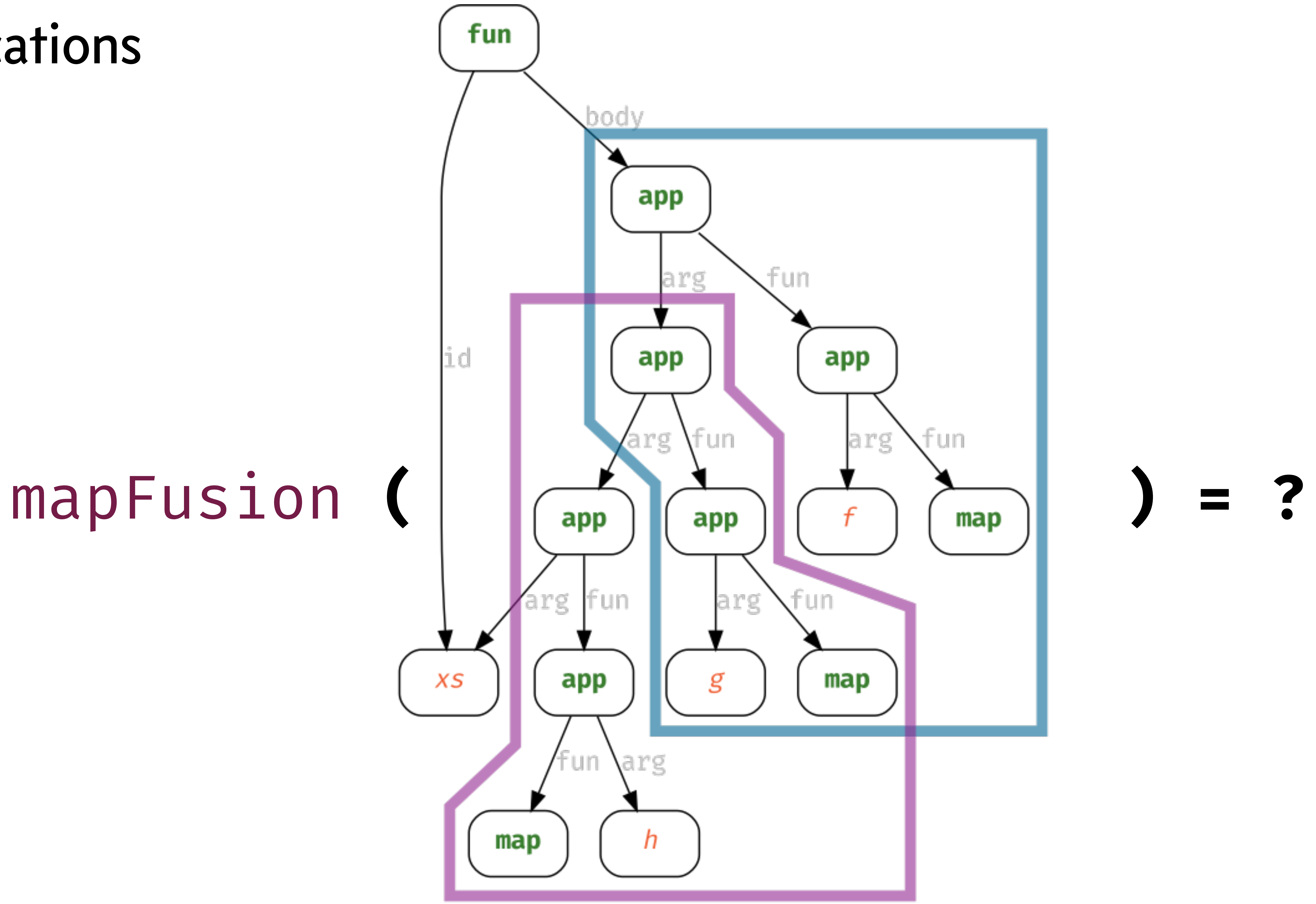
```
def try[P]: Strategy[P] ⇒ Strategy[P] =
      s ⇒ p ⇒ (s <+ id)(p)
```

