Declarative Transformations in the Polyhedral Model

Oleksandr Zinenko, Lorenzo Chelini, Tobias Grosser

Declarative Transformations in the Polyhedral Model

Oleksandr Zinenko, Lorenzo Chelini, Tobias Grosser

RESEARCH REPORT
N° 9243
September 2018
Project-Team Parkas

ISSN 0249-6399   ISRN INRIA/RR--9243--FR+ENG

Declarative Transformations in the Polyhedral Model

Oleksandr Zinenko[*], Lorenzo Chelini[†], Tobias Grosser[‡]

Project-Team Parkas

Research Report n° 9243 — September 2018 — 30 pages

Abstract: Despite the availability of sophisticated automatic optimizers, performance-critical code sections are in practice still tuned by human experts. Pragma-based languages such as OpenMP or OpenACC are the standard interface to apply such transformations to large code bases and loop-transformation pragmas would be a straightforward extension to provide fine-grained control over a compilers loop optimizer. However, the manual optimization of programs via explicit sequences of directives is unlikely to fully solve this problem as expressing complex optimization sequences explicitly results in difficult to read and non-performance-portable code.
We address this problem by presenting a novel framework of composable program transformations based on the internal tree-like program representation of a polyhedral compiler. Based on a set of tree matchers and transformers, we describe an embedded transformation language which provides the foundation for the development of program optimization tactics. Using this language, we express core building blocks such as loop tiling, fusion, or data-layout-transformations, and compose them to higher-level transformations expressing algorithm-specific optimization strategies for stencils, dense linear-algebra, etc. We expect our approach to simplify the development of polyhedral optimizers and integration of polyhedral and syntactic approaches.

Key-words: polyhedral model, loop nest optimization, tree rewriting

[*] Inria Paris and DI, École Normale Supérieure
[†] TU Eindhoven
[‡] ETH Zürich

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Transformations Declaratives dans le Modèle Polyédrique

Résumé : Malgré l'existence d'outils sophistiqués d'optimisation automatique, les parties des programmes dont la performance est cruciale sont toujours optimisées manuellement par des humains experts. Les langages basés sur des directives "pragma", tels que OpenMP ou OpenACC, sont une interface typique pour exprimer les transformations sur des grandes bases de code source. Telles directives pour transformer des nids de boucles seraient une extension naturelle permettant de contrôler l'optimiseur de boucles d'une manière précise. Pourtant l'optimisation manuelle des programmes à travers les séquences des directives de transformation n'est pas toujours souhaitable car ces séquences longues et complexes produisent des programmes peu lisibles et ne beneficiant pas de la portabilité de performance entre les différentes architectures matérielles.

Nous proposons une nouvelle approche pour définir les transformations composables des programmes basée sur la représentation interne d'un compilateur polyédrique sous forme de l'arbre. Grâce à un ensemble des "motifs" et "transformateurs" des arbres, nous décrivons un langage de transformation sur lequel nous basons le développement des tactiques d'optimisation. Avec ce langage, il est possible d'exprimer les transformations basiques, telles que le tuilage, la fusion ou la transposition de données, ainsi que la composition de ces transformations afin de définir une stratégie d'optimisation pour les grandes classes des programmes, telles que les pochoirs, les contractions de tenseurs, etc. Notre approche pourrait simplifier le développement des optimiseurs polyédriques et l'integration des transformations polyédriques et syntaxiques.

Mots-clés : modèle polyédrique, transformations des nids de boucles, réécriture des arbres

1 Introduction

Software-based solutions address an increasingly large number of significantly complex real-world problems. With combined effect of increased expressivity of high-level programming languages, increased domain-specificity, and diminishing returns of hardware scaling, the quality of optimizing compilers becomes important more than ever to keep the cost of software abstractions manageable. Compilers now have to address multiple, often hardly compatible, goals from code size to energy efficiency, from reducing program runtime to reducing the compilation turnaround time so as to improve the programmer's user experience.

Classical optimizing compiler typically makes use of several internal representations of the program it considers at different stages of the compilation process. Optimization passes are usually based on some form of a graph: abstract syntax trees (often with types and other attributes), control-flow or dependency graphs, etc. These passes often consist in performing tree transformations or graph reductions using a set of pre-defined or user-supplied rules.

Polyhedral compilation, on the other hand, transforms (parts of) the program into a different representation based on integer sets, better amenable to mathematical optimization [14]. While this representation enables one to perform a complex combination of loop optimizations at once directly on the mathematical representation, it is rarely used in complete isolation from the graph-based syntactic representations since only specific constrained parts of the input program can be modeled. In practice, polyhedral optimization techniques are not only used together with conventional program transformation approaches, but also rely on classical graph and tree-based algorithms internally [48]. For example, Pluto loop scheduling algorithm operates on a dependence *graph* whose edges are annotated with integer sets denoting iteration-wise dependences obtained by the polyhedral analysis [9].

The *schedule tree* structure [40] used by the popular *isl* library [36] to represent loop schedules is the glaring example of the practical intertwining of polyhedral and syntax-based techniques. It combines integer set-based or "polyhedral" iteration sets and schedules with the explicit loop nesting structure and coarser-grain relations between dynamic executions of program statements. It also includes ways to precisely control code generation, which are usually not considered by the polyhedral transformations themselves [19]. This code-orientation of schedule tree representation has been leveraged to implement advanced transformations and extensions to the polyhedral model that would have require a separate, syntactic description otherwise.

Practical applications of polyhedral compilation techniques often perform only a *part* of the transformation in the polyhedral representation itself. Loop tiling is often performed as a separate step, controlled by an imperative algorithm that uses the results of polyhedral analyses. Loop fusion is often done outside the main schedule model. Device mapping, parallelization, vectorization or loop unrolling require transformations that cannot be expressed purely in terms of polyhedral schedules and are delegated to subsequent algorithmic steps.

We propose to combine polyhedral analyses and program transformations with more conventional compiler construction techniques based on tree rewriting. We introduce the pattern-matching framework for the schedule tree representation and the integer relations that are at the core of the polyhedral representation. We leverage this framework to build a declarative polyhedral program transformation engine and demonstrate how it can be used to easily express existing and new polyhedral-based optimizations through case studies. Unlike existing script-based polyhedral transformers, our search-and-match approach is not specific to a particular program but allows one to build reusable program transformation tactics in a declarative way, similarly to production optimizing compilers.

2 Background: Polyhedral Compilation

The *polyhedral model* is a unified framework to model and transform loops in imperative programs [15]. After over three decades of active research and attempts to integrate into production compilers [7, 31, 21], it has recently gathered new attention as a method of choice for generating efficient code targeting both CPUs and accelerators from domain-specific languages [34, 27].

The key power of the polyhedral framework is also a major obstacle to its wider adoption: instead of operating on conventional syntax- or instruction-based intermediate representations, it transforms parts of the program into a linear-algebraic form amenable to mathematical optimization. Specific algorithms and tools have been developed to convert high-level languages [6, 39] and intermediate complier representations [21] as well as algorithms to generate the transformed code back [4, 20, 10] have been developed to make the model applicable in practice.

The internal operation of a polyhedral optimizer is widely regarded as a black-box although tools exist to describe the performed transformations in terms of loop transformation primitives [2].

