Abstraction Raising in General-Purpose Compilers

Abstraction Raising in General-purpose Compilers

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de rector magnificus prof.dr.ir. F.P.T. Baaijens, voor een commissie aangewezen door het College voor Promoties in het openbaar te verdedigen op dinsdag 31 augustus 2021 om 13.30 uur

door

Lorenzo Chelini

geboren te Lucca, Italië

Dit proefschrift is goedgekeurd door de promotor en de samenstelling van de commissie is als volgt:

voorzitter:	prof.dr. K.A. Williams
1^e promotor:	prof.dr. H. Corporaal
2^e promotor:	assoc.prof.dr. T. Grosser (The University of Edinburgh)
copromotor:	dr. R. Jordans
leden:	prof.dr. M. van den Brand
	prof.dr. C. Kessler (Linköping University)
	prof.dr. Z. Su (ETHZ)
adviseurs:	dr. M. Kong (The University of Oklahoma)
	dr. A. Cohen (Google Research)

Het onderzoek dat in dit proefschrift wordt beschreven is uitgevoerd in overeenstemming met de TU/e Gedragscode Wetenschapsbeoefening.

PhD's thesis 2021

Abstraction Raising in General-Purpose Compilers

LORENZO CHELINI

Department of Electrical Engineering
Eindhoven University of Technology

Doctorate Commitee:

prof.dr. K.A. Williams	Eindhoven University of Technology, chairman
prof.dr. H. Corporaal	Eindhoven University of Technology, 1^e promotor
assoc.prof.dr. T. Grosser	The University of Edinburgh, 2^e promotor
dr. R. Jordans	Eindhoven University of Technology, copromotor
prof.dr. M. van den Brand	Eindhoven University of Technology
prof.dr. C. Kessler	Linköping University
prof.dr. Z. Su	ETHZ
dr. M. Kong	The University of Oklahoma
dr. A. Cohen	Google Research

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

A catalogue record is available from the Eindhoven University of Technology Library.
ISBN: 978-90-386-5361-7

LORENZO CHELINI
Department of Electrical Engineering
Eindhoven University of Technology

Summary

Abstraction Raising in General-purpose Compilers

Despite years of research, general-purpose compilers still struggle to obtain competitive performance on novel, more heterogeneous hardware. One of the main culprits is the low-level abstraction at which general-purpose compilers operate and the one-size-fits-all optimization heuristics they use. Consequently, developers rely on hand-written optimizations when a substantial performance level is required using low-level architecture-specific constructs. However, low-level optimizations and architecture-specific constructs obscure the high-level algorithm making the code less readable and portable. Traditionally, developers turned their attention to Domain-Specific Languages (DSL) (with accompanying compilers) or hand-tuned optimized libraries to compensate for the low-level abstraction of general-purpose compilers. In the last decade, multiple DSLs emerged and established themself successfully in various domains. For example, TVM in the machine learning domain and Stella for stencil computations.
However, despite being successful, DSLs are not immediately available to general-purpose code unless by a tedious and error-prone manual rewriting of the application. Worse, DSLs developments require effort and expertise in multiple domains and they are usually stand-alone solutions with little reusability of transformations and do not compose well enough to optimize multi-domain applications. Optimized libraries, on their part, allow developers to build high-performance applications from basic building blocks. However, their use is still restricted to expert programmers, locks-in the applications with a particular library vendor, and requires an effort to port legacy code.
This creates an abstraction gap problem; on one side, there is the need to raise the abstraction level to obtain competitive performance on modern, more CISC-like hardware; on the other side, we need to rescue millions of investment in general-purpose software and do not force developers to rewrite their application every time new hardware is available. The proposed research aims to develop a solid foundation for abstraction raising in general-purpose compilers, thus reducing the need for DSLs and allowing general-purpose flows to keep up with the hardware innovation pace, which pushes for more CISC-like, high-level accelerator operations.
We start our journey by developing Loop Tactics, a novel framework of composable program transformations based on an internal tree-like program representation of a polyhedral compiler. The framework relies on a declarative embedded C++ API built around easy-to-program matchers and builders, providing the foundation for developing loop optimization strategies. Using our matchers and builders, we express computational patterns and core building blocks, such as loop tiling, fusion, and data-layout transformations, and compose them into algorithm-specific optimizations. With Loop Tactics, we demonstrate that it is possible and beneficial to

directly bring specialized optimizations into general-purpose compilers.
However, despite serving its goal, Loop Tactics is still restricted to a single abstraction due to limitations of a general-purpose compiler's internal program representation, which does not enable to model multiple abstraction levels. The rise of multi-level rewriting lead by Google, with the MLIR infrastructure, enables us to overcome the single abstraction limitations and move to a more progressive and more layered abstractions-raising approach. This leads us to introduce Multi-level Tactics, which is, to the best of our knowledge, the first framework to bring abstraction raising in a multi-level intermediate representation framework.
We conclude our journey by demonstrating how abstraction raising can be used to effectively program novel accelerators. Considering computing-in-memory (CIM), based on memristor crossbars, as our use case, we demonstrate how we can transparently detect and offload computational motifs amendable for in-memory acceleration.

In summary, this thesis' contributions are: (1) a solid foundation to enable abstraction raising in general-purpose compilers using declarative matchers and builders. (2) A demonstration of how abstraction raising helps compilation for novel architectures by transparently detecting and offloading computational motifs amendable to be executed on the targeted architecture. (3) A set of tools to allow entering the MLIR lowering pipeline from low-level general-purpose languages.

Keywords: MLIR, LLVM.

Contents

List of Figures

List of Tables

Acronyms

ABI Application Binary Interface.
AI Artificial Intelligence.
API Application Program Interface.
AWS Amazon Web Services.

BLAS Basic Linear Algebra Subprograms.

CISC Complex Instruction Set Computer.

DLT Data Layout Transformation.
DSL Domain-Specific Language.
DSP Domain-Specific Processor.

GEMM General Matrix-Matrix Multiplication.

HPC High-Performance Computing.

IR Intermediate Representation.
ISA Instruction Set Architecture.

LLVM Low Level Virtual Machine.

ML Machine Learning.
MLIR Multi-Level Intermediate Representation.

ODS Operation Description Language.

SCF Structured Control Flow.
SCoP Static Control Part.
SSA Static Single Assignment.

TC Tensor Comprehensions.
TDL Tactics Description Language.
TDS Tactics Description Specification.
TPU Tensor Processing Unit.
TTGT Transpose Transpose GEMM Transpose.

1

Introduction

Improvements in compute performance have been critical components in advancing our society in the last 50 years. Since 2012, the amount of computing performance used in the most extensive AI training runs has been increasing exponentially, doubling every 3 to 4 months. In only six years, if we compare two well-known deep-learning networks such as Alex Net (2012) and AlphaGo Zero (2018), compute performance requirements increased by more than 300.000× [5]. A shift towards more specialized hardware (e.g., TPUs [6]) drove, and it's driving this trend. Today, all major silicon vendors provide specialized chips to sustain the exponential growing computational power required to fuel innovation; for example, in the new M1 chip from Apple (Figure 1.1), a large part of the die houses custom accelerators (Neural Engine, GPU, and domain-specific processors (DSPs)).

While hardware evolves, software innovation for heterogeneous hardware is held back by a 40-years old deal between software and hardware. The agreement states that the hardware may change significantly under the hood, but yesterday's software should run on tomorrow's machines. Thus, hardware may change, but it looks the same to software, thanks to a frozen and stable instruction set architecture (ISA). This consistent and never-changing language made it possible to effectively decouple hardware innovation without disrupting the software infrastructure and billions of investments. Unfortunately, this contract does not hold today anymore; heterogeneous hardware forces us to change the ISA by moving toward a more CISC-like, complex, and high-level instruction set. Compilers that connect software and hardware are struggling to keep up with a every changing landscape of ISAs. Conse-

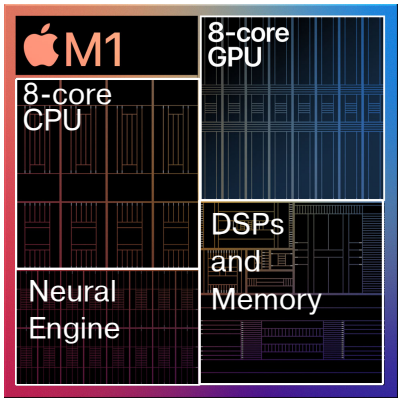


Figure 1.1: In the new Apple SoC M1, heterogeneity accounts for more 50% of the die approximately [1].

quently, general-purpose compilers are not able to obtain competitive performance on modern and heterogeneous hardware.

If performance is required, developers resort to low-level and architecture-specific programming models, which incurs a great loss of productivity. Often, developers write a vanilla application using concise high-level language. Later in a time-consuming process, translate the application to low-level high-performance code using architecture-specific details and programming models. To get the best performance, developers need to understand the hardware and write a different version of the code for each target device. The resulting code is tied to architectural details, is hard to read, debug, and maintain. It is difficult, if not impossible, to port this code to new architectures and specialized accelerators remain inaccessible. Worse, there are multiple versions of the original programs: one (version) written using a high-level vanilla language that is too high-level to obtain good performances and multiple low-level versions that are too hard to understand and maintain.

A typical response to address the hardware paradigm shift is introducing a new language or providing developers with manually tuned domain-specific libraries. Both solutions, as we will discuss in the next sections, are non-optimal. Ideally, we need a compiler mechanism that allows matching existing and new hardware with software automatically. We need to rethink the role of general-purpose compilers and move from a very fixed world to one where there are constant changes in the instruction set. Library APIs and Domain-Specific Languages (DSLs) should become the new ISA [7].

In the next sections, we start by describing the "optimization challenge". Specifically, we want to highlight the complexity of optimizing high-performance code for modern architecture and why general-purpose compilers fail to obtain competitive performance when compared with domain-specific compilers and tuned libraries. We do that by considering a ubiquitous kernel: general matrix-matrix multiplications (GEMM). Next, we introduce the problem statement and our contributions.

1.1 The Optimization Challenge

Optimizing programs is essential for all the application domains. Optimizing a program can assume different meanings; for example, we can optimize a program for energy efficiency, memory usage, or execution time. As far as we are concerned in this thesis, we focus on execution time. Specifically for us, optimization aims at reducing a program execution time. Optimizing for performance is critical in high-performance computing (HPC) and machine learning (ML) applications, as the difference between an unoptimized program and its optimized version is often multiple orders of magnitude difference. For example, in the ML domain, neural networks need to be trained before the inference phase. The training phase is the most time-consuming and requires performing basic linear-algebra operations on trillions of data. Given the dataset's size, even the smallest optimization can lead to a considerable reduction in training time, and comparable energy reductions.

```
int i, j, k = 0;
for (i = 0; i < 1024; i++)
  for (j = 0; j < 1024;l j++)
    for (k = 0; k < 1024; k++)
S:    C[i][j] += A[i][k] * B[k][j];
```

Listing 1: C++ code showing a naive matrix-matrix multiplication.

```
int i, j, k, tmp1, tmp2, tmp3, jc, ic, kc = 0;
for (j = 0; j <= 31; j++)
  for (k = 0; k <= 31; k++) {
    for (tmp1 = 32 * j; tmp1 <= 32 * j + 31; tmp1++)
      for (tmp2 = 32 * k; tmp2 <= 32 * k + 31; tmp2++)
        packing(A, tmp1, tmp2);
    for (i = 0; i <= 31; i += 1) {
      for (tmp1 = 32 * i; tmp1 <= 32 * i + 31; tmp1++)
        for (tmp3 = 32 * k; tmp3 <= 32 * k + 31; tmp3++)
          packing(B, tmp1, tmp3);
      for (jc = 0; jc <= 15; jc++)
        for (ic = 0; ic <= 15; ic++)
          for (kc = 0; kc < = 31; kc++) {
S:          /* unroll and compute GEMM */
          }
    }
  }
```

Listing 2: C++ code showing an optimized generic matrix-matrix multiplication implementation following the OpenBLAS/BLIS optimization strategy.

Example: Optimizing GEMM

To show the complexity of today's optimizations and how optimizations are tied to the target platform, we show how we can derive high-performance code for a simple general matrix-matrix multiplication (GEMM) using the OpenBLAS/BLIS optimization. GEMM is perhaps the most optimized kernel in history due to its widespread use in HPC and ML applications.

In its simplest form, a GEMM kernel consists of a triple nested loop with a multiply-and-accumulate operation in the innermost loop (Listing 1). The code is arguably easy to understand, given some basic knowledge of C or C++. However, the performance achieved by this simple implementation when executed on a modern CPU is far from impressive. A complete restructure of the code is necessary if we want to achieve higher performance on modern hardware.

Listing 2 shows a snippet of an optimized GEMM as a result for the OpenBLAS/BLIS optimization. High-performance is achieved by carefully mapping each matrix to a given cache level to reuse data in subsequent operations. The used data's size becomes smaller as the loop depth increases, nicely matching the memory hierarchy, where faster memories feature lower capacity than slower caches. The innermost loop is unrolled to improve vectorization. The optimization further introduces copy loops for matrix A and B to improve access to consecutive memory locations. The

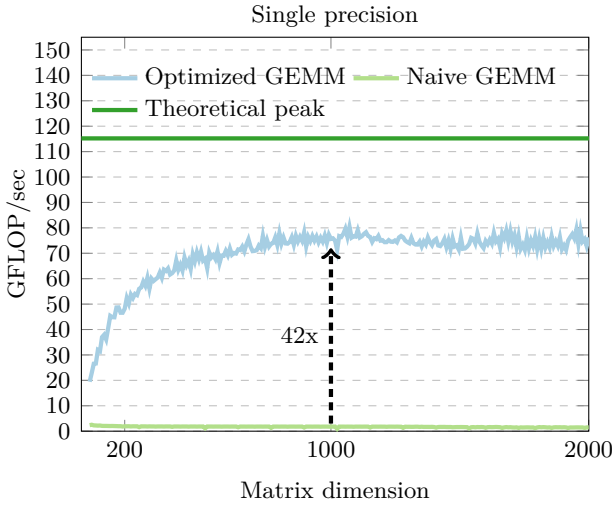


Figure 1.2: Performance obtained for single-precision operands. Comparison between a naive GEMM and an OpenBlis/BLAS implementation on an Intel Core i7-7700 (Kaby Lake family).

code in Listing 2 is by far more complex. It is not as easy to understand as the previous Listing, and the vanilla algorithm is lost. The optimization complexity also requires expertise in understanding hardware characteristics, which common developers may not have. Arguably, we believe that a compiler should generate the code in Listing 2. By providing a general-purpose compiler with the ability to identify computational motifs by reasoning at higher abstraction levels, we can match many and varying high-level APIs and DSLs (with accompanying transformations) to existing code automatically.

Figure 1.2 shows the performance difference between the two Listings. The dark green line is the machine's theoretical peak—an Intel Core i7-7700 (Kaby Lake family) clocked at 3.6 GHz. The light green line at the bottom of the picture shows a general-purpose compiler's achieved performance on the vanilla matrix-matrix multiplication (Clang 6.0). The light blue shows the optimized version's performance—a performance improvement of more than 42 ×.

Today, such optimization is not accessible to general-purpose flow, as they reason at a low-level of abstractions and perform a one-size-fits-all optimization strategy. Consequently, today domain-specific compilation and high-performance libraries are the de-facto solutions to obtain high-performance code. Let us briefly discuss them.

Domain-specific Compilation

As the hardware increases in complexity, domain-specific compilation proved to be a viable solution to generate high-performance code on modern hardware. Domain-specific compilers, having higher-level semantics and fewer restrictions than general-purpose compilers, can perform higher-level optimizations and translations. For example, a domain-specific compiler for linear algebra can reason about the program at the level of matrix and vector arithmetic, as opposed to loops, instructions, and scalars.

Domain scientists write their programs using a high-level language that closely matches their domain of interest (e.g., tensor operations in the linear algebra domain). A domain-specific compiler then optimizes the higher level, more semantically rich language by relying on more context information. For example, in the linear-algebra domain, a domain-specific compiler can easily detect that a double matrix transposition is no transposition, thus removing the unnecessary computation. Today, domain-specific compilation is the de-facto solution to obtain high-performance code without incurring in productivity, portability, and performance issues.
While the use of DSLs and accompanying compilers provides a solution to obtain high-performance code, it is costly in development time. The development of a DSL requires expertise in multiple domains, from language design to code generators. Worse, DSLs are usually stand-alone solutions with little reusability, and they do not compose well enough for optimizing multi-domain applications. This lack of reusability makes developing DSLs a cumbersome task, as a new development process needs to start from scratch for every new compiler, albeit tools to speed up the development like Delite [8] exist. This approach is not sustainable in an ever-growing landscape of hardware devices and application domains. Finally, general-purpose code is not immediately available to DSLs unless of a tedious manual rewriting.

Developing High-performance Libraries

Developing a high-performance library is a very challenging and time-consuming process. For every building block of the library (e.g., matrix-matrix multiplication), library developers need to provide high-performance implementations for different architectures and input sizes, varying in precision and storage format. Every new chip generation, regardless of CPUs or GPUs, comes with its characteristics, and these characteristics play a significant role in how the code needs to be optimized. Similarly, depending on the problem size and precision required in the computation, library developers need to choose among different algorithmic implementations. For example, the new Tensor Core available in the NVIDIA's Volta and newer GPUs architectures requires specific input dimension and work on 16-bit operands. Storage formats, row-major or column-major, also need to be considered as they significantly impact memory performance due to different access patterns. Even worse, libraries lock-in an application to a particular vendor, and in some cases, to a specific library version. Locking-in an application in a fast-paced hardware innovation world is not ideal and reduces application readiness once a new architecture family is launched. Providing and maintaining high-performance libraries is only feasible for very few basic-building blocks and requires huge investment and effort from multiple performance engineers.

1.2 Problem Statement

With general-purpose compilers struggling to obtain competitive performance on diverse and heterogeneous hardware, domain-specific compilers and libraries are today's solutions to high-performance code. While both approaches are viable, they

are not optimal. On the one hand, libraries lock-in the application to a specific vendor, and the composition of highly efficient building blocks may not result in an overall efficient application. On the other hand, developing domain-specific compilers requires effort and expertise in multiple domains, and domain-specific compilers do not allow to optimize multi-domain applications. Also, while domain-specific compilation promises to better exploit application knowledge directly by connecting the software with the hardware, it is a solution that may not be attractive in the long-term. For every new language, developers need to rewrite their code in a time-consuming manual process, with no guarantees that the language will support new hardware in the future.
This creates friction; on one side, software needs to raise abstraction level without leaving behind general-purpose software or forcing developers to rewrite their applications. On the other side, the adoption of new hardware is delayed because there is no easy way to use and program it. We *need a mechanism that allows matching new hardware to existing and future software without developing a new compiler or language every time*. If we achieve this, we no longer care about the hardware ISA. Consequently, rather than asking programmers to rewrite their code in a new DSL or using a new API, it is now up to the compiler's job to perform this task. We effectively move the compiler's job from code generation to program generation. Library APIs and DSLs become the new ISA.

1.3 Thesis Contributions

In this thesis, we formulate ideas and mechanisms that allow us to reconnect software and hardware using modern compiler technology based on the multi-level intermediate representation rewriting and the polyhedral framework. We reduce the need to develop new languages and make it easier to exploit new hardware efficiently; we allow existing software to use future hardware and allow hardware innovations to connect to new and emerging applications.
We make the following contributions:

- *Declarative Loop Tactics*, a novel framework of composable program transformations based on an internal program representation of a polyhedral compiler, enabling developers to add highly-customized optimizations for a given computational pattern. With Declarative Loop Tactics, we bring domain-specific optimizations in general-purpose compilers (Chapter 3).
- Mature compiler technology based on multi-level intermediate representation rewriting allows us to reconnect software written at low-level abstraction with high-level one, thus enabling domain-specific compilation on semantically poor languages. In this context, we propose *progressive raising* a complementary approach to the progressive lowering in a multi-level IR compiler. Progressive raising enables lifting the level of abstraction of general-purpose code and leveraging subsequent high-level domain-specific transformations (Chapter 4).
- With the proliferation of domain-specific hardware, compiler support is lagging behind. Current approaches to program novel devices shift the burden on the programmer, which needs to use the provided API to exploit the hardware efficiently, reducing application readiness. In this context, we propose

a compiler-based mechanism that enables transparent code offloading for in-memory architectures (Chapter 5).
- A set of *basic building blocks to facilitate future research* in a multi-level intermediate representation compiler (Chapter 6).

Overall, in this thesis, we provide the fundamental building blocks that empower compiler developers with mechanisms that enable matching new hardware to existing and future software. Therefore, we do not ask application programmers to rewrite their application, but we shift this task to the compiler, increasing productivity and application readiness.

1.4 Thesis Overview

We structure this thesis as follows:

Chapter 2 introduces the necessary background required for understanding the contributions made. To provide a self-contained thesis, we start from the basic by briefly introducing how a compiler works and how its internal machinery interact. We then present general-purpose and domain-specific compilation, highlighting the main advantages and disadvantages. We quickly explain the SSA form, a well-known compiler intermediate representation, and a more mathematical-based representation: the polyhedral model. This model lays the foundation for our first contribution, Declarative Loop Tactics.