- Repeat

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

Traversals in ELEVATE

- Describing Precise Locations

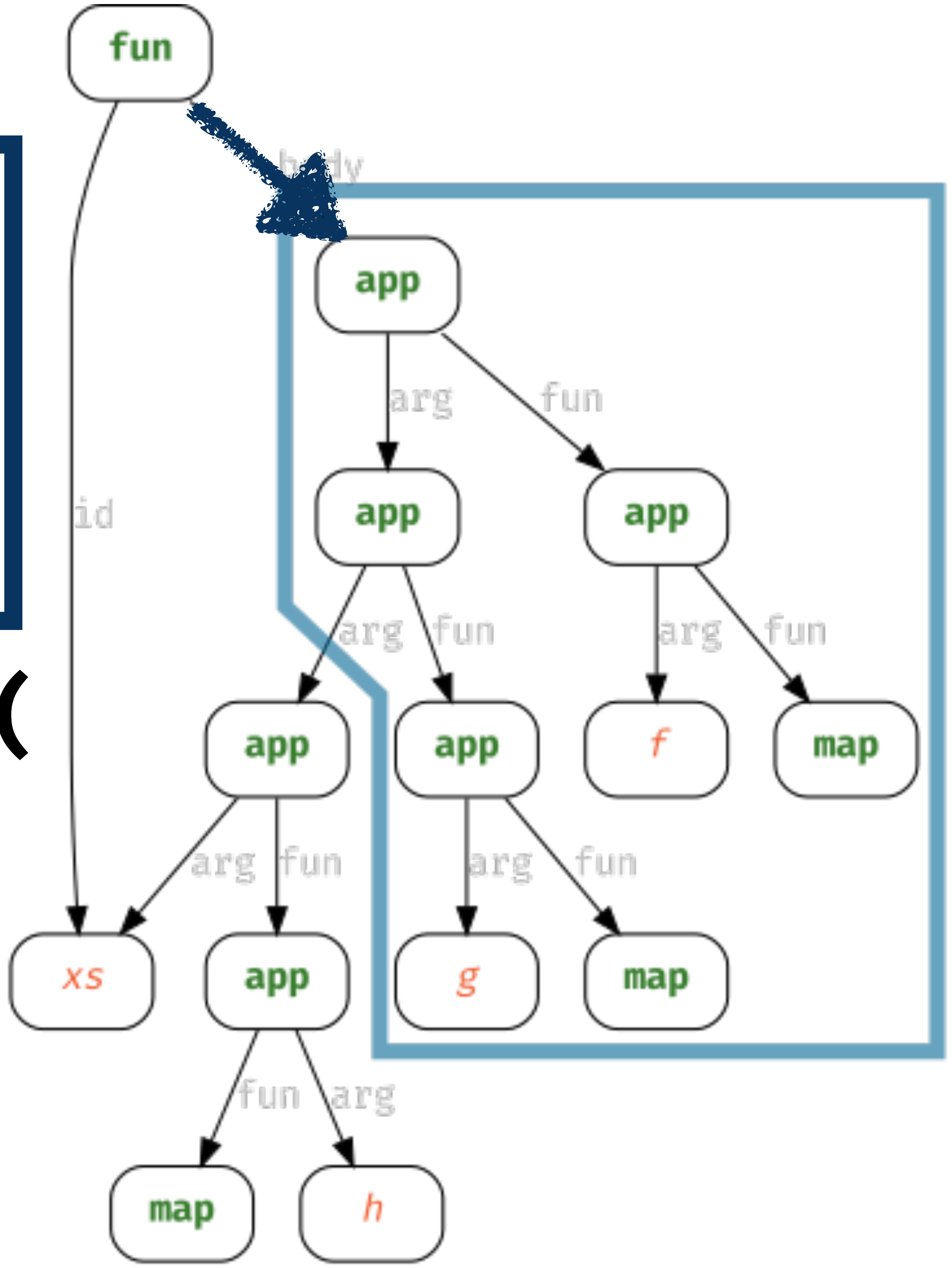



```
threemaps =fun(xs ⇒ map(f)(map(g)(map(h)(xs))))
```

Traversals in ELEVATE

- Describing Precise Locations

```
def body: Strategy[Rise] ⇒ Strategy[Rise] =
 s ⇒ p ⇒ p match {
  case fun(x,b) ⇒ s(b).mapSuccess(nb ⇒
fun(x,nb))
  case _ ⇒ Failure( body(s) )
}
```