2.1 Iteration Domains and Access Relations

The algebraic representation is based on integer sets and relations. Generally, the model is applicable to static control parts (SCoPs) that consist of loops whose boundaries are affine functions of the surrounding loop iterators and parameters (values of which are unknown at compile time but guaranteed to be constant). Every iteration of these loops is identified by an integer vector in a k-dimensional space, where k is the number of nested loops, with coordinates corresponding to the values of loop induction variables. Each statement other than control flow construct has an associated symbolic name and a set integer vectors describing at which iterations it is executed, commonly referred to as *iteration domain.* When the static control conditions are respected, the iteration domain can be concisely denoted by a system of affine inequalities. For example, the iteration domain of the `gemm` kernel in Listing 1 can be expressed as $(N) \to \{S_1(i,j) \mid 0 \le i,j < N; S_2(i,j,k) \mid 0 \le i,j,k < N\}$ in the tagged-tuple notation introduced by `iscc` [37]. Note that N is a *parameter* in this expression. Individual executions of statements inside loops are called *statement instances*.

```
for (int i = 0; i < N; ++i)
  for (int j = 0; j < N; ++j) {
S1: D[i][j] = beta * C[i][j];
    for (int k = 0; k < N; ++k)
S2:   D[i][j] += alpha * A[i][k] * B[k][j];
  }
```

Listing 1: Generalized Matrix Multiplication (`gemm`) kernel.

Internal computation is usually expressed as arithmetic expressions and function calls with array elements as arguments. The conditions on the array subscripts are essentially the same as those on control flow conditions. Array accesses can be thus precisely encoded as relations between vectors in the iteration spaces and vectors in the array space whose coordinates are values of the accessed subscripts. These relations are most often defined by piece-wise quasi-affine functions. In our running example from Listing 1, the access relation for `S1` is $(N) \to \{S_1(i,j) \to \mathrm{beta}() \mid 0 \le i,j < N; S_1(i,j) \to \mathrm{D}(d_1,d_2) \mid d_1 = i \wedge d_2 = j; S_1(i,j) \to \mathrm{C}(c_1,c_2) \mid c_1 = i \wedge c_2 = j\}$. Polyhedral access relations allow for fine-grain exact data-flow analysis [13] and enable loop transformations [5, 17].

2.2 Schedules and Schedule Trees

The iteration domain defines which statement instances should be executed but not in which order. The latter is defined by a *schedule* that maps points in the iteration space to points in the time space. Statement instances are executed in the lexicographical order of their coordinates in the time space.

While it is possible to express schedules as relations or piece-wise quasi-affine functions, it is often undesirable. This is because statements are likely to share part of their schedule due to the inherent nested strcuture of the generated code. It is also necessary to reflect the relative syntactic order of loops and statements within them, which is often achieved through auxiliary dimensions whose values are always constant for the given statement.[1]

To address these challenges, Verdoolaege *et.al.* proposed the *schedule tree* structure as a way to represent schedules in the polyhedral model [40]. In this structure, statements that share a common partial schedule share an ancestor that describes this partial schedule, removing duplication. The loop nesting maps naturally to parent/child relationships whereas syntactic ordering maps to the left-to-right order of sibling nodes. Beyond simple schedules, schedule trees have been successfully used to support multi-target code generation with

[1]For example, all instances of $S_1(i) \to (0,i)$ are scheduled before any instance of $S_2(i) \to (1,i)$.

offloading mechanisms [19, 38] and advanced hybrid polyhedral/syntactic transformation techniques [18, 45].

Nodes in schedule trees have one of the numerous types. Let us briefly present node types relevant to our examples.

- *domain*—the iteration domain, always located at the root of the tree;
- *band*—partial schedule of one or multiple loops (the name refers to the notion of *tilable band* of loops [8]);
- *filter*—restricts the subtree to a subset of the iteration domain;
- *sequence*—imposes the order between its children (statements or loops);
- *extension*—introduces new statement instances local to the subtree with respect to its prefix schedule.

All nodes except *sequence* have no more than one child. For the vectors in the iteration domain, one can reconstruct the relational form of the schedule by taking a range product of all partial schedules in the tree.

Listing 2 presents a simple 2D stencil program. The corresponding schedule tree is depicted in Figure 1. It uses two band nodes: the outer one provides a partial schedule representing the individual time steps. The inner one provides a partial schedule enumerating all data-points within a given time step. Finally, the individual statement types are enumerated within a sequence node. When combined across all dimensions, the schedule tree defines an execution order identical to the original execution order.

The tree structure of a schedule tree allows for the easy application of local transformations. Such transformations include the combination of nested band nodes into a single band, the splitting of a multi-dimensional band node into individual bands, but also more complex transformations such as tiling of nodes or even the application of affine transformations.

```
  for (int t = 0; t <= T; t++)
    for (int i = 1; i < 1023; i++)
      for (int j = 1; j < 1023; j++)
        if (t % 2 == 0)
S:         A[i][j] += B[i-1][  j] + B[  i][j+1]
                    + B[  0][  0]
                    + B[  i][j-1] + B[i+1][  j];
        else
T:         B[i][j] += A[i-1][  j] + A[  i][j+1]
                    + A[  0][  0]
                    + A[  i][j-1] + A[i+1][  j];
```

Listing 2: A simple 2D stencil kernel

This schedule can be rewritten in the relational form as $(T) \to \{S(t,i,j) \to (t,i,j,0); T(t,i,j) \to (t,i,j,1)\}$. Note that duplication of (t,i,j) and the presence of auxiliary trailing dimensions.

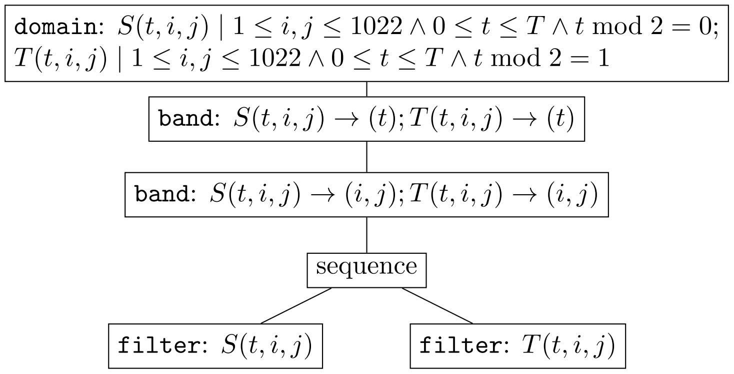
domain: $S(t,i,j) \mid 1 \le i,j \le 1022 \wedge 0 \le t \le T \wedge t \bmod 2 = 0;$
$T(t,i,j) \mid 1 \le i,j \le 1022 \wedge 0 \le t \le T \wedge t \bmod 2 = 1$
band: $S(t,i,j) \to (t); T(t,i,j) \to (t)$
band: $S(t,i,j) \to (i,j); T(t,i,j) \to (i,j)$
sequence
filter: $S(t,i,j)$
filter: $T(t,i,j)$

Figure 1: Schedule tree representation of the code in Listing 2.

3 Declarative Schedule Tree Transformations

Unlike conventional compilation approaches centered around syntax trees, polyhedral compilation techniques typically operate on some representation of a schedule disconnected from any syntactic representation. While schedule representations are often based on algebraic objects such as sets or relations in vector spaces offering large sets of operations and consistency guarantees, they do not easily lend themselves to classical tree rewriting compiler passes. Thanks to the nature of the schedule tree representation that combines schedule and syntactic aspects in a tree shape, it becomes possible to apply tree matching and rewriting techniques without leaving the polyhedral representation.

