
# 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**.

Additional Key Words and Phrases: Rewrite Rules, Optimization Strategies, Strategy Languages, ELEVATE

**ACM Reference Format:**
Bastian Hagedorn, Johannes Lenfers, Thomas Kœhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. 2020. Achieving High-Performance the Functional Way: A Functional Pearl on Expressing High-Performance Optimizations as Rewrite Strategies. *Proc. ACM Program. Lang.* 4, ICFP, Article 92 (August 2020), 29 pages. https://doi.org/10.1145/3408974

Authors' addresses: Bastian Hagedorn, University of Münster, Germany, b.hagedorn@wwu.de; Johannes Lenfers, University of Münster, Germany, j.le@wwu.de; Thomas Kœhler, University of Glasgow, UK, t.koehler.1@research.gla.ac.uk; Xueying Qin, University of Glasgow, UK, 2335466q@student.gla.ac.uk; Sergei Gorlatch, University of Münster, Germany, gorlatch@wwu.de; Michel Steuwer, University of Glasgow, UK, michel.steuwer@glasgow.ac.uk.


2475-1421/2020/8-ART92
https://doi.org/10.1145/3408974

## 1 INTRODUCTION

The tremendous gains in performance and efficiency that computer hardware continues to make are a vital driving force for innovation in computing. These improvements enables entire new areas of computing, such as deep learning, to deliver applications unthinkable even just a few years ago. Moore's law and Denard's scaling describe the exponential growth of transistor counts leading to improved performance and the exponential growth in performance per watt, leading to improved energy efficiency. Unfortunately, these laws are coming to an end, as observed in the 2017 ACM Turing Lecture by Hennessy and Patterson [2019]. As a result, the performance and energy efficiency gains no longer come for free for software developers. Programs have to be optimized for an increasingly diverse set of hardware devices by exploiting many subtle details of the computer architecture. Therefore, performance portability has emerged as a crucial concern as software naturally outlives the faster cycle of hardware generations. The accelerating trend towards specialized hardware, which offers extreme benefits for performance and energy efficiency if the specially 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. This entanglement makes them – by design – non performance portable. As an alternative, higher-level domain-specific approaches have emerged that allow programmers to declaratively describe the functional behavior without committing to a specific implementation. Prominent examples of this approach are virtually all machine learning systems such as TensorFlow [Abadi et al. 2015] or PyTorch [Paszke et al. 2017]. For these approaches, the compilers and runtime systems are responsible for optimizing the computations expressed as data-flow graphs. Programmers have limited control over the optimization process. Instead, large teams of engineers at Google and Facebook provide fast implementations for the most common hardware platforms, 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 Barham and Isard [2019], two of the original authors of TensorFlow.

TVM by Chen et al. [2018] and Halide by Ragan-Kelley et al. [2018, 2013] are two imperative state-of-the-art high-performance domain-specific code generators used in machine learning and image processing. Both of these systems 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 is unintuitive in some cases due to the lack of precise semantics, and both compilers have default and implicit behavior limiting experts' control. All of these reasons make writing schedules significantly harder than writing algorithms. 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, we will need to rethink how we separate, describe, and apply optimizations in a more principled way.

What has the functional programming community to offer to solve this challenge? Encoding program transformations as *rewrite rules* has been a long-established idea that emerged from the functional community. Bird and de Moor [1997] studied an algebraic programming approach where functional programs are rewritten by exploiting algebraic properties. The Glasgow Haskell

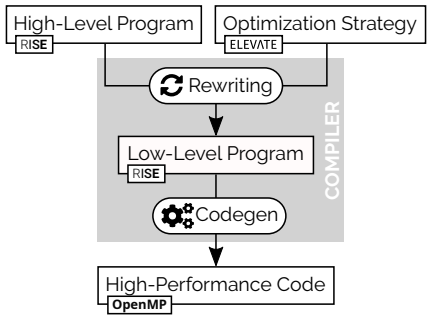


Fig. 1. Overview of our holistic functional approach to achieving high-performance: Computations are expressed as *High-Level Programs* written in the data-parallel language RISE. These programs are rewritten following the instructions of an *Optimization Strategy* expressed in the strategy language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

Compiler allows the specification of rewrite rules for program optimizations [Peyton Jones et al. 2001]. More recently, Lift by Steuwer et al. [2015] encodes optimization and implementation choices as rewrite rules for optimizing a high-level pattern-based data-parallel functional language using an automated stochastic search method applying the rewrites. Rewrite based approaches, such as Lift, have the advantage of being easily extensible towards new application domains (such as stencils discussed by Hagedorn et al. [2018]) as well as supporting new hardware features (such as specialized vector instructions which are encoded as new low-level patterns and introduced by a rewrite rule described by Steuwer et al. [2016]). Unfortunately, these rewrite approaches are so far limited in their practicality to deliver the high level of performance required in many real-world applications and achieved by current imperative approaches.

In this paper, we describe a practical holistic functional approach to high-performance code generation. We combine RISE, a data-parallel functional language for expressing computations, with a functional strategy language, called ELEVATE. RISE 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. We provide a purely functional high-performance code generation solution that is practical and allows expert programmers to precisely control optimizations. As we will see in our experimental results, our approach provides competitive performance compared to the imperative state-of-the-art while being built with and leveraging functional principles resulting in an elegant and composable design.

While the individual components of our approach are not necessary novel for the functional community, we believe that our overall design demonstrates an interesting novel application of functional-programming techniques for achieving high performance. RISE is inspired by functional data-parallel languages such as Lift [Steuwer et al. 2015, 2017], Accelerate by Chakravarty et al. [2011], and Futhark by Henriksen et al. [2017]. ELEVATE is inspired by strategy languages for rewrite systems – mainly unknown to the high-performance community – such as Stratego by Visser [2001a]. Kirchner [2015] provides a recent overview of the rewriting community's research.

Figure 1 shows an overview of our approach showing the compilation flow from a high-level program and an optimization strategy to high-performance code. In Section 2, we first motivate the need for a more principled way to separate, describe, and apply optimizations. RISE and its compilation is explained in Section 3.3 before we focus on ELEVATE (Section 4) and how optimization strategies are expressed in it (Section 5). We present an experimental evaluation comparing with TVM and Halide in Section 6. We finish with related work in Section 7 and a conclusion (Section 8).

```
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),lambda x, y: tvm.sum(A[x, k] * B[k, y], axis=k),name='C')
6  # Default schedule
7  s = tvm.create_schedule(C.op)
```

Listing 1 Matrix matrix multiplication in TVM. Lines 2–5 define the computation $A \times B$, line 7 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], name='pB')
6  C = tvm.compute((M,N), lambda x,y:tvm.sum(A[x,k] * pB[y//32,k,tvm.indexmod(y,32)], axis=k),name='C')
7  # Parallel schedule
8  s = tvm.create_schedule(C.op)
9  CC = s.cache_write(C, 'global')
10 xo, yo, xi, yi = s[C].tile(C.op.axis[0], C.op.axis[1], 32, 32)
11 s[CC].compute_at(s[C], yo)
12 xc, yc = s[CC].op.axis
13 k, = s[CC].op.reduce_axis
14 ko, ki = s[CC].split(k, factor=4)
15 s[CC].reorder(ko, xc, ki, yc)
16 s[CC].unroll(ki)
17 s[CC].vectorize(yc)
18 s[C].parallel(xo)
19 x, y, z = s[pB].op.axis
20 s[pB].vectorize(z)
21 s[pB].parallel(x)
```

Listing 2 Optimized Matrix matrix multiplication in TVM. Lines 2–6 define an optimized version of the algorithm in Listing 1, the other lines define a schedule specifying the optimizations for targeting CPUs.

## 2 MOTIVATION AND BACKGROUND

We motivate the need for more principled optimizations with a closer look at TVM, the current state-of-the-art in high-performance domain-specific compilation for machine learning. We then argue for achieving high-performance code generation using well-known functional programming techniques by separating computations and optimizations into two separate functional languages.

### 2.1 Scheduling Languages for High-Performance Code Generation

Halide by Ragan-Kelley et al. [2018] introduced into the domain of high-performance code generation the concept of decoupling a program into: the *algorithm*, describing the functional behavior, and the *schedule*, specifying how the underlying compiler should optimize the program. It has been designed to generate high-performance code for image processing pipelines [Ragan-Kelley et al. 2013] and has since inspired similar approaches such as TVM by Chen et al. [2018] in deep learning.

Listings 1 and 2 [TVM 2020] show two snippets of TVM Python code for generating matrix multiplication implementations. Listing 1 shows a simple version. The lines 2–5 define the matrix multiplication 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 reducing over the reduction domain $k$ using the `tvm.sum` operator (line 5). Line 7 instructs the compiler to use the default schedule which generates code to compute the output matrix sequentially.

*Modifications for Optimizing Performance.* Listing 2 shows an optimized version of the same computation. The schedule in lines 8–21 specifies multiple program transformations including tiling (line 10), vectorization (line 17), and loop unrolling (line 16) for optimizing the performance on multi-core CPUs. However, in order to optimize the memory access pattern, the algorithm has to be changed. In this example, a copy of the `B` matrix (`pB`) is introduced in line 5 (and used in line 6) 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 – clearly violating the promise of separating algorithm and schedule. Even for optimizations that do not require changing the algorithm, the separation between algorithm and schedule is blurred because both share the same Python identifiers and must, therefore, live in the same scope. This unsharp separation limits the reuse of schedules across algorithms.

The optimized parallel schedule uses eight built-in optimization primitives (`cache_write`, `tile`, `compute_at`, `split`, `reorder`, `unroll`, `vectorize`, `parallel`). Scheduling primitives provide high-level abstractions for typical program transformations aiming to optimize performance. Some are specific for the hardware (like `vectorize`), some are generally useful algorithmic optimizations 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. Adding a new optimization primitive to the existing schedule API requires extending the underlying TVM compiler. Even a primitive like `tile`, which can be implemented as a composition of `split` and `reorder` [Halide 2020], is provided as a built-in abstraction. Modern scheduling languages are not extensible with user-defined abstractions.

The behavior of some primitives is not intuitive, and the documentation provides only informal descriptions, e.g., 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 TVM compiler employs a set of implicit default optimizations that are out 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 where a programmer provides a schedule.

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

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

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

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

In this paper, we present such an approach combining the functional languages: RI**SE** and ELEVATE.