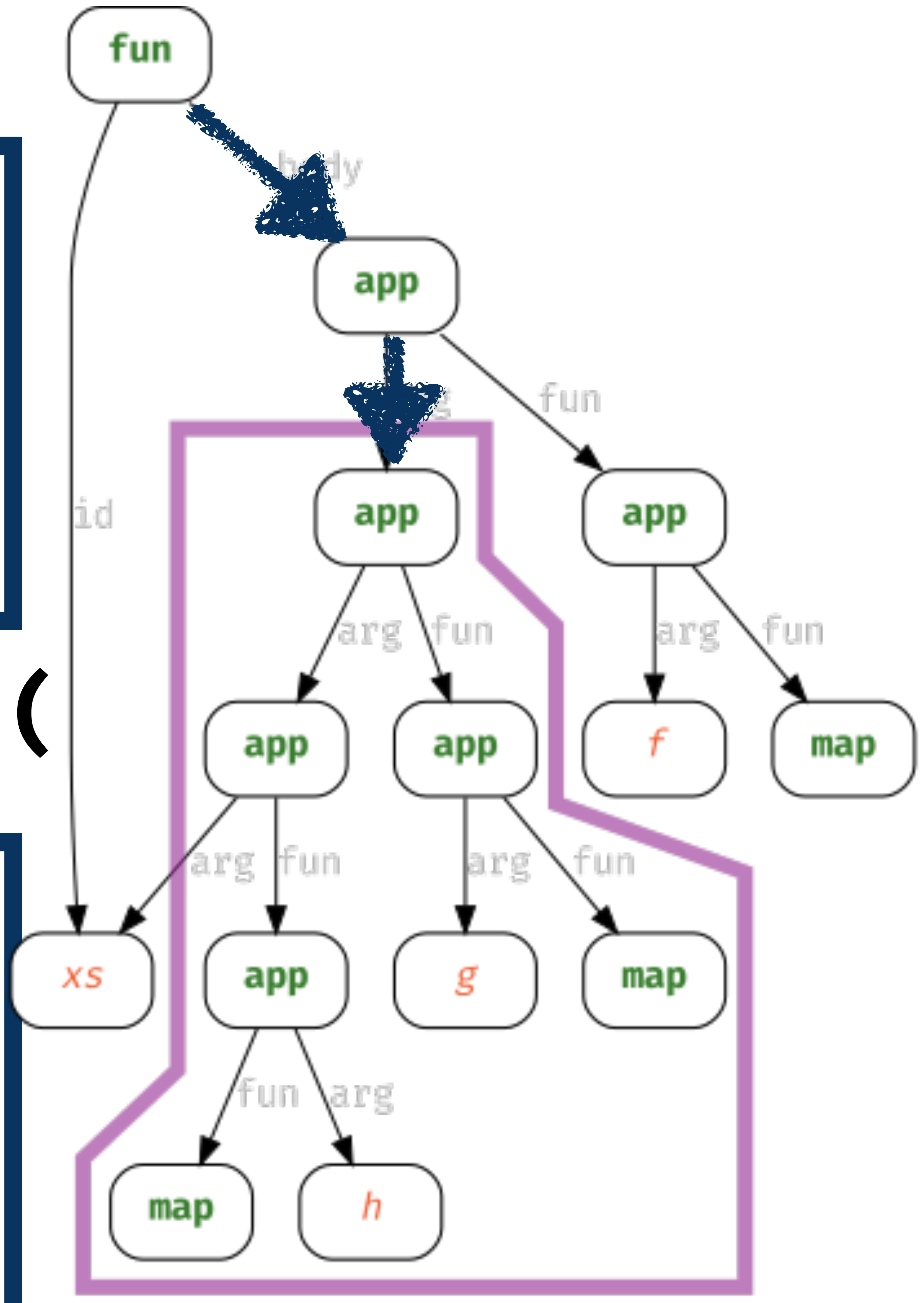
`body(mapFusion) (`  `) = ?`

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

Traversals in ELEVATE

- Describing Precise Locations

```
def body: Strategy[Rise] ⇒ Strategy[Rise] =
 s ⇒ p ⇒ p match {
  case fun(x,b) ⇒ s(b).mapSuccess(nb ⇒
fun(x,nb))
  case _ ⇒ Failure( body(s) )
}
```

body(argument(mapFusion)) () = ?


```
def argument: Strategy[Rise] ⇒ Strategy[Rise] =
 s ⇒ p ⇒ p match {
  case app(f,a) ⇒ s(a).mapSuccess(na ⇒
app(f,na))
  case _ ⇒ Failure( argument(s) )
}
```

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

Complex Traversals + Normalization in ELEVATE

- With three basic generic traversals

```
type Traversal[P] = Strategy[P] => Strategy[P]
def all[P]: Traversal[P];    def one[P]: Traversal[P];    def some[P]: Traversal[P]
```

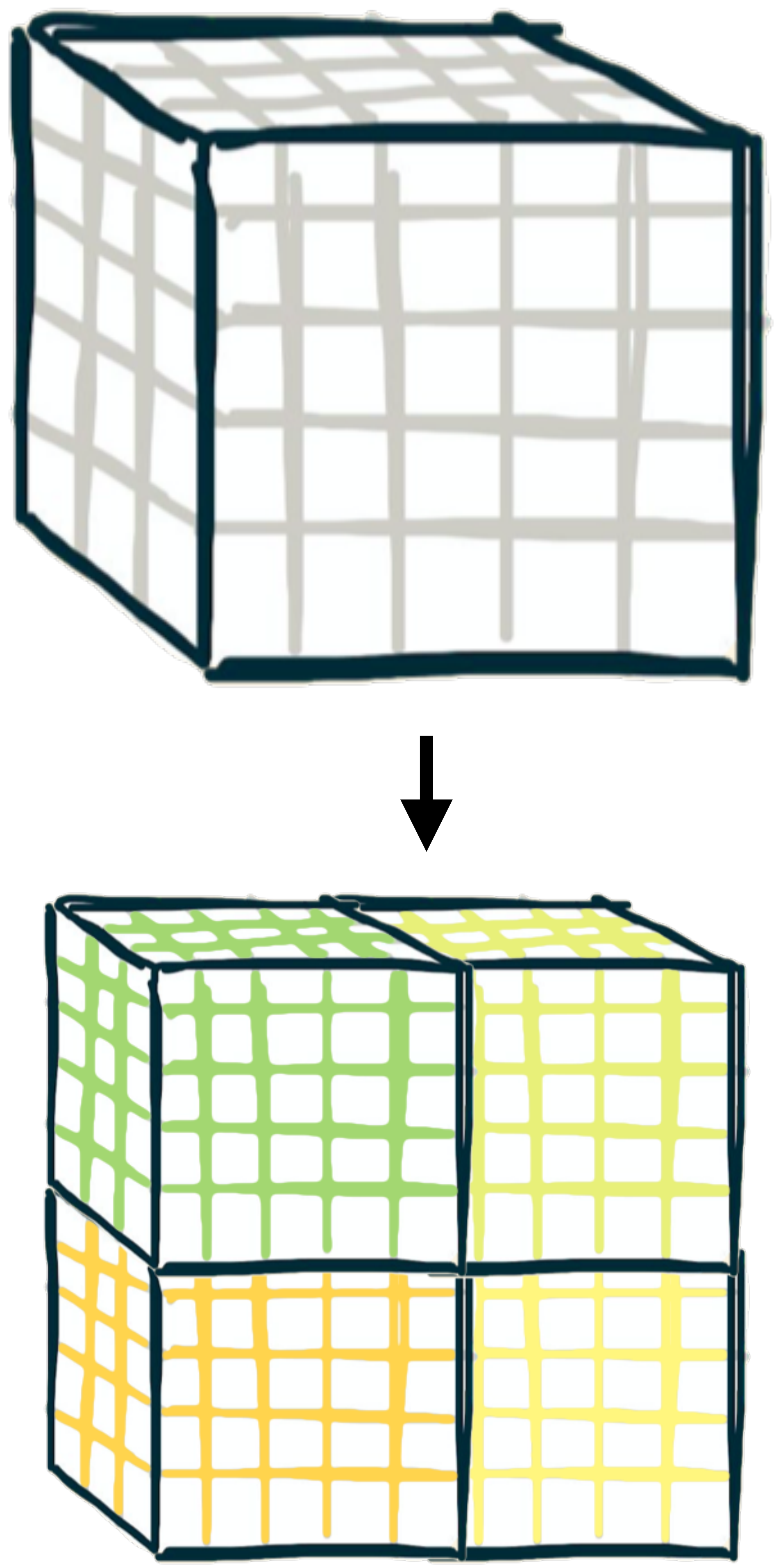
- we define more complex traversals:

```
def     topDown[P]: Traversal[P] = s => p => (s <+ one(topDown(s)))(p)
def    bottomUp[P]: Traversal[P] = s => p => (one(bottomUp(s)) <+ s)(p)
def  allTopDown[P]: Traversal[P] = s => p => (s ‘;‘ all(allTopDown(s)))(p)
def allBottomUp[P]: Traversal[P] = s => p => (all(allBottomUp(s)) ‘;‘ s)(p)
def      tryAll[P]: Traversal[P] = s => p => (all(tryAll(try(s))) ‘;‘ try(s))(p)
```

