DOI:10.1145/3580371

To view the accompanying Technical Perspective, visit doi.acm.org/10.1145/3580370

Achieving High Performance the Functional Way:

Expressing High-Performance Optimizations as Rewrite Strategies

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

Abstract

Optimizing programs to run efficiently on modern parallel hardware is hard but crucial for many applications. The predominantly used imperative languages 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 toward specialized hardware devices to further increase efficiency.

Many emerging domain-specific languages (DSLs) used in performance-demanding domains such as deep learning 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 paper, 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 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 such as Halide and TVM.

1. INTRODUCTION

As Moore's Law and Dennard scaling are coming to an end,[7] performance and energy efficiency gains no longer come for free for software developers that have to optimize for an increasingly diverse set of hardware by exploiting subtle details of the architecture. The accelerating trend toward specialized hardware, which offers extreme benefits for performance and energy efficiency if the optimized software exploits it, emphasizes the need for performance portability.

The predominant imperative and low-level programming approaches such as C, CUDA, or OpenCL force programmers to intertwine the code describing the program's functional behavior with optimization decisions, making them—by design—non performance portable. As an alternative, higher-level domain-specific approaches have emerged allowing programmers to declaratively describe the functional behavior without committing to a specific implementation. Prominent examples are machine learning systems such as TensorFlow[1] or PyTorch,[10] where the compilers and runtime systems are responsible for optimizing the computations expressed as data flow graphs. Large teams of engineers provide fast implementations for the most common hardware, for TensorFlow including Google's specialized TPU hardware. This labor-intensive support of new hardware is currently only sustainable for the biggest companies—and even they struggle as highlighted by two of the original authors of TensorFlow.[3]

TVM[4] and Halide[11] are two state-of-the-art high-performance domain-specific code generators used in machine learning and image processing. Both attempt to tackle the performance portability challenge by separating the program into two parts: schedules and algorithms. A *schedule* describes the optimizations to apply to an *algorithm* that defines the functional behavior of the computation. Schedules are implemented using a set of predefined *ad hoc* APIs that expose a fixed set of optimization options. TVM's and Halide's authors describe these APIs as a scheduling *language*, but they lack many desirable properties of a programming language. Most crucially, programmers are not able to define their own abstractions. Even the composition of existing optimization primitives can be unintuitive due to the lack of precise semantics and implicit behavior limiting experts' control. Furthermore, for some desirable optimizations, it is not sufficient to change the schedule, but programmers must redefine the algorithm itself—violating the promise of separating algorithm and schedule. To overcome the innovation obstacle of manually optimizing for specialized hardware and for achieving automated performance portability,

The original version of this paper was published in *Proceedings of the ACM on Programming Languages 4* (Aug. 2020), Article 92.

we will need to rethink how we separate, describe, and apply optimizations in a more principled way.

In this paper, we describe a more principled but still practical holistic functional approach to high-performance code generation (Figure 1). We combine RI**SE**, a data-parallel functional language for expressing computations, with a functional strategy language, called ELEVATE. RI**SE** provides well-known functional data-parallel patterns for expressing computations at a high level. ELEVATE enables programmers to define their own abstractions for building optimization strategies in a composable and reusable way. As we will see in our experimental results, our approach provides competitive performance compared with the state of the art while being built with and leveraging functional principles resulting in an elegant and composable design.

2. MOTIVATION AND BACKGROUND

We motivate the need for more principled optimizations with a study of TVM, the state of the art in high-performance domain-specific compilation for machine learning.

2.1. Scheduling languages for high-performance code generation

Halide[11] proposed decoupling a program into the *algorithm*, describing the functional behavior, and the *schedule*, specifying how the compiler should optimize. This has inspired similar approaches such as TVM[4] in deep learning.

Listings 1 and 2 show TVM Python code[15] for optimizing matrix multiplication. Listing 1 shows a simple version where lines 2–6 define the computation: A and B are multiplied by performing the dot product for each coordinate pair (x, y). The dot product is expressed as pairwise multiplications and summation using the `tvm.sum` operator (line 6). Line 8 instructs the compiler to use the default schedule-generating code to compute the output matrix sequentially.

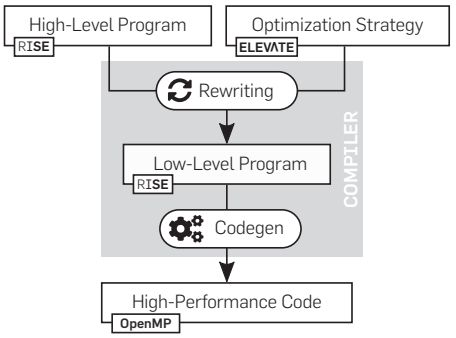
Modifications for optimizing performance. Listing 2 shows an optimized version. The schedule in lines 10–23 specifies multiple optimizations including tiling (line 12), vectorization (line 19), and loop unrolling (line 18) for optimizing the performance on multicore CPUs. However, in order to optimize the memory access pattern, the algorithm has to be changed: A copy of the `B` matrix (`pB`) is introduced in line 5 (and used in line 8) whose elements are reordered depending on the tile size. This optimization is not expressible with scheduling primitives and, therefore, requires the modification of the algorithm—violating the promise of separating algorithm and schedule. Generally, the separation between algorithm and schedule is blurred because both share the same Python identifiers and must live in the same scope. This unsharp separation limits the reuse of schedules across algorithms.

