# Progressive Raising in Multi-level IR


Lorenzo Chelini
*TU Eindhoven*
Eindhoven, The Netherlands
l.chelini@tue.nl

Andi Drebes
*Inria and École Normale Supérieure*
Paris, France
andi@programmierforen.de

Oleksandr Zinenko
*Google*
Paris, France
zinenko@google.com

Albert Cohen
*Google*
Paris, France
albertcohen@google.com

Nicolas Vasilache
*Google*
Zurich, Switzerland
ntv@google.com

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

Henk Corporaal
*TU Eindhoven*
Eindhoven, The Netherlands
h.corporaal@tue.nl



***Abstract*—Multi-level intermediate representations (IR) show great promise for lowering the design costs for domain-specific compilers by providing a reusable, extensible, and non-opinionated framework for expressing domain-specific and high-level abstractions directly in the IR. But, while such frameworks support the progressive lowering of high-level representations to low-level IR, they do not raise in the opposite direction. Thus, the entry point into the compilation pipeline defines the highest level of abstraction for all subsequent transformations, limiting the set of applicable optimizations, in particular for general-purpose languages that are not semantically rich enough to model the required abstractions.**

**We propose *Progressive Raising*, a complementary approach to the progressive lowering in multi-level IRs that raises from lower to higher-level abstractions to leverage domain-specific transformations for low-level representations. We further introduce *Multi-Level Tactics*, our declarative approach for progressive raising, implemented on top of the MLIR framework, and demonstrate the progressive raising from affine loop nests specified in a general-purpose language to high-level linear algebra operations. Our raising paths leverage subsequent high-level domain-specific transformations with significant performance improvements.**



***Index Terms*—MLIR, progressive raising, multi-level intermediate representation**


## I. Introduction

The increasing complexity of hardware resulting from the ongoing trend for heterogeneous systems has made it difficult for general-purpose compilers to generate efficient code automatically [1]. One of the main issues is the mismatch between the low level of abstraction at which general-purpose compilers operate and the various high-level abstractions for computation required by today's applications [2]. Although high-level programming languages allow for the specification of high-level operations, this information is often not captured by the low-level intermediate representation (IR) of general-purpose compilers or lost early in the compilation process during lowering [3].

Domain-specific languages (DSLs) and compilers attempt to capture and explicitly preserve high-level information throughout the compilation process and have been employed successfully to generate efficient code for modern hardware [4], [5]. However, such languages commit to a limited set of isolated abstractions and domain-specific optimizations, resulting in poor interoperability, limited reusability of software components, and few opportunities for inter-domain optimizations [6].

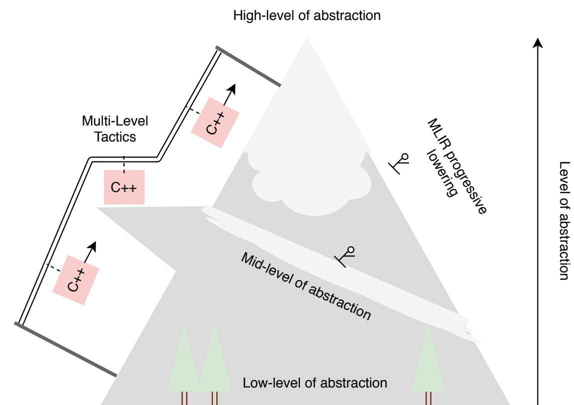


Fig. 1: Multi-Level Tactics lifts general-purpose languages to higher-abstraction levels to enable effective domain-specific compilation via progressive lowering.

Multi-level intermediate representations explicitly allow for the co-existence of multiple abstractions within the same compilation framework with interoperable representations, breaking the isolation between domains and enabling comprehensive optimizations. During compilation, the source program's high-level representation is progressively optimized and transformed to lower-level abstractions, until reaching a low-level, general-purpose representation for code generation [7].

Multi-level frameworks solve many issues of DSLs, but the optimizations in progressive lowering compilation scheme crucially rely on the adequate initial representation of the source program. If the initial representation is below the required level of abstraction for a given optimization, the optimization simply fails to apply. However, providing an adequate high-level input representation may not always be possible. General-purpose languages not being semantically rich enough to preserve the right level of information enter the lowering pipeline at a very low level, thus precluding most, if not all, domain-specific

optimizations. Asking the user to match the abstraction used internally by the framework manually is inconvenient, as it requires the user to learn the internals of the tool and the different abstraction levels.

We present *Multi-Level Tactics*, a technique that allows for raising between different levels of abstractions in multi-level IRs. In particular, our technique enables *progressive raising*, a complementary technique to the progressive lowering, successively lifting a lower-level representation to a higher-level specification of the source program. Multi-Level Tactics relaxes the requirement for specific entry representations and leverages existing domain-specific optimizations already available in multi-level IRs for low-level representations.

Figure 1 illustrates the main idea of this work: Starting with a less expressive representation in the valley, our solution enables lifting to intermediate levels of abstractions at different heights of the mountain or up to the highest level of representation at the peak. The low-level representation for code generation in the valley can be reached using different lowering and optimization paths, represented by the different slopes.

The contributions of this work are as follows:

- Multi-Level Tactics, a concept for progressive raising (or successive lifting) from lower to higher-level abstractions in multi-level IRs.
- A high-level declarative language to define lifting transformations independent of the loop, iterator, container, and indexation abstractions it targets and which are subject to rewriting and lifting.
- The integration of Multi-Level Tactics within the MLIR compiler infrastructure and implemented a set of transformations from loop-based MLIR dialects to a linear algebra dialect, as well as a subsequent optimization for matrix-chain multiplications showcasing the progressive raising throughout multiple levels of abstraction.
- An empirical evaluation of the code generated with our framework including a comparison with state-of-the-art optimizers, motivating the potential impact of successive lifting on performance.

The remainder of the paper is organized as follows: Section II introduces a multi-level IR framework with the necessary background information. Section III gives a high-level view of Multi-Level Tactics and how it enables progressive raising in a multi-level IR. In Section IV, we provide more details on the syntax introduced by our framework. Section V evaluates our framework's applicability for two different raising paths, Affine-to-Affine and Affine-to-Linalg. We conclude by comparing with prior works and highlighting future directions.

## II. The MLIR Infrastructure

The MLIR compiler infrastructure is a project under the LLVM umbrella that is well-suited for multi-level IR rewriting [7]. MLIR provides a non-opinionated intermediate representation (IR) with only few concepts being built-in, leaving most of the IR customizable. A customizable IR allows compiler developers to match the right abstraction level for their problem at hand by introducing custom types, operations, and attributes. Operations are the essential atomic constituents of the IR; each operation uses and produces new values. A value represents data at runtime, and it is associated with a type known at compile-time, whereas types model compile-time information about values. Complementary to this, attributes attach compile-time information to operations. Custom types, operations, and attributes are logically grouped into dialects. A dialect is a basic ingredient that enables the MLIR infrastructure to implement a stack of reusable abstractions. Each abstraction encodes and preserves transformation validity preconditions directly in its IR, reducing the complexity and the cost of analysis passes.

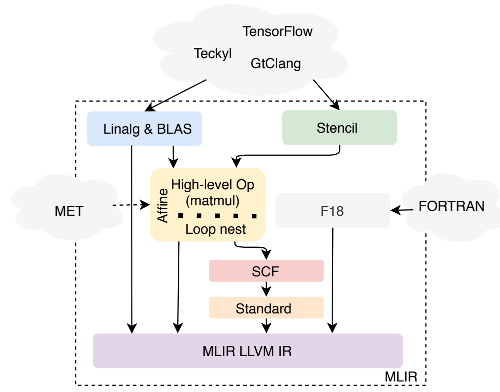


Fig. 2: Some of the available dialects in MLIR. The higher is a dialect the higher is its abstraction level. Different entry points in the MLIR toolchain are also shown.

