LoopOpt: Declarative Transformations Made Easy

Lorenzo Chelini
TU Eindhoven
l.chelini@tue.nl

Martin Kong
University of Oklahoma
mkong@ou.edu

Tobias Grosser
University of Edinburgh
tobias.grosser@ed.ac.uk

Henk Corporaal
TU Eindhoven
h.corporaal@tue.nl

Abstract

Despite years of research, the optimization strategy of loop-level optimization frameworks remains fragile when addressing modern and heterogeneous architectures. Furthermore, optimizers act as an opaque operation, a black-box, to the users, forcing them to tedious an error-prone manual optimization if imprecise cost models are used. Current solutions, to drive loop-level optimizers rely on pragmas or bake the transformations recipes in the source code using imperative embedded scripting. But, the optimization of programs via a sequence of imperative directives is unlikely to solve this problem fully as expressing optimization is still an error-prone and time-consuming task for the users. The ideal solution would be a declarative approach that allows the users to opt-in if the optimizer has made a poor optimization decision but avoid baking the transformation script within a given application or bind it to a particular loop nest. Based on such an idea, we propose LoopOpt, an interactive tool that enables users to design optimizations in partnership with the compiler in a declarative way. Thus, our approach opens the polyhedral black-box allowing users to design complex optimizations sequences in a declarative way.

Keywords: Loop Tactics, Polyhedral model, Declarative Code Optimization, Polly

ACM Reference Format:
Lorenzo Chelini, Martin Kong, Tobias Grosser, and Henk Corporaal. 2021. LoopOpt: Declarative Transformations Made Easy. In *Proceedings of 24th International Workshop on Software and Compilers for Embedded Systems (SCOPES '21).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3493229.3493301

SCOPES '21, November 1–2, 2021, Eindhoven, Netherlands

ACM ISBN 978-1-4503-9166-5/21/06.
https://doi.org/10.1145/3493229.3493301

1 Introduction

As the hardware increases in complexity, compilers are becoming increasingly ineffective at generating efficient code automatically. But even worse, compiler optimizers act as black-boxes to the developers, being driven by one-size-fits-all heuristics that can be affected and controlled by few compiler switches [1, 2]. As a result, if an imprecise cost model is used, poor performance is achieved. Given a poor performance code developers have two options to recover from such a situation: 1) perform a tedious and error-prone manual optimization; 2) look at the internal intricacies of the optimizer and redesign the program optimization scheme. Both solutions are not optimal. The former is likely to incur a loss in productivity and portability, as a manually optimized code is tied to low-level and architecture-specific details. The latter requires the developers to dig down the optimizer's internal machinery, which will likely require a steep learning curve, as the internal program representation of the optimizer is often disconnected from any syntactic form. Semi-automatic approaches based on pragmas or imperative embedded scripting have been proposed as a solution [1, 5, 12, 15, 21]. But they fall short for the following reasons: 1) They lack immediate feedback on transformation profitability. 2) The pragmas or the embedded scripts are by nature tightly coupled with the code structure. As a consequence, minor changes to the code require an update on the transformation script, effectively reducing script composability and reuse. 3) The syntax adopted by most of the tools is biased to the internal representation of the optimizer (i.e., beta prefixes in Clay [1]). As a consequence, expressing complicated loop optimization tends to be cumbersome and restricted to few users. 4) Semi-automatic approaches require the developers to annotate and inspect the source code, which is a time consuming and error-prone task.

To overcome these limitations, we propose a declarative transformation and interactive tool named LoopOpt. LoopOpt eases the design and exploration of complex program transformations by 1) providing immediate feedback on transformation profitability, targeting the memory subsystem, which is known to be the bottleneck of today modern architectures. 2) Decoupling code structure from code optimization effectively enabling composition and reuse of transformation scripts. It comes with a concise syntax similar to pragmas

making the tool familiar to common compiler developers, and not bias on the tool's internal representation (i.e., beta prefix in Clay).