Chapter 3 demonstrates the necessity and the advantages of Domain-specific compilation in general-purpose flow. We introduced declarative Loop Tactics (or Loop Tactics in short), a novel framework of composable program transformations based on an internal-tree-like representation. The framework is based on a declarative C++ API built around easy-to-program matchers and builders, providing the foundation for developing loop optimization strategies. By applying Loop Tactics to two domains; linear-algebra and stencils, we show how developers can add highly customized optimizations for a given computational pattern directly in a general-purpose compiler. Thus, we reduce the need for DSLs and extend the range of optimizations that a current general-purpose compiler can perform.

Chapter 4 introduces Multi-level Tactics, a novel approach to abstraction raising in a multi-level intermediate representation (IR) compiler. Specifically, we aim at laying the foundation for a complementary approach to progressive lowering, which we call progressive raising. Progressive raising enables to lift the level of abstraction from low-level IRs to higher-level ones and enable effective reuse of aggressive domain-specific optimizations available at higher abstraction levels.

Chapter 5 shows how we can use our abstraction raising mechanism to program novel heterogeneous targets. By focusing on in-memory computing, we demonstrate how abstraction raising enables modernizing legacy code to automatically and transparently invoke in-memory computing acceleration. We further present a high-level

flow that uses multi-level IR rewriting to reliably map tensor operations on the target in-memory device.

Chapter 6 presents a set of basic-building blocks we developed to facilitate future research in multi-level intermediate representation rewriting.

Chapter 7 concludes our journey.

In the following Chapters, text and figures are taken and readapted from our previously published research papers.

2
Background

This chapter introduces the technical background required to understand the contributions of this thesis. We briefly touch upon what a compiler is and how it works (to be self-contained). The chapter then introduces general-purpose and domain-specific compilation flows, focusing on the optimizer and its intermediate representation, the SSA form, and a more mathematical oriented one, the polyhedral model. It further discusses domain-specific compilation and the advantages of this compilation on a general-purpose one. It does that by introducing *Tensor Comprehensions*, a domain-specific compiler for machine learning. Afterward, it introduces the new compilation concept of multi-level intermediate representation rewriting and how multi-level IR promises to lower the cost of designing domain-specific compilers by choosing the level of representation that is right for your problem or target device.

2.1 Compiler Internals

The world as we know depends on software. The compiler's task is to translate software written in a given programming language to a form that a computer can execute. Thus a compiler is a program that can read a program in one language, the source language, and translate it into an equivalent program in a destination language. If the destination language is a machine language, then the user can execute the source program on the targeted machine.
Besides a compiler, several other tools are required to create an executable program. For example, different modules stored in separate files can constitute a single source program. The task of collecting the sources into a single compilation unit is the task of the preprocessor. The preprocessor may also expand shorthand, known as macros, into the source language statements. The compiler takes a single compilation unit and produces an assembly program as output, which the assembler uses to create a relocatable machine code. Multiple compilation pieces likely make a single extensive

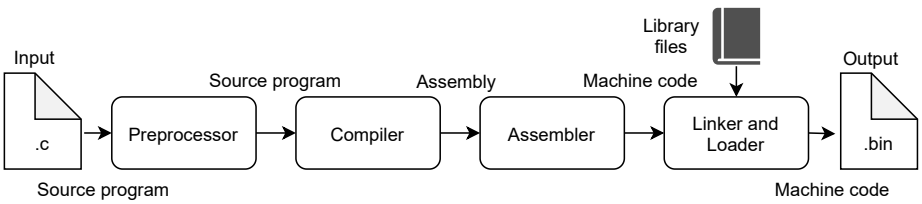


Figure 2.1: Building blocks to create an executable starting from a source file. Inspired from [2].

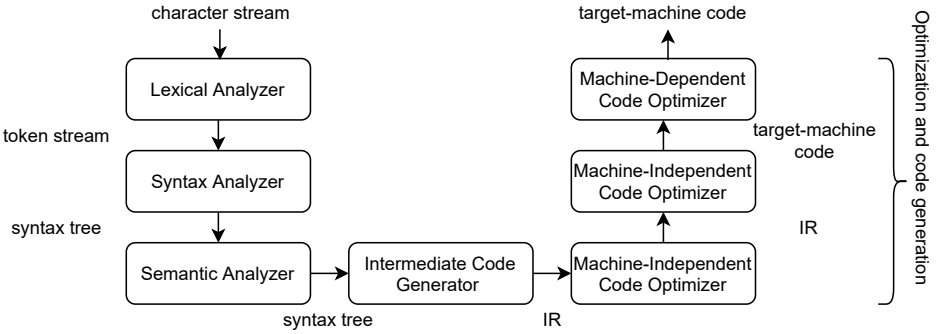


Figure 2.2: Phase of a general-purpose compiler. Inspired from [2].

program; it is the linker's task to take as input the different relocatable machine codes and link them together. The loader then puts together all of the executable object files into memory for execution (Figure 2.1).

2.1.1 The Structure of a Compiler

Mapping a source language to a destination language requires two phases: analysis and synthesis. The analysis breaks the source program into tokens and imposes a specific syntax and semantic on them. If the analysis part detects an ill-formed program, it provides error messages to the user, who will need to take corrective actions. The result of the analysis phase is an intermediate representation (IR) of the source language. The synthesis part then lowers the intermediate representation to a machine-executable program. The analysis part is often called the compiler's frontend, while the synthesis part is the backend. Most of the compilers, if not all, have an intermediate optimization phase, which takes the IR generated by the frontend optimizes for the targeted machine, and passes it to the backend. Figure 2.2 shows the different phases that constitute the frontend, the optimizer, and the backend.

Lexical Analysis

Lexical analysis is the first phase of a compiler. It takes the output of the preprocessor, a modified stream of characters making up the source program. The lexical analyzer breaks the character stream into tokens by ignoring white spaces and comments and groups each character into a meaningful sequence following the language's rules.

Syntax Analysis

Parsing or syntax analysis immediately follows the lexical analysis phase. During syntax analysis, the parse creates a syntax tree representation (e.g., Clang AST) of the source program from the token obtained by the previous analysis. In a typical representation of a syntax tree, each node represents an operation, and the children's node represent the operation's arguments.

```
x = input();
if (x == 23)
  y = 1;
else
  y = x + 2;

print(y);
```

⇒

```
x = input();
if (x == 23)
  y1 = 1;
else
  y2 = x + 2;

y3 = φ(y1, y2);
print(y3);
```

Listing 3: SSA-based representation (right) for the simple code snippet on the left. The y-variable has been renamed to eliminate multiple assignments to the same variable.

Semantic Analysis

The semantic analyzer uses the syntax tree to check the source program for semantic consistency concerning the language definitions. Type checking happens in this phase. For example, some languages allow only index types to index array; if an integer of floating-point type is used, an error is reported when accessing the array.

Optimizer

After syntax and semantic analysis, a compiler generates a low-level, machine-like representation. The code optimization phase attempts to improve the intermediate representation quality to result in a better target code. "Improve" can assume many objectives. For example, we may want to improve the code quality to have a faster version of the original or improve the code quality to reduce the memory footprint. Depending on the programmer's objective, the optimization passes are carried out by the optimizer. There is also a great variation in the number of code optimizations a compiler can perform. Optimization works on a low-level machine-like representation. The Static Single Assignment (SSA) form is a widely used representation in compiler IRs [9]. SSA is an intermediate representation constructed for a set of program variables referred to as SSA variables. In the SSA form, each assigned value to an SSA variable creates a unique named temporary that holds the given variable's value. A named temporary can be thought of as a register. The SSA form provides numerous advantages, including making use-def chains explicit in the IR, which in turn helps to simplify some optimizations. It is widely understood by the compiler community and used in major frameworks. Listing 3 shows a simple SSA-based representation (right) for the simple C code on the left. The variable y on the left has been renamed on the right (y1 and y2) to eliminate multiple assignment statements to the same variables. The ϕ function, a special statement known as pseudo-assignment function, serves to merge the values from different incoming paths (if and else).

The polyhedral model is another form of intermediate representation based on mathematical entities, enabling more aggressive optimizations; see Section 2.3.

```
Pass Arguments: -targetlibinfo -tti -tbaa -scoped-noalias -assumption-cache-tracker -profile-summary-info -forceattrs -inferattrs
-domtree -callsite-splitting -ipsccp -called-value-propagation -attributor -globalopt -domtree -mem2reg -deadargelim -domtree -basicaa
-aa -loops -lazy-branchprob -lazy-block-freq -opt-remark-emitter -instcombine -simplifycfg -basiccg -globals-aa -prune
-inline -functionattrs -argpromotion  -domtree -sroa -basicaa -aa -memoryssa -early-csememssa -speculative-execution
-basicaa -aa -lazy-value-info -jump-threading -correlatedpropagation -simplifycfg -domtree -aggressive-instcombine -basicaa -aa
-loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -instcombine -libcalls-shrinkwrap -loops -branch-prob -block-freq
-lazy-branch-prob -lazy-block-freq -opt-remark-emitter -pgo-memop-opt -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq
-opt-remark-emitter -tailcallelim -simplifycfg -reassociate -domtree -loops -loop-simplify -lcssa-verification -lcssa -basicaa -aa
-scalarevolution -loop-rotate -licm -loop-unswitch -simplifycfg -domtree -basicaa -aa -loops -lazybranch-prob -lazy-block-freq
-opt-remark-emitter -instcombine -loop-simplify -lcssaverification -lcssa -scalar-evolution -indvars -loop-idiom -loop-deletion
-loop-unroll -mldstmotion -phi-values -basicaa -aa -memdep -lazy-branch-prob -lazy-block-freq -opt-remark-emitter -gvn -phi-values
-basicaa -aa -memdep -memcpyopt -sccp -demanded-bits -bdce -basicaa -aa -loops -lazy-branch-prob -lazy-block-freq -opt-remark-emitter
-instcombine -lazy-value-info -jumpthreading -correlated-propagation -basicaa -aa -phi-values -memdep -verify
```

Listing 4: Some of the passes available in the Clang compiler when targeting high-performance code, deciding which pass to apply and which order is an open research question. Compiler writers decide a predefined order considering the general case.

Code Generation

The code generation step takes as input the intermediate representation of the source program and maps it to the target machine code. This phase is where registers or memory locations are selected for each of the program's variables.

2.1.2 General-purpose Compilers

General-purpose compilers like Clang encode optimizations in passes. Each pass performs a specific optimization or performs an analysis creating meta-data information used by subsequent passes. Passes are applied in a specific order predefined by compiler developers. The end user can affect the pass ordering by using compilation arguments or generation options.
Listing 4 shows the available passes in Clang when targeting high-performance. The `-O3` option enables the generation of high-performance code. `-O3` option is like the `-O2` except it allows for optimizations that take longer to perform or generate larger code to make the program run faster. The order of appearance reflects the order in which each pass is applied. Clang may apply for the same pass multiple times (e.g., `basiccg`).
Deciding which passes to apply and in which order is an open question in general-purpose compilers. Today, predefined optimization sequences like `-O3` use a fixed amount of passes in a predefined order decided by compiler writers, considering the general case. This fixed optimization scheme allows for one-size-fits-all optimizations but fails to optimize for a given application domain. A current way to adapt the optimizations of a general-purpose compiler for a given application is to try out multiple combinations of pass arguments and pass-input values. This approach is not sustainable as it needs to explore an ample optimization space.
As a consequence of the one-size-fits-all optimization scheme, general-purpose compilers often do not deliver the required performance when targeting novel hardware and more specialized domain applications. Left with a low-performing code, developers resort to manual application rewriting, intertwining target-specific optimizations with the vanilla algorithm. The resulting code is hard to maintain, debug and port to new hardware. Domain-specific compilation emerged as a possible solution that can provide productivity, performance, and portability for high-level programs in a specific domain. Domain-specific compilation flourished in the last decade, and some of them established as the de-facto solution to generate high-performance code

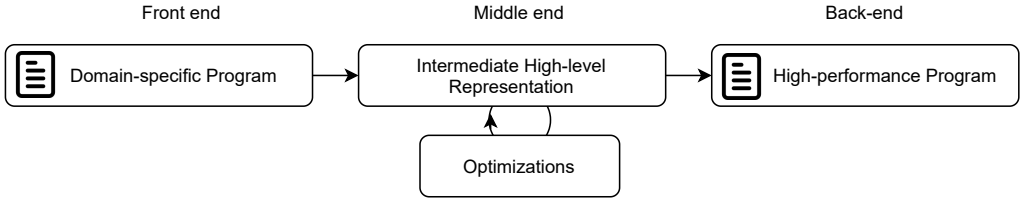


Figure 2.3: Overview of a domain-specific compilation: a domain scientist writes a program using an high-level language close to its domain of interest (a domain-specific language). The program is compiled by a domain-specific tool chain that generates high-performance code.

in different disciplines. *Tensor Comprehensions* [3] in the linear-algebra domain or Halide [10] in the image processing domain are two examples.

2.1.3 Domain-specific Compilers

By raising the abstraction level and customizing the optimizations to a particular domain, domain-specific compilers gain an advantage over traditional general-purpose ones by reasoning about data structures and operations at the level of domain abstractions. For example, a domain-specific compiler in the linear-algebra domain can reason about operations between tensors instead of loops and scalar variables. Also, a DSL compiler can enforce domain-specific restrictions and apply optimizations that are correct by construction. On the other hand, a general-purpose compiler must be restrictive and cautious on the optimizations to use. For example, in a general-purpose language that uses pointers such as C or C++, knowledge of pointer behavior is required to understand if a storage location is accessed in multiple ways. A pointer alias analysis attempts to determine, at compile-time, what a pointer is pointing to. Unfortunately, in general, such analysis is not undecidable [11], and approximation methods are used. A domain-specific language can enforce non-aliasing for all the storage locations, thus allowing the compiler to be more aggressive and less cautious in applying optimizations.
Domain-specific compilers come with an accompanying domain-specific language (DSL). Figure 2.3 shows the basic building blocks that allow compiling a DSL source program. The compiler front-end lowers a DSL to an internal intermediate representation. Already at this stage, developers need to address multiple challenges. For example, what kind of syntax and semantic domain-specific language should have to adequately capture the application domain? Once the syntax and semantic of the language have been defined, developers need to write the middle-level optimizer. The middle-level optimizer takes as input an intermediate representation (IR) and applies a set of optimization on it. Here, developers need to define an IR representing domain-specific computations and encode optimizations—a non-trivial task that requires multiple interactions between the language and the IR itself. Finally, the compiler backend uses the optimized IR to generate a high-performance program. At the backend stage, the main challenge is code generation. Definitions of the language and the IR mainly depend on the application domain; whenever developers need to support a new domain, IR and language need to be redesigned

```
def mv(float(M,K) A, float(K) x) -> (c) {
  c(i) +=! A(i,k) * x(k)
}
```

Listing 5: Matrix-vector multiplication in *Tensor Comprehensions* notation.

from ground-up.
Despite the performance benefit of adopting a DSL, the cost of developing a DSL and its accompanying compiler is immense and it requires expertise in multiple domains from language design to computer architecture. Worse, there is little or no reuse between DSLs, almost by definition: a DSL is specific to a given domain and cannot express other domains. Domain-specific compilers have similar restrictions; they can optimize a given domain but are not optimal for other domains. This lack of reusability makes developing novel DSLs difficult, hinders their widespread adoption, and makes the approach not sustainable, given a growing number of applications and hardware landscape.
Multiple works such as Delite [8] aim at simplifying the development of DSL by providing common components, optimizations, and code generators that multiple DSL implementations can reuse. Unfortunately, they do not fully solve the problem, as composability of multiple domains remains a problem.

A Linear-algebra Domain-specific Language Using domain-specific languages (DSLs), domain scientists express computations using familiar abstractions. For example, *Tensor Comprehensions* is a productive-oriented language in the machine learning domain that allows the definition of neural network layers as simple mathematical expressions between tensors. Listing 5 demonstrates how the language enables the definition of a simple matrix-vector multiplication between an $M \times K$ matrix `A` and a K-vector `x`, resulting in a vector `c`, all composed of single-precision floating-point values.
The syntax of *Tensor Comprehensions* borrows from Einstein Notation, where universal quantifiers (becoming loops in a program) are introduced implicitly. The shape and the type of inputs are provided explicitly in the function signature, while those of the output tensor are inferred from the index variables. The index variable `i` iterates over the first dimension of `A`, its range is therefore $[0, M-1]$. Since `i` also indexes the output vector `c`, the size of `c` is M. Similarly, *Tensor Comprehensions* deduces the range for `k` from the tensors on the right-hand side of the expression.
The algorithm proceeds iteratively if more than one iterator is involved in a subscript, using the ranges computed on the previous step together with the known sizes of the tensors on the right-hand side to infer the shape of the tensor on the left-hand side. The iterator k only appears on the right-hand side, which indicates that the entire expression is a reduction over k. The "`!`" mark in the operator denotes a default-initialized reduction. For sum-reductions, as in the example, the left-hand side is initialized with zeros of an appropriate type. The type of the output matrix is inferred from the result type of the multiplication and the reduction. As the sum of multiplications of single-precision floating-point values has the same type, c is a vector of single-precision floating-point elements.

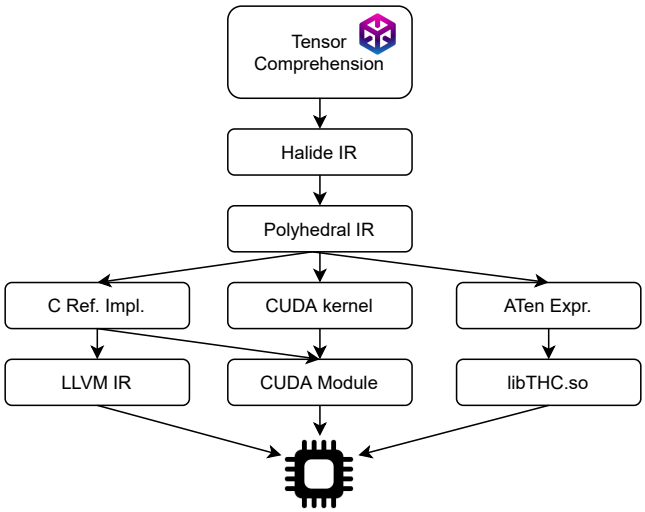


Figure 2.4: Overview of the *Tensor Comprehensions* domain-specific compiler. Inspired from [3].

A Linear-algebra Domain-specific Compiler To have a concrete example of the basic building block that constitutes a DSL and its accompanying compiler, let us consider the compilation flow for the *Tensor Comprehensions* language. The *Tensor Comprehensions* framework is a productivity-oriented system for expressing machine learning workloads embedded into PyTorch, providing a concise input notation (the *Tensor Comprehensions* language) with range inference capabilities and leveraging the polyhedral model for optimal code generation. The framework is capable of targeting both GPUs and CPUs. In contrast to conventional ML systems, *Tensor Comprehensions* is not limited to a predefined set of operators or layers (e.g., convolution or matrix-matrix multiplication) but allows the user to specify custom computations using index expressions on tensors. Types and shapes of intermediate tensors only need to be provided explicitly where automatic inference fails due to ambiguity. *Tensor Comprehensions* generates efficient GPU code that fits into a single GPU kernel or an ML framework *operator*.

The compilation flow automation abstracts the implementation details away from programmers, allowing them to reason in mathematical terms. It also enables the compiler to perform advanced program transformations using the precise analysis of the polyhedral model. For example, a common high-level optimization is operator fusion, eliminating the requirement for temporary storage and spurious serialization by combining multiple ML operations.

Figure 2.4 shows the internal machinery of the *Tensor Comprehensions* framework. At the highest level of abstraction *Tensor Comprehensions* uses the same graph-based IR as the domain-specific Halide compiler, initially designed for image processing pipelines. Despite its original design goal, the Halide IR proved to express machine learning and linear-algebra computation successfully. Halide IR lowers to a loop-based program representation, the polyhedral model. The polyhedral model is a mathematical-based representation that enables to model every single iteration in a loop program. Thanks to its mathematical nature, it allows strong reasoning on dependencies analysis and enables effective program optimizations. We will give

a more in-depth description of the model in Section 2.3. From this mathematical abstraction, there are multiple lowering paths to generate machine code depending on the targeted hardware. For the CPU code generation path, *Tensor Comprehensions* lowers the mathematical abstraction to LLVM IR and then piggy-back on the LLVM framework to generate machine-executable code. For the GPU code generation path, *Tensor Comprehensions* lowers the mathematical model to CUDA code, which can get just-in-time compiled and execute. An autotuned and serializable compilation engine complements the *Tensor Comprehensions* flow by interacting with scheduling and mapping strategies to search the optimization space.