Figures 2 shows some of the dialects available in MLIR and their entry points. The Linalg dialect models linear-algebra operations on either tensor or buffer operands. At a similar abstraction level, the Stencil dialect represents iterative kernels that update array elements according to a given stencil pattern [8]. At a lower abstraction level, the Affine dialect models a simplified polyhedral representation, while F18 models FORTRAN specific constructs (i.e., dispatch table). SCF and Standard represent structured control flow and a collection of miscellaneous operations, respectively. Lastly, MLIR LLVM dialect models LLVM-IR constructs.

IR customization is enabled through a declarative system, mostly based on TableGen. TableGen is a data modelling tool for defining and maintaining records of domain-specific information, and is extensively used across the LLVM codebase [9]. For example, MLIR declaratively describes operations using an Operation Description Specification language (ODS) built on top of TableGen. Declarative specification for new operations speed up the development of new custom IRs and reduces the amount of code duplication as well as the probability of errors. TableGen files are only "containers" of domain-specific information. They do not have any meaning without a backend. It is up to the backend at compile time to interpret the stored information and generate the C++ declarations and definitions.

Progressive lowering (downward arrows in Figure 2) enables

```
%0 = linalg.matmul(%A, %B, %C) // linalg dialect
           |
           v
affine.for %i = 0 to %N // affine dialect
  affine.for %j = 0 to %N
    affine.for %k = 0 to %N {
      %0 = affine.load %C[%i, %j] : memref<?x?xf32>
      %1 = affine.load %A[%i, %k] : memref<?x?xf32>
      %2 = affine.load %B[%k, %j] : memref<?x?xf32>
      %3 = std.mulf %1, %2
      %4 = std.addf %3, %0
      affine.store %4, %C[%i, %j] : memref<?x?xf32>
    }
```

Listing 1: Progressive lowering in MLIR. A linalg matmul translates into a nested loop with loads from 2-d memrefs, one addition, and one multiplication in the Affine dialect.

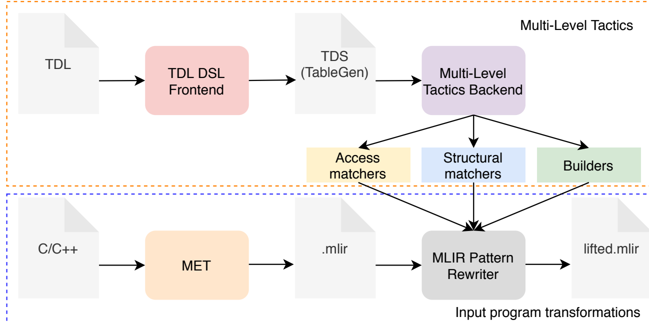


Fig. 3: Multi-Level Tactics compilation flow (orange box) and transformations of the input program (blue box).

lowering operations from high-level abstractions dialects to low-level IRs. Listing 1 shows an example of progressive lowering of a linalg.matmul operation to a triple affine loop containing a multiplication and an addition in the Affine dialect. Subsequent lowering fuses the operations of the body to a multiply-and-accumulate operation (MAC). Efficient progressive lowering is enabled through a pattern rewriting infrastructure [7].

## III. Multi-Level Tactics Overview

The key idea of Multi-Level Tactics is to allow for transformations from lower to higher levels of abstractions. By providing the infrastructure for raising from common abstractions of general-purpose languages to specialized high-level dialects, Multi-Level Tactics enables domain-specific optimizations for general-purpose code in multi-level IR compilers.

Figure 3 illustrates the overall flow for transformations using Multi-Level Tactics. The steps at the top allow tactics specification and the generation of the code working on the actual IR (orange box). In contrast, the bottom steps represent the transformations of the input program (blue box).

All tactics are described in a high-level, declarative specification *Tactics Description Language (TDL)*. TDL enables to specify the computational pattern to which the tactic applies and a set of replacement expressions. The TDL DSL frontend processes the high-level declarative specification and emits a TableGen entry called *Tactics Description Specification (TDS)*. Sections III-A and III-B provide a detailed description of these two formats. The actual code for the matching of IR operations

```
    for (int a = 0; a < N; ++a)
      for (int b = 0; b < M; ++b)
        for (int c = 0; c < N; ++c)
          for (int d = 0; d < O; ++d)
S1:         C[a][b][c] += A[a][c][d] * B[d][b];
    }
```

Listing 2: Contraction abc-acd-db.

and memory accesses, as well as for building replacement operations, is generated by the Multi-Level Tactics backend and uses the MLIR pattern rewriting infrastructure upon execution (more details in Section III-C).

Transformations on the input program start by translating C code to Affine using the MLIR Extraction Tool (MET). MET enables entering the MLIR pipeline at the Affine level from C code for the subset of the language within the polyhedral model. During translation, the Cécode is canonicalized by distributing loops to simplify pattern recognition. The Affine representation is then fed into the MLIR pattern rewriter engine, where the generated tactics have been hooked. The output of the entire flow is a lifted MLIR where loop nests have been raised to high-level operations. [1]

### A. Tactics Description Language - TDL

The key user-facing component of Multi-Level Tactics is the Tactics Description Language (TDL). Pattern and replacements in TDL are specified in a syntax borrowed from Tensor Comprehensions, which is itself a slight variation of the ubiquitous Einstein tensor index notation [10]. As a concrete example, we illustrate the syntax for the pattern and replacement for a transformation lifting a sequential, loop-based implementation of a contraction of the form `abc-acd-bd` (Listing 2). We raise the contraction by rewriting it with an equivalent Transpose-Transpose-GEMM-Transpose (TTGT) expression taking advantage of highly efficient GEMM implementations offered by vendor-optimized libraries. The TTGT computation first flattens the tensors into matrices via explicit tensor transpositions and reshape operations, then executes GEMM, and finally folds back the resulting matrix into the original tensor layout.

Listing 3 shows the user-defined tactic to detect and optimize the contraction. The signature of the tactic is composed of the keyword `def`, followed by a name (`TTGT`). The body consists of two parts delimited by the keywords `pattern` and `builder`: `pattern` describes the computational motif to be detected in the user code, whereas `builder` describes the transformation recipe. For example, Lines 5 and 6 reshape the tensors $C$ and $A$ into matrices. Additionally, a transposition $(a, b, c) \rightarrow (a, c, b)$ is performed for $C$ before reshaping. Line 7 specifies the GEMM operation, while Line 8 folds back the result into the original tensor layout, again emitting an implicit transposition $(a, c, b) \rightarrow (a, b, c)$ after folding.

### B. Tactics Description Specification (TDS)

The role of the TDL DSL frontend is to generate the TableGen-based Tactics Description Specification (TDS). Each

[1]Multi-Level Tactics can also lift from SCF.

```
def TTGT {
    pattern
    C(a,b,c) += A(a,c,d) * B(d,b)
    builder
    D(f,b) = C(a,b,c)     where f = a * c
    E(f,d) = A(a,c,d)     where f = a * c
    D(f,b) += E(f,d) * B(d,b)
    C(a,b,c) = D(f,b)     where f = a * c
}
```

Listing 3: TDL to match a contraction abc-acd-db and apply the TTGT optimization on each detected pattern.

```
def TTGT : Tactic<C(a, b, c) += A(a, c, d) * B(d, b), [
  transposeBuilder<In<[C]>, Out<[D]>, Expr<{0, 2, 1}>>,
  reshapeBuilder<In<[D]>, Out<[E]>, Expr<{{0, 1}, 2}>>,
  reshapeBuilder<In<[A]>, Out<[F]>, Expr<{{0, 1}, 2}>>,
  matmulBuilder<In<[F, B]>, Out<[E]>>,
  reshapeBuilder<In<[E]>, Out<[D]>, Expr<{{0, 1}, 2}>>,
  transposeBuilder<In<[D]>, Out<[C]>, Expr<{0, 2, 1}>>,
]>;
```

Listing 4: TDS to match a contraction abc-acd-db and apply the TTGT optimization on each detected pattern.