```
// 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
```

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

Fig. 3. Matrix matrix multiplication in RISE (top) and the tiling optimization strategy in ELEVATE (bottom).

Figure 3 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 that improves memory usage by increasing spatial and temporal locality of the data. 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 a precise semantics. We do not employ implicit behavior and instead generate low-level code according to the optimization strategy.

In the remainder of the paper, we explore our holistic functional approach for achieving high-performance. We describe how to define optimizations typically provided in high-performance scheduling languages as optimization strategies in ELEVATE defined as compositions of simple rewrite rules for data-parallel functional programs. We start by briefly introducing the computational language RISE and its compilation to parallel imperative code.

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

In this section, we give a brief introduction of RISE (Section 3.1), how low-level patterns represent hardware features (Section 3.2), and finally, how we generate imperative code (Section 3.3).

### 3.1 A Brief Introduction to RISE

RISE is a functional programming language that uses data-parallel patterns to express computations over multi-dimensional arrays. RISE is a spiritual successor of Lift, which was initially introduced by Steuwer et al. [2015]. Lift has demonstrated that functional, high-performance code generation is feasible for different domains, including dense linear algebra, sparse linear algebra, and stencil computations (see [Steuwer et al. 2017] and [Hagedorn et al. 2018]). RISE is implemented as a deep embedded DSL in Scala and generates parallel OpenMP code for CPUs and OpenCL for GPUs.

Figure 4 shows the abstract syntax of RISE expressions and types, and the provided high-level and low-level primitives for expressing data-parallel computations. RISE provides the usual $\lambda$-calculus constructs of abstraction (written `fun(x, e)`), application (written with parenthesis), identifiers, and literals (underlined). The type system separates data types from function types to prevent functions from being stored in memory. RISE uses a restricted form of dependent function types for types that contain expressions of kind *nat* representing natural numbers or *data* for type-level variables ranging over data types. Natural numbers are used to represent the length of arrays in the type and might consist of arithmetic formulae with binary operations such as addition and multiplication. Data types are array types, pair types, index types representing array indices up to $n$, scalar types, or vector types that correspond directly to the SIMD vector types of the underlying hardware. Atkey et al. [2017] give precise typing rules for such a type system.

**RISE Syntax of Expressions and Types:**

$$
\begin{aligned}
e &::= \texttt{fun}(x, e) | e(e) | x | \underline{l} | P && \text{(Abstraction, Application, Identifier, Literal, Primitives)}\\
\kappa &::= nat | data && \text{(Natural Number Kind, Datatype Kind)}\\
\tau &::= \delta | \tau \to \tau | (x : \kappa) \to \tau && \text{(Data Type, Function Type, Dependent Function Type)}\\
n &::= \underline{0} | n + n | n \cdot n | \ldots && \text{(Natural Number Literals, Binary Operations)}\\
\delta &::= n.\delta | \delta \times \delta | idx[n] | \texttt{float} | n\texttt{<float>} && \text{(Array Type, Pair Type, Index Type, Scalar Type, Vector Type)}
\end{aligned}
$$

**High-Level Primitives**:

$$
\begin{aligned}
\texttt{id} &: (\delta : data) \to \delta \to \delta\\
\texttt{add} \mid \texttt{mult} \mid \ldots &: (\delta : data) \to \delta \to \delta \to \delta\\
\texttt{fst} &: (\delta_1\ \delta_2 : data) \to \delta_1 \times \delta_2 \to \delta_1\\
\texttt{snd} &: (\delta_1\ \delta_2 : data) \to \delta_1 \times \delta_2 \to \delta_2\\
\texttt{map} &: (n : nat) \to (\delta_1\ \delta_2 : data) \to (\delta_1 \to \delta_2) \to n.\delta_1 \to n.\delta_2\\
\texttt{reduce} &: (n : nat) \to (\delta : data) \to (\delta \to \delta \to \delta) \to \delta \to n.\delta \to \delta\\
\texttt{zip} &: (n : nat) \to (\delta_1\ \delta_2 : data) \to n.\delta_1 \to n.\delta_2 \to n.(\delta_1 \times \delta_2)\\
\texttt{split} &: (n\ m : nat) \to (\delta : data) \to nm.\delta \to n.m.\delta\\
\texttt{join} &: (n\ m : nat) \to (\delta : data) \to n.m.\delta \to nm.\delta\\
\texttt{transpose} &: (n\ m : nat) \to (\delta : data) \to n.m.\delta \to m.n.\delta\\
\texttt{generate} &: (n : nat) \to (\delta : data) \to (idx[n] \to \delta) \to n.\delta
\end{aligned}
$$

**Low-Level Primitives**:

$$
\begin{aligned}
\texttt{map}\{\texttt{Seq}|\texttt{SeqUnroll}|\texttt{Par}\} &: (n : nat) \to (\delta_1\ \delta_2 : data) \to (\delta_1 \to \delta_2) \to n.\delta_1 \to n.\delta_2\\
\texttt{reduce}\{\texttt{Seq}|\texttt{SeqUnroll}\} &: (n : nat) \to (\delta_1\ \delta_2 : data) \to (\delta_1 \to \delta_2 \to \delta_1) \to \delta_1 \to n.\delta_2 \to \delta_1\\
\texttt{toMem} &: (\delta_1\ \delta_2 : data) \to \delta_1 \to (\delta_1 \to \delta_2) \to \delta_2\\
\texttt{mapVec} &: (n : nat) \to (\delta_1\ \delta_2 : data) \to (\delta_1 \to \delta_2) \to n\texttt{<}\delta_1\texttt{>} \to n\texttt{<}\delta_2\texttt{>}\\
\texttt{asVector} &: (n\ m : nat) \to (\delta : data) \to nm.\delta \to n.m\texttt{<}\delta\texttt{>}\\
\texttt{asScalar} &: (n\ m : nat) \to (\delta : data) \to n.m\texttt{<}\delta\texttt{>} \to nm.\delta
\end{aligned}
$$

Fig. 4. The syntax of expressions and types of RISE as well as high- and low-level primitives.

In the paper, we use the following syntactic sugar: we write reverse function application in the style of F-sharp as `e |> f` (equivalent to `f(e)`); we write function composition like this `g << f` and in the reverse form as `f >> g`, both meaning `f` is applied before `g`; we may write `+` and `*` as inline binary operators instead of calling the equivalent functions **`add`** and **`mult`**.

RISE defines a set of high-level primitives that describe computations over multi-dimensional arrays. These primitives are well-known in the functional programming community: **`id`**, **`fst`**, **`snd`**, and the binary functions **`add`** and **`mult`** have their obvious meaning. **`map`** and **`reduce`** are the well-known functions operating on arrays and allowing for easy parallelization. **`zip`**, **`split`**, **`join`**, and **`transpose`** shape multi-dimensional array data in various ways. Finally, **`generate`** creates an array based on a generating function. Since RISE does not support general recursion, every RISE program terminates.

### 3.2 A Functional Representation of Hardware Features

More interesting are the low-level primitives that RISE offers to indicate how to exploit the underlying hardware. Generally, programmers do not directly use these primitives; instead, they are introduced by rewrite rules. The different **`map`** pattern variations indicate if the given function

is applied to the array using a sequential loop, by unrolling this loop, or using a parallel loop where each iteration might be performed in parallel. Similarly, the `reduce` variations indicate if the 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 the value `a` will be stored in memory and that the stored value is accessible in the expression `b` with the name `x`. The last three 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.

### 3.3 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. Steuwer et al. [2015] initially proposed this process in Lift. From the low-level representation, we generate imperative parallel code. This design leads to a clear separation of concerns – one of the key aims that we set out for our approach. All optimization decisions, such as how to parallelize the `reduce` primitive, must be made in the rewriting stage before code generation. The code generation process becomes deterministic and only translates the annotated implementation strategy into the target imperative language such as OpenMP or OpenCL. Atkey et al. [2017] describe a compilation process that is guaranteed to be *strategy-preserving*; meaning, the compiler makes no implicit implementation decisions. Instead, the compiler respects the implementation and optimization decisions explicitly encoded in the low-level RISE program.

Lift promises the rewriting process to be fully automatic using a stochastical search method. However, there are many cases where this is either impractical because the rewriting process takes too long, or expert programmers want precise control about applying optimizations to a particular program targeting a particular hardware device. Therefore, we introduce a language that allows a programmer to specify optimization strategies as compositions of rewrite rules.

## 4 ELEVATE: A LANGUAGE FOR DESCRIBING OPTIMIZATION STRATEGIES

In this section, we describe our functional language for describing optimization strategies: ELEVATE. It complements our functional language for describing computations. ELEVATE is heavily inspired by earlier works on strategy languages for term rewriting systems, e.g., Stratego [Visser et al. 1998].

### 4.1 Language Features and Types

ELEVATE is a functional language with a standard feature set, including function recursion, algebraic data types, and pattern matching. Besides the standard scalar data types such as `int`, types of interests are function types and pair types. Our current implementation is a shallow embedded DSL in Scala, and we use Scala-like notation for ELEVATE strategies in the paper.

### 4.2 Strategies

A *strategy* is the fundamental building block of ELEVATE. Strategies encode program transformations and are modeled as functions with the following type:

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

Here, `P` is the type of the rewritten program. `P` could, for example, be `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])
```

