Abstraction raising in MLIR

Komisarczyk, Konrad

Award date:
2021

Master Thesis

Department of Mathematics and Computer Science
Electronic Systems Research Group

Abstraction raising in MLIR

Konrad Komisarczyk

Supervised by:
dr. ir. Roel Jordans
ir. Lorenzo Chelini

Eindhoven, March 4, 2021

Abstract

The increasing complexity in hardware architecture poses challenges to general-purpose compilers, which struggle when compared to domain-specific compilation flows. General-purpose compilers operate as a black-box at a low-level of abstraction, limiting the set of possible optimizations that can be applied on a given piece of code. Besides, general-purpose languages are not semantically rich enough to explicitly provide the necessary (high-level) information for optimization purposes. As a consequence, obtaining good performance out of a general-purpose compiler is hard. Given a poor performance, code developers have two choices: 1) optimize the code using extensive rewriting of the original, possibly requiring custom assembly's or 2) lift the level of abstraction by manually re-targeting the code to a domain-specific language.

Both solutions are sub-optimal. The former ties the code to a specific platform and programming model, thus reducing portability. The latter requires expensive and tedious manual translation. This work aims at enabling domain-specific optimizations *directly* on general-purpose code by lifting the level of abstraction. To the best of our knowledge, the proposed framework, which we call PET-to-MLIR, is the only work which combines declarative C++ API for declarative pattern detection with domain-specific compilation within the novel MLIR compiler framework.

With our PET-to-MLIR tool, we show that embedding domain-specific optimizations within a general-purpose compilation flow using the MLIR infrastructure broadens the range of possible optimizations, thus improving performance. We compare the optimization enabled by our tool with state-of-the-art optimizers and current general-purpose compilers, clearly showing the performance benefit of our approach. By implementing our abstraction raising we reduce the need for domain-specific compilers and languages.

Contents

Chapter 1

Introduction

Architectural and hardware trends such as increasing number of cores, larger and deeper memory hierarchy, and widening vector units, together with more demanding and complex applications, are making general-purpose compilers ineffective [6]. At the same time, with the end of Moore's law, we witness an architectural trend toward heterogeneity and application-specific architectures. This poses incredible challenges to general-purpose compilers and makes it imperative to design compiler optimizations that depart from single, general-purpose optimization criteria. As the traditional compilers struggle, more domain-specific languages emerged and with them domain-specific (DS) compilers. DS compilers implement specialized internal representations preserving the high-level semantics of the language and apply transformations by assuming domain-specific knowledge for a given class of programs [7]. The importance of high-level semantics during the optimizations stems from the fact that without the domain knowledge embedded by construction in the intermediate representations, the compiler has to resort to complex compiler analysis [12]. Today, the internal representation of general-purpose compiler is not rich enough to capture domain-specific knowledge, thus preventing specialized optimizations. Consequently, general-purpose compilers without the ability to recognize whether the code belongs to a particular domain, by design, cannot adopt DS optimizations strategies.

Nevertheless, it has been shown that it is possible to decouple the benefits of DS transformations and the necessity of resorting to multiple DSLs in References [6, 10] by enabling DS techniques directly on loop nests written in general-purpose languages. Due to the absence of initial assumptions about the input code's functionality, the proposed solutions rely on pattern detection methods. In particular, existing tools define a set of popular code patterns (or idioms), attempting to extract them from the input code, and optimize them with customized transformations. This shifts the problem from maintaining multiple DS compilers to maintaining a set of customized optimizations per pattern. Recently, however, a new concept of a hybrid compiling framework emerged, which facilitates the design of DS compilers and strives to connect techniques from domain-specific and general-purpose flows within a common infrastructure [16]. It is also based on a principle of *progressive lowering*, i.e., lowering code from high-level to low-level abstraction in small steps across the abstraction level and preserving high-level semantics as long as it can be useful. As a result, this design principle reduces the gap between high-level operations and low-level IRs, and the effort of a single raising step becomes smaller. We show how this principle can allow us to use the abstraction raising more effectively. Contrary to the current state-of-art tools enabling DS optimizations like Loop Tactics, we generate high-level operations to broaden the scope of DS analysis and reuse the DS optimizers within the MLIR rather than introducing optimizations ourselves. In this work, we introduce abstraction raising capabilities to such common infrastructure called Multi-level Intermediate Representation (MLIR) and hence propose a compilation flow, which allows general-purpose code to use already existing aggressive domain-specific optimizations offered by MLIR [16].

Our framework relies on a currently available polyhedral optimizer named Loop Tactics. We extend their work with our code generator called PET-to-MLIR. Compared to Loop Tactics' pattern detection, we use the abstraction raising to perform DS optimizations by generating high-level operations rather than optimizing

directly on the general-purpose code. Specifically, we decouple the optimization stage from the detection stage. Instead of pairing customized optimizations with a detected pattern, we first complete the detection fully and then apply the domain-specific optimizations. The annotations provide the necessary information for further high-level code generation performed within the MLIR infrastructure. Effectively we postpone Loop Tactics DS optimizations, reintroduce higher-level semantics and provide more context from neighboring operations for optimization purposes. This allows defining optimization passes, which rely on only the operation semantics for general-purpose C implementations without resorting to complex loop analysis. The analysis is only used for the MLIR code generation before commencing the optimization steps. For instance, a double transpose pass for a high-level SSA representation can be defined sufficiently by identifying whether a transpose operation in a given program takes an argument generated by another transpose as input. Without the abstraction raising step, it is impossible to access the optimizers using higher abstraction representations for the transpose's loop implementation at the C or C++ level. Moreover, instead of optimizing ourselves, we enable optimizations by choosing an adequate IR as a lifting target within the MLIR infrastructure. Consequently, we automatically reuse the DS compilation flow for the lifted parts and general-purpose compilation for the remaining ones. This is possible due to MLIR's flexibility, which allows to intermix different dialects together in the same compilation module.

By introducing our code generator, which can generate code within the MLIR framework on varying abstraction levels based on the amount of information extracted by Loop Tactics, we bridge the gap between domain-specific and general-purpose compilers.

1.1 Motivation

Restricting the compiler's input domain by developing a dedicated compiler for a particular domain (i.e., Halide for stencil computations) has revolutionized the generation of high-performance code [12]. The DSLs are becoming more popular as the high-level abstractions improve both the programmer's productivity and the DS compiler's analysis. The compilation from the DSLs simplifies the correctness analysis of the possible transformations and allows more aggressive heuristics [25]. For instance, there are DS compilers for Matlab, which exploit matrix algebra properties for matrix operations sequences and, by reassiociating matrix products, they are also able to reduce the workload, or even lower the operation's complexity [18]. There are tools, which introduce a subset of these opportunities for general-purpose code for specific parts via recognizing popular code structures. However, as discussed further in Section 2, tools using abstraction raising typically apply an optimization tactic upon the detection, like Loop Tactics, and, more rarely, generate high-level operations for a specific domain. Our approach, on the other hand, allows generating and mixing both general-purpose and domain-specific code at the operation level granularity within one compiling framework.

Although, in theory, the need for abstraction raising can be solved at the beginning of any project by a comprehensive design, resources, and a fully informed choice of a programming framework, we argue that the compiler should also be able to support domain-specific optimizations for legacy C or C++ code. Apart from the better reuse and support for the legacy code, another benefit would be the increase of programmer's productivity by allowing them to use general-purpose languages without learning the syntax of new domain-specific languages.

We use the abstraction raising to address the main drawbacks that comes hand in hand with the DSLs' main advantage. Namely, that the DSLs by design limit their generality. The programmers have to study multiple existing solutions and their advantages per domain involved. There exist many DSLs offering varying quality, support, and performance within a domain [28]. Furthermore, this puts more emphasis on early lock-in choice, potentially leading to rewriting of existing solutions as the applications evolve [10]. Applications typically involve multiple domains, and the programmers need to compose their solutions efficiently or turn to general-purpose languages for tasks with softer performance restrictions [29]. Furthermore, many competitive DSLs are stand-alone frameworks with low reusability and composability, which excludes them from multi-domain applications [6, 29].

For the multi-domain applications, the abstraction raising allows avoiding the potential high-code fragmentation without resigning from the DS optimizations. In other words, it allows using general-purpose languages without impairing performance drastically [7]. As discussed in Related Works, Chapter 2, there are available tools that enable the specialized transformation of specific loop patterns by recognizing the implemented

functionality with analyzing the loop's regular data access patterns or matching the textual representations. For example, Loop Tactics would recognize the matrix multiplication's loop implementation and replace the code fragment with a hand-tuned optimization designed for the GEMM-like kernels [6]. However, due to the optimization typically performed upon the detection without generating any high-level code, these tools prevent optimizing across multiple patterns. Across the discussed tools, the common optimization strategy for a single C or C++ loop pattern is to invoke a vendor-optimized library, like BLAS's matmul for the linear-algebra example but an optimization like loop fusion across consecutive patterns is not enabled.

The optimization performed on a single detected loop pattern without high-level code generation, including code replacement with vendor-optimized routines, can be sub-optimal. The library routines are tuned in isolation, and, in some cases, the automatic general-purpose loop optimizers like PLuTo can outperform the vendor routines with loop fusion for sequences of the BLAS-like operations [19]. This means that, when detecting completely independent matrix-vector products, it is sufficient to replace the code with highly specialized routines. However, the optimal approach becomes less obvious when detecting operations, which share memory accesses, especially when loop fusion between separate loop nests means a reduction of read accesses. Therefore when extracting high-level information it is beneficial to postpone the code transformation until after the detection phase and only then perform high-level code generation. This allows a choice between the strategies based on the successful detections' composition. That is, call BLAS or decide to lower as general-purpose loop nest towards a generic, polyhedral-like loop nest optimizer having detected a sequence of specific and co-dependent operations. This choice is only possible due to the abstraction raising followed by the high-level code generation.

However, the invocation of a manually optimized routine for an isolated kernel and the use of an automatic general-purpose loop optimizer for their composition are not the only options. The automated loop analysis is heuristic-based and can involve completely different optimization techniques based on the considered domain and target. For instance, computation inlining is one of the most effective transformations for stencil kernels and is typically included in DS optimizers, while it is absent in PLuTo [13]. This is an example of an automated DS analysis across the detections, where the sequences of high-level operations use specialized heuristics. Tools like Loop Tactics performing code transformation upon pattern detection are not designed to optimize general compositions of loop patterns. It is possible to define a specific and complex pattern, but defining a fusion of a specific kernel's sequence of an unknown length is not directly possible. We can enable such analysis by generating high-level operations.

The final important part of the motivation is the recent announcement of the compiling framework implementing the aforementioned concept of a hybrid compiling framework - The Multi-Level Intermediate Representation (MLIR). The MLIR allows the coexistence of the general-purpose and DS techniques in one unified infrastructure and, by doing so, facilitates abstraction raising for diverse applications. Contrary to the baseline LoopTactics' optimization strategy, we define raising transformation to MLIR's higher-level dialects and reuse their DS knowledge, rather than define immediate rescheduling transformations ourselves. MLIR gives us a unique opportunity to rethink abstraction raising. It allows the coexistence of multiple levels of abstractions and entering MLIR with code fragments precisely to DS representations at the level where raised operations are analyzed. As a result, it enables aggressive domain-specific optimization for a detected code block in a context of other raised blocks belonging to the same domain, which are represented without redundant abstractions, e.g., detection and generation co-dependent matrix-matrix and matrix-vector multiplications operating on a generalized tensor type.