The optimized parallel schedule uses built-in optimization primitives. Some are hardware-specific (like `vectorize`), some are algorithmic optimizations useful for many applications (like `tiling` to increase data locality), and others are low-level optimizations (like `unroll` and `reorder` that transform loop nests). However, TVM's scheduling language is not easily extensible, as adding optimization primitives requires extending the underlying compiler.

The behavior of some primitives is not intuitive, and only informal documentation is provided, for example, for `cache_write`: *"Create a cache write of original tensor, before storing into tensor."* Reasoning about schedules is difficult due to the lack of clear descriptions of optimization primitives.

If no schedule is provided (as in Listing 1), the compiler employs a set of implicit default optimizations that are out

Figure 1. Overview of our holistic functional approach to achieving high performance: Computations are expressed as *High-Level Programs* written in the data-parallel language RISE. These programs are rewritten following the instructions of an *Optimization Strategy* expressed in the strategy language ELEVATE. From the rewritten *Low-Level Programs* that encode optimizations explicitly, *High-Performance Code* is generated.




```
1  # Naive algorithm
2  k = tvm.reduce_axis((0, K), 'k')
3  A = tvm.placeholder((M, K), name='A')
4  B = tvm.placeholder((K, N), name='B')
5  C = tvm.compute((M, N),
6       lambda x,y: tvm.sum(A[x,k] * B[k,y],axis=k),name='C')
7  # Default schedule
8  s = tvm.create_schedule(C.op)
```

Listing 1: Matrix–matrix multiplication in TVM. Lines 2–6 define the computation $A \times B$, line 8 instructs the compiler to use the default schedule computing the output matrix sequentially in a row-major order.

```
1  # Optimized algorithm
2  k = tvm.reduce_axis((0, K),'k')
3  A = tvm.placeholder((M, K), name='A')
4  B = tvm.placeholder((K, N), name='B')
5  pB = tvm.compute((N/32,K,32), lambda x,y,z: B[y,x*32+z],
6                   name='pB')
7  C = tvm.compute((M,N),lambda x,y:tvm.sum(
8       A[x,k]*pB[y/32,k,tvm.indexmod(y,32)]axis=k),name='C')
9  # Parallel schedule
10 s = tvm.create_schedule(C.op)
11 CC = s.cache_write (C, 'global')
12 xo,yo,xi,yi = s[C].tile(C.op.axis[0], C.op.axis[1],32,32)
13 s[CC].compute_at (s[C], yo)
14 xc, yc = s[CC].op.axis
15 k, = s[CC].op.reduce_axis
16 ko, ki = s[CC].split(k, factor=4)
17 s[CC].reorder(ko, xc, ki, yc)
18 s[CC].unroll(ki)
19 s[CC].vectorize(yc)
20 s[C].parallel(xo)
21 x, y, z = s[pB].op.axis
22 s[pB].vectorize(z)
23 s[pB].parallel(x)
```

Listing 2: Optimized matrix–matrix multiplication in TVM. Lines 2–8 define an optimized version of the algorithm in Listing 1, the other lines define a schedule specifying the optimizations for targeting CPUs.

of reach for the user's control. The implicit optimizations sometimes lead to the surprising behavior that algorithms without a schedule perform better (e.g., due to auto-vectorization) than ones with a provided schedule.

2.2. The need for a principled way to separate, describe, and apply optimizations

Out of the shortcomings of scheduling APIs, we identify desirable features for a more principled way to separate, describe, and apply optimizations for high-performance code generation. Our holistic functional approach aims to do the following:

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 separated from the computational program facilitating reusability of programs and strategies.
3. *Enable composability*: Computations *and* strategies should be written as compositions of user-defined (possibly domain-specific) building blocks; *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.

In this paper, we present such an approach combining two functional languages: RISE and ELEVATE.

Figure 2 shows an example of a RISE program defining matrix multiplication computation as a composition of well-known data-parallel functional patterns. Below is an ELEVATE strategy that defines one possible optimization by applying the well-known tiling optimization. The optimization strategy is defined as a sequential composition (`;`) of user-defined strategies that are themselves defined as compositions of simple rewrite rules giving the strategy precise semantics. We do not use implicit behavior and instead generate low-level code according to the optimization strategy.

Figure 2. Matrix-matrix multiplication in RISE (top) and the tiling optimization strategy in ELEVATE (bottom).

```
// 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) )))) ) )
```

```
val tiledmm =            // Optimization Strategy in ELEVATE
  (tile(32, 32)`@`outermost (mapNest(2)) `;` lowerToC)(mm)
```

3. RISE: A LANGUAGE FOR EXPRESSING DATA-PARALLEL COMPUTATIONS

RISE is a functional programming language with data-parallel patterns for expressing computations over multidimensional arrays. RISE is a spiritual successor of LIFT[12,13,14] that has demonstrated that functional, high-performance code generation is feasible for different domains, including dense linear algebra, sparse linear algebra, and stencil computations.[6]

The top of Figure 2 shows an example of a RISE program using usual functional constructs of function abstraction (written `fun(x, e)`), application (written with parenthesis), identifiers, and literals (underlined). We use the following syntactic sugar: We write reverse function application as `e |> f` (equivalent to `f(e)`); we write function composition as `g << f` and in the reverse form as `f >> g`, both meaning `f` is applied before `g`. The type system allows to symbolically track the length of arrays using a restricted form of dependent types. Grammar and typing rules are given elsewhere.[5,2]

RISE defines a set of high-level primitives that describe computations over multidimensional arrays. These primitives are well known in the functional programming community: `fst`, `snd`, and the binary functions `add` and `mult` have their obvious meaning. `map` and `reduce` are the well-known higher-order functions operating on arrays and allowing for easy parallelization. `zip`, `split`, `join`, and `transpose` shape multidimensional array data in various ways.