- With these traversals we define normal forms, e.g. $\beta\eta$-normal-form:

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

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

Complex optimisations defined as strategies

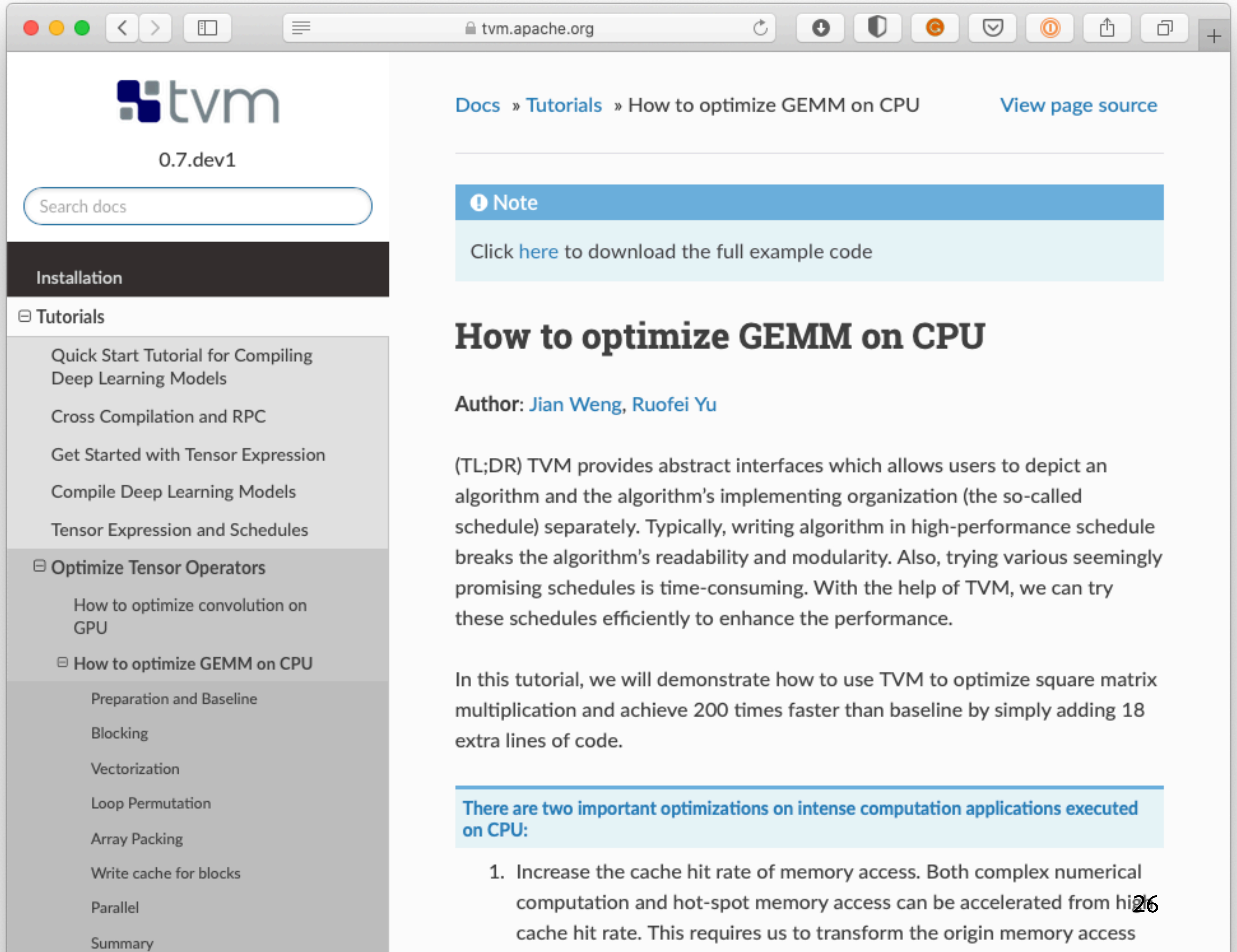


```
def tile: Int → Int → Strategy =
  (dim) ⇒ (n) ⇒ dim match {
    case 1 = function(splitJoin(n))

    case 2 = fmap(function(splitJoin(n))) ;
             function(splitJoin(n)) ; interchange(2)
    case i = fmap(tile(dim-1, n)) ;
             function(splitJoin(n)) ; interchange(n)
  }
```

Tiling defined as composition of rewrites not a built-in!