1.2 Outline

We organize the thesis as follows:

- **Chapter 1** continues with defining the problem statement and covers the background information. The Background Information consists of: 1) specific domain-specific optimizations, which we have in mind when motivating the problem 2) state-of-art general-purpose loop optimization 3) modeling loop nests with polyhedral framework 4) code lowering before and with MLIR.
- **Chapter 2** discusses the related works, which introduce domain-specific optimizations based on algorithm detection and abstraction raising. Further, it concludes the differences between other works and

our goal and highlights the thesis's contributions.

- **Chapter 3** describes the proposed solution in detail and step-by-step covers the abstraction raising from C/C++ code to MLIR's high-level representation.
- **Chapter 4** describes the experiment setup and measurement methods.
- **Chapter 5** introduces research questions, which are addressed with the experiments and discusses the obtained results.
- **Chapter 6** concludes the thesis and discusses future work.

1.3 Problem statement

As introduced in the previous sections we want to enable domain-specific optimizations, which are currently unavailable to general-purpose code. We propose to solve the problem with abstraction raising in a structured way that allows the reuse of the existing techniques. This means generating high-level operations, which explicitly encode semantic information in their operation type without manual analysis and rewriting. Consequently, the generated operations can be recognized by DS optimizers, thus enabling the reuse of existing infrastructure. We define a solution that meets the goal with the following requirements:

1) Automatic. The productivity, which comes with the automation of enabling the optimizations rather than manual translation, is the main motivation of the project. Both the extraction of high-level information and the generation of high-level operations starting from the low-level code must be fully automatic. The only manual effort predicted would take place to initialize the proposed compiling process. The developer would define features of the functions considered for raising. The type of feature, which allows determining the performed functionality, depends on the loop-modeling framework, e.g., access patterns in the polyhedral model or pre- and post-conditions in program synthesis. Then developer would provide recipes to generate the corresponding high-level operation coupled to these features. This means that having defined features of a specific loop-based program, the chosen extraction mechanism shall automatically provide us with a labeled program. These labels will act as an annotation passed to the compiler instructing it about the function detected within that part.

2) Flexible. We want to minimize the necessity of lock-in choices of specific stand-alone DSL. With the expressiveness matching multiple domains, the solution must be capable of generating operations for more than one DS representation.

3) Fine-grained. We are looking for a solution that is versatile in terms of recognizing loop patterns. Fine-grained granularity means the model used for pattern expression should capture enough low-level details to express structures for various applications. Polybench/C has encoded all the common idioms in High-Performance Computing for the dense linear-algebra, data-mining, and stencil-based domains. Therefore, we will use the benchmark as a minimum range of various loop structures that the considered algorithm classification method should be capable of encoding.

4) Effective. The solution must provide performance gains compared to the current state-of-art general-purpose loop optimizer PLuTo with a subset of domain-specific techniques introduced in Subsection 1.3.1.

5) Reusing the existing techniques. The proposed solution must be effective without re-implementing the optimization passes ourselves. The method will generate DS operations and, by doing so, enable access to the techniques present in the compiling frameworks that defined them.

1.4 Background information

This section covers the background information important to understand the project. We describe examples of domain-specific optimizations that motivate the thesis, loop analysis, state-of-art general-purpose loop optimization, and MLIR.

1.4.1 Domain-specific optimizations

To give a better insight into the optimization potential we briefly introduced in the Motivation Section 1.1, we discuss the DS optimization techniques we want to enable. We start from the vendor-optimized routines

and then describe the high-level complementary and conflicting approaches for optimal code generation.

Vendor-optimized routines

The vendor-optimized routines are provided by manually developed target-specific libraries. Vendor-optimized routines like BLAS are still typically insuperable performance-wise for tasks with a high flop to memory access ratios (like matrix-matrix multiplication) compared with automatic and more portable approaches. For these kernels, the routines' main drawback is their strict target-dependence, and, thus, the necessity of expert's effort duplication for each rapidly evolving architecture. For this reason, they are typically developed for popular, widely utilized functions, which can also be used as basic blocks for more complex domain-specific operations [30]. As discussed in the next section, there is a tremendous effort in research to reduce the performance gap between precompiled routines and their automatic optimization. It must be noted, however, the even for the high flop to memory ratios the routines can be suboptimal for small scale computations [27]. Library developers tend to introduce additional checks to switch assemblies, e.g., Intel MKL introduced GEMM specialized for small matrices [27].

For other kernels, with a lower floating-point operation to memory ratios, the gap exists mainly when analyzing single BLAS-like operations. When considering benchmarks composed of sequences of such operations, a higher memory efficiency can be achieved mainly with an inter-procedural analysis, e.g., loop fusion across the BLAS patterns [19].

Complexity reduction and optimizing with matrix properties.

A target-independent method to aggressively optimize a program is to verify whether the workload can be reduced using the high-level properties of the considered operations or heuristic-based algorithms, which can under-perform for specific problem sizes. The most basic example of such reduction can be achieved with associativity or distribution axioms in matrix algebra for a given sequence of matrix operations:

$$b = A \cdot B \cdot C \cdot x \tag{1.1}$$

The re-association of the products to start executing from the right (a mat-vec product) can reduce the work from $\mathrm{O}(n^3)$ to $\mathrm{O}(n^2)$. Similarly, assuming only matrix-matrix products of $(A \cdot B \cdot C)$, the matrix-chain reordering is a method minimizing the sizes of the intermediate products, which results in a smaller problem size:

$$(A[m \times n] \cdot B[n \times k]) \cdot C[k \times l] \rightarrow flops = m \cdot k \cdot n + m \cdot k \cdot l \tag{1.2}$$

$$A[m \times n] \cdot (B[n \times k] \cdot C[k \times l]) \rightarrow flops = n \cdot l \cdot k + n \cdot l \cdot m \tag{1.3}$$

Empirical cost functions for the fusion of BLAS-like operations

An interesting example of DS loop optimization, which is not complementary to BLAS but is an alternative is the Build-To-Order compiler. It introduces polyhedral-like analysis but is more effective in competing with vendor-optimized routines than PLuTo by using a higher abstraction level starting point. Firstly, the BTO approach does not accept loop nests as input but focuses on optimal loop nest generation from matrix operations. It limits the input domain to a subset of Matlab language and attempts to fuse the operations across the sequence of level 1 and 2 BLAS-like operations. Therefore, the compiler focuses on the high-level functionality of which the bottleneck is the memory bandwidth. In its transformation strategy, the compiler incorporates the input code context into a heuristic algorithm, which uses empirically established execution cost values to predict fusion candidates' performance [9].

1.4.2 General-purpose loop optimization

One of the first approaches to provide an efficient, general-purpose solution to the affine scheduling problem dates back to 1992 and is in many polyhedral optimizers refereed to as Feautrier's function [8]. For instance, a simple yet powerful observation has been made that the polyhedral space's dependency distances are equivalent to the reuse distances. This way of representing affine schedules and mathematically interpreting their properties has initially allowed to search for fine-grained parallelism and reduce latencies. Based on these principles, further works led to a fully automatic, heuristic-based tool for coarse-grained parallelism - PLuTo.

PLuTo effectively limits the exploration space of the full combination of schedules and introduces a heuristic-based cost function for modern multicore architectures [5]. PLuTo has been implemented in Graphite and Polly, which are parts of general-purpose compilers, respectively GCC and Clang. The Polyhedral Parallel Code Generator is a PLuTo-based tool specialized in GPU code generation. The tool searches for two levels of parallelism to offload loops to CUDA blocks and threads.

The main limitation of the automatic loop scheduling is its reliance on realistic cost functions and parameters modeling cache sizes. The cost functions model the trade-off between data movement, levels of parallelism, synchronization, and recomputation. In the memory hierarchy context, the seemingly redundant recomputation can result in significant execution time gains. This occurs especially when the memory communication due to large tile sizes becomes too costly [5]. The automatic code transformations, based on the empirically proposed functions within the polyhedral framework, have proven to be effective in optimizing single-processor execution for specific regular kernels and approaching the hand-tuned methods [2].

However, as concluded in Park et al., the heuristic-based approaches fail to expose parallelism for a wide range of applications and more sophisticated processing architectures due to over-simplistic models, which do not reflect the behavior of modern complex processors [21]. Another issue with the models is that they limit the optimization search space to facilitate analysis and offer feasible selection times. For instance, currently, many polyhedral algorithms assume singular value schedules meaning that all statement instances in the iteration space are mapped to exactly one point in time space [13]. As a result, such optimization analysis excludes an important strategy, i.e., computation inlining.

There are efforts to replace the PLuTo-based approaches using feedback iterative optimization instead of finding the best-modeled transformation set. The work of Cohen et al. proves the concept on small programs and emphasizes scalability problems [23]. Park et al. continued the approach by involving machine learning to organize the transformation set with a limited number of optimization primitives, which could be composed in multiple transformation vectors according to their presented procedure [21]. They were not, however, able to outperform the PLuTo algorithm despite a much higher transformation selection time. This was achieved by another work, which replaces the execution features with Polyhedral features. The most obvious advantage of such an approach is removing the necessity for full compilation to extract the representation of the code [33]. Nevertheless, the high selection times still prevent this approach from being practical in general-purpose compilers, and PLuTo-based solutions remain the leader of automatic, general-purpose loop optimization.

1.4.3 Polyhedral model

As briefly mentioned in the heuristic-based BTO-BLAS compiler and emphasized in the general-purpose loop optimizations, a framework for the systematic loop modeling is a critical tool because it allows a mathematical analysis of looping programs.

Loop nests are typically the most computation-intensive parts of the programs. The loop optimizations based on polyhedral analysis are heavily used in general-purpose compilers. “Polyhedral modeling has been the basis for multiple major advances in compiler technology over the last decades and is present in major production compilers such as GCC, LLVM, and IBM XL.” [34] Polyhedral modeling remarkably simplifies loop analysis and thus is a backbone of both abstraction raising and automatic loop scheduling. Polyhedral models are abstract from individual operations and provide separation of concern in explicitly expressing three fundamental aspects of looping programs: number of executions, executions’ relative ordering, and access subscripts. This principle simplifies the expression of computation patterns at their semantic level. We introduce fundamental concepts of the polyhedral framework as the abstraction raising strategy we use further in the project relies on matching these patterns, expressed with polyhedral representations.

The polyhedral model is a mathematical framework that models each operation’s execution for every iteration in a loop nest individually. The executions are modeled with an integer lattice, which composes convex polyhedrons in a multidimensional space constrained by affine hyperplanes or half-spaces. As such, the model offers information for the loop’s fine-grain and coarse-grain parallelization due to statement level analysis. It can be used to prove the semantic correctness of transformed programs by mathematically proving the legality of the applied transformation with the dependence analysis.