We bring the following contributions:

- We revisit classical loop transformations (i.e., tiling) in the light of a new declarative approach based on matchers and builders, which ensures transformations composability and reuse.
- We enable the search for the composition of program transformations in a declarative way. LoopOpt allows the users to describe loops and computational motif declaratively and apply transformation recipes on top of them.
- We evaluate our approach on a modern architecture, showing that with our approach is possible to obtain performance which closely match and sometimes overcome the one of state-of-the-art polyhedral optimizers.

2 Motivating Example

Loop-optimizer frameworks provide state-of-the-art yet not flawless automatic optimizations. To mitigate the problem, semi-automatic approaches such as Clay [1], AlphaZ [21], and Chill [5], emerged as a possible solution, allowing the users to steer the optimizer. 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 1. 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, or the loop depths or both would change. Worse, the user must keep track of the transformation effects on the loop structure to spell subsequent transformations. With our approach, on the other hand, it is possible to make the transformations declarative. Instead of binding it to specific loops, we can bind it to a computational motif. Thus the transformation recipe can be applied repeatedly to the entire program without any user intervention. Besides, our approach by providing a GUI, allows optimizations to be nicely spelt and avoids the mental exercise of tracking transformation effects. The inlined code below shows the exact same transformation recipe of Listing 1 in LoopOpt.

```
pattern[C(i,j) += A(i,k) * B(k,j)]
tile(i,32), tile(j,32), tile(k,32)
```

The first part declaratively describes a computational motif to be located in the codebase, and it is introduced by the

```
/* 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 < N; i++)
  for (int j = 0; j < N; j++)
    for (int k = 0; k < N; ++k)
S1:   C[i][j] += alpha * A[i][k] * B[k][j];
```

Listing 1. Imperative tiling optimization for a GEMM statement using Clay.

`pattern` keyword. We use Tensor Comprehensions (TC) syntax to describe a computational motif, a slight variation of the ubiquitous Einstein notation [17]. Each pattern directive is lowered to a matcher. The second part of the tactic describes how the motif should be optimized, in this case, using the `tile` directive. Each directive, on the other hand, is lowered to one or multiple builders. A more in-depth and formal description of the matchers, builders, and tactic syntax will be given in Section 3.3 and Section 4.2. For now, we remark that a given tactic is not bound to any loop structure and can be repeatedly applied to the entire program. Besides, a tactic does not require the user to annotate the source code, as our pattern matching and rewriting framework automatically locates (and rewrite) a specified computational motif.

3 Polyhedral Representation of Programs

After more than three decades of active research, the polyhedral model [7] has become the cornerstone to model and transform loops in imperative program powering production compilers such as GCC [16], LLVM [8], and IBM XL [3]. The model applies to loop-based programs, referred to as SCoPs, where loop bounds and array subscripts are affine functions of surrounding loop iterations and fixed parameters.

3.1 Iteration domain and Memory Accesses

Individual statements are represented by the iteration domain, which assigns to each statement a symbolic name and an integer vector in a k-dimensional space, where k is the depth of the surrounding loops. Each point in such a vector represents a particular statement instance. For example, the iteration domain for the GEMM kernel reported in Listing 1 is $\{S_1(i,j,k) \mid 0 \leq i,j,k < N\}$ where N is a parameter. On the other hand, memory accesses are expressed as piecewise quasi-affine functions, which map the iteration space with the array space, whose coordinates are the values of the accessed subscripts. For the statement S1 in our running example, the accesses for the C reference are described as: $\{S_1(i,j) \rightarrow \mathrm{C}_{\mathrm{read}}(i,j); S_1(i,j) \rightarrow \mathrm{C}_{\mathrm{write}}(i,j)\}$.

3.2 Schedule Tree

The order in which the statement instances are executed is defined by the schedule, which maps a point in the iteration space with a point in the time space. Within the polyhedral

model, the schedule is represented as a tree [20], where each node represents a partial schedule, and the order of loops and statements is determined by the node parent-child relation. The root of the tree is always a domain node, which encodes the iteration domain. Below such node a combination of the following nodes may exist: 1) band which defines the partial schedule of one or multiple loops; 2) `filter` which restricts the statement instances of the iteration domain; 3) sequence which imposes an order among its children.