A functional representation of hardware features. Besides the high-level primitives, RISE offers low-level primitives to indicate how to exploit the underlying hardware. Generally, programmers do not directly use these primitives; instead, they are introduced by rewrite rules. The `mapSeq` and `mapPar` patterns indicate if a function is applied to an array using a sequential or a parallel loop. Similarly, `reduceSeq` and `reduceSeqUnroll` indicate if the sequential reduction loop should be unrolled or not. RISE does not provide a parallel reduction as a building block because it is expressable using other low-level primitives such as `mapPar.toMem(a)(fun(x,b))` indicates that `a` is stored in memory and accessible in `b` with the name `x`. Three more low-level patterns, `mapVec`, `asVector`, and `asScalar`, enable the use of SIMD-style vectorization. The low-level primitives presented here are OpenMP-specific for expressing parallelization on CPUs; a similar set of low-level primitives exists for targeting the OpenCL programming language for GPUs.

Strategy-preserving code generation from RISE. The compilation of RISE programs is slightly unusual. A high-level program is rewritten using a set of rewrite rules into the low-level patterns. From the low-level representation, imperative parallel code is generated. This design leads to a clear separation of concerns—one of the key aims that we set out for our approach. All optimization decisions must be made in the rewriting stage before code generation. Atkey et al.[2] describe a code generation process that is guaranteed to be *strategy preserving*, meaning that no implicit implementation decisions are made. Instead, the compiler respects the implementation and optimization decisions

explicitly encoded in the low-level RISE program.

4. ELEVATE: A LANGUAGE FOR DESCRIBING OPTIMIZATION STRATEGIES

ELEVATE is a functional language for describing optimization strategies with a standard feature set, including recursion, algebraic data types, and pattern matching. It is heavily inspired by earlier work on strategy languages,[8] for example, Stratego,[16, 18] and complements our functional language RISE that describes computations. Our current implementation is a shallow-embedded DSL in Scala, and we use Scala-like notation for ELEVATE strategies in the paper.

4.1. Strategies

A *strategy* is the fundamental building block of ELEVATE encoding a program transformation as a function with type:

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

Here, `P` is the type of the rewritten program, such as `Rise` for RISE programs. A `RewriteResult` is an applicative error monad encoding the success or failure of applying a strategy:

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

In case of success, `Success` contains the rewritten program; otherwise, `Failure` contains the unsuccessful strategy.

The simplest examples of strategies are strategies that always succeed (`id`) and always fail (`fail`):

```
def id[P]:Strategy[P] = (p:P) => Success(p)
def fail[P]:Strategy[P] = (p:P) => Failure(fail)
```

4.2. Rewrite rules as strategies

In ELEVATE, rewrite rules are also strategies, that is, functions satisfying the type given above. The left-hand side of the well-known `mapFusion` rule (Figure 3) is expressed in RISE as:

```
val p: Rise = fun(xs, map(f)(map(g)(xs)))
```

The fusion rule is implemented in ELEVATE as follows:

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

We are mixing RISE (i.e., `map(f)`) and ELEVATE expressions and use `app(f, x)` to pattern-match the function application that we write as `f(x)` in RISE. The expression nested inside `Success` is the result of the rewrite rule application. Figure 3 shows all rewrite rules used as basic building blocks for expressing optimizations such as tiling, discussed later.

4.3. Strategy combinators

An idea that ELEVATE inherits from Stratego[17] is to describe strategies as compositions—one of our key aims. Therefore, we introduce strategy combinators.

The `seq` combinator composes two strategies `fs` and `ss` sequentially by applying the first strategy to the input program `p`, and then, the second strategy is applied to the result.

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

The `seq` strategy is successful when it applied both strategies successfully in succession; otherwise, `seq` fails. In our combinator's implementation, we use the monadic interface of `RewriteResult` and use the standard Haskell operators `>>=` for monadic bind, `<|>` for mplus, and `<$>` for fmap.

The `lChoice` combinator is given two strategies and applies the second one only if the first failed.

```
def lChoice[P]:
  Strategy[P] => Strategy[P] => Strategy[P] =
  fs => ss => p => fs(p) <|> ss(p)
```

We use `<+` as an infix operator for `lChoice` and `';'` for `seq`. Using these basic combinators, we define others such as `try` that applies a strategy and, in case of failure, applies the identity strategy. Therefore, `try` never fails.

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

`repeat` applies a strategy until it is no longer applicable.

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

4.4. Traversals as strategy transformers

In the implementation of the `mapFusion` strategy, the **match** statement will try to pattern-match its argument—the entire program. This means that a strategy on its own is hard to reuse in different circumstances.