2.2 Multi-level IR

Decades of research and development make compiler design a mature field with a wide range of well-known algorithms in code generation, static analysis, and program transformations. Multiple technologies emerged as the de-facto solutions to compile general-purpose languages; the LLVM compiler infrastructure is one of them. A common characteristic of all these technologies is the single level of abstraction at which they operate. For example, the level of abstraction of the IR of the LLVM compiler infrastructure is comparable to C. This "single-abstraction" has been invaluable till now and proved to be successful in compiling general-purpose languages like C, C++, or Java. Simultaneously, a higher-level abstraction may be a better fit to model many problems in some cases. The lack of a higher abstraction pushed many languages to adopt their representation to solve domain-specific problems. For example, machine learning systems like TensorFlow [12] have many IRs like XLA HLO, TensorRT [13], nGraph [14], and Core ML [15]. While the development of a domain-specific IR is a well-known art, the engineering and implementation cost is still too high. Worse, there is no code reused between the different domains, and an advance in one community cannot benefit the others. In most cases, this means a lower quality compiler: slow compile-time, buggy implementations, and poor-user experience [16].
Multi-level IR aims to address the "single-abstraction" limitations of today's general-purpose compilers directly. MLIR being the very first Multi-level IR framework. By providing a toolkit to define and introduce new abstractions, MLIR makes it cheap and relatively fast to introduce new IRs, enables reuse across communities, and provides a solid foundation for a high-quality compiler. In the next section, we introduce the main pillars and present an overview of the MLIR framework.

2.2.1 The MLIR Infrastructure

The MLIR compiler infrastructure is a project under the LLVM umbrella that is well-suited for multi-level IR rewriting [16]. MLIR provides an intermediate representation (IR) with only few concepts being built-in, leaving most of the IR customizable. A customizable IR allows compiler developers to match the right abstraction level for their problem at hand by introducing custom types, operations, and attributes. Operations are the essential atomic constituents of the IR; the semantic of the operation describes what is computed and how. In MLIR there is no a fixed set of

```
%result = "dialect.operation"(%operand, %operand)
       {attribute = dialect<"value">} ({
                      // op region which contains a single basic
                      // block with a nested operation.
                      ^basic_block(%block_argument: !dialect.type):
                        "another.operation"() : () -> ()
                }) : (!dialect.type) -> !dialect.result_type
```

Listing 6: Operations, regions, and blocks form a nested structure. Operations are the main entities in MLIR. Operations contain a list of regions. Regions contain a list of blocks that have other operations, enabling a recursive structure. The Listing shows a single region containing one basic block which in turn contains an operation "another.operation".

operations. An operation has an attached region that may contain other operations. Within a region, a block contains a list of operations with no control flow. The last operation in a basic block is a terminator that can transfer the control flow to (basic) blocks or regions. Listing 6 shows the nested structure of operations, regions and blocks.

Each operation uses and produces new values. A value represents data at runtime, and it is associated with a type known at compile-time, whereas types model compile-time information about values. Complementary to this, attributes attach compile-time information to operations. Attributes are typed (e.g., string) and attached to operations. In the generic MLIR syntax attributes are enclosed in curly braces (Listing 6). Custom types, operations, and attributes are logically grouped into dialects. A dialect is a basic ingredient that enables the MLIR infrastructure to implement a stack of reusable abstractions. Each abstraction encodes and preserves transformation validity preconditions directly in its IR, reducing the complexity and the cost of analysis passes.

Figures 2.5 shows an high-level overview of some of the dialects available in MLIR and their entry points in the compiler pipeline. Each dialect models a specific domain, for example, the Linalg dialect captures linear-algebra operations on either tensor or buffer operands. At a similar abstraction level, the Stencil dialect represents iterative kernels that update array elements according to a given stencil pattern [17]. At a lower abstraction level, the Affine dialect models a simplified polyhedral representation, while F18 models FORTRAN specific constructs (e.g., dispatch table). SCF and Standard represent structured control flow and a collection of miscellaneous operations, respectively. Lastly, MLIR LLVM dialect models LLVM-IR constructs.

IR customization is enabled through a declarative system, mostly based on TableGen. TableGen is a data modelling tool for defining and maintaining records of domain-specific information, and is extensively used across the LLVM codebase [18]. For example, MLIR declaratively describes operations using an Operation Description Specification language (ODS) built on top of TableGen. Declarative specification for new operations speed-up the development of new custom IRs and reduces the amount of code duplication as well as the probability of errors. TableGen files are only "containers" of domain-specific information. They do not have any meaning

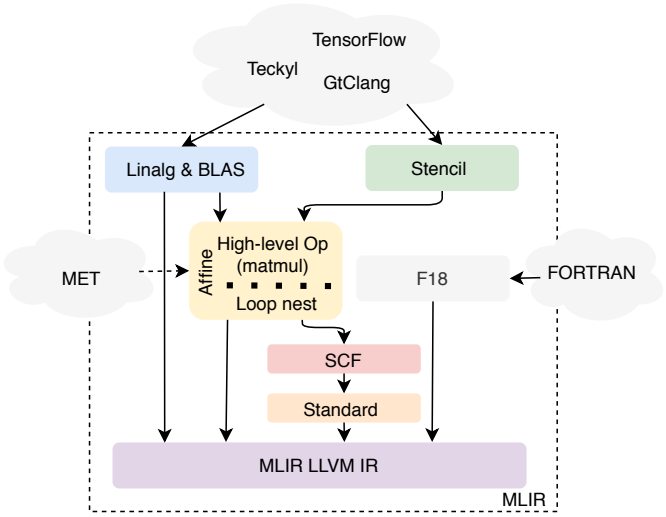


Figure 2.5: Some of the available dialects in MLIR. The higher a dialect is the higher is its abstraction level. Different entry points in the MLIR toolchain are also shown.

without a backend. It is up to the backend at compile time to interpret the stored information and generate the C++ declarations and definitions.
Progressive lowering (downward arrows in Figure 2.5) enables lowering operations from high-level abstractions dialects to low-level IRs. Listing 7 shows an example of progressive lowering of a linalg.matmul operation to a triple affine loop containing a multiplication and an addition in the Affine dialect. Subsequent lowering (not shown) fuses the operations of the body to a multiply-and-accumulate operation (MAC). Efficient progressive lowering is enabled through a pattern rewriting infrastructure that allows transformations as composition of small local rewrites [16].

2.3 Polyhedral Program Representation

The *polyhedral model* is a unified framework to model and transform loops in imperative programs [19]. After more than three decades of active research and attempts to integrate this model into production compilers [20, 21, 22], the polyhedral model has recently attracted renewed attention as a method for generating efficient code from domain-specific languages [23, 24] that targets both CPUs and accelerators. The use of solid mathematical reasoning enables precise static analysis and exact transformations. Dynamic extensions also allow using the model on programs that lack static information [25, 26, 27].

2.3.1 Mathematical Foundations

The polyhedral model is an algebraic representation of a subset of the imperative program. It enables encoding and reasoning on the individual statement instances represented as a set of points in multi-dimensional space, constrained by affine hyperplanes forming a polyhedron. Integer polyhedra and Presburger relations are the

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

                ⇓ lowers to
affine.for %i = 0 to %N          // affine dialect
  affine.for %j = 0 to %N
    affine.for %k = 0 to %N {
      %0 = affine.load %C[%i, %j] : memref<?x?xf32>
      %1 = affine.load %A[%i, %k] : memref<?x?xf32>
      %2 = affine.load %B[%k, %j] : memref<?x?xf32>
      %3 = std.mulf %1, %2
      %4 = std.addf %3, %0
      affine.store %4, %C[%i, %j] : memref<?x?xf32>
    }
                ⇓ lowers to
%S = constant 1 : index
for %i = 0 to %N step %S          // scf dialect
  for %j = 0 to %N step %S
    for %k = 0 to %N step %S {
      %0 = load %C[%i, %j] : memref<?x?xf32>
      %1 = load %A[%i, %k] : memref<?x?xf32>
      %2 = load %B[%k, %j] : memref<?x?xf32>
      %3 = std.mulf %1, %2
      %4 = std.addf %3, %0
      store %4, %C[%i, %j] : memref<?x?xf32>
    }
```

Listing 7: Progressive lowering in MLIR. A linalg matmul translates into a nested loop with loads from 2-d memrefs, one addition, and one multiplication in the Affine dialect. The Affine dialect is then lowered to SCF.

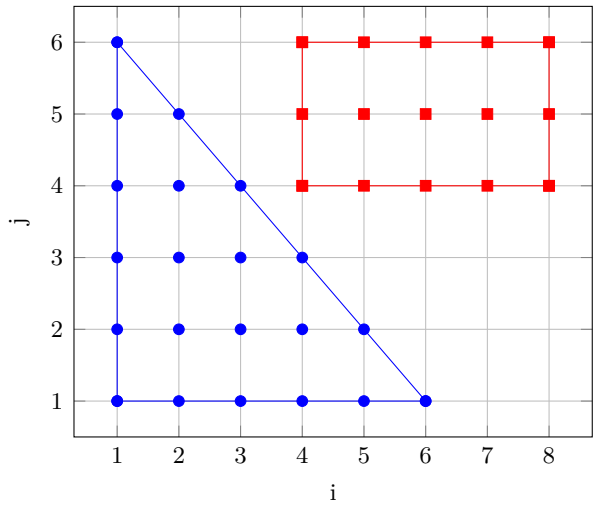


Figure 2.6: Two dimensional dense integer sets. Example taken from [4].

mathematical foundation of the model. In this chapter, we introduce basic mathematical formalisms on which the model is built. We start by introducing integer sets and relations; then, we highlight the three main pillars of the model: domains (or iteration space), schedule, and access relations (or memory accesses).

Integer Sets

An integer set is a set of named integer tuples. A named integer tuple consists of an identifier and a sequence of integer values. For example, "A[1, 2, 3]" is a named integer tuple with identifier "A" and a sequence of integers "[1, 2, 3]". A set contains zero or more named integer tuples as elements and can be described by Presburger formulas. Affine constraints on integer variables with logical operators $\neg$, $\vee$, $\wedge$ and existing quantifier $\exists$, $\forall$ define a Presburger formula. Affine constraints can be either equality or inequality constraints.
Figure 2.6 shows an example of a two-dimensional dense integer set. It consists of a blue shaped triangle containing blue points and a red shaped rectangle containing red points. Only the red and blue points located at integer coordinated form the set, which can be described in set notation as

$$S_1 = \{(i, j) \mid (a \leq i, j \wedge i + j < b) \vee (4 \leq i, j \wedge i \leq b \wedge j \leq 6)\}$$

where a = 1 and b = 8 are parameters and S_1 is the name of the set. The colored shapes (triangle and rectangle) form a set of convex shapes that enclose the set's points. From Figure 2.6, we can easily derive the affine constraints. Considering the red rectangle, we have $i \geq 4 \wedge i \leq 8$ for the left and right bound of the rectangle, and $j \geq 4 \wedge j \leq 6$ for the lower and upper bound, respectively. Generally, an integer set has the form

$$S_{name} = \{\vec{s} \in Z^d \mid f(\vec{s}, \vec{p})\}$$

with $\vec{s}$ being integer tuples in the integer set. d is the set dimensionality, while $\vec{p} \in Z^d$ a vector of d parameters and $f(\vec{s}, \vec{p})$ a Presburger formula that evaluates to

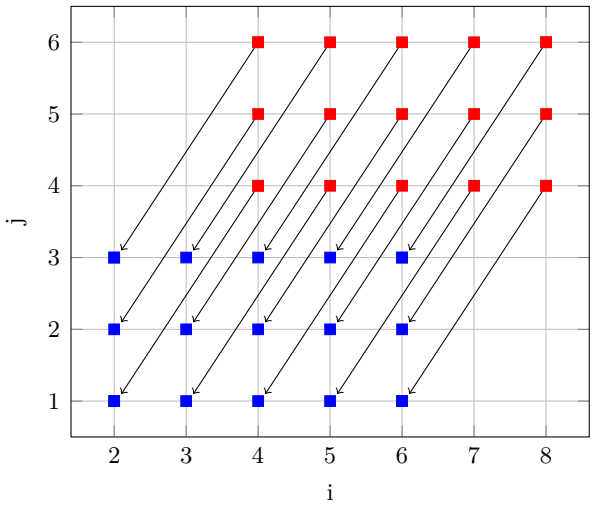


Figure 2.7: Two dimensional integer map. Example taken from [4].

true if and only if $\vec{s}$ is an element of the set S_{name} give parameters $\vec{p}$. In our running example, the Presburger formula is the red and blue points: $(a \leq i, j \wedge i + j < b) \vee (4 \leq i, j \wedge i \leq b \wedge j \leq 6)$. On integer sets, we can define basic operators such as union or intersection. For example, the set in Figure 2.6 can be seen as the union of two disjoint sets, one represented by the blue triangle and one by the red rectangle.

Integer Relations

A binary relation between integer sets defines an integer map. The left-hand term in the relation is called *domain*, while the right-hand side is called *range*. Figure 2.7 shows an integer map which maps points of the domain to points in the range. The relation can be described as

$$M_1 = \{(i, j) \rightarrow (i - 2, j - 3)\}$$

The elements in the range (blue points) are a shifted version of the domain elements (red points). Specifically, each point has been shifted by -2 along the i axis and -3 along the y axis. Each black arrow represents the relation between one element of the domain and one element of the range.
The general form of an integer map is

$$M_{name} = \{\vec{i} \rightarrow \vec{o} \in Z^{d1} \times Z^{d2} \mid f(\vec{i}, \vec{o}, \vec{p})\}$$

where $\vec{i}$ describes the domain expressed as d1-dimensional tuples while $\vec{o}$ describes the range expressed as d2-dimensional tuples. $\vec{p} \in Z^e$ expresses a vector of e parameters. Finally, $f(\vec{i}, \vec{o}, \vec{p})$ represents a Presburger formula that evaluates to true if and only if $\vec{i}$ and $\vec{o}$ are related by M for given parameters $\vec{p}$.
We can define basic operations on integer maps, most of them similar to the ones on integer sets. Operations can be applied to the entire map or just its domain or range. As a simple example, by applying the map M_1 to the red set shown in Figure 2.7, we obtain the blue one.

Libraries for Integer Sets and Maps

Several libraries enable the manipulation of integer sets and relations. Among them, *isl* [28] and Omega [29] are perhaps the most widely used. Both libraries offer similar functionalities, but the underlying algorithms are in most of the cases different. *isl* relies on integer division in its internal representation, while Omega uses the intersection of polyhedra and lattice points to represent sets and relations. Besides, Omega does not support named sets or maps. In certain situations, rational polytopes (a convex hull [1] of a finite set of points) can approximate integer sets by enclosing their points. PolyLib [30] and PPL [31] are two libraries developed to manipulate and perform computation on rational polytopes.
In the rest of this thesis, we will use *isl* and adopt its notation due to its widespread adoption in production compilers like GCC and LLVM and an easy to use templated C++ interface [32]. A nice introduction to *isl* is available in [33].

Limitations of the Model

The model is restricted to static control parts. A SCoP is a code region where the control flow is static. The control flow should not depend on the input data but on parameters that are guaranteed to be constant during execution, known as symbolic parameters. Typically, a SCoP consists of a nested loop where the loop boundaries are affine functions of the surrounding loops and symbolic parameters. Parametric over-approximation of control-flow and runtime speculation are two approaches used today to mitigate such limitations. The statements in the SCoP are side effects free, and loops have boundaries that depend on outer loop iterators and symbolic parameters. Each array subscript is required to be an affine expression of outer loop iterators and symbolic parameters. This requirement comes from the need to model array subscripts in linear-algebraic terms; thus, the following non-affine subscripts $A[i * j]$ is not allowed.

The Origin of the Model

The idea of modeling individual statement executions dates back to 1967 when Karp et al. used such an approach to extract parallelism [34]. A subsequent paper of Lamport used hyperplanes to separate independent parts of a loop nest [35]. Unfortunately, back then, two main problems prevented the immediate application of the model: (1) most of the programs depend on input parameters; they are parametric to given input values. (2) Defining a traversal order of the iterations using control flow constructs was a challenging task. The introduction of parametric integer programming solved the first problem. Integer programming enables to find the optimal solution to linear programming problems depending on the value of integer parameters. Ancourt and Irigoin solved the second problem in their algorithm for scanning polyhedra with DO loops [36]. For more details on the model's origin, an excellent overview is presented by Zinenko in his thesis titled "Interactive Program Restructuring" [37].

[1] The convex hull of a set of points S in n dimensions is the intersection of all convex sets containing S

Polyhedral Frameworks

The polyhedral model works on an internal mathematical representation that operates at a higher-level of abstraction than the traditional syntax-tree representation of a general-purpose compiler. Multiple polyhedral extractors transform the code or better raise it to the model in an automated fashion through parsing. In general, all the extractors identify parts of the program that can be raised automatically or rely on user-specified annotations (e.g., pragmas). Compilers supporting polyhedral optimization have internal abstraction raising algorithms. Several compilers such as IBM-XL [38] or R-Stream [39] have proprietary algorithms others, such as GRAPHITE [40] for GCC and Polly [22] for LLVM, are open source. Extractors also exist for general-purpose languages. Clan [41] is an extensible raising tool for C-like language used by both Pluto [42] and PoCC [43]. Pet uses a real C compiler (Clang) to parse the input code and raise it to the polyhedral model. LooPo relies on custom parsing techniques to raise a subset of C or Fortran or a combination of both as input [44].

2.3.2 Representing Programs

Iteration domains, access relations, and schedule are the core constituents of the polyhedral model.

- **Iteration domain or domain**: represents the set of statement instances that are part of the SCoP.
- **Memory accesses**: a set of read and write access relations that relate each statement instances with the accessed location.
- **Schedule**: assigns to each statement instance an execution time. Program optimizations change the order in which statements are executed. Thus each optimization consists in changing the schedule but not the number of statements that are executed.

Domain and Memory Accesses

Every loop's iteration is identified by an integer vector in a k-dimensional space, where k is the depth of the loop, the coordinates of which correspond to the values of the loop induction variables. Each statement other than control-flow constructs has an associated symbolic name and a set of integer vectors describing the iterations at which it is executed, commonly referred to as the *iteration domain*. When the static control conditions are respected, the iteration domain can be concisely denoted by a system of affine inequalities. For example, the iteration domain of the GEMM kernel in Listing 8 is expressed as $\{S_1(i,j) \mid 0 \leq i,j < N; S_2(i,j,k) \mid 0 \leq i,j,k < N\}$ in the tagged-tuple notation introduced by iscc [45], where N is a *parameter*. Individual executions of statements inside loops are called *statement instances*.
Internal computation is rendered as arithmetic expressions and function calls with array elements as arguments. The conditions imposed on the array subscripts are essentially the same as control-flow conditions. Array accesses can be thus precisely encoded as relations between vectors in the iteration space and vectors in the array space whose coordinates are values of the accessed subscripts. These relations are

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

Listing 8: Generalized matrix multiplication kernel.

defined by piece-wise quasi-affine functions[2]. In our running example from Listing 8, the access relation for statement `S1` is

$$\{S_1(i,j) \to \text{beta}(); S_1(i,j) \to \text{D}(i,j); S_1(i,j) \to \text{C}(i,j)\}.$$

where beta is a constant while D and C two-dimensional arrays.

Schedule

The iteration domain defines the statement instances that should be executed but not their order. The latter is defined by a *schedule* that maps points in the iteration space to points in the time space. Although it is possible to express schedules as relations or piece-wise quasi-affine functions, it is often undesirable to do so because statements are likely to share part of their schedules due to the commonly nested structure of the code. In addition, it is also necessary to reflect the relative syntactic order of loops and statements within them, which is often achieved through auxiliary dimensions whose values are always constant for the given statement. For example, the schedule of Listing 9 in relation form is
$\{S(t,i,j) \to (t,i,j,0); T(t,i,j) \to (t,i,j,1)\}$. Note the duplication of (t,i,j) and the presence of auxiliary dimensions to specify the execution order within the loop nest.
Addressing these challenges, Verdoolaege et al. [47] proposed *schedule trees* as a way to represent schedules in the polyhedral model. In schedule trees, statements that share a common partial schedule share an ancestor that describes this partial schedule, thus removing duplication. This loop nesting maps naturally to a parent-child relation, whereas textual ordering maps to the left-to-right order of sibling nodes.
Nodes in schedule trees can be one of numerous types. Let us briefly present node types relevant to our examples.

- *Domain*—the iteration domain, always located at the root of the tree;
- *Band*—partial schedule of one or multiple loops
  (the name refers to the notion of a *tilable band* of loops [48]);
- *Filter*—restricts the instances of the iteration domain;
- *Sequence*—imposes the order among its children;
- *Set*—does not impose any order among its children;
- *Mark*—allows the user to mark specific subtrees.

[2] A piece-wise quasi-affine expression is a list of pairs of Presburger sets and quasi-affine expressions [46]

