EINDHOVEN UNIVERSITY OF TECHNOLOGY

PET-to-MLIR

Citation for published version (APA):
Komisarczyk, K., Chelini, L., Vadivel, K., Jordans, R., & Corporaal, H. (2020). PET-to-MLIR: A polyhedral front-end for MLIR. In A. Trost, A. Zemva, & A. Skavhaug (Eds.), *2020 23rd Euromicro Conference on Digital System Design (DSD)* (pp. 551-556). Article 9217876 Institute of Electrical and Electronics Engineers. https://doi.org/10.1109/DSD51259.2020.00091

Document license:
CC BY-NC-ND

DOI:
10.1109/DSD51259.2020.00091

Document status and date:
Published: 08/10/2020

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 17. Nov. 2023

PET-to-MLIR: A polyhedral front-end for MLIR

Konrad Komisarczyk, Lorenzo Chelini, Kanishkan Vadivel, Roel Jordans, Henk Corporaal
Eindhoven University of Technology

***Abstract*—We present PET-to-MLIR, a new tool to enter the MLIR compiler framework from C source. The tool is based on the popular PET and ISL libraries for extracting and manipulating quasi-affine sets and relations, and Loop Tactics, a declarative optimizer. The use of PET brings advanced diagnosis and full support for C by relying on the Clang parser. ISL allows easy manipulation of the polyhedral representation and efficient code generation. Loop Tactics, on the other hand, enable us to detect computational motifs transparently and lift the entry point in MLIR, thus enabling domain-specific optimizations in general-purpose code.**

We demonstrate our tool using the Polybench/C benchmark suite and show that it can lower most of the benchmarks to the MLIR's affine dialect successfully. We believe that our tool can benefit research in the compiler community by providing an automatic way to translate C code to the MLIR affine dialect.

***Index Terms*—MLIR, Loop Tactics, polyhedral model**

I. Introduction and Motivation

MLIR is a novel and promising approach to build reusable and extensible compiler infrastructures [1]. The design rationale behind MLIR is to provide a non-optioned and fully customizable IR, which allows reducing the cost of building domain-specific compilers and significantly simplify the compilation for heterogeneous hardware. Having a fully customizable IR enables the representation of multiple abstraction levels. High-level IR representations can be lowered to low-level IR by progressive lowering. Dialects in MLIR enable IR extensibility and thus the representation of numerous abstraction levels. Specifically, you can think of a dialect as a namespace of operations, types and attributes. One of the available dialects in MLIR is the affine dialect. The affine dialect is a simplified polyhedral representation that has already been proven to be a powerful abstraction to generate high-performance code [2].

While some front-ends already exist to enter the high-abstraction levels in MLIR and then progressively lower them to affine, to the best of our knowledge, a C front-end is not available yet. As a consequence, developers need to lower C code manually if they want to experiment with the affine dialect and, more broadly, with the MLIR compiler infrastructure, which is a considerable loss in productivity. We advocate that an automatic tool that translates C code automatically would be a preferred solution. In this work, we propose PET-to-MLIR a front-end for a subset of C code based on state-of-the-art polyhedral technology. Besides, using Loop Tactics [3] we show how we can lift the entry point of C code, thus exploiting more aggressive domain-specific optimizations available in MLIR (i.e. replacing code with calls to optimized vendor-libraries). To summarize, our main contributions are:

- An automatic tool to enter the MLIR compiler framework at the affine dialect starting from a polyhedral-friendly C code.
- A demonstration of the usefulness of our tool by translating most of the Polybench/C benchmarks to MLIR's affine dialect.

In the next sections, we demonstrate the code generation flow and the tooling that we used to build PET-to-MLIR. Finally, we show how PET-to-MLIR can translate most of the Polybench and highlight its current limitations and future research directions.

II. Background

Since PET-to-MLIR is based on PET, ISL, and Loop Tactics we first provide an high-level overview of the projects (Section II-A and Section II-B). Section II-C briefly introduces the MLIR compiler framework.