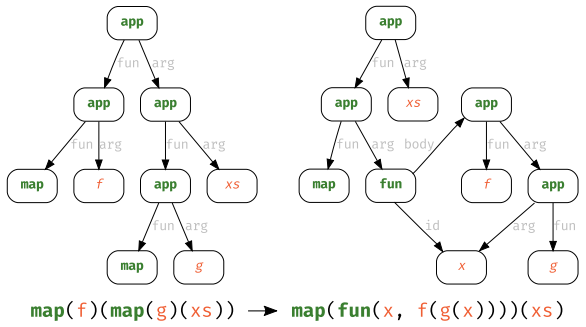


Fig. 5. RISE's *map-fusion* rule as an AST transformation.

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

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

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

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

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

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

Fig. 6. Rewrite rules of high-level RISE expressions used for optimizations in this paper

In case of a successful application, `Success` contains the transformed program, in case of a failure, `Failure` contains the unsuccessful strategy.

The simplest example of a strategy is `id` that always succeeds by returning its input program:

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

The `fail` strategy does the opposite and always fails while recording that it was the failing strategy:

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

### 4.3 Rewrite Rules as Strategies

In ELEVATE, rewrite rules are also strategies, i.e., functions satisfying the same type given above. Let us suppose we want to apply some well-known rewrite rules such as the fusion of two `map` calls: `map(f) << map(g)` $\rightsquigarrow$ `map(f << g)`. In RISE, the left-hand side of the rule is expressed as:

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

Figure 5 (left) shows the AST representation of the body of this expression, with function applications explicit as `app` nodes. 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) }
```

Note that we are mixing RISE (i.e., `map(f)`) and ELEVATE expressions. We 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 rewritten expression shown in Figure 5 on the right.

```
mapSplit : (n: ℕ) → {m: ℕ} → {s t: Set} → (f: s → t) → (xs: Vec s (m * n)) →
           map (map f) (split n {m} xs) ≡ split n {m} (map f xs)
simplification : (n: ℕ) → {m: ℕ} → {t: Set} → (xs: Vec t (m*n)) → (join ∘ split n {m}) xs ≡ xs
{- Split-join rule proof -}
splitJoin : {m: ℕ} → {s: Set} → {t: Set} → (n: ℕ) → (f: s → t) → (xs: Vec s (m * n)) →
            (join ∘ map (map f) ∘ split n {m}) xs ≡ map f xs
splitJoin {m} n f xs =
    begin
      (join ∘ map (map f) ∘ split n {m}) xs
    ≡⟨⟩
      join (map (map f) (split n {m} xs))
    ≡⟨ cong join (mapSplit n {m} f xs) ⟩
      join (split n {m} (map f xs))
    ≡⟨ simplification n {m} (map f xs) ⟩
      map f xs
    ■
```

Listing 3. Proof of correctness of the `splitJoin` rewrite rule in Agda

Figure 6 shows rewrite rules that are used as basic building blocks in this paper for expressing optimizations such as tiling, discussed later in Section 5. One advantage of the functional rewrite approach is that these rules are intuitive and that there is a clear pathway to prove their correctness. We have formally proven correctness in Agda, a dependently typed programming language originally developed by Norell [2007]. We encoded the semantics of the RISE patterns in Agda and expressed the rewrite rules as types following the propositions-as-types interpretation [Wadler 2015]. Providing a well-defined semantics allows precise reasoning about the rewrite rules and their composition as strategies - one of the key aims that we set out for our approach.

Listing 3 shows the proof for the `splitJoin` rewrite rule in Agda. The proof makes use of two lemmas shown at the top. The `mapSplit` lemma says that the **`split`** primitive splitting a one-dimensional array into a two-dimensional one can either be applied before two nested **`map`**s or after a single one. The `simplification` lemma states that **`split`** and its opposite pattern **`join`** cancel each other out. In the future, we want to directly generate implementations from Agda for rewrite rules which integrate with our Scala DSL, guaranteeing that we only use proven rules.

### 4.4 Strategy Combinators

An idea that ELEVATE inherits from Stratego [Visser 2004] is to describe strategies as compositions – one of the key aims that we set out for our approach. Therefore, we introduce strategy combinators.

The **`seq`** combinator is given two strategies `fs` and `ss` and applies the first strategy to the input program `p`. Afterward, 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`**. Additionally, the **`try`** combinator 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)
```

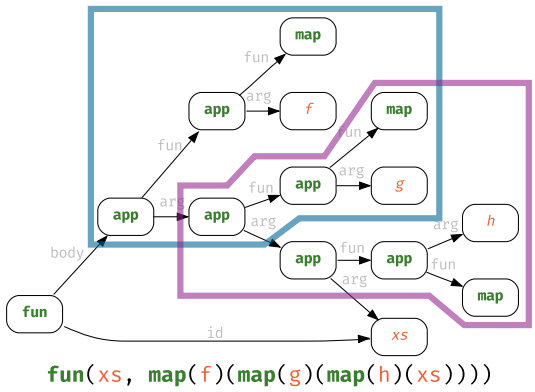


Fig. 7. Two possible locations for applying the *map-fusion* rule within the same program.

### 4.5 Traversals as Strategy Transformers

We implement the `mapFusion` strategy we saw in the previous subsection as a function in ELEVATE. Therefore, its `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.

Also, a strategy is often applicable at multiple 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 threemaps = fun(xs, map(f)(map(g)(map(h)(xs))))
```

We may fuse the first or last two `map`s, as shown in Figure 7.

In ELEVATE, we use *traversals* to describe at which exact location a strategy is applied. Luttik et al. [1997] proposed three basic traversals encoded as strategy transformers:

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

`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. `some` applies a given strategy to at least one sub-expression but potentially more if possible. `one` and `some` are allowed to choose sub-expressions non-deterministically.

In ELEVATE, we see these three 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 7; therefore, `all`, `one`, and `some` correspond to the obvious implementations on the tree-based representation.

To fuse the first two `map`s in Figure 7, we use the `one` traversal: `one(mapFusion)(threemaps)`. This will apply 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 to apply `mapFusion` two levels down in the AST: `one(one(mapFusion))(threemaps)`. This successfully applies the fusion strategy to the expression highlighted in the lower-left purple box in Figure 7.

### 4.6 RISE-Specific Traversal Strategies

The traversals we have discussed so far are not specific to RISE. These traversals are flexible but offer only limited control as for `one` and `some`, the selection of sub-expressions is either non-deterministic or implementation-dependent (as for RISE). Especially 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 $\lambda$-calculus as its representation. Therefore, it makes sense to introduce traversals that navigate the two core concepts of $\lambda$-calculus: `fun`ction abstraction and `app`lication.

To apply a strategy to the body of a function abstraction, we define the following traversal:

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

The `body` traversal applies the strategy `s` 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:

```
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)) }
```

For the RISE program shown in Figure 7, we can now describe a precise traversal path in the AST. To fuse the first two `map`s, we may write `body(mapFusion)(threemaps)`, and to fuse the others, we write `body(argument(mapFusion))(threemaps)`. Both versions describe the precise path from the root to the sub-expression at which the fusion rule is applicable.

### 4.7 Complete Expression Traversal Strategies

All of the traversal primitives introduced so far apply their given strategies only to immediate sub-expressions. Using strategy combinators and traversals, we can define recursive strategies which 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  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)
```

`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. `allTopDown` and `allBottomUp` do not use `lChoice`, insisting on applying the given strategy to every sub-expression. 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. Visser [2004] initially proposed these traversals, and we use them here with slightly different names more fitting for our use case.

### 4.8 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 $\beta$-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 can not be applied anymore. Therefore, after `normalize` successfully finishes, it is not possible to apply the given strategy to any sub-expression any more.