In addition, a strategy is often applicable at multiple

Figure 3. Rewrite rules of high-level RISE expressions used for optimizations in this paper.

$$\epsilon \rightsquigarrow \texttt{id} \qquad \text{(addId)}$$

$$(\texttt{id} : m.n.\delta \to m.n.\delta) \rightsquigarrow \texttt{transpose >> transpose} \qquad \text{(idToTranspose)}$$

$$\texttt{transpose >> map(map(f))} \rightsquigarrow \texttt{map(map(f)) >> transpose} \qquad \text{(transposeMove)}$$

$$\texttt{map(f)} \rightsquigarrow \texttt{split(n) >> map(map(f)) >> join} \qquad \text{(splitJoin)}$$

$$\texttt{map(f >> g)} \leftrightsquigarrow \texttt{map(f) >> map(g)} \qquad \text{(mapFusion/mapFission)}$$

$$\texttt{map(f) >> reduce(fun((acc,y), op(acc)(y)))(init)} \leftrightsquigarrow \texttt{reduce(fun((acc,y), op(acc)(f(y))))(init)} \qquad \text{(fuseReduceMap/fissionReduceMap)}$$

places within the same program or only applicable at a specific location. For example, the `mapFusion` strategy is applicable twice in the following RISE program:

```
val maps3 = fun(xs, map(f)(map(g)(map(h)(xs))))
```

We may fuse the first or last two `map`s, as shown in Figure 4.

In ELEVATE, we use *traversals* to describe at which exact location a strategy is applied. Luttik and Visser[9] proposed two basic traversals encoded as strategy transformers:

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

Traversal `all` applies a given strategy to all sub-expressions of the current expression and fails if the strategy is not applicable to all sub-expressions. `one` applies a given strategy to exactly one sub-expression and fails if the strategy is not applicable to any sub-expression.

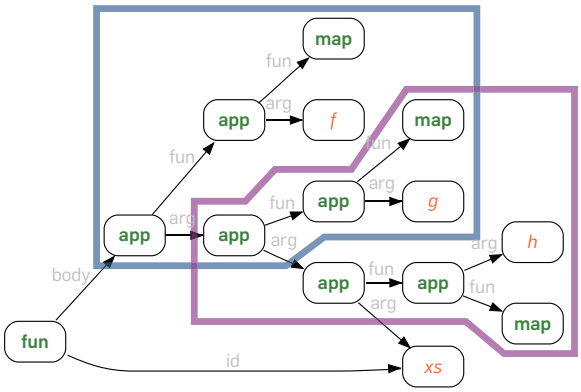
In ELEVATE, we view these basic traversals as a type class: an interface that has to be implemented for each program type `P`. The implementation for RISE is straightforward. RISE programs are represented by ASTs such as the one in Figure 4; therefore, `all` and `one` correspond to the obvious implementations on the tree-based representation.

To fuse the first two `map`s in Figure 4, we use the `one` traversal: `one(mapFusion)(maps3)`. This applies the `mapFusion` strategy, not at the root of the AST, but instead one level down, first trying to apply the strategy (unsuccessfully) to the function parameter and then (successfully) to the function body highlighted in the upper-right blue box.

To fuse the last two `map`s, we use the `one` traversal twice: `one(one(mapFusion))(maps3)`. This successfully applies the fusion strategy to the expression highlighted in the lower-right purple box in Figure 4.

The traversals we have discussed so far are not specific to RISE. These traversals are flexible but offer only limited control as for `one` the selection of sub-expressions is either nondeterministic or implementation-dependent (as for RISE).

Figure 4. Two possible locations for applying the *map-fusion* rule within the same program.



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

Particularly in the context of program optimization, it rarely makes sense to apply a strategy to `all` sub-expressions.

In ELEVATE, one can easily specify program language-specific traversals. As we have seen in the previous section, RISE is a functional language using λ-calculus as its representation. Therefore, it makes sense to introduce traversals that navigate the two core concepts of λ-calculus: `fun`ction abstraction and `app`lication. To apply a strategy to the body of a function abstraction, we define the `body` traversal that applies a strategy to the function body, and if successful, a function is built around the transformed body. Similarly, we define traversals `function` and `argument` for function applications.

For the RISE program shown in Figure 4, we can now describe a precise path in the AST. To fuse the first two `map`s, we may write `body(mapFusion)(maps3)`, and to fuse the others, we write `body(argument(mapFusion))(maps3)`.

All traversal primitives introduced so far apply their given strategies only to immediate sub-expressions. Using strategy combinators and traversals, we can define recursive strategies that traverse entire expressions:

```
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 tryAll[P]: Traversal[P] =
  s => p => (all(tryAll(try(s))) ';' try(s))(p)
```

`topDown` and `bottomUp` are useful strategies traversing an expression either from the top or from the bottom, trying to apply a strategy at every sub-expression and stopping at the first successful application. If the strategy is not applicable at any sub-expression, `topDown` and `bottomUp` fail. The `tryAll` strategy is often more useful as it wraps its given strategy in a `try` and thus never fails but applies the strategy wherever possible. Also, note that the `tryAll` strategy traverses the AST bottom-up instead of top-down.

