Declarative Loop Tactics for Domain-specific Optimization

LORENZO CHELINI, Eindhoven University of Technology and IBM Research Zurich
OLEKSANDR ZINENKO, Inria
TOBIAS GROSSER, ETH Zurich
HENK CORPORAAL, Eindhoven University of Technology

Increasingly complex hardware makes the design of effective compilers difficult. To reduce this problem, we introduce *Declarative Loop Tactics*, which is a novel framework of composable program transformations based on an internal tree-like program representation of a polyhedral compiler. The framework is based on a declarative C++ API built around easy-to-program matchers and builders, which provide the foundation to develop loop optimization strategies. Using our matchers and builders, we express computational patterns and core building blocks, such as loop tiling, fusion, and data-layout transformations, and compose them into algorithm-specific optimizations. Declarative Loop Tactics (Loop Tactics for short) can be applied to many domains. For two of them, stencils and linear algebra, we show how developers can express sophisticated domain-specific optimizations as a set of composable transformations or calls to optimized libraries. By allowing developers to add highly customized optimizations for a given computational pattern, we expect our approach to reduce the need for DSLs and to extend the range of optimizations that can be performed by a current general-purpose compiler.

CCS Concepts: • **Software and its engineering → Compilers**;

Additional Key Words and Phrases: Loop tactics, polyhedral model, declarative loop optimizations

ACM Reference format:
Lorenzo Chelini, Oleksandr Zinenko, Tobias Grosser, and Henk Corporaal. 2019. Declarative Loop Tactics for Domain-specific Optimization. *ACM Trans. Archit. Code Optim.* 16, 4, Article 55 (December 2019), 25 pages.
https://doi.org/10.1145/3372266

1 INTRODUCTION

As hardware increases in complexity, it becomes difficult for general-purpose compilers to produce efficient code automatically, as they operate at a very low level driven by one-size-fits-all optimization strategies [27]. As a consequence, when a substantial level of performance is required, we often rely on hand-written optimizations performed by expert programmers using low-level and architecture-specific constructs. Unfortunately, code optimized for performance obscures the high-level algorithm and is tied to a specific architecture, thus impeding code portability and

This work was partially supported by the European Commission Horizon 2020 program 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.
Authors' addresses: L. Chelini, Eindhoven University of Technology and IBM Research Zurich; email: l.chelini@tue.nl; O. Zinenko, Inria; email: oleksandr.zinenko@inria.fr; T. Grosser, ETH Zurich; email: tobias.grosser@inf.ethz.ch; H. Corporaal, Eindhoven University of Technology; email: h.corporaal@tue.nl.

1544-3566/2019/12-ART55
https://doi.org/10.1145/3372266

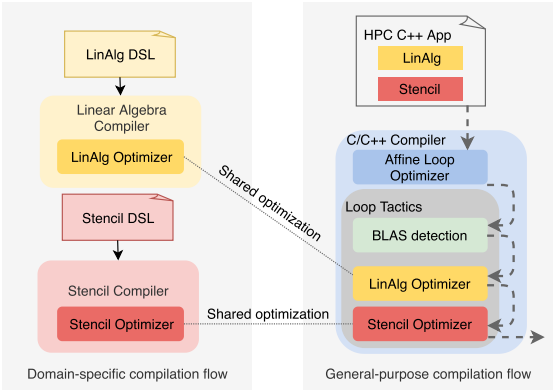

Fig. 1. Declarative Loop Tactics (dark gray box), or simply Loop Tactics, enable easy implementation of advanced domain-specific optimizations used in DSL compilers, and transparent BLAS invocation into a general-purpose compiler by means of a declarative C++ API.

readability. GEMM kernels, for example, require specific data-layout and loop-tiling techniques to improve temporal and spatial locality [31]. However, stencil kernels benefit from a specific data-layout transformation for efficient vectorization on short-SIMD architectures and the application of loop tiling techniques to improve temporal locality [22].

Combining all these techniques is difficult for experienced programmers and definitely beyond the scope of general-purpose compilers, the main reason for which is the lack of a framework to identify and represent computational patterns. This problem is typically addressed by using hand-optimized libraries or developing domain-specific languages (DSLs). Although DSLs have been proven to be effective, they are not immediately applicable to general-purpose code, and their development requires effort and expertise in multiple domains [44]. Worse, DSLs are usually stand-alone solutions with little reusability of transformations and do not compose well enough to allow the optimization of multi-domain applications [46]. Optimized libraries allow programmers to build applications from high-performance routines, the use of which, however, is still restricted to expert users and requires manual effort to port legacy code [45].

We propose the idea of *loop tactics* and its implementation as a framework to express computational patterns and corresponding transformations *declaratively*. The idea is illustrated in Figure 1. By expressing the computational pattern declaratively on a higher-level (polyhedral) representation, and the transformation to apply in terms of basic-block components, we make it easier to embed domain-specific optimizations directly into a general-purpose compiler that is unaware of domain-specific representation. Furthermore, the pattern matching capability allows the compiler to recognize program fragments corresponding to highly optimized vendor routines. Such fragments can be replaced—transparently for the application—with efficient library calls whenever available or automatically transformed otherwise. Loop Tactics combines[1] polyhedral analyses and program transformations with more conventional compiler construction techniques based on tree rewriting. It is built on top of the popular *isl* library [50] and provides a declarative C++ API.

We make the following contributions:

- *Loop Tactics*, a framework for declaratively expressing computational patterns and a set of transformations as matcher-builder pairs in the polyhedral model (Section 3).

[1]We use the singular third-person for Loop Tactics.

- Its implementation embedded both into a source-to-source compiler and a general compilation flow based on LLVM (Section 4).
- We demonstrate how to use Loop Tactics to perform domain-specific optimizations [22, 31], notably reducing code complexity and footprint compared to manual pattern matching [15] (Sections 5.1 and 5.2).
- We demonstrate how to use Loop Tactics to invoke routines from vendor-optimized libraries automatically and transparently. If optimized routines are not available for the target or no patterns are detected, then the generated code is *on par* with Polly [19] and competitive with Pluto [7]—both of them state-of-the-art polyhedral optimizers (Section 5.3).

2 POLYHEDRAL COMPILATION

The *polyhedral model* is a unified framework to model and transform loops in imperative programs [14]. After more than three decades of active research and attempts to integrate this model into production compilers [6, 21, 36], the polyhedral model has recently attracted renewed attention as a method for generating efficient code that targets both CPUs and accelerators from domain-specific languages [34, 49]. Iteration domains, access relations, and schedules are the core constituents of the model.

2.1 Domains and Memory Accesses

The algebraic representation is based on integer sets and relations. The model applies to static control parts (SCoPs) that consist of loops whose boundaries are affine functions of the surrounding loop iterators and parameters (values that are unknown at compilation time but constant at run time). Every iteration of these loops is identified by an integer vector in a k-dimensional space, where k is the depth of nested loops, the coordinates of which correspond to the values of the loop induction variables. Each statement other than control-flow constructs has an associated symbolic name and a set of integer vectors describing the iterations at which it is executed, commonly referred to as the *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 is expressed as $\{S_1(i,j) \mid 0 \leq i,j < N; S_2(i,j,k) \mid 0 \leq i,j,k < N\}$ in the tagged-tuple notation introduced by `iscc` [51], where N is a *parameter*. 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 kernel.

Internal computation is rendered as arithmetic expressions and function calls with array elements as arguments. The conditions imposed on the array subscripts are essentially the same as the control-flow conditions. Array accesses can be thus precisely encoded as relations between vectors in the iteration space and vectors in the array space whose coordinates are values of the accessed subscripts. These relations are defined by piece-wise quasi-affine functions. In our running example from Listing 1, the access relation for statement `S1` is $\{S_1(i,j) \rightarrow \text{beta}(); S_1(i,j) \rightarrow \text{D}(i,j); S_1(i,j) \rightarrow \text{C}(i,j)\}$.

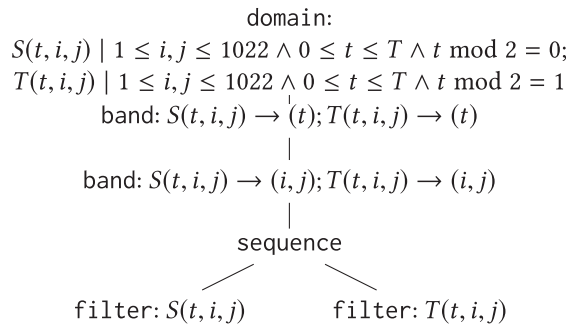

Fig. 2. Schedule tree representation of Listing 2.

2.2 Schedule Trees

The iteration domain defines the statement instances that should be executed but not their order. The latter is defined by a *schedule* that maps points in the iteration space to points in the time space. Although it is possible to express schedules as relations or piece-wise quasi-affine functions, it is often undesirable to do so, because statements are likely to share part of their schedules due to the commonly nested structure of the code. In addition, 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. For example, the schedule of Listing 2 in relation form is $\{S(t,i,j) \rightarrow (t,i,j,0); T(t,i,j) \rightarrow (t,i,j,1)\}$. Note the duplication of (t,i,j) and the presence of auxiliary dimensions to specify the execution order within the loop nest.

Addressing these challenges, Verdoolaege et al. [53] proposed *schedule trees* as a way to represent schedules in the polyhedral model. In schedule trees, statements that share a common partial schedule share an ancestor that describes this partial schedule, thus removing duplication. This loop nesting maps naturally to a parent-child relation, whereas textual ordering maps to the left-to-right order of sibling nodes.

Nodes in schedule trees can be one of 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 a *tilable band* of loops [7]);
- *Filter*—restricts the instances of the iteration domain;
- *Sequence*—imposes the order among its children;
- *Extension*—introduces new statement instances local to the subtree with respect to its prefix schedule.

All nodes except *sequence* have at most one child. Listing 2 presents a 2D stencil program. The corresponding schedule tree is depicted in Figure 2. It uses two band nodes: the outer one provides a partial schedule representing the individual time steps, whereas the inner one provides a partial schedule enumerating all data points within a given time step. Loops cannot be interchanged across bands. Finally, the individual statement types are enumerated within a sequence node. When combined across all dimensions, the schedule tree defines the execution order of the program.