A. ISL and PET

PET is a library used to extract the polyhedral model starting from a C code fragment [4]. The extracted code fragment is called static-control part, or ScOP for short. PET is based on the LLVM C front-end (Clang) and ISL. The use of Clang gives PET full support for C99 and variable vector length arrays. Besides, Clang reports useful diagnosis messages that communicate with the user, which part of the input code does not satisfy the requirements of the polyhedral model. On the other hand, ISL allows extensive support for static piece-wise quasi-affine expressions and conditions [5]. Such broad support is not available in other polyhedral extractors such as Clan [6]. In short, PET utilizes Clang to obtain a high-level AST and generates a compact, ISL-based, polyhedral representation. On top of creating the expressions typical of the polyhedral model (i.e., schedule and access relations), PET introduces structures containing additional information about the source program (i.e., array names and array extent).

Within PET, individual statements are represented by the *iteration domain*, a set of *access relations*, and belong to a *schedule*. The iteration domain assigns to each statement a symbolic name and an integer vector in a k-dimensional space where k is the depth of the surrounding loops. Each point in such a vector represents a particular statement instance. For example, the iteration domain for the statement S1 in the GEMM kernel (Listing 1) is $\{S_1(i,j) \mid 0 \leq i,j < 1024; S_2(i,j,k) \mid 0 \leq i,j,k < 1024\}$.

Access relations are represented as piece-wise quasi-affine functions, which map the iteration space with the array space, whose coordinates are the values of the accessed

```
    for (int i = 0; i < 1024; ++i)
      for (int j = 0; j < 1024; ++j) {
S1:     C[i][j] = beta * C[i][j];
        for (int k = 0; k < 1024; ++k)
S2:       C[i][j] += alpha * A[i][k] * B[k][j];
      }
```

Listing 1: Generalized matrix multiplication (GEMM) kernel.

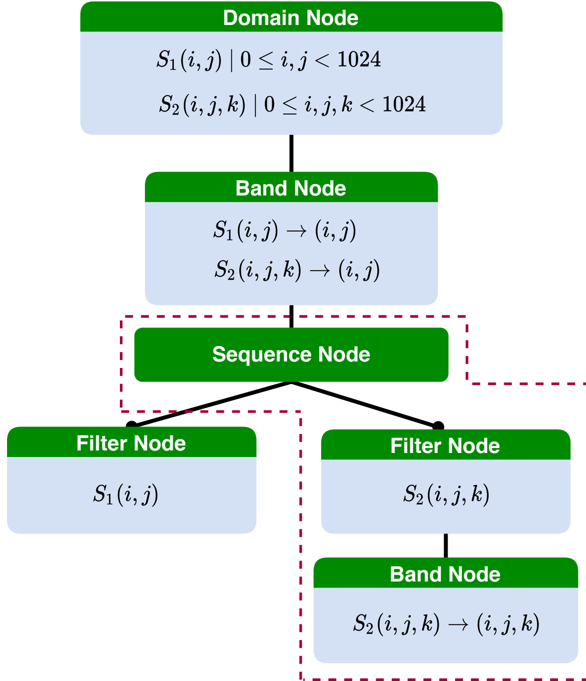

Fig. 1. Schedule tree representation for Listing 1. The subtree highlighted in the red-box is detected by the matchers in Listing 2.

subscripts. For the statement S1 in our running example, the accesses are described as: $\{S_1(i,j) \to \text{beta}(); S_1(i,j) \to \text{C}_{\text{read}}(i,j); S_1(i,j) \to \text{C}_{\text{write}}(i,j)\}$.

The order in which the statement instances are executed is defined by the schedule, which maps a point in the iteration space to a point in the time space. The schedule is represented as a tree [7], where each node represents a partial schedule, and the order of loops and statements is determined by the node parent-child relation. The root of the tree is always a domain node, which encodes the iteration domain. Below such node a combination of the following nodes may exist: 1) `band` which defines the partial schedule of one or multiple loops; 2) `filter` which restricts the statement instances of the iteration domain; 3) `sequence` which imposes an order among its children. Figure 1 illustrates the schedule tree for our running example. It uses an outer band node to encapsulate the partial schedule of loop `i` and `j`. The band is followed by a sequence node that establishes the order of executions between statement S_1 and S_2. Filter nodes ensure that each branch of the sequence node only executes the statement indicated in the filter node. The innermost band represents the partial schedule for the `k` loop.

