Achieving High-Performance the Functional Way

Expressing High-Performance Optimizations as Rewrite Strategies

BASTIAN HAGEDORN, JOHANNES LENFERS, THOMAS KOEHLER, XUEYING QIN, RONGXIAO FU,
SERGEI GORLATCH (University of Münster, Germany), ORNELA DARDHA (University of Glasgow), MICHEL STEUWER (University of Edinburgh)

https://michel.steuwer.info/files/publications/2020/ICFP-2020.pdf

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 built-in operations and optimizations but freely define their own computational patterns in RISE and optimization strategies in ELEVATE in a composable and reusable way. We show how our holistic functional approach achieves competitive performance with the state-of-the-art imperative systems Halide and TVM.

CCS Concepts: • **Software and its engineering → Functional languages**; **Compilers**; • **Theory of computation → Rewrite systems**.

92

Why do we care about "*High-Performance*"?

Training modern machine learning models is crazily (computational) expensive

Why do we care about "*High-Performance*"?

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

XLNet: Generalized Autoregressive Pretraining for Language Understanding

Zhilin Yang*[1], Zihang Dai*[12], Yiming Yang[1], Jaime Carbonell[1], Ruslan Salakhutdinov[1], Quoc V. Le[2]
[1]Carnegie Mellon University, [2]Google Brain
{zhiliny,dzihang,yiming,jgc,rsalakhu}@cs.cmu.edu, qvl@google.com

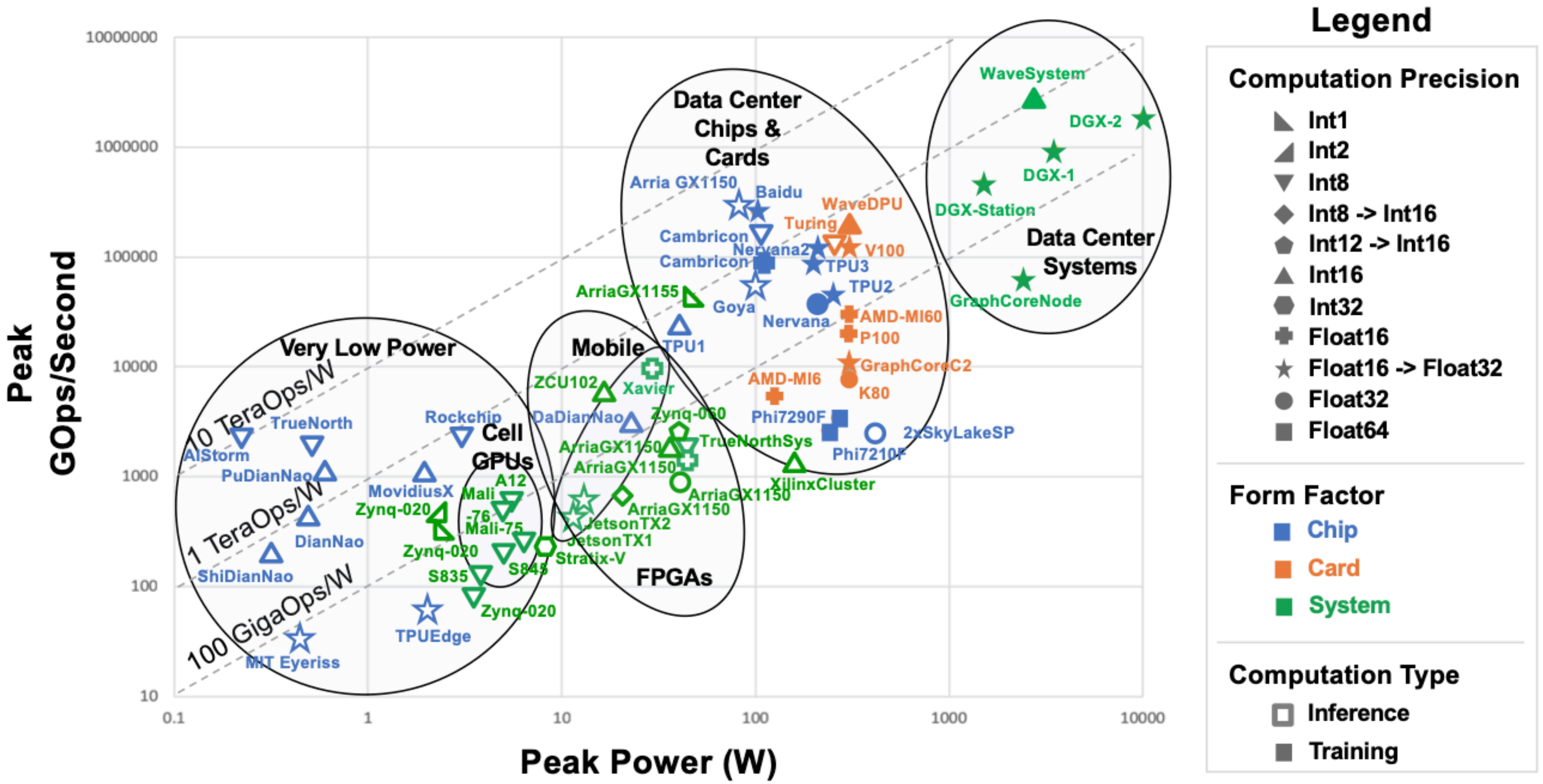
Abstract

With the capability of modeling bidirectional contexts, denoising autoencoding based pretraining like BERT achieves better performance than pretraining approaches based on autoregressive language modeling. However, relying on corrupting the input with masks, BERT neglects dependency between the masked positions and suffers from a pretrain-finetune discrepancy. In light of these pros and cons, we propose XLNet, a generalized autoregressive pretraining method that (1) enables learning bidirectional contexts by maximizing the expected likelihood over all permutations of the factorization order and (2) overcomes the limitations of BERT thanks to its autoregressive formulation. Furthermore, XLNet integrates ideas from Transformer-XL, the state-of-the-art autoregressive model, into pretraining. Empirically, XLNet outperforms BERT on 20 tasks, often by a large margin, and achieves state-of-the-art results on 18 tasks including question answering, natural language inference, sentiment analysis, and document ranking.[1].

4:11 pm · 24 Jun 2019 · Twitter for Android

323 Retweets and comments **651** Likes

The Boom of Machine Learning Accelerators



Who is going to program (and optimize for) all of these hardware devices?

How do we control optimizations for performance sensitive code?

- Rely on compiler heuristics

clang.llvm.org

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.