The tree structure not only simplifies the schedule representation but allows 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 modifications such as band tiling or compositions 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[i][j] + B[i][j-1] + B[i+1][j];
            else
T:            B[i][j] +=   A[i-1][j] + A[i][j+1] + A[i][j] + A[i][j-1] + A[i+1][j];
```

Listing 2. A 2D 5-point stencil kernel.

3 DECLARATIVE LOOP TACTICS

With *Declarative Loop Tactics*, we introduce a framework to make modern constraint-based loop transformations as accessible as classical tree-based compiler transformations. Unlike conventional compilation approaches centered around syntax trees, polyhedral compilation techniques typically operate on some representation of a schedule disconnected from any syntactic form. Such schedule representations are algebraic objects allowing an entirely new schedule to be computed as a solution to a (sequence of) linear optimization problems. In doing so, however, polyhedral transformation acts as a black box for compiler developers, leaving them with the only option of manual optimizations if imprecise cost functions are used to solve the linear optimization problem. Unlike polyhedral compilation flows, Declarative Loop Tactics—thanks to the nature of the schedule tree representation that combines schedule and syntactic aspects in a tree shape—allows compiler developers to transform suitable programs *step-by-step*, combining tree manipulation mechanisms that are commonplace in compilers with the precise analyses of the polyhedral model. In the following, we present our framework, built around tree and access relation matchers describing computational patterns and schedule tree builders describing loop transformations. We start by informally introduce Loop Tactics using examples, while Section 6 provides a more formal description of the syntax using the extended Backus-Naur form.

3.1 Polyhedral Schedule Tree Matchers

General concepts. A schedule tree matcher enables a declarative description of a pattern in the schedule tree. It 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 children nodes, and an optional filter. In addition to all schedule tree node types, it may expect *any* node type or a non-empty *list* thereof. Tree leaves can be matched explicitly using a special node type. As an example, 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 Loop Tactics–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.

Matching Procedure. A 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

a simultaneous depth-first pre-order 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, then it is immediately reported, and the traversal stops. Multiple patterns can be recognized at once using classical prefix-tree and self-similarity approaches.

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 children nodes. This approach enables static checking of tree invariant properties (e.g., some node types only allow to have one child) or only 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 such as permutability or number of schedule dimensions. For example, identifying a permutable band node requires both structural and non-structural properties, as shown in Listing 3. 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, then the underlying subtree is guaranteed to have the structure described by the matcher.

3.2 Polyhedral Schedule Tree Builders

The imperative-style schedule tree construction interface provided by *isl* does not allow for declarative tree construction. As a consequence, 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). You can think of a builder as a description of the subtree to build. It may be transformed into a stand-alone tree if needed (i.e., if it is rooted at a domain or an extension node) or grafted at a leaf of an existing tree.

Listing 4 shows how matchers and builders can be combined together to tile a simple rectangular loop. The matcher looks for permutable band nodes anywhere in the tree and captures both the node (`scheduleTree`) and its child subtree (`body`). Permutability for a given band is tested using Loop Tactics' function `isPermutable`. The builder splits the node into two nested bands by taking the integer division and the modulo parts of the schedule via Loop Tactics–provided functions `tileSchedule` and `pointSchedule`. The child subtree is kept intact. The example can be extended to support more advanced transformations like full/partial tile separation or to limit tiling to the deepest bands without losing a significant portion of the transformation code clarity.

```
schedule_node node, body;
auto matcher = band(node, isPermutable, anyTree(body));
auto builder =
  band([&](){ return tileSchedule(node, tileSize);},
    band([&]() { return pointSchedule(node, tileSize);},
      subtree(body)));
replaceDFSPreorderOnce(scheduleTree, matcher, builder);
```

Listing 4. Declarative specification of rectangular loop tiling. Functions tileSchedule and pointSchedule divide and take modulo tile sizes, respectively.

3.3 Polyhedral Relation Matchers

General Concepts. Polyhedral relation matchers allow the caller to identify relations that have certain properties in a union of access relations [4, 38]. 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, 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 be provided by the user. An example of a pattern that yields multiple candidates is "access relation with one output dimension fixed to a (literal) 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 to be *invalid assignments*. At the same time, if a placeholder is reused within the same matcher expression, we also require that all appearances be assigned the same candidate. In any case, the candidate comparison only takes the candidate descriptions into account, not the differences between spaces of the relations. This behavior allows the matcher to connect different relations in the union with greater precision. For example, it captures the fact that two relations exist that are either both bijective or both non-bijective. To support edge cases, the user can override the definition of a valid assignment, for example, to allow the same candidate to be assigned to different placeholders.

Matching Procedure. The matching procedure is decoupled into two stages, which provides sufficient flexibility without sacrificing performance. First, the engine traverses all relations one-by-one and defines the set of suitable candidates for each placeholder. If at least one of the placeholders has no suitable candidates, then the absence of a 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. As 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 more placeholder-candidate pairs to the assignment. If the partial assignment is reported to be invalid, then further exploration is not performed. This approach can be easily transformed into a branch-and-bound approach, if the assignment needs to be optimal in some sense, or altered to change the exploration to validation ratio if the validation itself is expensive.

Programming Interface. Implementation of the matching procedure, or the matching engine, is a template definition in the 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. We provide pattern and candidate descriptions for affine expressions $\omega = k * \iota + c$, where k and c coefficients form the pattern whereas ω and ι define a candidate by matching one of the output and input dimensions, respectively. 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 operate only within 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 that progressively construct the matcher. Listing 5 illustrates how one can identify whether the same 2D array is accessed directly and with transposition.

```
auto readsAndWrites = /* get read and write accesses */;
auto _i = placeholder(), _j = placeholder();
auto _A = arrayPlaceholder();
auto matcher = allOf(access(_A, _i, _j), access(_A, _j, _i));
```

Listing 5. Access relation matcher for transposed accesses.

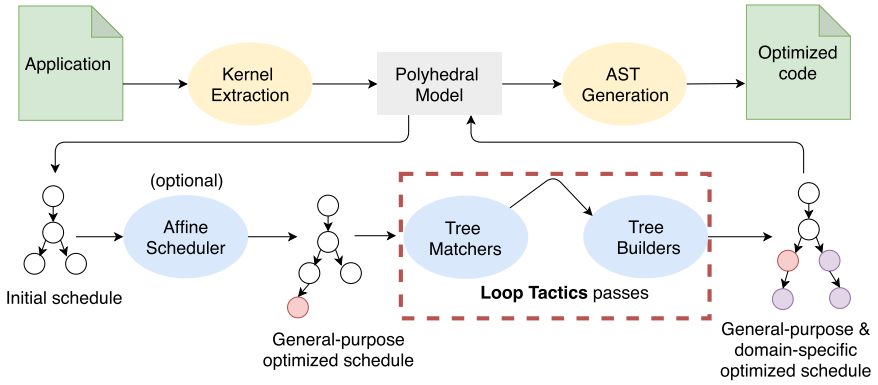

Fig. 3. Loop Tactics can be easily integrated into the architecture of state-of-the-art optimizers. It can be placed immediately after a general-purpose polyhedral optimizer (affine scheduler) or replace it.

Sometimes, it is also more convenient to consider individual output dimensions separately. For example, in the transposed access case, the caller would like to know whether, in $\{(i,j) \rightarrow A(a_1 = i, a_2 = j); (i,j) \rightarrow B(b_1 = j, b_2 = i)\}$, 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, thus projecting the access relation onto that dimension and continuing to check the properties of a relation with one-dimensional output space. Such per-dimensional behavior is particularly useful, for example, to detect the presence of temporal or spatial locality in array accesses.

4 COMPILATION FLOW

Before proceeding with the evaluation, let us describe where Loop Tactics fits into different compilation flows. The high-level view is illustrated in Figure 3. Loop Tactics can be applied instead of or in addition to an existing affine scheduler, operating on schedule trees. It relies on the caller to supply the input as schedule trees and convert the transformed trees back into the original representation. As such, it is applicable to both source-to-source and IR-to-IR flows.

4.1 IR-to-IR Flow

First, we demonstrate how Loop Tactics is embedded into the LLVM infrastructure by leveraging Polly [19]. Loop Tactics is not concerned with lowering to the LLVM IR or performing target-specific low-level optimizations such as instruction selection. Polly processes the input LLVM IR to detect the parts of the IR that are suitable for optimization, referred to as SCoPs. It then represents the IR as a schedule tree that Loop Tactics can consume. Polly may run an affine scheduler [55] that transforms the tree before sending it to the Loop Tactics framework. Pattern detection using tree matchers and transformations using tree builders occur as additional passes that modify the schedule tree. After all the optimizations have been performed, Polly translates the schedule tree into an AST and then further down into the LLVM IR. We use this flow in Sections 5.1 and 5.3, where we compile LLVM IR down to executable using LLVM tools.

4.2 Source-to-source Flow

For source-to-source compilation, we embedded Loop Tactics into the *pet* tool [52], which we use to extract SCoPs from C code and produce C or CUDA code. Similarly to Polly, *pet* relies on *isl*'s schedule trees, making it possible to (largely) reuse the integration of Loop Tactics into the

polyhedral flow. After the tree has been transformed by the affine scheduler and/or Loop Tactics, we send it back to *pet* for code generation. To obtain a final executable, we rely on general-purpose compilers such as the Intel compiler as described in Section 5.2.

5 EVALUATION

In this section, we evaluate the applicability of our framework by considering two domain-specific optimizations for GEMM-like and stencil kernels. The former is based on a recently introduced custom optimization pass in Polly that re-creates hand-tuned optimizations for GEMM-like kernels (see Section 5.1). The latter focuses on a data-layout transformation to reduce stream alignment conflicts in stencil patterns, namely dimension-lifted transposition (Section 5.2). Subsequently, we show how Loop Tactics can be used to call routines from vendor-optimized libraries transparently if such libraries are available for the target. Otherwise, it generates optimized code (see Section 5.3). For the case studies considered here, the hardware platforms are a Volta V100 GPU, an IBM Power 9 AC922 clocked at 3.8 GHz, and an Intel Core i7-7700 (Kaby Lake family) clocked at 3.6 GHz with Intel Turbo Boost up to 4.2 GHz. Intel Turbo Boost is disabled. Every single measurement or result reported in the following sections is the arithmetic mean of five runs.

5.1 Hand-tuned GEMM-like Optimization

Generalized matrix multiplication (`GEMM` BLAS kernel) is one of the important computation patterns and is the most commonly optimized kernel in history [31]. However, state-of-the-art compilers achieve only a fraction of the theoretical machine performance for a simple textbook-style implementation [15]. A recent improvement within Polly introduced a custom transformation for `GEMM`-like kernels that is controlled outside of the main affine scheduling mechanism [15]. 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` represents some operations between tensors. The matrix–matrix multiplication from Listing 1 is an example of such a contraction with E = $(\times, +)$.

Candidate loops. To qualify for the transformation, the kernel must:

- be a perfectly nested loop that satisfies the requirements of the polyhedral model;
- contain three non-empty one-dimensional loops with induction variables incremented by one;
- contain 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 accesses to tensors A, B, C and permutations of the enclosed indices, respectively. The term E is a generic expression that contains at least three reads from tensors A, B, and C;
- ensure that the interchange of I and J is valid, whereas K is interchangeable if and only if K contains only one element, or an associative operation is used to update C.

The candidate loops are found by implementing the aforementioned conditions as schedule tree and access relation matchers with a set of callback functions (Listing 6). In particular, we look for a subtree containing a three-dimensional permutable band with a single statement (leaf) featuring specific access patterns: at least three two-dimensional read accesses to different arrays, one write access, and a permutation of indices that satisfies the placeholder pattern $[i, j] \rightarrow [i, k][k, j]$.

Expressing transformations. The domain-specific optimization is derived from Reference [31] as follows: First, we rearrange the band dimensions such that `j` will be the outermost dimension, followed by `k` and `i`. Then, we apply multi-level tiling and loop interchange to create three nested loops around the macro-kernel and two additional nested loops around the micro-kernel.