```
    for (int t = 0; t <= T; t++)
      for (int i = 1; i < 1023; i++)
        for (int j = 1; j < 1023; j++)
          if (t % 2 == 0)
S:          A[i][j] += B[i-1][j] + B[i][j+1] + B[i][j] + B[i][j-1]
                     + B[i+1][j];
          else
T:          B[i][j] += A[i-1][j] + A[i][j+1] + A[i][j] + A[i][j-1]
                     + A[i+1][j];
```

Listing 9: A 2D 5-point stencil kernel.

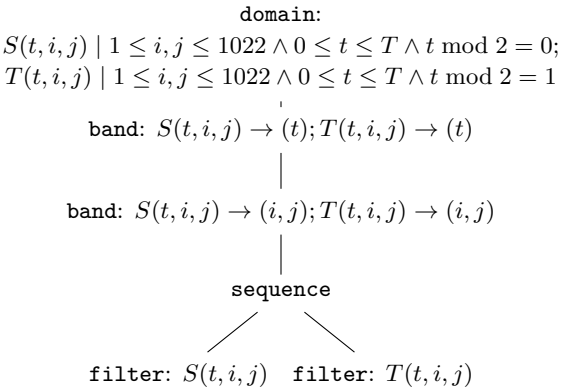


Figure 2.8: Schedule tree representation of Listing 9.

- *Extension*—introduces new statement instances local to the subtree with respect to its prefix schedule.

All nodes except *sequence* have at most one child.
Listing 9 presents a 2D stencil program. The corresponding schedule tree is depicted in Figure 2.8. It uses two band nodes: the outer one provides a partial schedule representing the individual time steps, whereas the inner one provides a partial schedule enumerating all data points within a given time step. Loops cannot be interchanged across bands. Finally, the individual statement types are enumerated within a sequence node. When combined across all dimensions, the schedule tree defines the execution order of the program.
The tree structure not only simplifies the schedule representation but allows the easy application of local transformations. Such transformations include the combination of nested band nodes into a single band, the splitting of a multi-dimensional band node into individual bands, but also more complex modifications such as band tiling or compositions of affine transformations.

2.3.3 Summary and Conclusion

This chapter introduced a high-level overview of the basic building blocks and internal machinery of a general-purpose compiler. We highlighted today's compiler's main limitation, mainly the single abstraction paradigm and the shift in trend to multi-level IR framework to overcome such limitation. In this context, we presented

the MLIR framework, the first multi-level IR compiler, which provides a toolkit to design and optimize IRs. Finally, we introduced the polyhedral model, a state-of-the-art program representation for loop-based programs. Contrary to other models, it allows modeling every single iteration of the program, allowing precise dependencies computation and powerful loop transformations. We introduced the three main pillars on which the model is based: the iteration domain, schedule, and access relations.

3

Abstraction Raising in the Polyhedral Model

With Declarative Loop Tactics or Loop Tactics for short, a framework of composable program transformations based on an internal tree-like program representation of a polyhedral compiler, we enable abstraction raising in general-purpose compilers. The framework is based on a declarative embedded C++ API built around easy-to-program matchers and builders, providing the foundation for developing loop optimization strategies. Using our matchers and builders, we express computational patterns and core building blocks, such as loop tiling, fusion, and data-layout transformations, and compose them into algorithm-specific optimizations.
Several application domains can benefit from Loop Tactics; for two of them, stencils and linear-algebra, we show how developers can express sophisticated domain-specific optimizations as a set of composable transformations or calls to optimized libraries. By allowing developers to add highly customized optimizations for a given computational pattern, we expect our approach to reduce the need for DSLs and extend the range of optimizations that a general-purpose compiler can perform.

3.1 Declarative Loop Tactics

With *Declarative Loop Tactics* we introduce a framework to make modern constraint-based loop transformations as accessible as classical tree-based compiler transformations. Unlike conventional compilation approaches centered around syntax trees, polyhedral compilation techniques typically operate on some representation of a schedule disconnected from any syntactic form. Such schedule representations are algebraic objects allowing an entirely new schedule to be computed as a solution to a (sequence of) linear optimization problems. In doing so, however, polyhedral transformation acts as a black box for compiler developers leaving them with the only option of manual optimizations if imprecise cost functions are used to solve the linear optimization problem. Unlike others polyhedral compilation flows, Declarative Loop Tactics—thanks to the nature of the schedule tree representation that combines schedule and syntactic aspects in a tree shape—allows compiler developers to transform suitable programs *step-by-step*, combining tree manipulation mechanisms that are commonplace in compilers with the precise analyses of the polyhedral model. In the following, we present our framework, built around tree and access relation

This Chapter is based on: L. Chelini et al., "Declarative Loop Tactics for Domain-specific Optimization". ACM Trans. Archit. Code Optim. 2020

```
auto matcher =
  sequence(
    filter(band(isPermutable, // filtering function
                anyTree()),   // wildcard node
    filter(leaf())));         // explicit leaf
```

Listing 10: Schedule tree matchers declaratively describe the structure of a tree.

matchers describing computational patterns, and schedule tree builders describing loop transformations. We start by informally introducing Loop Tactics using examples, while Section 3.4 provides a more formal description of the syntax using the extended Backus-Naur form. The provided examples require some knowledge on the schedule tree structure, introduced in Chapter 2 paragraph 2.3.2.

3.1.1 Polyhedral Schedule Tree Matchers

General concepts A Schedule tree matcher enables a declarative description of a pattern in the schedule tree. It essentially replicates the node type-based structure of the schedule tree with additional filtering and wildcarding capabilities. A node matcher consists of an expected (possibly unspecified) node type, a list of children nodes, and an optional filter. In addition to all schedule tree node types, it may expect *any* node type or a non-empty *list* thereof. Tree leaves can be matched explicitly using a special node type. As an example, the matcher shown in Listing 10 matches all the sub-trees starting at a sequence node with exactly two filters as children, the first of which has a permutable band as a child (checked via a Loop Tactics provided function `isPermutable`) while the second has no children.

Matching Procedure A tree matcher is specified by the top node it must match, referred to as relative root. The matching procedure starts at the specified node in the schedule tree and performs a simultaneous depth-first pre-order traversal of the schedule tree and the tree formed by descendants of the relative root matcher. If a mismatch is detected until the entire matcher is traversed, it is immediately reported, and the traversal stops. Multiple patterns can be recognized at once using classical prefix-tree and self-similarity approaches.

Programming Interface The API to construct schedule tree matchers is designed to visually resemble the structure of the tree itself in a declarative way. Named variadic functions correspond to node types, and the argument lists enumerate children nodes. This approach enables static checking of tree invariant properties (e.g., some node types only allow to have one child) or only children of a specific type. Leading arguments include *optionally* a filtering function and a reference to the "placeholder" node. The filtering function allows the caller to control the matching more precisely by considering non-structural aspects of the schedule tree such as permutability or number of schedule dimensions. For example, identifying a permutable band node requires both structural and non-structural properties, as shown in Listing 10. The references are used to capture certain nodes in the matched subtree, similarly to captured groups in regular expressions, for future use by the

```
schedule_node node, body;
auto matcher = band(node, isPermutable, anyTree(body));
auto builder =
  band([&](){ return tileSchedule(node, tileSize);},
    band([&]() { return pointSchedule(node, tileSize);},
      subtree(body)));
replaceDFSPreorderOnce(scheduleTree, matcher, builder);
```

Listing 11: Declarative specification of rectangular loop tiling. Functions tileSchedule and pointSchedule divide and take modulo tile sizes, respectively.

caller. If the captured nodes are non-empty, the underlying subtree is guaranteed to have the structure described by the matcher.

3.1.2 Polyhedral Schedule Tree Builders

The imperative-style schedule tree construction interface provided by *isl* does not allow for declarative tree construction. As a consequence, we provide schedule tree builders whose programming interface is close to that of the tree matchers. Named variadic functions specify the type of the node. The nesting of the function calls reflects the structure of the tree, and the optional leading arguments accept functions that are used to build the non-structural properties of each specific node type (partial schedules for band nodes, conditions for filter nodes, and so on). You can think of a builder as a description of the subtree to build. It may be transformed into a standalone tree if needed (i.e., if it is rooted at a domain or an extension node) or grafted at a leaf of an existing tree.
Listing 11 shows how matchers and builders can be combined together to tile a simple rectangular loop. The matcher looks for permutable band nodes anywhere in the tree and captures both the node (`scheduleTree`) and its child subtree (`body`). Permutability for a given band is tested using Loop Tactics' function `isPermutable`. The builder splits the node into two nested bands by taking the integer division and the modulo parts of the schedule via Loop Tactics provided functions `tileSchedule` and `pointSchedule`. The child subtree is kept intact. The example can be extended to support more advanced transformations like full/partial tile separation or to limit tiling to the deepest bands without loosing a significant portion of the transformation code clarity.

3.1.3 Polyhedral Relation Matchers

General Concepts Polyhedral relation matchers allow the caller to identify relations that have certain properties in a union of access relations [49, 50]. The capturing mechanism operates through *placeholders*, each of which has two data components: a constant *pattern* and a variable *candidate*. Each relation is checked against the pattern and, in case of a match, may yield one or more candidates. The description of what constitutes a match and how the candidates are generated is external to the matching engine and can be provided by the user. An example of a pattern that yields multiple candidates is "access relation with one output dimen-

sion fixed to a (literal) constant". In this case, a candidate *may* be generated for each output dimension fixed to a constant or, for other use cases, for each value of the constant.
A *match* is an assignment of candidates to placeholders. A union of relations may yield zero or more matches against the given matcher. In many cases, candidates assigned to different placeholders are required to be distinct. We require this by default, considering other cases to be *invalid assignments*. At the same time, if a placeholder is reused within the same matcher expression, we also require that all appearances be assigned the same candidate. In any case, the candidate comparison only takes the candidate descriptions into account, not the differences between spaces of the relations. This behavior allows the matcher to connect different relations in the union with greater precision. For example, it captures the fact that two relations exist that are either both bijective or both non-bijective. To support edge cases, the user can override the definition of a valid assignment, for example to allow the same candidate to be assigned to different placeholders.

Matching Procedure The matching procedure is decoupled into two stages, which provides sufficient flexibility without sacrificing performance. First, the engine traverses all relations one by one and defines the set of suitable candidates for each placeholder. If at least one of the placeholders has no suitable candidates, the absence of a match is reported immediately, and the procedure stops. Then the engine traverses the space of all possible assignments of candidates to the placeholders and checks whether the assignment is valid. As the space of possible assignments is combinatorially large, we opt for a branch-and-cut traversal approach. Partially formed assignments are passed to the validation function before adding more placeholder-candidate pairs to the assignment. If the partial assignment is reported to be invalid, further exploration is not performed. This approach can be easily transformed into a branch-and-bound approach, if the assignment needs to be optimal in some sense, or altered to change the exploration to validation ratio if the validation itself is expensive.

Programming Interface Implementation of the matching procedure, or the matching engine, is a template definition in the C++ sense. An instantiation of the matching engine is specified by data structures for the pattern and the candidate. In addition, it implements functions to define a set of candidates for a given access relation and whether a candidate assignment to the matchers constitutes a valid match. We provide pattern and candidate descriptions for affine expressions $\omega = k * \iota + c$, where k and c coefficients form the pattern whereas ω and ι define a candidate by matching one of the output and input dimensions, respectively. When targeting access relation unions, we can assume that the source space of all relations is the same (e.g., the schedule space), so it is convenient to operate only within the relation range.
The programming interface is similar in spirit to that of schedule tree matchers, except for the tree parent-child relations, and is based on nested function calls that progressively construct the matcher. Listing 12 illustrates how one can identify whether the same 2D array is accessed directly and with transposition.

```
auto readsAndWrites = /* get read and write accesses */;
auto _i = placeholder(), _j = placeholder();
auto _A = arrayPlaceholder();
auto matcher = allOf(access(_A, _i, _j), access(_A, _j, _i));
```

Listing 12: Access relation matcher for transposed accesses.

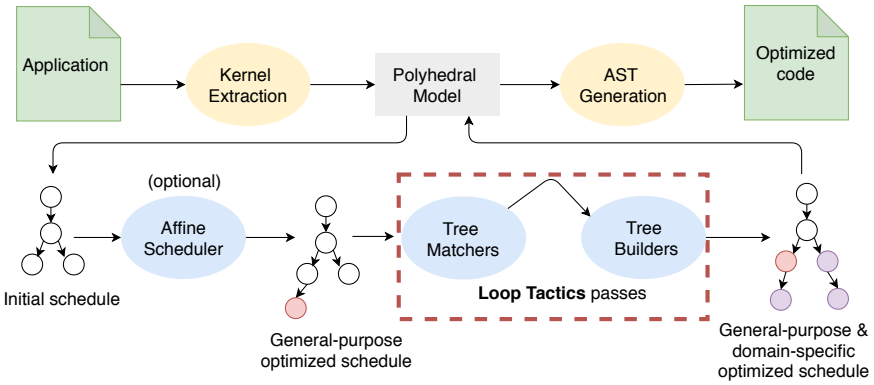


Figure 3.1: Loop Tactics can be easily integrated into the architecture of state-of-the-art optimizers. It can be placed immediately after a general-purpose polyhedral optimizer (affine scheduler) or replace it.

Sometimes, it is more convenient to consider individual output dimensions separately. For example, in the transposed access case, the caller would like to know whether, in $\{(i,j) \to A(a_1 = i, a_2 = j); (i,j) \to B(b_1 = j, b_2 = i)\}$, the first dimension of the output space in one relation (a_1) is equal to the second dimension of the output space in another relation (b_2). We handle such situations by augmenting the pattern description with the expected position in the output space, thus projecting the access relation onto that dimension and continuing to check the properties of a relation with one-dimensional output space. Such per-dimensional behavior is particularly useful, for example to detect the presence of temporal or spatial locality in array accesses.

3.2 Compilation Flow

Before proceeding in illustrating how Loop Tactics enables domain-specific compilation in general-purpose flow, let us describe where Loop Tactics fits into different compilation flows. The high-level view is illustrated in Figure 3.1. Loop Tactics can be applied instead of or in addition to an existing affine scheduler, operating on schedule trees. It relies on the caller to supply the input as schedule trees and convert the transformed trees back into the original representation. As such, it is applicable to both source-to-source and IR-to-IR flows.

IR-to-IR flow First, we demonstrate how Loop Tactics is embedded into the LLVM infrastructure by leveraging Polly [51]. Loop Tactics is not concerned with lowering to the LLVM IR or performing target-specific low-level optimizations such as instruction selection. Polly processes the input LLVM IR to detect the parts of

the IR that are suitable for optimization, referred to as SCoPs. It then represents the IR as a schedule tree that Loop Tactics can consume. Polly may run an affine scheduler [52] that transforms the tree before sending it to the Loop Tactics framework. Pattern detection using tree matchers and transformations using tree builders occur as additional passes that modify the schedule tree. After all the optimizations have been performed, Polly translates the schedule tree into an AST and then further down into the LLVM IR. We use this flow in Sections 3.3.1 and 3.3.3, where we compile LLVM IR down to executable using LLVM tools.

Source-to-source flow For source-to-source compilation, we embedded Loop Tactics into the *pet* tool [53], which we use to extract SCoPs from C code and produce C or CUDA code. Similarly to Polly, *pet* relies on *isl*'s schedule trees, making it possible to (largely) reuse the integration of Loop Tactics into the polyhedral flow. After the tree has been transformed by the affine scheduler and/or Loop Tactics, we send it back to *pet* for code generation. To obtain a final executable, we rely on general-purpose compilers such as the Intel compiler as described in Section 3.3.2.

3.3 Results

In this section, we evaluate the applicability of our framework by considering two domain-specific optimizations for GEMM-like and stencil kernels. The former is based on a recently introduced custom optimization pass in Polly that re-creates hand-tuned optimizations for GEMM-like kernels, see Section 3.3.1. The latter focuses on a data-layout transformation to reduce stream alignment conflicts in stencil patterns, namely dimension-lifted transposition (Section 3.3.2). Subsequently, we show how Loop Tactics can be used to call routines from vendor-optimized libraries transparently if such libraries are available for the target. Otherwise, it generates optimized code, see Section 3.3.3. For the case studies considered here, the hardware platforms are a Volta V100 GPU, an IBM Power 9 AC922 clocked at 3.8 GHz, and an Intel Core i7-7700 (Kaby Lake family) clocked at 3.6 GHz with Intel Turbo Boost up to 4.2 GHz. Intel Turbo Boost is disabled. Every single measurement or result reported in the following sections is the arithmetic mean of five runs.

3.3.1 Hand-tuned GEMM-like Optimization

Generalized matrix multiplication (GEMM BLAS kernel) is one of the important computation patterns and is the most commonly optimized kernel in history [54]. However, state-of-the-art compilers achieve only a fraction of the theoretical machine performance for a simple textbook-style implementation [55]. A recent improvement within Polly introduced a custom transformation for GEMM-like kernels that is controlled outside of the main affine scheduling mechanism [55]. This transformation applies to a generalized case of tensor contraction of the form `C[i][j] = E(A[i][k], B[k][j], C[i][j])`, where the dimension `k` is contracted and `E` represents some operations between tensors. The matrix–matrix multiplication from Listing 8 is an example of such a contraction with $\mathrm{E} = (\times, +)$.

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

auto is3D =
  [&](schedule_node band) {
  // is the band's
  // schedule 3-dimensional?
  return band.dim() == 3;
};

auto isPermutable =
  [&](schedule_node band) {
  // is the band permutable?
  return band.permutable() == 1;
};
```

```
auto hasGemmPattern = [&](schedule_node n)
{
  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),
          // for syrk use (_A, _j, _k)
          access(_B, _k, _j));
  auto mWrite = allOf(access(_C, _i, _j));
  return match(reads, mRead).size() == 1
    && match(writes, mWrite).size() == 1;
};
```

Listing 13: Schedule tree and access relation matcher for a tensor contraction $C \to \alpha \times C + \beta \times A \times B$. Callback functions `is3D` and `isPermutable` to test node properties are also shown. Simply changing one line in the access relation matcher callback (hasGemmPattern) is enough to capture other patterns (i.e., syrk). See Section 3.3.3 for more details.

Candidate loops To qualify for the transformation, the kernel must

- be a perfectly nested loop that satisfies the requirements of the polyhedral model;
- contain three non-empty one-dimensional loops with induction variables incremented by one;
- contain an innermost statement $C_{\pi C(IJ)} = E(A_{\pi A(IK)}, B_{\pi B(KJ)}, C_{\pi C(IJ)})$, where $A_{\pi A(IK)}$, $B_{\pi B(KJ)}$, $C_{\pi C(IJ)}$ and $\pi A(IK)$, $\pi B(KJ)$, $\pi C(IJ)$ are accesses to tensors A, B, C and permutations of the enclosed indices, respectively. The term E is a generic expression that contains at least three reads from tensors A, B and C.
- ensure that the interchange of I and J is valid, whereas K is interchangeable if and only if K contains only one element, or an associative operation is used to update C.

The candidate loops are found by implementing the aforementioned conditions as schedule tree and access relation matchers with a set of callback functions (Listing 13). In particular, we look for a subtree containing a three-dimensional permutable band with a single statement (leaf) featuring specific access patterns: at least three two-dimensional read accesses to different arrays, one write access, and a permutation of indices that satisfies the placeholder pattern $[i, j] \to [i, k][k, j]$.

```
/* obtain node, e.g., from a
   matcher */
schedule_node node = /*...*/;
auto macroKernel =
 band([&]() {
  return tileSchedule(
   node,/*L2 sizes*/);},
   band([&]() {
    auto sched =
     pointSchedule(node,/*L2 sizes*/);
    return swapDims(sched, dimI, dimK);
   }));
```

```
/* obtain node, e.g., from a
   matcher */
schedule_node node = /*...*/;
auto microKernel =
 band([&]() {
  return tileSchedule(
   node,/*Vec sizes*/);},
   band([&]() {
    return unrollAll(
     pointSchedule(
      node,/*Vec sizes*/));
  }));
  /* Apply packing
     transformation. */
```

Listing 14: Macro-kernel builder (left) performs tiling and loop permutation for improving L2-cache locality. Micro-kernel builder (right) performs further tiling and unrolling to enable vectorization.

Expressing transformations The domain-specific optimization is derived from [54] as follows: first, we rearrange the band dimensions such that `j` will be the outermost dimension, followed by `k` and `i`. Then we apply multi-level tiling and loop interchange to create three nested loops around the macro-kernel and two additional nested loops around the micro-kernel. Specifically, the builder for the macro-kernel reported in Listing 14 on the left performs L2-cache tiling and interchanges the newly created point loops. Tiling is applied using Loop Tactics' functions `tileSchedule` and `pointSchedule`, whereas loop interchange uses `swapDims`. After applying the first builder, we define and apply the second builder to create the micro-kernel loops by tiling the points loops such that they fit into vector registers and fully unroll the new innermost one to simplify subsequent vectorization. Loop unrolling is performed using the Loop Tactics' function `unrollAll`. Finally, we perform the packing transformation expressed as a matcher-builder pair by relying on the existing data-layout infrastructure available in the polyhedral optimizer (Listing 14 right-hand part).