TDS file consists of a series of instantiations of TableGen templates, from which C++ code is generated at compile time. The choice of a two-step process instead of direct code generation from TDL allows common routines for pattern matching and infrastructure to be factored as reusable templates in TDS and thus reduces complexity.

Listing 4 shows the generated TDS definition for the TTGT tactic from Listing 3. Each TDS entry derives from a base class `Tactic`, which allows defining the pattern and a list of builders. The pattern specification matches the `pattern` entry in the TDL tactic. The list of builders, on the other hand, corresponds to a set of high-level operations with a one-to-one mapping with operations exposed by high-level dialects (i.e., Linalg). For example, the `transposeBuilder` will create a Linalg `TransposeOp` or a function call to a transpose BLAS routine, depending on the lowering path selected by the user. Considering our running example, line 5 in Listing 3 will map to a transpose operation followed by a reshape one (lines 2 and 3 in Listing 4). Similarly, lines 6 and 7 will map to a reshape and a matmul operation, respectively. Finally, line 8 (Listing 3) will map to a reshape operation followed by a transpose one (lines 6 and 7 in Listing 4). Ultimately, each entry in TableGen gets lowered to C++ matchers and builders via Multi-Level Tactic's TableGen backend at compile-time, as we will see in the next section.

### C. Matchers and Builders

The code generated from each TDS entry consists of four parts: Structural matchers, operation matchers, access matchers, and builders.

*Structural and Operation Matchers:* The role of a structural matcher is to detect control flow patterns in the IR. It essentially replicates the control flow-based structure of the IR with additional filtering capabilities. A structural matcher consists of a control flow operation type, a list of children operations, and an optional filtering function. For example, Listing 5, matches all the IR subtrees, consisting of a two-dimensional, perfectly

```
auto isMAC = [&a, &b, &c](Body loop) {
  auto MACOp = m_Op<AddOp>(a, m_Op<MulOp>(b, c));
  return MACOp.match(loop);
}
/* instantiate the context */
For(
  For(isMAC) // filtering function
));
```

Listing 5: Structural matchers declaratively describe the control-flow structure of the IR.

nested loop, where the innermost loop body contains a MAC operation, verified via the callback function `isMAC`. The top operation, referred to as a relative root, defines a structural matcher. The matching operation starts at a specified operation in the IR. It then recursively walk the operation's descendant as well as the relative root matcher descendants. If a mismatch is encountered during the traversal, the procedure stops, and the failure is reported immediately. The API to construct structural matchers is designed to resemble the structure of the IR itself visually. Leading arguments include optionally a callback function, which allows the caller to control the matching more precisely. For example, identify a MAC operation in the loop body requires checking a non-structural property. Each matcher must belong to a context which handles memory allocation and ownership. Operation matchers, on the other hand, verify the types of arithmetic operations. We rely on the m_Op matcher, which carries the type of operation to be matched. m_Op can be chained to detect sequences of operations. In our running example, `MACOp` looks for an Add operation with two arguments, where the second one is a Mul operation. The arguments of each operations are captured for later inspection.

*Access matchers:* Complementary to structural matcher, Multi-Level Tactics provides access pattern matchers. The access matchers rely on the idea of "placeholders", modeled as `m_Placeholder` and `m_ArrayPlaceholder`. The former can match any induction dimension of the form $k * \iota + c$, where $k$ and $c$ are coefficients forming the pattern, whereas $\iota$ defines a candidate by matching the underneath `mlir::Value` representing the induction variable. In contrast to this, `m_ArrayPlaceholder` can only match tensor accesses and takes a list of `m_Placeholder` as inputs. In both cases, a match is an assignment of a candidate to the placeholder. Candidates assigned to different placeholders are required to be distinct, while multiple references to the same placeholder within a matcher expression must refer to the same candidate. Placeholders can be combined in *placeholder expressions* (e.g., _C{_i, _j) and can be used in the `m_Op` matcher which allows for the specification of a particular type of operation (i.e., StoreOp or LoadOp). The matching procedure starts by inspecting the last store instruction within an MLIR block–an ordered list of operations without control flow. It then walks backwards following the use-def chain. If for one or more placeholders no candidates can be found during the backward traversal, the absence of a match is reported immediately, and the procedure stops. The programming interface is similar in spirit to that of structural matchers providing a declarative way

```
/* instantiate the context */
auto _i = m_Placeholder(), _j = m_Placeholder();
auto _A = m_ArrayPlaceholder();
auto matcher = m_Op<LoadOp>(_A({2*_i+1, _j+5}));
```

Listing 6: Declarative access pattern matcher.

```
1  For(For(For(For(access_callback())))));
2
3  auto access_callback = [&a](Body loop) {
4      {
5      AccessPatternContext pctx(/* MLIR ctx */);
6
7      auto _a = m_Placeholder();
8      auto _b = m_Placeholder();
9      auto _c = m_Placeholder();
10     auto _d = m_Placeholder();
11
12     auto _C = m_ArrayPlaceholder();
13     auto _A = m_ArrayPlaceholder();
14     auto _B = m_ArrayPlaceholder();
15
16     auto var0 = m_Op<AffineStoreOp>(_C({_a, _b, _c}));
17     /* check the store is the last instruction in
18        the block */
19
20     auto var1 = m_Op<AffineLoadOp>(_C({_a, _b, _c}));
21     auto var2 = m_Op<AffineLoadOp>(_A({_a, _c, _d}));
22     auto var3 = m_Op<AffineLoadOp>(_B({_d, _b}));
23     auto body = m_Op<AddOp>(var1, m_Op<MulOp>(var2, var3));
24     /* match the body starting from the store op
25        and make sure we have only the defined
26        operations in the block */
27
28     a = pctx[_a] // read out the matched value
29     ...
30     }
31 };
```

Listing 7: Structural and access matchers emitted by Multi-Level Tactic's backend for the tactic defined in Listing 3.

of specifying access patterns. Similarly, each placeholder must belong to a context that orchestrates the matching procedure, handles memory allocation and ownership. Listing 6 shows how the caller can identify a load from a 2D array.

*Builders:* The replacement for the matched patterns of a transformation is generated by the builders, which instantiate IR operations either directly (e.g., when generating calls to vendor-optimized libraries) , or using already existing infrastructure from the MLIR ecosystem (e.g., EDSC builders [11]).

Going back to our running example, Line 1 in Listing 7 shows the structural matchers emitted for our contraction. These match all 4-dimensional loop nests for which evaluation of `access_callback`, invoking the generated access matchers in Lines 5 to 30, evaluates to true. In particular, the access matchers look for an MLIR block which contains exactly three read accesses to different tensors: one write access, an index permutation that satisfies the placeholder pattern $[a, b, c] \rightarrow [a, c, d][d, b]$, and two arithmetic operations (+/*) which define the computation of the contraction.

## IV. Multi-Level Tactic syntax

Figure 4 and 5 show the grammar of the Tactics Description Language (TDL), and the Tactics Description Specification (TDS), respectively. TDL extends the TC syntax [2] with the `pattern` and the `builder` keywords. Figure 4 shows a simplified version of the TDL core syntax. Figure 5

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

Fig. 4: Simplified EBNF syntax for Tactics Description Language. Brackets denote optional clauses, curly brackets indicate repetitions, and square brackets contain a textual description for simplicity.