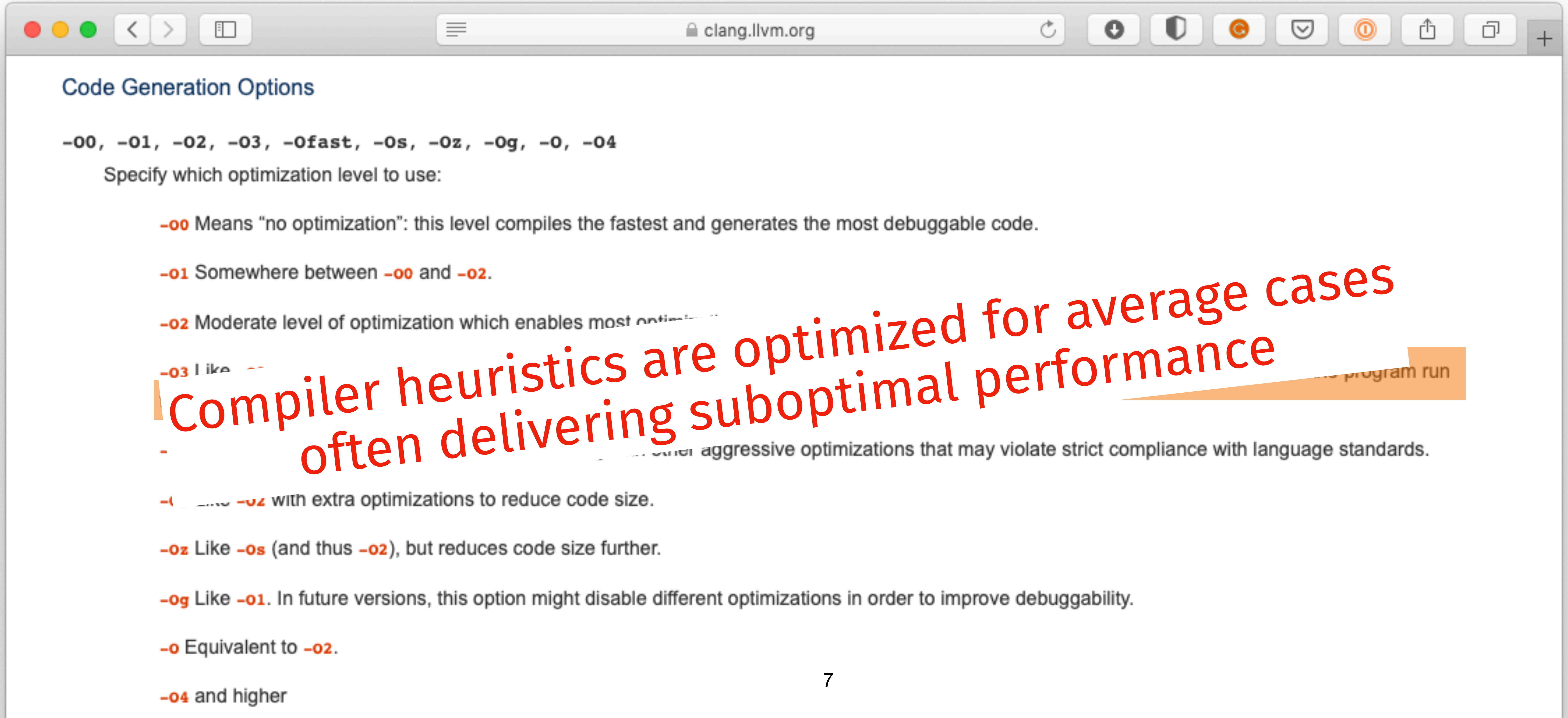
`-Og` Like `-O1`. In future versions, this option might disable different optimizations in order to improve debuggability.

`-O` Equivalent to `-O2`.

`-O4` and higher

How do we control optimizations for performance sensitive code?

- Rely on compiler heuristics



How do we control optimizations for performance sensitive code?

- Rely on compiler heuristics
- Write low-level code

```
__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;
}
```

Straightforward matrix multiplication

10-100x
performance

30x
lines of code

```
__global__ optimized_matmul(const __half *A, const __half *B, __half *C,
                            int K, int M, int N) {
  // ... 164 lines skipped
  #pragma unroll
  for (int mma_k = 1; mma_k < 4; mma_k++) {

    // load A from shared memory to register file
    #pragma unroll
    for (int mma_m = 0; mma_m < 4; mma_m++) {
      int swizzle1 = swapBits(laneLinearIdx, 3, 4);
      laneIdx = make_uint3(
        ((swizzle1 % 32) % 16), (((swizzle1 % 32)/16) % 2), (swizzle1/32));
      if (laneIdx.x < 16) { if (laneIdx.y < 2) {
        const int4 * a_sh_ptr = (const int4 *) &A_sh[((((warpIdx.y*64) +
        (mma_m*16)+laneIdx.x))*32)+((((((laneLinearIdx>>1))&3)^mma_k)*8))];
        int4 * a_rf_ptr = (int4 *) &A_rf[(mma_k & 1)][mma_m][0][0];
        *a_rf_ptr = *a_sh_ptr; }}}

    // load B from shared memory to register file
    #pragma unroll
    for (int mma_n = 0; mma_n < 4; mma_n++) {
      int swizzle2 = swapBits((swapBits(laneLinearIdx, 2, 3)), 3, 4);
      laneIdx = make_uint3(
        ((swizzle2%32)%16), (((swizzle2%32)/16)%2), (swizzle2/32));
      if (laneIdx.y < 2) { if (laneIdx.x < 16) {
        const int4 * b_sh_ptr = (const int4 *) &B_sh[
          ((((warpIdx.x*64) + (mma_n*16) + laneIdx.x)) * 32) +
          (((((((swapBits(laneLinearIdx,2,3))>>1))&3)^mma_k)*8))];
        int4 * b_rf_ptr = (int4 *) &B_rf[(mma_k & 1)][0][mma_n][0];
        *b_rf_ptr = *b_sh_ptr; }}}

    // compute matrix multiplication using tensor cores
    #pragma unroll
    for (int mma_m = 0; mma_m < 4; mma_m++) {
      #pragma unroll
      for (int mma_n = 0; mma_n < 4; mma_n++) {
        int * a = (int *) &A_rf[((mma_k - 1) & 1)][mma_m][0][0];
        int * b = (int *) &B_rf[((mma_k - 1) & 1)][0][mma_n][0];
        float * c = (float *) &C_rf[mma_m][mma_n][0];

        asm volatile( \
          "mma.sync.aligned.m8n8k4.row.col.f32.f16.f16.f32\n" \
          "    {%0, %1, %2, %3, %4, %5, %6, %7}, \n" \
          "    {%8, %9}, \n" \
          "    {%10, %11}, \n" \
          "    {%0, %1, %2, %3, %4, %5, %6, %7}; \n" \
               : "+f"(c[0]), "+f"(c[2]), "+f"(c[1]), "+f"(c[3])
               , "+f"(c[4]), "+f"(c[6]), "+f"(c[5]), "+f"(c[7])
               :  "r"(a[0]),  "r"(a[1])
               ,  "r"(b[0]),  "r"(b[1]));
        asm volatile( \
          "mma.sync.aligned.m8n8k4.row.col.f32.f16.f16.f32\n" \
          "    {%0, %1, %2, %3, %4, %5, %6, %7}, \n" \
          "    {%8, %9}, \n" \
          "    {%10, %11}, \n" \
          "    {%0, %1, %2, %3, %4, %5, %6, %7}; \n" \
               : "+f"(c[0]), "+f"(c[2]), "+f"(c[1]), "+f"(c[3])
               , "+f"(c[4]), "+f"(c[6]), "+f"(c[5]), "+f"(c[7])
               :  "r"(a[2]),  "r"(a[3])
               ,  "r"(b[2]),  "r"(b[3])); }}}
  // ... 95 lines skipped
}
```

Optimized matrix multiplication
(321 lines of code in total)

How do we control optimizations for performance sensitive code?

- Rely on compiler heuristics
- Write low-level code

10-100x performance

30x

```
__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 * bloc

    float acc = 0.0;
    for (int k = 0; k < K; k+
        acc += A[y * M + k] *
    }
    C[y * N + x] = acc;
}
```