3.3 Declarative Loop Tactics

Our interactive tool builds upon Loop Tactics, a framework to make modern constraint-based loop transformations as accessible as classical tree-based compiler transformations [4]. Loop Tactics introduces three main components: Schedule tree matchers, access relation matchers, and tree builders.

Schedule tree Matchers and Builders A schedule tree matcher enables the declarative description of the schedule subtree to match. Fundamentally, it replicates the node type-based structure of the schedule tree with additional filtering—via callback functions—and wildcarding capabilities. For example, a matcher may check band permutability if used to find tiling opportunities or outermost band parallelism if used for device mapping. Besides, it allows to capture specific schedule nodes that serve as pointers into the matched sub-tree. A builder uses captured nodes to implement transformations.

Tree builders use a syntax similar to the schedule tree matchers and enable declarative tree reconstruction. Loop optimizations are carried out using builders.

Access relation matchers Access relation matchers allow the caller to identify memory accesses with certain properties in a union of relation. The matching mechanism operates through placeholders (`placeholder` and `arrayPlaceholder`). A placeholder can match any affine expression of the form $\omega = k * \iota + c$ where k and c are coefficients while ω and ι are candidate patterns. As a simple example, Listing 2 shows how we can detect and optimize a GEMM pattern (Listing 1). Lines 15-16 locate the structural part of the GEMM. The innermost band in line 16 further restricts the matching to only the subtree that satisfies the callback "hasGemmPattern", which looks for a GEMM-like access pattern. A GEMM-like access pattern must read from three different arrays (lines 7-9) and write to a single one (line 10). Besides, the index permutation should satisfies the placeholder pattern [i,j] → [i,k][k,j]. Once the GEMM pattern has been matched, the builder (Line 18-21) applies the tiling transformation by splitting the original band into two nested ones. The outermost band having the tile loop schedule, while the innermost one having the point loops schedule.

4 LoopOpt - Interactive Code Optimization

Figure 1 shows the building blocks of LoopOpt and the interaction point with the users. Let us briefly describe the

```
1  auto hasGemmPattern = [&](schedule_node node)
2  {
3    auto _i, _j, _k = placeholder();
4    auto _A, _B, _C = arrayPlaceholder();
5    auto reads = /* get reads accesses */;
6    auto writes = /* get writes accesses */;
7    auto mRead = allOf(access(_C, _i, _j),
8                       access(_A, _i, _k),
9                       access(_B, _k, _j));
10   auto mWrite = allOf(access(_C, _i, _j));
11   return match(reads, mRead).size() == 1 &&
12     match(writes, mWrite).size() == 1;
13 };
14
15 auto matcher =
16   band(hasGemmPattern);
17
18 auto builder =
19   band([&]() { return tileSchedule(body, tileSizes); },
20     band([&]() { return pointSchedule(body, tileSizes); },
21       subtree(body)));
```

Listing 2. Schedule tree and access relation matchers (Line 1 to 16) for the GEMM statement in Listing 1. A builder (Line 18 to 21) to applies the tiling transformation is also shown.

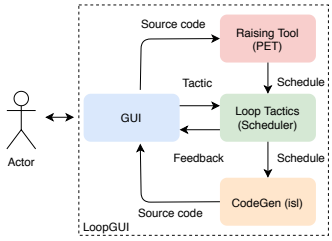

Figure 1. LoopOpt basic blocks and interaction points with the user. LoopOpt enables the specification of custom recipes for given computational motifs. The user interacts with the GUI only by entering a given transformation recipe (or tactic). The system returns immediate feedback and generates the transformed code on-demand.