```
auto matcher =
  band(and_(is3D,
            isPermutable,
            hasGemmPattern),
    leaf());

auto is3D =
  [&](schedule_node band) {
  // is the band's schedule 3-dimensional?
  return band.dim() == 3;
};

auto isPermutable =
  [&](schedule_node band) {
  // is the band permutable?
  return band.permutable() == 1;
};
```

```
auto hasGemmPattern = [&](schedule_node node)
{
  auto _i = placeholder();
  auto _j = placeholder();
  auto _k = placeholder();
  auto _A = arrayPlaceholder();
  auto _B = arrayPlaceholder();
  auto _C = arrayPlaceholder();
  auto reads = /* get read accesses */;
  auto writes = /* get write accesses */;
  auto mRead  =
    allOf(access(_C, _i, _j),
          access(_A, _i, _k),
          // for syrk use (_A, _j, _k)
          access(_B, _k, _j));
  auto mWrite = allOf(access(_C, _i, _j));
  return match(reads, mRead).size() == 1 &&
    match(writes, mWrite).size() == 1;
};
```

Listing 6. Schedule tree and access relation matcher for a tensor contraction $C \rightarrow \alpha \times C + \beta \times A \times B$. Callback functions `is3D` and `isPermutable` to test node properties are also shown. Simply changing one line in the access relation matcher callback (hasGemmPattern) is enough to capture other patterns (i.e., syrk). See Section 5.3 for more details.

Specifically, the builder for the macro-kernel reported in Listing 7 on the left performs L2-cache tiling and interchanges the newly created point loops. Tiling is applied using Loop Tactics' functions `tileSchedule` and `pointSchedule`, whereas loop interchange uses `swapDims`. After applying the first builder, we define and apply the second builder to create the micro-kernel loops by tiling the point loops such that they fit into vector registers and fully unroll the new innermost one to simplify subsequent vectorization. Loop unrolling is performed using the Loop Tactics' function `unrollAll`. Finally, we perform the packing transformation expressed as a matcher-builder pair by relying on the existing data-layout infrastructure available in the polyhedral optimizer (Listing 7, right-hand part).

```
/* obtain node, e.g., from a matcher */
schedule_node node = /*...*/;
auto macroKernel =
 band([&]() {
  return tileSchedule(
   node,/*L2 sizes*/);},
   band([&]() {
    auto sched =
     pointSchedule(node,/*L2 sizes*/);
    return swapDims(sched, dimI, dimK);
   }));
```

```
/* obtain node, e.g., from a matcher */
schedule_node node = /*...*/;
auto microKernel =
 band([&]() {
  return tileSchedule(
   node,/*Vec sizes*/);},
   band([&]() {
    return unrollAll(
     pointSchedule(node,/*Vec sizes*/));
   }));
/* Apply packing transformation. */
```

Listing 7. Macro-kernel builder (left) performs tiling and loop permutation for L2-cache locality. Micro-kernel builder (right) performs further tiling and unrolling to enable vectorization.

Evaluation. To evaluate the quality of our implementation, we hash the binaries generated by our custom Polly, with Loop Tactics embedded, with an original version containing the custom transformation (git: commit 592b2406) and compare them for strict equality. We consider

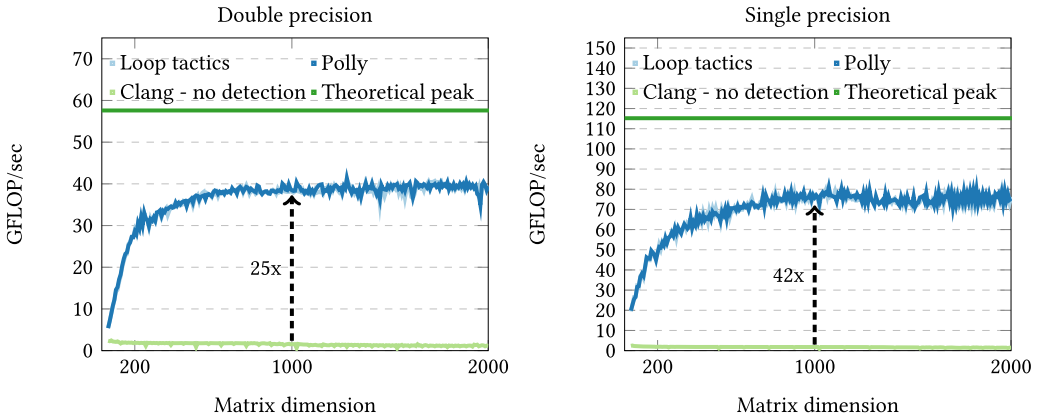

Fig. 4. Performance obtained for double and single-precision operands. The GEMM tactic produces the same binary code as a hand-tuned, Polly's custom transformation.

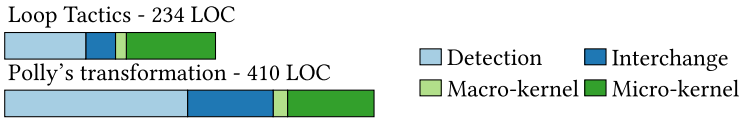

Fig. 5. The GEMM tactic reduces the compiler code footprint by almost a factor of 2 compared to the Polly's custom transformation.

a tensor contraction with E = $(\times, +)$ of the form $C \rightarrow \alpha \times C + \beta \times A \times B$, where A, B, and C are square matrices and α and β are constants set to 1 as in Reference [15]. We use the compilation strings `Clang -O3 -march=native -mllvm -polly -ffp-contract=fast -ffast-math` and `Clang -O3 -march=native -polly -mllvm -polly-enable-matchers-optimization -ffp-contract=fast -ffast-math` for the original Polly and the modified one, respectively. Identical binaries imply the same performance as shown in Figure 4, which presents the results for a single-threaded implementation using double and single-precision operands on the Intel machine. Each plot shows the achieved GFLOP/s on the y-axis versus the matrix dimensions on the x-axis. Both Loop Tactics and the Polly's custom transformation achieved non-negligible speedups compared with `Clang -O3 -march=native` (Clang - no detection). However, the combination of matchers and builders is able to reduce the compiler code footprint by a factor of almost 2 (see Figure 5). In addition, it is now possible for compiler developers to add highly customized optimizations within a short time frame of weeks instead of months as for ad hoc pattern matching [15]. Finally, as fast compilation time is important, we evaluate the overhead introduced by Loop Tactics. To do so, we compare the compilation time of our custom Polly with the original version, both compiled in Release mode. The original version takes 0.354 seconds, while Loop Tactics takes 0.362 seconds, which correspond to only a 2% increase. Loop Tactics introduces a negligible compile-time overhead.

5.2 Short-SIMD Stencil Vectorization

Stencils represent an important class of computation patterns used in many scientific domains [42, 47]. 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.

Unfortunately, polyhedral compilation based on affine scheduling is often counterproductive for stencils. Direct attempts to minimize dependence distances lead to serialization of computation or complex control flow overhead or both, making the transformed code *slower* than the original [54, 58]. Several techniques, more or less closely related to the polyhedral compilation, address specifically stencil parallelization and vectorization [18]. As these techniques are often not adapted to non-stencil cases, one must first find the stencil-like part of the program and then apply the transformation. Let us illustrate how an effective technique to remove stream-alignment conflicts through a data-layout transformation [22] can be expressed in our framework.

Candidate loops. The data-layout transformation (DLT) technique [22] applies only to vectorizable *innermost* loops with no loop-carried dependencies. In schedule tree nomenclature, we should start by finding all band nodes that have no other band nodes as children and that have the last coincidence flag set (assuming that coincidence—the possibility of parallel execution—was computed based on all dependencies). This is shown in Listing 8.

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

Listing 8. Tree matcher for DLT transformation.

If the band is permutable, then it does not matter whether the coincident flag is set for the last dimension or any other dimensions of the band members, because 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 in the original paper. The transformation validity is further restricted to statements that access either the same or contiguous elements of an array in consecutive iterations of the innermost loop. This can be represented by stating that the innermost accesses dimension (Python-style index -1) must have a stride of either 0 or 1, as shown in Listing 9.