Statement instances and iteration domain

Every execution of an operation belonging to a loop nest is represented by an integer lattice in a multidimensional space. Each point within the lattice is called a statement instance. The statement instances are described by the iteration domain, which defines the points in space (polyhedron or union of polyhedrons) belonging to the program.

```
    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.

For the example of GEMM presented in Listing 1, the set statement instances i.e iteration domain is defined with a set of inequations as follows:

$$S1(i,j) \quad | \quad 0 \leq i < 1024 \wedge 0 \leq j < 1024$$
$$S2(i,j,k) \quad | \quad 0 \leq i < 1024 \wedge 0 \leq j < 1024 \wedge 0 \leq k < 1024$$

Execution order

After establishing the set of programmed executions, the execution order of these points has to be defined. This order is necessary as a fundamental feature expressing the source program's behavior. It is also essential to express potential data dependencies based on the causality, which determines an accessed value source. The mapping from the iteration space to the time-space is done by introducing a scheduling relation. The relation maps the input dimensions onto a time vector. The time vector represents a lexicographical order with the higher dimensions being less dominant ordering units, illustrated as follows: (year, month, week).

The scheduling relation for the considered example maps iteration vector to a time vector:

$$S1(i,j) \rightarrow (i,j,0)$$
$$S2(i,j,k) \rightarrow (i,j,k+1)$$

Memory accesses

Accesses are also represented with the relations, i.e., access relations that map the iteration space to the, possibly multidimensional, memory indices. In short, each access is a multidimensional vector with entries expressed by affine expressions composed of loop's index variables and parameters. An example of a write and a read accesses for the considered GEMM program are given below:

$$W_{S1,C}(i,j) \rightarrow (i,j)$$
$$R_{S2,A}(i,j,k) \rightarrow (i,k)$$

Schedule tree representation

Schedule trees are one of the polyhedral schedule representations that combines all the aforementioned elements in a way that simplifies the order encoding by utilizing the tree's structure [32]. A schedule tree representing the GEMM code is shown in Fig. 1.1. Each schedule tree starts with a domain node that defines the iteration space for the scheduled subtree.

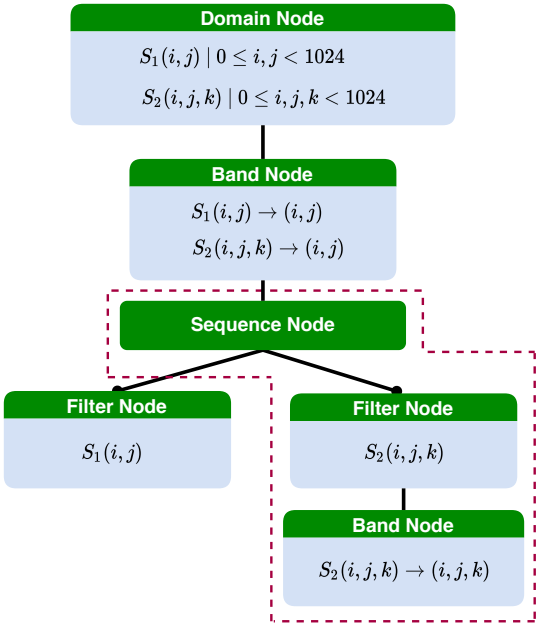

Figure 1.1: A schedule tree representing GEMM program from List. 1. The dashed line marks the subtree representing the matrix-multiplication that follows the nested data initialization. Image from Komisarczyk et. al. [15]

For the given example the domain node is followed by the 1) `band` which defines the partial schedule of the loops identified with the dimensional indices. Next, there is the `sequence` which imposes ordering between the node's children. Finally, the `filter` nodes allow scheduling individual statement instances by restricting iteration domain considered by its subtree. In the example, the filter nodes separate further scheduling of S1 and S2. In short, the first band node schedule the first two loops for the (i,j) dimensions, while the sequence node ensures the order within that loop nest, where each S1 instance is followed by the for-loop running entire range of iterations for dimension k on S2. Note that due to the existence of the `sequence`, the offset in S2's scheduling relation is no longer necessary.

Benefits of polyhedral model

1. Affine transformations are a generic set of transformations, which after an application to a polyhedron result in a polyhedron.
2. Compositions of transformations are easily expressed.
3. Transformation legality is easily checked (mathematical expression of dependence violation).
4. Natural expression of parallelism e.g., the simplest form of expressing the parallelism in the model is by extending the schedule space with a dimension per execution unit (year, month, factory number, week).
5. Explicit schedule encoding in schedule trees simplifies detecting patterns in loop structure.

1.4.4 MLIR and progressive lowering

The code lowering process within traditional frameworks comprises a fixed number of sequential stages, gradually translating programmer-written code to machine code. The more stages there are, and, thus, the more progressive the lowering process is, the more modularity and reuse can be achieved. Before the MLIR, the reuse has been obtained by introducing new front-ends, which could enter a compiling module and access sequences of passes it implemented. Figure 1.2 shows a typical lowering pipeline for a programming language. It involves multiple compiling units working on different abstraction levels. The LLVM itself comprises multiple stages and offers extensive support for low-level machine code generation. For this

reason, it has become a targetted lowering framework by many programming models, which introduced their own IRs to perform their high-level optimizations. These IR-based stages built on top of the LLVM IR are high-level domain-specific SSA-based optimizers, which by building on top of the LLVM IR framework, create a specific stack of IR transformers. Each stack as a whole is aimed at supporting a variety of platforms and programming models. The problem with this model is a duplication of effort in implementing techniques within each compiling unit. Every IR-based compiling unit is an SSA-typed optimizer and, the main difference between them is that each IR defines its own abstractions (high-level operations and type systems, e.g., matrix operations working with the tensor domain). Each compiling unit implements compiler passes performing similar tasks: enabling transformations, optimizing transformations, lowering, and cleanup [16].

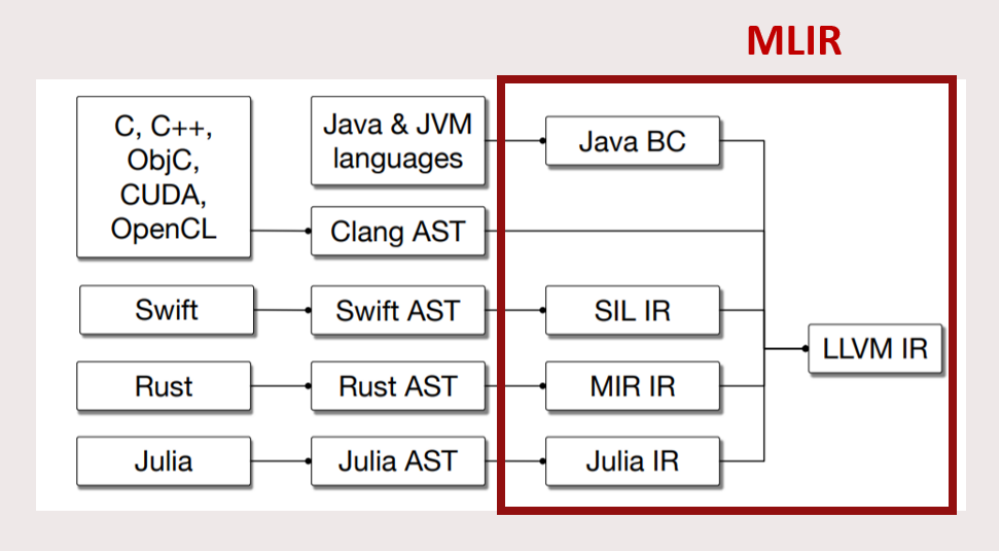

Figure 1.2: Duplicated infrastructure for IRs built on top of the LLVM IR. The red box shows the intended range of MLIR's unifying infrastructure. Image from Lattner et. al. [16]

The different domain-specific programming models could benefit from the shared infrastructure due to the lower development costs and shared techniques for general IR analysis. They could also experiment with innovations appearing in other domains and share techniques regarding cross-language features like OpenMP. Lastly, the abstraction gap between model-specific IRs and LLVM IR is relatively large, and as such, it makes the abstraction raising more difficult.

Progressive lowering with the MLIR

The Multi-Level-Intermediate-Representation is a hybrid IR framework guiding the design of customizable intermediate representations with little built-ins and fundamental concepts. It proposed a more flexible approach, where the way of defining DS type systems and operations known from the traditional IRs is standardized with dialects(domain-specific IRs). In MLIR, a code module comprised of the multi-level IR can contain a mix of many dialects, where the membership of the operation to a dialect is explicitly indicated with an operation's name.

By doing so, the framework reduced the granularity of a lowering step to single operations rather than entire code modules. Furthermore, it significantly reduces the cost of building domain-specific compilers, mainly due to two levels of reuse. The first is provided by the shared infrastructure dedicated to defining the domain-specific representations with dialects. The second, which lies at the core of this project, is the reuse of techniques between dialects, which are present within the framework. As a consequence, the new additions to the dialect's set can include passes and operations belonging to other dialects, but also simply lower to one or more other dialects partially or completely. This introduces an entirely new level of flexibility in the ordering of techniques, which belong to separate DS compilers.

To give an illustrative example, the MLIR's linalg dialect that defines matrix operations can be lowered via Affine dialect but it also has an alternative of lowering independently of the Affine. Figure 1.3 shows the partial lowering of linalg.matmul operation, where the lower-level Affine dialect is used to implement tiling of the linalg.matmul operation. Based on the high-level semantics ingrained in the matmul operation, it can be defined that the inner loops, which result from tiling, can be expressed with linalg.matmul rather than just loop element-wise implementation. The example outlines how the principle of partial and progressive lowering can help to preserve high-level semantics, while accessing constructs from the lower level dialect and how the compiler's passes can be composed.

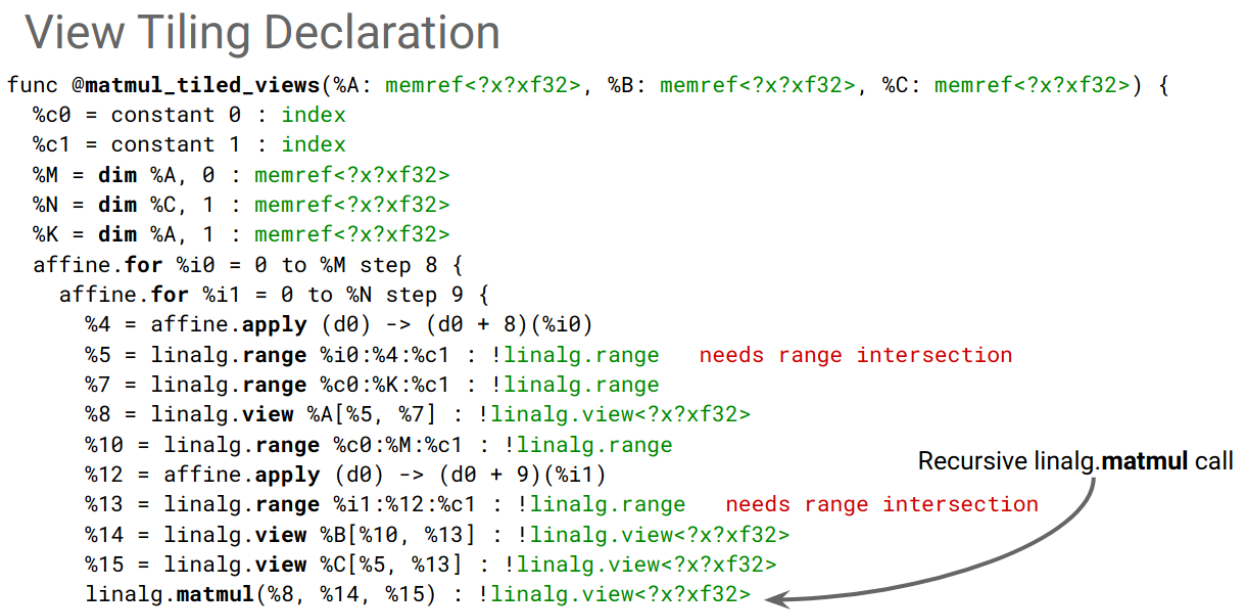

Figure 1.3: Tiling optimization pass preserves the high-level semantics of matrix multiplication with partial lowering. Image from the MLIR tutorial [17].

1.5 Contributions

The thesis's primary contribution is the design of a method performing abstraction raising, which enables domain-specific optimizations in multi-domain applications. Specifically, my contributions are:

- A raising mechanism's proposal for general-purpose languages, thus enabling domain-specific compilation for general-purpose code by reusing already existing domain-specific optimization available in MLIR.
- Proof of concept with PET-to-MLIR's implementation and currently available abstraction raising results for matmul/matvec based on Polybench/C kernels compared to results from the current state-of-art general-purpose loop optimizer PLuTo.

Furthermore, our design by extending the polyhedral ISL-based tools resulted in a code generator that glues together the mainstream Integer Set Library (used by both GCC and LLVM) polyhedral framework with the polyhedral representation - Affine dialect in the novel MLIR compiling framework.

The thesis's secondary contributions are:

- An entry point to the MLIR framework for a subset of C or C++ code, thus increasing programmer productivity, as with PET-to-MLIR they gain an automated tool to enter the MLIR framework starting from C or C++ code. Before the PET-to-MLIR the researchers had to lower the C or C++ code

manually to experiment with the polyhedral simplified form within MLIR or with the MLIR compiler infrastructure in general [4].

- A demonstration of Polybench/C kernels currently handled by the PET-to-MLIR code generator.

Chapter 2

Related Work

This chapter introduces the related approaches, which enable domain-specific optimizations with function detection. The discussion is concluded with the thesis's contributions in the context of the existing solutions.

2.1 Frameworks for function detection

Chelini et al. and Ginsbach et al. have already compared works describing various detection mechanism implementations [6, 10]. The works indicate three main pathways we could take for detecting a loop's functionality based on its semantic properties rather than just textual representations (Req. 2) namely using: the polyhedral model, idioms, or program synthesis. As described by Nugteren et al., the polyhedral model being a tool designed to express looping behaviors is a fitting tool for our purposes being formally defined, complete, and able to capture all information compared to the original C code [20]. The main problem indicated in the paper is the framework being non-intuitive to use. This, however, was significantly improved upon with the introduction of schedule trees and Loop Tactics, which operates on schedule trees and offers declarative C++ API. The idiomatic framework lacks a unified API implementation. The program synthesis on the other hand requires defining postconditions, however, based on the publicly available documentation it is not clear how to capture examples for the linear algebra domain as opposed to the polyhedral and idiomatic approaches. For these reasons, we choose to rely on the Loop Tactics, which demonstrates its work on Polybench covering the domains of interest for us.

Having chosen the framework for pattern detection, in the further subsections we discuss the various detections' integration methods applied in other works and their range of enabled optimizations based on those methods.

2.2 Enabling domain-specific optimizations with direct transformations

The most direct approach to the optimization based on pattern detection is to locally optimize upon detections. Such an approach either involves customized DS optimizations or mainstream vendor libraries. The main reason to optimize without abstraction raising is to achieve high performance within the general-purpose frameworks without creating dependencies to stand-alone DS frameworks. For instance, Polly automatically detects GEMM from LLVM IR and applies predefined manual optimizations achieving performance similar to openBLAS. Loop Tactics took it a step further by introducing the concept of matcher-builder pairs. It developed a declarative API to systematically define both searched program properties with matchers and customized transformations triggered upon detection with builders.