Evaluation To evaluate the quality of our implementation, we hash the binaries generated by our custom Polly, with Loop Tactics embedded, with an original version containing the custom transformation (git commit 592b2406) and compare them for strict equality. We consider a tensor contraction with E $= (\times, +)$ of the form $C \rightarrow \alpha \times C + \beta \times A \times B$, where A, B, and C are square matrices and α and β are constants set to 1 as in [55]. We use the compilation strings `Clang -O3 -march=native -mllvm -polly -ffp-contract=fast -ffast-math` and `Clang -O3 -march=native -polly -mllvm -polly-enable-matchers-optimization -ffp-contract=fast -ffast-math` for the original Polly and the modified one, respectively. Identical binaries imply the same performance as shown in Figure 3.2 that presents the results for a single-threaded implementation using double and single-precision operands on the Intel machine. Each plot shows the achieved GFLOP/s on the y-axis versus the matrix dimensions on the x-axis. Both Loop Tactics and the Polly's cus-

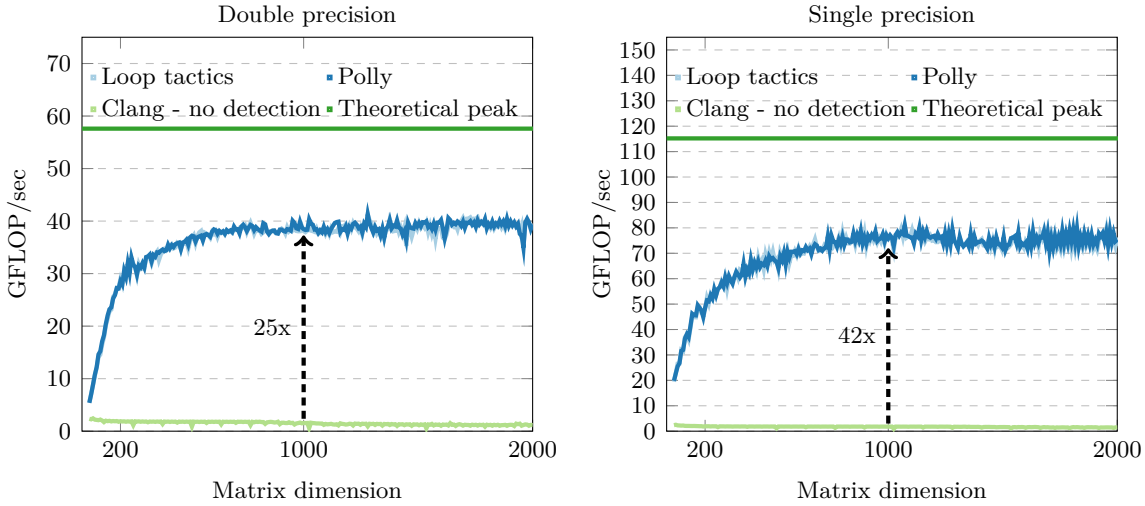


Figure 3.2: Performance obtained for double and single-precision operands. The GEMM tactic produces the same binary code as a hand-tuned, Polly's custom transformation.

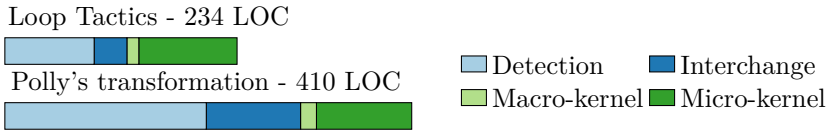


Figure 3.3: The GEMM tactic reduces the compiler code footprint by almost a factor of 2 compared to the Polly's custom transformation.

tom transformation achieved non-negligible speedups compared with `Clang -O3 -march=native` (Clang - no detection). However, the combination of matchers and builders is able to reduce the compiler code footprint by a factor of almost 2, see Figure 3.3. In addition, it is now possible for compiler developers to add highly customized optimizations within a short time frame of weeks instead of months as for ad-hoc pattern matching [55]. Finally, as fast compilation time is important, we evaluate the overhead introduced by Loop Tactics. To do so we compare the compilation time of our custom Polly with the original version, both compiled in Release mode. The original version takes 0.354 seconds while Loop Tactics 0.362 seconds, which correspond to only a 2% increase. Loop Tactics introduces a negligible compile-time overhead.

3.3.2 Short-SIMD Stencil Vectorization

Stencils represent an important class of computation patterns used in many scientific domains [56, 57]. As they typically involve accesses to multiple adjacent array elements along multiple dimensions inside nested loops with (mostly) static control flow, they fit the polyhedral model.
Unfortunately, polyhedral compilation based on affine scheduling is often counterproductive for stencils. Direct attempts to minimize dependence distances lead to serialization of computation or complex control flow overhead, or both, making the

```
band(capturedBand,
     and_(not_(hasDescendant(band(anyTree()))),
               isLastCoincident),
     anyTree());
```

Listing 15: Tree matcher for DLT transformation.

```
auto readsAndWrites = /* get read and write accesses */;
auto stride0Accesses =
    match(readsAndWrites, access(dim(-1, stride(ctx, 0))));
auto stride1Accesses =
    match(readsAndWrites, access(dim(-1, stride(ctx, 1))));
auto allAccesses = match(readsAndWrites, access(any()));
return stride0Accesses.size()
    + stride1Accesses.size() == allAccesses.size();
```

Listing 16: Access relation matchers for DLT transformation.

transformed code *slower* than the original [58, 59]. Several techniques, more or less closely related to the polyhedral compilation, address specifically stencil parallelization and vectorization [60]. As these techniques are often not adapted to non-stencil cases, one must first find the stencil-like part of the program and then apply the transformation. Let us illustrate how an effective technique to remove stream-alignment conflicts through a data-layout transformation [61] can be expressed in our framework.

Candidate loops The data-layout transformation (DLT) technique [61] applies only to vectorizable *innermost* loops with no loop-carried dependencies. In schedule tree nomenclature, we should start by finding all band nodes that have no other band nodes as children and that have the last coincidence flag set (assuming that coincidence—the possibility of parallel execution—was computed based on all dependencies). This is shown in Listing 15.
If the band is permutable, it does not matter whether the coincident flag is set for the last dimension or any other dimensions of the band members because the loops can be trivially permuted. This extension to the original technique and the corresponding tree transformation are easy to propose and implement in our approach, but were not considered in the original paper [61]. The transformation validity is further restricted to statements that access either the same or contiguous elements of an array in consecutive iterations of the innermost loop. This can be represented by stating that the innermost accesses dimension (Python-style index `-1`) must have a stride of either 0 or 1, as shown in Listing 16.

Vector-lane conflicts Finally, DLT is only necessary when vector-lane conflicts exist that cannot be removed through loop shifting or, in particular, if the same value is accessed through *different references* in different iterations of the innermost loops. This condition can be transformed into a sequence of set operations forming a system of constraints followed by an emptiness check. Although it can be used inside the relation matcher to create a list of candidates before checking whether all

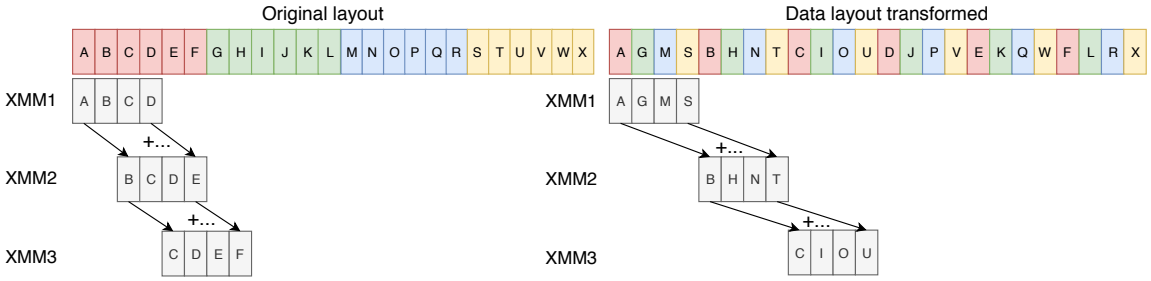


Figure 3.4: Data layout transformation for a SIMD with four vector lanes. Elements mapping to vector slots in vector registers (XMMi) of four elements each is also shown.

accesses do indeed match, it is arguably more pragmatic to perform the operations directly.

Expressing data-layout transformation The data-layout transformation consists of placing adjacent array elements at a distance of L from each other, where L is the number of vector lanes. Figure 3.4 shows the original and the transformed memory layout for $L = 4$. It can be seen that previously adjacent array elements (i.e., A and B) are now spaced further apart. The data layout mapping can be expressed as an affine function

$$\{(\vec{\iota}, i) \to (\vec{\alpha}, a) \mid \vec{\alpha} = \vec{\iota} \wedge a = L \cdot i - (B_U - B_L) \cdot \lfloor \tfrac{i \cdot L}{B_U - B_L + 1} \rfloor \} \quad (1)$$

where B_L and B_U are the lower and the upper inclusive bounds on the accessed elements, respectively. Figure 3.4 further shows how elements map to vector slots in vector registers of four elements each for a single iteration of a Jacobi-3pt stencil. Array elements in the layout transformed can be loaded into the vector slots using *aligned* loads. Therefore the compiler doesn't need to issue additional *non-aligned* loads to bring interacting elements in the same vector slot before performing the arithmetic operation as it is the case for the original layout. If this transformation had been performed on the iteration space rather than on the subscript space, it would have corresponded to loop strip-mining followed by loop interchange and coalescing. Once such a union of affine functions is constructed, it can be used in a tree builder that injects the transformation as presented in Listing 17. First, the builder introduces the new statements for the copies "to" and "from" the transformed array through an extension node, lines 7–9 and 13–15, respectively. We take also into account the fact that additional boundary cells need to be inserted during the copy "to". Below the extension node, copies "to" are scheduled before the main computation (lines 10–12), which itself is scheduled before the copies "from" using a combination of sequence and filter nodes. Finally, partial schedules are specified for the copies, and the original subtree is replicated for the main computation. In practice, the transformation is only applicable to loop iterations that fully fit into vector lanes[1] and requires additional edge-case handling otherwise. Although these

[1]For example, this can be ensured with full/partial tile separation, given that the tile size is equal to the number of vector lanes.

```
schedule_node capturedBand = /* obtain node, e.g., from a matcher */;
auto dlt = /* build DLT function using equation (1) */;
auto from = /* build copy function from DLT layout: B[i] = B_DLT[dlt(i)] */;
auto to = /* build copy function to DLT layout: A_DLT[dlt(i)] = A[i] */;
extension([&](){return from.domain(dlt()).unite(to.domain(dlt()));},
  sequence(
    // introduce schedule for copy-in to DLT layout
    filter([&](){ return to.range();},
      band([&](){ return to.range().schedule();}))
    // introduce DLT and original subtree computation
    filter([&](){ return dlt().domain();},
      subtree(capturedBand))
    // introduce schedule for copy-out from DLT layout
    filter([&](){ return from.range();},
      band([&](){ return from.range().schedule();})));
```

Listing 17: Loop Tactics enables easy integration of data-layout transformations using builders to inject copy-in and copy-out around the main computation.

edge cases can also be expressed declaratively as tree builders, they are not essential to our presentation so we have omitted them for the sake of clarity.
Listing 17 also lays the foundation for most data-layout or memory-related transformations that rely on *copying* the data to and from one array to another. Most such transformations follow the extension/sequence/3-filter pattern where only the properties of nodes are different. Copies "to" or "from" may be omitted in cases where the initial or the final values are irrelevant to the task, creating two other possible tree transformation patterns.

Access rewriting The final step of DLT consists of changing the original stencil computation to access the transformed array instead of the original one and, optionally, emitting vectorization pragmas. To enable vectorization, it is necessary to access the transformed array in *sequential* order, and the data required by each iteration now needs different subscripts. In particular, adjacent elements are now located at a distance of L. This can be easily expressed using our relation-rewriting mechanism (Listing 18). The pattern constitutes the stride-1 access subscript to be transformed, whereas the candidate is an affine function that defines the new subscript and array name. This affine function consists of a linear and a constant part, and the latter needs to account for the vector length chosen and the boundary cells introduced.

Evaluation We implement the DLT tactic in the source-to-source compilation flow by relying on ICC to vectorize our transformed code as was done in [61]. To evaluate the quality of our transformation, we apply the DLT to four variants of the Jacobi kernel—a three-point single-dimensional stencil, a five-point "star", a nine-point "box" two dimensional stencil, and a seven-point three-dimensional stencil. While building our tactic, we consider $L = 8$, the number of vector lanes, to exploit the highest vector instruction set available on our Intel i7-7700 (AVX2). Neverthe-

```
auto stride1Accesses = /* obtain strided-1 accesses from matchers */;
auto aff = /* build affine expr. for pattern */;
findAndReplace(stride1Accesses, replace(
    access(dim(-1, aff)), // pattern
    access(dim(-1, [&]() { aff.payload_.id = "A_DLT"; // candidate
                           aff.payload_.linear + L * aff.payload_.constant;
                           return aff; }))));
```

Listing 18: Rewriting access relations for DLT transformation.

```
/* Original computation */
vmovups A(,%rdx,4), %ymm7
vmovups 32+A(,%rdx,4), %ymm11
vaddps 4+A(,%rdx,4), %ymm7, %ymm8
vaddps 36+A(,%rdx,4), %ymm11, %ymm12
vaddps 8+A(,%rdx,4), %ymm8, %ymm9
vaddps 40+A(,%rdx,4), %ymm12, %ymm13
vmulps %ymm9, %ymm5, %ymm10
vmulps %ymm13, %ymm5, %ymm14
vmovups %ymm10, 4+B(,%rdx,4)
vmovups %ymm14, 36+B(,%rdx,4)
/* More code */
```

```
/* After data-layout transformation */
vmovups 32+A_DLT(,%rdx,4), %ymm7
vmovups 64+A_DLT(,%rdx,4), %ymm11
vmovups 96+A_DLT(,%rdx,4), %ymm15
vaddps A_DLT(,%rdx,4), %ymm7, %ymm4
vaddps %ymm11, %ymm7, %ymm8
vaddps %ymm11, %ymm4, %ymm5
vaddps %ymm15, %ymm8, %ymm9
vmulps %ymm5, %ymm3, %ymm6
vmulps %ymm9, %ymm3, %ymm10
vmovups %ymm6, 4+B_DLT(,%rdx,4)
vmovups %ymm10, 36+B_DLT(,%rdx,4)
/* More code */
```

Listing 19: Comparison between the original computation and the one after data-layout transformation. DLT effectively reduces loads from A reference.

less, L is merely a parameter and can be changed to address wider or narrower vector extensions. We compile using the Intel C compiler ICC v19.0 with the `-O3 -xHost` options. We compare the layout-transformed ICC auto-vectorized version generated by our compilation flow with a reference auto-vectorized code (without layout transformation). The original program is tiled such that all the references fit nicely in the L1 cache, as done in [61]. Similar to [61] we assume an outer loop with T iterations around the stencil loops—as it is common in stencil codes—such that a one-time layout transformation cost to copy from the original layout to the transformed one represents a negligible overhead. For the stencils observed, considering $T = 500$ as in Polybench 4.1, the overhead introduced is less than 3% of the total execution time, hence we don't account for it. The speedups for single-precision operands are shown in Figure 3.5. The auto-vectorized DLT code improves upon the reference code in all cases. Performance gains are due to the reduction in unaligned loads in the innermost loop. For the 3-pt stencil, the speedup is minimal. Even though the number of unaligned loads is slightly reduced for 1D stencils, the overhead of a small number of additional loads is well hidden by modern CPU architectures. However, the 5-pt, the 9-pt 2D and the 7-pt 3D stencils achieve speedups of 50, 23 and 11% respectively, due to a significant reduction in unaligned loads and reduced register pressure. The assembly snippet in Listing 19 shows two innermost loop computations for a 3-pt stencil. Computation with the DLT array (A_DLT) shows fewer memory loads compared with the original computation.

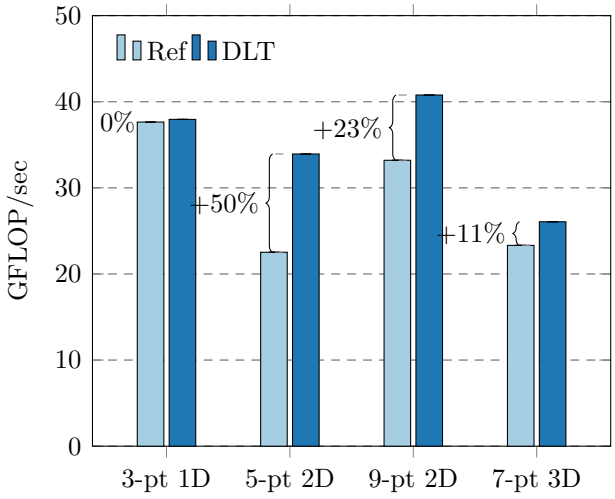


Figure 3.5: Performance and speedup for Jacobi kernels. Ref is the auto-vectorized version without data layout transformation. DLT is the auto-vectorized version with layout transformation.

3.3.3 Transparent BLAS Optimization

Matching Libraries Until now, we have demonstrated how Loop Tactics enables domain-specific optimizations by specifying custom transformations for a given computational pattern. However, Loop Tactics matching mechanism can also be used to recognize code fragments that correspond to common high-performance library calls. Such vendor-optimized libraries are often necessary to achieve peak performance on accelerators [62]. Thus Loop Tactics matching mechanism opens up the possibility to invoke automatically and transparently for the application routines from vendor-optimized libraries. Domain-specific compilers such as Halide [63], TVM [64], and Tensor-flow [65] already lower high-level constructs to vendor-optimized routines; however, they focus on specific domains. Loop Tactics aims at enabling such optimization in general-purpose compilers. We illustrate how this can be achieved in a *portable* way by Loop Tactics targeting both CPU and GPU BLAS libraries. BLAS parameters (i.e., values of α and β) are automatically collected by Loop Tactics. For GPUs, Loop Tactics handles the data transfers as demonstrated in Listing 17. We implemented matchers for level-3 BLAS `GEMM`, `SYMM`, `SYRK`, `SYR2K`, `TRMM`, and a level-2 BLAS `GEMV`, and applied them to all kernels in Polybench 4.1 using the `LARGE` input set. The BLAS kernels are detected by *composing* together elementary matchers and elementary matchers are *reused* not only across targets but also across kernels. For example, the `gemm` pattern is expressed as two elementary matchers as shown in Listing 20 where gemmInit detects an initialization statement while gemmCore detects the GEMM core statement. The exact same matchers are reused for other two kernels (`2mm` and `3mm`). Moreover, just changing the access matchers callbacks (hasGemmPattern) is enough to capture other patterns with the same tree structure (i.e., `syrk`). Similarly the same matcher for level-2 BLAS `GEMV` is reused for `gemver`, `gesummv`, `bicg` and `mvt`. Each matcher is assigned a priority which implements its firing policy. Matchers for level-3 BLAS are assigned the same priority, which is

```
auto isGemm/*isSyrk*/Like = [&](schedule_node node) {
  auto gemm/*syrk*/Core =
    band(and_(is3D, isPermutable, hasGemm/*hasSyrk*/Pattern), leaf());
  auto gemm/*syrk*/Init =
    band(and_(is2D, hasDescendant(gemm/*syrk*/Core), has2DInitPattern),
      anyTree());
  return isMatching(node, gemm/*syrk*/Core)
      || isMatching(node, gemm/*syrk*/Init);
};
auto is2DInit(auto _iCore, auto _jCore, auto _CCore, auto reads,
  auto writes, auto node) {
  auto has2DInitPattern = [&](schedule_node node) {
    auto _i = placeholder(), _j = placeholder();
    auto _C = arrayPlaceholder();
    auto mRead  = allOf(access(_C, _i, _j));
    auto mWrite = allOf(access(_C, _i, _j));
    return match(reads, mRead).size() == 1
        && match(writes, mWrite).size() == 1
        && _iCore == _i && _jCore == _j && _CCore == _C;
  };
}(node);
```

Listing 20: Matchers can be reused and composed together. The GEMM pattern can be expressed as a composition of two elementary matchers (gemmCore and gemmInit). Changing the access relation matcher callback hasGemmPattern with hasSyrkPattern is enough to reuse the same matchers to capture other patterns with the same tree structure (i.e., syrk). hasSyrkPattern can be easily derived as shown in Listing 13.

higher than those for level-2. Among matchers with the same priority the firing policy is decided by the user. Matchers for level-3 BLAS are assigned the same priority, which is higher than those for level-2. Therefore level-3 matchers are chosen first in case of multiple matches.