*Beta-Eta-Normal-Form.* $\lambda$-calculus (and RISE) allows for semantically equivalent but syntactically different expressions. For example, `fun(x => f(x))` is equivalent to `f` *iff* `x` does not appear free in `f`. Transforming between these representations is called *$\eta$-reduction* and *$\eta$-abstraction*, which we also implement as ELEVATE strategies:

```
def etaReduction: Strategy[Rise] = p => p match {
  case fun(x1, app(f, x2)) if x1 == x2 && not(contains(x1))(f) => Success(f)
  case _                                                       => Failure(etaReduction)}

def etaAbstraction: Strategy[Rise] = p => p match {
   case f if hasFunctionType(f) => Success( fun(x, f(x)) )
   case _                       => Failure(etaAbstraction) }
```

Note that we can use two ELEVATE strategies, `not` and `contains`, in the pattern guard of the `etaReduction` strategy:

```
def not: Strategy[P] => Strategy[P] = s => p => s(p) match {
  case Success(_) => Failure(not(s))
  case Failure(_) => Success(p) }

def contains[P]: P => Strategy[P] = r => p => topDown(isEqualTo(r))(p)

def isEqualTo[P]: P => Strategy[P] = r => p =>
  if(p == r) Success(p) else Failure(isEqualTo(r))
```

The `RewriteResult` obtained by applying `not(contains(x1))` to `f` is implicitly cast to a `Boolean` eventually. The `contains` strategy traverses `f` from top to bottom and checks if it finds `x1` using the `isEqualTo` strategy.

The simplest normal-form we often use in the following is the *$\beta\eta$-normal-form* (`BENF`) which exhaustively applies $\beta$- and $\eta$-reduction: `def BENF = normalize(betaReduction <+ etaReduction)`. Since not every function abstraction is $\eta$-reducible, the function arguments of RISE's higher-order primitives `map` and `reduce` might have different syntactic forms. Different syntactic forms complicate the development of rewrite rules as they are always defined to match a particular syntactic structure. In order to simplify the application of strategies and the development of new rules, we make heavy use of an additional normal-form, which unifies the syntactic structure of function arguments to higher-order primitives.

*Data-Flow-Normal-Form.* The Data-Flow-Normal-Form (`DFNF`) is an essential normal-form for RISE programs because it ensures a particular syntactic structure that we rely on during rewriting. Specifically, `DFNF` makes the data flow in a RISE program explicit in two ways: First, by ensuring a function abstraction is present in every higher-order primitive, and second, by ensuring every higher-order primitive is fully applied. For example, `DFNF` ensures that a `map` is always provided two arguments: a function and an input array, even if it could be $\beta$-reduced.

```
1  def DFNF = BENF `;` // (1) normalize using beta-eta-normal-form
2    // (2) ensure that the argument of a map is a function abstraction
3    normalize(argOf(map, not(isFun) `;` etaAbstraction))`;`
4    // ... similar normalization for reduce primitive (left out for brevity)
5    // (3) ensure every map is provided two arguments (and every reduce is provided three arguments)
6    normalize(
7      // (3.1) if there is a map in 2 hops (or there is a reduce in three hops) ...
8      one(function(isMap) <+ one(function(isReduce))) `;`
9      // (3.2) ... and the current node is not an apply ...
10     not(isApplication) `;`
11     // (3.3) eta-abstract
12     one((function(isMap) <+ one(function(isReduce))) `;` etaAbstraction)
```

Listing 4. Definition of the Data-Flow-Normal-Form (DFNF)

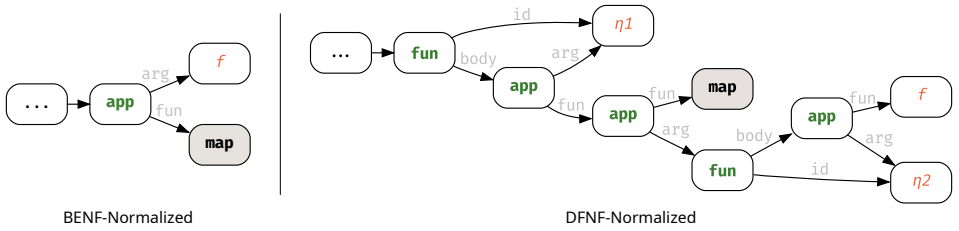


Fig. 8. Two semantically equivalent but syntactically different versions of the RISE expression `map(f)`. The left version is beta-eta normalized using BENF, and the right version is in data-flow-normal-form (DFNF).

Figure 8 shows the result of applying the DFNF to the RISE expression `map(f)`. First, the input expression is normalized using BENF (line 1) which generally decreases the AST size.

Second, we unify the syntactic form of the function arguments of higher-order primitives **map** and **reduce**. Listing 4 shows this unification for the **map** primitive in line 3, the definition for the **reduce** primitive is similar. The `argOf` traversal is similar to `argument` we already introduced; however, it only traverses to the argument of the function application if the applied function matches the given input (**map** in this case). Essentially, whenever the function argument of a **map** primitive is not already a function abstraction (as on the left side of Figure 8), we eta-abstract it. Using the `not` and `isFun` predicates that are themselves strategies, we describe the desired form in a declarative way.

Third, we ensure that higher-order primitives are fully applied. For example, the **map** primitive on the left side of Figure 8 is not applied to an array argument. By applying DFNF, the array argument $\eta 1$ is added by eta-abstracting again, as shown on the right side of the figure. This normalization is also naturally expressed using traversals and predicates: Whenever, we can reach a **map** node in two hops (**one**(`function(isMap)`)), and the current node is not already an **app**-node (`not(isApplication)`), we know that the **map** primitive is not applied to an array argument, so we apply eta-abstraction.

Even though the size of the AST might increase significantly by applying DFNF instead of only BENF, we now have a unified syntactic structure. This structure simplifies the traversal and implementation of more complex optimization strategies, as we will see in the following section.

*Confluence and Termination.* Confluence (multiple non-deterministic rewrite paths eventually produce the same result) and termination are desirable properties for normal-forms in term rewriting systems. In ELEVATE, confluence only becomes a factor when the implementations of **one** and **some** are non-deterministic. For RISE, we are not interested in having multiple non-deterministic rewrite paths but instead need precise control over where, when, and in which order specific rules are applied. Therefore, we avoid non-determinism and do not need to worry about confluence.

Termination of normal-forms, and ELEVATE programs in general, must be evaluated on a case by case basis as it critically depends on the chosen set of strategies. For example, it is trivial to build a non-terminating normal-form using the `id` strategy that is always applicable. We currently do not prevent the creation of non-terminating strategies, similar to how almost all general-purpose computational languages do not prevent writing non-terminating programs. In the future, we are interested in introducing a more powerful type system for ELEVATE to better assist the user in writing well-behaved strategies.

## 5 EXPRESSING HIGH-PERFORMANCE OPTIMIZATIONS AS REWRITE STRATEGIES

In the domain of deep learning, high-performance optimizations are particularly important. While Visser et al. [1998] showed that strategy languages can be used to build program optimizers, the optimizations implemented as strategies were not targeted towards high-performance code generation but rather to optimize a functional ML-like language. To the best of our knowledge, this paper is the first to describe a holistic functional approach for high-performance code generation that implements high-performance optimizations as rewrite strategies and can compete with state-of-the-art imperative solutions.

In this section, we explore using ELEVATE to encode high-performance optimizations by leveraging its ability to define custom abstractions. We use TVM by Chen et al. [2018] as a comparison for a state-of-the-art imperative optimizing deep learning compiler with a scheduling API implemented in Python. TVM allows to express computations using an EDSL (in Python) and control the application for optimizations using a separate scheduling API. We use RISE as the language to express computations and develop separate strategies in ELEVATE, implementing the optimizations equivalent to those available in TVM's scheduling API.

We start by expressing basic scheduling primitives such as `parallel` and `vectorize` 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 ELEVATE Strategies

The TVM scheduling primitives `parallel`, `split`, `vectorize`, and `unroll` specify loop transformations targeting a single loop. We implement those as rewrite rules for RISE. The `parallel` scheduling primitive indicates that a particular loop shall be computed in parallel. 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 rewrite into the sequential variant `mapSeq` in the same style.