4.5. Normalization

When implementing rewrite rules, such as the `mapFusion` rule as strategies, the match statement expects the input expression to be in a particular syntactic form. For a functional language like RISE, we might, for example, expect that expressions are fully β-reduced. To ensure that expressions satisfy a *normal-form*, we define:

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

The `normalize` strategy repeatedly applies a given strategy to every possible sub-expression until it cannot be applied anymore. Therefore, after `normalize` successfully finishes, it is not possible to apply the given strategy to any sub-expression any more.

For RISE specifically, we use in the following two normal forms whose implementation is explained in the original paper[5]:

1. the Beta-Eta-Normal-Form (`BENF`) transforms a RISE expression such that the standard lambda calculus

transformations β-reduction and η-reduction are no longer applicable, and
2. the Data-Flow-Normal-Form (DFNF) ensures a particular syntactic structure for higher-order RISE primitives like **map**. Specifically, DFNF ensures that a function abstraction is present in every higher-order primitive and that higher-order primitive are fully applied.

5. OPTIMIZATIONS AS STRATEGIES

In this section, we explore the use of ELEVATE to encode high-performance optimizations by leveraging its ability to define custom abstractions. We use TVM[4] as a comparison for a state-of-the-art imperative optimizing deep learning compiler with a scheduling API implemented in Python. TVM allows expressing computations using an EDSL (in Python) and controlling the application for optimizations using a separate scheduling API.

We start by expressing basic scheduling primitives such as **parallel** in ELEVATE. Then, we explore the implementation of more complex scheduling primitives like **tile** by composition in ELEVATE, whereas it is a built-in optimization in TVM. Following our functional approach, we express sophisticated optimization strategies as compositions of a small set of general rewrite rules resulting in a more principled and even more powerful design. Specifically, the tiling optimization strategy in ELEVATE can tile arbitrary many dimensions instead of only two, while being composed of only five RISE-specific rewrite rules.

5.1. Basic scheduling primitives as strategies

TVM's scheduling primitives **parallel**, **split**, and **unroll** specify loop transformations. We implement those as rewrite rules for RISE. The **parallel** primitive indicates the parallel computation of a particular loop. In RISE, this is indicated by transforming a high-level **map** into its low-level **mapPar** version as expressed in the following ELEVATE strategy:

```
def parallel: Strategy[Rise] = p => p match {
  case map => Success( mapPar )
  case _   => Failure( parallel )}
```

We define a strategy for the sequential **mapSeq** similarly.

TVM's **split** primitive implements loop blocking. In RISE, this is achieved by rewriting the computation over an array expressed by **map**(f): First, the input is split into a two-dimensional array using **split**(n), then f is **map**ped twice to apply the function to all elements of the now nested array, and finally, the resulting array is attended into the original one-dimensional form using **join**.

```
def split(n:Int):Strategy[Rise] = p => p match {
  case app(map, f) =>
    Success( split(n) >> map(map(f)) >> join )
  case _ => Failure( split(n) ) }
```

The unroll strategy rewrites the high-level **map** and **reduce** primitives into specific RISE low-level primitives that will be unrolled by the RISE compiler during code generation.

5.2. Multidimensional tiling as a strategy

Tiling is a crucial optimization improving the cache hit rate by exploiting locality within a small neighborhood of elements. TVM's **tile** is a more complicated scheduling primitive to implement because it is essentially a combination of two traditional loop transformations: loop blocking and loop interchange. In fact, **tile** in TVM is a built-in combination of **split** for loop blocking and **reorder** for loop interchange. We already saw how to implement **split** using ELEVATE. We will now discuss how to implement a tile strategy using a combination of rules, normal-forms, and domain-specific traversals. Where TVM only implements 2D tiling, our generalized strategy tiles an arbitrary number of dimensions.

We require five basic rules for expressing our multidimensional tiling strategy: splitJoin, addId, idToTranspose, transposeMove, and mapFission (all shown in Figure 3). In addition, we require three standard λ-calculus-specific transformations: η- and β-reduction, and η-abstraction. We implement these rules as basic ELEVATE strategies.

Listing 3 shows the ELEVATE implementation of the tiling optimization. The multidimensional tileND strategy expects a list of tile sizes, one per tiled dimension. The intuition for the implementation is simple: First, we ensure that the required rules are applicable to the input expression by normalizing the expression using the DFNF normal form. Then, we apply the previously introduced split strategy to every **map** to be blocked, recursively going from the innermost to outermost, as explained below. Finally, the interchange strategy rearranges the blocked loops in the expected final order; this strategy is explained in detail in the original paper.[5]

To recursively apply the loop blocking strategy to nested **map**s, we make use of the RISE-specific traversal fmap:

```
def fmap: Traversal[Rise] = s =>
  function(argOf(map, body(s)))
```

fmap traverses to the function argument of a **map** primitive and applies the given strategy s. Note that the strategy requires the function argument of a **map** primitive to be a function abstraction, which we can assume because we normalize the expression using DFNF. The fmap strategy is useful because it can be nested to "push" the application of the given strategy inside of a **map**-nest. For example,

```
fmap(fmap(split(n)))(DFNF(map(map(map(f)))))
```

```
1 def tileND(n: List[Int]): Strategy[Rise] =
2   DFNF ';' (n.size match {
3     case 1 => function(split(n.head))          // loop-blocking
4     case i => fmap (tileND(d-1)(n.tail)) ';' // recurse
5               function (split(n.head)) ';'    // loop-blocking
6               interchange (i)      })         // loop-reorder
```

Listing 3: ELEVATE strategy implementing tiling recursively for arbitrary dimensions.

skips two **map**s and applies loop blocking to the innermost **map**. In Listing 3 line 4, we use `fmap` to recursively call `tileND` applying loop blocking first to the inner **map**s before to the outer **map**.