A baseline strategy that can be applied by both Polly and Loop Tactics is simply relying on library calls when they are available for the targeted platform. That is the main and only strategy for many other works [6]. For instance, Sevastini et al. and Kessler et al. perform pattern matching at the AST level and replace the detected subtrees with optimized library calls [14, 26]. As already explained, the main drawback of these approaches is optimizing single kernels due to the transformation happening upon the detection.

2.3 Enabling domain-specific optimizations with high-level code generation.

The alternative approach that can be found in the literature is to perform complete abstraction raising, that is, to generate high-level DSL directives as a consequence of the detection. Pottengar et al. introduced syntactic pattern-recognition to transform the code fragments into reduction operations defined in Polaris compiler's DS IR [3, 22]. Menon et al. introduced text-based pattern detection as a part of a source-to-source Matlab compiler [18]. The compiler transforms Matlab's loop nests into a mixture of high-level matrix operations and low-level loops. It does that by rewriting specific statement patterns into an algebraic language called AMF. The AMF representation defined a set of rewriting rules (axioms), which perform aggressive rewriting to reduce computation complexity based on algebraic properties.

The closest approach to our work regarding the portability was HERD presented by Cheung et al. [7]. They incorporate program verification techniques to detect the input code's fragments that can be translated into a chosen DSL. Similar to our approach, their method is not designed for a specific domain, and they do not focus on specific DSLs in their compiler's design. In HERD, the developers can include a DSL function of interest for raising by defining so-called kernel-translators. The compiler will attempt to search parts in the input code that can be reconstructed with the available translators. They focus on translating stencils into Halide DSL and database queries into SQL. The main difference between HERD and our approach is that PET-to-MLIR completely separates the concern of high-level code generation from the detection mechanism. We believe it is an advantage in terms of simplifying the potential use of our framework for developers and preventing the translation's quality from affecting the detection procedure. The authors themselves emphasize the necessity to study the ease of developing the kernel translators. It is difficult for us to assess the translators' programmability as the examples of such translators for the Halide and SQL are not shown. Simultaneously, we rely on Loop Tactics declarative C++ API for defining the searched patterns, and for the translation, we use constructs belonging to the more mainstream project MLIR. Furthermore, their work targets multiple domains through separate DSLs, while our work enters one framework with the whole input code and aims to combine multiple domains of interest without fragmentation.

2.4 Conclusions

Contrary to the area of algorithm classification/pattern detection, there are not many works present in the literature, which propose a holistic approach from pattern detection to optimizing via abstraction raising. The most versatile compiling flow involving abstraction raising intended for multi-domain use is HERD. To the best of our knowledge, there are no attempts to perform abstraction raising involving the MLIR framework. The main difference between our goals and HERD is the focus on polyhedral-friendly loop nests and the incorporation of the reuse offered by the novel framework. Contrary to HERD, we do not intend to target stand-alone DSLs but rather work with dialects on the operation level.

Chapter 3

Approach

In this chapter, we describe the proposed solution that addresses the problem statement. The starting point is the observation that we need to incorporate the loop pattern detection with the MLIR framework. As discussed in the Related Work chapter, for that purpose we will use an existing tool - Loop Tactics, which allows for the schedule tree's traversal search for predefined loop patterns. However, The MLIR framework currently lacks a front-end for the ISL's polyhedral framework, including Loop Tactics, and C or C++ programs in general. The researchers have to lower the C/C++ code manually to experiment with the polyhedral simplified form within MLIR [4]. This means we have to propose a way to generate the MLIR code from the schedule trees. For this reason, we analyze the program representation used by Loop Tactics and choose an adequate entry-point to MLIR for the polyhedral-friendly C/C++ loops.

After the incorporation of the frameworks, i.e., enabling the valid MLIR code generation from Loop Tactics' output (schedule trees), we will use MLIR's standardized interface (dialect's operation builders) to generate high-level operations. These operations as a part of higher-level DS dialects should be subjected to existing dialects' DS passes. It must be noted, however, that even though the MLIR encourages the development of more DS optimizers, the MLIR is still a novel tool and heavily developed simultaneously with this project. This means the full potential of the framework's core dialects and independently developed dialects is not available. For this reason, and a limited scope of the project, we will attempt to prove the concept of abstraction raising within the MLIR framework by identifying currently accessible optimization candidates based on the techniques listed in Section 1.3.

3.1 Entry point selection

Figure 3.1 illustrates current potential entry points along with our proposal for the C/C++. Out of the illustrated dialects, the Affine dialect is the best candidate for the entry-point as it is by design a simplified polyhedral form. The dialect, compared to the schedule trees, is able to express the same computations but is more imperative. For example, it has a defined loop structure with iterators, which simplifies code lowering. However, it also makes more complex transformations, like skewing, more problematic compared to transforming schedules in a traditional polyhedral form [16]. In order to generate the Affine dialect from other polyhedral representations, for instance, ISL's schedule tree, we will need to establish specific loop structure that implements the encoded schedule.

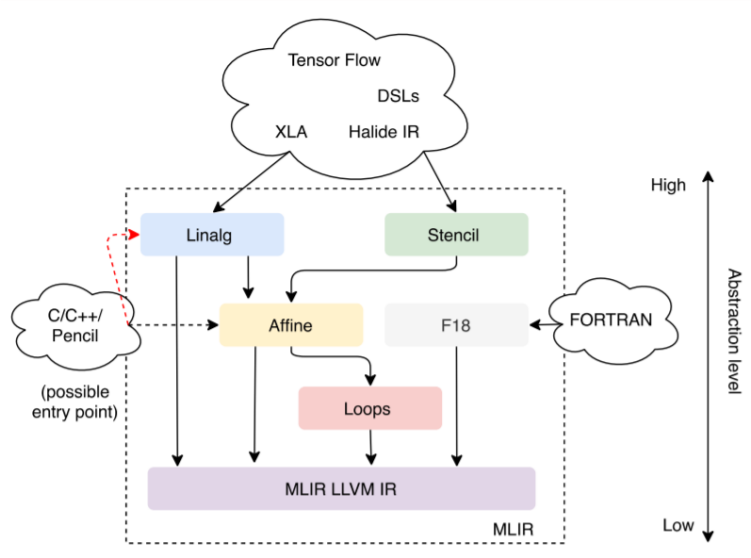

Figure 3.1: A possible C/C++ entry point to MLIR from LLVMIR to Affine. The red arrow indicates a possible raising of the entry-point for detected functions to Linalg.

3.2 Loop Tactics in current compiling flows

Loop Tactics operates on the ISL-based schedule tree representation, and it is both integrated with Clang and Polly. Based on the available tools, it is possible to modify schedule trees with Loop Tactics and generate either LLVM IR or a C code. The LLVM IR is at the abstraction level significantly lower than the C/C++. Therefore, using this flow would mean lowering the entire considered code module to a representation that does not contain, for instance, loops and raising back parts all the way to representations abstracting from loop constructs with operations like GEMM.

In the latter case, the optimized C code has to be lowered with a complete general-purpose compiling flow. This is also a sub-optimal solution, and thus we need to implement a new flow that generates the MLIR code according to our needs. However, it is useful to observe at this point that the main difference between the schedule tree representation and the Affine dialect is the slightly lower abstraction level, i.e., specific loop structure implementing the schedule, and that the loop structure also has to be reconstructed when re-generating C code from the ISL's schedule. This means we can re-use parts of the ISL's C source-to-source flow and imitate it to generate the Affine dialect instead.

Therefore, we modify the C's source-to-source flow to a PET-to-MLIR flow that utilizes existing tools and a new PET-to-MLIR code generator to enter the compiling framework from the C/C++ source. In the following subsections, we introduce the flow, for which our prototype has been by now published [15]. The tool flow is presented in Fig. 3.2 and each stage is discussed respectively.

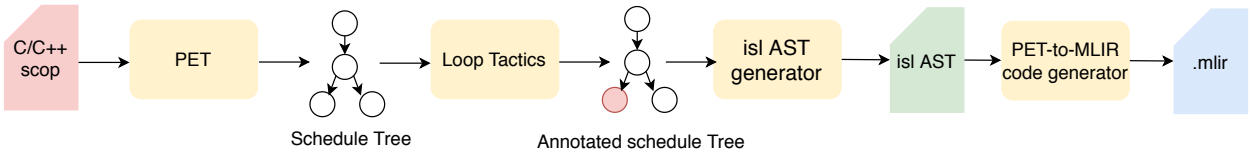

Figure 3.2: The proposed flow generating to MLIR with raised entry point. Image from Komisarczyk et. al. [15]

3.3 Polyhedral extraction

The Polyhedral Extraction Tool (PET) begins the code conversion. PET is a library, which extracts a polyhedral model from the C/C++ source. PET accepts the C/C++ code and is the only tool used within the proposed flow that imposes a limitation on the input. The code that satisfies the conditions to enter PET has to be expressible with the polyhedral model's constructs. Such parts of programs are formally defined as static-control parts (ScOPs). PET can automatically detect ScOPs in the input program. Users can also label parts of the inputs code as ScOPs, in which case PET will report the instances not meeting the requirements for the extraction. The tool includes Clang, which, in consequence, is the actual C front-end of the tool-flow and provides full C99 support and variable vector length arrays. Additionally, Clang introduces its expressive diagnostic toolkit. Next to Clang, another significant library used by PET is Integer Set Library that provides support for polyhedral model encoding, analysis, and transformation. When compared to other established polyhedral extractors like Clan PET provides the most extensive support [31].

The output of the PET is a schedule tree representing the polyhedral model of the input ScOP and a ScOP's context. The context contains additional information necessary to reconstruct executable code from the abstract polyhedral representation, i.e., operations corresponding to statement labels, array names, and array extent. Typically ScOP's context is used to enforce allocation of the size in the source program rather than based on the schedule analysis.

The other part of the output - the schedule tree - for a GEMM program has been presented in the Schedule Trees section, Fig. 1.1. The program encoded now with schedule trees is fed to Loop Tactics for the pattern-based analysis.

3.4 Schedule tree annotation

Within Loop Tactics, we can use structural and access matchers that allow us to express computational patterns. When Loop Tactics during the schedule tree traversal detects the pattern, the tool replicates the matched subtree with a mark node on top. We refer to such trees as annotated, where the information annotated with the mark node will be used in later stages. The annotation conveys the name of the detected functionality and ordered arguments, which allow the construction of the high-level operation call. The arguments could also be extracted at the later stage of the MLIR code generation, however, the access matchers are by design the perfect tool for determining the role of individual array identifiers robustly.

Figure 1.1 shows the input schedule tree, where the red lines mark the right subtree scheduling S2. For that subtree, we would like to match a matrix multiplication. However, to ensure robustness and scaling of the approach it is necessary to only define basic properties of the matmul itself rather than define a set of patterns that would depend on the parent nodes. For this reason Loop Tactics first performs normalization based on the ISL's dependence analysis. The statements are separated respectively into two- and three-dimensional loop nests. After such normalization, the main structural properties of the GEMM have been exposed: a three-dimensional band followed by a statement by a statement with a GEMM access pattern.

Listing 2 presents a matcher for GEMM. It first searches for a band node that satisfies the callback `is3D`. Next, it will check the property expressed in a callback containing the access matcher: `hasGemmPattern`. The access matcher defines a pattern with three reads to different arrays (line 12 to 15), one write access (line 17). Within the accesses the access subscripts must satisfy the access pattern $[i, j] \rightarrow [i, k][k, j]$. Lastly, after verifying the operations performed by the matched statement the tool returns the annotated schedule tree.