3.1 Structural Schedule Tree Matchers

As first step, we define the schedule tree matchers to enable declarative description of pattern in the schedule tree. A structural matcher essentially replicates the node type-based structure of the schedule tree with additional filtering and wildcarding capabilities. A node matcher consists of an expected (possibly unspecified) node type, a list of child node matchers and an optional filter. In addition to all schedule tree node types, a matcher may expect *any* node type or a non-empty *list* thereof. Since schedule trees are often deeper than wider due to only two nodes types being allowed to have multiple children, matching a single node of any type is often sufficient to recognize large portions of a schedule tree. Tree leaves can be matched using a special node type.

3.2 Matching Procedure

The tree matcher is specified by the top node it must match, referred to as relative root. The matching procedure starts at the specified node in the schedule tree and performs simultaneous depth-first preorder traversal of the schedule tree and the tree formed by descendants of the relative root matcher. If a mismatch is detected until the entire matcher is traversed, it is immediately reported and the traversal stops. Our choice of explicitly marking tree leaves in the matcher allows us to easily transform the matcher into a sequential form if necessary. Once this transformation is performed, it becomes possible to apply more efficient string-based algorithms for matching against a set of strings or finding all substrings by treating each combination of node type and filter as a unique symbol in some abstract alphabet.

3.3 Programming Interface

The API to construct schedule tree matchers is designed to visually resemble the structure of the tree itself in a declarative way. Named variadic functions correspond to node types, and the argument lists enumerate child nodes. This approach enables static checking of most tree invariants, e.g., some node types only allowed to have one child or children of a specific type. Leading arguments include *optionally* a filtering function and a reference to the "placeholder" node. The filtering function allows the caller to control the matching more precisely by considering non-structural aspects of the schedule tree. For example, matching only one-dimensional band nodes requires both structural and non-structural properties. The references are used to capture certain nodes in the matched subtree, similarly to captured groups in regular expressions, for future use by the caller. If the captured nodes are non-empty, the underlying subtree is guaranteed to have the structure described by the matcher.

The matcher shown in Listing 3 matches all the sub-trees starting at a sequence node with exactly two filters as children, the first of which has a permutable band as a child (checked via a user-provided function `isPermutable`) while the second has no children.

```
auto matcher =
  sequence(
    filter(
      band(isPermutable, // filtering function
        anyTree()),      // wildcard node
    filter(
      leaf())));         // explicit leaf
```

Listing 3: Schedule tree matchers declaratively describe the structure of a tree.

3.4 Extensibility of the Approach: Traversal Matchers

Practically, filtering functions are not limited to matching decisions and can be used to extend the tree matching engine. For example, they can be used to include arbitrary tree traversals in the matching procedure. It suffices to provide a higher-order function that produces the necessary filtering functions, typically closures over the traversal procedure and the nested matcher. Listing 4 illustrates the approach by defining a "descendant" matcher for coincidence-featuring leaf bands and by using it while looking for sequence nodes that can have such descendants.

```
auto isCoincidentLeaf =
  band(hasCoincidence,
    leaf());
auto matcher =
  sequence(hasDescendant(isCoincidentLeaf),
    anyForest());
```

Listing 4: The matcher API leverages functional programming techniques to support extensions. Here, `hasDescendant` is a higher-order function.

3.5 Schedule Tree Builders

The imperative-style schedule tree construction interface does not allow for declarative tree construction. Instead, we provide schedule tree builders whose programming interface is close to that of the tree matchers. Named variadic functions specify the type of the node, the nesting of the function calls reflects the structure of the tree and the optional leading arguments accept functions that are used to build the non-structural properties of each specific node type (partial schedules for band nodes, conditions for filter nodes and so on). Once again, declarative-style interface requires other components of functional programming to be expressive enough, in particular, to enable lazy or delayed initialization of non-structural properties. The builder itself is merely a description of the subtree to build (one may think of it as an alternative, simplified schedule tree structure). It may be transformed into a standalone tree if possible, i.e., if it is rooted at a domain or an extension node, or grafted at a leaf of the existing tree.

3.6 Putting it All Together

The aforementioned lazy initialization supports the declarative find-and-replace procedure on schedule trees. First, a pattern is described using schedule tree matchers, optionally with callback functions to check additional node properties. Then, the transformed tree structure is described using the builders with delayed initialization of the node properties. Finally, one can traverse the tree in any order, trying to match a subtree rooted at the node being traversed and, in case of a match, replace the subtree with the one produced by the builder.

Listing 5 demonstrates how simple rectangular loop tiling transformation can be declared. The matcher looks for permutable band nodes anywhere in the tree and captures both the node and its child subtree. The builder splits the band into two nested bands by taking the integer division and the modulo parts of the schedule while keeping the child subtree intact. It can be trivially extended to support more advanced techniques like full/partial tile separation or to limit tiling to the deepest bands without loosing a significant portion of the transformation code clarity.

```
isl::schedule_node node, continuation;
auto matcher = band(node, isPermutable,
                 anyTree(continuation));
auto builder =
  band([&](){ return tileSchedule(node); },
    band([&]() { return pointSchedule(node); },
      subtree(continuation)));
replaceDFSPreorderOnce(schedule, matcher, builder);
```

Listing 5: Declarative specification of rectangular loop tiling. Functions tileSchedule and pointSchedule divide and take modulo by tile sizes, respectively.

4 Polyhedral Relation Matchers

4.1 General Concepts

Let us first introduce the concepts of access relation matchers before continuing with the matching algorithm and examples. Polyhedral relation matchers allow the caller to identify relations that have certain properties in a union of relations. In *isl* notation, individual relations in a union are uniquely identified by the space in which they live.

A polyhedral relation matcher is formed by a non-empty set of groups of relation property descriptors. For a relation to be considered for a match, it must have all properties described in a *group*. The connection between groups in the same matcher is controlled by the user. In particular, they may be allowed or not allowed to match the same relation in the union.

Beyond checking the properties, we are interested in capturing some information about the matching relation, e.g. the coefficients of the linear access function or whether a relation that was matched thanks to its surjectivity is also bijective. The capturing mechanism operates through *placeholders*, each of which has two data components: a constant *pattern* and a variable *candidate*. Each relation is checked against the pattern and, in case of a match, it may yield one or more candidates. The description of what constitutes a match and how the candidates are generated is external to the matching engine and can, e.g., be provided by the user. An example of a pattern that yields multiple candidates is "access relation with some output dimensions fixed to a non-parametric constant". In this case, a candidate *may* be generated for each output dimension fixed to a constant or, for other use cases, for each value of the constant.

A *match* is an assignment of candidates to placeholders. One union of relations may yield zero or more matches against the given matcher. In many cases, candidates assigned to different placeholders are required to be distinct. We require this by default, considering other cases *invalid assignments*. At the same time, if a placeholder is reused within the same matcher expression, we also require that all appearances get assigned the same candidate. In any case, the candidate comparison does not take into account the differences between spaces of the relations, only the candidate descriptions. This behavior allows the matcher to connect different relations in the union with more precision. For example, it captures the fact that exists two relations that are either both bijective or both non-bijective. To support edge cases, the caller can override the definition of valid assignment, e.g., to allow the same candidate to be assigned to different placeholders.

