MultiLevel Tactics: Lifting loops in MLIR

Lorenzo Chelini[1] Andi Drebes[2] Oleksandr Zinenko[3] Albert Cohen[3]
Henk Corporaal[1] Tobias Grosser[4] Nicolas Vasilache[3]

[1]TU Eindhoven
[2]Inria and École Normale Supérieure
[3]Google
[4]ETH Zurich

January 12, 2020

Abstract
We propose MultiLevel Tactics, or ML Tactics for short, an extension to MLIR that recognizes patterns of high-level abstractions (e.g., linear algebra operations) in low-level dialects and replaces them with the corresponding operations of an appropriate high-level dialect. Our current prototype recognizes matrix multiplications in loop nests of the *Affine* dialect and lifts these to the *Linalg* dialect. The pattern recognition and replacement scheme are designed as reusable building blocks for transformations between arbitrary dialects and can be used to recognize commonly recurrent patterns in HPC applications.

ML Tactics
The MLIR infrastructure is a collection of modular and reusable software components for the progressive lowering of operations from high-level abstractions to a low-level IR. The compiler technology is non-opinionated: instead of providing a fixed set of abstraction levels and operations, it supports custom dialects with custom abstractions, operations and transformations as first-class citizens. This enables developers to build specialized compilers operating at an appropriate level of abstraction for their application or hardware targets.

However, the compilation pipeline currently works in a single direction: high-level operations can be transformed into operations with a lower level of abstraction, but low-level operations are never raised to high-level dialects. This means that the entry point into the compilation pipeline defines the highest level of abstraction for all subsequent transformations, potentially limiting the set of applicable optimizations. Aggressive transformations that rely on high-level information (e.g., calling BLAS operations) may remain inaccessible, and the performance of the generated code might remain behind its potential.

Merely relying on a specification of the source program is inconvenient, as this requires programmers to match the abstractions used internally by the compiler manually. It also excludes all source programs written in general-purpose languages (e.g., C/C++) which are not sufficiently expressive to preserve the required high-level information. To mitigate this problem, we propose an extension to MLIR based on the concepts of Loop Tactics [1] that allows for the implementation of transparent pattern-based transformations from low-level to high-level dialects. Our current prototype enables to lift from `Affine` to `Linalg` as shown in Figure 1 (red arrow).

The extension provides two kinds of matchers: structural matchers and access relation matchers. The former extend the NestedMatchers, already available in MLIR, to recognize specific loop structures (i.e., 2-dimensional or 3-dimensional affine loop nests), while the latter can be used to match specific access patterns using placeholders for induction variables and array accesses. An induction variable placeholder can be any affine expression $\omega = k * \iota + c$, where k is the coefficient, and c is the increment, whereas ι defines the bounded induction variable. A simple access matcher for a GEMM kernel is shown in Listing 1.

Conclusion We present a prototype that recognizes loop nests implementing a general matrix-matrix multiplication (GEMM) in the *Affine* dialect and lifts them to the *Linalg* dialect by replacing these occurrences with a `linalg.matmul` operation.

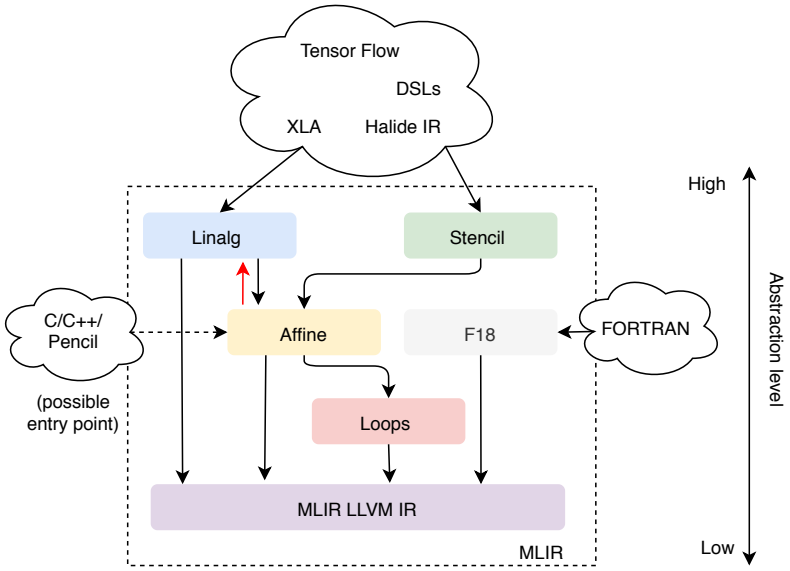

Figure 1: MLIR lowering pipeline with different dialects at different abstraction levels. General-purpose codes are not sufficiently expressive enough to enter the higher part of the lowering pipeline, excluding the possibility for more aggressive optimizations. ML Tactics provides the necessary infrastructure to lift from low-level dialects to high-level ones. Our current prototype lifts from `Affine` to `Linalg` (red arrow).

```
// bind placeholders to induction variables
// captured by the structural matcher.
auto _i = placeholder(/*ctx*/, m_SpecificVal(i));
auto _j = placeholder(/*ctx*/, m_SpecificVal(j));
auto _k = placeholder(/*ctx*/, m_SpecificVal(k));

// capture 'A', 'B' and 'C' necessary for
// the builder.
MemRefType A, B, C;
auto _A = arrayPlaceholder(m_Val(A));
auto _B = arrayPlaceholder(m_Val(B));
auto _C = arrayPlaceholder(m_Val(C));

// build GEMM access pattern.
auto a = m_Op<AffineLoadOp>(_A({_i, _k}));
auto b = m_Op<AffineLoadOp>(_B({_k, _j}));
auto c = m_Op<AffineLoadOp>(_C({_i, _j}));
auto gemm = m_Op<AddFOp>(c, m_Op<MulFOp>(a, b));
```

Listing 1: Access pattern matcher for a GEMM kernel. Placeholders must be assigned an induction variable captured by a structural matcher. On the other hand, array placeholders allow to capture the underneath MemRefType that will be used by the declarative builder (EDSC) available in MLIR to build the `linalg.matmul` operation.

Future Work Shortly we will extend ML Tactics to cover a higher number of commonly recurring linear-algebra patterns in HPC applications and liftback from `Affine` to `Linalg`. Now that the FORTRAN dialect is available (F18), we will study how the U.S. climate model [3] written in FORTRAN code can be lifted to Stencil to leverage high-level optimizatons [2].

References

[1] Lorenzo Chelini, Oleksandr Zinenko, Tobias Grosser, and Henk Corporaal. Declarative loop tactics for domain-specific optimization. *ACM Trans. Archit. Code Optim.*, 16(4), December 2019.

[2] Jean-Michel Gorius. Modeling stencils in a multi-level intermediate representation. 2019.

[3] William M Putman and Shian-Jiann Lin. Finite-volume transport on various cubed-sphere grids. *Journal of Computational Physics*, 227(1):55–78, 2007.