```
auto readsAndWrites = /* get read and write accesses */;
auto stride0Accesses = match(readsAndWrites, access(dim(-1, stride(ctx, 0))));
auto stride1Accesses = match(readsAndWrites, access(dim(-1, stride(ctx, 1))));
auto allAccesses = match(readsAndWrites, access(any()));
return stride0Accesses.size() + stride1Accesses.size() == allAccesses.size();
```

Listing 9. Access relation matchers for DLT transformation.

Vector-lane conflicts. Finally, DLT is only necessary when vector-lane conflicts exist that cannot be removed through loop shifting or, 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. Although it can be used inside the relation matcher to create a list of candidates before checking whether all accesses do indeed match, it is arguably more pragmatic to perform the operations directly.

Expressing data-layout transformation. The data-layout transformation consists of placing adjacent array elements at a distance of L from each other, where L is the number of vector lanes. Figure 6 shows the original and the transformed memory layout for $L = 4$. It can be seen that previously adjacent array elements (i.e., A and B) are now spaced further apart. The data layout mapping can be expressed as an affine function

$$\left\{ (\vec{\imath}, i) \rightarrow (\vec{\alpha}, a) \mid \vec{\alpha} = \vec{\imath} \wedge a = L \cdot i - (B_U - B_L) \cdot \left\lfloor \frac{i \cdot L}{B_U - B_L + 1} \right\rfloor \right\}, \tag{1}$$

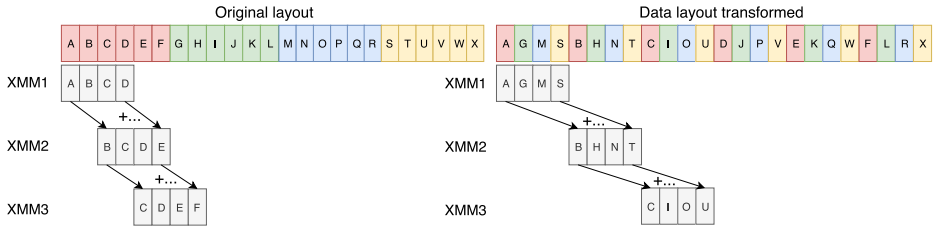

Fig. 6. Data layout transformation for a SIMD with four vector lanes. Elements mapping to vector slots in vector registers of four elements each is also shown.

where B_L and B_U are the lower and the upper inclusive bounds on the accessed elements, respectively. Figure 6 further shows how elements map to vector slots in vector registers of four elements each for a single iteration of a Jacobi-3pt stencil. Array elements in the layout transformed can be loaded into the vector slots using *aligned* loads. Therefore, the compiler does not need to issue additional *non-aligned* loads to bring interacting elements in the same vector slot before performing the arithmetic operation, as it is the case for the original layout. If this transformation had been performed on the iteration space rather than on the subscript space, then it would have corresponded to loop strip-mining followed by loop interchange and coalescing. Once such a union of affine functions is constructed, it can be used in a tree builder that injects the transformation as presented in Listing 10. First, the builder introduces the new statements for the copies "to" and "from" the transformed array through an extension node, lines 7–9 and 13–15, respectively. Below the extension node, copies "to" are scheduled before the main computation (lines 10–12), 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 handling otherwise. Although these edge cases can also be expressed declaratively as tree builders, they are not essential to our presentation, so we have omitted them for the sake of clarity.

```
1  schedule_node capturedBand = /* obtain node, e.g., from a matcher */;
2  auto dlt = /* build DLT function using equation (1) */;
3  auto from = /* build copy function from DLT layout: B[i] = B_DLT[dlt(i)] */;
4  auto to = /* build copy function to DLT layout: A_DLT[dlt(i)] = A[i] */;
5  extension([&](){return from.domain(dlt()).unite(to.domain(dlt()));},
6    sequence(
7      // introduce schedule for copy-in to DLT layout
8      filter([&](){ return to.range();},
9        band([&](){ return to.range().schedule();}))
10     // introduce DLT and original subtree computation
11     filter([&](){ return dlt().domain();},
12       subtree(capturedBand))
13     // introduce schedule for copy-out from DLT layout
14     filter([&](){ return from.range();},
15       band([&](){ return from.range().schedule();})));
```

Listing 10. Loop Tactics enables easy integration of data-layout transformations using builders to inject copy-in and copy-out around the main computation.

[2]For example, this can be ensured with full/partial tile separation, given that the tile size is equal to the number of vector lanes.

Listing 10 also lays the foundation for most data-layout or memory-related transformations that rely on *copying* the data to and from one array to another. Most such transformations follow the extension/sequence/3-filter pattern where only the properties of nodes are different. Copies "to" or "from" 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 of 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 *sequential* order, and the data required by each iteration now needs different subscripts. In particular, adjacent elements are now located at a distance of L. This can be easily expressed using our relation-rewriting mechanism (Listing 11). The pattern constitutes the stride-1 access subscript to be transformed, whereas the candidate is an affine function that defines the new subscript and array name. This affine function consists of a linear and a constant part, and the latter needs to account for the vector length chosen and the boundary cells introduced.

```
auto stride1Accesses = /* obtain strided-1 accesses from matchers */;
auto aff = /* build affine expr. for pattern */;
findAndReplace(stride1Accesses, replace(
    access(dim(-1, aff)), // pattern
    access(dim(-1, [&]() { aff.payload_.id = "A_DLT"; // candidate
                           aff.payload_.linear + L * aff.payload_.constant;
                           return aff; }))));
```

Listing 11. Rewriting access relations for DLT transformation.

Evaluation. We implement the DLT tactic in the source-to-source compilation flow by relying on ICC to vectorize our transformed code, as was done in Reference [22]. To evaluate the quality of our transformation, we apply the DLT to four variants of the Jacobi kernel—a three-point single-dimensional stencil, a five-point "star," a nine-point "box" two-dimensional stencil, and a seven-point three-dimensional stencil. While building our tactic, we consider $L = 8$, the number of vector lanes, to exploit the highest vector instruction set available on our Intel i7-7700 (AVX2). Nevertheless, L is merely a parameter and can be changed to address wider or narrower vector extensions. We compile using the Intel C compiler ICC v19.0 with the `-O3 -xHost` options. We compare the layout-transformed ICC auto-vectorized version generated by our compilation flow with a reference auto-vectorized code (without layout transformation). The original program is tiled such that all the references fit nicely in the L1 cache, as done in Reference [22]. Similar to Reference [22], we assume an outer loop with T iterations around the stencil loops—as it is common in stencil codes—such that a one-time layout transformation cost to copy from the original layout to the transformed one represents a negligible overhead. For the stencils observed, considering $T = 500$ as in Polybench 4.1, the overhead introduced is less than 3% of the total execution time, hence, we donot account for it. The speedups for single-precision operands are shown in Figure 7. The auto-vectorized DLT code improves upon the reference code in all cases. Performance gains are due to the reduction in unaligned loads in the innermost loop. For the 3-pt stencil, the speedup is minimal. Even though the number of unaligned loads is slightly reduced for 1D stencils, the overhead of a small number of additional loads is well hidden by modern CPU architectures. However, the 5-pt, the 9-pt 2D, and the 7-pt 3D stencils achieve speedups of 50%, 23%, and 11%, respectively, due to a significant reduction in unaligned loads and reduced register pressure. The assembly snippet in Listing 12 shows two innermost loop computations for a 3-pt stencil. Computation with the DLT array (A_DLT) shows fewer memory loads compared with the original computation.

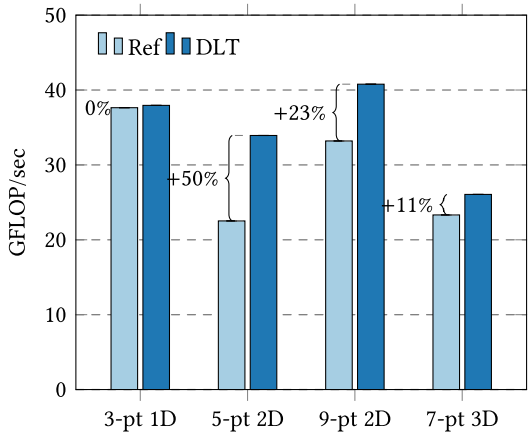

Fig. 7. Speedup for Jacobi kernels. Ref is the auto-vectorized version without data layout transformation. DLT is the auto-vectorized version with layout transformation.