Matcher Specification, Extension and Composition

The implementation of the matching procedure, or the matching *engine*, is a template definition in C++ sense. An instantiation of the matching engine is specified by data structures for the pattern and the candidate. In addition, it implements functions to define a set of candidates for a given access relation and whether a candidate assignment to the matchers constitutes a valid match.

```
std::vector<Candidate> candidates(isl::map relation,
    Pattern pattern);
bool isValidMatch(std::vector<Placeholder>,
    std::vector<Candidate>);
```

Attempts to use different types of patterns in the same matcher expression are disallowed by construction. If several properties need to be combined to find candidates, the caller must defined how exactly two properties should be combined together, e.g. both present at the same time, at least one, and so on. We argue that, at a price of small code overhead, it allows to support an arbitrarily large combinatorial space of patterns combined freely from the simple ones.

4.2 Matching Procedure

The matching procedure is decoupled into two stages, providing sufficient flexibility without sacrificing performance.

First, the engine traverses all relations one by one and defines, for each placeholder, the set of suitable candidates using the user-provided function. If at least one of the placeholders has no suitable candidates, the absence of match is reported immediately and the procedure stops. Then the engine traverses the space of all possible assignments of candidates to the placeholders and checks whether the assignment is valid using another user-provided function. Since the space of possible assignments is combinatorially large, we opt for a branch-and-cut traversal approach. Partially-formed assignments are passed to the validation function before adding one more placeholder-candidate pair to the assignment. If the partial assignment is reported invalid, further exploration is not performed. This approach can be easily transformed into a branch-and-bound if the assignment needs to be optimal in some sense, or altered to change the exploration/validation ratio if the validation itself is expensive. In cases where invalid partial assignment does not mean that full assignment cannot be valid, the validity filter can always return “valid” for partial assignments.

4.3 Programming Interface by Example: Access Matchers

Let us illustrate relation matchers as applied to polyhedral access relations. In most cases, these relations are in fact affine functions in vector spaces mapping the schedule vector to the subscript vector. We provide pattern and candidate descriptions for simple affine expressions ($\omega = k * \iota + c$ where k and c coefficients form the pattern while ω and ι define a candidate by matching one of the output and input dimensions respectively) as well as for fixed non-parametric strides in the relation range. When targeting access relation unions, we can assume that the source space of all relations is the same, e.g., the schedule space, so it is convenient to only operate on the relation range.

The programming interface is similar in spirit to that of schedule tree matchers, except for the tree parent-child relations, and is based on nested function calls progressively constructing the matcher. Listing 6 illustrates how one can identify if the same 2D array is accessed directly and with transposition.

```
isl::ctx ctx = /*...*/;
auto _i = placeholder(ctx);
auto _j = placeholder(ctx);
auto _A = arrayPlaceholder();
auto matcher = allOf(access(_A, _i, _j),
                     access(_A, _j, _i));
auto result = match(readsAndWrites, matcher);
some_call(result[_i]); // extracts the candidate for _i
```

Listing 6: Access relation matcher for transposed accesses.

The example above uses additional syntactic sugar to operate separately on output dimensions of the access relation. Contrary to the generic property matchers that are applicable to entire relations, for accesses, it is often required to separately consider individual output dimensions. In the transposed access example, the caller would like to know that, in $(\dots) \rightarrow \{(i,j) \rightarrow A(a_1 = i, a_2 = j); (i,j) \rightarrow B(b_1 = j, b_2 = i)\}$, the first dimension of the output space in one relation (a_1) is equal to the second dimension of the output space in another relation (b_2). We handle such situations by augmenting the pattern description with the expected position in the output space, projecting the access relation onto that dimension and continuing to check the properties on a relation with one-dimensional output space. Such per-dimensional behaviour is particularly useful for, e.g., detecting the presence of temporal or spatial locality in array accesses. However, it is not applicable beyond relations that are also affine functions in some vector space.

4.4 Relation Transformers

Since relation matchers are significantly less constrained than tree matchers (due to relations being also less structured), we opted for an approach that expresses directly the *transformation* of a relation rather than building it from scratch. Each pattern-candidate pair may optionally provides a function that transforms a relation given the candidate assignment.

```
isl::map transformMap(isl::map map,
    const Candidate& candidate, const Pattern& pattern);
```

If provided, this function will be repeatedly applied to the matching relation for each candidate assignment. A sanity check verifies that the same relation does not get *transformed* through different placeholder *groups* (it may be matched several times as long as only one of the matches has an associated transformation function) and that each placeholder has a unique assignment within each relation.

For access relations, it is often convenient to rewrite some dimensions, i.e. array subscripts, keeping others without modification. Relation transformers can achieve that by using the same per-dimension separation mechanism as the relation matchers.

Listing 7 exemplifies the use of relation transformers to transpose accesses to 2D and 3D arrays. Note that each "replace" call describes an individual pattern that gets checked separately during the same pass of the replacement procedure.

```
isl::ctx ctx = /*...*/;
auto _i = placeholder(ctx);
auto _j = placeholder(ctx);
auto _k = placeholder(ctx);
auto result = findAndReplace(reads,
  replace(access(_j, _i), access(_i, _j)),
  replace(access(_k, _j, _i), access(_k, _i, _j)));
```

Listing 7: Access relation transformation transposing the two innermost subscripts of 2D and 3D arrays.

5 Case studies

In this section we demonstrate the applicability and flexibility of our framework considering three use cases. We start by showing how our framework improves upon state-of-the-art user-driven optimizers such as Clay, decoupling the transformation strategy from the code structure. Subsequently, with the second use case we demonstrate how we can achieve the same level of optimization of a newly introduces compiler optimization in the LLVM polyhedral optimizer in a more flexible and concise way, effectively reducing code requirements and unlocking the opportunity for a rule-based optimization engine. We conclude with the third use case showing how we can implement a data-layout transformation to reduce stream alignment issue in stencil patterns, namely dimension-lifted transposition.

5.1 Case: Generalizing Directive-Based Loop Transformations

Several tools were proposed to perform classical loop transformations, such as loop fusion or tiling [42], in the polyhedral framework, including AlphaZ [44], CHiLL [11, 33], Clay [3] among others. All these tools expose some language to specify program transformations applicable to loops. Although the loop transformations are abstracted to their common names, the languages can be seen as *imperative* in a sense that they require to specify *which loops* are targeted using external tags or language-level annotations.

Consider how tiling a simple loop nest is expressed in Clay, as shown in Listing 8. Each transformation directive starts with a "beta-prefix" that identifies the loop it applies to, followed by two target loop depths (where to place the tiles loop and the points loop in the nest) and, finally, by the requested tile size. Adding, for example, a time loop around this transformation, or an initialization statement for `C[i][j]` would break the transformation script immediately since the "beta-prefixes" and/or the loop depths would change. Worse, the user must keep track of the transformation effects on the loop structure to properly spell subsequent transformations. Also note that Clay and the related Chlore algorithm internally transform beta-prefixes into a tree structure, similar to the combination of sequence and filter nodes in the schedule tree [46].

```
/* Clay
  tile([0,0,0],1,1,32);
  tile([0,0,0,0],3,2,32);
  tile([0,0,0,0,0],5,3,32);
*/
for (int i = 0; i < 1024; i++)
 for (int j = 0; j < 1024; j++)
  for (int k = 0; k < 1024; ++k)
   C[i][j] += alpha * A[i][k] * B[k][j];
```

Listing 8: Core statement of the matrix-matrix multiplication.