Evaluation Figure 3.6 compares the achieved GFLOP/s for double-precision operands using Loop Tactics, an original Polly version (git commit 592b2406) and the Pluto source-to-source compiler [42] (git commit f62d61b8). We compare with the out-of-the-box optimizations available in Polly and Pluto. Besides, for BLAS kernels (`gemm` to `mvt`), where Loop Tactics invokes highly-optimized routines, we use the Pluto compiler to explore different fusion heuristics and tile sizes. Specifically, for each selected benchmark, we generate several variants for each tile size and fusion heuristic available in Pluto (maximum fusion, no fusion, and smart fusion). The set of tile sizes chosen are powers of two and non-powers of two. We choose an interval from 1 up to one-fourth of the problem size, including non-powers of two sizes for every two powers of two tile sizes [66]. For each benchmark, the explored space consists of more than 3000 variants; we report the best founded (Pluto best). The optimized code generated by Pluto is lowered to binary using native compilers. We use XLC 16.1 for the Power architecture and ICC v19.0 for the Intel one. The

compilation strings are `xlc -O4 -qhot -qarch=pwr9 -qtune=auto -qsimd=auto` and `icc -O3 -march=native` for XLC and ICC, respectively.
Results for the Power 9 are shown on top; results for the Intel i7-7700 CPU are shown at the bottom. For Power 9, Loop Tactics' performance exceeds or is comparable to that of Polly and Pluto. The largest speedups are observed for `trmm` (16×) when comparing with Polly, while for `symm` (8×) when comparing with the default Pluto optimization (Pluto default). The lowest speedups are achieved for `gemm` (2×) and `gemver` (1.9×) when comparing with Polly and Pluto default, respectively. If no BLAS kernels are detected (`atax` to `seidel2d`), Loop Tactics generates code that is *on-par* with Polly. An exception is made for Jacobi stencils, where Loop Tactics is faster due to DLT optimization. On-par performance with Polly has been achieved by relying on the *isl* scheduler as Polly does and re-implementing Polly imperative tiling optimization, which is the default Polly optimization, in our declarative style approach based on matchers an builders. An example of how to implement tiling using Loop Tactics has been presented in Listing 11. When we compare Loop Tactics' results with Pluto, minor performance regression can be seen in the stencil and solver benchmarks. Exception made, once more, for the Jacobi-2d stencil thanks to the DLT transformation. For Intel i7-7700, speedups are achieved for all except two BLAS kernels in the benchmark. The largest speedups are for `symm` for both Polly and Pluto default, 14×, and 17×, respectively. The lowers are for `gemm` (1.30×) when comparing with Polly and for `gemver` (0.7×) when comparing with Pluto default. The only benchmark where Loop Tactics performs worse than Pluto and equal to Polly even by calling a BLAS function is `gesummv`. This loss in optimization opportunity is the result of additional overhead introduced by Loop Tactics to acquire the library handle dynamically. Such overhead is constant and around 1.5ms. It can be avoided by static linking[2] and if so for the `gesummv` kernel we expect to obtain the same performance as measured from the MKL library, 2.73 GFLOP/s instead of 1.65 GFLOP/s. The other benchmark where Loop Tactics produces worse performance than the tuned version generated with Pluto is `gemver`. The reason is that only half of the statements can be mapped to BLAS calls; for the remaining statements, Loop Tactics performs a simple default tiling strategy with squared tiles of 32 elements each. On the other hand, Pluto performs aggressive loop fusion, which helps in reducing the control flow overhead and tile the fused loop achieving better performance.
Finally, we evaluate the impact of Loop Tactics on compilation time. Here we compare against an original Polly version. Both Loop Tactics and Polly are compiled in Release mode. For the 25 kernels considered, the original version takes 14.88 seconds while Loop Tactics 12.00 seconds, which is a 20% reduction. The reduction is because at compilation time, only pattern detection is performed by Loop Tactics while optimizations happen at run-time by linking with BLAS libraries.
Figure 3.7 (left) demonstrates how Loop Tactics improves performance over Polly by using *the same matchers* as CPU to automatically detect and call cuBLAS on an Nvidia Volta V100 GPU. We only illustrate the benchmarks where Loop Tac-

[2]We opt for dynamic linking as it is usually the common way to interact with BLAS libraries as it ensures binary portability [67]. Nevertheless, there is no restriction in Loop Tactics that will prevent us from switching to a static linking in the future, if needed.

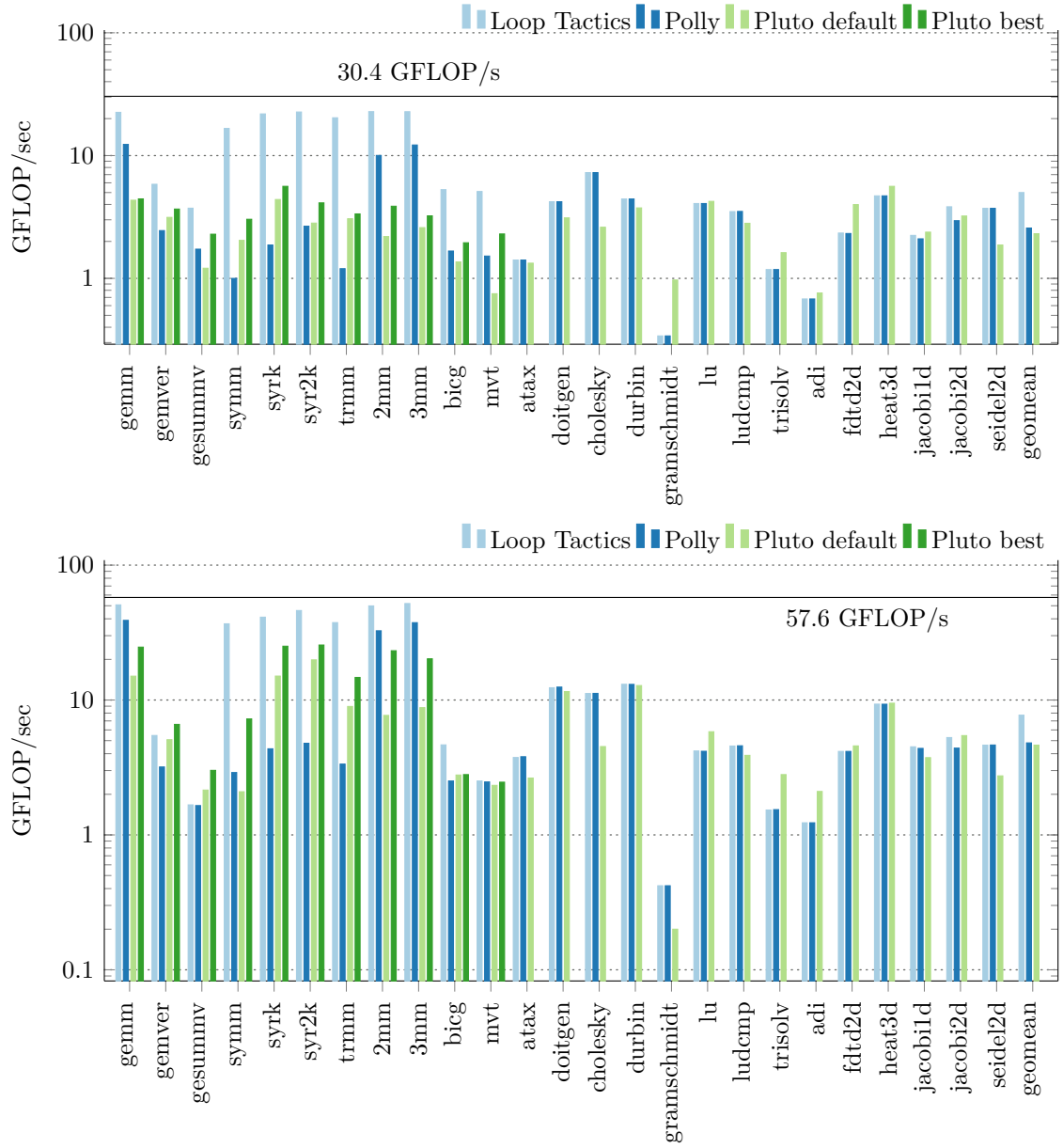


Figure 3.6: Loop Tactics outperforms Polly and it is comparable to Pluto for double-precision operands (average of 5 runs) by matching library calls: (top) calling IBM ESSL on Power 9 with a theoretical peak of 30.4 GFLOP/s; (bottom) calling MKL on Intel i7-7700 CPU with a theoretical peak of 57.6 GFLOP/s.

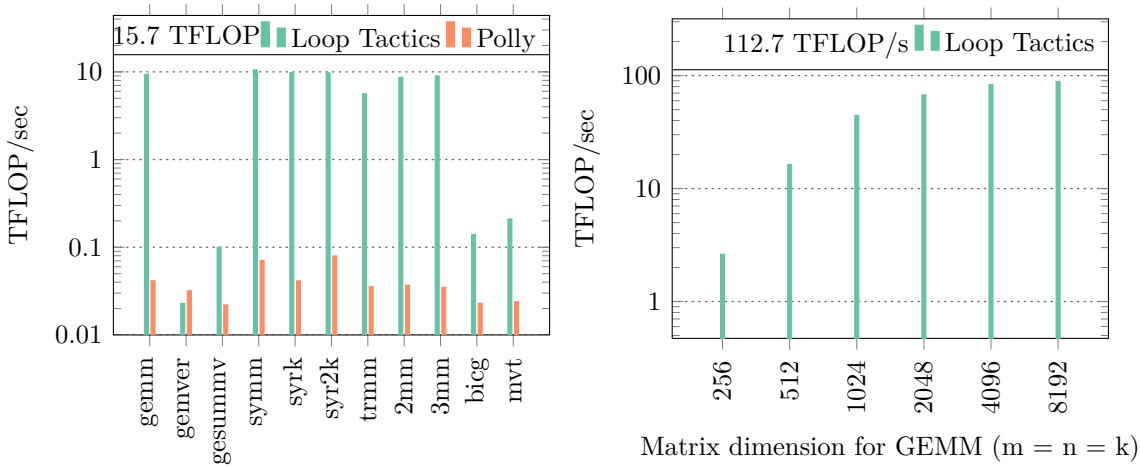


Figure 3.7: Performance achieved for single-precision operands, average of five runs; (left) Loop Tactics produces speedups over original Polly by calling CuBLAS on a Volta V100 GPU with a single-precision theoretical peak of 15.7 TFLOP/s; (right) if lower-precision is acceptable during computation (i.e., 16-bit) Loop Tactics enables the user to transparently exploit tensor cores for GEMM.

tics detect BLAS kernels as in the other case, we fall-back to Polly. The largest speedup is achieved for `symm` (148×). A minor performance regression is obtained by calling the level-2 BLAS `GEMV` in `gemver`. Figure 3.7 (right) further illustrates the performance achieved using GPU tensor cores, which are usable through vendor libraries and currently inaccessible to Polly. For matrices of size 8192, we achieve 88.8 TFLOP/s, 79% of the theoretical peak of 112.7 TFLOP/s. Manually using tensor cores requires the programmer to guarantee the following conditions: (1) only `GEMM` pattern supports tensor cores execution; (2) the matrix dimensions should be multiple of eight; (3) math mode in cuBLAS should be set to `CUBLAS_TENSOR_OP_MATH`; (4) input data type should fit in half-precision floating point. Loop Tactics hides all this complexity and requires the user to only check (4), and whenever possible Loop Tactics will use tensor cores. Specifically, condition (1) is ensured by the matcher (Listing 20) and condition (3) is implemented by Loop Tactics' runtime library. If (2) does not hold, Loop Tactics will automatically fall back to not using tensor cores.

3.4 Loop Tactics Syntax

The syntax using the extended Backus-Naur form for matchers and builders is shown in Figure 3.9. Matchers and builders are designed to replicate the node type-based structure of the schedule tree in a declarative way. Matchers take a callback and/or a reference to an schedule_node object as optional leading arguments. A callback allows fine-grained control over the matching procedure, whereas a reference will point to the matched node. Builders take either the node kind-specific parameter as their leading argument, or a callback that can produce them. Both matchers and builders take a call to another matcher or builder as remaining arguments, respectively. An exception is made for "leaf" and "anyTree". Matchers and builders walk the schedule

node type	param	node type	param
context, guard	set	expansion, extension	union_map
filter, domain	union_set	set, sequence	void
mark	id	band	multi_union_pw_aff

Table 3.1: Node types with different input parameters.

⟨*matcher*⟩ ::= ⟨*matcher-type*⟩(⟨*matcher-list*⟩)
| ⟨*matcher-type*⟩([⟨*capture*⟩],[⟨*callback*⟩],⟨*matcher-list*⟩)
| leaf() | anyTree()
⟨*node-type*⟩ ::= 'band' | 'context' | 'domain' | 'expansion'
| 'extension' | 'filter' | 'guard' | 'leaf' | 'mark' | 'set'
| 'sequence'
⟨*matcher-type*⟩ ::= 'anyTree' | ⟨*node-type*⟩
⟨*matcher-list*⟩ ::= ⟨*matcher*⟩[{',' ⟨*matcher*⟩}]
⟨*capture*⟩ ::= /*ref to schedule_node var*/
⟨*callback*⟩ ::= /*callable obj bool(schedule_node var)*/
⟨*builder*⟩ ::= ⟨*node-type*⟩(⟨*param*⟩,⟨*builder-list*⟩)
| ⟨*node-type*⟩(⟨*callback*⟩,⟨*builder-list*⟩)
| 'subtree'(capture)
⟨*builder-list*⟩ ::= ⟨*builder*⟩ {',' ⟨*builder*⟩}
⟨*param*⟩ ::= /* see Table 3.1 */
⟨*bool*⟩ ::= isMatching(node, ⟨*matcher*⟩)
⟨*node*⟩ ::= replaceBFS(node, ⟨*matcher*⟩, ⟨*builder*⟩)
⟨*node*⟩ ::= replaceDFSPreorder(node, ⟨*matcher*⟩, ⟨*builder*⟩)
⟨*node*⟩ ::= replaceDFSPostorder(node, ⟨*matcher*⟩, ⟨*builder*⟩)

Figure 3.8: Extended Backus–Naur form for matchers and builders.

tree using replaceDFSPreorder | replaceDFSPostorder | replaceBFS. The traversal functions take three arguments: a node from where the traversal should start, a matcher to verify whether the subtree rooted at the current node matches, and a builder that rebuilds the subtree in case of a match. We allow multiple matches. Matchers can use "AnyTree" to capture any subtree, whereas builders can use "subtree" to rebuild any subtree. The syntax for defining an access relation matcher is a function call that takes an optional array placeholder, followed by a list of dimension placeholders. The position of the latter is specified by a "dim" call or is inferred from their position in the argument list, see Figure 3.9. A union of relations can be matched against a list of matchers using "match" and "allOf" or "anyOf" as combiners. "findAndReplace" finds and replaces a given pattern in a union of relations. The candidate pattern and the replacement are built via the "replace" function call.

⟨*isl::union_map*⟩ ::= findAndReplace(union_map,
⟨*replacement*⟩)
⟨*matches*⟩ ::= match(union_map, ⟨*access-matcher*⟩)
⟨*access-matcher*⟩ ::= allOf(⟨*access*⟩ {',' ⟨*access*⟩}
| ⟨*access-matcher*⟩ ::= anyOf(⟨*access*⟩ {',' ⟨*access*⟩}
⟨*access*⟩ ::= access(⟨*args*⟩)
⟨*args*⟩ ::= any()
| (⟨*array-placeholder*⟩,⟨*positioned-arg-list*⟩)
| (⟨*positioned-arg-list*⟩)
⟨*positioned-arg-list*⟩ ::= dim-arg
| 'dim'('integer', ⟨*dim-arg*⟩)
⟨*dim-arg*⟩ ::= placeholder | stride
⟨*array-placeholder*⟩ ::= /*res of calling arrayPlaceholder()*/
⟨*placeholder*⟩ ::= /*res of calling placeholder()*/
⟨*stride*⟩ := stride('integer')
⟨*replacement*⟩ := replace(⟨*access*⟩, ⟨*access*⟩)

Figure 3.9: Extended Backus–Naur form for access matchers.

3.5 LoopOpt - Interactive Code Optimization

Loop Tactics comes with a graphical user interface (LoopOpt), enabling easy usage of matchers and builders. LoopOpt eases the steep learning curve polyhedral model's internal representations and provides the user with an easy mechanism to derive matchers and builders.
Figure 3.10 shows the building blocks of LoopOpt and the interaction point with the users. Let us briefly describe the internal machinery of LoopOpt. To start, the user interacts only with the GUI (see Section 3.5.1). Once the user opens a given application, we use *pet* as our raising tool to extract and model a SCoP in the polyhedral model. From the SCoP, we obtain the schedule tree, and we feed it to Loop Tactics. Loop Tactics lowers each directive (Section 3.5.2) provided by the user to schedule tree matchers and builders, and applies them to the extracted tree. As a result, we obtain an optimized schedule that we send back to *isl* for code generation. At the end of this process, the user will visualize how her directives optimized the original application. While spelling the transformation recipe, the user can ask for performance feedback. We provide two kinds of feedback: How the memory system behaves for the current schedule and how the current schedule performs. The former is obtained by running Haystack, a fast cache emulator, which provides number of cache misses in the millisecond's range, thus providing cache-aware optimizations [68]. The latter is obtained by code generating the current schedule and timing its execution.

3.5.1 Graphical User Interface

Figure 3.11 shows our GUI. It combines three main views: ① is an editable window where the user can specify the tactic to apply to a given computational pattern in a given program. A new program can be open via File → Open. ② is a read-only

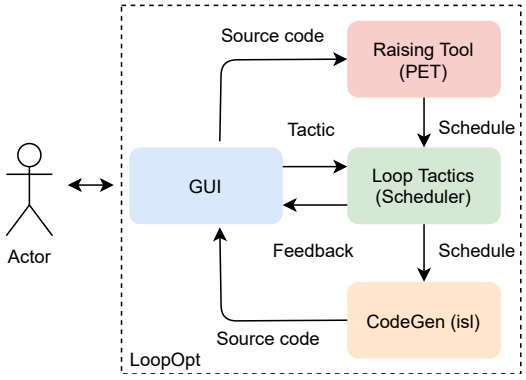


Figure 3.10: LoopOpt basic blocks and interaction points with the user. LoopOpt enables the specification of custom recipes for given computational motifs. The user interacts with the GUI only by entering a given transformation recipe (or tactic). The system returns immediate feedback and generates the transformed code on-demand.

window showing the live-updated code. The code will be updated by applying the specific transformation for each directive inserted by the user. ③ is a read-only window that provides feedback to the user (i.e., after running the Haystack cache emulator using the `runCachEmul` directive). The interface also provides a menu bar located on the top of the main window, with items for file operations.

3.5.2 Specification Language

A transformation recipe in our tool is called a "tactic" and consists of two parts: the specification of a pattern and a set of rewriting rules that describe how the pattern should be optimized. The pattern description is enclosed in the `pattern` directive, and it is spelt using Tensor Comprehensions (TC) notation. Figure 3.12 shows the EBNF notation for the `pattern` directive. To be concrete, let us assume that we want to detect a GEMM statement as the one in Listing 1. To do so, the user will write the following TC expression as pattern directive:

```
C(i,j) += A(i,k) * B(k,j)
```

The expression will get lowered to the tree, and access matchers reported in Listing 21. The tree matcher looks for a `band` node which satisfies two properties expressed as callback functions: `hasDimensionalty` and `hasPattern`. The former, ensures structural properties, looking for a band node containing three schedule dimensions which corresponds to a triple nested loop. The latter, guarantees access pattern properties, making sure that they equal the one of a GEMM pattern. The structural properties to be matched (i.e., the number of bands) is determined from the number of induction variables—three, in this case, i, j and k, while access pattern properties can be easily derived from the tensor specifications obtained from TC syntax (i.e., access to three different 2-d tensors). Once the pattern keyword has been specified, LoopOpt instantiates the matchers and runs them on the program. If any match happens, LoopOpt annotates the surrounding loops with (unique) tags that can be later referred by the rewriting rules.
How the pattern should be optimized is specified as a composition of (instances of)

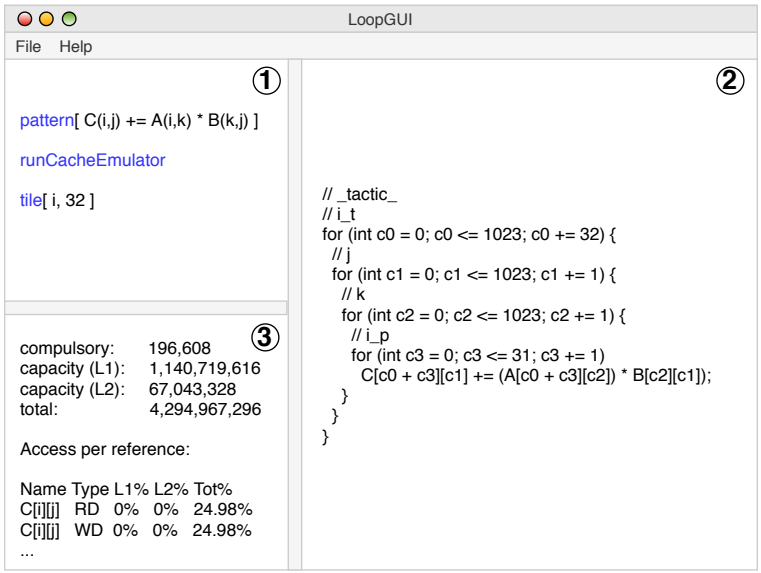


Figure 3.11: Loop Tactics interface with live-update and synchronized views. (1) Editable view to specify the transformation tactic; (2) live-update code (3) code editor switching between original code and user-provided feedback.

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

Figure 3.12: Simplified EBNF syntax for `pattern` keyword. Brackets denote optional clauses, curly brakets denote repetitions, and square brackets contain textual description.

```
auto hasDimensionality = [&](schedule_node node) {
  // check for a 3 nested loop.
  return node.schedule().dim == 3
};
auto hasPattern = [&](schedule_node node) {
  // check GEMM access patterns, same as `hasGemmPattern`.
};
auto matcher =
  band(_and(hasDimensionality, hasPattern));
```

Listing 21: Generated tree and access relation matchers for a GEMM statement C(i, j) += A(i, k) * B(k, j).