```
/* Original computation */
vmovups   A(,%rdx,4), %ymm7
vmovups   32+A(,%rdx,4), %ymm11
vaddps    4+A(,%rdx,4), %ymm7, %ymm8
vaddps    36+A(,%rdx,4), %ymm11, %ymm12
vaddps    8+A(,%rdx,4), %ymm8, %ymm9
vaddps    40+A(,%rdx,4), %ymm12, %ymm13
vmulps    %ymm9, %ymm5, %ymm10
vmulps    %ymm13, %ymm5, %ymm14
vmovups   %ymm10, 4+B(,%rdx,4)
vmovups   %ymm14, 36+B(,%rdx,4)
/* More code */
```

```
/* After data-layout transformation */
vmovups   32+A_DLT(,%rdx,4), %ymm7
vmovups   64+A_DLT(,%rdx,4), %ymm11
vmovups   96+A_DLT(,%rdx,4), %ymm15
vaddps    A_DLT(,%rdx,4), %ymm7, %ymm4
vaddps    %ymm11, %ymm7, %ymm8
vaddps    %ymm11, %ymm4, %ymm5
vaddps    %ymm15, %ymm8, %ymm9
vmulps    %ymm5, %ymm3, %ymm6
vmulps    %ymm9, %ymm3, %ymm10
vmovups   %ymm6, 4+B_DLT(,%rdx,4)
vmovups   %ymm10, 36+B_DLT(,%rdx,4)
/* More code */
```

Listing 12. Comparison between the original computation and the one after data-layout transformation. DLT effectively reduces loads from A reference.

5.3 Transparent BLAS Optimization

Matching Libraries. Until now, we have demonstrated how Loop Tactics enables domain-specific optimizations by specifying custom transformations for a given computational pattern. However, Loop Tactics matching mechanism can also be used to recognize code fragments that correspond to common high-performance library calls. Such vendor-optimized libraries are often necessary to achieve peak performance on accelerators [32]. Thus, Loop Tactics matching mechanism opens up the possibility to invoke automatically and transparently for the application routines from vendor-optimized libraries. Domain-specific compilers such as Halide [39], TVM [11], and Tensor-flow [1] already lower high-level constructs to vendor-optimized routines; however, they focus on specific domains. Loop Tactics aims at enabling such optimization in general-purpose compilers. We illustrate how this can be achieved in a *portable* way by Loop Tactics targeting both CPU and GPU BLAS libraries. BLAS parameters (i.e., values of α and β) are automatically collected by Loop Tactics. For GPUs, Loop Tactics handles the data transfers as demonstrated in Listing 10. We implemented matchers for level-3 BLAS `GEMM`, `SYMM`, `SYRK`, `SYR2K`, `TRMM`, and a level-2 BLAS `GEMV`, and applied them to all kernels in Polybench 4.1 using the `LARGE` input set. The BLAS kernels

are detected by *composing* together elementary matchers and elementary matchers that are *reused* not only across targets but also across kernels. For example, the gemm pattern is expressed as two elementary matchers as shown in Listing 13 where gemmInit detects an initialization statement while gemmCore detects the GEMM core statement. The exact same matchers are reused for other two kernels (2mm and 3mm). Moreover, just changing the access matchers callbacks (hasGemmPattern) is enough to capture other patterns with the same tree structure (i.e., syrk). Similarly, the same matcher for level-2 BLAS GEMV is reused for gemver, gesummv, bicg, and mvt. Each matcher is assigned a priority that implements its firing policy. Matchers for level-3 BLAS are assigned the same priority, which is higher than those for level-2. Among matchers with the same priority, the firing policy is decided by the user.