Straightforward matrix multiplication

```
__global__ optimized_matmul(const __half *A, const __half *B, __half *C,
                            int K, int M, int N) {
  // ... 164 lines skipped
  #pragma unroll
  for (int mma_k = 1; mma_k < 4; mma_k++) {

    // load A from shared memory to register file
    #pragma unroll
    for (int mma_m = 0; mma_m < 4; mma_m++) {
      int swizzle1 = swapBits(laneLinearIdx, 3, 4);
      laneIdx = make_uint3(
        ((swizzle1 % 32) % 16), (((swizzle1 % 32)/16) % 2), (swizzle1/32));
      if (laneIdx.x < 16) { if (laneIdx.y < 2) {
        const int4 * a_sh_ptr = (const int4 *) &A_sh[((((warpIdx.y*64) +
        (mma_m*16)+laneIdx.x))*32)+((((((laneLinearIdx>>1))&3)^mma_k)*8))];
        int4 * a_rf_ptr = (int4 *) &A_rf[(mma_k & 1)][mma_m][0][0];
        *a_rf_ptr = *a_sh_ptr; }}}

    // load B from shared memory to register file
    #pragma unroll
    for (int mma_n = 0; mma_n < 4; mma_n++) {
      int swizzle2 = swapBits((swapBits(laneLinearIdx, 2, 3)), 3, 4);
      laneIdx = make_uint3(
        ((swizzle2%32)%16), (((swizzle2%32)/16)%2), (swizzle2/32));
      if (laneIdx.y < 2) { if (laneIdx.x < 16) {
        const int4 * b_sh_ptr = (const int4 *) &B_sh[
          ((((warpIdx.x*64) + (mma_n*16) + laneIdx.x)) * 32) +
          (((((((swapBits(laneLinearIdx,2,3))>>1))&3)^mma_k)*8))];
        int4 * b_rf_ptr = (int4 *) &B_rf[(mma_k & 1)][0][mma_n][0];
        *b_rf_ptr = *b_sh_ptr; }}}

    // compute matrix multiplication using tensor cores
    #pragma unroll
    for (int mma_m = 0; mma_m < 4; mma_m++) {
      #pragma unroll
      for (int mma_n = 0; mma_n < 4; mma_n++) {
        int
                    (mma_k - 1) & 1)][mma_m][0][0];
                    (mma_k - 1) & 1)][0][mma_n][0];
                    rf[mma_m][mma_n][0];

                    .row.col.f32.f16.f16.f32\n" \
                    %5, %6, %7}, \n" \

                    %1, %2, %3, %4, %5, %6, %7}; \n" \
                : "+f"(c[0]), "+f"(c[2]), "+f"(c[1]), "+f"(c[3])
                , "+f"(c[4]), "+f"(c[6]), "+f"(c[5]), "+f"(c[7])
                :  "r"(a[0]),  "r"(a[1])
                ,  "r"(b[0]),  "r"(b[1]));
        asm volatile( \
          "mma.sync.aligned.m8n8k4.row.col.f32.f16.f16.f32\n" \
          "     {%0, %1, %2, %3, %4, %5, %6, %7}, \n" \
          "     {%8, %9}, \n" \
          "     {%10, %11}, \n" \
          "     {%0, %1, %2, %3, %4, %5, %6, %7}; \n" \
                : "+f"(c[0]), "+f"(c[2]), "+f"(c[1]), "+f"(c[3])
                , "+f"(c[4]), "+f"(c[6]), "+f"(c[5]), "+f"(c[7])
                :  "r"(a[2]),  "r"(a[3])
                ,  "r"(b[2]),  "r"(b[3])); }}}
  // ... 95 lines skipped
}
```

Optimized matrix multiplication
(321 lines of code in total)

Low-level code is error prone, hard to debug & specific for a single device or architecture

How do we control optimizations for performance sensitive code?

- Rely on compiler heuristics
- Write low-level code
- Scheduling APIs

Halide

tvm

Tiramisu-Compiler / **tiramisu**

Fireiron NVIDIA

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;
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);
```

Optimization Schedule

Halide compiler

Optimised Code

How do we control optimizations for performance sensitive code?

- Rely on compiler heuristics
- Write low-level code
- Scheduling APIs

Halide

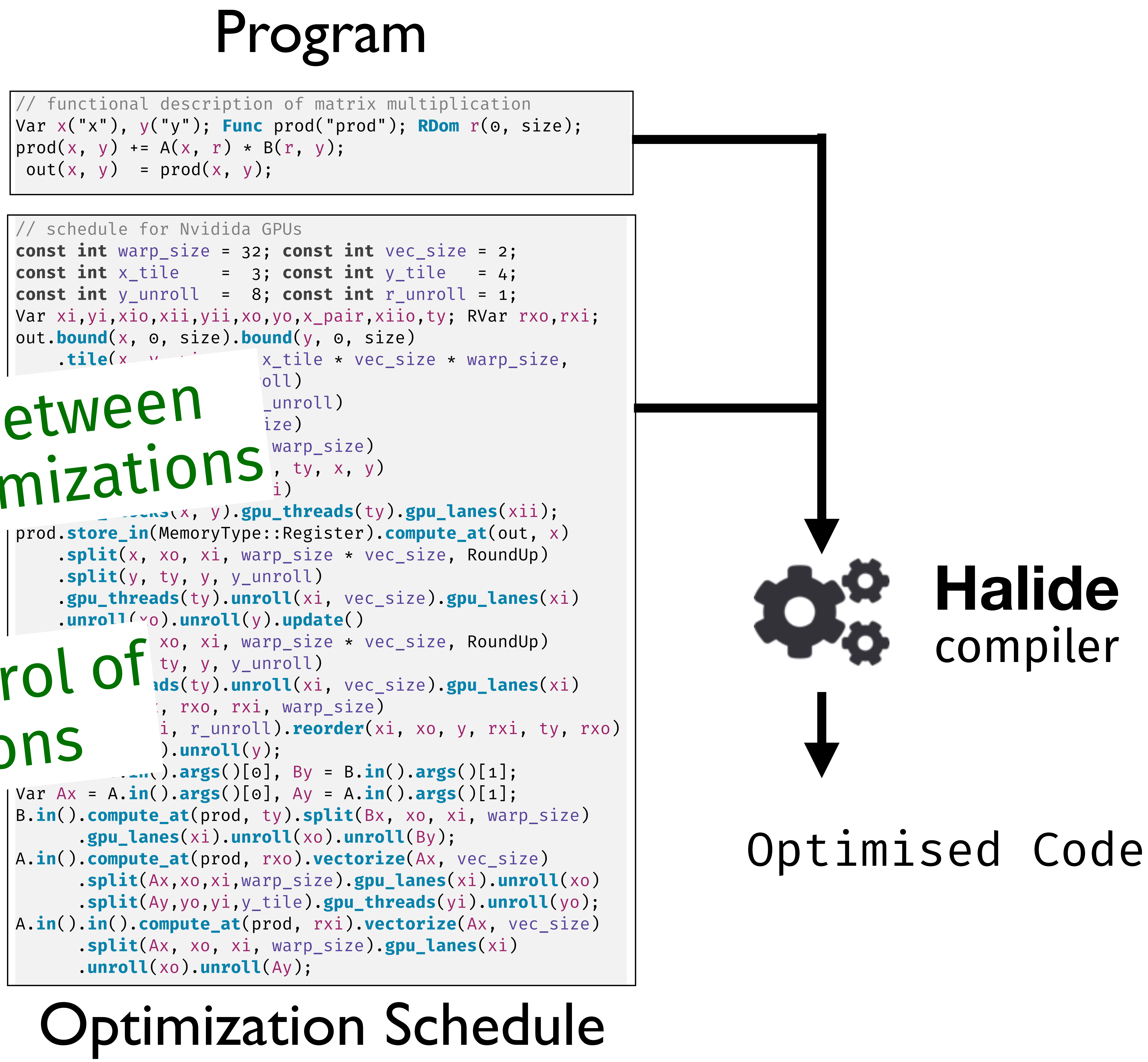
tvm

Tiramisu-Compiler / tiramisu

Fireiron NVIDIA

Clear separation between computation & optimizations

Detailed control of optimizations



Problems with Scheduling APIs

Hinders reuse

Hinders understanding

Not well-d… …ntics

No extensibility

Only fix… …isations

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

Optimization Schedule

Halide compiler

Code

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

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.
   They should not be changed to express optimizations;

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.

4. *Allow reasoning*

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

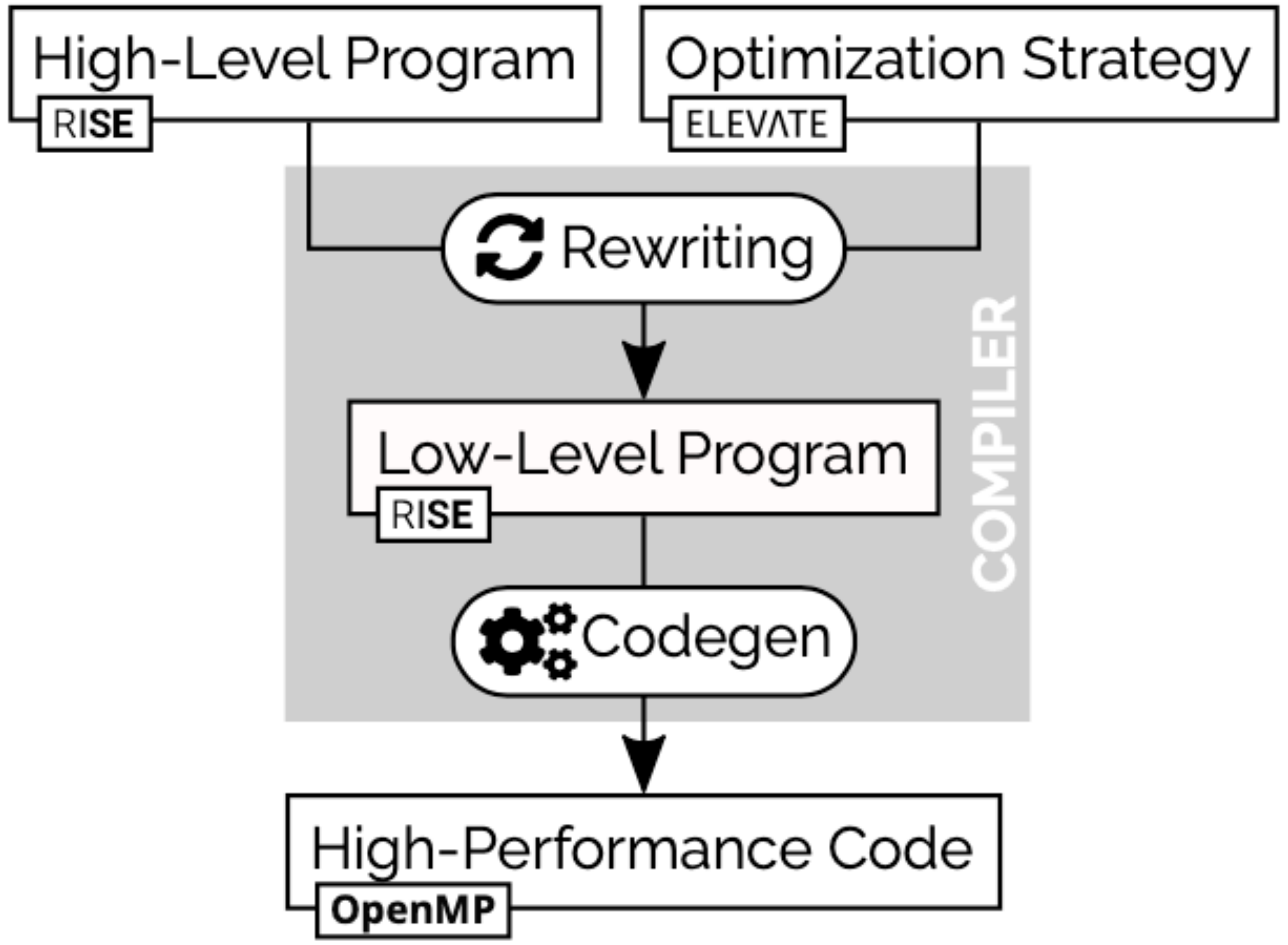
5. *Be explicit*

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

"*The Functional Way*" for Achieving High-Performance