With our approach, it is possible to make the transformations more *declarative*. Instead of binding it to specific loops, we can *declare* how a compatible schedule tree should look like and express the transformations in terms of matched nodes and relations between them. Both the matcher and the builder are expressed as compilable C++ code in Listing 9, offering virtually unlimited flexibility as to controlling program transformations.

```
isl::schedule_node node;
auto matcher =
  band(node, _and(is3D, isPermutable),
    leaf());
auto builder =
  band([&]() { return tileSchedule(node, {32,32,32}); },
    band([&]() { return pointSchedule(node, {32,32,32}); }));
```

Listing 9: Declarative transformation for tiling 3D loops with a tile size of 32.

The search-and-replace algorithm can now be applied to *any* arbitrarily complex schedule tree, repeatedly if necessary, tiling three-dimensional loops in the entire program. Note that we are using helper functions to define properties of the nodes in the transformed trees, such as `tileSchedule` and `pointSchedule` computing partial schedules of the bands as shown in Listing 9. A number of this function is shipped with the framework and others can be easily derived using *isl*.

5.2 Case: Hand-Tuned Optimization for GEMM-like Kernels

Generalized matrix multiplication (`gemm` BLAS kernel) is one of the important computation patterns and is the most optimized kernel in history [26]. Pre-optimized versions of `gemm` are commonplace, yet they often under-perform on matrix sizes different from those they were designed [35]. In such cases, even state-of-the-art compilers only achieve a fraction of the theoretical machine performance for a simple textbook style implementation [16].

A recent improvement in Polly loop optimizer introduced a custom transformation for `gemm`-like kernels that is controlled outside of the main affine scheduling mechanism [16]. This transformation applies to a generalized case of tensor contraction of the form `C[i][j] = E(A[i][k], B[k][j], C[i][j])` where the dimension `k` is contracted and `E` is some operation between tensors. The matrix-matrix multiplication from Listing 8 is an example of such contraction with $E = (\times, +)$. To qualify for the transformation, the kernel must respect the following:

- perfectly nested loop satisfying the requirements of the polyhedral model;
- contains three non-empty one dimensional for loops with induction variables incremented by one;

- contains an innermost statement $C_{\pi C(IJ)} = E(A_{\pi A(IK)}, B_{\pi B(KJ)}, C_{\pi C(IJ)})$ where $A_{\pi A(IK)}$, $B_{\pi B(KJ)}$, $C_{\pi C(IJ)}$ and $\pi A(IK)$, $\pi B(KJ)$, $\pi C(IJ)$ are access to tensors A, B, C and permutations of the enclosed indexes respectively. E is a generic expression that contains at least three reads from tensors A, B and C.

- the interchange of I and J is valid, while for P is interchangeable iff P contains only one element or an associative operation is used to update C.

The candidate loops are found implementing the aforementioned conditions as schedule tree and access relation matchers. In particular, we are looking for a three-dimensional permutable band with a single statement (leaf) featuring specific access patterns: at least three two-dimensional read accesses to different arrays; an permutation of indices that satisfies the placeholder pattern $[i, j] \to [i, k][k, j]$. This can be expressed by using a relation matcher and a set of callback functions as shown in Listing 10.

```
auto isGemmLike = [&](isl::schedule_node) {
  auto _i = placeholder(ctx);
  auto _j = placeholder(ctx);
  auto _k = placeholder(ctx);
  auto _A = arrayPlaceholder();
  auto _B = arrayPlaceholder();
  auto _C = arrayPlaceholder();

  /* Restrict the access relations to this subtree */
  reads = reads.intersect_domain(node.domain());
  writes = writes.intersect_domain(node.domain());
  auto mRead  = allOf(access(_A, _i, _j),
                      access(_B, _i, _k),
                      access(_C, _k, _j));
  auto mWrite = allOf(access(_A, _i, _j));
  return match(reads, mRead).size() == 1 &&
    match(writes, mWrite).size() == 1;
};

auto matcher = band(_and(is3D, isPermutable, isGemmLike)),
                 leaf());
```

Listing 10: Combined Schedule Tree and Access Matcher for a simple Tensor Contraction

The kernel transformation is derived from [16] as follows: First, we re-arrange the band dimensions such that `j` will be the outermost dimension, followed by `k` and `i`. Then we apply multiple level tiling and loop interchange to create macro and micro-kernel. Specifically, the builder for the macro-kernel reported in Listing 11 performs L2-cache tiling and interchanges the newly created point loops.

```
isl::schedule_node node = /* obtain node, e.g., from a matcher */;
auto macroKernel =
  band([&]() { return tileSchedule(node, /*L2-specific sizes*/); },
    band([&]() { return swapDims(pointSchedule(node), -2, -1); }
));
```

Listing 11: Tiling for L2-cache locality. Negative indexes in `swapDims` have Python semantics.

After applying the first builder, we define and apply the second builder to create the micro-kernel tiling the points loops so that they fit into vector registers and fully unroll the new innermost loops to simplify subsequent vectorization.

```
/*Apply the previous builder and get the child band.*/
node = macroKernel.insertAt(node.cut()).child(0);
auto microKernel =
  band([&]() { return tileSchedule(node, /*Vectorization sizes*/); },
    band([&]() { return unrollAll(pointSchedule(node)); }));
```

Listing 12: Further tiling and unrolling to enable vectorization.

Overall, the combination of matcher and builder holds in dozens of lines whereas Polly's implementation of the same transformation needs several hundred lines.

We believe that our declarative approach can greatly simply the optimizer implementation through collection of pattern-based optimizations. It also unleashes the opportunity for a rule-based optimization engine where expert user can try out new optimization strategies, and plug them in the optimizer by declaratively restricting the parts of the program to which they apply.

5.3 Case: Stencil Vectorization through Data-Layout Transformation

Stencil patterns occupy a special place in the loop optimization literature. As they typically involve accesses to multiple adjacent array elements along multiple dimensions inside nested loops with (mostly) static control flow, they fit the polyhedral model. Furthermore, many important practical use cases can be implemented as stencils, for example difference schemes for systems of differential equations or convolutional neural networks.

However, polyhedral compilation based on affine scheduling are often counterproductive for stencil patterns. Direct attempts to minimize dependence distances lead to effective serialization of computation and/or substantially complex control flow overhead, making the transformed code *slower* than the original [41, 47]. Several techniques, more or less closely related to the polyhedral compilation, address specifically stencil parallelization and vectorization [18]. Since these techniques are often not adapted to non-stencil cases, one has to first find the stencil-like part of the program and then apply the transformation.

Declarative schedule tree and relation patterns simplify the matching of stencils in the polyhedral representation. Let us illustrate how a stencil vectorization technique based on data layout transformation *that has not been defined in terms of polyhedral model* [23] can be expressed in our framework.

