Progressive Raising in Multi-level IR
Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, Tobias Grosser, Henk Corporaal

To cite this version:
Lorenzo Chelini, Andi Drebes, Oleksandr Zinenko, Albert Cohen, Nicolas Vasilache, et al.. Progressive Raising in Multi-level IR. CGO 2021: International Symposium on Code Generation and Optimization, Feb 2021, Seoul / Virtual, South Korea. hal-03139764

HAL Id: hal-03139764
https://inria.hal.science/hal-03139764
Submitted on 12 Feb 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
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].

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.

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
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.

```mlir
%0 = linalg.matmul(%A, %B, %C) // linalg dialect

affine.for %k = 0 to %N  // affine dialect
  affine.for %j = 0 to %N {
    %0 = affine.load %C[i, k] : memref<?x?f32>
    %1 = affine.load %A[i, k] : memref<?x?f32>
    %2 = affine.load %B[k, j] : memref<?x?f32>
    %3 = std.mulf %1, %2
    %4 = std.addf %3, %0
    affine.store %4, %C[i, j] : memref<?x?f32>
  }
```

Listing 2: Contraction abc-acd-db.

```c
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];
```

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

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 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 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.

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 C code 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.

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 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) → (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) → (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<[], Expr<{0, 2, 1}>,>
    reshapeBuilder<In<[F], Out<[D], Expr<{0, 1, 2}>,>
    matmulBuilder<In<[F, B], Out<[E], Expr<{0, 1, 2}>,>
    reshapeBuilder<In<[A], Out<[F], Expr<{0, 1, 2}>,>
    matmulBuilder<In<[E], Out<[D], Expr<{0, 1, 2}>,>
    transposeBuilder<In<[D], Out<[C], Expr<{0, 1, 2}>,>
}>

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 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 \( k + c \), whereas \( c \) 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
Listing 6: Declarative access pattern matcher.

```cpp
// instantiate the context */
auto _a = m_Placeholder();
auto _A = m_ArrayPlaceholder();
auto matcher = m_Op<AddOp>(_A(_d, _b));

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

```cpp
// read out the matched value
...}
```

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.

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

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

IV. MULTI-LEVEL TACTIC SYNTAX

Figure 4 and 5 show 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\*i+c, where k and c are coefficients from the pattern and i 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
permits of calling 

<table>
<thead>
<tr>
<th>CPU</th>
<th>Clock rate</th>
<th>OS</th>
<th>RAM</th>
<th>L1/L2/L3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel i9-9900K</td>
<td>3.6 GHz</td>
<td>Ubuntu 18.04</td>
<td>64 GB</td>
<td>32/256/16384 KB</td>
</tr>
<tr>
<td>AMD 2920X</td>
<td>4.3 GHz</td>
<td>Ubuntu 18.04</td>
<td>64 GB</td>
<td>1.125/6/32 MB</td>
</tr>
</tbody>
</table>

**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 improving 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].

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 $E = (\times, +)$ 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
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 GFLOPS/s,
raising to matmul with Multi-Level Tactics and subsequent
optimization with OpenBLAS/Blis yields 23.59 GFLOPS/s,
corresponding to a 13.4× 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 re-
places 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 1\(\frac{1}{4}\)-th of the problem
size and the available fusion heuristics (maximum fusion,
no fusion, and smartfusion).

The horizontal lines at 145.5 GFLOPS/s and
63.6 GFLOPS/s indicate the performance for a single-
precision matrix multiplication (2048 × 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 GFLOPS/s, MKL-DNN: 63.6 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×) while the lowest speedup can be
observed for gemm (2.3×). 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 ms for the atax kernel, its performance
would be on-par with Pluto-best at 6.5 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×) and the lowest speedup for gemm (3.78×).

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 compensated 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
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 \( A_1 \times A_2 \times \ldots \times A_n \) of sizes \( p_{i-1} \times p_i \), the matrix-chain optimization problem consists in finding the optimal parentheses 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 \( 1.152 \cdot 10^8 \) 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 \texttt{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 \texttt{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 parentheses. We additionally report the execution time for the optimized (time OP) and naive (time IP) parentheses. 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 s (2.37 GFLOPS/s). In contrast, the new order only requires 0.212 s for completion, corresponding to a speedup that is proportional to the reduced scalar operations of 6.08×.

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.
The research of Lorenzo Chelini and Andi Dreibes 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 Grosner 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.

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 toolsing 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

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

<table>
<thead>
<tr>
<th>N</th>
<th>Matrix Dimensions</th>
<th>Initial Parenthesization (IP)</th>
<th>Optimal Parenthesization (OP)</th>
<th>Time IP</th>
<th>Time OP</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>800 1100 900 1200 100</td>
<td>(((((A₁ × A₂) × A₃) × A₄) × A₅) × A₆)</td>
<td>(A₁ × (A₂ × (A₃ × (A₄ × (A₅ × A₆))))</td>
<td>1289 s</td>
<td>0.212 s</td>
<td>6.08X</td>
</tr>
<tr>
<td>6</td>
<td>1500 4000 2000 2200 600 1400 1000</td>
<td>(((((A₁ × A₂) × A₃) × A₄) × A₅) × A₆)</td>
<td>(A₁ × ((A₂ × A₃) × (A₄ × (A₅ × A₆))))</td>
<td>28490 s</td>
<td>7.762 s</td>
<td>3.67X</td>
</tr>
</tbody>
</table>

### REFERENCES


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.