```
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;
};

auto matcher =
  band(
    and_((is3D,hasGemmPattern,isPermutable),
    leaf());
```

Listing 2: Structural matcher, access relation matcher for the right-most subtree in Fig. 1.1.

3.5 ISL AST generation

Handwritten code after the extraction typically results in a regular schedule. By looking at the GEMM example or other simple schedules it might seem feasible to re-generate the loop structure and implement loop iterators based on the original source. The main problem with that approach is the fact, it prevents most of the schedule transformations during the stage and requires assumptions about what ISL can do during its normalization at the annotation stage. A more robust alternative is using the ISL AST generator. The ISL AST generator is the current state-of-art tool in code generation from polyhedral models. It generates code from arbitrary polyhedral schedules, which requires using heuristics for choices during the generation that affect resulting code quality. An example of such a choice is generating a loop nest for existentially quantified dimensions. It can be done either by an approximation or exact quantifier elimination. The ISL's generator compared to others typically generates significantly smaller code [11]. The output - ISL AST - for the GEMM example is parsed shown in Fig. 3.3.

The module node represents an equivalent of the MLIR's module, which is a top-level container isolated from the external values. Contrary to the schedule tree, the ISL AST has an implicit sequential ordering at each tree-level. In the example, the module is followed first by the `iterator`, which defines a specific for-loop instruction. The `iterator` always specifies a fresh symbolic induction variable, explicit (possibly symbolic) initial value, affine condition, and a positive increment. All children of the iterator belong to the same and possibly nested loop nest. As can be noted in the presented AST, the loop nests are now separated. Each subtree is finalized with at least one `user` node. These nodes fill the control flow with the actual statement execution.

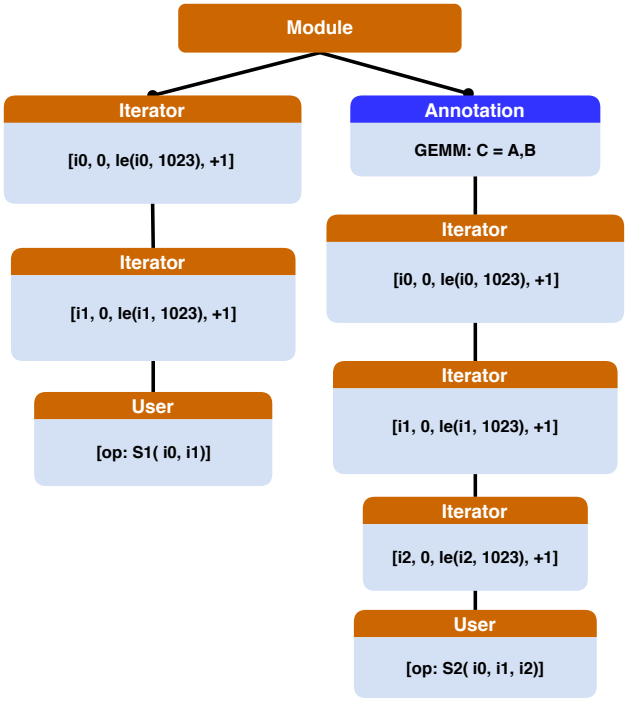

Figure 3.3: ISL AST for GEMM after normalization + annotation.

3.6 MLIR code generator

With the ISL's AST generated we now have the loop structure and the PET statements that abstracted from the original operations. The missing step is the emission of the Affine dialect's instructions implementing control flow and implements statements using any available dialect. We need to introduce a tool, which will scan through the nodes and for each encountered iterator node will generate an affine.for instruction. Then for each leaf within the ISL AST, the PET's SCOP structure has to be accessed to retrieve operation based on statement identifier. The tool will recursively generate the MLIR instructions for each argument. An example of the Affine code resulting from scanning the GEMM's ISL AST without using the annotation is shown in Listing 3.