TVM's `split` scheduling primitive implements loop-blocking (also known as strip-mining). In RISE, this is achieved by transforming 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 flattened 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 app(app(reduce, op), init) => Success(
    split(n) >> reduce(fun(a, fun(y, op(a, reduce(op)(init)(y)) )))(init) )
  case _ => Failure( split(n) ) }
```

It is important to note that RISE does not materialize the intermediate two-dimensional array in memory. Instead, we only use this representation inside the compiler for code generation. In TVM, the `split` scheduling primitive can also be used to block reduction loops for which we use the `reduce` primitive in RISE. To make the `split` strategy applicable to both `map` and `reduce` primitives, we add a second case to the strategy which blocks a single `reduce` into two nested reductions.

The `vectorize` scheduling primitive indicates that a loop shall be computed in a SIMD-fashion and its equivalent ELEVATE `vectorize` strategy implementation is similar to the `split` strategy:

```
def vectorize(n: Int): Strategy[Rise] = p => p match {
  case app(map, f) if isScalarFun(f) => Success(asVector(n) >> map(mapVec(f)) >> asScalar)
  case _                             => Failure( vectorize(n) ) }
```

First, it splits the input of scalars into an array of vectors using the `asVector` primitive. Then, `f` is mapped twice and transformed to perform vectorized computations using `map(mapVec(f))`, and finally, the resulting array of vectors is transformed into an array of scalars again.

Vectorization is most efficient when applied to the innermost loop of a loop-nest. In RISE, this corresponds to applying the `vectorize` strategy to the innermost `map` of potentially nested `map`s. Applying a strategy to the innermost `map` of nested `map`s is achieved in ELEVATE by traversing the expression beginning from the bottom (for example using `bottomUp(vectorize)`). The additional constraint `isScalarFun(f)` ensures that only functions operating on scalars are vectorized by inspecting `f`'s type. The restriction to scalar functions for `vectorize` is a current limitation of RISE.

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

```
def unroll: Strategy[Rise] = p => p match {
  case map    => Success( mapSeqUnroll )
  case reduce => Success( reduceSeqUnroll )
  case _      => Failure( unroll ) }
```

### 5.2 Multi-dimensional Tiling as an ELEVATE 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. We construct a generalized strategy out of a few simple building blocks that can tile an arbitrary number of dimensions, whereas TVM only implements 2D tiling.

We require five basic rules for expressing our multidimensional tiling strategy: `splitJoin`, `addId`, `idToTranspose`, `transposeMove`, and `mapFission` (all shown in Figure 6). We implement these rules as basic ELEVATE strategies, as shown in the previous sections. In addition, we require three standard $\lambda$-calculus-specific transformations: $\eta$- and $\beta$-reduction, and $\eta$-abstraction.

Our tiling strategy expects a list of tile sizes, one per tiled dimension:

```
def tileND: List[Int] => Strategy[Rise]
```

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

Listing 5. ELEVATE strategy implementing tiling recursively for arbitray dimensions.

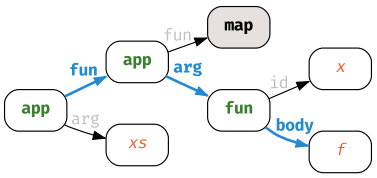


Fig. 9. Visualization of the `fmap` traversal which traverses to the function argument of a RISE-`map` primitive

The two-dimensional tiling, which is equivalent to TVM's built-in `tile` scheduling primitives, is expressed as `tileND(List(x,y))(mm)`. For this 2D case, we also write `tile(x,y)(mm)`.

Listing 5 shows the ELEVATE implementation of the tiling optimization. The intuition for our `tileND` 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 innermost to outermost. Finally, we interchange dimensions accordingly.

We start to explain how we recursively traverse (using `fmap`) to apply loop-blocking and then discuss how we interchange dimensions in RISE (`interchange`).

*Recursively Applying Loop-Blocking.* 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)))
```

Figure 9 visualizes the traversal of the `fmap` strategy; the traversed path is highlighted in blue. `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. This syntactic structure can be assumed 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)))))
```

skips two `map`s and applies loop-blocking to the innermost `map`. In Listing 5 line 3, we use `fmap` to recursively call `tileND` applying loop-blocking first to the inner `map`s before to the outer `map`:

```
fmap(tileND(d-1)(n.tail)) ';'   // apply loop-blocking to inner map
function(split(n.head)) ';' ... // apply loop-blocking to outer map
```

*Loop-Interchange in `tile`.* After recursively blocking all `map`s, we use `interchange` to rearrange the dimensions in the correct order. For simplicity, we describe the two-dimensional case of tiling matrix multiplication. The untiled matrix multiplication implementation contains a loop nest of depth three, corresponding to a `reduce` primitive nested in two `map` primitives in RISE. For brevity, we write this loop-nest as $(M.N.K)$, indicating the dimensions each loop iterates over from outermost to innermost. After applying loop-blocking to the outermost two loops, the loop-nest

has been transformed into a 5-dimensional loop-nest ($M.mTile.N.nTile.K$). To create the desired tiling iteration order ($M.N.mTile.nTile.K$), we need to swap two inner loops. To achieve this, we introduce two **transpose** patterns inside the map nest using the rules shown in Figure 6:

```
val loopInterchange2D = // interchange-strategy used for 2D-tiling
fmap(                          // in: map(map(map(map(dot)))) ...
  addId `;`                    // map(id « map(map(map(dot))))
  idToTranspose `;`            // map(transpose « transpose « map(map(map(dot))))
  DFNF `;`                     // normalize intermediate expression
  argument(transposeMove)) `;` // map(transpose « map(map(map(dot))) « transpose)
normalize(mapFission) // out: map(transpose) « map(map(map(map(dot)))) « map(transpose)
```

Creating the two **transpose** patterns inside the **map** nest swaps the iteration order in the desired way. The general `interchange` case simply adds multiple **transpose** pairs in the required positions.

Using normalization, domain-specific traversals, and five RISE-specific rewrite rules, we were able to define a multidimensional tiling strategy. As every rewrite rule, as well as their compositions, is correct, as shown earlier, we ensure the correctness of the overall optimization.

### 5.3 Reordering as an ELEVATE Strategy

Finally, we briefly discuss the implementation of TVM's **reorder** strategy, which enables arbitrary loop-interchanges. Generally, TVM's **reorder** is a generalization of the loop-interchange optimization we discussed in the previous subsection. Due to the loopless nature of RISE, implementing TVM's **reorder** primitive as a strategy is slightly more involved. Instead of merely interchanging perfectly nested loops, we achieve the same optimization effect in RISE by interchanging the nesting of **map** and **reduce** patterns. Therefore, there are multiple possible combinations to consider.

The most straightforward case are two nested **map**s that correspond to a two-dimensional loop-nest. To interchange the loops created by the two **map**s, we introduce two **transpose** primitives and move one before and one after the **map**-nest, as discussed in the previous subsection.

In addition to interchanging loop-nests created by nested **map**s, we also need to consider interchanging nested **map** and **reduce** primitives. For computations including matrix multiplication, hoisting reduction loops higher up in a loop nest is often beneficial as shown in the following listings:

```
1 for (int i = 0; i < M; i++) {      /* map    */
2   float acc = 0.0f;                /* reduce */
3   for (int j = 0; j < N; j++) {
4     acc += xs[j + (N * i)]; }
5   out[i] = acc;
6 }
```

Listing 6. Reducing the rows of a matrix *xs* with the reduction as the *inner* loop

```
1 float acc[M];                          /* reduce */
2 for (int i = 0; i < M; i++) { acc[i] = 0.0f; }
3 for (int j = 0; j < N; j++) {
4   for (int i = 0; i < M; i++) {    /* map    */
5     acc[i] += xs[j + (N * i)]; }}
6 for (int i = 0; i < M; i++) {out[i] = acc[i];}
```

Listing 7. Reducing the rows of a matrix *xs* with the reduction as the *outer* loop

Transforming the code in Listing 6 into the code in Listing 7 enables different opportunities for parallelizing the reduction. For example, the computation in Listing 7 can now be easily vectorized.

The following rule implements this interchange of nested **map** and **reduce** primitives.

```
map(reduce(+)(0))(xs :: M.N.float)
↪
reduce(fun(acc, fun(y,
  map(fun(x, fst(x) + snd(x)))(zip(acc)(y))))) // reduce-op
 (generate(N)(0))                             // reduce-init
 (transpose(xs))                              // reduce-input
```

By transposing the input and modifying the operator of the `reduce` primitive we can interchange the nesting. Specifically, instead of reducing scalar values as in the input expression, the operator of the `reduce` is transformed to reduce arrays using the inner `map`. Due to the more complex structure of the AST after performing such a transformation, the strategy producing and traversing this tree is similarly complicated and will not be discussed in detail. While it is possible to implement TVM's `reorder` primitive, this particular loop transformation is just not a good fit for the pattern-based abstractions in the RISE language.

### 5.4 Abstractions for Describing Locations in RISE

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

```
k,      = s[C].op.reduce_axis
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: e.g., `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 possible `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. Using `topDown(split)` always blocks the outermost, loop and `tryAll(split)` blocks every loop in the loop nest. Similarly, 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 example for strategy predicates are `isReduce` and `isApp` that check if the current node is a `reduce` primitive or an applied function respectively:

```
def isReduce: Strategy[Rise] = p => p match {
    case reduce => Success(reduce)
    case _      => Failure(isReduce) }

def isApp(funPredicate: Strategy[Rise]): Strategy[Rise] = p => p match {
    case app(f,_) => (_ => p) <$> funPredicate(f)
    case _        => Failure(isApp(s)) }
```

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 and use Scala's implicit classes for this in our implementation.