5.3. Abstractions for describing locations in RISE

In TVM, named identifiers describe locations at which optimizations should be applied. For example, TVM's **split** is invoked with an argument specifying the loop to block:

```
1  k,    = s[C].op.reduce_axis
2  ko,ki = s[C].split(k, factor=4)
```

Using identifiers ties schedules to computational expressions and makes reusing schedules hard. ELEVATE does not use names to identify locations, but instead uses the traversals defined in Section 4. This is another example of how we facilitate reuse—one of the key aims of our approach.

By using ELEVATE's traversal strategies, we apply the basic scheduling strategies in a much more flexible way: For example, **topDown**(`parallel`) traverses the AST from top to bottom and will thus always parallelize the outermost **map**, corresponding to the outermost for loop. **tryAll**(`parallel`) traverses the whole AST instead, and all **map**s are parallelized.

In order to apply optimizations on large ASTs, it is often insufficient to use the **topDown** or **tryAll** traversals. For example, we might want to block a specific loop in a loop nest. None of the introduced traversals so far allow the description of a precise loop conveniently, or rather a precise location, required for these kinds of optimizations. Strategy *predicates* allow us to describe locations in a convenient way. A strategy predicate checks the program for a syntactic structure and returns **Success** without changing the program if the structure is found. Two simple examples for strategy predicates are `isReduce` and `isApp` that check if the current node is a **reduce** primitive or an applied function, respectively. These predicates can be composed with the regular traversals to define precise locations. The '**@**' strategy allows us to describe the application of strategies at precise locations conveniently:

```
def '@'[P](s:Strategy[P],t:Traversal[P]) = t(s)
```

We write this function in infix notation.

The left-hand side of the '**@**' operator specifies the strategy to apply, and the right-hand side specifies the precise location as a traversal. This nicely separates the strategy to apply from the traversal describing the location. This abstraction is especially useful for complex optimization strategies with nested location descriptions. For RI**SE**, we specify multiple useful traversals and predicates, which can be extended as needed. Two useful ones are `outermost` and `mapNest` that are defined as follows:

```
def outermost: Strategy[Rise] => Traversal[Rise]
    = pred => s => topDown(pred `;` s)
def mapNest(d: Int): Strategy[Rise] = p =>
  (d match {
    case x if x == 0 => Success(p)
    case x if x < 0 => Failure(mapNest(d))
    case _ => fmap(mapNest(d-1))(p)})
```

`outermost` traverses from top to bottom visiting nested primitives from outermost to innermost, trying to apply the predicate. If the predicate is applied successfully, it applies the given strategy `s`. Similarly, we define function `innermost`, which instead uses **bottomUp**. The `mapNest` predicate recursively traverses a `DFNF`-normalized **map** nest of a given depth using nested `fmap` traversals. If the traversal is successful, a **map** nest of depth d has been found.

By combining these abstractions, we conveniently describe applying the tiling optimization to the two outermost loop nests elegantly in ELEVATE:

```
(tile(32,32) `@`outermost(mapNest(2)))(mm)
```

6. EXPERIMENTAL EVALUATION

In this section, we evaluate our functional approach to high-performance code generation. We use ELEVATE strategies to describe optimizations that are equivalent to TVM schedules using matrix–matrix multiplication as our case study. We compare the performance achieved using code generated by the RI**SE** compiler and code generated by TVM. The original paper[5] also includes a performance comparison with Halide.

6.1. Optimizing matrix-matrix multiplication

For our case study of matrix-matrix multiplication, we follow a tutorial from the TVM authors that discuss seven progressively optimized versions: *baseline, blocking, vectorized, loop permutation, array packing, cache blocks*, and *parallel*. For each TVM schedule, we developed an equivalent ELEVATE strategy using the TVM-like scheduling abstractions and the traversal utilities. The *vectorized, loop permutation* and *cache blocks* versions are not discussed here for brevity but are discussed in the original paper[5]; we discuss the rest here.

Baseline. For the *baseline* version, TVM uses a default schedule, whereas ELEVATE describes the implementation decisions explicitly as shown in Figure 5—one of the key aims that we set out for our approach.

The TVM algorithm shown in Listing 1 computes the dot product in a single statement in line 6. The RI**SE** program shown at the top of Figure 5 describes the dot

Figure 5. RISE matrix multiplication expression (top) and *baseline* strategy in ELEVATE (bottom).

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

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

product with separate `map` and `reduce` primitives, which are fused as described in the ELEVATE program below using the `fuseReduceMap` rewrite rules from Figure 3. The `lowerToC` strategy rewrites `map` and `reduce` into their sequential versions. Both systems generate equivalent C code of two nested loops iterating over the output matrix and a single nested reduction loop performing the dot product.