Case Study: Implementing TVM's Scheduling API

- We attempt to express the same optimizations described in the TVM tutorial:


tvm.apache.org

tvm
0.7.dev1

Search docs

Installation
⊟ Tutorials
Quick Start Tutorial for Compiling Deep Learning Models
Cross Compilation and RPC
Get Started with Tensor Expression
Compile Deep Learning Models
Tensor Expression and Schedules
⊟ Optimize Tensor Operators
How to optimize convolution on GPU
⊟ How to optimize GEMM on CPU
Preparation and Baseline
Blocking
Vectorization
Loop Permutation
Array Packing
Write cache for blocks
Parallel
Summary

Docs » 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

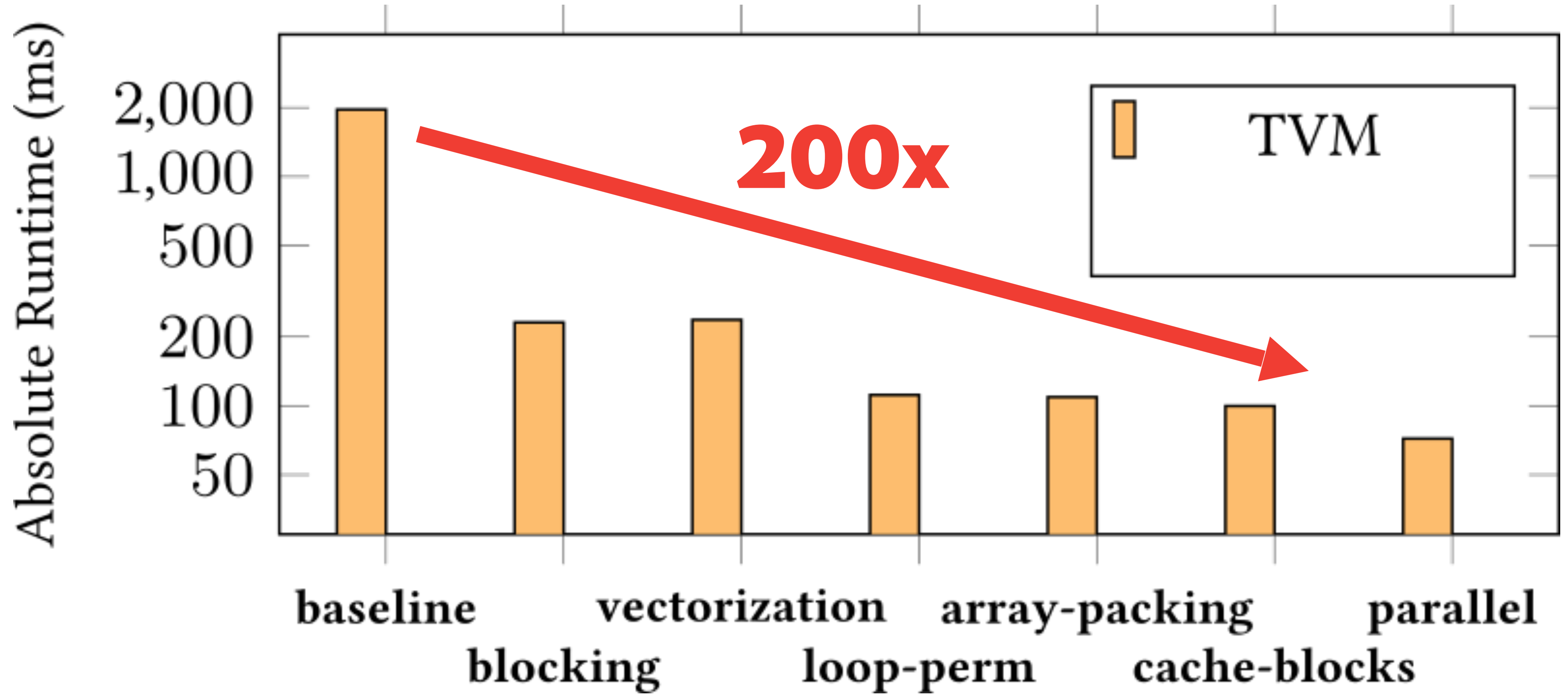
Author: Jian Weng, Ruofei Yu

(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.

There are two important optimizations on intense computation applications executed on CPU:

1. Increase the cache hit rate of memory access. Both complex numerical computation and hot-spot memory access can be accelerated from high cache hit rate. This requires us to transform the origin memory access



Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

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)
```

Baseline Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

Clear separation of concerns

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

Be explicit

Enable composability

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 behavior

Baseline Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE **Strategies**

ELEVATE

```
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)
```

tvm

```
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)
```

Loop Permutation with blocking Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE **Strategies**

ELEVATE

tvm

Facilitate reuse

User-defined vs. build 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)
```

```
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)
```

No clear separation of concerns

Loop Permutation with blocking Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

ELEVATE

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

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

val arrayPacking = packB ‘;;‘ loopPerm
(arrayPacking ‘;‘ lowerToC )(mm)
```

tvm

```
# 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)
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0], C.op.axis[1], bn, bn)
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)
x, y, z        = s[pB].op.axis
s[pB].vectorize(z)
s[pB].parallel(x)
```

Array Packing Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

Clear separation of concerns vs No clear separation of concerns

ELEVATE

tvm

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

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

val arrayPacking = packB ‘;;‘ loopPerm
(arrayPacking ‘;‘ lowerToC )(mm)
```