⟨*tactic*⟩ ::= ⟨*pattern*⟩ { ⟨*builder*⟩ }
⟨*pattern*⟩ ::= ⟨*TC-expression*⟩
⟨*builderId*⟩ ::= reshapeBuilder | transposeBuilder | matmulBuilder | matvecBuilder | convBuilder
⟨*input*⟩ ::= ⟨*whitespace separated list of string*⟩
⟨*output*⟩ ::= ⟨*string*⟩
⟨*affineExpr*⟩ ::= ⟨*string*⟩
⟨*builder*⟩ ::= ⟨*builderId*⟩(⟨*input*⟩, ⟨*output*⟩, [⟨*affineExpr*⟩])

Fig. 5: Simplified EBNF syntax for Tactics Description Specification. Curly brackets denote repetitions, and angle brakets contain textual description.

shows the grammar for TDS where each entry derives from the `Tactic` class which allows for the specification of the pattern using TC syntax and a list of builders. TDS supports five different builders: reshape, transpose, matmul, matvec and convolution, each mapping to a high-level operation (i.e., linalg.matmul). All builders support multiple inputs, except `transpose` and `reshape`, which only process a single input. All builders produce a single output. The `transpose` and `reshape` builder require an affine expression as the third argument, specifying how the dimensions should be transposed or reshaped, respectively.

Figure 6 shows the grammar for the access and structural matchers using the extended Backus-Naur form. Each access matcher, placeholder and array placeholder, must belong to a context. The context orchestrates the matching by tracking the assignment of `mlir::Value` to placeholders. Matchers cannot be constructed if the context is not already instantiated—when the context goes out of scope, everything is freed. A placeholder is a result of calling `m_Placeholder`. Placeholders support operators overloading to match any affine expression of the form $k{*}\iota{+}c$, where $k$ and $c$ are coefficients from the pattern and $\iota$ defines the candidate. A placeholder list is an ordered collection of placeholders. The position of the placeholder in a placeholder list is implicitly inferred by its position in the arguments list. The position is taken into account during the matching. An array placeholder can be constructed by calling `m_ArrayPlaceholder`, and a list of placeholders can be assigned to it. Similarly, structural matchers can be constructed only after the context has been instantiated. Structural matchers can take a callback as an optional leading argument. A callback

⟨*AccessPatternContext*⟩ ::= /*res of calling AccessCtx()*/
⟨*StructuralPatternContext*⟩ ::= /*res of calling NestedPatternCtx()*/
⟨*placeholder*⟩ ::= /*res of calling m_Placeholder()*/
⟨*placeholder-list*⟩ ::= ⟨*placeholder*⟩[{',' ⟨*placeholder*⟩}]
⟨*array-placeholder*⟩ ::= /*res of calling m_ArrayPlaceholder()*/
⟨*array-placeholder*⟩ ::= ⟨*array-placeholder*⟩, ⟨*placeholder-list*⟩
⟨*load-matcher*⟩ ::= m_Op⟨*LoadOp*⟩(⟨*placeholder-list*⟩)
⟨*load-matcher*⟩ ::= m_Op⟨*LoadOp*⟩(⟨*array-placeholder*⟩)
⟨*store-matcher*⟩ ::= m_Op⟨*StoreOp*⟩(⟨*placeholder-list*⟩)
⟨*store-matcher*⟩ ::= m_Op⟨*StoreOp*⟩(⟨*array-placeholder*⟩)
⟨*void*⟩ ::= m_Capt(⟨*Value &*⟩)
⟨*StructuralMatcher*⟩ ::= ⟨*StructuralMatcher-type*⟩(⟨*StructuralMatcher-list*⟩)
| ⟨*StructuralMatcher-type*⟩([⟨*callback*⟩],⟨*StructuralMatcher-list*⟩)
⟨*node-type*⟩ ::= 'FOR' | 'IF'

Fig. 6: EBNF syntax for access and structural matchers.

| CPU | Clock rate | OS | RAM | L1/L2/L3 |
|---|---|---|---|---|
| Intel-i9-9900K | 3.6 GHz | Ubuntu 18.04 | 64 GB | 32/256/16384 KB |
| AMD 2920X | 4.3 GHz | Ubuntu 18.04 | 64 GB | 1.125/6/32 MB |

TABLE I: Experimental setup.

allows for finer-grained control over the matching by testing i.e., access pattern properties.

## V. Evaluation

In this section, we illustrate our framework's applicability for two raising paths, Affine-to-Affine and Affine-to-Linalg. In the former, we show how Multi-Level Tactics is capable of lifting the level of abstraction within the Affine dialect and improve the performance of computations involving generalized matrix-matrix multiplication. In the latter, Multi-Level Tactics is used to raise loop nests in the Affine dialect to high-level operations from the Linalg dialect. At the Linalg level, Multi-Level Tactics emits Blas calls or relies on the Linalg code generator path.

Finally, we show how Multi-Level Tactics leverages further high-level optimizations detecting and subsequently optimizing matrix chain multiplications. Figure 7 re-proposes (a simplified) MLIR lowering pipeline extended with the three raising paths enabled by Multi-Level Tactics, and discussed in the next sections.

The experiments (Table I) have been conducted on two test platforms: an Intel Core i9-9900K (Coffee Lake) and an AMD Threadripper 2920X. All results were obtained considering the minimal execution time of five independent runs for single-precision operands.

### A. Raising from Affine loop nests to Affine high-level operations

Generalized matrix-multiplication (GEMM) is a frequently occurring pattern with decades of research on its optimization for various architectures [12]. A recent improvement within MLIR [13] introduced a custom high-level operation `matmul` in the Affine dialect that lowers to high-performance code by implementing the OpenBLAS/Blis optimization [14].

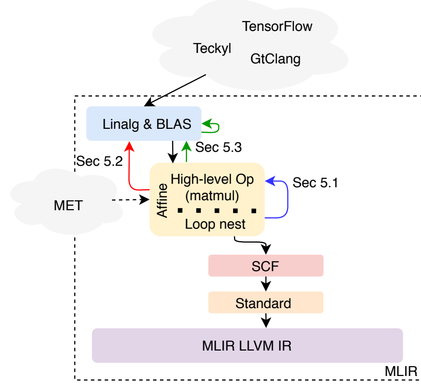


Fig. 7: Multi-Level Tactics raising paths.

```
def GEMM {
    pattern = builder
    C(i,j) += A(i,k) * B(k,j)
}
```

Listing 8: TDL to raise to a matmul. The user can raise by specifing `-raise-affine-to-affine`.

Currently, the operation needs to be instantiated directly in manual code generation or through modification of an existing high-level frontend, such as the Teckyl frontend for tensor computations [15]. However, while this may be conceivable for specialized tools and specific use cases, the implementation requires detailed knowledge of MLIR internals, is time-consuming, and needs to be repeated for each high-level operation and all IR entry points.

Multi-Level Tactics solves these issues by providing a convenient way for the user to express high-level patterns, such as GEMM, and to automatically locate and replace these at the affine level. Listing 8 shows the user-defined tactics necessary to detect a contraction $\mathrm{E} = (\times, +)$ of the form $C \rightarrow C(i,j) + A(i,k) \times B(k,j)$, where $A$, $B$, and $C$ are matrices. Once defined, the tactic is applied to all loop nests with a GEMM-like access pattern and replaces them with the `matmul` operation.

We test the reliability of our tactic on semantically equivalent GEMM kernels from Polybench 4.2 and Darknet written C in different styles. Polybench uses multi-dimensional arrays references to encode multi-dimensions array accesses. Contrary, Darknet—a widely used, open-source deep learning framework—uses linearized array references [16]. In all cases, before running Multi-Level Tactics, we perform loop distribution via MET to isolate the GEMM kernels from other statements. Figure 8 shows the number of callsites detected by Multi-Level Tactics compared with an Oracle, representing a perfect matching. The GEMM kernels from Darknet are missed, since the linearized, 1-d accesses are not matched by the 2-d array references emitted by the GEMM tactic in Listing 8. A delinearization pass in MLIR, as done in the LLVM polyhedral optimizer [17], can solve this issue.