Blocking. For the *blocking* version, we reuse the `baseline` and `lowerToC` strategy, but first, we use the abstractions built in the previous sections, emulating the TVM schedule as shown in Listings 4 and 5. We first apply `tile`, then `split`, and then `reorder`, just as specified in the TVM schedule. To `split` the reduction, we need to fission the fused map and reduce primitives again using `fissionReduceMap`. We describe locations using `outermost` and `innermost` applying `tile` to the outermost `map`s and `split` to the nested reduction. In contrast to TVM, for `reorder`, we identify dimensions by index rather than by name. We introduce the '`;;`' combinator for convenience denoting that we apply `DFNF` to normalize intermediate expressions between each step.

Array packing. As already discussed in the motivation section, some optimizations are not expressible in TVM's scheduling API without changing the algorithm—clearly violating the separation between specifying computations and optimizations. Specifically, the *array packing* optimization permutes the B matrix's elements in memory improving memory access patterns by introducing an additional computation `pB` in Listing 2 line 6, before using it in line 8.

For our implementation of the *array packing* optimization, we are not required to change the RI**SE** program, but define and apply the array packing of matrix *B* simply as another rewrite step in ELEVATE (Listing 6 line 10). Our `arrayPacking` strategy is itself composed out of other strategies, for example, `storeInMemory` or `loopPerm`, which are left out for brevity here but are explained in detail in the original paper.[5]

Parallel. The TVM version *parallel* changes the algorithm yet again to introduce a temporary buffer (`CC` in Listing 2 line 11) for the accumulation along the K-dimension to improve the cache writing behavior and unrolls the inner reduction loop. For expressing the *parallel* version in ELEVATE (Listing 6 line 12), we reuse the `arrayPacking` strategy (line 13) while adding strategies for unrolling the innermost reduction (line 16) and parallelizing the outermost loop (line 14).

```
1  val appliedReduce = isApp(isApp(isApp(isReduce)))
2  val blocking = ( baseline ';'
3    tile(32,32)       '@' outermost (mapNest(2))    ';;'
4    fissionReduceMap '@' outermost (appliedReduce)';;'
5    split(4)          '@' innermost (appliedReduce)';;'
6    reorder(List(1,2,5,6,3,4)))
7  (blocking ';' lowerToC)(mm)
```

Listing 4: ELEVATE *blocking* strategy

```
1  # blocking version
2  xo, yo, xi, yi = s[C].tile(
3    C.op.axis[0],C.op.axis[1],32,32)
4  k,              = s[C].op.reduce_axis
5  ko, ki          = s[C].split(k, factor=4)
6  s[C].reorder(xo, yo, ko, ki, xi, yi)
```

Listing 5: TVM *blocking* schedule

6.2. Rewriting overhead and performance

We now investigate the scalability, overhead, and performance of our functional rewrite-based approach.

Scalability and overhead. To evaluate scalability and the overhead of rewriting, we are counting the number of successfully applied rewrite steps performed when applying a strategy to the RI**SE** matrix multiply expression. We count every intermediate step, which includes traversals as these are implemented as rewrite steps too.

Figure 6 shows the number of rewrites for each version. No major optimizations are applied to the *baseline* version, and 657 rewrite steps are performed. However, as soon as interesting optimizations are applied, we reach about 40,000 steps for the next three versions and about 63,000 for the most complicated optimizations. These high numbers clearly show the scalability of our compositional approach, in which complex optimizations are composed of a small set of fundamental building blocks. It also shows that abstractions are required to control this many rewrite steps. The high-level strategies encode practical optimizations and hide massive numbers of individual rewrite steps that are performed. Applying the strategies to the RI**SE** expression took less than two seconds per version on a commodity laptop, demonstrating the moderate overhead of our unoptimized implementation.

Performance comparison against TVM. Finally, we are interested in the performance achieved when optimizing

```
1  val appliedMap = isApp(isApp(isMap))
2  val isTransposedB = isApp(isTranspose)
3
4  val packB = storeInMemory(isTransposedB,
5   permuteB ';;'
6   vectorize(32) '@' innermost(appliedMap) ';;'
7   parallel      '@' outermost(isMap)
8  ) '@' inLambda
9
10 val arrayPacking = packB ';;' loopPerm
11
12 val par = (
13   arrayPacking ';;'
14   (parallel '@' outermost(isMap))
15             '@' outermost(isToMem) ';;'
16    unroll   '@' innermost(isReduce))
17
18 (par ';' lowerToC)(mm)
```

Listing 6: ELEVATE *parallel* strategy

Figure 6. Total number of successful rewrite steps when applying different optimization strategies.

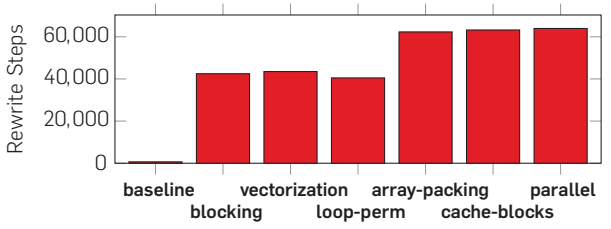
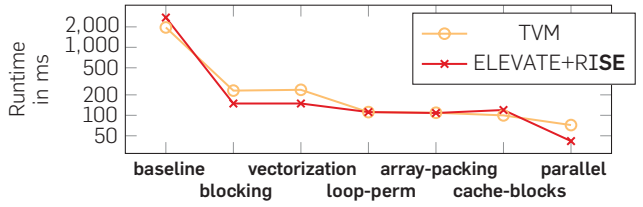


Figure 7. Performance of TVM- vs. RISE-generated code that has been optimized by ELEVATE strategies.