```
func scop_entry(%arg0: memref<1024x1024xf32>, %arg1: memref<1024x1024xf32>,
                %arg2: memref<1024x1024xf32>, %arg3: f32, %arg4: f32) {
  affine.for %arg5 = 0 to 1024 {
    affine.for %arg6 = 0 to 1024 {
      %0 = affine.load %arg2[%arg5, %arg6] : memref<1024x1024xf32>
      %1 = mulf %arg4, %0 : f32
      affine.store %1, %arg2[%arg5, %arg6] : memref<1024x1024xf32>
      affine.for %arg7 = 0 to 1024 {
        %2 = affine.load %arg0[%arg5, %arg7] : memref<1024x1024xf32>
        %3 = mulf %arg3, %2 : f32
        %4 = affine.load %arg1[%arg7, %arg6] : memref<1024x1024xf32>
        %5 = mulf %3, %4 : f32
        %6 = affine.load %arg2[%arg5, %arg6] : memref<1024x1024xf32>
        %7 = addf %5, %6 : f32
        affine.store %7, %arg2[%arg5, %arg6] : memref<1024x1024xf32>
      }
    }
  }
  return
}
```

Listing 3: Affine IR emitted for Listing 1 without using annotations.

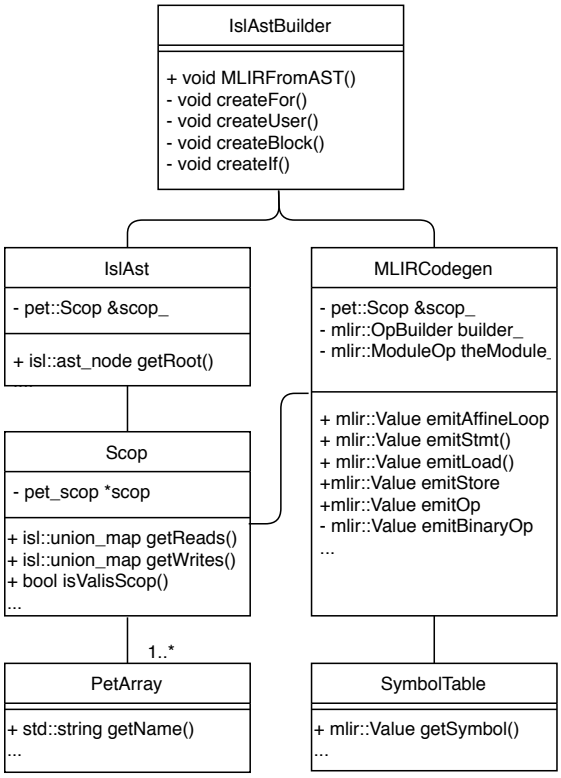

Figure 3.4: pet-to-mlir class diagram.

Figure 3.4 illustrates the relations between all the key elements containing program information with the class diagram for PET-to-MLIR generator. The topmost class (`IslAstBuilder`) provides a public method `MLIRFromAST` that represents the entry point in the translation process. The class `IslAstBuilder` has private references to `IslAst` and `MLIRCodegen`. The former is a handy class that exposes methods to interact with the ISL's AST. The latter encapsulates the bulk of the logic to generate affine IR by walking the AST and the parse tree for each statement. `MLIRCodegen` uses `SymbolTable` to keep track of the symbol emitted during code-generation. Finally, the `Scop` class is a wrapper around a `pet_scop*` and contains helper methods to extract information from the detected scop. The class Scop may contain one or more PetArray. Each PetArray is a wrapper around a `pet_array` and models a memory access (i.e., multi-dimensional tensor or scalar).

3.7 Covered domains

The Polybench/C benchmark consists of 30 kernels from multiple application domains. Within the project's timeline we implemented most generation for kernels from linear-algebra and stencil domains except `durbin, cholesky, gramschmidt` and `ludcmp` [15]. In Table 3.1 we present 18 kernels, which are currently handled by the tool and generate valid MLIR code. The correctness analysis is limited to verifying the results against the GCC compiler to ensure that the further measured results correspond to valid computation. Including other kernels would require implementing support for the other C constructs appearing in these kernels like constant accesses to arrays. However, these additional kernels did not introduce any new candidates for raising compared to the already handled kernels. Therefore, they were postponed and will be part of the Future Work, Section 6.2.

Table 3.1: Tested kernels from Polybench/C representing the range of currently covered domains by PET-to-MLIR code generator.

Benchmark	Description
2mm	2 Matrix Multiplications (alpha * A * B * C + beta * D)
3mm	3 Matrix Multiplications ((A*B)*(C*D))
atax	Matrix Transpose and Vector Multiplication
bicg	BiCG Sub Kernel of BiCGStab Linear Solver
doitgen	Multi-resolution analysis kernel
gemm	Matrix-multiply C=alpha.A.B+beta.C
gemver	Vector Multiplication and Matrix Addition
gesummv	Scalar, Vector and Matrix Multiplication
jacobi-1D	1-D Jacobi stencil computation
jacobi-2D	2-D Jacobi stencil computation
lu	LU decomposition
mvt	Matrix Vector Product and Transpose
seidel	2-D Seidel stencil computation
symm	Symmetric matrix-multiply
syr2k	Symmetric rank-2k update
syrk	Symmetric rank-k update
trisolv	Triangular solver
trmm	Triangular matrix-multiply

3.8 Optimization opportunities within contemporary MLIR dialects

In this section, we discuss the optimization opportunities present in the MLIR to choose a dialect offering reuse of automated optimization and generate high-level operations. We require the candidate dialects to provide us with the operation builders in order to demonstrate the generation within MLIR with a standardized declarative API.