```
// 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)
```



ELEVATE — **A Language for Describing Optimisation Strategies**

- In ELEVATE Optimisation **Strategies** are encoded as functions

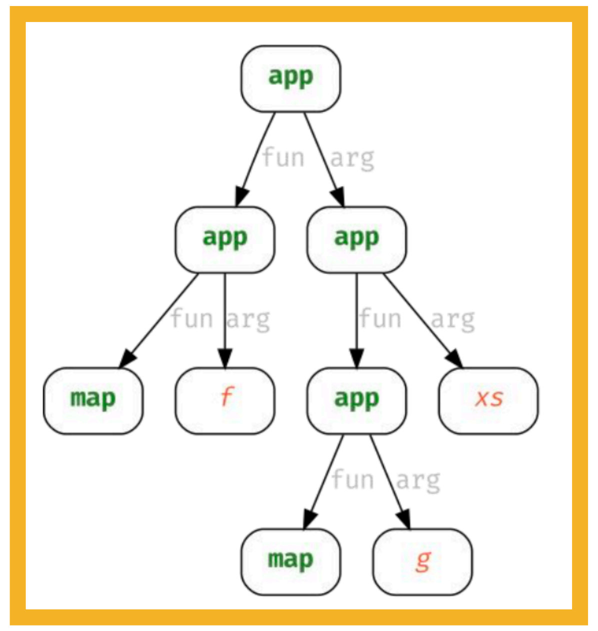
```
type Strategy[P]: P → RewriteResult[P]
```

Rewritten Program
or
Failure

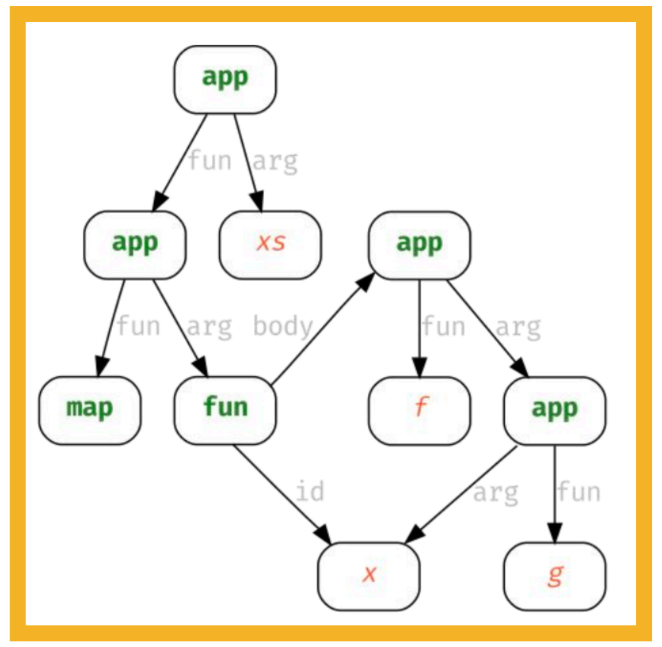
- *Rewrite rules* are examples of basic strategies

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

mapFusion(



) =



Strategy Combinators

- Sequential composition (;):

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

- 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)
```