```
auto isGemm/*isSyrk*/Like = [&](schedule_node node) {
  auto gemm/*syrk*/Core =
    band(and_(is3D, isPermutable, hasGemm/*hasSyrk*/Pattern), leaf());
  auto gemm/*syrk*/Init =
    band(and_(is2D, hasDescendant(gemm/*syrk*/Core), has2DInitPattern),
      anyTree());
  return isMatching(node, gemm/*syrk*/Core) || isMatching(node, gemm/*syrk*/Init);
};
auto is2DInit(auto _iCore, auto _jCore, auto _CCore, auto reads, auto writes,
  auto node) {
  auto has2DInitPattern = [&](schedule_node node) {
    auto _i = placeholder(), _j = placeholder();
    auto _C = arrayPlaceholder();
    auto mRead  = allOf(access(_C, _i, _j)), mWrite = allOf(access(_C, _i, _j));
    return match(reads, mRead).size() == 1 && match(writes, mWrite).size() == 1
    && _iCore == _i && _jCore == _j && _CCore == _C;
  };
}(node);
```

Listing 13. Matchers can be reused and composed together. The GEMM pattern can be expressed as a composition of two elementary matchers (gemmCore and gemmInit). Changing the access relation matcher callback hasGemmPattern with hasSyrkPattern is enough to reuse the same matchers to capture other patterns with the same tree structure (i.e., syrk). hasSyrkPattern can be easily derived as shown in Listing 6.

Evaluation. Figure 8 compares the achieved GFLOP/s for double-precision operands using Loop Tactics, an original Polly version (git commit 592b2406) and the Pluto source-to-source compiler (git commit f62d61b8). We compare with the out-of-the-box optimizations available in Polly and Pluto. Besides, for BLAS kernels (gemm to mvt), where Loop Tactics invokes highly optimized routines, we use the Pluto compiler to explore different fusion heuristics and tile sizes. Specifically, for each selected benchmark, we generate several variants for each tile size and fusion heuristic available in Pluto (maximum fusion, no fusion, and smart fusion). The set of tile sizes chosen are powers of two and non-powers of two. We choose an interval from 1 up to one-fourth of the problem size, including non-powers of two sizes for every two powers of two tile sizes [27]. For each benchmark, the explored space consists of more than 3K variants; we report the best founded (Pluto best). The optimized code generated by Pluto is lowered to binary using native compilers. We use XLC 16.1 for the Power architecture and ICC v19.0 for the Intel one. The compilation strings are `xlc -O4 -qhot -qarch=pwr9 -qtune=auto -qsimd=auto` and `icc -O3 -march=native` for XLC and ICC, respectively.

Results for the Power 9 are shown on top; results for the Intel i7-7700 CPU are shown at the bottom. For Power 9, Loop Tactics' performance exceeds or is comparable to that of Polly and Pluto.

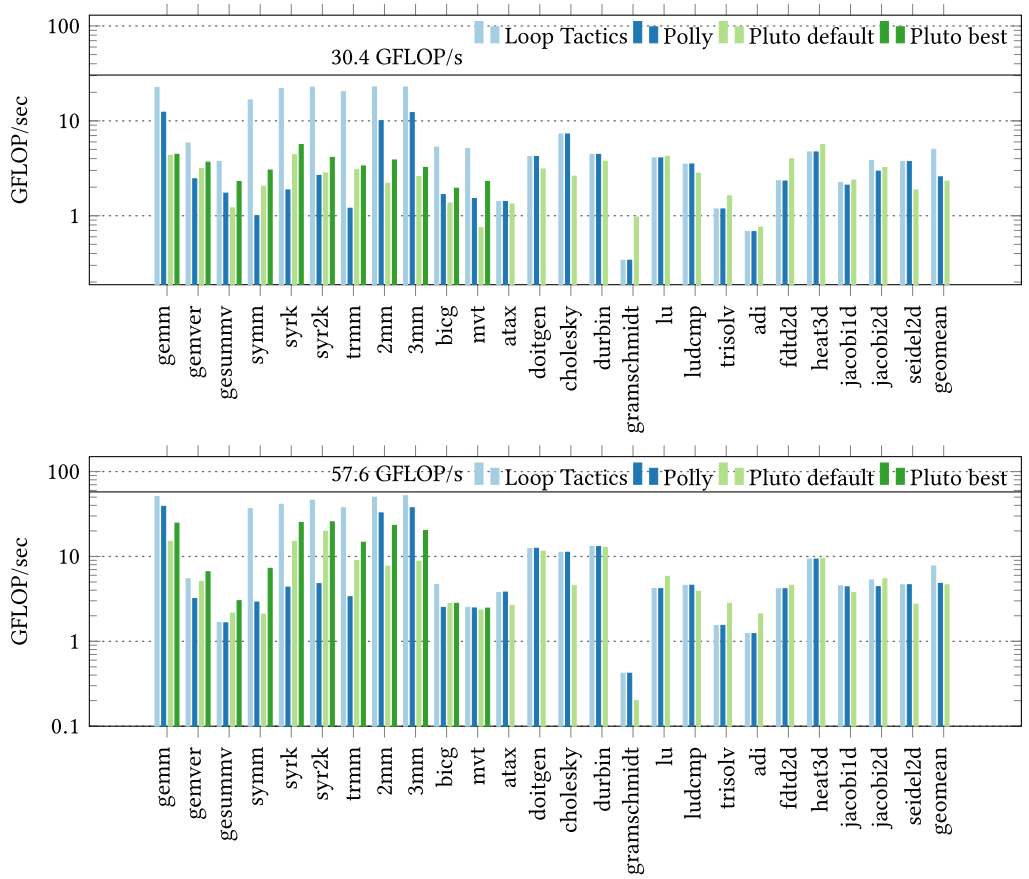

Fig. 8. Loop Tactics outperforms Polly and is comparable with Pluto for double-precision operands (average of 5 runs) by matching library calls: (top) calling IBM ESSL on Power 9 with a theoretical peak of 30.4 GFLOP/s; (bottom) calling MKL on Intel i7-7700 CPU with a theoretical peak of 57.6 GFLOP/s.

The largest speedups are observed for `trmm` (16×) when comparing with Polly, while for `symm` (8×) when comparing with the default Pluto optimization (Pluto default). The lowest speedups are achieved for `gemm` (2×) and `gemver` (1.9×) when comparing with Polly and Pluto default, respectively. If no BLAS kernels are detected (`atax` to `seidel2d`), then Loop Tactics generates code that is *on par* with Polly. An exception is made for Jacobi stencils, where Loop Tactics is faster due to DLT optimization. On-par performance with Polly has been achieved by relying on the *isl* scheduler, as Polly does, and re-implementing Polly imperative tiling optimization, which is the default Polly optimization, in our declarative style approach based on matchers and builders. An example of how to implement tiling using Loop Tactics has been presented in Listing 4. When we compare Loop Tactics' results with Pluto, minor performance regression can be seen in the stencil and solver benchmarks. Exception made, once more, for the Jacobi-2d stencil, thanks to the DLT transformation. For Intel i7-7700, speedups are achieved for all except two BLAS kernels in the benchmark. The largest speedups are for `symm` for both Polly and Pluto default: 14× and 17×, respectively. The lowers are for `gemm` (1.30×) when comparing with Polly and for `gemver` (0.7×) when comparing with Pluto default. The only benchmark where Loop Tactics performs worse than Pluto and equal to Polly even by calling a BLAS function is `gesummv`. This loss in optimization

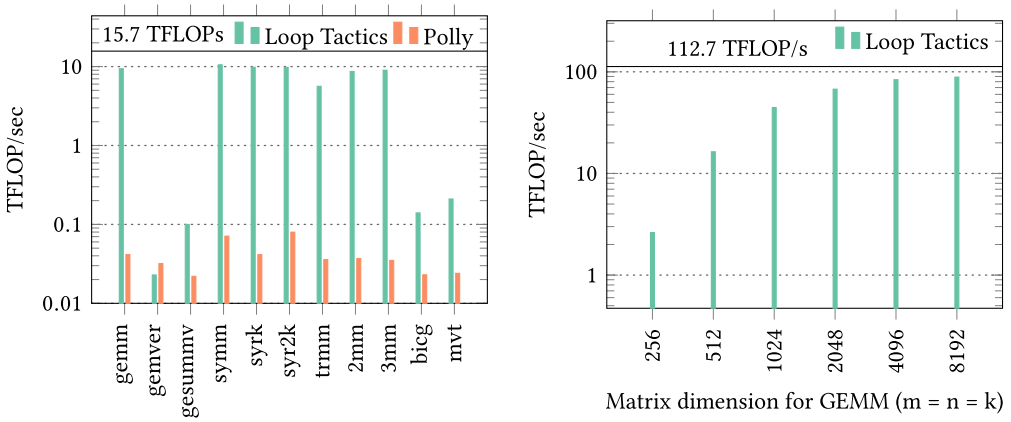

Fig. 9. Performance achieved for single-precision operands, average of five runs; (left) Loop Tactics produces speedups over original Polly by calling CuBLAS on a Volta V100 GPU with a single-precision theoretical peak of 15.7 TFLOP/s; (right) if lower-precision is acceptable during computation (i.e., 16-bit) Loop Tactics enables the user to transparently exploit tensor cores for GEMM.

opportunity is the result of additional overhead introduced by Loop Tactics to acquire the library handle dynamically. Such overhead is constant and around 1.5 ms. It can be avoided by static linking[3] and if so for the `gesummv` kernel, we expect to obtain the same performance as measured from the MKL library, 2.73 GFLOP/s instead of 1.65 GFLOP/s. The other benchmark where Loop Tactics produces worse performance than the tuned version generated with Pluto is `gemver`. The reason is that only half of the statements can be mapped to BLAS calls; for the remaining statements, Loop Tactics performs a simple default tiling strategy with squared tiles of 32 elements each. However, Pluto performs aggressive loop fusion, which helps in reducing the control flow overhead and tiling the fused loop, achieving better performance.

Finally, we evaluate the impact of Loop Tactics at compilation time. Here, we compare against an original Polly version. Both Loop Tactics and Polly are compiled in Release mode. For the 25 kernels considered, the original version takes 14.88 seconds, while Loop Tactics 12.00 seconds, which is a 20% reduction. The reduction is because at compilation time, only pattern detection is performed by Loop Tactics, while optimizations happen at run time by linking with BLAS libraries.

Figure 9 (on the left) demonstrates how Loop Tactics improves performance over Polly by using *the same matchers* as CPU to automatically detect and call cuBLAS on an Nvidia Volta V100 GPU. We only illustrate the benchmarks where Loop Tactics detect BLAS kernels; as in the other case, we fall back to Polly. The largest speedup is achieved for `symm` (148×). A minor performance regression is obtained by calling the level-2 BLAS `GEMV` in `gemver`. Figure 9 (on the right) further illustrates the performance achieved using GPU tensor cores, which are usable through vendor libraries and currently inaccessible to Polly. For matrices of size 8,192, we achieve 88.8 TFLOP/s, 79% of the theoretical peak of 112.7 TFLOP/s. Manually using tensor cores requires the programmer to guarantee the following conditions: (1) only `GEMM` pattern supports tensor cores execution; (2) the matrix dimensions should be multiple of eight; (3) math mode in cuBLAS should be set to `CUBLAS_TENSOR_OP_MATH`; (4) input data type should fit in half-precision floating point. Loop Tactics hides all this complexity and requires the user to only check (4), and, whenever possible,

[3]We opt for dynamic linking, usually the common way to interact with BLAS libraries, as it ensures binary portability [2]. Nevertheless, there is no restriction in Loop Tactics that will prevent us from switching to a static linking in the future, if needed.

⟨*matcher*⟩ ::= ⟨*matcher-type*⟩(⟨*matcher-list*⟩)
| ⟨*matcher-type*⟩([⟨*capture*⟩],[⟨*callback*⟩],⟨*matcher-list*⟩)
| leaf() | anyTree()
⟨*node-type*⟩ ::= 'band' | 'context' | 'domain' | 'expansion'
| 'extension' | 'filter' | 'guard' | 'leaf' | 'mark' | 'set'
| 'sequence'
⟨*matcher-type*⟩ ::= 'anyTree' | ⟨*node-type*⟩
⟨*matcher-list*⟩ ::= ⟨*matcher*⟩[{',' ⟨*matcher*⟩}]
⟨*capture*⟩ ::= /*ref to schedule_node var*/
⟨*callback*⟩ ::= /*callable obj bool(schedule_node var)*/
⟨*builder*⟩ ::= ⟨*node-type*⟩(⟨*param*⟩,⟨*builder-list*⟩)
| ⟨*node-type*⟩(⟨*callback*⟩,⟨*builder-list*⟩)
| 'subtree'(capture)
⟨*builder-list*⟩ ::= ⟨*builder*⟩ {',' ⟨*builder*⟩}
⟨*param*⟩ ::= /* see Table 1 */
⟨*bool*⟩ ::= isMatching(node, ⟨*matcher*⟩)
⟨*node*⟩ ::= replaceDFSPreorder(node, ⟨*matcher*⟩, ⟨*builder*⟩)
⟨*node*⟩ ::= replaceDFSPostorder(node, ⟨*matcher*⟩, ⟨*builder*⟩)
⟨*node*⟩ ::= replaceBFS(node, ⟨*matcher*⟩, ⟨*builder*⟩)

⟨*isl::union_map*⟩ ::= findAndReplace(union_map,
⟨*replacement*⟩)
⟨*matches*⟩ ::= match(union_map, ⟨*access-matcher*⟩)
⟨*access-matcher*⟩ ::= allOf(⟨*access*⟩ {',' ⟨*access*⟩}
| ⟨*access-matcher*⟩ ::= anyOf(⟨*access*⟩ {',' ⟨*access*⟩}
⟨*access*⟩ ::= access(⟨*args*⟩)
⟨*args*⟩ ::= any()
| (⟨*array-placeholder*⟩,⟨*positioned-arg-list*⟩)
| (⟨*positioned-arg-list*⟩)
⟨*positioned-arg-list*⟩ ::= dim-arg
| 'dim'('integer', ⟨*dim-arg*⟩)
⟨*dim-arg*⟩ ::= placeholder | stride
⟨*array-placeholder*⟩ ::= /*res of calling arrayPlaceholder()*/
⟨*placeholder*⟩ ::= /*res of calling placeholder()*/
⟨*stride*⟩ := stride('integer')
⟨*replacement*⟩ := replace(⟨*access*⟩, ⟨*access*⟩)

Fig. 10. Extended Backus–Naur form for matchers and builders on the left and extended Backus–Naur form for access matchers on the right.

Table 1.

node type	param	node type	param
context, guard	set	expansion, extension	union_map
filter, domain	union_set	set, sequence	void
mark	id	band	multi_union_pw_aff

Loop Tactics will use tensor cores. Specifically, condition (1) is ensured by the matcher (Listing 13) and condition (3) is implemented by Loop Tactics' runtime library. If (2) does not hold, then Loop Tactics will automatically fall back to not using tensor cores.

6 LOOP TACTICS SYNTAX

The syntax using the extended Backus-Naur form for matchers and builders is shown in Figure 10 and Table 1. Matchers and builders are designed to replicate the node type-based structure of the schedule tree in a declarative way. Matchers take a callback and/or a reference to an schedule_node object as optional leading arguments. A callback allows fine-grained control over the matching procedure, whereas a reference will point to the matched node. Builders take either the node kind-specific parameter as their leading argument or a callback that can produce them. Both matchers and builders take a call to another matcher or builder as remaining arguments, respectively. An exception is made for "leaf" and "anyTree." Matchers and builders walk the schedule tree using replaceDFSPreorder | replaceDFSPostorder | replaceBFS. The traversal functions take three arguments: a node from where the traversal should start, a matcher to verify whether the subtree rooted at the current node matches, and a builder that rebuilds the subtree in case of a match. We allow multiple matches. Matchers can use "AnyTree" to capture any subtree, whereas builders can use "subtree" to rebuild any subtree. The syntax for defining an access relation matcher is a function call that takes an optional array placeholder, followed by a list of dimension placeholders. The position of the latter is specified by a "dim" call or is inferred from their position in the argument

list (see Figure 10). A union of relations can be matched against a list of matchers using "match" and "allOf" or "anyOf" as combiners. "findAndReplace" finds and replaces a given pattern in a union of relations. The candidate pattern and the replacement are built via the "replace" function call.

7 RELATED WORK