B. Loop Tactics

Loop Tactics is an optimizer based on the ISL library supporting the declarative specification of affine transformations. Loop Tactics introduces three main concepts: 1) Schedule tree matchers which allow recognizing patterns in the schedule tree. 2) Access relation matchers that allow inspecting access pattern properties, and 3) builders that allow reconstructing a matched subtree, thus applying a given transformation.

Essentially, a schedule tree matcher replicates the node type-based structure of the schedule tree to match. It allows additional wildcarding (i.e., anyTree) and filtering (i.e., via C++ callback functions). Besides, it allows for capturing specific nodes that can be passed to the builders for optimization purposes. A builder uses a syntax similar to the tree matchers but describes how the tree should be reconstructed. Each tree modification reflects to a transformation in the original input code. Finally, access relation matchers allow testing access pattern properties via `placeholder` and `arrayPlaceholder`. In summary, a loop transformation can be declaratively expressed by specifying a matcher pattern that captures a set of nodes and a builder that rewrites the captured nodes to create a new subtree.

As an example, Listing 2 shows how the right-part of the subtree in Listing 1 can be matched. Lines 25 to 30 show the structural matcher. The matcher looks for a sequence node that has as descendant a band node that satisfies the callback `hasGemmPattern`, which in turn matches for a GEMM-specific access pattern. Such a pattern must have at least three two-dimensional reads to different arrays (line 12 to 15), one write access (line 17), and a permutation of indexes that satisfies the access pattern $[i,j] \to [i,k][k,j]$. Finally, the builder (line 33 to 38) rebuilds the subtree by splitting the band node into two nested bands, which reflects the tiling transformations. For each dimension i, j and k, we use a tile factor of 32.

C. MLIR

Increasingly heterogeneous and complex hardware makes the design of effective code generators difficult. Addressing this issue, the MLIR framework has been recently introduced under the LLVM umbrella. The motivation behind MLIR is to facilitates the design and implementation of code generators by significantly reducing the cost of building domain-specific compilers. MLIR is a non-opinionated, meaning that it comes with a limited set of builtins, leaving most of the intermediate representation customizable. A logical group of operations, types, and attributes make a dialect. The affine is one of them. Figure 2 shows the available dialects in MLIR and their entry points in its compilation pipeline. PET-to-MLIR enables polyhedral friendly C code fragments to enter the affine dialect.

III. A Bird's Eye View of PET-to-MLIR

Figure 3 shows the high-level view of our tool. We use PET to construct the polyhedral model from a given C code fragment. In the default mode, we require the user to delimit

```
auto hasGemmPattern = [&](schedule_node node) {
  auto _i = placeholder();
  auto _j = placeholder();
  auto _k = placeholder();
  auto _A = arrayPlaceholder();
  auto _B = arrayPlaceholder();
  auto _C = arrayPlaceholder();

  auto reads = /* get read accesses */;
  auto writes = /* get write accesses */;

  auto mRead = allOf(
    access(_C, _i, _j),
    access(_A, _i, _k),
    access(_B, _k, _j));

  auto mWrite = allOf(access(_C, _i, _j));

  return match(reads, mRead).size() == 1 &&
         match(writes, mWrite).size() == 1;
};

schedule_node body, continuation;

auto matcher =
  sequence(
    hasDescendant(
      filter(band
        (body, hasGemmPattern, // filter func.
          anyTree( // wildcard
          continuation)}())))));

auto builder =
  band([&]() { return
    tileSchedule(body, {32,32,32}); },
    band([&]() { return
        pointSchedule(body, {32,32,32}); },
      subtree(body)));
```

Listing 2: Structural matcher, access relation matcher for the right-most subtree in Listing 1.

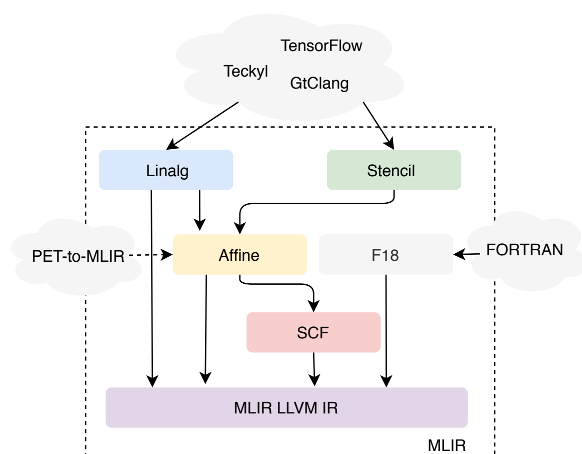

Fig. 2. Dialects available in MLIR and their entry points. PET-to-MLIR enables entering in the affine dialect from C code. The black arrows pointing downward represent MLIR's progressive lowering (i.e., from high-level of abstractions (Linalg) down to the lowest (LLVM-IR)).