internal machinery of LoopOpt. To start, the user interacts only with the GUI (see Section 4.1). Once the user opens a given application, we use the Polyhedral Extraction Tool (PET [19]) as our raising tool to extract and model a SCoP in the polyhedral model. From the SCoP, we obtain the schedule tree, and we feed it to Loop Tactics. Loop Tactics lowers each directive (Section 4.2) provided by the user to schedule tree matchers and builders, and applies them to the extracted tree. As a result, we obtain an optimized schedule that we send back to ISL (Integer Set Library [18]) for code generation. At the end of this process, the user will visualize how her directives optimized the original application. While spelling the transformation recipe, the user can ask for performance feedback. We provide two kinds of feedback: how the memory system behaves and how the current schedule

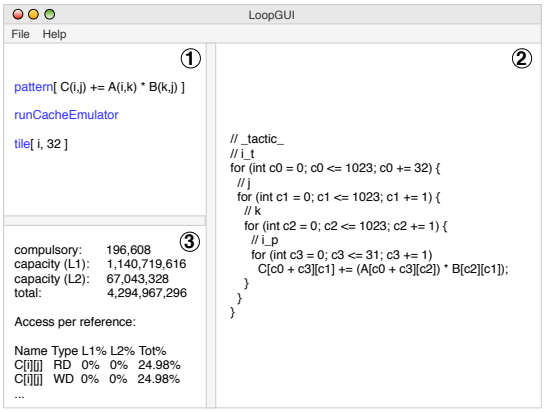

Figure 2. Loop Tactics interface with live-update and synchronized views. (1) Editable view to specify the transformation tactic; (2) live-update code (3) code editor switching between original code and user-provided feedback.

⟨*id*⟩ ::= [C identifier]
⟨*binOp*⟩ ::= '+' | '-' | '*' | ...
⟨*idList*⟩ ::= [comma separated id list]
⟨*stmt*⟩ ::= id (idList) '=' id (idList) { binOp ⟨ *id (idList)* ⟩ }
⟨*stmtList*⟩ ::= [whitespace separated stmt list]
⟨*pattern*⟩ ::= ⟨*stmt*⟩

Figure 3. Simplified EBNF syntax for `pattern` keyword. Brackets denote optional clauses, curly brakets denote repetitions, and square brackets contain textual description.

performs. The former is obtained by running Haystack, a fast cache emulator, which provides number of cache misses in the millisecond's range, thus providing cache-aware optimizations [9]. The latter is obtained by code generating the current schedule and timing its execution.

4.1 Graphical User Interface

Figure 2 shows our GUI. It combines three main views: ① is an editable window where the user can specify the tactic to apply to a given computational pattern in a given program. A new program can be open via File → Open. ② is a read-only window showing the live-updated code. The code will be updated by applying the specific transformation for each directive inserted by the user. ③ is a read-only window that provides feedback to the user (i.e., after running the Haystack cache emulator using the `runCachEmul` directive). The interface also provides a menu bar located on the top of the main window, with items for file operations.

4.2 Specification languages

A transformation recipe in our tool is called a "tactic" and consists of two parts: the specification of a pattern and a set of rewriting rules that describe how the pattern should be optimized. The pattern description is enclosed in the `pattern` directive, and it is spelt using Tensor Comprehensions (TC) notation [17]. Figure 3 shows the EBNF notation for the

```
auto hasDimensionality = [&](schedule_node node) {
  // check for a 3 nested loop.
  return node.schedule().dim == 3
};
auto hasPattern = [&](schedule_node node) {
  // check GEMM access patterns, same as `hasGemmPattern`.
};
auto matcher =
  band(_and(hasDimensionality, hasPattern));
```

Listing 3. Generated tree and access relation matchers.

Directive	Short description
`reverse`(loopTAG)	Reverse iteration order
`unroll`(loopTAG, factor)	unroll loop by factor
`tile`(loopTAG, factor)	tile loop by factor, and sink point loop
`interchange`(loopTAG, loopTAG)	interchange loops
`runCacheEmul`	run cache emulator on current schedule
`timeSchedule`	generate code for current schedule and time it

Table 1. Directives exposed by LoopOpt.