Directive	Short description
`reverse`(loopTAG)	Reverse iteration order
`unroll`(loopTAG, factor)	unroll loop by factor
`tile`(loopTAG, factor)	tile loop by factor, and sink point loop
`interchange`(loopTAG, loopTAG)	interchange loops
`runCacheEmul`	run cache emulator on current schedule
`timeSchedule`	generate code for current schedule and time it

Table 3.2: Directives exposed by LoopOpt.

primitive, building-block operations. Currently, LoopOpt supports basic operations, which allow affine transformations on loop nests. Table 3.2 shows the operations supported, among with the directives the user can specify to obtain feedback on transformation profitability.
All the supported transformations work on loops. Each of them must specify the loop it applies to via the `loopTAG`. `tile` and `unroll` take an additional parameter which specifies the unrolling factor or the tile size of the targeted loop, respectively. For performance feedback, the user has two choices: `runCacheEmul` and `timeSchedule`. The former allows the user to run the Haystack cache emulator on the current schedule, which reflects the transformations applied so far. As output, the cache emulator provides the absolute number of misses for the L1 and L2 level-cache, and the percentage of cache misses for a given array reference, this latter information can be used to understand transformation profitability. The second option the user has is the `timeSchedule` directive. `timeSchedule` generates code from the current schedule, inserting proper initialization statements, and times its execution by running the code on the user machine.

3.6 Related Work

We can broadly classify optimizers under two main umbrellas: optimizers with automatic scheduling and optimizers with scheduling languages. The former uses cost functions or rewriting rules to derive the optimal schedule; the user can affect the optimizer decisions by few exposed command-line flags. The latter leaves complete control to the users by providing a scheduling language.

Optimizers with automatic scheduling. Years of research in automatic compilation lead to sophisticated general-purpose optimizers such as Pluto [42], Polly [22] and, GRAPHITE [21]. Despite being fully automatic, therefore increasing productivity, general-purpose optimizers don't always obtain the best performance for commonly recurrent kernels. Suboptimal performance is in most cases, the result of a generic one-size-fits-all optimization strategy. Loop tactics enables such optimizers to detect computational patterns easily and hook a custom optimization for each of them. Custom optimizations can be expressed as composable transformations or call to optimized libraries. As most of these optimizers already power production compilers such as GCC and LLVM we also expect general-purpose compilers will benefit from Loop Tactics.

Optimizers with scheduling languages. Kelly et al. proposed the first framework for high-level transformations based on the polyhedral model [69]. Girbal et al. proposed the URUK framework to apply loop transformations for cache hierarchy and parallelism using unimodular schedules [70]. Yuki et al. developed AlphaZ, a framework to express programs as a set of equations based on the Alpha language and manipulated it using script-driven transformations [71]. The implementation also supports memory (re)-allocation and explicitly represents reductions. Similarly, Yi et al. presented POET, an interpreted program transformation language designed for applying and exploring complex code transformations in different programming languages [72]. Donadio et al. introduced Xlanguage, an embedded DSL based on C/C++ pragmas that allows users to generate multi-version programs by specifying the type of transformations to apply as well as the transformation parameters [73]. Bagnères et al. provided feedback from a polyhedral compiler by expressing it as a sequence of loop transformation directives [50]. Their input language, Clay, and the related Chlore algorithm allow users to examine, refine or freeze sequences of loop transformation directives. Chen et al. introduced CHiLL, a high-level transformation and parallelization framework that uses a model-driven empirical optimization engine to generate and evaluate different code variants [74, 75]. Khan et al. built a meta-optimizer on top of the CHiLL compiler to automatically generate transformation recipes and perform extensive autotuning [76]. Teixeira et al. proposed a language to express the collection of transformations that can be applied to user-defined code regions [77]. Baghdadi et al. proposed Tiramisu a polyhedral compiler which introduces novel commands to explicitly target distributed systems. More recently, Kruse et al. submitted a proposal to improve pragma-based transformations in Clang [78]. All these tools expose some scheduling-based language to specify program transformations applicable to loops. But the scheduling language can be seen as *imperative* as it requires the user or the autotuner to specify which loops are targeted using external tags or language-level annotations. Our approach, on the other hand, is based on a declarative language. Instead of binding the transformation to a specific loop, we *declare* how a compatible schedule tree should look like and express the transformation in terms of matched nodes and relations between them. Another difference with respect to fully automatic compilers such as Halide [63] and TVM [64] is that Loop Tactics aims at bringing domain-specific optimizations *directly* in general-purpose compilers, while Halide and TVM born as domain-specific. Moreover, both Halide and TVM uses intervals to represents iteration spaces. Thus, non-rectangular iterations spaces cannot be naturally represented, which is not the case for Loop Tactics.

Optimizers with rewriting rules Algorithm recognition is a well-known problem in computer science and was an ongoing and extensive research topic during the 1990s [79, 80, 81, 82]. Although the approach details vary considerably for their application domain, the underlying goal always falls under the umbrella of program optimization. Previous works can be broadly classified accordingly to the level of abstraction used to match a set of predefined constructs: text, syntactic, and semantic level [83]. Text-level tools operate on the source code of the program directly. Syntactic ones work at the abstract syntax tree level (AST), while semantic ones go

on step further by annotating the AST with data and control flow information.

Text-level and syntactic-level methods: Pottenger et al. implemented a pattern-recognition tool on top of the Polaris compiler [84] to recognize and parallelize reduction operations [85]. Their method directly matches code statements with a set of predefined patterns. It then uses a data-dependence analysis to prove that all loop iterations refer to different elements of the reduced array. Sarvestani et al. introduced an idiom-detection tool for kernel recognition in DSP applications based on the Cetus compiler [86, 87]. Each detected kernel is replaced with a highly optimized parallel version. Patterns are described in XML format. Kessler et al. developed PARAMAT to detect and replace code segments at the AST-level with calls to parallel library routines [80]. For FORTRAN code, Di Martino et al. designed the PAP recognizer with the main focus of computer-aided program analysis and parallelization [82]. The tools provide a graphical user interface that shows each detected pattern and its corresponding source-line location. Bravenboer et al. proposed Stratego a language for program transformations using rewriting rules in the context of generative programming (i.e., from code refactoring to code optimization) [88]. More recently, Clang AST matchers have been developed as a domain-specific language for matching predicates on Clang AST. Contrary to our work, however, these approaches try to match code directly at the statement or AST level. Hence, variations in the programming style have a huge impact on the effectiveness of such tools. On the other hand, Loop Tactics is embedded into Polly, which runs in the latest stage of the LLVM pipeline after inlining, constant expression propagation, and dead code elimination. Consequently, by position, it is less strongly affected by programming style. Furthermore, Polly protects against the linearization of multi-dimensional arrays because it can recover the 2D structure [89]. Finally, most of the tools mentioned above provide only pattern recognition and rely on directives or library calls to improve performance. On the other hand, Loop Tactics provides builders that can be used to specify a specific transformation "recipe" for each detected pattern.

Semantic-level methods: The XARK compiler—perhaps the most representative example among semantic-level methods—provides code recognition based on the analysis of Strongly Connected Components (SCCs) [90]. SCCs components are analyzed by looking at use-def chains in the Gated Single Assignment (GSA) form, an extension of the SSA form. The detection of a given kernel is carried out in two phases. In the first, the use-def chains are used to recognize basic statements such as conditional linear induction variables and array assignments. In the second phase, the use-def chains between statements are analyzed to identify more complex computation kernels (i.e., consecutively written array and masked operations). Compared with Loop Tactics the XARK compiler, or an SSA-based tool in general, has the advantage of being able to detect irregular access patterns such as linked-list traversal. Detection of such patterns falls beyond the capabilities of the polyhedral model and hence of Loop Tactics. But SSA-based tools rely on the use of use-def chain on scalar registers to reconstruct the access patterns, fundamentally limiting the complexity of the access patterns that such tools can recognize. On the other hand, Loop Tactics uses access relations [49, 50], that *precisely* and *explicitly* encode the relation between the iteration space and the array space. Therefore Loop Tac-

tics can easily and concisely express complex access patterns on multi-dimensional arrays and inspect all sorts of possible index permutations (i.e., transposed accesses as shown in Listing 12). Other works at the semantic-level focus on specific families of patterns that can be covered by Loop Tactics. Suganuma et al. focused on reduction detection and parallelization [91]. Their algorithm inspects strongly connected components with cyclic dependences typical of reduction operations. It then tries to recognize if the statement is actually a reduction by finding a reduction operator and a reduction function. Gerlek et al. developed a technique to detect classes of induction variables based on the use-def chain available in the SSA form [92]. Pinter et al. proposed a method for extracting parallelizable idioms from scientific applications [93]. Their technique uses the computational graph, which encodes the flow of data and the dependencies among values in the program. Recognition of computational idioms is carried out by matching specific patterns. Matchers and replacements rules are described using the graph specific grammar, similar to what Loop Tactics does at the schedule tree level. However, their technique requires some preprocessing steps such as loop distribution and unrolling to expose the kernels in the computation graph. Such preprocessing steps are not needed in Loop Tactics. Sujeeth et al. in their DELITE framework use rewriting rules to perform domain-specific optimizations [8]. Despite our approach is similar to what they propose our end goal is not. With Loop Tactics we aim at reducing the need of developing DSL compilers by bringing domain-specific optimizations *directly* in general-purpose compilers. Delite, on the other hand, aims at lowering the effort of developing DSLs.

3.7 Summary and Conclusion

We introduced Declarative Loop Tactics or Loop Tactics for short, and its implementation as a framework to express computational patterns and transformations in a declarative way on schedule trees, an internal representation of a polyhedral compiler. We showed how Loop Tactics allows one to express domain-specific optimizations directly in general-purpose compilers as a set of composable transformations or as calls to optimized libraries. As a result, (1) we reduce the need to design domain-specific compilers, (2) we allow the transparent use of vendor-optimized libraries, and (3) unlock the effective optimization of multi-domain applications. Compared with previous arts Loop Tactics enables us to avoid binding the transformations to a specific loop but allows us to declare how a compatible schedule tree should look like and express the transformations in terms of matched nodes and relation between them.
We expect our approach to significantly extend the range of optimizations compared to what current general-purpose compilers can achieve by allowing developers to plugin highly customized optimizations for a given computational pattern.

4

Abstraction Raising in a Multi-level IR Compiler

With *Progressive Raising*, a complementary approach to the progressive lowering in multi-level IRs, that raises from lower to higher-level abstractions we enable leveraging domain-specific transformations for low-level representations. To implement progressive raising, we introduce *Multi-level Tactics*, our declarative approach implemented on top of the MLIR framework. We demonstrate the progressive raising from affine loop nests specified in a general-purpose language to high-level linear algebra operations.

4.1 Multi-level Tactics

Multi-level intermediate representations (IR) show great promise for lowering the design costs for domain-specific compilers by providing a reusable, extensible, and non-opinionated framework for expressing domain-specific and high-level abstractions directly in the IR. But, while such frameworks support the progressive lowering of high-level representations to low-level IR, they do not raise in the opposite direction. Thus, the entry point into the compilation pipeline defines the highest level of abstraction for all subsequent transformations, limiting the set of applicable optimizations, in particular for general-purpose languages that are not semantically rich enough to model the required abstractions.
The key idea of Multi-level Tactics is to allow for transformations from lower to higher levels of abstractions. By providing the infrastructure for raising from common abstractions of general-purpose languages to specialized high-level dialects, Multi-level Tactics enables domain-specific optimizations for general-purpose code in multi-level IR compilers.
Figure 4.1 illustrates the overall flow for transformations using Multi-level Tactics. The steps at the top allow tactics specification and the generation of the code working on the actual IR (orange box). In contrast, the bottom steps represent the transformations of the input program (blue box).
All tactics are described in a high-level, declarative specification *Tactics Description Language (TDL)*. TDL enables to specify the computational pattern to which the tactic applies and a set of replacement expressions. The TDL DSL frontend processes the high-level declarative specification and emits a TableGen entry called

This Chapter is based on: L. Chelini et al., "Progressive Raising in Multi-level IR" IEEE/ACM International Symposium on Code Generation and Optimization (CGO), 2021.

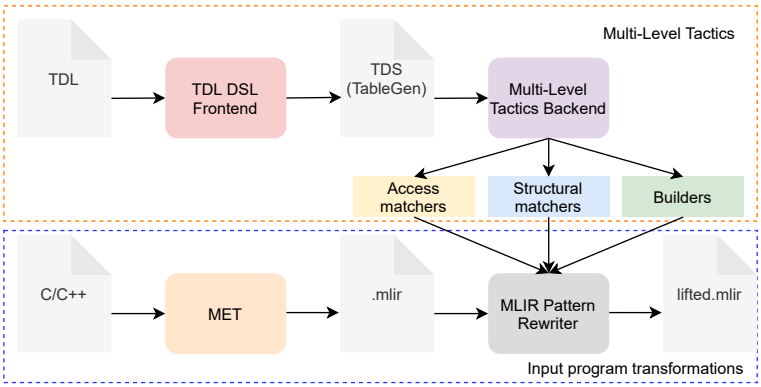


Figure 4.1: Multi-level Tactics compilation flow (orange box) and transformations of the input program (blue box).

Tactics Description Specification (TDS). Sections 4.1.1 and 4.1.2 provide a detailed description of these two formats. The actual code for the matching of IR operations and memory accesses, as well as for building replacement operations, is generated by the Multi-level Tactics backend and uses the MLIR pattern rewriting infrastructure upon execution (more details in Section 4.1.3).
Transformations on the input program start by translating C code to Affine using the MLIR Extraction Tool (PET-to-MLIR or MET). MET enables entering the MLIR pipeline at the Affine level from C code for the subset of the language within the polyhedral model. During translation, the C code is canonicalized by distributing loops to simplify pattern recognition. The Affine representation is then fed into the MLIR pattern rewriter engine, where the generated tactics have been hooked. The output of the entire flow is a lifted MLIR where loop nests have been raised to high-level operations. [1]

4.1.1 Tactics Description Language - TDL

The key user-facing component of Multi-level Tactics is the Tactics Description Language (TDL). Pattern and replacements in TDL are specified in a syntax borrowed from Tensor Comprehensions, which is itself a slight variation of the ubiquitous Einstein tensor index notation [94]. As a concrete example, we illustrate the syntax for the pattern and replacement for a transformation lifting a sequential, loop-based implementation of a contraction of the form abc-acd-bd (Listing 22). We raise the contraction by rewriting it with an equivalent Transpose-Transpose-GEMM-Transpose (TTGT) expression taking advantage of highly efficient GEMM implementations offered by vendor-optimized libraries. The TTGT computation first flattens the tensors into matrices via explicit tensor transpositions and reshape operations, then executes GEMM, and finally folds back the resulting matrix into the original tensor layout.
Listing 23 shows the user-defined tactic to detect and optimize the contraction. The signature of the tactic is composed of the keyword `def`, followed by a name (TTGT). The body consists of two parts delimited by the keywords `pattern` and

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

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

Listing 22: Contraction abc-acd-db.

```
1 def TTGT {
2     pattern
3     C(a,b,c) += A(a,c,d) * B(d,b)
4     builder
5     D(f,b) = C(a,b,c)    where f = a * c
6     E(f,d) = A(a,c,d)    where f = a * c
7     D(f,b) += E(f,d) * B(d,b)
8     C(a,b,c) = D(f,b)    where f = a * c
9 }
```

Listing 23: TDL to match a contraction abc-acd-db and apply the TTGT optimization on each detected pattern.

`builder`: `pattern` describes the computational motif to be detected in the user code, whereas `builder` describes the transformation recipe. For example, Lines 5 and 6 reshape the 3D tensors C and A into matrices. Additionally, a transposition $(a, b, c) \rightarrow (a, c, b)$ is performed for C before reshaping. Line 7 specifies the GEMM operation, while Line 8 folds back the result into the original tensor layout, again emitting an implicit transposition $(a, c, b) \rightarrow (a, b, c)$ after folding.

4.1.2 Tactics Description Specification - TDS

The role of the TDL DSL frontend is to generate the TableGen-based Tactics Description Specification (TDS). Each TDS file consists of a series of instantiations of TableGen templates, from which C++ code is generated at compile time. The choice of a two-step process instead of direct code generation from TDL allows common routines for pattern matching and infrastructure to be factored as reusable templates in TDS and thus reduces complexity.
Listing 24 shows the generated TDS definition for the TTGT tactic from Listing 23. Each TDS entry derives from a base class `Tactic`, which allows defining the pattern and a list of builders. The pattern specification matches the `pattern` entry in the TDL tactic. The list of builders, on the other hand, corresponds to a set of high-level operations with a one-to-one mapping with operations exposed by high-level dialects (i.e., Linalg). For example, the `transposeBuilder` will create a Linalg `TransposeOp` or a function call to a transpose BLAS routine, depending on the lowering path selected by the user. Considering our running example, line 5 in Listing 23 will map to a transpose operation followed by a reshape one (lines 2 and 3 in Listing 24). Similarly, lines 6 and 7 will map to a reshape and a matmul operation, respectively. Finally, line 8 (Listing 23) will map to a reshape operation

```
1 def TTGT : Tactic<C(a, b, c) += A(a, c, d) * B(d, b), [
2   transposeBuilder<In<[C]>, Out<[D]>, Expr<{0, 2, 1}>>,
3   reshapeBuilder<In<[D]>, Out<[E]>, Expr<{{0, 1}, 2}>>,
4   reshapeBuilder<In<[A]>, Out<[F]>, Expr<{{0, 1}, 2}>>,
5   matmulBuilder<In<[F, B]>, Out<[E]>>,
6   reshapeBuilder<In<[E]>, Out<[D]>, Expr<{{0, 1}, 2}>>,
7   transposeBuilder<In<[D]>, Out<[C]>, Expr<{0, 2, 1}>>,
8 ]>;
```

Listing 24: TDS to match a contraction abc-acd-db and apply the TTGT optimization on each detected pattern.

followed by a transpose one (lines 6 and 7 in Listing 24). Ultimately, each entry in TableGen gets lowered to C++ matchers and builders via Multi-level Tactic's TableGen backend at compile-time, as we will see in the next section.

4.1.3 Matchers and Builders

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

Structural and Operation Matchers

The role of a structural matcher is to detect control flow patterns in the IR. It essentially replicates the control flow-based structure of the IR with additional filtering capabilities. A structural matcher consists of a control flow operation type, a list of children operations, and an optional filtering function. For example, Listing 25, matches all the IR subtrees, consisting of a two-dimensional, perfectly nested loop, where the innermost loop body contains a Multiply and Accumulate (MAC) operation, verified via the callback function `isMAC`. The top operation, referred to as a relative root, defines a structural matcher. The matching operation starts at a specified operation in the IR. It then recursively walk the operation's descendant as well as the relative root matcher descendants. If a mismatch is encountered during the traversal, the procedure stops, and the failure is reported immediately. The API to construct structural matchers is designed to resemble the structure of the IR itself visually. Leading arguments include optionally a callback function, which allows the caller to control the matching more precisely. For example, identify a MAC operation in the loop body requires checking a non-structural property. Each matcher must belong to a context which handles memory allocation and ownership. Operation matchers, on the other hand, verify the types of arithmetic operations. We rely on the m_Op matcher, which carries the type of operation to be matched. m_Op can be chained to detect sequences of operations. In our running example, `MACOp` looks for an Add operation with two arguments, where the second one is a Mul operation. The arguments of each operations are captured for later inspection.

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

Listing 25: Structural matchers declaratively describe the control-flow structure of the IR.

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

Listing 26: Declarative access pattern matcher.

Access matchers

Complementary to structural matcher, Multi-level Tactics provides access pattern matchers. The access matchers rely on the idea of "placeholders", modeled as `m_Placeholder` and `m_ArrayPlaceholder`. The former can match any induction dimension of the form $k * \iota + c$, where k and c are coefficients forming the pattern, whereas ι defines a candidate by matching the underneath `mlir::Value` representing the induction variable. In contrast to this, `m_ArrayPlaceholder` can only match tensor accesses and takes a list of `m_Placeholder` as inputs. In both cases, a match is an assignment of a candidate to the placeholder. Candidates assigned to different placeholders are required to be distinct, while multiple references to the same placeholder within a matcher expression must refer to the same candidate. Placeholders can be combined in *placeholder expressions* (e.g., _C{_i, _j) and can be used in the `m_Op` matcher which allows for the specification of a particular type of operation (i.e., StoreOp or LoadOp). The matching procedure starts by inspecting the last store instruction within an MLIR block–an ordered list of operations without control flow. It then walks backwards following the use-def chain. If for one or more placeholders no candidates can be found during the backward traversal, the absence of a match is reported immediately, and the procedure stops. The programming interface is similar in spirit to that of structural matchers providing a declarative way of specifying access patterns. Similarly, each placeholder must belong to a context that orchestrates the matching procedure, handles memory allocation and ownership. Listing 26 shows how the caller can identify a load from a 2D array.

Builders

The replacement for the matched patterns of a transformation is generated by the builders, which instantiate IR operations either directly (e.g., when generating calls to vendor-optimized libraries), or using already existing infrastructure from the MLIR ecosystem (e.g., EDSC builders [95]).

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

Listing 27: Structural and access matchers emitted by Multi-level Tactic's backend for the tactic defined in Listing 23.

Going back to our running example, Line 1 in Listing 27 shows the structural matchers emitted for our contraction. These match all 4-dimensional loop nests for which evaluation of `access_callback`, invoking the generated access matchers in Lines 5 to 30, evaluates to true. In particular, the access matchers look for an MLIR block which contains exactly three read accesses to different tensors: one write access, an index permutation that satisfies the placeholder pattern $[a, b, c] \rightarrow [a, c, d][d, b]$, and two arithmetic operations $(+/*)$ which define the computation of the contraction.