As the quality of the OpenBLIS/BLAS transformation has

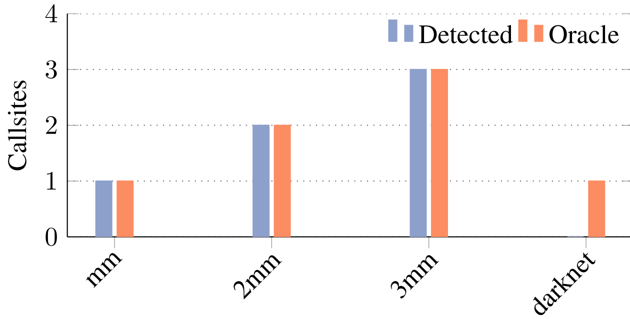


Fig. 8: Number of callsites detected by Multi-Level Tactics compared to perfect matching (Oracle).

already been demonstrated in the original paper, we only report the result of detecting a single 2088x2048 SGEMM multiplication on the AMD system. Compilation of a sequential loop nests implementing a GEMM operation with `clang -O3` (release 6.0.0) achieves a baseline performance of $1.76\,\mathrm{GFLOPS/s}$, raising to `matmul` with Multi-Level Tactics and subsequent optimization with OpenBLAS/Blis yields $23.59\,\mathrm{GFLOPS/s}$, corresponding to a $13.4\times$ speedup.

### *B. Raising from Affine to Linalg*

The lifting from loop nests to `matmul` operations improves performance significantly but remains specialized to GEMM operations and the specific OpenBLAS/Blis optimization. In this section, we generalize the lifting with Multi-Level Tactics from loop nests to the Linalg dialect, covering matrix multiplications, matrix-vector products, convolutions and TTGT conversions of tensor contractions to matrix products.

We first evaluate a single-step lifting scheme that replaces loop nests with Linalg operations and subsequently lowers these operations using the default Linalg lowering path (`MLT-Linalg`). This leverages optimizations already implemented for Linalg (e.g., tiling for caches)[2]. We then present a two-step scheme, which first raises to Linalg and then invokes a second pass replacing linalg operations with calls to a vendor-optimized BLAS library (`MLT-Blas`).

As input programs for the evaluation, we use a set of linear algebra benchmarks from the Polybench 4.2 suite and collected from previous studies related to tensor contractions [19]. For the contractions, we include tensors with different dimensionality from relevant domains used in coupled-cluster methods [20] and chemistry kernels [21]. For Polybench 4.2, we selected only those benchmarks that can be mapped to current available Linalg operations, leading to the exclusion of `syrk`, `symm`, `syr2k`, `trmm`, and `doitgen`.

Figure 9 shows the performance results for each of the selected benchmarks:

- `Clang -O3` refers to compilation of the sequential C code with Clang (release 6.0.0)
- `MLT-Linalg` refers to raising to Linalg using Multi-Level Tactics and subsequent lowering using the default scheme. `MLT-Blas`, on the other hand, replaces Linalg operations with calls to a BLAS library.
- `Pluto-default` refers to source-to-source compilation using Pluto[3] with a tiling factor of 32 along each dimension and the default *smartfuse* fusion heuristic, which attempts to balance locality and parallelism. We lower Pluto's optimized code using Clang.
- `Pluto-best` is the best result for Pluto from over $3,000$ combinations of tile sizes from 1 to $\frac{1}{4}$-th of the problem size and the available fusion heuristics (maximum fusion, no fusion, and smartfusion).

The horizontal lines at $145.5\,\mathrm{GFLOPS/s}$ and $63.6\,\mathrm{GFLOPS/s}$ indicate the performance for a single-precision matrix multiplication ($2048 \times 2048$) of the Intel Math Kernel Library for Deep Neural Network (MKL-DNN) on the respective system. We use the MKL-DNN also for the AMD system, as the performance gap between the MKL-DNN and OpenBLAS is less than 3% (OpenBLAS: $65.9\,\mathrm{GFLOPS/s}$, MKL-DNN: $63.6\,\mathrm{GFLOPS/s}$).

As expected, for both architectures `Clang` provides the lowest performance due to the low level of abstraction and the broad optimization strategy of a general-purpose compiler. Pluto with the default settings, generally outperforms Clang but it is not able to match Multi-Level Tactics with the default Linalg lowering path. For `atax`, `bicg`, `mvt`, `gemver` and `gesummv`, `Pluto-default` and `Pluto-best` yield code that is as fast or faster than Multi-Level Tactics substituting BLAS operations with calls to the MKL-DNN. The best settings for Pluto significantly increase performance over the default, but fail to match BLAS performance for `2mm`, `3mm`, `gemm`, `conv2d-nchw` and the contractions [4].

When comparing `MLT-Blas` with `Pluto-best` for the AMD the largest speedups are observed for kernels where Multi-Level Tactics maps to level-3 BLAS (`2mm` to `abcd-aebf-fdec`). The highest speedup is for `ab-cad-dcb` ($294\times$) while the lowest speedup can be observed for `gemm` ($2.3\times$). Considering kernels which map to level-2 BLAS (`atax` to `gesummv`), `Pluto-best` obtains better performance than `MLT-Blas`. This loss in optimization opportunity is the result of additional overhead introduced by Multi-Level Tactics (and MLIR) to link vendor-optimized libraries dynamically. As an example, neglecting the constant overhead of $1.5\,\mathrm{ms}$ for the `atax` kernel, its performance would be on-par with `Pluto-best` at $6.5\,\mathrm{GFLOPS/s}$. A similar observation can be made on the Intel system. For level-3 BLAS kernels, the performance of `MLT-BLAS` exceeds `Pluto-Best`, with the highest speedup for `ab-acd-dbc` ($66\times$) and the lowest speedup for `gemm` ($3.78\times$).

When comparing `MLT-Linalg` with `Pluto-default` for the AMD system, `Pluto-default` gives slightly better

[2]As of git version 48c28d5, Linalg primarily performs tiling, but work is in progress to have more competitive performance with Blas libraries [18].

[3]git commit f62d61b8

[4]Contractions are accelerated by rewriting the pattern using the TTGT transformation and invoking GEMM, transpose, and reshape routines available in MKL-DNN or the Linalg dialect.