`pattern` directive. To be concrete, let us assume that we want to detect the statement S1 in our running example (Listing 1). To do so, the user will write the following TC expression as pattern directive: `C(i,j) += A(i,k) * B(k,j)`

The expression will get lowered to the tree, and access matchers reported in Listing 3. The tree matcher looks for a `band` node which satisfies two properties expressed as callback functions: `hasDimensionalty` and `hasPattern`. The former, ensures structural properties, looking for a band node containing three schedule dimensions which corresponds to a triple nested loop. The latter, guarantees access pattern properties, making sure that they equal the one of a GEMM pattern. The structural properties to be matched (i.e., the number of bands) is determined from the number of induction variables—three, in this case, i, j and k, while access pattern properties can be easily derived from the tensor specifications obtained from TC syntax (i.e., access to three different 2-d tensors). Once the pattern keyword has been specified, LoopOpt instantiates the matchers and runs them on the program. If any match happens, LoopOpt annotates the surrounding loops with tags that can be later referred by the rewriting rules.

How the pattern should be optimized is specified as a composition of (instances of) primitive, building-block operations. Currently, LoopOpt supports basic operations, which allow affine transformations on loop nests. Table 1 shows the operations supported, among with the directives the user can specify to obtain feedback on transformation profitability.

All the supported transformations work on loops. Each of them must specify the loop it applies to via the `loopTAG`. `tile` and `unroll` take an additional parameter which specifies the unrolling factor or the tile size of the targeted loop, respectively. For performance feedback, the user has two

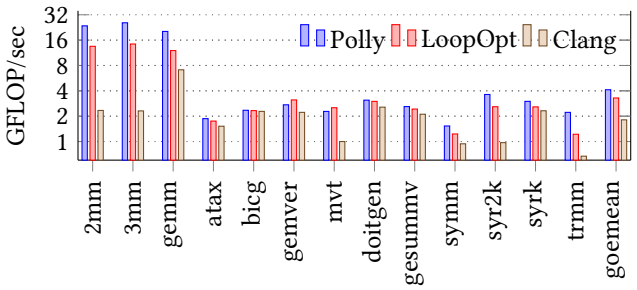

Figure 4. LoopOpt enables to get performance comparable, in some cases, with state-of-the-art optimizers like Polly, and better than current general-purpose compilers like Clang.

choices: `runCacheEmul` and `timeSchedule`. The former allows the user to run the Haystack cache emulator on the current schedule, which reflects the transformations applied so far. As output, the cache emulator provides the absolute number of misses for the L1 and L2 level-cache, and the percentage of cache misses for a given array reference, this latter information can be used to understand transformation profitability. The second option the user has is the `timeSchedule` directive. `timeSchedule` generates code from the current schedule, inserting proper initialization statements, and times its execution by running the code on the user machine.

5 Evaluation

We present an evaluation of LoopOpt using the linear-algebra kernels in the Polybench 4.2 (Figure 4. benchmark suite. Our experimental setup consist of an Intel Core i9-9900K (Coffee Lake) clocked at 3.6 GHz and running Ubuntu 18.04. The machine has 64GB of RAM, a L1 of 32KB, a L2 of 256KB and 16MB of L3. All the results are obtained considering the minimal execution time of 5 independent (single-thread) runs, for single-precision operand. We use the default large configuration for Polybench 4.2. We compare with Polly(`git 098130f`), a state-of-the-art polyhedral optimizer available in the LLVM compiler infrastructure, and consider Clang -O3 as our baseline (release 6.0.0). For each, program in the test suite we use LoopOpt to wrote a tactic. We experimented with different combinations of transformations to improve performance by improving locality (mainly loop interchange and tiling). We select those which showed significant improvements. All tactics were written and analyzed in week time. To lower the C code generated by LoopOpt we use Clang -O3. For the GEMM kernels (`2mm`, `3mm`, and `gemm`) we implement a tactic that partially reflects the OpenBLIS transformation [13]. Specifically, for each GEMM pattern, we first rearrange the band dimensions such that j will be the outermost dimension followed by k and i. Then we apply tiling and loop interchange to create three nested loops around the macro-kernel and two additional loops around the micro-kernel. The micro-kernel is a loop around an outer product

```
for (int i = 0; i < 1024; i++)
  for (int j = 0; j < 1024; j++)
    x2[i] = x2[i] + A[j][i] * y2[j];
```