```
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   %0 = llvm.mlir.constant(0 : i32) : !llvm.i32
12   %1 = llvm.mlir.constant(0 : i32) : !llvm.i32
13   linalg.matmul(%0, %1, %arg2, %arg0, %arg1, %arg3, %cst) :
14     (!llvm.i32, !llvm.i32, memref<1024x1024xf32>, memref<1024x1024xf32>,
15      memref<1024x1024xf32>, f32, f32) -> ()
16
17   return
18 }
```

Listing 4: Affine + Linalg IR emitted for Listing 1 with running Loop Tactics.

The MLIR currently provides an LLVM IR dialect, which can be automatically converted to the traditional LLVM IR. This means that the generated code can access the general-purpose optimization available at the C level with Clang including Polly optimizer along with other LLVM's automated optimizations for any dialect with a defined lowering path to the LLVM's dialect.

The most mature high-level dialect, with which it is possible to enter the MLIR with a higher-level abstraction than C/C++, is Linalg. Linalg was originally limited to semantics derived from the linear algebra domain. As stated in the dialect's rationale, including the operations like Conv or Matmul "enables building high-level productivity, i.e., a first codegen solution that leverages both compiler optimizations and efficient library implementations" [1]. For instance, the dialect allows tiling strategies, which preserve an opportunity for library calls as tiling Matmul operations in the Linalg dialect results in a program implementing smaller instances of the Matmul operation. The preservation of the semantics integrates legality assumptions in the operations' types. For example, further tiling can be applied without the affine analysis and be followed by other semantic-based optimization stages afterward. The Linalg dialect's role makes it a candidate for abstraction raising to enable complexity reduction based on the matrix semantic being present, BLAS calls, or automated offloading to accelerators. Using the builders API, the function replacing the original loop nest with the linalg.matmul operation can be done based on the annotations with less than 10 LoCs. Listing 4 presents the result.

3.9 Summary

As described in the project's motivation and problem statement, the main goal of the project is to enable domain-specific optimizations by entering progressive lowering within the novel compiling framework MLIR. We propose a tool flow comprised of existing tools and our PET-MLIR code generator. By doing so we want to provide the C/C++ front-end to the MLIR for the polyhedral-friendly code. The tool-flow extracts polyhedral subset from C/C++ code and allows arbitrary schedule manipulation at the schedule tree stage. The baseline functionality generates the MLIR code at the Affine level and the correctness is tested against a GCC-compiled Polybench/C benchmark. The tool is designed to raise the abstraction and generate high-level operations for a set of Polybench kernels and show that we can delay the routine calls and implement them in the MLIR. In the following Chapter 4 we experiment with the MLIR's opportunities for the heterogenous code module containing Affine and Linalg to trigger the optimizations present in the dialects like matmul-chain-reordering. This way we can show specific performance/retargetability gains without implementing any DS optimization strategies ourselves.

Chapter 4

Experiment Setup

In this chapter, we describe the hardware and software used for obtaining results. Furthermore, we explain the measurement method together with the performance metrics used.

4.1 Hardware

The experiment execution and measurements are performed on a system comprised of:

- Intel i9-9900K 3.60GHZ
- GeForce RTX 2080 Ti

4.2 Software

- PET-to-MLIR[1]
- GCC 7.4.0
- Clang 12.0
- PLuTo 0.11.4
- PolyBench/C 4.2.1

4.3 Performance Metrics

To compare the obtained performance in the context of the machine's theoretical peak we will report the obtained performance in GFLOPS/s per kernel and calculate the theoretical peak GFLOPS for the Intel processor as follows:

GFLOPS = the total number of cores on the machine · CPU frequency in GHz · float operations per CPU cycle,

where float operations per CPU cycle are defined for the SIMD instructions with: fpc = floating point numbers in one SIMD register · number of SIMD operands · 2(assuming a MACC operation per cycle).

The recipe for the CPU yields the following parameters:

$$fpc = \frac{256}{32} \cdot 2\, operands \cdot 2\, flops = 32$$

$$GFLOPS/s = 8\, cores \cdot 3.6\, GHz \cdot 32 = 921$$

$$GFLOPS/s = 1\, core \cdot 3.6\, GHz \cdot 32 = 115.2$$

[1] github.com/Komisarczyk/mlir/tree/LT : 44a1c62

4.4 Measuring generated MLIR code's performance

We evaluate the applicability of our tool by lowering to the Affine different kernels from the Polybench 4.2 benchmarks suite. We limit the experimental part to the Linalg dialect as it is the most mature DS representation within the MLIR. Therefore, for GEMM-like kernels (`2mm`, `3mm`, `gemm`, `mvt`, `atax`, `bicg`) we additionally run Loop Tactics to lift the entry point and thus exploit domain-specific optimizations as the invocation of vendor-optimized routines and/or matrix properties.

4.4.1 Affine code generation

First, we verify that the generated Affine does not impair the general-purpose performance and that we generate correct behavior. 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 produced values of each kernel's output matrices with the output of the GCC compiler. We compare the code quality of the Affine code by measuring the average execution time of five independent runs and compare it with the Clang -O3 optimization mode's execution times.

To compile the input C kernels via our flow we use the PET-to-MLIR to generate the Affine code, then use `mlir-opt` and `mlir-translate` to lower towards LLVM IR, and finally, compile the resulting IR with the Clang -O3.

Table 4.1: FLOPS used for GFLOPS/s calculation based on the large Polybench/C problem size.

Benchmark	Operation calculation	FLOPS for large problem size
2mm	$5{\cdot}N_i \cdot N_j \cdot N_k + N_i \cdot N_j$	3,312,000,000
3mm	$6{\cdot}N_i \cdot N_j \cdot N_k$	5,400,000,000
atax	$4{\cdot}N_i \cdot N_j$	15,960,000
bicg	$4{\cdot}N_i \cdot N_j$	15,960,000
doitgen	$4{\cdot}N_i \cdot N_j \cdot N_k \cdot N_l$	1,075,200,000
gemm	$3{\cdot}N_i \cdot N_j \cdot N_k + N_i \cdot N_j$	2,640,000,000
gemver	$10{\cdot}N_i \cdot N_j + N_i$	40,002,000
gesummv	$4{\cdot}N_i \cdot N_j + 3 \cdot N_i$	6,763,900
jacobi-1D	$6{\cdot}N_i \cdot N_j$	5,994,000
jacobi-2D	$10{\cdot}N_i \cdot N_j \cdot N_k$	8,424,020,000
lu	$\frac{1}{6}N_i \cdot N_j \cdot N_k + \frac{3}{2}N_i \cdot N_j + \frac{1}{3}N_k$	8,002,000,000
mvt	$4{\cdot}N_i \cdot N_j$	16,000,000
seidel	$9{\cdot}N_i \cdot N_j \cdot N_k$	17,964,018,000
symm	$\frac{5}{2}N_i \cdot N_j \cdot N_k + \frac{5}{2}N_i \cdot N_j$	1,800,000,000
syr2k	$\frac{1}{3}N_i \cdot N_j \cdot N_k + \frac{7}{2}N_i \cdot N_j + \frac{1}{2}N_k$	2,880,000,000
syrk	$\frac{3}{2}N_i \cdot N_j \cdot N_k + 2N_i \cdot N_j + \frac{1}{2}N_k$	1,440,000,000
trisolv	$N_i \cdot N_j$	4,002,000
trmm	$N_i \cdot N_j \cdot N_k$	1,200,000,000

4.4.2 GEMM-like kernels and Loop Tactics

In this subsection, we describe the measurements including the abstraction raising with PET-to-MLIR + Loop Tactics, and compare with PLuTo. PLuTo, a general-purpose loop optimizer, is capable of parallelizing code and utilizing openMP instructions to enable the use of multi-threading. Therefore, for the following measurements, we include the multi-threading and compare with both PLuTo and the machine's theoretical peak performance.

Vendor-optimized routines

The first part of the experiment aims to present the raised code's performance benefits within our flow is by generating the linalg.matmul, linalg.matvec, and linalg.vecmat operations. The MLIR has a prototype lowering pass towards naive loop implementations of these functions. For this reason, we reason we simply replace the MLIR's BLAS interface with MKL functions calling the MKL or cuBLAs routines based on the

linalg operation type and its arguments.

The linalg.matmul and linalg.matvec operations are directly implemented with the MKL's sgemm and gemv routines. The linalg.vecmat operations for the Intel processor are implemented with sgemm routine and an extension of the vector's dimensions, which generalizes it to matrix multiplication.

In our programs, data allocation is encoded in C-convention with row-major data layout. This means the translation to CUDA subprogram meant to run on GPU requires special attention in order to prevent errors and sub-optimal data manipulation. Therefore, for the cuBLAS implementation of all three linalg operations, we use the cuBLAS' sgemm routine in order to use efficient data layout and intermediate operations' ordering to avoid redundant data transpose due to the CUDA's column-major default data layout. So to compute the resulting C matrix from C = Ax we use the property:

$$Ax \xrightarrow[layout]{column\ major} x^T A^T \xrightarrow[x^T A^T=(Ax)^T]{cuBLAS\ sgemm} C^T \xrightarrow[layout]{row\ major} C$$

We can directly detect the computational motifs and replace them with routines for the following Polybench kernels: 2mm, 3mm, gemm, atax, bicg, gemver, and mvt.

In order to measure the execution times of the library routine accurately, we follow the guidelines provided by the MKL library. This means discarding the first routine measurement and averaging over a such number of runs that results in a total measured workload of about a second. This means executions of 100-10000 times the considered routine depending on how long is the original execution time. The main concern about the recommended measurement method is the difficulty to ensure the inclusion of cold cache misses. However, according to the tests, while discarding the first measurement and allocating new memory for the consecutive run reduces the overhead related to the allocation of SIMD buffers, but still results in a high variance in measured times due to the thread spawning for multi-threading MKL. The variance is relatively high (up to 50% of measured execution time for level 2 BLAS routines). For this reason, we focus on Polybench's large problem size, which is designed to not fit in the level-3 cache (25MB problem sizes vs 16 MB shared cache). Taking a minimum function is an option to assume only the fastest operating system's service, however, that such approach would provide results that do not correspond to the realistic performance achieved by simple code replacement with routines.

To obtain the more accurate measurements of a varying number of the library calls, we separately measured the value initialization parts of kernels that are not expressed with BLAS calls and summed them with the averaged routines' execution times.

Matmul reordering

3mm kernel in Polybench/c 4.2.1 performs a chain multiplication in the following order:

$$A = (AB)(CD)$$

For the default problem sizes, this order is sub-optimal, however, the difference in dimension sizes of the input matrices are not large. This means that re-association even though beneficial, results in gains close to the measurement deviation. For this reason, we modify the default problem sizes by reducing one of the input dimensions and measure the performance gains compared to the calculated GFLOPs reduction.

When applying the dynamic reassociation to the modified problem size, the resulting order of operations is:

$$A = A(B(CD))$$

Problem size	Polybench product order's FLOPS	Reordered FLOPS	Ratio
Medium 180x190x200x21x220	15195600	2431800	6.25
Large 800x900x1000x110x1200	1689600000	305800000	5.53
Extra Large 1600x1800x2000x220x2400	13516800000	2446400000	5.53

The table shows the reduction of workload by a factor between 5-6x

Chapter 5

Results

This chapter summarizes the Research Questions addressing the effectiveness requirement (Req. 4) of the Problem Statement, and presents the corresponding results.

5.1 Research Questions

- How does our PET-to-MLIR flow affect the quality of the general-purpose code (without detection) in terms of achievable performance, compared to applying a production compiler to the original source code? (Section 5.2)
- How does the PET-to-MLIR affect performance with abstraction raising when vendor-optimized routines are available? (Section 5.3)
- How can the PET-to-MLIR affect performance with abstraction raising using domain-specific analysis? (Section 5.4)

5.2 Affine general-purpose code quality

Figure 5.1 presents the obtained GFLOP/s from the Affine representations of the original kernels generated with the PET-to-MLIR flow. It can be observed that based on the results it is not possible to clearly label the better compilation flow across all types of the presented kernels. On average the Clang starting from the source C code achieves 4.25% better GFLOPs/sec. The greatest gap in performance in favour of pure Clang compilation is reported for the syrk kernel with 36.65% more GFLOPS/s. On the other hand, the most extreme example in favour of PET-to-MLIR + Clang was reported better for the heat3-d with a relative advantage of 11.62%. By looking at similarities and constructs used at the stage of Affine code generation we conclude that the issues derive from a faulty LLVM IR generation with the mlir-translate tool. Failure to expose some features in the LLVM IR's generation by the MLIR, e.g, data layout specification and target triple, can have a major effect. [24]