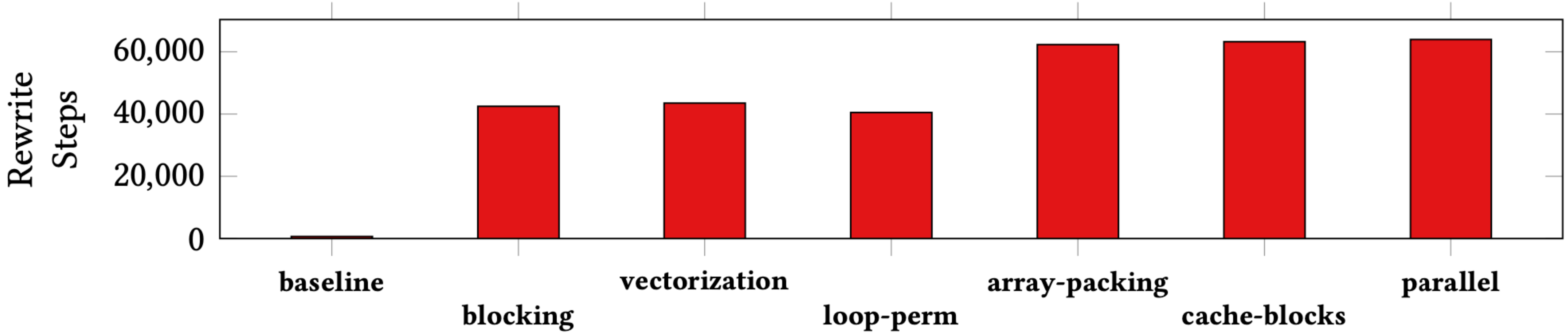
Facilitate reuse

```
# 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)
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0], C.op.axis[1], bn, bn)
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)
x, y, z         = s[pB].op.axis
s[pB].vectorize(z)
s[pB].parallel(x)
```