Candidate Loops The data-layout transformation (DLT) technique [23] applies only to *innermost* loops with no loop-carried dependences. In schedule tree nomenclature, we should start by finding all band nodes that don't have other band nodes as children and that have the last coincidence flag set (assuming coincidence was computed based on all dependences).

```
band(capturedBand,
     and_(not_(hasDescendant(band(anyTree()))),
          isLastCoincident),
     anyTree());
```

Note that if the band is permutable, it does not matter whether the coincident flag is set for the last dimension or any other of the band member since the loops can be trivially permuted. This extension to the original technique and the corresponding tree transformation are easy to propose and implement in our approach but were not considered by the original paper.

Access Strides The transformation validity is further restricted to statements that access either the same or the consecutive elements of an array in consecutive iterations of the innermost loop. This can be easily matched against by computing the range *stride* of the access relation, i.e. the constant distance between elements in its range. From the relation matcher perspective, the pattern includes the expected stride along with the algorithm to compute it while the candidate is just the space of the matching relation. This can be represented by saying all the last (Python-style index `-1`) access must have stride either 0 or 1.

```
match(rw, access(dim(-1, stride(ctx, 0)))).size() +
match(rw, access(dim(-1, stride(ctx, 1)))).size() ==
match(rw, access(any())).size()
```

Vector Lane Conflicts Finally, DLT is only necessary when there exist *vector lane* conflicts that cannot be removed through loop shifting, in particular, if the same value is accessed through *different references* in different iterations of the innermost loops. This condition can be transformed into a sequence of set operations forming a system of constraints followed by an emptiness check.

While it can be used inside the relation matcher to create a list of candidates before checking that all accesses do match, it is arguably more pragmatic to perform the operations directly.

Injecting Data Layout Transformation The data layout transformation consists in placing hitherto adjacent array elements at distance L from each other, where L is the number of vector lanes. This is expressed by a union of affine functions defined over disjoint subdomains $\bigcup_{s\in[0,L),s\in\mathbb{Z}}\{(\vec{\iota}, i) \to (\vec{\alpha}, a) \mid \vec{\alpha} = \vec{\iota} \wedge a = s - B_L + L(i - \lfloor\frac{s\cdot(B_U-B_L+1)}{L}\rfloor) \wedge s\frac{B_U-B_L+1}{L} \leq i - B_L < (s+1)\frac{B_U-B_L+1}{L}\}$ where B_L and B_U are the lower and the upper inclusive bound on the accessed elements. If this transformation had been performed on the iteration space rather than on the subscript space, it would have corresponded to loop strip-mining followed by loop interchange and coalescing. Once such union of affine functions is constructed, it can be used to declare a schedule tree builder that injects the transformation itself as presented in Listing 13. First, it introduces the new statements for the copies to and from the transformed array through an extension node. Below it, copies "to" are scheduled before the main computation, which itself is scheduled before the copies "from" using a combination of sequence and filter nodes. Finally, partial schedules are specified for the copies and the original subtree is replicated for the main computation. In practice, the transformation is only applicable to loop iterations that fully fit into vector lanes [2] and requires additional edge case otherwise. While these cases can also be expressed declaratively as tree builders, they are not essential to our presentation so we omit them for the sake of clarity. The code below features the corresponding builder construction, which clearly indicates the intended subtree structure.

This example lays the foundation for most data-layout or memory-related transformation that rely on *copying* the data from one array to another and back. Most such transformations follow the extension/sequence/3-filter pattern with only the properties of nodes differing. Copies "to" or "from" additional memory may be omitted in cases where the initial or the final values are irrelevant to the task, creating two other possible tree transformation patterns.

Access rewriting The final step of DLT consists in changing the original stencil computation to access the transformed array instead of the original one and, optionally, emitting vectorization pragmas. To enable vectorization, it is necessary to access the transformed array in the *sequential* order and the data required by each iteration now requires different subscripts. In particular, hitherto adjacent elements are now located at distance L. This can be easily expressed using our relation rewriting mechanism. The pattern constitutes the single access subscript (last range dimension of the access relation) and the candidates are affine functions that define that subscript[3]. These affine functions

[2]This can be ensured with, e.g., full/partial tile separation given tile size is equal to the number of vector lanes.

[3]In the polyhedral model, subscripts should be affine.

```
auto s = [&]() { return capturedBand
                  .get_prefix_schedule_union_map(); };
auto dlt = /* construct the DLT function */;
auto from = dlt.set_tuple_id(isl::dim::range, "from");
auto to   = dlt.set_tuple_id(isl::dim::range, "to");
extension([&](){ return from.apply_domain(s()).unite(
                          to.apply_domain(s()));},
  sequence(
    filter([&](){ return to.range();},
      band([&](){ return to.range().affine_hull()
                           .wrapped_domain_map();}))
    filter([&](){ return s().domain();},
      subtree(capturedBand))
    filter([&](){ return from.range();},
      band([&](){ return from.range().affine_hull()
                           .wrapped_domain_map();})));
```

Listing 13: Base case of data layout transformation for stencil vectorization. Other data motion transformations follow the same tree structure.

can be decomposed into the linear and the constant part, and the latter needs to be scaled by L in order to reflect the data layout transformation.

```
auto aff = affine(ctx);
findAndReplace(stride1Accesses,
    replace(access(dim(-1, aff)),
            access(dim(-1, aff.payload_.linear +
                       L * aff.payload_.constant)));
```

Only the stride-1 accesses should be transformed. They can be easily extracted by reusing a part of the polyhedral relation matcher used during the detection phase. Depending on the underlying implementation, this rewriting can happen directly on the access relations or during code generation.

Even including the pattern and candidate descriptors, the entire code for implementing this advanced vectorization technique fits into a couple of hundred lines and can be easily combined with other techniques expressed in the same way.

5.4 Case: Mapping to GPUs and Accelerators

Recent evolutions in hardware for high-performance computing led to the increasing popularity of the accelerator-based architecture. GPUs and other accelerators now power world's fastest supercomputers. Most accelerators share a common programming model that consists of:

- one or several accelerator programs using a specialized language or a subset of existing language (typically C++), referred to as *kernels*;

- a host program that manages the acclerator device, explicitly transfers the data between host and accelerator, and schedules the kernels for execution.

To make computation efficient, accelerators impose stringent conditions on the operations that can be executed by the kernels, especially when it comes to parallel execution. Therefore, the first task most automatic device mappers perform is to find parts of the program that can be executed on the accelerator.

Let us consider GPU mapping as an illustrative example using the algorithm implemented in PPCG that offers a good trade-off between performance and simplicity [38]. PPCG operates on schedule trees and starts by traversing the tree from the top to find band nodes that have both the permutability and at least one coincident flag. Subtrees rooted at such nodes are mapped to the GPU entirely and therefore not traversed in search of other bands. This can be expressed declaratively as "permutable coincident bands that don't have permutable coincident bands as ancestors", which can be trivially transformed into a schedule tree matcher as shown in Listing 14.

```
isl::schedule_node node, child;
auto mappable = and_(isPermutable, hasCoincidence);
auto matcher =
  band(node,
       and_(mappable,
            not_(hasAscendant(band(mappable, anyTree())))),
       anyTree(child));
```

Listing 14: Schedule Tree Matcher mimicing PPCG's behavior.