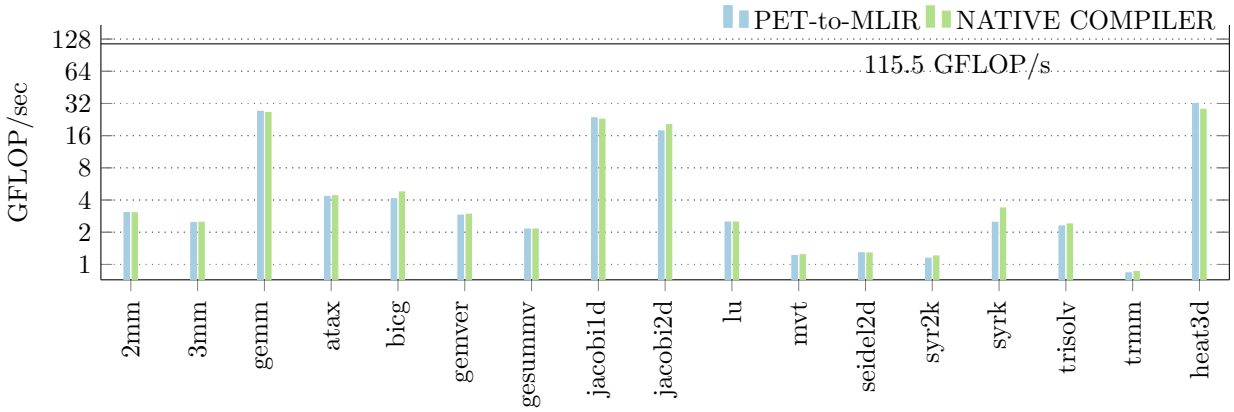

Figure 5.1: Performance obtained by lowering C code to the Affine in MLIR via PET-to-MLIR.

5.3 Code replacement with vendor-optimized routines

Figure 5.2 shows the GFLOP/s gains via simple code replacement based on the Linalg abstraction level. It can be observed that as discussed in Section 1.4.1 the ability to replace the DS parts of the input C/C++ kernels produces high-performance gains that cannot be contributed only to basic multi-threading. Furthermore, the PLUTO shows that indeed it is able to obtain respectable performance gains using general-purpose analysis, however, a simple application of the default settings without the tool-aware user's tuning can impair the performance drastically (*atax*, *bicg*, *gemver* and *mvt*). PET-MLIR + CUBLAS shows the gains derived from using an accelerator when high parallelism is available by reaching GFLOP/s equal to the host's theoretical peak. For level 2 BLAS kernels, CUBLAS still shows significant gains relative to the pure general-purpose compilation flow, however, the overhead due to the data transfer makes it less efficient than the MKL BLAS.

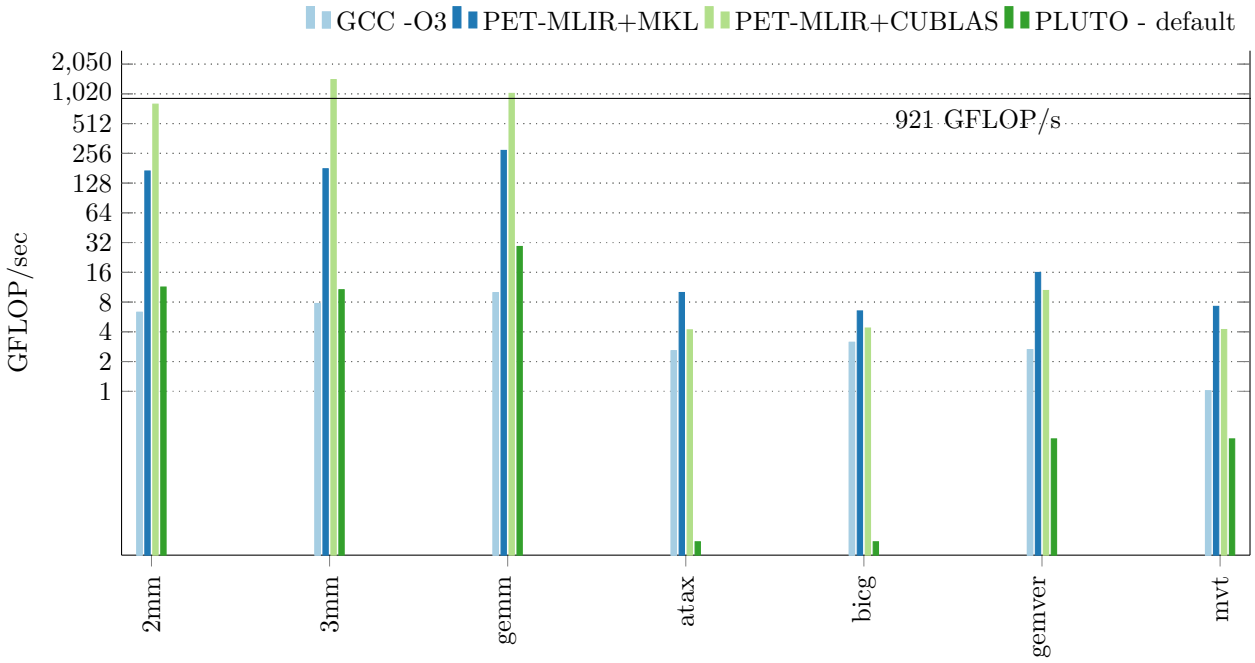

Figure 5.2: Performance comparison between general-purpose optimizers and code replacement for the linalg operations.

5.4 Domain-specific analysis - matmul reordering

For this experiment, since the number of FLOPs in the given workload drops with the Linalg optimization pass, instead of GFLOPs/sec we use execution times as a better indication. Performance gains for the matmul reordering are shown by execution times reported in Table 5.1. As reported, the execution times are reduced by factors between 3-4, while the workload was reduced by 5-6 times. We contribute this effect to the memory overhead, especially cold cache misses, which are not accounted for in the FLOP calculation and scale differently compared to FLOPs based on the kernel type.

Table 5.1: Reduced execution times for imbalanced 3mm input matrices via matmul reordering.

	Polybench/C	**Reordered**	**Unit**	**Time Ratio**
Medium	57.15	14.69	*ms*	3.89
Large	4.65	1.07	*sec*	4.34
extraLarge	21.99	6.95	*sec*	3.16

Chapter 6

Conclusions

This chapter concludes the proposal together with the work's results and discusses the further potential of the proposal in Future Work Section.

6.1 Conclusions

We refer to the requirements introduced in Section 1.3 to assess whether the goals of the project have been fulfilled.

We have created a prototype code generator, which implements the flow proposed in Chapter 3. Chapter 5 shows the implemented solution accepts C/C++ SCoPs and automatically (Req. 1) generates a set of the following high-level operations: linalg.matmul, linalg.matvec, linalg.vecmat (Req. 5). We incorporated the ISL's polyhedral framework for the detection mechanism to provide a multi-domain solution that allows expressing (Req. 3) and generating high-level ops (Req. 2) for all domains represented in Polybench/C. This is, however, not reflected in the presented results due to the limited number of mature DS optimizers. Although it must be noted that the linear-algebra examples do not introduce any domain-specific assumptions with regards to the operations' generation. Hence, apart from the implementation effort, there are no obstacles to generating stencil operations. Finally, in Result Sections 5.3 and 5.4 we show the enabled improvements derived from DS-specific analysis within the MLIR's DS optimizer (Req. 4,5).

We conclude that within the MLIR, the abstraction raising becomes a feasible and portable solution improving on the major shortcomings of other raising flows discussed in the Related Work. Moreover, compared to PLuTo it offers an opportunity for a conscious choice of optimization strategy for popular patterns. The abstraction raising approach does not depend on tuning, and thus, does not introduce the risk of harming the performance for less experienced users in the general-purpose applications.

6.2 Future Work

The literature study performed at the early stages of the project motivated the project with the potential domain-specific analysis and optimizations, so the results could prove benefits for a broad range of domains and types of dynamic optimizations, i.e., including input-dependent domain-specific heuristics. However, as mentioned in Section 2.8 the MLIR's dialect did not mature in the area of specialized automated passes within the project's timeline. The Tensorflow and Stencil dialects are interesting cases to cover with PET-to-MLIR, which require more implementation effort on the side of the dialects and their documenting rather than our raising flow.

Section 2.7 described the current capabilities of the tool in handling the polyhedral-friendly code. Our version of the code generator does allow researchers to cover a decent range of loop programs or modify the loop structure with constructs supported with the prototype. PET-to-MLIR compiling flow, in order to become a fully-functional front-end, requires the implementation of all node types defined in the ISL AST generator, especially the if-nodes.

An expected improvement for the tool's practical use is the maturing of the linalg dialect itself. By introducing more automated optimizing passes like target-dependent tiling. An introduction of logic similar to the Build-To-Order BLAS compiler would be a very interesting extension of this project.

Finally, the analysis of polyhedral representation and their potential for pattern matching and polyhedral analysis was outside of the project's scope. However, given that the Loop Tactics-like tool could be implemented on top of the MLIR's Affine dialect it would be desirable to implement the entire raising flow within the MLIR framework.

Bibliography

[1] MLIR tutorial - Linalg Rationale. 03 2020. `https://mlir.llvm.org/docs/Rationale/RationaleLinalgDialect/`.

[2] Aravind Acharya and Uday Bondhugula. Pluto+: Near-complete modeling of affine transformations for parallelism and locality. *SIGPLAN Not.*, 50(8):54–64, January 2015.

[3] William Blume, Rudolf Eigenmann, Keith Faigin, John Grout, Jay Hoeflinger, David Padua, Paul Petersen, William Pottenger, Lawrence Rauchwerger, Peng Tu, and Stephen Weatherford. *Polaris: Improving the Effectiveness of Parallelizing Compilers*, pages 141–154. 04 2006.

[4] Uday Bondhugula. High Performance Code Generation in MLIR: An Early Case Study with GEMM, 2020. `https://arxiv.org/abs/2003.00532`.

[5] Uday Bondhugula, Muthu Baskaran, Sriram Krishnamoorthy, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model. In *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, CC'08/ETAPS'08, page 132–146, Berlin, Heidelberg, 2008. Springer-Verlag.

[6] 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.

[7] Alvin Cheung, Shoaib Kamil, and Armando Solar-Lezama. Bridging the gap between general-purpose and domain-specific compilers with synthesis. In *1st Summit on Advances in Programming Languages, SNAPL 2015, May 3-6, 2015, Asilomar, California, USA*, pages 51–62, 2015.

[8] Paul Feautrier. Some efficient solutions to the affine scheduling problem: I. one-dimensional time. *Int. J. Parallel Program.*, 21(5):313–348, October 1992.

[9] Jiří Filipovič, Matúš Madzin, Jan Fousek, and Ludundefinedk Matyska. Optimizing CUDA Code by Kernel Fusion: Application on BLAS. *J. Supercomput.*, 71(10):3934–3957, October 2015.

[10] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach. *SIGPLAN Not.*, 53(2):139–153, March 2018.

[11] Tobias Grosser, Sven Verdoolaege, and Albert Cohen. Polyhedral ast generation is more than scanning polyhedra. *ACM Trans. Program. Lang. Syst.*, 37(4), July 2015.

[12] Tobias Gysi, Christoph Müller, Oleksandr Zinenko, Stephan Herhut, Eddie Davis, Tobias Wicky, Oliver Fuhrer, Torsten Hoefler, and Tobias Grosser. Domain-Specific Multi-Level IR Rewriting for GPU The Open Earth Compiler for GPU-Accelerated Climate Simulation. 3(c). `https://arxiv.org/abs/2005.13014`.

[13] M.H. Jongen, L.J.W. Waeijen, R. Jordans, L. Jozwiak, and H. Corporaal. Optimization through recomputation in the polyhedral model. In *Eighth International Workshop on Polyhedral Compilation*

Techniques, January 2018. 8th International Workshop on Polyhedral Compilation Techniques (IMPACT 2018), January 23, 2018, Manchester, UK, IMPACT ; Conference date: 23-01-2018 Through 23-01-2018.

[14] Christoph W. Kessler. Pattern-driven automatic parallelization. *Sci. Program.*, 5(3):251–274, August 1996.

[15] K. Komisarczyk, L. Chelini, K. Vadivel, R. Jordans, and H. Corporaal. PET-to-MLIR: A polyhedral front-end for MLIR. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 551–556, 2020.

[16] Chris Lattner, Jacques Pienaar, Mehdi Amini, Uday Bondhugula, River Riddle, Albert Cohen, Tatiana Shpeisman, Andy Davis, Nicolas Vasilache, and Oleksandr Zinenko. Mlir: A compiler infrastructure for the end of moore's law. *arXiv preprint arXiv:2002.11054*, 2020.

[17] Vasilache N. Mehdi A. and A. Zinenko. MLIR Tutorial: Building a Compiler with MLIR. `http://llvm.org/devmtg/2019-04/slides/Tutorial-AminiVasilacheZinenko-MLIR.pdf`.

[18] Vijay Menon and Keshav Pingali. High-level semantic optimization of numerical codes. In *Proceedings of the 13th International Conference on Supercomputing*, ICS '99, page 434–443, New York, NY, USA, 1999. Association for Computing Machinery.

[19] Thomas Nelson, Geoffrey Belter, Jeremy G. Siek, Elizabeth Jessup, and Boyana Norris. Reliable generation of high-performance matrix algebra. *ACM Trans. Math. Softw.*, 41(3), June 2015.

[20] Cedric Nugteren, Pieter Custers, and Henk Corporaal. Algorithmic species: A classification of affine loop nests for parallel programming. *ACM Trans. Archit. Code Optim.*, 9(4), January 2013.

[21] Eunjung Park, John Cavazos, Louis-noël Pouchet, Cédric Bastoul, P Sadayappan, Eunjung Park, John Cavazos, Louis-noël Pouchet, Cédric Bastoul, and Albert Cohen. Predictive Modeling in a Polyhedral Optimization Space To cite this version : HAL Id : hal-00918653. 2013.

[22] Bill Pottenger and Rudolf Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th International Conference on Supercomputing*, ICS '95, page 444–448, New York, NY, USA, 1995. Association for Computing Machinery.

[23] L. Pouchet, C. Bastoul, A. Cohen, and N. Vasilache. Iterative optimization in the polyhedral model: Part i, one-dimensional time. In *International Symposium on Code Generation and Optimization (CGO'07)*, pages 144–156, 2007.

[24] LLVM project. Performance tips for frontend authors. `https://llvm.org/docs/Frontend/PerformanceTips.html`.

[25] Chandan Reddy. *Polyhedral Compilation for Domain Specific Languages*. Theses, Ecole normale supérieure, March 2019. `https://hal.archives-ouvertes.fr/tel-02385670/`.

[26] Amin Sarvestani, Erik Hansson, and Christoph Kessler. Extensible recognition of algorithmic patterns in dsp programs for automatic parallelization. *International Journal of Parallel Programming*, 41, 12 2013.

[27] Daniele G. Spampinato, Diego Fabregat-Traver, Paolo Bientinesi, and Markus Püschel. Program generation for small-scale linear algebra applications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, CGO 2018, page 327–339, New York, NY, USA, 2018. Association for Computing Machinery.

[28] Arvind K. Sujeeth, Kevin J. Brown, Hyoukjoong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. Delite: A compiler architecture for performance-oriented embedded domain-specific languages. *ACM Trans. Embed. Comput. Syst.*, 13(4s), April 2014.

[29] Arvind K. Sujeeth, Tiark Rompf, Kevin J. Brown, Hyoukjoong Lee, Hassan Chafi, Victoria Popic, Michael Wu, Ar Prokopec, Vojin Jovanovic, Martin Odersky, and Kunle Olukotun. Composition and reuse with compiled domain-specific languages. In *In Proceedings of ECOOP*, 2013.

[30] Field G. Van Zee and Robert A. van de Geijn. Blis: A framework for rapidly instantiating blas functionality. *ACM Trans. Math. Softw.*, 41(3), June 2015.

[31] Sven Verdoolaege and Tobias Grosser. Polyhedral extraction tool. In *In Second International Workshop on Polyhedral Compilation Techniques IMPACT'12*, 2012.

[32] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. Schedule Trees. In *IMPACT - 4th Workshop on Polyhedral Compilation Techniques, associated with HiPEAC*, Vienna, Austria, January 2014. ACM.

[33] I. Zhang. Application of machine learning for polyhedral optimizations Application of Machine Learning for Polyhedral Optimizations. 2019. `https://research.tue.nl/en/studentTheses/application-of-machine-learning-for-polyhedral-optimizations`.

[34] Oleksandr Zinenko. *Interactive Program Restructuring.* Theses, Université Paris Saclay (COmUE), November 2016. `https://tel.archives-ouvertes.fr/tel-01414770`.