the code to be extracted with pragmas. Specifically, `pragma scop` and `pragma endscop` must delimit a code fragment. If PET cannot model a construct in the scop (i.e., non-piecewise quasi-affine accesses), our tool will bail-out with a warning. We also support an autodetect mode, which in turn relies on the `--autodetect` options exposed by PET. In this case, PET will try to detect automatically a code fragment that fits the polyhedral model. In this case, no warnings are emitted if no scops are detected. The `-I` option can be used to pass include paths to PET.

Once the scop has been extracted, and before code generation, optionally, the user can invoke Loop Tactics to detect computational motifs automatically and lift the entry point in MLIR. Lifting the entry point will enable some domain-specific optimizations that are not available at the affine level. In the evaluation section, we will show how recovering domain-specific information on general-purpose code can boost the performance by emitting vendor-optimized routines.

For each scop, PET-to-MLIR emits an `mlir::FuncOp` with a void signature. The inputs of the function match with the input and output of the scop. To identify the input, we use the scop's array list. The array list keeps track, for each array reference, of the following information: 1) the extent (i.e., the size of the array), 2) element type (i.e., float or double). 3) The set of constraints on the array parameters to ensure that it has a valid size. 4) Two additional flags: declared and exposed. The former tells us if an array is declared within the scop. The latter if the array is visible outside the scop. If an array is marked as exposed, it will be inserted as an input parameter to the function. Whereas, if an array is marked as declared, it will be allocated and deallocated within the function.

A PET scop also contains a context and a list of statements with line locations. The context defines the parameter values for which the scop is executed. We do not allow any symbolic constant in the context, if any, the tool will bail-out with a warning. The list of statements keeps track of statement information. Specifically, each statement consists of a line number, a domain, a schedule, and a parse tree. The latter reflects the structure of the C statement. In the parse tree, each node corresponds to `pet_expr`. A `pet_expr` carries the type of operation, as well as, the arguments to be modeled. An argument can be an access to an array (i.e., a read or write access) or another `pet_expr`. PET-to-MLIR, builds each statement by recursively walking each parse tree, and creating the corresponding operation using the builders exposed by the affine dialect.

To emit control flow operations, PET-to-MLIR walks the ISL AST. For each node, PET-to-MLIR emits the corresponding operation in the affine dialect. In more details, for each `isl_ast_node_for` PET-to-MLIR emits an `Affine::ForOp`. Currently, we can handle loops that count upward, downward and triangular ones. The for loop needs to be in the form `for (int i = init(n); condition(n, i); i+=s)` where `n` and `s` are numbers. For each `isl_ast_node_user` PET-to-MLIR generates a statement. Specifically, from the AST node, the statement id

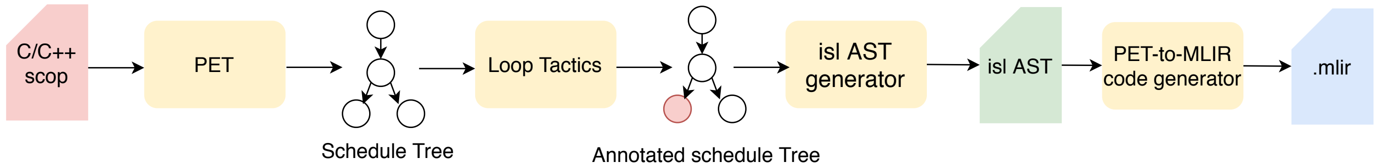

Fig. 3. PET-to-MLIR tool flow. Scops are extracted using PET. Optionally the user can use Loop Tactics to detect computational motifs. After Loop Tactic from the schedule tree, we generate ISL AST and provide a code generator to emit affine.

is extracted, and the statement look-up in the statement list. We emit the statement operations by walking the parse tree. `isl_ast_node_block` are handled in the same way as `isl_ast_node_user` with the difference that they contains multiple statements. `isl_ast_node_mark` are used to emit BLAS calls. Mark nodes are allowed to be inserted by Loop Tactics only and carry the information on what BLAS pattern has been detected (i.e., _GEMM or _BATCHED_GEMM). Whenever the PET-to-MLIR code generator hits a mark node, the subtree is replaced with a function call to a BLAS library. Finally, ISL AST nodes of type `isl_ast_node_if` are not yet handled.