Optimizers with automatic scheduling. Years of research in automatic compilation led to sophisticated general-purpose optimizers such as Pluto [8], Polly [21], and GRAPHITE [36]. Despite being fully automatic, therefore increasing productivity, general-purpose optimizers donot always obtain the best performance for commonly recurrent kernels. Suboptimal performance is, in most cases, the result of a generic one-size-fits-all optimization strategy. Loop tactics enables such optimizers to detect computational patterns easily and hook a custom optimization for each of them. Custom optimizations can be expressed as composable transformations or call to optimized libraries. As most of these optimizers already power production compilers such as GCC and LLVM, we also expect general-purpose compilers will benefit from Loop Tactics.

Optimizers with scheduling languages. Kelly et al. proposed the first framework for high-level transformations based on the polyhedral model [23]. Girbal et al. proposed the URUK framework to apply loop transformations for cache hierarchy and parallelism using unimodular schedules [17]. Yuki et al. developed AlphaZ, a framework to express programs as a set of equations based on the Alpha language and manipulated it using script-driven transformations [57]. The implementation also supports memory (re)-allocation and explicitly represents reductions. Similarly, Yi et al. presented POET, an interpreted program transformation language designed for applying and exploring complex code transformations in different programming languages [56]. Donadio et al. introduced Xlanguage, an embedded DSL based on C/C++ pragmas that allows users to generate multi-version programs by specifying the type of transformations to apply as well as the transformation parameters [13]. Bagnères et al. provided feedback from a polyhedral compiler by expressing it as a sequence of loop transformation directives [4]. Their input language, Clay, and the related Chlore algorithm allow users to examine, refine, or freeze sequences of loop transformation directives. Chen et al. introduced CHiLL, a high-level transformation and parallelization framework that uses a model-driven empirical optimization engine to generate and evaluate different code variants [10, 40]. Khan et al. built a meta-optimizer on top of the CHiLL compiler to automatically generate transformation recipes and perform extensive autotuning [26]. Teixeira et al. proposed a language to express the collection of transformations that can be applied to user-defined code regions [48]. Baghdadi et al. proposed Tiramisu a polyhedral compiler that introduces novel commands to explicitly target distributed systems. More recently, Kruse et al. submitted a proposal to improve pragma-based transformations in Clang [29]. All these tools expose some scheduling-based language to specify program transformations applicable to loops. But the scheduling language can be seen as *imperative,* as it requires the user or the autotuner to specify which loops are targeted using external tags or language-level annotations. Our approach, however, is based on a declarative language. Instead of binding the transformation to a specific loop, we *declare* how a compatible schedule tree should look like and express the transformation in terms of matched nodes and relations between them. Another difference with respect to fully automatic compilers such as Halide [39] and TVM [11] is that Loop Tactics aims at bringing domain-specific optimizations *directly* in general-purpose compilers, while Halide and TVM born as domain-specific. Moreover, both Halide and TVM use intervals to represent iteration spaces. Thus, non-rectangular iterations spaces cannot be naturally represented, which is not the case for Loop Tactics.

Pattern matching. Algorithm recognition is a well-known problem in computer science and was an ongoing and extensive research topic during the 1990s [12, 24, 25, 33]. Although the approach

details vary considerably for their application domain, the underlying goal always falls under the umbrella of program optimization. Previous works can be broadly classified accordingly to the level of abstraction used to match a set of predefined constructs: text, syntactic, and semantic level [28]. Text-level tools operate on the source code of the program directly. Syntactic ones work at the abstract syntax tree level (AST), while semantic ones go one step further by annotating the AST with data and control flow information.

Text-level and syntactic-level methods. Pottenger et al. implemented a pattern-recognition tool on top of the Polaris compiler [5] to recognize and parallelize reduction operations [37]. Their method directly matches code statements with a set of predefined patterns. It then uses a data-dependence analysis to prove that all loop iterations refer to different elements of the reduced array. Sarvestani et al. introduced an idiom-detection tool for kernel recognition in DSP applications based on the Cetus compiler [30, 41]. Each detected kernel is replaced with a highly optimized parallel version. Patterns are described in XML format. Kessler et al. developed PARAMAT to detect and replace code segments at the AST-level with calls to parallel library routines [24]. For FORTRAN code, Di Martino et al. designed the PAP recognizer with the main focus of computer-aided program analysis and parallelization [12]. The tools provide a graphical user interface that shows each detected pattern and its corresponding source-line location. Bravenboer et al. proposed Stratego, a language for program transformations using rewriting rules in the context of generative programming (i.e., from code refactoring to code optimization) [9]. More recently, Clang AST matchers have been developed as a domain-specific language for matching predicates on Clang AST. Contrary to our work, however, these approaches try to match code directly at the statement or AST level. Hence, variations in the programming style have a huge impact on the effectiveness of such tools. However, Loop Tactics is embedded into Polly, which runs in the latest stage of the LLVM pipeline after inlining, constant expression propagation, and dead code elimination. Consequently, by position, it is less strongly affected by programming style. Furthermore, Polly protects against the linearization of multi-dimensional arrays, because it can recover the 2D structure [20]. Finally, most of the tools mentioned above provide only pattern recognition and rely on directives or library calls to improve performance. However, Loop Tactics provides builders that can be used to specify a specific transformation "recipe" for each detected pattern.

Semantic-level methods. The XARK compiler—perhaps the most representative example among semantic-level methods—provides code recognition based on the analysis of Strongly Connected Components (SCCs) [3]. SCCs are analyzed by looking at use-def chains in the Gated Single Assignment (GSA) form, an extension of the SSA form. The detection of a given kernel is carried out in two phases. In the first, the use-def chains are used to recognize basic statements such as conditional linear induction variables and array assignments. In the second phase, the use-def chains between statements are analyzed to identify more complex computation kernels (i.e., consecutively written array and masked operations). Compared with Loop Tactics the XARK compiler—or an SSA-based tool in general—has the advantage of being able to detect irregular access patterns such as linked-list traversal. Detection of such patterns falls beyond the capabilities of the polyhedral model and hence of Loop Tactics. But SSA-based tools rely on the use of use-def chain on scalar registers to reconstruct the access patterns, fundamentally limiting the complexity of the access patterns that such tools can recognize. However, Loop Tactics uses access relations [4, 38] that *precisely* and *explicitly* encode the relation between the iteration space and the array space. Therefore, Loop Tactics can easily and concisely express complex access patterns on multi-dimensional arrays and inspect all sorts of possible index permutations (i.e., transposed accesses as shown in Listing 5). Other works at the semantic-level focus on specific families of patterns that can be covered by Loop Tactics. Suganuma et al. focused on reduction detection and parallelization [43]. Their

algorithm inspects strongly connected components with cyclic dependences typical of reduction operations. It then tries to recognize if the statement is actually a reduction by finding a reduction operator and a reduction function. Gerlek et al. developed a technique to detect classes of induction variables based on the use-def chain available in the SSA form [16]. Pinter et al. proposed a method for extracting parallelizable idioms from scientific applications [35]. Their technique uses the computational graph, which encodes the flow of data and the dependencies among values in the program. Recognition of computational idioms is carried out by matching specific patterns. Matchers and replacement rules are described using the graph-specific grammar, similar to what Loop Tactics does at the schedule tree level. However, their technique requires some preprocessing steps such as loop distribution and unrolling to expose the kernels in the computation graph. Such preprocessing steps are not needed in Loop Tactics. Sujeeth et al. in their DELITE framework use rewriting rules to perform domain-specific optimizations [44]. Despite our approach being similar to what they propose, our end goal is not. With Loop Tactics, we aim at reducing the need of developing DSL compilers by bringing domain-specific optimizations *directly* in general-purpose compilers. Delite, however, aims at lowering the effort of developing DSLs.

8 CONCLUSION

This article presents Declarative Loop Tactics (Loop Tactics for short) and its implementation as a framework to express computational patterns and transformations on schedule trees, an internal representation of a polyhedral compiler. We show how Loop Tactics allows one to express domain-specific optimizations *directly* in general-purpose compilers as a set of composable transformations or as calls to optimized libraries. As a result, (1) we reduce the need to design domain-specific compilers, (2) we allow the transparent use of vendor-optimized libraries, and (3) unlock the effective optimization of multi-domain applications. We expect our approach to extend the range of optimizations significantly compared to what current general-purpose compilers can achieve by allowing developers to plug in highly customized optimizations for a given computational pattern.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI'16)*. 265–283.

[2] Varun Agrawal, Abhiroop Dabral, Tapti Palit, Yongming Shen, and Michael Ferdman. 2015. Architectural support for dynamic linking. *ACM SIGARCH Comput. Archit. News*, Vol. 43. ACM, 691–702.

[3] Manuel Arenaz, Juan Touriño, and Ramon Doallo. 2008. XARK: An extensible framework for automatic recognition of computational kernels. *ACM Trans. Prog. Lang. Syst.* 30, 6 (2008), 32.

[4] Lénaïc Bagnères, Oleksandr Zinenko, Stéphane Huot, and Cédric Bastoul. 2016. Opening polyhedral compiler's black box. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'16)*. ACM, New York, NY, 128–138. DOI: https://doi.org/10.1145/2854038.2854048

[5] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. 1995. Polaris: Improving the effectiveness of parallelizing compilers. In *Languages and Compilers for Parallel Computing*, Keshav Pingali, Utpal Banerjee, David Gelernter, Alex Nicolau, and David Padua (Eds.). Springer Berlin, 141–154.