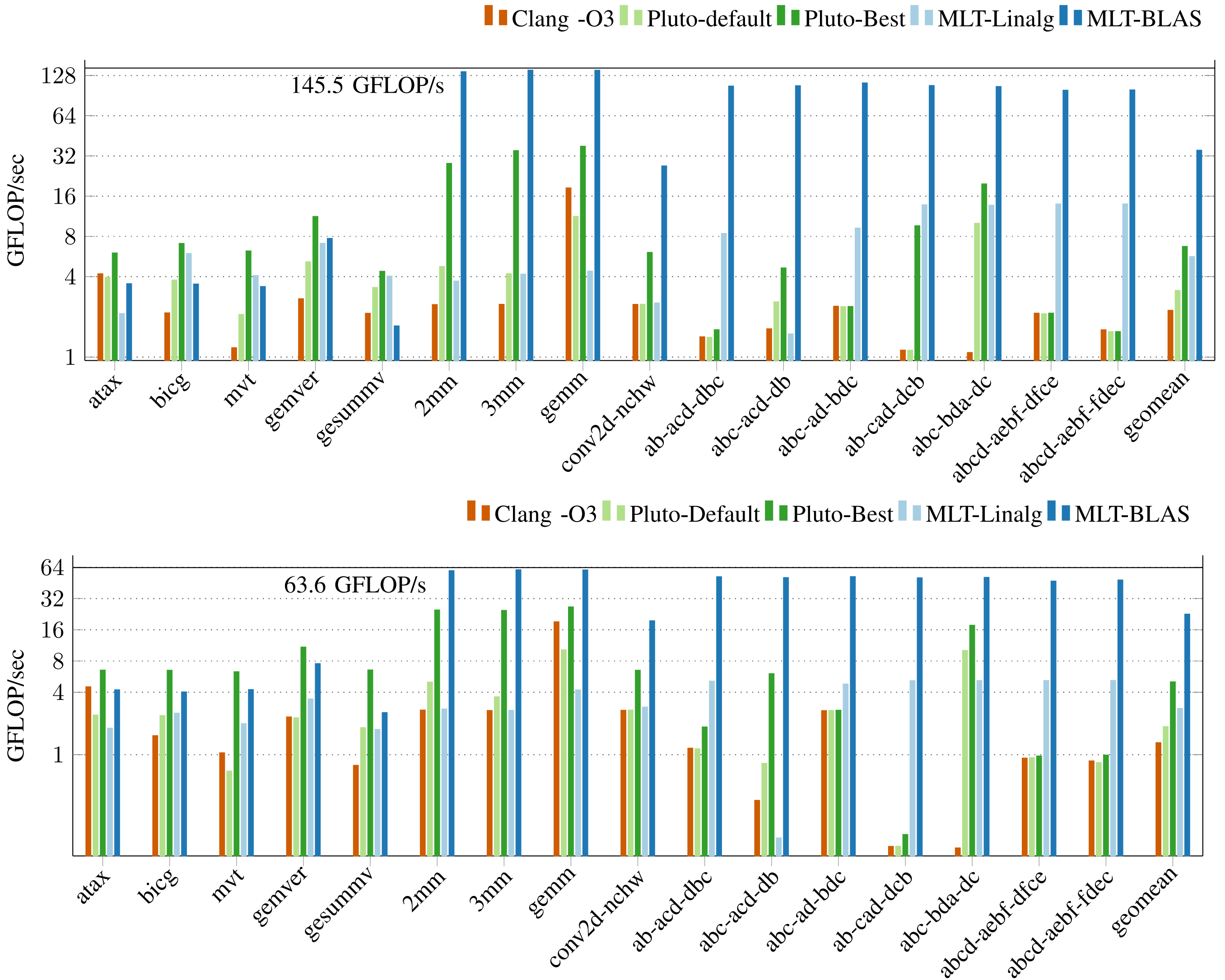


Fig. 9: Performance obtained for single-precision operands for two different architectures. Multi-Level Tactics allows recovering semantic information in general-purpose code and exploit domain-specific optimizations by lifting to the Linalg dialect. At the Linalg abstraction, we follow the Linalg code generation path or emit calls to vendor-optimized libraries directly. Results for the Intel i9 are shown on top, and for the AMD 2920X at the bottom.

performance for all level-3 BLAS kernels except for the contractions. This is expected, as Pluto's smartfuse heuristic applied on top of the tiling transformations significantly reduces control-flow overhead, while for the contractions, the TTGT transformation allows for significant improvements in data locality for `MLT-Linalg`. An exception is `abc-acd-db` and `abc-bda-dc`, where in the latter Pluto vectorizes the innermost loop, resulting in better performance. Similar behavior is observed for the Intel system, where `Pluto-default` performs better on all kernels except for contractions, exception made for `abc-acd-db`.

Finally, we evaluate the compile-time overhead introduced by Multi-Level Tactics. Lowering the 16 benchmarks considered above from Affine to MLIR LLVM takes 0.64 s with an unmodified MLIR version[5] compiled in release mode. In comparison, it takes 0.72 s for Multi-Level Tactics to lift from Affine to Linalg and then lower to MLIR LLVM, which represents an increase of only 12% of the compilation time.

### *C. A case for progressive raising: reordering matrix chain multiplications*

We have shown how Multi-Level Tactics can be employed to implement a single-step raising procedure, e.g., from Affine to Linalg. As an illustration for progressive raising using Multi-Level Tactics, we present an optimization reducing the number of operations in matrix chain multiplications exploiting associativity. This problem has multiple applications

[5]git commit 48c28d5

```
using namespace Linalg
auto A, B, C, D;
auto OutMatMul1, OutMatMul2, OutMatMul3;
auto _chain =
  m_Op<MatmulOp>(m_Capt(A), m_Capt(B),
    m_Op<MatmulOp>(OutMatMul1, m_Capt(C),
      m_Op<MatmulOp>(OutMatMul2, m_Capt(D), OutMatMul3)))
```

Listing 9: Structural matcher to detect a chain of 3 matrices multiplications in the Linalg dialect.

in real-world problems spanning from robotics to computer animation [22], [23] and is formulated as follows: Given a product of $n$ matrices of the form $A_1 \times A_2 \times ... \times A_n$ of sizes $p_{i-1} \times p_i$, the matrix-chain optimization problem consists in finding the optimal parenthesizations that minimize the number of scalar multiplications [24].

For example, consider the product of three matrices $A_1$, $A_2$ and $A_3$ with sizes $800 \times 1100$, $1100 \times 1200$ and $1200 \times 100$, the parenthesization $(A_1 \times A_2) \times A_3$ results in $1.152 \cdot 10^9$ multiplications, while the parenthesization $A_1 \times (A_2 \times A_3)$ obtained by the optimization requires only $2.2 \cdot 10^8$ multiplications.

As a starting point, we consider a matrix-chain multiplication expressed as a set of nested loops in C and use MET to enter the MLIR compilation pipeline at the Affine dialect. We then use the tactic shown in Listing 8 with the compilation flag `-raise-affine-to-linalg` to raise from the Affine loop-based abstraction to Linalg. To detect chains of matrix multiplications, we use a set of rewriting rules based on our `m_Op` matcher. Listing 9 shows an example for chains of three matrices. The input matrices $A$, $B$, and $C$ and the final result $D$ are captured via `m_Capt` to enable the builder to generate the re-parenthesized expression with the minimal number of scalar multiplications.

We evaluate the impact of the above transformation on the AMD system on three different matrix-chain multiplications taken from the literature [22]. Table II reports the sizes of the various matrices we consider as well as the initial and optimal parenthesizations. We additionally report the execution time for the optimized (time OP) and naive (time IP) parenthesizations. In all cases, the reduction in the scalar multiplication is reflected by faster execution time. For example, if we consider the chain with four matrices, the original order's execution takes $1.289\,\mathrm{s}$ ($2.37\,\mathrm{GFLOPS/s}$). In contrast, the new order only requires $0.212\,\mathrm{s}$ for completion, corresponding to a speedup that is proportional to the reduced scalar operations of $6.08\times$.

In conclusion, the experiments presented in this section illustrate real-world examples of how Multi-Level Tactics can be used to raise from lower-level representations of general-purpose languages to domain-specific abstractions without user intervention. The concise notations allow for quick prototyping and implementation of raising procedures, leveraging high-level transformations with significant performance improvements that general-purpose compilers fail to apply due to the low level of abstraction and the generic optimizations.

## VI. Related Work

Idiom recognition is an old and well-known problem in computer science since the 1990s. Each previous work can be broadly classified in one of the following categories: text, syntactic, and semantic. Text-based tools operate directly on the source code while syntactic ones at the AST level [25], [26], [27]. On the other hand, semantic tools go one step below and annotate the AST with data and control flow information. The first two categories have fallen out of fashion, mainly due to the weak robustness against code changes, leaving the stage for the more systematic semantic approaches. In this category, Ginsbach et al. propose IDL an Idiom Description Language that gets lowered to a set of constraints [28]. A match is a code fragment that adheres to the set of specifications. Their approach is fully automated and implemented in the LLVM compiler. Compared with Multi-Level Tactics, IDL has the advantages of detecting spare-linear algebra, which is not yet supported in MLIR and Multi-Level Tactics. But relying on a constraints solver to discover idioms increases the compilation time by a large margin. In the paper, they report an increase in compilation time of 82% on average. Contrary, as demonstrated in the evaluation, Multi-Level Tactics has a negligible overhead. In a follow-up work, Ginsbach et al. propose LiLAC, a language and a compiler for accelerating sparse and dense linear algebra [29]. Idiom discovery is still based on a constraints solver, but the pattern specification is made easier by introducing a DSL. Similarly, to TDL their DSL enables the specification of the "what" and the "how". The "what" defines what to match while the "how" how the library should be invoked to accelerate the "what". LiLAC can accelerate spare linear algebra but each pattern maps to a single BLAS call. Vice-versa, Multi-Level Tactics enables expressing a pattern as a composition of library calls but does not support sparse linear algebra. Arenaz et al. with the XARK compiler enable idiom recognition by analyzing use-def chains in Strongly Connected Components (SCCs) on the Gate Single Assignment Form, an extension of the popular SSA form [30]. However, the linearization of multi-dimensional arrays fundamentally limits the complexity of the patterns that can be detected. Contrary, Multi-Level Tactics works within the MLIR infrastructure, which enables whenever possible (i.e., the access is not already linearized) to treat multi-dimensional accesses as first-class citizens. Chelini et al. with Declarative Loop Tactics bring domain-specific optimizations in general-purpose flow by providing compiler developers with a tool to add highly customized optimizations for a given computation motif [1]. Despite Multi-Level Tactics shares a lot of commonality with Loop Tactics, it goes one step further by lowering the barrier of writing matchers and boosting productivity by enabling their automatic synthesis via TableGen. Felleisen et al. introduce Racket, a language extension API, to extend the host language's syntax and semantics [31]. Thus enabling programmers to embed context sensitive-information for optimization purpose. While MLT has some similarity with Racket, and in more general with