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 RISE, 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 and visits nested primitives from outermost to innermost, trying to apply the predicate. Only if the predicate is applied successfully, it applies the given strategy `s`. Similarly, we define an `innermost` function which uses **`bottomUp`** instead of **`topDown`**. The `mapNest` predicate recursively traverses inside 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)
```

Identifying locations in the AST could potentially be simplified by tagging sub-expressions, i.e., naming AST sub-graphs. However, then rewrite rules would have to additionally describe how to name the rewritten program. For our current use-cases, the presented location descriptions are sufficient, but we are exploring further ways to simplify the description of specific AST locations.

## 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 primary case study. We compare the performance achieved using code generated by the RISE compiler and code generated by TVM. Afterward, we investigate a different domain and compare against Halide, the state-of-the-art compiler for image processing.

### 6.1 Optimizing Matrix Matrix Multiplication with ELEVATE Strategies

For our case study of matrix-matrix multiplication, we follow a tutorial from the TVM authors that discusses seven differently optimized versions: *baseline, blocking, vectorized, loop permutation, array packing, cache blocks*, and *parallel*. Each version is designed to improve the previous version progressively. For each TVM schedule, we show an equivalent strategy implemented with ELEVATE and evaluate the performance achieved. Using the TVM-like scheduling abstractions implemented as strategies and the traversal utilities, we now discuss how to describe entire schedules in ELEVATE.

*Baseline.* For the *baseline* version, TVM uses a default schedule, whereas ELEVATE describes the implementation decisions explicitly – one of the key aims that we set out for our approach:

```
// 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)
```

Listing 8. RISE matrix multiplication epxression (top) and *baseline* strategy in ELEVATE (bottom)

```
# 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)
```

Listing 9. TVM matrix multiplication algorithm and *baseline* (default) schedule

```
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 10. 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 11. TVM *blocking* schedule

```
1 val loopPerm = (
2  tile(32,32)       ‘@‘ outermost(mapNest(2))    ‘;;‘
3  fissionReduceMap ‘@‘ outermost(appliedReduce) ‘;;‘
4  split(4)          ‘@‘ innermost(appliedReduce) ‘;;‘
5  reorder(Seq(1,2,5,3,6,4))                      ‘;;‘
6  vectorize(32)     ‘@‘ innermost(isApp(isApp(isMap))))
7 (loopPerm ‘;‘ lowerToC)(mm)
```

Listing 12. ELEVATE *loop permutation* strategy

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

Listing 13. TVM *loop permutation* schedule

The TVM algorithm computes the dot product in a single statement in Listing 9 lines 5-7. The RISE program shown at the top of Listing 8 describes the dot product with separate **map** and **reduce** primitives, which are fused as described in the ELEVATE program below using the `fuseReduceMap` rewrite rules from Figure 6. The `lowerToC` strategy rewrites **map** into **mapSeq** and **reduce** into **reduceSeq**. 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. For the following optimized versions, we do not repeat the RISE and TVM programs if they are similar to the previous version.

*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 Listing 10 and 11. First, we `tile`, then we `split`, and then we `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`, we apply `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. It denotes that we apply `DFNF` to normalize intermediate expressions between each step.

*Vectorized and Loop Permutation.* The *loop permutation* version in Listing 12 performs vectorization of the innermost **map** using `vectorize` and applies a different reordering of dimensions as in the blocking version before.

*Array Packing.* So far, ELEVATE's strategies and TVM's schedules were reasonably similar. The *array packing* version is the first to emphasize the flexibility of our holistic functional approach. 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. Here specifically, the *array packing* version in Listing 15 permutes the elements of the B matrix in memory to improve the memory access patterns by introducing an additional computation `pB` in lines 6-7, before using it in the computation in lines 8-10.

For our implementation of the *array packing* version in Listing 14, we are not required to change the RISE program, but define and apply the array packing of matrix *B* simply as another rewrite step in ELEVATE using the `storeInMemory` strategy described below. To complete the entire strategy for

```
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)
```

Listing 14. ELEVATE *array packing* strategy

```
# 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)
```

Listing 15. TVM *array packing* schedule and algo.

this version, we compose the array packing together with `permuteB` that uses **`transpose`** primitives to implement the permutation similarly to `interchange` for the `tile` strategy. Finally, we can simply reuse the prior `loopPerm` strategy to complete this version.

The strategy for storing sub-expressions in memory uses the **`toMem`** primitive of RI**SE** and is defined as follows:

```
def storeInMemory(what: Strategy[Rise], how: Strategy[Rise]): Strategy[Rise] = { p =>
  extract(what)(p) »= (extracted => { how(extracted) »= (storedSubExpr => {
    val idx = Identifier(freshName("x"))
    replaceAll(what, idx)(p) match {
      case Success(replaced) => Success(toMem(storedSubExpr)(fun(idx, replaced)))
      case Failure(_) =>        Failure(storeInMemory(what, how))
  }})}}

  // helper-functions
  def replaceAll(exprPredicate: Strategy[Rise], withExpr: Rise): Strategy[Rise] =
    p => tryAll(exprPredicate `;` insert(withExpr))(p)
  def insert(expr: Rise): Strategy[Rise] = p => Success(expr)
  // find and return Rise expr which matches the exprPredicate
  def extract(exprPredicate: Strategy[Rise]): Strategy[Rise] = ...
}

def inLambda(s: Strategy[Rise]): Strategy[Rise] =
  isLambda `;` ((p:Rise) => body(inLambda(s))(p)) <+ s
```

The `storeInMemory` strategy expects two arguments: `what` - a strategy predicate describing the sub-expression to store and `how` - the strategy that specifies how to perform the copy. In the `arrayPacking` strategy, we want to store a permuted version of the transposed B (described by the `isTransposedB` predicate) to memory. Since every RI**SE** sub-expression can be stored in memory at any time, the `storeInMemory` strategy only fails if the desired sub-expression (described by `what`) cannot be found or cannot be stored as specified in `how`.

```
val par = (
  arrayPacking `;;`
  (parallel `@` outermost(isMap))
            `@` outermost(isToMem) `;;`
   unroll   `@` innermost(isReduce))

(par `;` lowerToC)(mm)
```

Listing 16. ELEVATE *parallel* strategy

```
s = tvm.create_schedule(C.op)
CC = s.cache_write(C, 'global')
xo, yo, xi, yi = s[C].tile(
  C.op.axis[0], C.op.axis[1], bn, bn)
s[CC].compute_at(s[C], yo)
xc, yc = s[CC].op.axis
k, = s[CC].op.reduce_axis
ko, ki = s[CC].split(k, factor=4)
s[CC].reorder(ko, xc, ki, yc)
s[CC].unroll(ki)
s[CC].vectorize(yc)
s[C].parallel(xo)
x, y, z = s[pB].op.axis
s[pB].vectorize(z)
s[pB].parallel(x)
```

Listing 17. TVM *parallel* schedule

The `toMem` primitive is introduced in two steps: First, the sub-expression needs to be removed from the original expression (`p`) and be replaced with the new identifier `x`. Second, the value for the new identifier `x` needs to be extracted from the original expression `p`.

*Cache Blocks and Parallel.* The TVM version in Listing 17 changes the algorithm yet again (not shown for brevity) to introduce a temporary buffer (`cc`) for the accumulation along the K-dimension to improve the cache writing behavior and unrolls the inner reduction loop. The RISE code generator makes accumulators for reductions always explicit. Therefore, we reuse the *array packing* version adding strategies for unrolling the innermost reduction and parallelizing the outermost loop in the body of the `toMem` primitive.

## 6.2 Scalability and Overhead of Rewriting

We have demonstrated that it is feasible to implement a TVM-like scheduling language by expressing schedules as compositions of reusable strategies. Using our holistic functional approach, we express the computation only once in RISE and express all optimizations as ELEVATE strategies.

In this section, we are interested in the scalability and overhead of our functional rewrite-based approach using matrix-multiply as a realistic case study of high-performance code generation.

We are counting the number of successfully applied rewrite steps performed when applying a strategy to the RISE matrix multiply expression. We count every intermediate step, which includes traversals as these are implemented as rewrite steps too. For example, `id(fun(x,x))` counts as one rewrite step whereas `(body(id))(fun(x,x))` counts as two steps because we also count the traversal into the function body as an intermediate step.

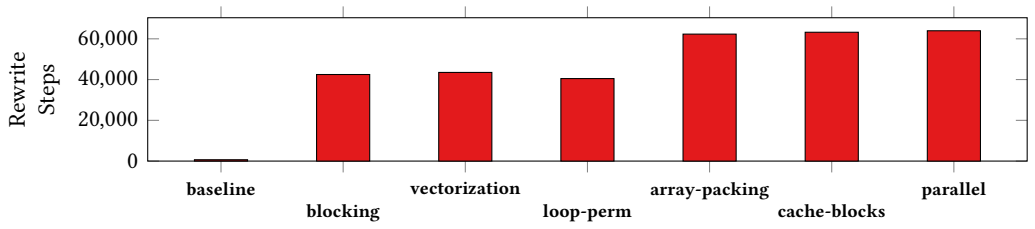


Fig. 10. Total number of successful rewrite steps when applying different optimization strategies.