Array Packing Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

Number of successful rewrite steps

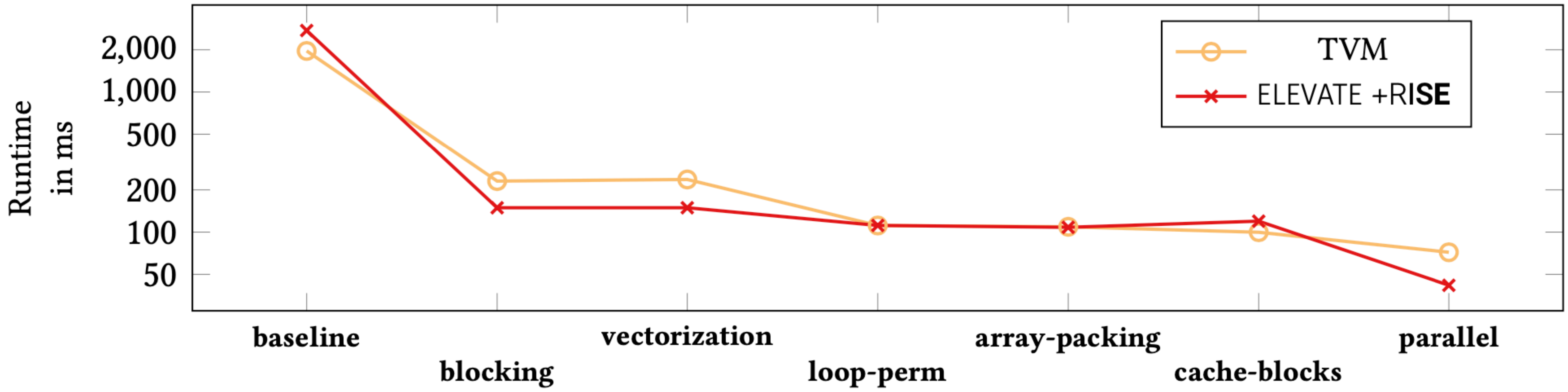


Rewriting took less than 2 seconds with our unoptimised implementation

Rewrite based approach scales to complex optimizations

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

Performance of generated code



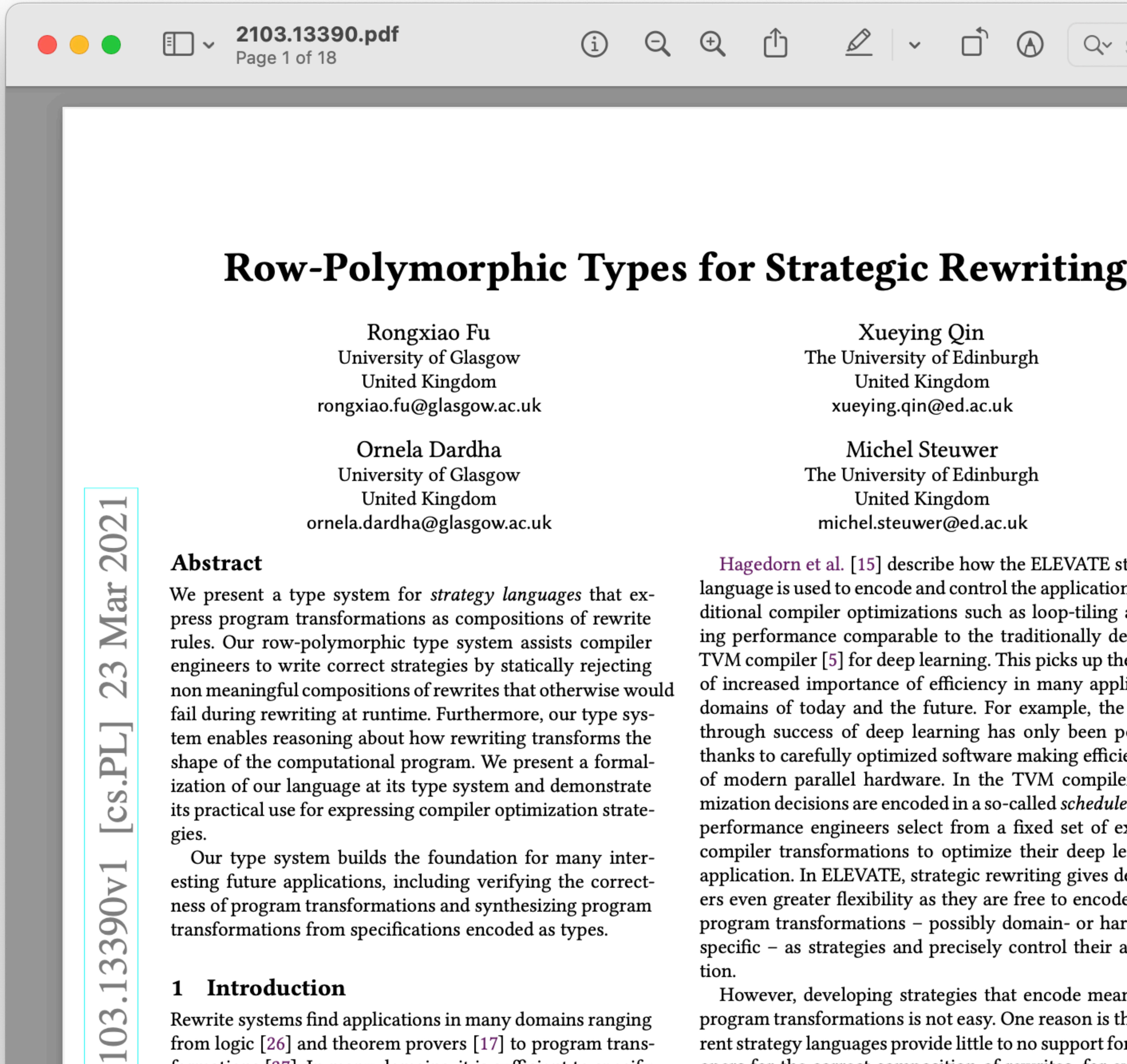
Competitive performance compared to TVM compiler

Types for ELEVATE?

- Can we build a type system for ELEVATE to statically reject bad compositions of rewrites?

Ongoing work using row-polymorphic types for this.

Preliminary result in an arXiv paper:
https://arxiv.org/abs/2103.13390


2103.13390.pdf
Page 1 of 18

arXiv:2103.13390v1 [cs.PL] 23 Mar 2021

Row-Polymorphic Types for Strategic Rewriting

Rongxiao Fu
University of Glasgow
United Kingdom
rongxiao.fu@glasgow.ac.uk

Xueying Qin
The University of Edinburgh
United Kingdom
xueying.qin@ed.ac.uk

Ornela Dardha
University of Glasgow
United Kingdom
ornela.dardha@glasgow.ac.uk

Michel Steuwer
The University of Edinburgh
United Kingdom
michel.steuwer@ed.ac.uk

Abstract

We present a type system for *strategy languages* that express program transformations as compositions of rewrite rules. Our row-polymorphic type system assists compiler engineers to write correct strategies by statically rejecting non meaningful compositions of rewrites that otherwise would fail during rewriting at runtime. Furthermore, our type system enables reasoning about how rewriting transforms the shape of the computational program. We present a formalization of our language at its type system and demonstrate its practical use for expressing compiler optimization strategies.

Our type system builds the foundation for many interesting future applications, including verifying the correctness of program transformations and synthesizing program transformations from specifications encoded as types.

1 Introduction

Rewrite systems find applications in many domains ranging from logic [26] and theorem provers [17] to program trans-

Hagedorn et al. [15] describe how the ELEVATE st
language is used to encode and control the application
ditional compiler optimizations such as loop-tiling
ing performance comparable to the traditionally de
TVM compiler [5] for deep learning. This picks up the
of increased importance of efficiency in many appl
domains of today and the future. For example, the
through success of deep learning has only been p
thanks to carefully optimized software making effici
of modern parallel hardware. In the TVM compile
mization decisions are encoded in a so-called *schedule*
performance engineers select from a fixed set of e
compiler transformations to optimize their deep le
application. In ELEVATE, strategic rewriting gives d
ers even greater flexibility as they are free to encode
program transformations – possibly domain- or har
specific – as strategies and precisely control their a
tion.

However, developing strategies that encode mean
program transformations is not easy. One reason is th
rent strategy languages provide little to no support fo