[6] U. Bondhugula, S. Dash, O. Gunluk, and L. Renganarayanan. 2010. A model for fusion and code motion in an automatic parallelizing compiler. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. 343–352.

[7] Uday Bondhugula, Albert Hartono, J. Ramanujam, and P. Sadayappan. 2008. A practical and fully automatic polyhedral program optimization system. In *Proceedings of the ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI'08)*.

[8] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. 2008. A practical automatic polyhedral parallelizer and locality optimizer. *ACM SIGPLAN Not.* 43, 6 (2008), 101–113.

[9] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. 2008. Stratego/XT 0.17. A language and toolset for program transformation. *Sci. Comput. Prog.* 72, 1 (2008), 52–70. DOI: https://doi.org/10.1016/j.scico.2007.11.003

[10] Chun Chen, Jacqueline Chame, and Mary Hall. 2008. *CHiLL: A Framework for Composing High-level Loop Transformations*. Technical Report. Citeseer. USC Computer Science.

[11] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated end-to-end optimizing compiler for deep learning. In *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*. 578–594.

[12] B. Di Martino and G. Iannello. 1996. PAP recognizer: A tool for automatic recognition of parallelizable patterns. In *Proceedings of the 4th Workshop on Program Comprehension (WPC'96)* 164–174. DOI: https://doi.org/10.1109/WPC.1996.501131

[13] Sebastien Donadio, James Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David Padua, and Keshav Pingali. 2006. A language for the compact representation of multiple program versions. In *Languages and Compilers for Parallel Computing*, Eduard Ayguadé, Gerald Baumgartner, J. Ramanujam, and P. Sadayappan (Eds.). Springer Berlin, 136–151.

[14] Paul Feautrier and Christian Lengauer. 2011. *Polyhedron Model*. Springer, Boston, MA, 1581–1592. DOI: https://doi.org/10.1007/978-0-387-09766-4_502

[15] Roman Gareev, Tobias Grosser, and Michael Kruse. 2018. High-performance generalized tensor operations: A compiler-oriented approach. *ACM Trans. Archit. Code Optim.* 15, 3, Article 34 (Sept. 2018), 27 pages. DOI: https://doi.org/10.1145/3235029

[16] Michael P. Gerlek, Eric Stoltz, and Michael Wolfe. 1995. Beyond induction variables: Detecting and classifying sequences using a demand-driven SSA form. *ACM Trans. Prog. Lang. Syst.* 17, 1 (Jan. 1995), 85–122. DOI: https://doi.org/10.1145/200994.201003

[17] 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. *Int. J. Parallel Prog.* 34, 3 (June 2006), 261–317. DOI: https://doi.org/10.1007/s10766-006-0012-3

[18] Tobias Grosser, Albert Cohen, Justin Holewinski, P. Sadayappan, and Sven Verdoolaege. 2014. Hybrid hexagonal/classical tiling for GPUs. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'14)*. ACM, New York, NY, Article 66, 10 pages. DOI: https://doi.org/10.1145/2544137.2544160

[19] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. 2012. Polly-Performing polyhedral optimizations on a low-level intermediate representation. *Parallel Proc. Lett.* 22, 4 (2012), 1250010.

[20] Tobias Grosser, J. Ramanujam, Louis-Noel Pouchet, P. Sadayappan, and Sebastian Pop. 2015. Optimistic delinearization of parametrically sized arrays. In *Proceedings of the 29th ACM International Conference on Supercomputing (ICS'15)*. ACM, New York, NY, 351–360. DOI: https://doi.org/10.1145/2751205.2751248

[21] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. 2011. Polly-polyhedral optimization in LLVM. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*, Vol. 2011. 1.

[22] Tom Henretty, Kevin Stock, Louis-Noël Pouchet, Franz Franchetti, J. Ramanujam, and P. Sadayappan. 2011. 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)*. Springer-Verlag, 225–245. http://dl.acm.org/citation.cfm?id=1987237.1987255.

[23] Wayne Kelly and William Pugh. 1998. *A Framework for Unifying Reordering Transformations*. Technical Report. University of Maryland.

[24] Christoph W. Kessler. 1996. Pattern-driven automatic parallelization. *Sci. Prog.* 5, 3 (Aug. 1996), 251–274. DOI: https://doi.org/10.1155/1996/406379

[25] C. W. Kessler and C. H. Smith. 1999. The SPARAMAT approach to automatic comprehension of sparse matrix computations. In *Proceedings of the 7th International Workshop on Program Comprehension*. 200–207. DOI: https://doi.org/10.1109/WPC.1999.777759

[26] Malik Khan, Protonu Basu, Gabe Rudy, Mary Hall, Chun Chen, and Jacqueline Chame. 2013. A script-based autotuning compiler system to generate high-performance CUDA code. *ACM Trans. Archit. Code Optim.* 9, 4, Article 31 (Jan. 2013), 25 pages. DOI: https://doi.org/10.1145/2400682.2400690

[27] Martin Kong and Louis-Noël Pouchet. 2019. Model-driven transformations for multi-and many-core CPUs. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 469–484.

[28] W. Kozaczynski, J. Ning, and A. Engberts. 1992. Program concept recognition and transformation. *IEEE Trans. Softw. Eng.* 18, 12 (Dec. 1992), 1065–1075. DOI: https://doi.org/10.1109/32.184761

[29] Michael Kruse and Hal Finkel. 2018. A proposal for loop-transformation pragmas. In *Evolving OpenMP for Evolving Architectures*, Bronis R. de Supinski, Pedro Valero-Lara, Xavier Martorell, Sergi Mateo Bellido, and Jesus Labarta (Eds.). Springer International Publishing, Cham, 37–52.

[30] Sang-Ik Lee, Troy A. Johnson, and Rudolf Eigenmann. 2004. Cetus—An extensible compiler infrastructure for source-to-source transformation. In *Languages and Compilers for Parallel Computing*, Lawrence Rauchwerger (Ed.). Springer Berlin, 539–553.

[31] Tze Meng Low, Francisco D. Igual, Tyler M. Smith, and Enrique S. Quintana-Orti. 2016. Analytical modeling is enough for high-performance BLIS. *ACM Trans. Math. Softw.* 43, 2, Article 12 (Aug. 2016), 18 pages. DOI: https://doi.org/10.1145/2925987

[32] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter. 2018. NVIDIA tensor core programmability, performance precision. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW'18)*. 522–531. DOI: https://doi.org/10.1109/IPDPSW.2018.00091

[33] Robert Metzger and Zhaofang Wen. 2000. *Automatic Algorithm Recognition and Replacement: A New Approach to Program Optimization*. The MIT Press.

[34] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. 2015. PolyMage: Automatic optimization for image processing pipelines. *SIGARCH Comput. Archit. News* 43, 1 (Mar. 2015), 429–443. DOI: https://doi.org/10.1145/2786763.2694364

[35] Shlomit S. Pinter and Ron Y. Pinter. 1994. Program optimization and parallelization using idioms. *ACM Trans. Prog. Lang. Syst.* 16, 3 (May 1994), 305–327. DOI: https://doi.org/10.1145/177492.177494

[36] Sebastian Pop, Albert Cohen, Cédric Bastoul, Sylvain Girbal, Georges-André Silber, and Nicolas Vasilache. 2006. GRAPHITE: Polyhedral analyses and optimizations for GCC. In *Proceedings of the GCC Developers Summit*. Citeseer, 2006.

[37] Bill Pottenger and Rudolf Eigenmann. 1995. Idiom recognition in the Polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing (ICS'95)*. ACM, New York, NY, 444–448. DOI: https://doi.org/10.1145/224538.224655

[38] William Pugh and David Wonnacott. 1994. Static analysis of upper and lower bounds on dependences and parallelism. *ACM Trans. Prog. Lang. Syst.* 16, 4 (1994), 1248–1278.

[39] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *ACM SIGPLAN Not.*, Vol. 48. ACM, 519–530.

[40] Gabe Rudy. 2010. *CUDA-CHiLL: A Programming Language Interface for GPGPU Optimizations and Code Generation*. Ph.D. Dissertation. School of Computing, University of Utah.

[41] Amin Shafiee Sarvestani, Erik Hansson, and Christoph Kessler. 2013. Extensible recognition of algorithmic patterns in DSP programs for automatic parallelization. *Int. J. Parallel Prog.* 41, 6 (1 Dec. 2013), 806–824. DOI: https://doi.org/10.1007/s10766-012-0229-2

[42] Gordon D. Smith. 1985. *Numerical Solution of Partial Differential Equations: Finite Difference Methods*. Oxford University Press.

[43] Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. 1996. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 10th International Conference on Supercomputing (ICS'96)*. ACM, New York, NY, 18–25. DOI: https://doi.org/10.1145/237578.237581

[44] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. 2014. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.* 13, 4s, Article 134 (Apr. 2014), 25 pages. DOI: https://doi.org/10.1145/2584665

[45] Arvind K. Sujeeth, Austin Gibbons, Kevin J. Brown, HyoukJoong Lee, Tiark Rompf, Martin Odersky, and Kunle Olukotun. 2013. Forge: Generating a high performance DSL implementation from a declarative specification. In *Proceedings of the 12th International Conference on Generative Programming: Concepts & Experiences (GPCE'13)*. ACM, New York, NY, 145–154. DOI: https://doi.org/10.1145/2517208.2517220

[46] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Aleksandar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. 2013. Composition and reuse with compiled domain-specific languages. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'13)*, Giuseppe Castagna (Ed.). Springer Berlin, 52–78.

[47] Allen Taflove and Susan C. Hagness. 2005. *Computational Electrodynamics: The Finite-difference Time-domain Method*. Artech House.

[48] Thiago S. F. X. Teixeira, Corinne Ancourt, David Padua, and William Gropp. 2019. Locus: A system and a language for program optimization. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO'19)*. IEEE Press, Piscataway, NJ, 217–228. http://dl.acm.org/citation.cfm?id=3314872.3314898.

[49] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *CoRR* abs/1802.04730 (2018). arxiv:1802.04730 http://arxiv.org/abs/1802.04730.

[50] Sven Verdoolaege. 2010. Isl: An integer set library for the polyhedral model. In *Proceedings of the 3rd International Congress Conference on Mathematical Software (ICMS'10)*. Springer, Berlin, 299–302. http://dl.acm.org/citation.cfm?id=1888390.1888455.

[51] Sven Verdoolaege. 2011. Counting affine calculator and applications. In *Proceedings of the 1st International Workshop on Polyhedral Compilation Techniques (IMPACT'11)*.

[52] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral extraction tool. In *Proceedings of the 2nd International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*.

[53] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. 2014. Schedule trees. In *Proceedings of the 4th Workshop on Polyhedral Compilation Techniques (IMPACT'14, Associated with HiPEAC'14)*. 9.

[54] Sven Verdoolaege and Alexandre Isoard. 2017. Consecutivity in the isl polyhedral scheduler. (2017).

[55] Sven Verdoolaege and Gerda Janssens. 2017. *Scheduling for PPCG*. Report CW 706. Department of Computer Science, KU Leuven, Leuven, Belgium. DOI: https://doi.org/10.13140/RG.2.2.28998.68169

[56] Qing Yi. 2012. POET: A scripting language for applying parameterized source-to-source program transformations. *Softw. Pract. Exper.* 42, 6 (June 2012), 675–706. DOI: https://doi.org/10.1002/spe.1089

[57] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan, and Sanjay Rajopadhye. 2012. Alphaz: A system for design space exploration in the polyhedral model. In *Proceedings of the International Workshop on Languages and Compilers for Parallel Computing*. Springer, 17–31.

[58] Oleksandr Zinenko, Sven Verdoolaege, Chandan Reddy, Jun Shirako, Tobias Grosser, Vivek Sarkar, and Albert Cohen. 2018. Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling. In *Proceedings of the 27th International Conference on Compiler Construction*. ACM, 3–13.

Received August 2019; revised October 2019; accepted November 2019