| N | Matrix Dimensions | Initial Parenthesization (IP) | Optimal Parenthesization (OP) | Time IP | Time OP | Speedup |
|---|---|---|---|---|---|---|
| 4 | 800 1100 900 1200 100 | $(((A_1 \times A_2) \times A_3) \times A_4)$ | $(A_1 \times (A_2 \times (A_3 \times A_4)))$ | 1.289 s | 0.212 s | 6.08X |
| 5 | 1000 2000 900 1500 600 800 | $((((A_1 \times A_2) \times A_3) \times A_4) \times A_5)$ | $((A_1 \times (A_2 \times (A_3 \times A_4))) \times A_5)$ | 5.850 s | 2.567 s | 2.27X |
| 6 | 1500 400 2000 2200 600 1400 1000 | $(((((A_1 \times A_2) \times A_3) \times A_4) \times A_5) \times A_6)$ | $(A_1 \times ((((A_2 \times A_3) \times A_4) \times A_5) \times A_6))$ | 28.490 s | 7.762 s | 3.67X |

TABLE II: Multi-Level Tactics enables the matrix-chain multiplication optimization at Linalg level by providing a raising path from source code written in C.

language-oriented programming framework, it also a significant difference [32], [33]. Specifically, MLT focus on the IR level and not the source level, benefiting compiler developers, not application developers. Brown et al. in the Delite framework use rewriting rules to apply domain-specific optimizations, while our end goal is similar (i.e., applying domain-specific optimizations), Multi-Level Tactics does that by *raising* from low-level abstractions [5]. Frameworks such as Halide decouple the function description of a problem from its execution strategy [34], [35]. Generally, the former is written by a domain expert while the latter by a performance one. Such tools require domain experts to get acquainted with the tooling syntax (i.e., Halide syntax) to write the problem's function description. On the other hand, using Multi-Level Tactics, a domain expert can write plain C++ code, thus lowering the language barrier–no need to learn a new language assuming the expert already knows C++. In parallel or even before, thanks to the decoupled nature of matchers and builders, a performance expert can provide a tactic to decide the best execution strategy.

## VII. Conclusion and Future Work

We presented Multi-Level Tactics and its implementation in the MLIR framework. At the heart of this paper is the idea of enabling progressive raising—a complementary path to the progressive lowering offered by multi-level IR compilers. Our progressive raising enables us to lift the entry point of general-purpose languages, thus enabling domain-specific optimizations in a multi-level IR.

We show how Multi-Level Tactics allows expressing patterns and builders concisely based on a Tensor Comprehension inspired syntax, and demonstrate our framework for two raising paths: Affine to Affine or Affine to Linalg. Besides, we show a case for progressive raising by implementing a transformation on top of the Linalg dialect by reordering chains of matrix multiplications. To the best of our knowledge, we are the first to provide an infrastructure and a demonstration to achieve progressive raising in a multi-level IR compiler like MLIR. Shortly, we will provide more raising paths.

## Acknowledgment

The research of Lorenzo Chelini and Andi Drebes is partially supported by the European Commission Horizon 2020 programme through the NeMeCo grant agreement, id. 676240, and the MNEMOSENE grant agreement, id 780215. The research of Tobias Grosser is partially supported through the Swiss National Science Foundation under the Ambizione programme (grant PZ00P2168016) and ARM Ltd. and Xilinx Inc., in the context of Polly Labs.

## References

[1] L. Chelini, O. Zinenko, T. Grosser, and H. Corporaal, "Declarative loop tactics for domain-specific optimization," *ACM Trans. Archit. Code Optim.*, vol. 16, Dec. 2019.

[2] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically," *ACM Trans. Archit. Code Optim.*, vol. 16, Oct. 2019.

[3] A. Hartono, B. Norris, and P. Sadayappan, "Annotation-based empirical performance tuning using orio," in *2009 IEEE International Symposium on Parallel Distributed Processing*, pp. 1–11, 2009.

[4] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, (New York, NY, USA), p. 519–530, Association for Computing Machinery, 2013.

[5] K. J. Brown, A. K. Sujeeth, H. J. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "A heterogeneous parallel framework for domain-specific languages," in *2011 International Conference on Parallel Architectures and Compilation Techniques*, pp. 89–100, 2011.

[6] A. K. Sujeeth, T. Rompf, K. J. Brown, H. Lee, H. Chafi, V. Popic, M. Wu, A. Prokopec, V. Jovanovic, M. Odersky, and K. Olukotun, "Composition and reuse with compiled domain-specific languages," in *Proceedings of the 27th European Conference on Object-Oriented Programming*, ECOOP'13, (Berlin, Heidelberg), p. 52–78, Springer-Verlag, 2013.

[7] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling Compiler Infrastructure for Domain Specific Computation," CGO'21, p. to appear.

[8] T. Gysi, C. Müller, O. Zinenko, S. Herhut, E. Davis, T. Wicky, O. Fuhrer, T. Hoefler, and T. Grosser, "Domain-Specific Multi-Level IR Rewriting for GPU," *arXiv preprint arXiv:2005.13014*, 2020.

[9] LLVM Developers, "Tablegen overview." https://llvm.org/docs/TableGen/. [Online; accessed 04-01-2021].

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

[11] M. Team, "Embedded domain specific constructs - EDSC." https://mlir.llvm.org/docs/EDSC/. [Online; accessed 01-09-2020].

[12] R. Gareev, T. Grosser, and M. Kruse, "High-performance generalized tensor operations: A compiler-oriented approach," *ACM Trans. Archit. Code Optim.*, vol. 15, Sept. 2018.

[13] U. Bondhugula, "High performance code generation in mlir: An early case study with gemm," *arXiv preprint arXiv:2003.00532*, 2020.

[14] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Trans. Math. Softw.*, vol. 43, Aug. 2016.

[15] A. Drebes, "Teckyl: An MLIR frontend for Tensor Operations." https://github.com/andidr/teckyl. [Online; accessed 19-06-2020].

[16] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020.

[17] T. Grosser, J. Ramanujam, L.-N. Pouchet, P. Sadayappan, and S. Pop, "Optimistic delinearization of parametrically sized arrays," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, (New York, NY, USA), p. 351–360, Association for Computing Machinery, 2015.

[18] N. Vasilache, "Progress on codegen with the vector dialect." https://drive.google.com/drive/folders/1lLhWopx_WCtFq3gTDGVJEzV9hFD7dwmI. [Online; accessed 20-08-2020].

[19] P. Springer and P. Bientinesi, "Design of a high-performance gemm-like tensor–tensor multiplication," *ACM Trans. Math. Softw.*, vol. 44, Jan. 2018.

[20] K. Stock, T. Henretty, I. Murugandi, P. Sadayappan, and R. Harrison, "Model-driven simd code generation for a multi-resolution tensor kernel," in *2011 IEEE International Parallel Distributed Processing Symposium*, pp. 1058–1067, 2011.