A. GEMM kernel

Listing 3 shows the generated affine code for our running example (Listing 1) without and with Loop Tactics. Let us start by describing the code snippet on top obtained by running without Loop Tactics and using the following compilation string `mlir-pet -I /path gemm.c`. In this case, all the inputs are marked as exposed; thus, PET-to-MLIR inserts them as inputs to an MLIR FuncOp (`scop_entry`). The two dimensional tensors `A, B` and `C` are modelled as `memref` types where each element is a `f32` type. Scalar `alpha` and `beta` as `f32` types. Line 5 and 6 in the code snippet reflects statement S1 in Listing 1, whereas line 9 to 14, correspond to statement S2. Let us now turn our attention on the code snippet at the bottom obtained by running Loop Tactics with the matcher reported in Listing 2. We can see that a function call has replaced the GEMM pattern. The function call has 7 operands. The first and the second tell us if either the matrix A or B or both are transposed. Arguments 3 to 6 are the matrices and the alpha constant. Finally, argument 7 is the beta constant, which in this case is set to one as we don't capture the initialization statement in the Loop Tactics' matcher. The operands are automatically collected by Loop Tactics.

IV. Evaluation

In this section, we evaluate the applicability of our tool by lowering to affine different kernels from the Polybench 4.2 benchmarks suite. For GEMM-like kernels (`2mm`, `3mm`, `gemm`) we additionally run Loop Tactics to lift the entry point and thus exploiting domain-specific optimizations as the invocation of vendor-optimized routines. All the results in this section are in single-precision and report the arithmetic mean of five independent runs. As platform, we use an Intel i9-9900K clocked at 3.60GHZ. We measure the peak performance using an SGEMM routine from the MKL library. PET-to-MLIR reports the performance of the C code lowered with our tool to affine and then just-in-time compiled with the `mlir-cpu-runner`. PET-to-MLIR + LT, on the other hand, shows the performance achieved by detecting computational motifs and replacing the code with BLAS functions. As Figure 4 shows, we can successfully lower kernels from the linear-algebra and stencil domain. An exception is made for `durbin, cholesky gramschmidt` and `ludcmp` as we currently do not support constant accesses to arrays and operations like division (not reported in the Figure). Although PET-to-MLIR is still a fairly new tool, it is already capable of translating most of the Polybench benchmark suite. The MLIR generated code is first syntactically checked with the `mlir-opt` tool, thus proving we generate a valid MLIR code. Behavior correctness, on the other hand, was tested by comparing the last ten values of each kernel's output matrices with the output of the GCC compiler.

V. Related Work

MLIR front-ends: Flang is the new LLVM front-end for FORTRAN code. It lowers FORTRAN code to the MLIR's F18 dialect to perform advanced loop optimizations [8]. Teckyl is an MLIR front-end for Tensor Operations. It allows the user to start from a program written in Tensor Comprehension notation and lowers it to the Linalg or the Loop dialect [9], [10]. TensorFlow allows also to enter MLIR via the TensorFlow IR dialect [11], [12].

Polyhedral extractors Perhaps the most well-known polyhedral extractors (and optimizers) are Polly [13] and graphite [14]. The former extracts the polyhedral representation from the intermediate representation of LLVM while the latter from GCC. Several other compilers such as R-Stream and IBM-XL uses polyhedral techniques and thus extractors. But they are proprietary compiler; thus, limited documentation is available. Clan, together with PET is one of the most well-spread polyhedral extractors for source-level code [4], [6]. But it comes with limitations that have been addressed in PET. SUIF is also used to extract polyhedral representation, but it is not maintained anymore and it does not support C99, thus it has fallen out of fashion [15]. PET-to-MLIR fits in such a category of tool, but it is orthogonal to them as it targets the affine dialect in MLIR.