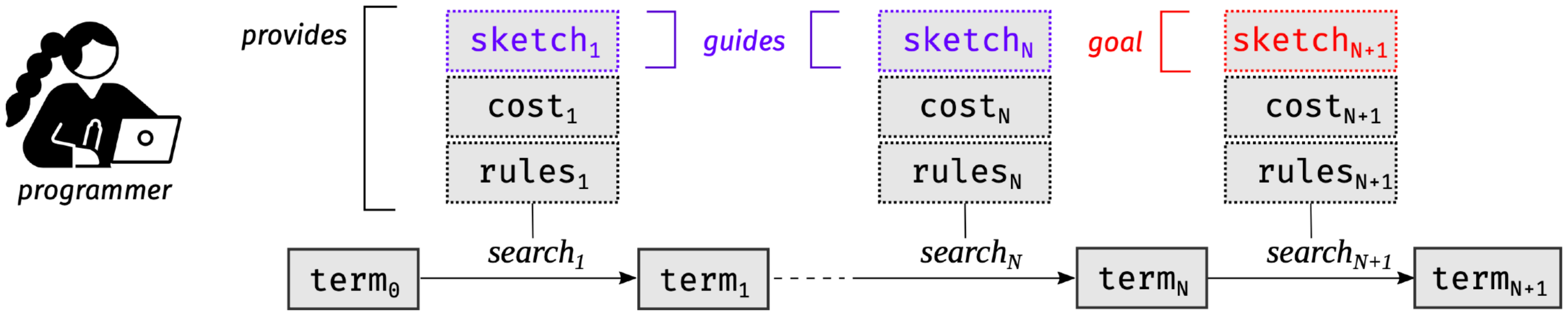
Sketch-Guided Equality Saturation

Automation vs. Manual control

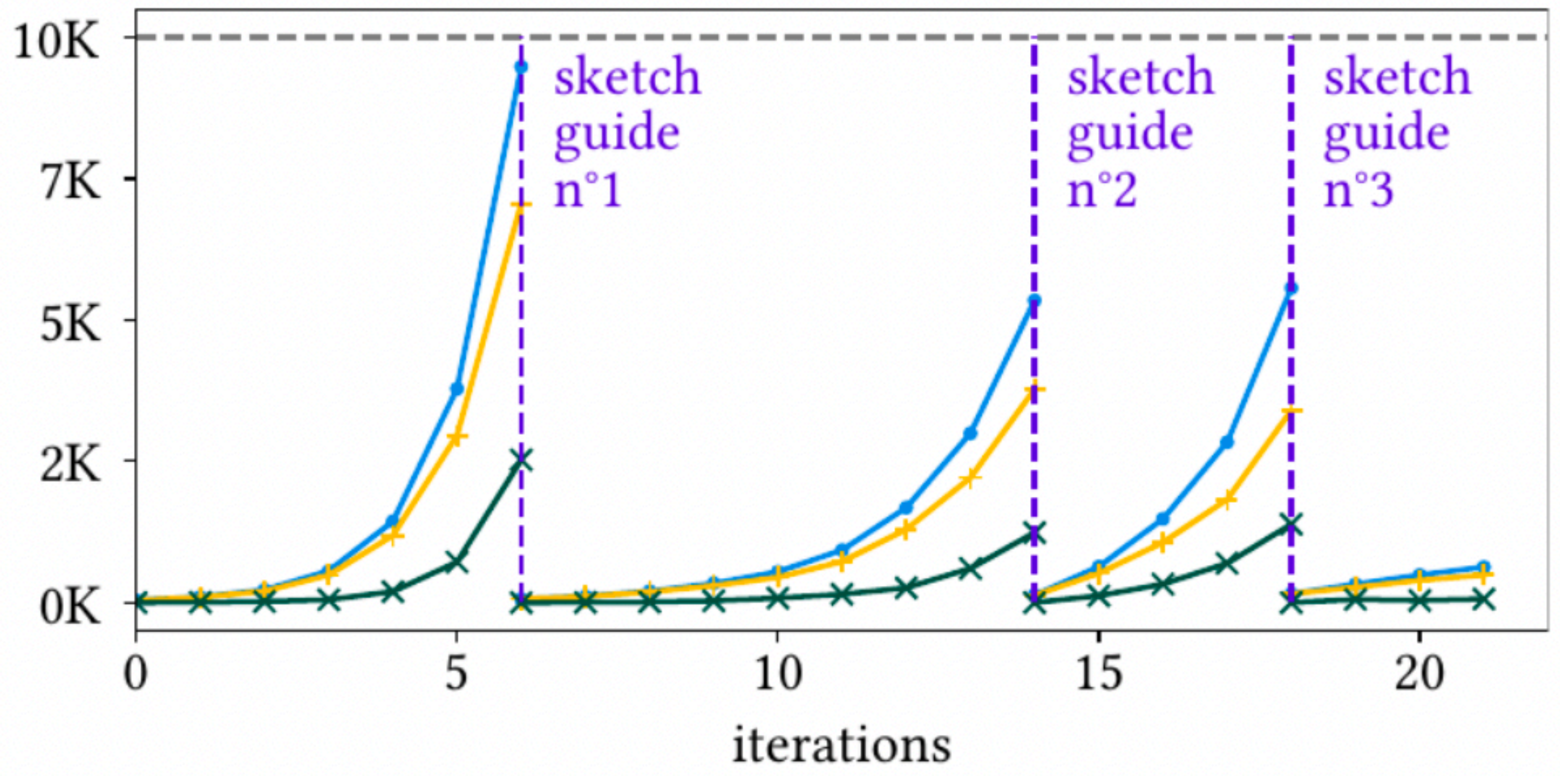
Idea:
Describe rewrite *goal* rather than rewrite sequence:

A *sketch* describes
a desired program shape

```
containsMap(m / 32,                     | for m / 32:
 containsMap(n / 32,                    |  for n / 32:
  containsReduceSeq(k / 4,              |   for k / 4:
   containsReduceSeq(4,                 |    for 4:
    containsMap(32,                     |     for 32:
     containsMap(32,                    |      for 32:
      containsAddMul))))))              |       .. + .. × ..
```



Break intractable equality saturation search
into multi tractable one, by human *guidance*.



All optimizations from this paper
are found in < 7 seconds *automatically*

Talk by **Thomas Kœhler** earlier this week at the E-Graph workshop. Paper: https://arxiv.org/abs/2111.13040

Achieving High-Performance the Functional Way

Expressing High-Performance Optimisations as Rewrite Strategies

Bastian Hagedorn, Johannes Lenfers, Thomas Kœhler, Xueying Qin, Sergei Gorlatch and Michel Steuwer

michel.steuwer@ed.ac.uk

https://michel.steuwer.info/

https://rise-lang.org/

https://github.com/rise-lang/shine

https://github.com/elevate-lang/elevate