[21] G. Baumgartner, A. Auer, D. E. Bernholdt, A. Bibireata, V. Choppella, D. Cociorva, Xiaoyang Gao, R. J. Harrison, S. Hirata, S. Krishnamoorthy, S. Krishnan, Chi-chung Lam, Qingda Lu, M. Nooijen, R. M. Pitzer, J. Ramanujam, P. Sadayappan, and A. Sibiryakov, "Synthesis of high-performance parallel programs for a class of ab initio quantum chemistry models," *Proceedings of the IEEE*, vol. 93, no. 2, pp. 276–292, 2005.

[22] B. B. Mabrouk, H. Hasni, and Z. Mahjoub, "Performance evaluation of a parallel dynamic programming algorithm for solving the matrix chain product problem," in *2014 IEEE/ACS 11th International Conference on Computer Systems and Applications (AICCSA)*, pp. 109–116, IEEE, 2014.

[23] S. S. Godbole, "On efficient computation of matrix chain products," *IEEE Transactions on Computers*, vol. 100, no. 9, pp. 864–866, 1973.

[24] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms*. MIT press, 2009.

[25] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford, "Polaris: Improving the effectiveness of parallelizing compilers," in *Languages and Compilers for Parallel Computing* (K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, eds.), (Berlin, Heidelberg), pp. 141–154, Springer Berlin Heidelberg, 1995.

[26] B. Pottenger and R. Eigenmann, "Idiom recognition in the polaris parallelizing compiler," in *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, (New York, NY, USA), pp. 444–448, ACM, 1995.

[27] S.-I. Lee, T. A. Johnson, and R. Eigenmann, "Cetus – an extensible compiler infrastructure for source-to-source transformation," in *Languages and Compilers for Parallel Computing* (L. Rauchwerger, ed.), (Berlin, Heidelberg), pp. 539–553, Springer Berlin Heidelberg, 2004.

[28] P. Ginsbach, T. Remmelg, M. Steuwer, B. Bodin, C. Dubach, and M. F. P. O'Boyle, "Automatic matching of legacy code to heterogeneous apis: An idiomatic approach," *SIGPLAN Not.*, vol. 53, p. 139–153, Mar. 2018.

[29] P. Ginsbach, B. Collie, and M. F. P. O'Boyle, "Automatically harnessing sparse acceleration," in *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, (New York, NY, USA), p. 179–190, Association for Computing Machinery, 2020.

[30] M. Arenaz, J. Touriño, and R. Doallo, "XARK: An Extensible Framework for Automatic Recognition of Computational Kernels," *ACM Trans. Program. Lang. Syst.*, vol. 30, Oct. 2008.

[31] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, "A programmable programming language," *Commun. ACM*, vol. 61, p. 62–71, Feb. 2018.

[32] M. P. Ward, "Language-oriented programming," *Software - Concepts and Tools*, vol. 15, no. 4, pp. 147–161, 1994.

[33] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as libraries," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), p. 132–141, Association for Computing Machinery, 2011.

[34] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *Acm Sigplan Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[35] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, *et al.*, "TVM: An automated end-to-end optimizing compiler for deep learning," in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pp. 578–594, 2018.

## Artifact Appendix

### *A. Abstract*

The artifact's goal is to show how Multi-Level Tactics (MLT) lifts general-purpose languages to higher-abstractions to enable effective domain-specific compilation via progressive lowering. The artifact consists of a docker container with accompanying scripts to replicate figure 8, 9 and Table 2. The docker container is the only piece needed to run all the experiments. Scripts to generate the figures and the table come with the docker.

### *B. Artifact check-list (meta-information)*

- **Algorithm:** Multi-Level Tactics a declarative approach for progressive raising implemented on top of the MLIR framework.
- **Program:** Polybench/C 4.2.1 beta and collected on previous studies on tensor contractions. Besides, we consider a 2-D convolution. For Polybench/C 4.2.1 we use a modified version where each kernel has been loop distributed and translated into the Affine dialect in MLIR. All the benchmarks used come with the MLT repository https://github.com/LoopTactics/mlir (cgo branch).
- **Compilation:** Any C++11-compatible compiler to bootstrap LLVM/MLIR.
- **Data set:** LARGE DATASET predefined in Polybench/C 4.2.1.
- **Run-time environment:** Any Unix system supported by LLVM.
- **Hardware:** Any platform supported by LLVM.
- **Output:** The result are PDF files replicating Figures 8, 9 and Table 2. The scripts are already in the docker container. Figure 8 and 9 express results using GFLOP/s while Table 2 using seconds. Intermediate files are also generated and are named result_X.txt. All the result_X.txt files contain results expressed in seconds.
- **How much disk space required (approximately)?:** The docker is 7GB, 15GB of disk space should be enough.
- **How much time is needed to prepare workflow (approximately)?:** Mainly the time to build MLT (more than 20 minutes).
- **How much time is needed to complete experiments (approximately)?:** 20/25 minutes.
- **Publicly available?:** Yes, via Github and Dockerhub.

### *C. Description*

*1) How delivered:*

- We deliver the artifact via docker. Available at: https://hub.docker.com/r/lchelini/cgo
- MLT and MLT's TDL DSL are available at: https://github.com/LoopTactics/mlir and https://github.com/LoopTactics/TacticsDSL
- A presentation of MLT at the MLIR Open Design Meeting is available here

*2) Hardware dependencies:* Any platform supported by LLVM, see https://llvm.org/docs/GettingStarted.html#hardware

*3) Software dependencies:* The docker container has all the dependencies, which are:

- All requirements needed to compile LLVM/MLIR see https://llvm.org/docs/GettingStarted.html#software
- MKL-DNNL available at https://github.com/chelini/mkl-dnn.git
- MKL libraries

For the MLT's TDL DSL we suggest using llvm-9.0, that can be installed using `sudo apt-get install llvm-9-dev` on your machine. No need to install it in the provided docker container.

### *D. Installation*

No installation required.

### *E. Experiment workflow*

The docker container comes with three files:

- experiment5.1.sh to reproduce Figure 8
- experiment5.2.sh to reproduce Figure 9
- experiment5.2.time.sh to reproduce the overhead introduce by MLT
- experiment5.3.sh to reproduce Table 2

Steps:

- `$ docker pull lchelini/cgo`
- `$ docker run -it lchelini/cgo`
- `$ ./build.sh`
- `$ ./experiment5.1.sh`
- `$ ./experiment5.2.sh`
- `$ ./experiment5.2.time.sh`
- `$ ./experiment5.3.sh`

After running `./experiment5.X.sh` a `main.pdf` with results can be found in `llvm-project/mlir/benchmark_section5.X`. To open the pdf file, copy it outside the container using `docker cp` command, see https://docs.docker.com/engine/reference/commandline/cp/. The script `./experiment5.2.time.sh` print on stdout, no file are generated.

### *F. Evaluation and expected result*

In experiment 5.1, we evaluate MLT's reliability by considering different flavours of GEMMs written in different styles but semantically equivalent. We expect to miss a raising opportunity only for the `Darknet` benchmark as we do not emit matchers for linearized access patterns, nor do we provide a delinearization pass.

In experiment 5.2, we demonstrate how lifting to higher abstractions (i.e., `MLT-Linalg` or `MLT-Blas`) allows us to get better performance than `Clang -O3`. We expect `MLT Linalg` and `MLT Blas` to reach higher GFLOP/s than the baseline `Clang -O3`. Besides, we provide also another baseline: `Pluto`. We expect `Pluto` to be better than `Clang -O3` but less effective (or comparable) to `MLT Linalg`. We expect `Pluto` to be less effective than `MLT BLAS`. Note that `Pluto-best` is not available as it relies on expensive autotuning and takes days to converge.

In experiment 5.3, we show a case for a more progressive raising by implementing the matrix-chain reordering transformation. We expect `Time IP` > `Time OP`. `IP` is the time for the matrix chain without reordering, while `OP` is the time of the reordered chain.