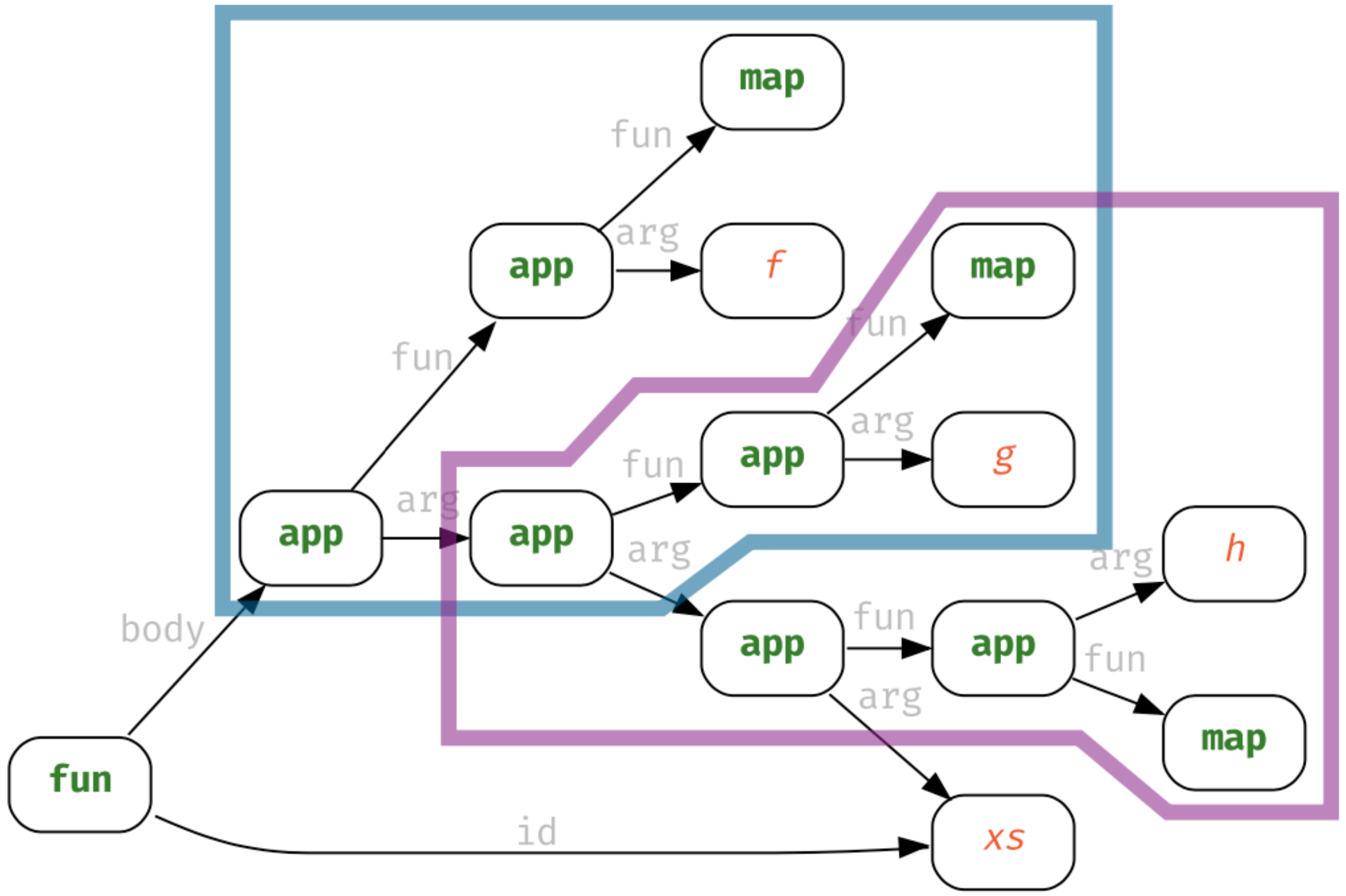
Traversal Strategies

- Where to apply a rewrite strategy?

  Two possible locations for applying

  ```
  def mapFusion: Strategy = …
  ```

  within the same expression



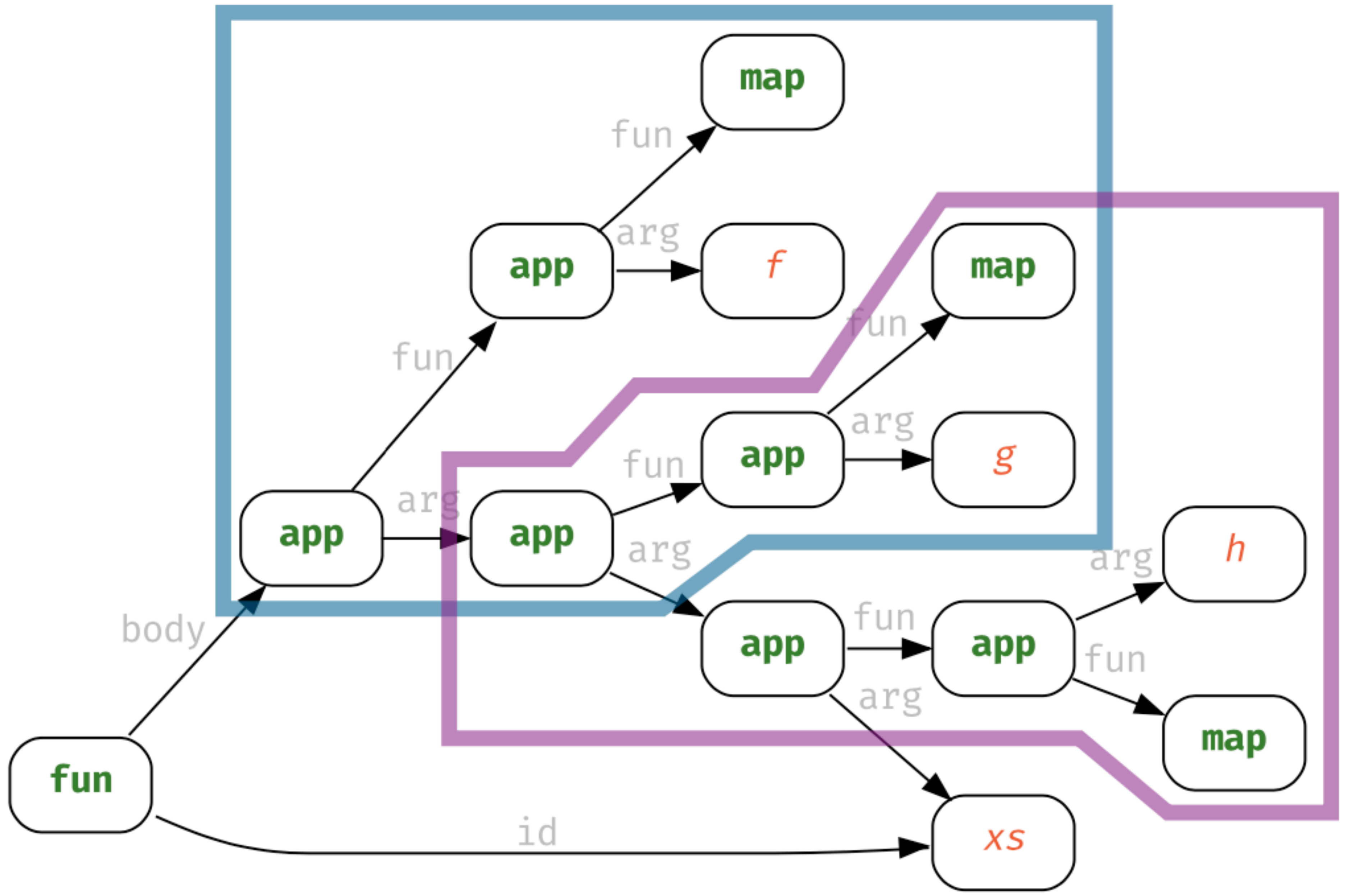

```
threemaps = fun(xs, map(f)(map(g)(map(h)(xs))))
```

Traversal Strategies

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

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

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




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

```
body(mapFusion)(threemaps)  vs  body(argument(mapFusion))(threemaps)
```

Complex Traversals + Normalization

- 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:

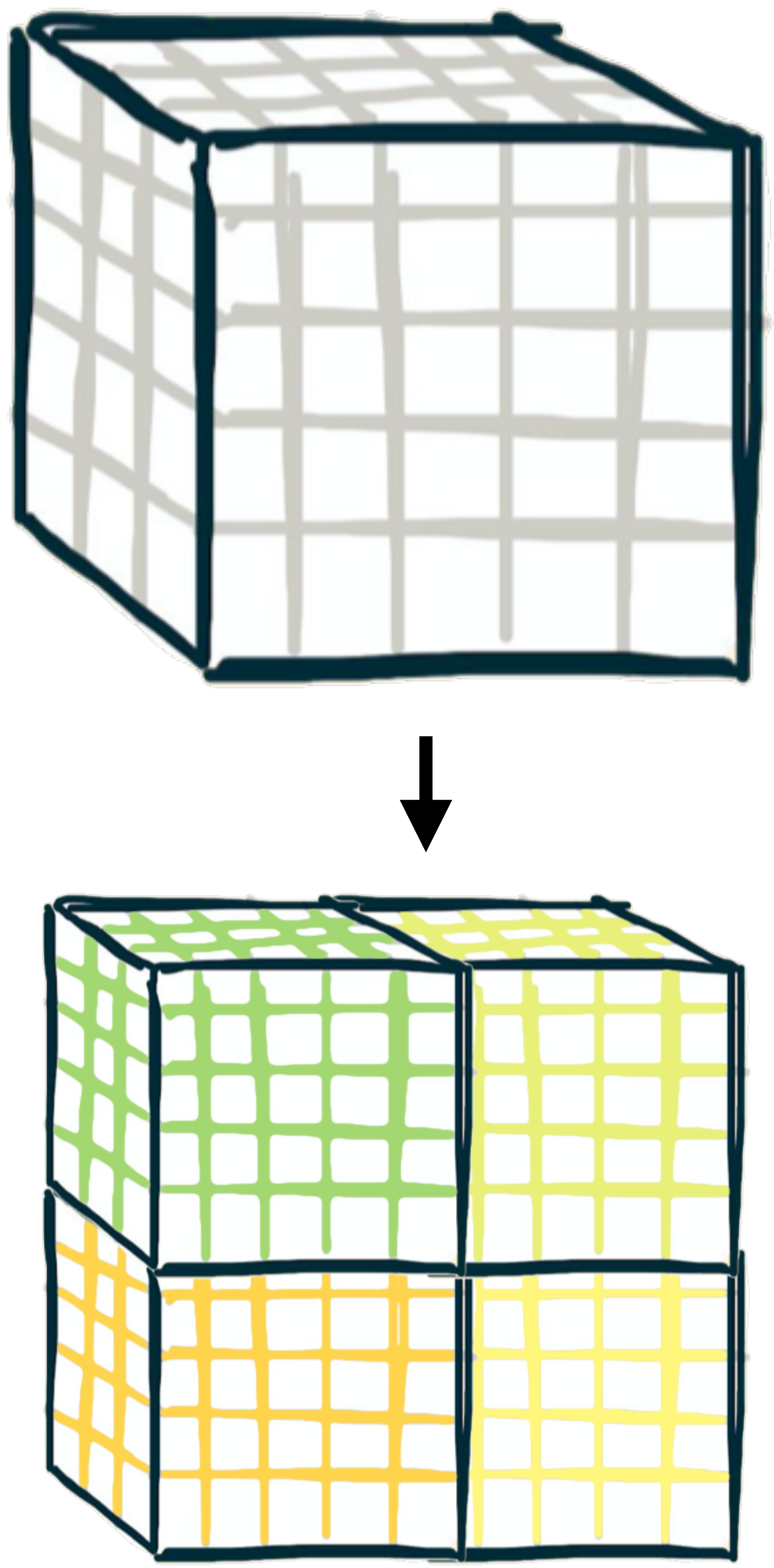
```
def     topDown[P]: Traversal[P] = s => p => (s <+ one(topDown(s)))(p)
def    bottomUp[P]: Traversal[P] = s => p => (one(bottomUp(s)) <+ s)(p)
def  allTopDown[P]: Traversal[P] = s => p => (s ‘;‘ all(allTopDown(s)))(p)
def allBottomUp[P]: Traversal[P] = s => p => (all(allBottomUp(s)) ‘;‘ s)(p)
def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

Tiling defined as an optimisation strategy

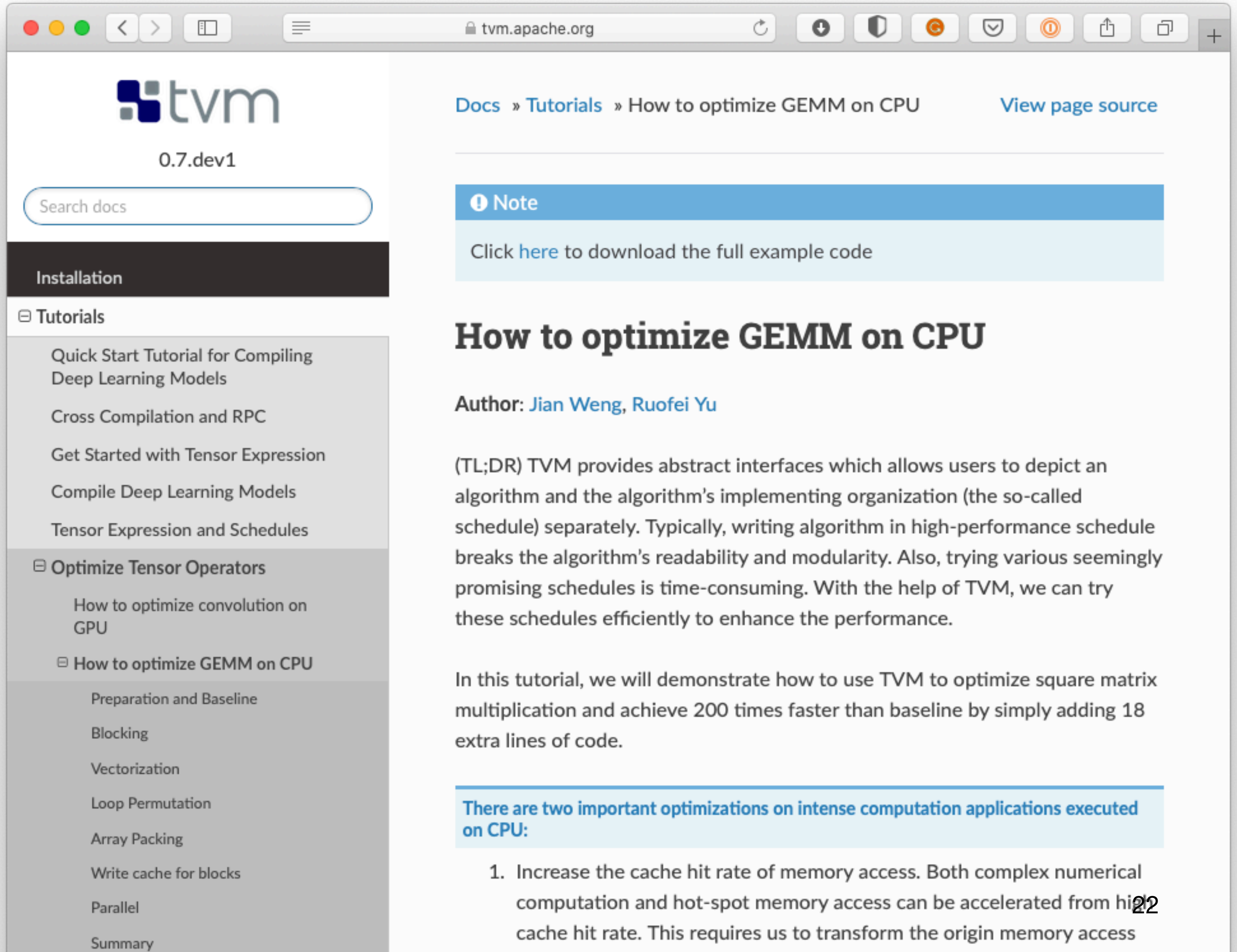