When all such nodes are found, the basic mapping transformation is performed as follows. First, a special *context* node introduces thread and block identifiers to the program. They can only appear below this node, which separates the top part of the tree that is transformed to host code from the bottom subtree that is transformed to the GPU device code. Below the context a *extension* node introduces calls to the data motion functions that ensure the data is transferred to and from device before and after the kernel, respectively. Depending on the host API being synchronous or not, explicit device synchronization calls can be emitted as well. Data transfers and synchronizations are ordered using a *sequence* node with corresponding filters. Finally, the original band node is tiled with the resulting tiles loops being mapped to blocks and points loops being mapped to threads using filter nodes.[4] All these steps can be concisely expressed as a schedule tree builder in Listing 15.

Alternatively, the GPU mapping can be *composed* from different builders applied in sequence, hence reusing their definitions. First, a data-copy builder similar to that of Listing 13 performs data copies. Then, a tiling builder similar to Listing 9 kicks in. Finally, a thread- and block-mapping builder, or even the

[4]GPU accelerator code is expressed in terms of one thread rather than parallel loops, so filters restrict the generated code to that one thread.

```
auto builder =
  context(/*introduce block/thread ids*/,
    extension("{[...] -> copy_to[...];"
              " [...] -> copy_from[...];"
              " [...] -> dev_sync[]",
      sequence(
        filter("{copy_to[...]}"),
        filter([&]() { return node.domain(); },
          filter(/*block mapping*/,
            band([&]() { return tileSchedule(node); },
              filter(/*thread mapping*/,
                band([&]() { return pointSchedule(node); },
                  subtree(child)))))),
        fitler("{copy_from[...]}"),
        filter("{dev_sync[]}"))));
```

Listing 15: Schedule Tree Builder mimicing PPCG's behavior.

same builder with different identifiers applied twice can be used to introduce thread and block identifiers. First-class support for builder combination is left for future work.

Once the schedule tree is transformed, a full implementation of an automatic GPU mapper must also provide a customized code generator that emits either CPU or GPU code depending on its position in the schedule tree. This can be done, for example, by keeping track of the points that belong to a subtree that got mapped to the GPU, which can be obtained when the matcher is called.

While this example demonstrates only the minimal GPU mapping strategy, it shows how easily one can devise new strategies using our framework. For example, one can chose to exploit finer-grain parallelism by mapping the innermost mappable loops instead of the outermost. This only requires to replace `hasAscendant` with `hasDescendant` in the matcher. More complex strategies like the one proposed in Tensor Comprehensions [35] can be implemented as a series of matchers and builders. Finally, intra-kernel optimizations such as explicit copies to fast shared or private memory can be easily obtained by modifying the data-copy builder of Listing 13. In practice, data-copy transformations only differ by the mapping function they use, while the schedule tree structure remains essentially the same. Thanks to tree node properties being decoupled from its structure in schedule tree builders, it is possible to make the copy builder parameterizable with the mapping and derive the remaining properties from that. Similarly, the entire device mapping builder can be adapted to different architectures, including the upcoming ones for, e.g, near-memory computing, by simple changes to node properties and code generation.

6 Related Work

6.1 Directive-based Transformation Engines

Kelly et.al. proposed arguably the first framework to expose high-level transformation on top of the polyhedral model [25]. Subsequently, Girbal et.al. proposed the URUK framework to apply loop transformations for cache hierarchy and parallelism based on unimodular schedules [17]. Yuki et.al. developed AlphaZ, a framework that allows the users to express the program as a set of equations based on the Alpha language and manipulate it using script-driven transformations [44]. In their implementation, the authors also support memory (re)-allocation and explicitly represent reductions. Similarly, Yi et.al. presented POET, an interpreted program transformation language designed for applying and explore complex code transformations in different programming languages [43]. Donadio et.al. introduced Xlanguage, an embedded DSL based on C/C++ pragma-based that allows the users to generate multi-version programs specifying the type of transformations to apply as well as the transformation parameters [12]. Bagnères et.al provided feedback from a polyhedral compiler by expressing it as a sequence of loop transformation directives [3]. Their input language, Clay, and the related Chlore algorithm allow the users to examine, refine or freeze sequences of loop transformation directives. Chen et.al. introduced CHiLL, a high-level transformation and parallelization framework that makes use of a model-driven empirical optimization engine to generate and evaluate different code variants [11]. It also allows the user to define a transformation script specifying how the code should be optimized. The framework has also been extended with high-level constructs to generate GPU code [32].

The vast majority of polyhedral directive-based loop transformation engines require the user to target specific loop nests or statements within them, as well as to track the changes to the loop structure throughout the transformation process. Our approach on the other hand is to declaratively describe parts of the program that are suitable for a particular optimization making it applicable to arbitrarily complex programs without modifying the transformation script. Since the schedule tree also captures the syntactic information *via* sequence and filter nodes, one can also use our framework to target specific syntax-driven parts of the program rather than their properties.

6.2 Code Offloading for Accelerators

Hseih et.al. proposed to offload code to accelerators based on a simple cost function [24]. The key idea is to statically identify the code blocks with greater potential in bandwidth saving. If the bandwidth saving in offloading the code block is higher than the cost of initiate and complete it the offload block is marked as "beneficial" to be offloaded. Pattnaik et al. developed an affinity prediction model to understand where to execute a given kernel, i.e., main core or near-memory accelerator [30]. The prediction model is based on five prediction metrics (i.e. memory intensity) and it is computed offline. Nair et.al. proposed a code offloading mechanism based on OpenMP 4.0 user annotations [29]. Their compiler uses OpenMP pragmas to identify code section that will be executed on the near-memory accelerator, and identify the data region accessed. CAIRO relies on a LLC cache profiler and analytical model to decide potential offloading candidates [22]. Nai et.al. proposed to offload all the graph operation on the graph properties on the near-memory accelerator mapping host atomic instructions directly into near-memory atomic instructions using uncacheable memory support available in modern architectures [28]. Ahn et.al. proposed an ISA extension on the host processor allowing the programmer to decide what to offload [1].

All these algorithms are *complementary* to our approach as they mostly control *when* to perform the offloading while featuring a hard-coded implementation of how to perform it. They can be used in the matching procedure to find subtrees suitable for offloading while the transformation is performed by a combination of schedule tree builder, relation rewriter and code generator.

7 Conclusion

We presented a flexible framework to perform pattern matching and declaratively express transformations on schedule trees, an internal representation of the polyhedral compiler. Unlike the majority of polyhedral approaches, pattern matching tree-rewriting transformations in our framework can be implemented and augmented using classical compiler construction technology applicable to other tree-like structures in the internal representation of a compiler. We believe our approach can simplify the integration of polyhedral optimizers into production compilers.

Our approach allowed us to easily express and compose a set of program transformations that use the polyhedral analyses but do not require the conventional combination of affine loop transformations, usually obtained by solving linear optimization problems. Complementary to existing loop transformation directives, it now becomes possible to declare, refine and reuse transformations for, e.g., data layout or device mapping.

Future work will focus on serializing and sharing the match and build patterns for both the schedule trees and the expressions, as well as on more efficient matching algorithms and whole-program uses. A domain-specific declarative polyhedral language can also be considered to avoid the syntactic hurdles of using declarative lazy-evaluated style in C++. Finally, reusable code polyhedral transformation patterns expressed as matcher/builder pairs can be shipped together with high-performance libraries that rely on generated code or with compilers for novel architectures.