Listing 4. Problematic access pattern in the `mvt`. Thanks to `runCacheEmul` which gives immediate performance feedback, the user can write a tactic to interchange loop *i* and *j* and improve performance.

that can be implemented in assembly. The macro-kernel is a two-dimensional loops around the micro-kernel. We then unroll the point loops in the micro-kernel to simplify subsequent vectorization. We get quite good performance, but we did not reach the same performance level as Polly, which implements the full BLIS transformation. The same performance should eventually be reachable once we support the packing data-layout transformation which ensures strided accesses for A and B arrays, which Polly does.

Turning our attention on matrix-vector like kernels (`atax` to `gesummv`) we can notice that LoopOpt gets better performance than Polly on `mvt` and `gemver`. These two kernels make an interesting case for providing an immediate feedback on transformation profitability, focusing on the memory subsystem. A closer look by running the Haystack cache emulator via the `runCacheEmul` shows a problematic column-wise access in both kernels. Listing 4 shows the problematic access in the `mvt` benchmark where we can see a column-wise traversal for array reference A, which reflects in an increase of L1 and L2 misses detected by Haystack. Thanks to this hint, we wrote a tactic that exchanges the *i* and *j* loop, thus getting better access pattern property and better performance. For the remaining kernels, we write a default tactic that applies a tiling transformation with a tile factor of 32 along each dimension similar to what Polly does. While we get better performance than Clang, we miss the same level of optimization of Polly as Polly runs the isl optimizer and applies additional metadata information that indicates, for instance, that two arrays do not alias.

6 Related Work

Directive-based Loop Transformation Frameworks: Multiple works have already explored the composition of loop transformation through directive-based or programmer-assisted frameworks; the unifying reordering transformations framework is probably the very first of them [10]. Yuki et al. developed AlphaZ, a framework to express transformations as a set of equations using the Alpha language [21]. The framework uses a script-driven approach to spell loop transformations. Similarly, Donadio et al introduced the Xlanguage, an embedded DSL based on C/C++ pragmas, that allows generating multi-versioned programs by spelling the transformations to apply [6]. 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 [5]. Namjoshi et al. developed Loopy, a framework integrated into Polly for loop optimization, each user transformation is verified for correctness by using the precise dependencies analysis of the polyhedral model. Müller-Pfefferkorn et al. proposed Goofi to assist programmers in applying loop transformations for cache hierarchy and parallelism [14]. Goofi provides a user interface to make it easier to apply such transformations. Kruse et al. submitted a proposal to enhance pragma-based transformations in the Clang front-end [12]. A proposal prototype has been implemented using Clang and Polly. One of the key aspects of their proposal is the possibility of assigning identifiers to loops and referring to them in loop transformations, which we adopt in our work. All these tools expose some scheduling-based languages, but they imperatively do that. LoopOpt, on the other hand, adopts a declarative specification. Instead of binding the transformation recipe to a loop nest, we bind it to a computational motif.

Interactive Loop Transformation Frameworks:
Zinenko et al. leveraged the geometric nature of the polyhedral model by developing Clint, a visualization representation tool for parallelism extraction based on the precise dependence analysis of the polyhedral model and real-time feedback to ensure code correctness [22]. Kennedy et al. introduced the ParaScop Editor, an interactive parallel programming tool assisting in parallelizing Fortran code by combining programmer expertise with extensive analysis and program transformations [11].

7 Conclusion

We present LoopOpt an interactive tool to spell transformations for a computational motif declaratively. Contrary to existing tools, LoopOpt's declarative nature allows decoupling code structure from code optimization , thus making the transformation recipe resilient to source-code changes.