```
def tile: Int → Int → Strategy =
  (dim) ⇒ (n) ⇒ dim match {
    case 1 = function(splitJoin(n))
    case 2 = fmap(function(splitJoin(n))) ;
             function(splitJoin(n)) ; interchange(2)
    case i = fmap(tile(dim-1, n)) ;
             function(splitJoin(n)) ; interchange(n)
  }
```

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

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

- 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

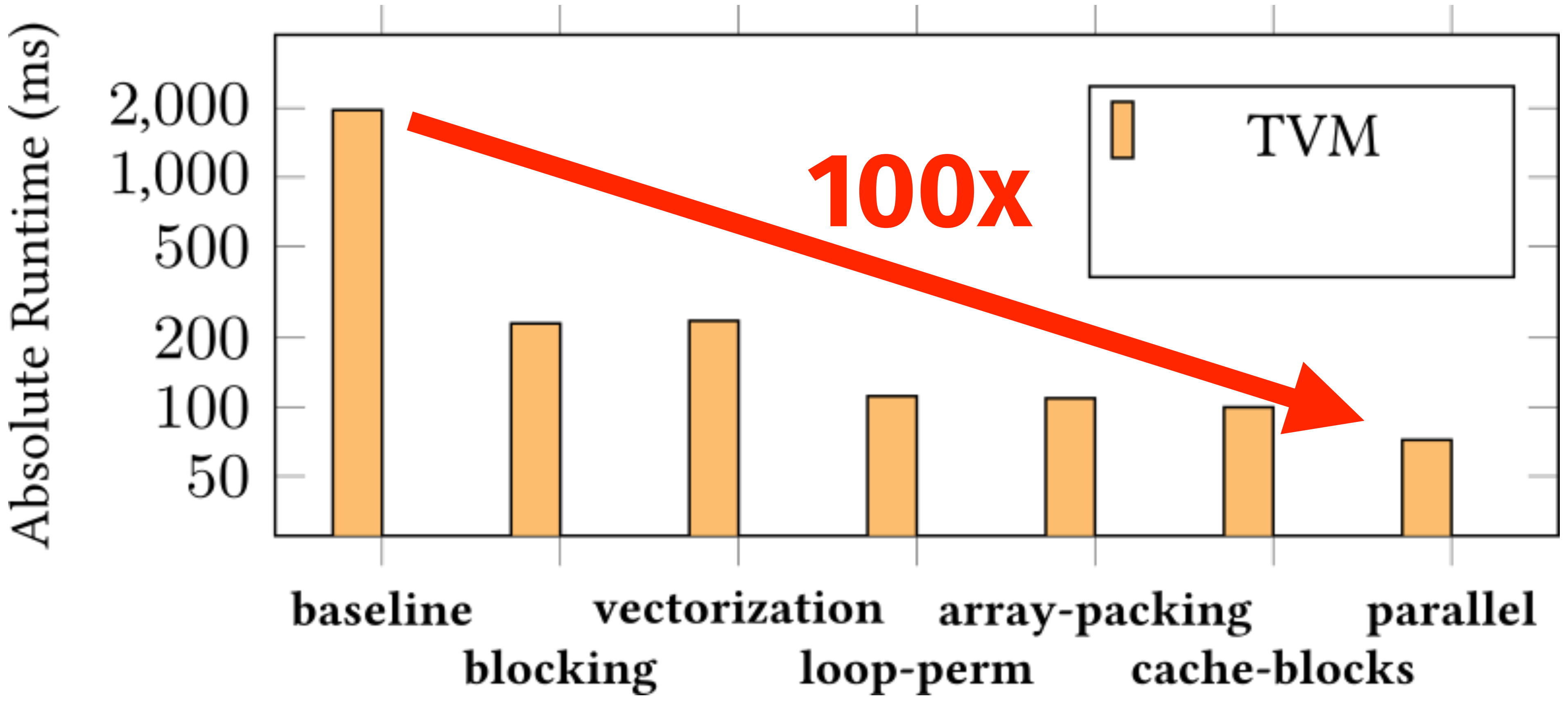
Author: Jian Weng, Ruofei Yu

(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

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

Enable composability

Be 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 behavior

Baseline Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

ELEVATE

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)
```

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)
```

No clear separation of concerns

Loop Permutation with blocking Strategy

Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

Clear separation of concerns vs No clear separation of concerns

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)
```

Facilitate reuse

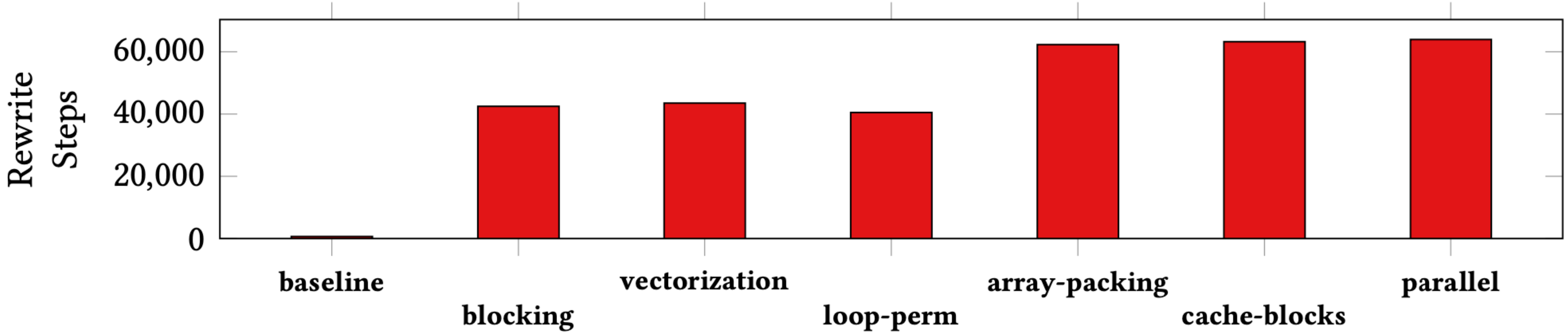
tvm

```
# 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

Number of successful rewrite steps

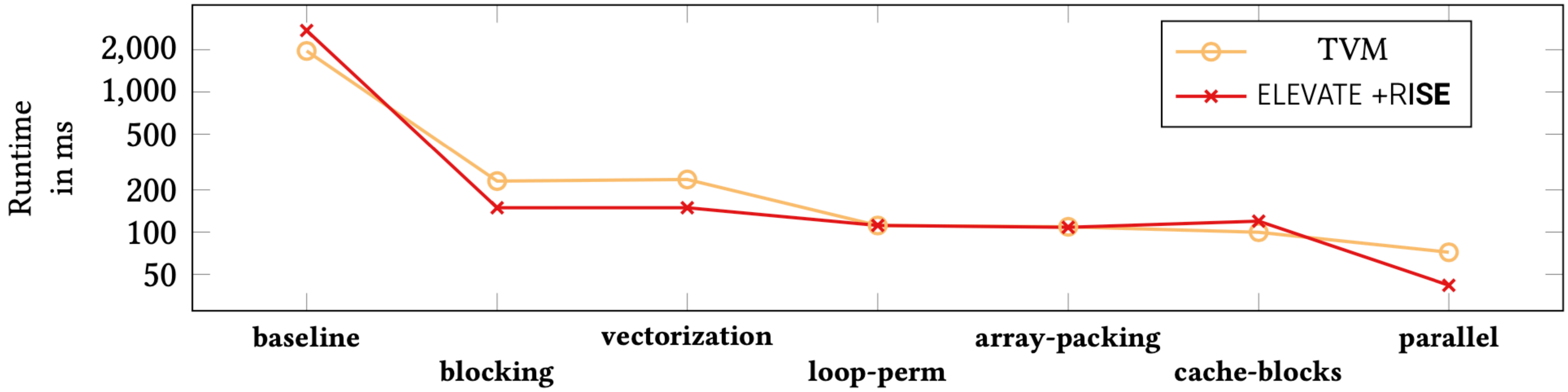


Rewriting took less than 2 seconds with our unoptimised implementation

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 **?**

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.

- Can types help to write ELEVATE strategies?
- We are developing a row-polymorphic version of ELEVATE
  joined work with Rongxiao Fu and Ornela Dardha

Datatypes in a row

Record:
```
type Point = {X: Int | Y: Int | Z: Int | *}
```

Variant:
```
type Color = <Red: {*} | Green: {*} | Blue: {*} | *>
```

$$\begin{tikzcd} & \texttt{*} \\ \texttt{row} \arrow[ur] \arrow[dr] & \\ & \texttt{Tag: Type | row} \end{tikzcd}$$