VI. Limitations and Future Work

Although PET-to-MLIR is already able to handle the majority of the Polybench benchmarks suite, it is still relatively

```
1  func scop_entry(%arg0: memref<1024x1024xf32>, %arg1: memref<1024x1024xf32>,
2                  %arg2: memref<1024x1024xf32>, %arg3: f32, %arg4: f32) {
3    affine.for %arg5 = 0 to 1024 {
4      affine.for %arg6 = 0 to 1024 {
5        %0 = affine.load %arg2[%arg5, %arg6] : memref<1024x1024xf32>
6        %1 = mulf %arg4, %0 : f32
7        affine.store %1, %arg2[%arg5, %arg6] : memref<1024x1024xf32>
8        affine.for %arg7 = 0 to 1024 {
9          %2 = affine.load %arg0[%arg5, %arg7] : memref<1024x1024xf32>
10         %3 = mulf %arg3, %2 : f32
11         %4 = affine.load %arg1[%arg7, %arg6] : memref<1024x1024xf32>
12         %5 = mulf %3, %4 : f32
13         %6 = affine.load %arg2[%arg5, %arg6] : memref<1024x1024xf32>
14         %7 = addf %5, %6 : f32
15         affine.store %7, %arg2[%arg5, %arg6] : memref<1024x1024xf32>
16       }
17     }
18   }
19   return
20 }
```

```
1  func scop_entry(%arg0: memref<1024x1024xf32>, %arg1: memref<1024x1024xf32>,
2                  %arg2: memref<1024x1024xf32>, %arg3: f32, %arg4: f32) {
3    affine.for %arg5 = 0 to 1024 {
4      affine.for %arg6 = 0 to 1024 {
5        %0 = affine.load %arg2[%arg5, %arg6] : memref<1024x1024xf32>
6        %1 = mulf %arg4, %0 : f32
7        affine.store %1, %arg2[%arg5, %arg6] : memref<1024x1024xf32>
8      }
9    }
10   %cst = constant 1.000000e+00 : f32
11   %2 = llvm.mlir.constant(0 : i32) : !llvm.i32
12   %3 = llvm.mlir.constant(0 : i32) : !llvm.i32
13   call matmul(%2, %3, %arg2, %arg0, %arg1, %arg3, %cst) :
14     (!llvm.i32, !llvm.i32, memref<1024x1024xf32>, memref<1024x1024xf32>,
15      memref<1024x1024xf32>, f32, f32) -> ()
16
17   return
18 }
```

Listing 3: Affine IR emitted for Listing 1 with and without running Loop Tactics.

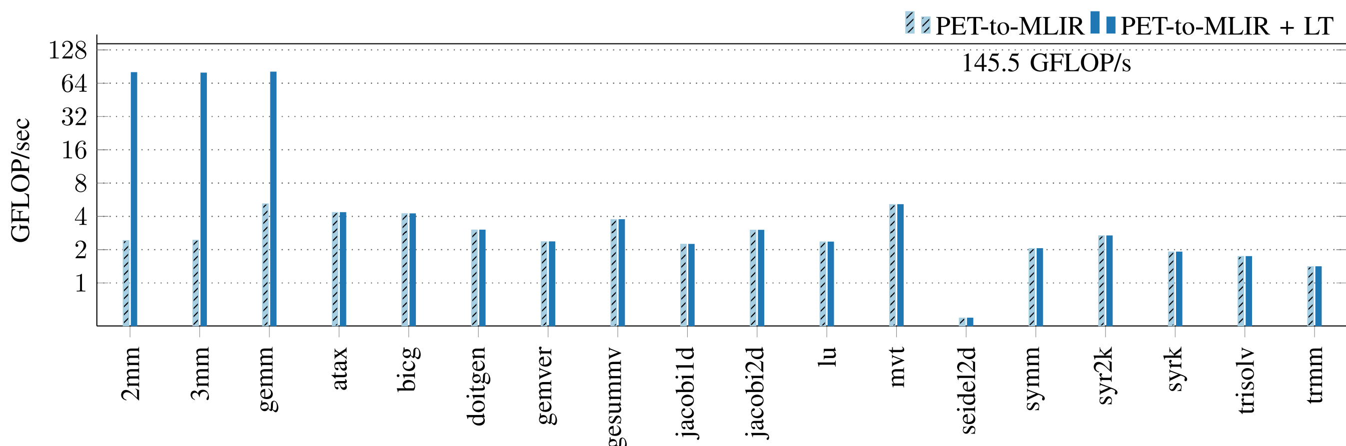

Fig. 4. Performance obtained by lowering C code to the Affine in MLIR via PET-to-MLIR.