RISE programs with ELEVATE compared with TVM. Ideally, the RISE code optimized with ELEVATE should be similar to the TVM-generated code and achieve competitive performance. We performed measurements on an Intel multicore CPU. For a detailed description of the experimental setup see the original paper.[5]

Figure 7 shows the performance of RISE- and TVM-generated code. The code generated by RISE controlled by the ELEVATE optimization strategies performs competitively with TVM. Our experiment indicates a matching trend across versions compared with TVM, showing that our ELEVATE strategies encode the same optimizations used in the TVM schedules. The most optimized parallel RISE generated version improves the performance over the baseline by about 110×. The strategies developed in an extensible style by composing individual rewrite steps using ELEVATE are practically usable and provide competitive performance for important high-performance code optimizations.

7. CONCLUSION

In this paper, we presented a holistic functional approach to high-performance code generation. We presented two functional languages: RISE for describing computations as compositions of data-parallel patterns and ELEVATE for describing optimization strategies as composition of rewrite rules. We showed that our approach successfully: *separates concerns* by truly separating the computation and strategy languages; *facilitates reuse* of computational patterns as well as rewrite rules; *enables composability* by building programs as well as rewrite strategies as compositions of a small number of fundamental building blocks; *allows reasoning* about programs and strategies with well-defined semantics; and *is explicit* by empowering users to be in control over the optimization strategy that is respected by our compiler. In contrast to existing imperative systems with scheduling APIs such as Halide and TVM, programmers are not restricted to apply a set of built-in optimizations but define their own optimization strategies. Our experimental evaluation demonstrates that our holistic functional approach achieves competitive performance compared with the state-of-the-art code generator TVM. C

References

1. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorow.org.
2. Atkey, R., Steuwer, M., Lindley, S., Dubach, C. Strategy preserving compilation for parallel functional code. *CoRR*, abs/1710.08332, 2017.
3. Barham, P., Isard, M. Machine learning systems are stuck in a rut. In *HotOS*. ACM, 2019, 177–183.
4. Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E.Q., Shen, H., Cowan, M., Wang, L., Hu, Y., Ceze, L., Guestrin, C., Krishnamurthy, A. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018*. (Carlsbad, CA, USA, October 8–10, 2018), 2018, 578–594.
5. Hagedorn, B., Lenfers, J., Koehler, T., Qin, X., Gorlatch, S., Steuwer, M. Achieving high-performance the functional way: A functional pearl on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang. 4*, (ICFP), 2020.
6. Hagedorn, B., Stoltzfus, L., Steuwer, M., Gorlatch, S., Dubach, C. High performance stencil code generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization, CGO 2018*, (Vösendorf/Vienna, Austria, February 24–28, 2018), 2018, 100–112.
7. Hennessy, J.L., Patterson, D.A. A new golden age for computer architecture. *Commun. ACM 62*, 2 (2019), 48–60.
8. Kirchner, H. Rewriting strategies and strategic rewrite programs. In *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*, 2015, 380–403.
9. Luttik, S.P., Visser, E., et al. *Specification of rewriting strategies*. Universiteit van Amsterdam. Programming Research Group, 1997.
10. Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., Lerer, A. Automatic differentiation in pytorch. 2017.
11. Ragan-Kelley, J., Adams, A., Sharlet, D., Barnes, C., Paris, S., Levoy, M., Amarasinghe, S. P., Durand, F. Halide: Decoupling algorithms from schedules for high-performance image processing. *Commun. ACM 61*, 1 (2018), 106–115.
12. Steuwer, M., Fensch, C., Lindley, S., Dubach, C. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance opencl code. In *ICFP*. ACM, 2015, 205–217.
13. Steuwer, M., Remmelg, T., Dubach, C. Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation. In *CASES*. ACM, 2016, 15:1–15:10.
14. Steuwer, M., Remmelg, T., Dubach, C. Lift: A functional data-parallel IR for high-performance GPU code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017* (Austin, TX, USA, February 4–8, 2017), 2017, 74–85.
15. TVM. How to optimize gemm on cpu, 2020.
16. Visser, E. Stratego: A language for program transformation based on rewriting strategies. In *Rewriting Techniques and Applications, 12th International Conference, RTA 2001, Utrecht, The Netherlands, May 22–24, 2001, Proceedings*, 2001, 357–362.
17. Visser, E. Program transformation with Stratego/XT. In *Domain-specific program generation*. Springer, 2004, 216–238.
18. Visser, E., Benaissa, Z., Tolmach, A.P. Building program optimizers with rewriting strategies. In *Proceedings of the third ACM SIGPLAN International Conference on Functional Programming (ICFP '98)* (Baltimore, Maryland, USA, September 27–29, 1998), 1998, 13–26.

Bastian Hagedorn (bhagedorn@nvidia.com), NVIDIA, Würselen, Germany.

Johannes Lenfers, Sergei Gorlatch ({j.le, gorlatch}@wwu.de), University of Münster, Germany.

Thomas Koehler (thomas.koehler@thok.eu), University of Glasgow, U.K.

Xueying Qin, Michel Steuwer (xueying.qin, michel.steuwer@ed.ac.uk), The University of Edinburgh, U.K.