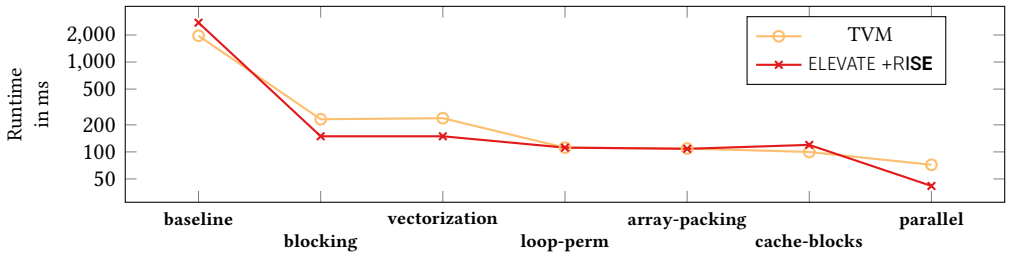


Fig. 11. Performance of TVM vs. RISE generated code that has been optimized by ELEVATE strategies.

Figure 10 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. Applying the strategies to the RISE expression took less than two seconds per version on a commodity notebook with our unoptimized implementation.

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. At the same time, developing and debugging such sophisticated strategies using ELEVATE is still possible as there is a clear pathway towards developing debugging tools for recording traces or rewrites or reporting which strategy was not applicable. Additionally, due to the compositional nature of our strategy approach, one can easily inspect intermediate RISE expressions. In contrast, debugging TVM or Halide schedules is much harder as the scheduling primitives are provided as black-box function calls operating on the internal intermediate program representation.

### 6.3 Performance Comparison against TVM

In this section, we are interested in the performance achieved when optimizing RISE programs with ELEVATE compared to TVM. Ideally, the RISE code optimized with ELEVATE should be similar to the TVM-generated code and achieve competitive performance. We generated LLVM code with TVM (version 0.6.dev) and C code for RISE annotated with OpenMP pragmas for the versions that include parallelization or vectorization. The RISE generated C code was compiled with clang (v.9.0.0) using `-Ofast -ffast-math -fopenmp`, which echoes the settings used by TVM and Halide[1]. We measured on an Intel core i5-4670K CPU (frequency at 3.4GHz) running Arch Linux (kernel 5.3.11-arch1-1). We measured wall-time for RISE-generated code and used TVM's built-in measurement API. We measured 100 iterations per version, reporting the median runtimes in milliseconds.

Figure 11 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 to 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.

[1] https://github.com/halide/Halide/issues/2905

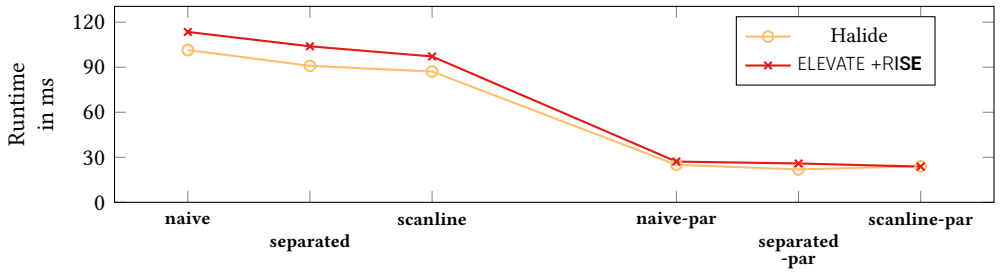


Fig. 12. Performance evaluation of Halide and RISE generated code for the binomial filter application. Optimization decisions for RISE are implemented as ELEVATE strategies.

### 6.4 Performance Comparison against Halide

Since the scheduling languages of Halide and TVM are very similar, we additionally performed a third experiment comparing against the Halide compiler that is specialized in high-performance code generation for image processing. Specifically, we implemented a binomial image processing filter in RISE, corresponding to the example described in [Ragan-Kelley et al. 2013]. As with the comparison against TVM, we used ELEVATE to describe three different optimization strategies (each with a sequential and a parallel version), which are equivalent to three different schedule programs using Halide. Optimizing the binomial filter application also requires to modify the algorithm in Halide, similar to the array packing example in TVM. In contrast, we were again able to express all optimizations using ELEVATE only.

Figure 12 shows the performance of the Halide and RISE generated code measured on an ARM Cortex A7 quad-core[2]. We can see – not surprisingly – that the non-parallel versions on the left are significantly slower than the parallel versions. The Halide generated code is 10-15% faster than the RISE generated code. Improvements to the RISE code generator might close this gap in the future. Crucially, we again observe the same trend for performance improvements across versions demonstrating that our extensible and rewrite based approach is capable of achieving competitive performance to state-of-the-art compilers used in production.

## 7 RELATED WORK

*High-Performance Code Generation.* Halide by Ragan-Kelley et al. [2018] introduced the concept of schedules describing program optimizations separate from the algorithm describing the computation. Many other frameworks have adopted this concept in domains including machine learning (TVM [Chen et al. 2018]), graph applications (GraphIt [Zhang et al. 2018]), and polyhedral compilation (Tiramisu [Baghdadi et al. 2019], CHiLL [Chen et al. 2008; Hall et al. 2009], AlphaZ [Yuki et al. 2012], URUK [Girbal et al. 2006]).

These existing scheduling APIs are not designed as programming languages. They are often not expressive enough to cover all optimizations of interest and instead provide a fixed set of ad-hoc built-in primitives. In this work, we showed how to use ELEVATE to implement scheduling languages from first principles as compositions of rewrite rules targeting the RISE language describing computations.

There are many functional languages aimed towards high-performance code generation including Futhark Henriksen et al. [2017], Accelerate [McDonell et al. 2013], Obsidian [Svensson et al. 2008], and NOVA [Collins et al. 2014]. These projects either rely on hard-coded hand-tuned primitives

[2]The 4 LITTLE cores of a Samsung Exynos 5 Octa 5422

or fixed optimizations during compilation using code analysis and cost models. In contrast, our functional approach allows extensible optimizations using rewrite rules and low-level patterns.

*Rewriting in Compilers.* Rewrite rules and rewriting strategies have also been used to build compilers. The Glasgow Haskell Compiler [Peyton Jones et al. 2001] uses rewrite rules as a practical way to optimize Haskell programs. [Visser et al. 1998] describe how to build program optimizers using rewriting strategies however, on a significantly smaller scale not focusing on high-performance code generation. Other areas include building interpreters [Dolstra and Visser 2002], instruction selection [Bravenboer and Visser 2002] or constant propagation [Olmos and Visser 2002]. Lift [Hagedorn et al. 2018; Steuwer et al. 2015, 2017] showed how to use rewrite rules to generate high-performance code targeting accelerators. Google recently introduced MLIR [Lattner et al. 2020] with declarative rewrite rules to specify transformations for dialects.

Controlling the application of rewrite rules in compilers still largely relies on fixed built-in heuristics. In this work, we showed how to use ELEVATE instead, allowing a more flexible and practical approach towards using rewrite rules for describing optimizations in high-performance compilers.

*Term Rewriting and Strategy Languages.* ELEVATE is inspired by existing strategy languages, especially ELAN [Borovanský et al. 1998, 1996] and Stratego [Visser 2004; Visser et al. 1998], which introduce combinators to support user-defined strategies in the context of term rewriting systems. Similar rewriting systems include Maude [Clavel et al. 2002], PORGY [Andrei et al. 2011; Fernández et al. 2011; Pinaud et al. 2017], ASF+SDF [van den Brand et al. 2001], OBJ3 [Goguen et al. 1987] and TAMPR [Boyle et al. 1997]. Program transformations using rewrite rules and strategy languages have since been used in different domains including reverse engineering [Chikofsky and II 1990], refactoring [Fowler 1999], and obfuscation [Collberg et al. 1998]. Visser [Visser 2001b, 2005] and Kirchner [Kirchner 2015] provide surveys covering term rewriting, strategy languages and their application domains. Tactic languages including [Delahaye 2000] and [Felty 1993] are related to strategy languages but are designed for specifying proofs in theorem provers.

## 8 CONCLUSION

In this paper, we presented a holistic functional approach for high-performance code generation. We presented two functional languages: RI**SE** 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 correctness proofs, 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 to the state-of-the-art code generators Halide and TVM.

## ACKNOWLEDGMENTS

We thank the entire RI**SE** (rise-lang.org) and ELEVATE (elevate-lang.org) teams for their development efforts. We thank our reviewers and our shepherd Stefan Muller for their valuable feedback. The first author was financially supported by an NVIDIA Research Fellowship.

## REFERENCES


Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. https://www.tensorflow.org/ Software available from tensorflow.org.

Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, Olivier Namet, and Bruno Pinaud. 2011. PORGY: Strategy-Driven Interactive Transformation of Graphs. In *Proceedings 6th International Workshop on Computing with Terms and Graphs, TERMGRAPH 2011, Saarbrücken, Germany, 2nd April 2011*. 54–68. https://doi.org/10.4204/EPTCS.48.7

Robert Atkey, Michel Steuwer, Sam Lindley, and Christophe Dubach. 2017. Strategy Preserving Compilation for Parallel Functional Code. *CoRR* abs/1710.08332 (2017).

Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman P. Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*. 193–205. https://doi.org/10.1109/CGO.2019.8661197

Paul Barham and Michael Isard. 2019. Machine Learning Systems are Stuck in a Rut. In *HotOS*. ACM, 177–183.