new and under active development. At the time of this writing (git commit: `c832a7d`)[1], the tool comes with limitations that, however, do not preclude its usage. Perhaps the most prominent limitations are: 1) If and else construct are not yet handled in the code generation. Currently, when an if condition is detected in the code fragment to be translated, the tool exists with a warning. 2) Symbolic bounds are not yet handled, and for now, we require all the loop bounds to be statically known (`-DPOLYBENCH_USE_SCALAR_LB` in Polybench). 3) External function calls in the code fragment are not allowed, and the tool bails-out if a call is detected. 4) Other operations such as division, as well as, constant accesses to arrays are not handled yet. 5) Line locations are not tracked.

Future work will extend the test coverage and make sure the tool is feature complete. We will also work on emitting higher-level of abstraction dialects such as Linalg. Besides, we want to utilize the matchers in Loop Tactics to extract algorithmic information for general-purpose code. By defining structural and access patterns (matchers) for dialect-specific functions like pooling or transpose, we can enable parts of the originally general-purpose code to reach domain-specific dialects, thus allowing exploration of algorithmic-level optimizations.

VII. Conclusion

By exploiting the strengths of mature polyhedral tools, we have constructed, to the best of our knowledge, the first frontend for MLIR for polyhedral friendly C code. Although the development of the tool is still in progress, it can already handle a subset of the Polybench/C benchmark suite, as demonstrated in our evaluation.

Finally, by using Loop Tactics, we provide a way to lift the entry-point in the MLIR compilation pipeline, thus enabling domain-specific optimizations, such as the use of vendor-optimized libraries. This last point also highlights the importance of retaining semantic information in compiler IR, which is the reason why MLIR has been developed in the first place.

Acknowledgments

This work was partially supported by the European Commission Horizon 2020 programme through the MNEMOSENE grant agreement, id. 780215 and the NeMeCo grant agreement, id. 676240.

[1] https://github.com/LoopTactics/mlir.git

References

[1] C. Lattner, J. Pienaar, M. Amini, U. Bondhugula, R. Riddle, A. Cohen, T. Shpeisman, A. Davis, N. Vasilache, and O. Zinenko, "Mlir: A compiler infrastructure for the end of moore's law," *arXiv preprint arXiv:2002.11054*, 2020.

[2] U. Bondhugula, "High performance code generation in mlir: An early case study with gemm," 2020.

[3] L. Chelini *et al.*, "Declarative loop tactics for domain-specific optimization," *ACM TACO*, Nov. 2019. [Online]. Available: http://doi.acm.org/10.1145/3372266

[4] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12), Paris, France*, 2012, pp. 1–16.

[5] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *International Congress on Mathematical Software*. Springer, 2010, pp. 299–302.

[6] C. Bastoul, "Clan-a polyhedral representation extractor for high level programs," 2008.

[7] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen, "Schedule trees," in *International Workshop on Polyhedral Compilation Techniques, Date: 2014/01/20-2014/01/20, Location: Vienna, Austria*, 2014.

[8] E. Schweitz. (2019) An mlir dialect for high-level optimization of fortran.

[9] A. Debres, "Teckyl," https://github.com/andidr/teckyl, 2020.

[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] J. Pienaar, "Mlir in tensorflow ecosystem," 2020, compilers For Machine Learning (C4ML) 2020, San Diego, CA, USA.

[12] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 265–283.

[13] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in llvm," in *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, vol. 2011, 2011, p. 1.

[14] S. Pop, A. Cohen, C. Bastoul, S. Girbal, G.-A. Silber, and N. Vasilache, "Graphite: Polyhedral analyses and optimizations for gcc," in *Proceedings of the 2006 GCC Developers Summit*. Citeseer, 2006, p. 2006.

[15] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam *et al.*, "Suif: An infrastructure for research on parallelizing and optimizing compilers," *ACM Sigplan Notices*, vol. 29, no. 12, pp. 31–37, 1994.