Acknowledgments

This work was partially supported by the European Commission Horizon 2020 programme through the MNEMOSENE grant agreement, id. 780215 and the NeMeCo grant agreement, id. 676240 as well as through Polly Labs (Xilinx Inc, Facebook Inc, and ARM Holdings) and the Swiss National Science Foundation through the Ambizione program.

References

[1] J. Ahn, S. Yoo, O. Mutlu, and K. Choi. Pim-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, pages 336–348, June 2015.

[2] L. Bagnères, O. Zinenko, S. Huot, and C. Bastoul. Opening Polyhedral Compiler's Black Box. In *CGO 2016 - 14th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, Barcelona, Spain, Mar. 2016.

[3] L. Bagnères, O. Zinenko, S. Huot, and C. Bastoul. Opening polyhedral compiler's black box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 128–138, New York, NY, USA, 2016. ACM.

[4] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *PACT'13 IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 7–16, Juan-les-Pins, France, September 2004.

[5] C. Bastoul. Mapping Deviation: A Technique to Adapt or to Guard Loop Transformation Intuitions for Legality. In *CC'2016 25th International Conference on Compiler Construction*, Barcelone, Spain, Mar. 2016.

[6] C. Bastoul, A. Cohen, S. Girbal, S. Sharma, and O. Temam. Putting polyhedral loop transformations to work. In L. Rauchwerger, editor, *Languages and Compilers for Parallel Computing*, pages 209–225, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[7] U. Bondhugula, S. Dash, O. Gunluk, and L. Renganarayanan. A model for fusion and code motion in an automatic parallelizing compiler. In *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 343–352, Sept 2010.

[8] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A practical and fully automatic polyhedral program optimization system. In *ACM SIGPLAN PLDI*, 2008.

[9] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. *ACM SIGPLAN Notices*, 43(6):101–113, 2008.

[10] C. Chen. Polyhedra scanning revisited. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 499–508, New York, NY, USA, 2012. ACM.

[11] C. Chen, J. Chame, and M. Hall. Chill: A framework for composing high-level loop transformations. Technical report, Citeseer, 2008.

[12] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali. A language for the compact representation of multiple program versions. In E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, editors, *Languages and Compilers for Parallel Computing*, pages 136–151, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[13] P. Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, Feb 1991.

[14] P. Feautrier and C. Lengauer. Polyhedron Model. In D. Padua, editor, *Encyclopedia of Parallel Computing*, pages 1581–1592. Springer, 2011.

[15] P. Feautrier and C. Lengauer. *Polyhedron Model*, pages 1581–1592. Springer US, Boston, MA, 2011.

[16] R. Gareev, T. Grosser, and M. Kruse. High-performance generalized tensor operations: A compiler-oriented approach. *ACM Trans. Archit. Code Optim.*, 15(3):34:1–34:27, Sept. 2018.

[17] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam. Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies. *Int. J. Parallel Program.*, 34(3):261–317, June 2006.

[18] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege. Hybrid hexagonal/classical tiling for gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 66:66–66:75, New York, NY, USA, 2014. ACM.

[19] T. Grosser, S. Verdoolaege, and A. Cohen. Polyhedral AST generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37(4):12:1–12:50, July 2015.

[20] T. Grosser, S. Verdoolaege, and A. Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37(4):12:1–12:50, July 2015.

[21] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011.

[22] R. Hadidi, L. Nai, H. Kim, and H. Kim. Cairo: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory. *ACM Trans. Archit. Code Optim.*, 14(4):48:1–48:25, Dec. 2017.

[23] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan. Data layout transformation for stencil computations on short-vector simd architectures. In *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, CC'11/ETAPS'11, pages 225–245, Berlin, Heidelberg, 2011. Springer-Verlag.

[24] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler. Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pages 204–216, June 2016.

[25] W. Kelly and W. Pugh. A framework for unifying reordering transformations. Technical report, 1998.

[26] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti. Analytical modeling is enough for high-performance blis. *ACM Trans. Math. Softw.*, 43(2):12:1–12:18, Aug. 2016.

[27] R. T. Mullapudi, V. Vasista, and U. Bondhugula. Polymage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News*, 43(1):429–443, Mar. 2015.

[28] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim. Graphpim: Enabling instruction-level pim offloading in graph computing frameworks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 457–468, Feb 2017.

[29] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. . Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura. Active memory cube: A processing-in-memory architecture for exascale systems. *IBM Journal of Research and Development*, 59(2/3):17:1–17:14, March 2015.

[30] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das. Scheduling techniques for gpu architectures with

processing-in-memory capabilities. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, pages 31–44, Sept 2016.

[31] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache. Graphite: Polyhedral analyses and optimizations for gcc. In *Proceedings of the 2006 GCC Developers Summit*, page 2006. Citeseer, 2006.

[32] G. Rudy. *CUDA-CHiLL: A programming language interface for GPGPU optimizations and code generation.* PhD thesis, School of Computing, University of Utah, 2010.

[33] G. Rudy, M. M. Khan, M. Hall, C. Chen, and J. Chame. A Programming Language Interface to Describe Transformations and Code Generation. In K. Cooper, J. Mellor-Crummey, and V. Sarkar, editors, *Languages and Compilers for Parallel Computing*, number 6548 in Lecture Notes in Computer Science, pages 136–150. Springer Berlin Heidelberg, Oct. 2010.

[34] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR*, abs/1802.04730, 2018.

[35] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[36] S. Verdoolaege. Isl: An integer set library for the polyhedral model. In *Proceedings of the Third International Congress Conference on Mathematical Software*, ICMS'10, pages 299–302, Berlin, Heidelberg, 2010. Springer.

[37] S. Verdoolaege. Counting affine calculator and applications. In *First International Workshop on Polyhedral Compilation Techniques (IMPACT'11), Chamonix, France*, 2011.

[38] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, Jan. 2013.

[39] S. Verdoolaege and T. Grosser. Polyhedral extraction tool. In *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, Jan. 2012.

[40] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen. Schedule Trees. In *4th Workshop on Polyhedral Compilation Techniques (IMPACT, Associated with HiPEAC)*, page 9, Vienna, Austria, Jan. 2014.

[41] S. Verdoolaege and A. Isoard. Consecutivity in the isl polyhedral scheduler. 2017.

[42] M. J. Wolfe. *High Performance Compilers for Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[43] Q. Yi. Poet: A scripting language for applying parameterized source-to-source program transformations. *Softw. Pract. Exper.*, 42(6):675–706, June 2012.

[44] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye. Alphaz: A system for design space exploration in the polyhedral model. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 17–31. Springer, 2012.

[45] J. Zhao, M. Kruse, and A. Cohen. A polyhedral compilation framework for loops with dynamic data-dependent bounds. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 14–24, New York, NY, USA, 2018. ACM.

[46] O. Zinenko. *Interactive Program Restructuring.* PhD thesis, Université Paris-Saclay, 2016.

[47] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen. Unified Polyhedral Modeling of Temporal and Spatial Locality. Research Report RR-9110, Inria Paris, Nov. 2017.

[48] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen. Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 3–13. ACM, Feb. 2018.

Contents

RESEARCH CENTRE
PARIS

2 rue Simone Iff - CS 42112
75589 Paris Cedex 12

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399