```
Record := {row}

Variant := <row>
```

Rows are a generalisation of record and variant types

Datatypes in a row

Record:
```
type Point = {X: Int | Y: Int | Z: Int | *}
```

Variant:
```
type Color = <Red: {*} | Green: {*} | Blue: {*} | *>
```

```
type ColorfulPoint = {X: Int | Y: Int | Z: Int | Color: Color | *}
```

```
shiftX: (n: Int) -> (p: Point) -> Point
setRed: (p: ColorfulPoint) -> ColorfulPoint
```

How do we make Point and ColorfulPoint compatible?

Datatypes in a row

Record:
```
type Point = {X: Int | Y: Int | Z: Int | r}
```

Variant:
```
type Color = <Red: {*} | Green: {*} | Blue: {*} | *>
```

```
type ColorfulPoint = {X: Int | Y: Int | Z: Int | Color: Color | *}
```

```
shiftX: (n: Int) -> (p: Point) -> Point
setRed: (p: ColorfulPoint) -> ColorfulPoint
```

Types are compatible not via subtyping
but by instantiating row variables

Datatypes in a row

```
type Primitive = forall [p]. <Map: {*} | Reduce: {*} | Slide: {*} | p>
type Rise = forall [p]. e as <Id: {Name: Nat | *} |
  Lam: {Param: Nat | Body: e | *} | App: {Fun: e | Arg: e | *} |
  Primitive: Primitive[p] | *>
```

$$\begin{tikzcd} & \texttt{*} \\ \texttt{row} \arrow[ur] \arrow[r] \arrow[dr] & \texttt{r} \\ & \texttt{Tag: Type | row} \end{tikzcd}$$

```
Record := {row}

Variant := <row>

Recursive Variant := a as <row>
```

We represent computational expressions using a variant type

Encoding Representation of Programs in Types

RISE AST as a recursive row-polymorphic variant type:

```
type Rise = t as <
    Id:  { Name: Nat | * }
  | Lam: { Param: Nat | Body: t | * }
  | App: { Fun: t | Arg: t | * }
  | Primitive: <Map: {*} | Zip: {*} | Reduce: {*} | *>
  |  * >
```

Representing Programs in Types — Example

Suggared expression:

```
map g (map f xs)
```

Desuggared expression:

```
App {Fun: App {Fun: Primitive Map | Arg: g} |
     Arg: App {Fun: App {Fun: Primitive Map | Arg: f} |
               Arg: xs}}
```

:

Type

```
<App: { Fun: <App: {
              Fun: <Primitive: <Map: {*} | > | > |
              Arg: g | *} | > |
        Arg: <App: {
              Fun: <App: {
                    Fun: <Primitive: <Map: {*} | > | >
                    Arg: f | *} | > |
              Arg: xs | *} | > | } | >
```

Types of Strategies

```
type Strategy = forall p1 p2. p1 -> RewriteResult p2
```

```
type RewriteResult = forall p.
    < Success: p | Failure: {*} | * >
```

Map Fusion Strategy

Strategy Implementation

```
let mapFusion = lam expr = match expr with <
  -- map g (map f xs)
  App {Fun: App {Fun: Primitive Map | Arg: g} |
       Arg: App {Fun: App {Fun: Primitive Map | Arg: f}|
                 Arg: xs}} =>
   -- Success ( map fun(x => g (f x)) xs )
   Success (App {Fun: App {Fun: Primitive Map |
     Arg: Lam { Param: o | Body: App {Fun: g | Arg:
       App {Fun: f | Arg: Id {Name: o}}}}}} | Arg: xs}) >
```

:

Inferred Type

```
  < App: { Fun: < App: {
              Fun: < Primitive: <Map: {*} | *> | *> |
              Arg: g | } | *> |
            Arg: < App: {
              Fun: < App: {
                Fun: < Primitive: < Map: {*} | *> | *> |
                Arg: f | } | *> |
              Arg: xs | } | *> | } | *>
->
  < Success: < App: {
       Fun: < App: {
         Fun: < Primitive: <Map: {*} | > | > |
         Arg: < Lam: { Param: <o: {*} | > | Body: < App:{
             Fun: g |
             Arg: < App: {
               Fun: f |
               Arg: < Id: {Name: <o: {*} | > | *} | > |*
             } | > | * } | > | *} | > | * } | > |
       Arg: xs | * } | > | >
```

Map Fusion Strategy — With Syntactic Sugar

Strategy Implementation

```
let mapFusion: Rise -> RewriteResult Rise =
  lam expr
    match expr with <
      map g (map f xs) =>
        Success (map fun(x => g (f x)) xs)

    >
```

Inferred Type

```
 < App: { Fun: < App: {
              Fun: < Primitive: <Map: {*} | *> | *> |
              Arg: g | } | *> |
            Arg: < App: {
              Fun: < App: {
                Fun: < Primitive: < Map: {*} | *> | *> |
                Arg: f | } | *> |
```

:

```
< map g (map f xs) | * > ->
  < Success (map fun(x => g (f x)) xs) | >
```

```
          Fun: < Primitive: <Map: {*} | > | > |
          Arg: < Lam: { Param: <o: {*} | > | Body: < App:{
                Fun: g |
                Arg: < App: {
                  Fun: f |
                  Arg: < Id: {Name: <o: {*} | > | *} | > |*
                } | > | * } | > | *} | > | * } | > |
        Arg: xs | * } | > | >
```

Strategy Combinators and Types

```
let id: p -> <Success: p | > =
  lam expr = Success expr

let fail: p -> <Failure: {*} | > =
  lam expr = Failure

let seq: (p1 -> <Success: p2 | Failure: {*} | *>) ->
         (p2 -> <Failure: {*} | r: ~{Failure}>) ->
         (p1 -> <Failure: {*} | r: ~{Failure}>) =
  lam fs = lam ss = lam expr1 = match (fs expr1) with <
      Success expr2 => ss expr2
    | Failure => Failure
  >

let lChoice: (p1 -> <Success: p2 | Failure: {*} | *>) ->
             (p1 -> <Success: p2 | r: ~{Success}>)   ->
             (p1 -> <Success: p2 | r: ~{Success}>) =
  lam fs = lam ss = lam expr1 = match (fs expr1) with <
      Success expr2 => Success expr2
    | Failure => ss expr1
  >

let try: (p1 -> <Success: p1 | Failure: {*} | *>) ->
         (p1 -> <Success: p1 | >) =
  lam s = lChoice s id
```

Returns program p unchanged

Always fails

Combines types of first and second strategy

Possibility of failure depends on the second strategy

Can not fail

Safe Compositions

Inferred Type

seq mapFusion mapFusion :

```
< map h (map g (map f xs)) | * > ->
  < map fun(x => fun(y => h (g y)) (f x)) xs |
    Failure: {*} | >
```

Input program must be the composition of three maps

Future Applications for Strategy Types

- Verification of correctness of program transformations

  Types (of strategies) as propositions

- Synthesizing program transformations

  Types as specifications

Achieving High-Performance the Functional Way

- I have presented a new functional way to achieve high-performance:
  - Computations are expressed using functional patterns
  - Optimization strategies are build in a novel strategy language
  - We achieve performance similar to existing machine learning systems
- We are looking into how row-polymorphic types might help to write strategies

ICFP Paper at: https://michel.steuwer.info/files/publications/2020/ICFP-2020.pdf

A framework for systematically optimising domain-specific applications for specialised hardware