Richard Bird and Oege de Moor. 1997. *Algebra of Programming*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Christophe Ringeissen. 1998. An overview of ELAN. *Electr. Notes Theor. Comput. Sci.* 15 (1998), 55–70. https://doi.org/10.1016/S1571-0661(05)82552-6

Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. 1996. ELAN: A logical framework based on computational systems. *Electr. Notes Theor. Comput. Sci.* 4 (1996), 35–50. https://doi.org/10.1016/S1571-0661(04)00032-5

James M Boyle, Terence J Harmer, and Victor L Winter. 1997. The TAMPR program transformation system: Simplifying the development of numerical software. In *Modern software tools for scientific computing*. Springer, 353–372.

Martin Bravenboer and Eelco Visser. 2002. Rewriting Strategies for Instruction Selection. In *Rewriting Techniques and Applications, 13th International Conference, RTA 2002, Copenhagen, Denmark, July 22-24, 2002, Proceedings*. 237–251. https://doi.org/10.1007/3-540-45610-4_17

Manuel M. T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. 2011. Accelerating Haskell array codes with multicore GPUs. In *DAMP*. ACM, 3–14.

Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A framework for composing high-level loop transformations*. Technical Report. Citeseer.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. 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*. 578–594. https://www.usenix.org/conference/osdi18/presentation/chen

Elliot J. Chikofsky and James H. Cross II. 1990. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software* 7, 1 (1990), 13–17. https://doi.org/10.1109/52.43044

Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Jose F. Quesada. 2002. Maude: specification and programming in rewriting logic. *Theor. Comput. Sci.* 285, 2 (2002), 187–243. https://doi.org/10.1016/S0304-3975(01)00359-0

Christian S. Collberg, Clark D. Thomborson, and Douglas Low. 1998. Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs. In *POPL '98, Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Diego, CA, USA, January 19-21, 1998*. 184–196. https://doi.org/10.1145/268946.268962

Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. 2014. NOVA: A Functional Language for Data Parallelism. In *ARRAY@PLDI*. ACM, 8–13.

David Delahaye. 2000. A Tactic Language for the System Coq. In *LPAR (Lecture Notes in Computer Science)*, Vol. 1955. Springer, 85–95.

Eelco Dolstra and Eelco Visser. 2002. Building Interpreters with Rewriting Strategies. *Electr. Notes Theor. Comput. Sci.* 65, 3 (2002), 57–76. https://doi.org/10.1016/S1571-0661(04)80427-4

Amy P. Felty. 1993. Implementing Tactics and Tacticals in a Higher-Order Logic Programming Language. *J. Autom. Reasoning* 11, 1 (1993), 41–81.

Maribel Fernández, Hélène Kirchner, and Olivier Namet. 2011. A Strategy Language for Graph Rewriting. In *Logic-Based Program Synthesis and Transformation - 21st International Symposium, LOPSTR 2011, Odense, Denmark, July 18-20, 2011. Revised Selected Papers*. 173–188. https://doi.org/10.1007/978-3-642-32211-2_12

Martin Fowler. 1999. *Refactoring - Improving the Design of Existing Code.* Addison-Wesley. http://martinfowler.com/books/refactoring.html

Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. 2006. Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies. *International Journal of Parallel Programming* 34, 3 (2006), 261–317. https://doi.org/10.1007/s10766-006-0012-3

Joseph A. Goguen, Claude Kirchner, Hélène Kirchner, Aristide Mégrelis, José Meseguer, and Timothy C. Winkler. 1987. An Introduction to OBJ 3. In *Conditional Term Rewriting Systems, 1st International Workshop, Orsay, France, July 8-10, 1987, Proceedings*. 258–263. https://doi.org/10.1007/3-540-19242-5_22

Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. 2018. 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*. 100–112. https://doi.org/10.1145/3168824

Halide. 2020. Tutorial: Scheduling. https://halide-lang.org/tutorials/tutorial_lesson_05_scheduling_1.html

Mary Hall, Jacqueline Chame, Chun Chen, Jaewook Shin, Gabe Rudy, and Malik Murtaza Khan. 2009. Loop transformation recipes for code generation and auto-tuning. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 50–64.

John L. Hennessy and David A. Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.

Troels Henriksen, Niels G. W. Serup, Martin Elsman, Fritz Henglein, and Cosmin E. Oancea. 2017. Futhark: purely functional GPU-programming with nested parallelism and in-place array updates. In *PLDI*. ACM, 556–571.

Hélène Kirchner. 2015. Rewriting Strategies and Strategic Rewrite Programs. In *Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday*. 380–403. https://doi.org/10.1007/978-3-319-23165-5_18

Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2020. MLIR: A Compiler Infrastructure for the End of Moore's Law. arXiv:cs.PL/2002.11054

Sebastiaan Pascal Luttik, Eelco Visser, et al. 1997. *Specification of rewriting strategies.* Universiteit van Amsterdam. Programming Research Group.

Trevor L. McDonell, Manuel M. T. Chakravarty, Gabriele Keller, and Ben Lippmeier. 2013. Optimising purely functional GPU programs. In *ICFP*. ACM, 49–60.

Ulf Norell. 2007. *Towards a practical programming language based on dependent type theory.* Ph.D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden.

Karina Olmos and Eelco Visser. 2002. Strategies for Source-to-Source Constant Progagation. *Electr. Notes Theor. Comput. Sci.* 70, 6 (2002), 156–175. https://doi.org/10.1016/S1571-0661(04)80605-4

Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

Simon Peyton Jones, Andrew Tolmach, and Tony Hoare. 2001. Playing by the rules: rewriting as a practical optimisation technique in GHC. In *2001 Haskell Workshop* (2001 haskell workshop ed.). ACM SIGPLAN.

Bruno Pinaud, Oana Andrei, Maribel Fernández, Hélène Kirchner, Guy Melançon, and Jason Vallet. 2017. PORGY : a Visual Analytics Platform for System Modelling and Analysis Based on Graph Rewriting. In *17ème Journées Francophones Extraction et Gestion des Connaissances, EGC 2017, 24-27 Janvier 2017, Grenoble, France*. 473–476. http://editions-rnti.fr/?inprocid=1002327

Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman P. Amarasinghe, and Frédo Durand. 2018. Halide: decoupling algorithms from schedules for high-performance image processing. *Commun. ACM* 61, 1 (2018), 106–115. https://doi.org/10.1145/3150211

Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman P. Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *PLDI*. ACM, 519–530.

Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. 2015. Generating performance portable code using rewrite rules: from high-level functional expressions to high-performance OpenCL code. In *ICFP*. ACM, 205–217.

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2016. Matrix multiplication beyond auto-tuning: rewrite-based GPU code generation. In *CASES*. ACM, 15:1–15:10.

Michel Steuwer, Toomas Remmelg, and Christophe Dubach. 2017. 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*. 74–85. http://dl.acm.org/citation.cfm?id=3049841

Joel Svensson, Mary Sheeran, and Koen Claessen. 2008. Obsidian: A Domain Specific Embedded Language for Parallel Programming of Graphics Processors. In *IFL (Lecture Notes in Computer Science)*, Vol. 5836. Springer, 156–173.

TVM. 2020. How to optimize GEMM on CPU. https://docs.tvm.ai/tutorials/optimize/opt_gemm.html

Mark van den Brand, Arie van Deursen, Jan Heering, H. A. de Jong, Merijn de Jonge, Tobias Kuipers, Paul Klint, Leon Moonen, Pieter A. Olivier, Jeroen Scheerder, Jurgen J. Vinju, Eelco Visser, and Joost Visser. 2001. The ASF+SDF Meta-environment: A Component-Based Language Development Environment. In *Compiler Construction, 10th International Conference, CC 2001 Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2001 Genova, Italy, April 2-6, 2001, Proceedings*. 365–370. https://doi.org/10.1007/3-540-45306-7_26

Eelco Visser. 2001a. 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*. 357–362. https://doi.org/10.1007/3-540-45127-7_27

Eelco Visser. 2001b. A Survey of Strategies in Program Transformation Systems. *Electr. Notes Theor. Comput. Sci.* 57 (2001), 109–143. https://doi.org/10.1016/S1571-0661(04)00270-1

Eelco Visser. 2004. Program transformation with Stratego/XT. In *Domain-specific program generation*. Springer, 216–238.

Eelco Visser. 2005. A survey of strategies in rule-based program transformation systems. *J. Symb. Comput.* 40, 1 (2005), 831–873. https://doi.org/10.1016/j.jsc.2004.12.011

Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. 1998. 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*. 13–26. https://doi.org/10.1145/289423.289425

Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015), 75–84.

Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay V. Rajopadhye. 2012. AlphaZ: A System for Design Space Exploration in the Polyhedral Model. In *Languages and Compilers for Parallel Computing, 25th International Workshop, LCPC 2012, Tokyo, Japan, September 11-13, 2012, Revised Selected Papers*. 17–31. https://doi.org/10.1007/978-3-642-37658-0_2

Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman P. Amarasinghe. 2018. GraphIt: a high-performance graph DSL. *PACMPL* 2, OOPSLA (2018), 121:1–121:30. https://doi.org/10.1145/3276491