References

[1] Lénaïc Bagnères and Others. 2016. Opening Polyhedral Compiler's Black Box. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO '16)*. ACM, New York, NY, USA, 128–138. https://doi.org/10.1145/2854038.2854048

[2] Cédric Bastoul. 2016. Mapping Deviation: A Technique to Adapt or to Guard Loop Transformation Intuitions for Legality. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. ACM, New York, NY, USA, 229–239. https://doi.org/10.1145/2892208.2892216

[3] Uday Bondhugula and Others. 2008. A Practical Automatic Polyhedral Parallelizer and Locality Optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '08)*. ACM, New York, NY, USA, 101–113. https://doi.org/10.1145/1375581.1375595

[4] Lorenzo Chelini and Others. 2019. Declarative Loop Tactics for Domain-Specific Optimization. *ACM Trans. Archit. Code Optim.* 16, 4, Article 55 (Dec. 2019), 25 pages. https://doi.org/10.1145/3372266

[5] Chun Chen and Others. 2008. *CHiLL: A framework for composing high-level loop transformations.* Technical Report. Citeseer.

[6] Sebastien Donadio and Others. 2006. A Language for the Compact Representation of Multiple Program Versions. In *Languages and Compilers for Parallel Computing*, Eduard Ayguadé and Others (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 136–151.

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

[8] Tobias Grosser and Others. 2011. Polly-Polyhedral optimization in LLVM. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011. 1.

[9] Tobias Gysi and Others. 2019. A Fast Analytical Model of Fully Associative Caches. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 816–829. https://doi.org/10.1145/3314221.3314606

[10] W. Kelly and W. Pugh. 1995. A unifying framework for iteration reordering transformations. In *Proceedings 1st International Conference on Algorithms and Architectures for Parallel Processing*, Vol. 1. 153–162 vol.1. https://doi.org/10.1109/ICAPP.1995.472180

[11] Ken Kennedy and Others. 1991. Interactive parallel programming using the ParaScope Editor. *IEEE Transactions on Parallel & Distributed Systems* 3 (1991), 329–341.

[12] Michael Kruse and Hal Finkel. 2018. User-Directed Loop-Transformations in Clang. In *2018 IEEE/ACM 5th Workshop on the LLVM Compiler Infrastructure in HPC (LLVM-HPC)*. IEEE, 49–58.

[13] Tze Meng Low and Others. 2016. Analytical Modeling Is Enough for High-Performance BLIS. *ACM Trans. Math. Softw.* 43, 2, Article 12 (Aug. 2016), 18 pages. https://doi.org/10.1145/2925987

[14] Ralph Müller-Pfefferkorn and Others. 2004. Optimizing Cache Access: A Tool for Source-to-Source Transformations and Real-Life Compiler Tests. In *Euro-Par 2004 Parallel Processing*, Marco Danelutto and Others (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 72–81.

[15] Gabe Rudy and Others. 2010. A Programming Language Interface to Describe Transformations and Code Generation. In *Languages and Compilers for Parallel Computing*, Keith Cooper and Others (Eds.). Number 6548 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 136–150.

[16] Konrad Trifunovic and Others. 2010. GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation. In *GCC Research Opportunities Workshop (GROW'10)*. Pisa, Italy. https://hal.inria.fr/inria-00551516

[17] Nicolas Vasilache and Others. 2019. The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically. *ACM Trans. Archit. Code Optim.* 16, 4, Article 38 (Oct. 2019), 26 pages. https://doi.org/10.1145/3355606

[18] Sven Verdoolaege. 2010. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, Komei Fukuda, Joris van der Hoeven, Michael Joswig, and Nobuki Takayama (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 299–302.

[19] Sven Verdoolaege and Tobias Grosser. 2012. Polyhedral Extraction Tool. In *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*. Paris, France.

[20] Sven Verdoolaege and Others. 2014. Schedule trees. In *International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria.*

[21] Tomofumi Yuki and Others. 2012. Alphaz: A system for design space exploration in the polyhedral model. In *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 17–31.

[22] O. Zinenko and Others. 2014. Clint: A direct manipulation tool for parallelizing compute-intensive program parts. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 109–112. https://doi.org/10.1109/VLHCC.2014.6883031