4.2 Results

In this section, we illustrate our framework's applicability for two raising paths, Affine-to-Affine and Affine-to-Linalg. In the former, we show how Multi-level Tactics is capable of lifting the level of abstraction within the Affine dialect and improve the

Table 4.1: Multi-level Tactics experimental setup.

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

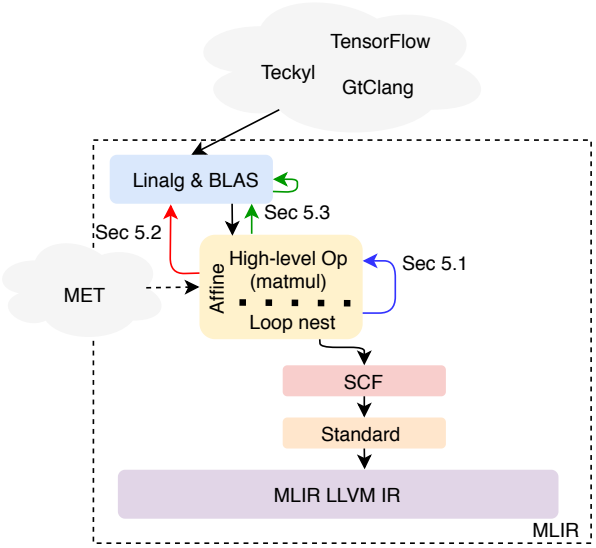


Figure 4.2: Multi-level Tactics raising paths.

performance of computations involving generalized matrix-matrix multiplication. In the latter, Multi-level Tactics is used to raise loop nests in the Affine dialect to high-level operations from the Linalg dialect. At the Linalg level, Multi-level Tactics emits Blas calls or relies on the Linalg code generator path.
Finally, we show how Multi-level Tactics leverages further high-level optimizations detecting and subsequently optimizing matrix chain multiplications. Figure 4.2 re-proposes (a simplified) MLIR lowering pipeline extended with the three raising paths enabled by Multi-level Tactics.
The experiments (Table 4.1) have been conducted on two test platforms: an Intel Core i9-9900K (Coffee Lake) and an AMD Threadripper 2920X. All results were obtained considering the minimal execution time of five independent runs for single-precision operands.

4.2.1 Raising Affine Loops to Affine High-Level Operations

Generalized matrix-multiplication (GEMM) is a frequently occurring pattern with decades of research on its optimization for various architectures [96]. A recent improvement within MLIR [97] introduced a custom high-level operation `matmul` in the Affine dialect that lowers to high-performance code by implementing the OpenBLAS/Blis optimization [98].
Currently, the operation needs to be instantiated directly in manual code generation or through modification of an existing high-level frontend, such as the Teckyl frontend for tensor computations [99]. However, while this may be conceivable for spe-

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

Listing 28: TDL to raise to a matmul. The user can raise by specifing `-raise-affine-to-affine`.

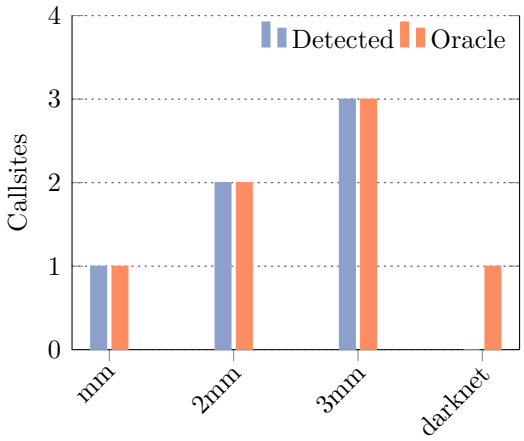


Figure 4.3: Number of callsites detected by Multi-level Tactics compared to perfect matching (Oracle).

cialized tools and specific use cases, the implementation requires detailed knowledge of MLIR internals, is time-consuming, and needs to be repeated for each high-level operation and all IR entry points.

Multi-level Tactics solves these issues by providing a convenient way for the user to express high-level patterns, such as GEMM, and to automatically locate and replace these at the affine level. Listing 28 shows the user-defined tactics necessary to detect a contraction E = $(\times, +)$ of the form $C \rightarrow C(i,j) + A(i,k) \times B(k,j)$, where A, B, and C are matrices. Once defined, the tactic is applied to all loop nests with a GEMM-like access pattern and replaces them with the `matmul` operation.

We test the reliability of our tactic on semantically equivalent GEMM kernels from Polybench 4.2 and Darknet written C in different styles. Polybench uses multi-dimensional arrays references to encode multi-dimensions array accesses. Contrary, Darknet—a widely used, open-source deep learning framework—uses linearized array references [100]. In all cases, before running Multi-level Tactics, we perform loop distribution via MET to isolate the GEMM kernels from other statements. Figure 4.3 shows the number of callsites detected by Multi-level Tactics compared with an Oracle, representing a perfect matching. The GEMM kernels from Darknet are missed, since the linearized, 1-d accesses are not matched by the 2-d array references emitted by the GEMM tactic in Listing 28. A delinearization pass in MLIR, as done in the LLVM polyhedral optimizer [101], can solve this issue.

As the quality of the OpenBLIS/BLAS transformation has already been demonstrated in the original paper, we only report the result of detecting a single 2088x2048 SGEMM multiplication on the AMD system. Compilation of a sequential loop nests

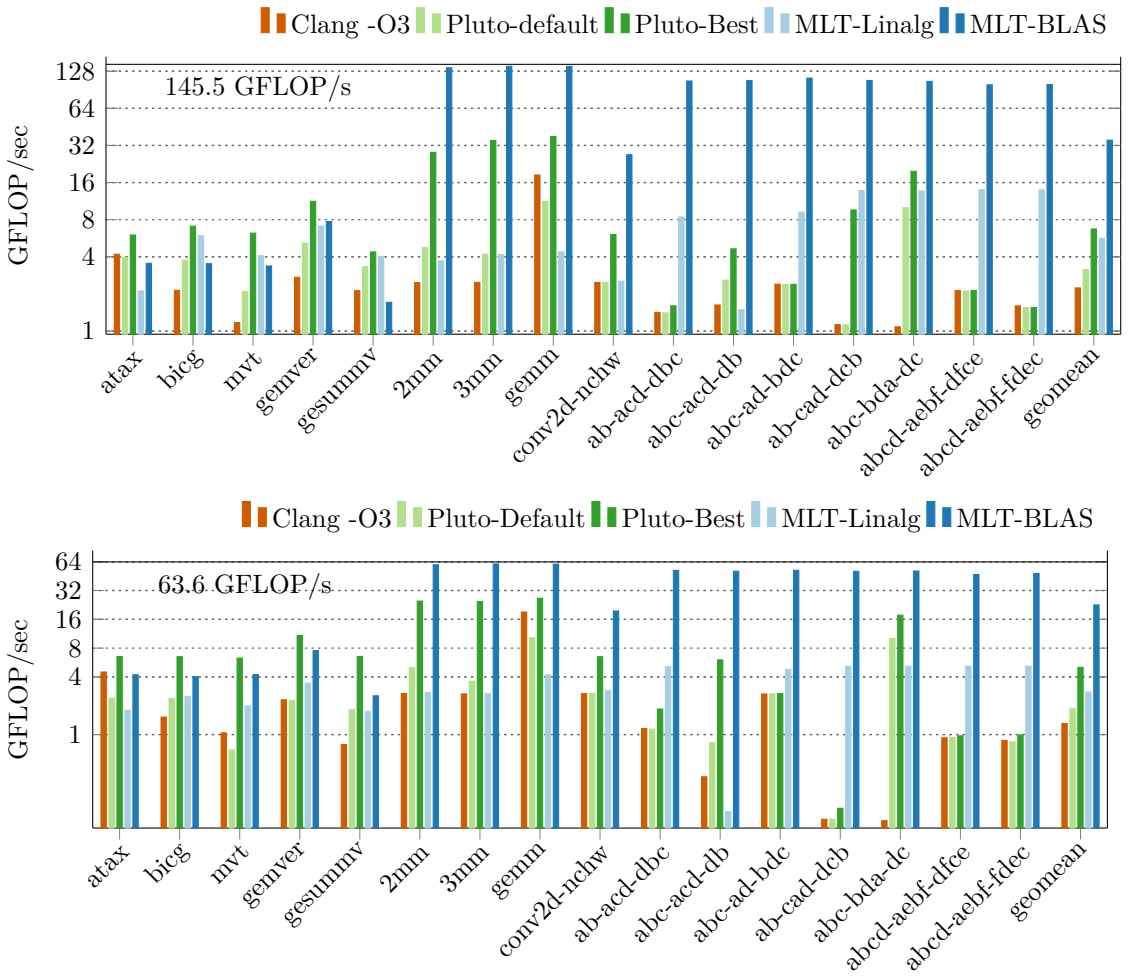


Figure 4.4: Performance obtained for single-precision operands for two different architectures. Multi-level Tactics allows recovering semantic information in general-purpose code and exploit domain-specific optimizations by lifting to the Linalg dialect. At the Linalg abstraction, we follow the Linalg code generation path or emit calls to vendor-optimized libraries directly. Results for the Intel i9 are shown on top, and for the AMD 2920X at the bottom.

implementing a GEMM operation with `clang -O3` (release 6.0.0) achieves a baseline performance of 1.76 GFLOPS/s, raising to `matmul` with Multi-level Tactics and subsequent optimization with OpenBLAS/Blis yields 23.59 GFLOPS/s, corresponding to a 13.4× speedup.

4.2.2 Raising Affine to Linalg

The lifting from loop nests to `matmul` operations improves performance significantly but remains specialized to GEMM operations and the specific OpenBLAS/Blis optimization. In this section, we generalize the lifting with Multi-level Tactics from loop nests to the Linalg dialect, covering matrix multiplications, matrix-vector products, convolutions and TTGT conversions of tensor contractions to matrix products.
We first evaluate a single-step lifting scheme that replaces loop nests with Linalg op-

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

As input programs for the evaluation, we use a set of linear algebra benchmarks from the Polybench 4.2 suite and collected from previous studies related to tensor contractions [103]. For the contractions, we include tensors with different dimensionality from relevant domains used in coupled-cluster methods [104] and chemistry kernels [105]. For Polybench 4.2, we selected only those benchmarks that can be mapped to current available Linalg operations, leading to the exclusion of `syrk`, `symm`, `syr2k`, `trmm`, and `doitgen`.

Figure 4.4 shows the performance results for each of the selected benchmarks:

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

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

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

When comparing `MLT-Blas` with `Pluto-best` for the AMD the largest speedups

[2] As of git version 48c28d5, Linalg primarily performs tiling, but work is in progress to have more competitive performance with Blas libraries [102].

[3] git commit f62d61b8

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

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

Listing 29: Structural matcher to detect a chain of 3 matrices multiplications in the Linalg dialect.

are observed for kernels where Multi-level Tactics maps to level-3 BLAS (`2mm` to `abcd-aebf-fdec`). The highest speedup is for `ab-cad-dcb` (294×) while the lowest speedup can be observed for `gemm` (2.3×). Considering kernels which map to level-2 BLAS (`atax` to `gesummv`), `Pluto-best` obtains better performance than `MLT-Blas`. This loss in optimization opportunity is the result of additional overhead introduced by Multi-level Tactics (and MLIR) to link vendor-optimized libraries dynamically. As an example, neglecting the constant overhead of 1.5 ms for the `atax` kernel, its performance would be on-par with `Pluto-best` at 6.5 GFLOPS/s. A similar observation can be made on the Intel system. For level-3 BLAS kernels, the performance of `MLT-BLAS` exceeds `Pluto-Best`, with the highest speedup for `ab-acd-dbc` (66×) and the lowest speedup for `gemm` (3.78×).
When comparing `MLT-Linalg` with `Pluto-default` for the AMD system, `Pluto-default` gives slightly better performance for all level-3 BLAS kernels except for the contractions. This is expected, as Pluto's smartfuse heuristic applied on top of the tiling transformations significantly reduces control-flow overhead, while for the contractions, the TTGT transformation allows for significant improvements in data locality for `MLT-Linalg`. An exception is `abc-acd-db` and `abc-bda-dc`, where in the latter Pluto vectorizes the innermost loop, resulting in better performance. Similar behavior is observed for the Intel system, where `Pluto-default` performs better on all kernels except for contractions, exception made for `abc-acd-db`.
Finally, we evaluate the compile-time overhead introduced by Multi-level Tactics. Lowering the 16 benchmarks considered above from Affine to MLIR LLVM takes 0.64 s with an unmodified MLIR version[5] compiled in release mode. In comparison, it takes 0.72 s for Multi-level Tactics to lift from Affine to Linalg and then lower to MLIR LLVM, which represents an increase of only 12% of the compilation time.

4.2.3 A Case for Progressive Raising: Reordering Matrix Chain Multiplications

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

[5]git commit 48c28d5

Table 4.2: Multi-level Tactics enables the matrix-chain multiplication optimization at Linalg level by providing a raising path from source code written in C.

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

animation [106, 107] and is formulated as follows: Given a product of n matrices of the form $A_1 \times A_2 \times ... \times A_n$ of sizes $p_{i-1} \times p_i$, the matrix-chain optimization problem consists in finding the optimal parenthesizations that minimize the number of scalar multiplications [108].

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

As a starting point, we consider a matrix-chain multiplication expressed as a set of nested loops in C and use MET to enter the MLIR compilation pipeline at the Affine dialect. We then use the tactic shown in Listing 28 with the compilation flag `-raise-affine-to-linalg` to raise from the Affine loop-based abstraction to Linalg. To detect chains of matrix multiplications, we use a set of rewriting rules based on our `m_Op` matcher. Listing 29 shows an example for chains of three matrices. The input matrices A, B, and C and the final result D are captured via `m_Capt` to enable the builder to generate the re-parenthesized expression with the minimal number of scalar multiplications.

We evaluate the impact of the above transformation on the AMD system on three different matrix-chain multiplications taken from the literature [106]. Table 4.2 reports the sizes of the various matrices we consider as well as the initial and optimal parenthesizations. We additionally report the execution time for the optimized (time OP) and naive (time IP) parenthesizations. In all cases, the reduction in the scalar multiplication is reflected by faster execution time. For example, if we consider the chain with four matrices, the original order's execution takes 1.289 s (2.37 GFLOPS/s). In contrast, the new order only requires 0.212 s for completion, corresponding to a speedup that is proportional to the reduced scalar operations of 6.08×.

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

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

Figure 4.5: Simplified EBNF syntax for Tactics Description Language. Brackets denote optional clauses, curly brackets indicate repetitions, and square brackets contain a textual description for simplicity.

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

Figure 4.6: Simplified EBNF syntax for Tactics Description Specification. Curly brackets denote repetitions, and angle brakets contain textual description.

4.3 Multi-level Tactic syntax

Figure 4.5 and 4.6 show the grammar of the Tactics Description Language (TDL), and the Tactics Description Specification (TDS), respectively. TDL extends the TC syntax [3] with the `pattern` and the `builder` keywords. Figure 4.5 shows a simplified version of the TDL core syntax. Figure 4.6 shows the grammar for TDS where each entry derives from the `Tactic` class which allows for the specification of the pattern using TC syntax and a list of builders. TDS supports five different builders: reshape, transpose, matmul, matvec and convolution, each mapping to a high-level operation (i.e., linalg.matmul). All builders support multiple inputs, except `transpose` and `reshape`, which only process a single input. All builders produce a single output. The `transpose` and `reshape` builder require an affine expression as the third argument, specifying how the dimensions should be transposed or reshaped, respectively.
Figure 4.7 shows the grammar for the access and structural matchers using the extended Backus-Naur form. Each access matcher, placeholder and array placeholder, must belong to a context. The context orchestrates the matching by tracking the assignment of `mlir::Value` to placeholders. Matchers cannot be constructed if the context is not already instantiated—when the context goes out of scope, everything is freed. A placeholder is a result of calling `m_Placeholder`. Placeholders support operators overloading to match any affine expression of the form $k * \iota + c$, where k and c are coefficients from the pattern and ι defines the candidate. A placeholder list is an ordered collection of placeholders. The position of the placeholder in a place-

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

Figure 4.7: EBNF syntax for access and structural matchers.

holder list is implicitly inferred by its position in the arguments list. The position is taken into account during the matching. An array placeholder can be constructed by calling `m_ArrayPlaceholder`, and a list of placeholders can be assigned to it. Similarly, structural matchers can be constructed only after the context has been instantiated. Structural matchers can take a callback as an optional leading argument. A callback allows for finer-grained control over the matching by testing i.e., access pattern properties.

4.4 Related Work

Idiom recognition is an old and well-known problem in computer science since the 1990s. Each previous work can be broadly classified in one of the following categories: text, syntactic, and semantic.

Text-level and syntactic-level methods Text-based tools operate directly on the source code while syntactic ones at the AST level [84, 85, 86]. These two categories have fallen out of fashion, mainly due to the weak robustness against code changes, leaving the stage for the more systematic semantic approaches.

Semantic-level methods In this category, Ginsbach et al. propose IDL an Idiom Description Language that gets lowered to a set of constraints [109]. A match is a code fragment that adheres to the set of specifications. Their approach is fully automated and implemented in the LLVM compiler. Compared with Multi-level Tactics, IDL has the advantages of detecting sparse-linear algebra, which is not yet supported in MLIR and Multi-level Tactics. But relying on a constraints solver to discover idioms increases the compilation time by a large margin. In their paper, they report an increase in compilation time of 82% on average. Contrary, as demonstrated in the evaluation, Multi-level Tactics has a negligible overhead. In

a follow-up work, Ginsbach et al. propose LiLAC, a language and a compiler for accelerating sparse and dense linear algebra [110]. Idiom discovery is still based on a constraints solver, but the pattern specification is made easier by introducing a DSL. Similarly, to TDL their DSL enables the specification of the "what" and the "how". The "what" defines what to match while the "how" how the library should be invoked to accelerate the "what". LiLAC can accelerate sparse linear algebra but each pattern maps to a single BLAS call. Vice-versa, Multi-level Tactics enables expressing a pattern as a composition of library calls but does not support sparse linear algebra. Arenaz et al. with the XARK compiler enable idiom recognition by analyzing use-def chains in Strongly Connected Components (SCCs) on the Gate Single Assignment Form, an extension of the popular SSA form [111]. However, the linearization of multi-dimensional arrays fundamentally limits the complexity of the patterns that can be detected. Contrary, Multi-level Tactics works within the MLIR infrastructure, which enables whenever possible (i.e., the access is not already linearized) to treat multi-dimensional accesses as first-class citizens. Chelini et al. with Declarative Loop Tactics bring domain-specific optimizations in general-purpose flow by providing compiler developers with a tool to add highly customized optimizations for a given computation motif [112]. Despite Multi-level Tactics shares a lot of commonality with Loop Tactics, it goes one step further by lowering the barrier of writing matchers and boosting productivity by enabling their automatic synthesis via TableGen. Felleisen et al. introduce Racket, a language extension API, to extend the host language's syntax and semantics [113]. Thus enabling programmers to embed context sensitive-information for optimization purpose. While MLT has some similarity with Racket, and in more general with language-oriented programming framework, it also a significant difference [114, 115]. Specifically, MLT focus on the IR level and not the source level, benefiting compiler developers, not application developers. Brown et al. in the Delite framework use rewriting rules to apply domain-specific optimizations, while our end goal is similar (i.e., applying domain-specific optimizations), Multi-level Tactics does that by *raising* from low-level abstractions [116]. Frameworks such as Halide decouple the function description of a problem from its execution strategy [63, 64]. Generally, the former is written by a domain expert while the latter by a performance one. Such tools require domain experts to get acquainted with the tooling syntax (i.e., Halide syntax) to write the problem's function description. On the other hand, using Multi-level Tactics, a domain expert can write plain C++ code, thus lowering the language barrier–no need to learn a new language assuming the expert already knows C++. In parallel or even before, thanks to the decoupled nature of matchers and builders, a performance expert can provide a tactic to decide the best execution strategy.

Idioms in software engineering The notion of recognizing the existence of idioms goes beyond what is described so far for more compiler-oriented approaches, and it can be found in other disciplines like software engineering. Software design patterns describe program components as a specializing implementation for a particular class of standard approaches. Many functional programming languages, like OCaml, encapsulate standard programming high-order functions like map and

reduce. The Berkeley Dwarft [117] and the Galois system [118] are collections of computational motifs that comprise a large portion of the most common parallel computing workloads. Both systems were studied from a perspective of architectural requirement or to guide the programmer by providing descriptions of commonly occurring problems, not with an automatic compiler recognition in mind.
Another abstraction related to computational motifs is an algorithmic skeleton [119] introduced to classify the behavior of parallel programs. The main motivation behind this classification is to introduce higher-level order functions and tools for parallel programming. Multiple frameworks [120] and libraries [121] use skeleton programming. Algorithmic skeletons have a similar abstraction to the Berkley Dwarft and are thus more oriented toward the design of libraries and DSLs.

4.5 Summary and Conclusion

We presented Multi-level Tactics and its implementation in the MLIR framework. At the heart of this Multi-level Tactics is the idea of enabling progressive raising—a complementary path to the progressive lowering offered by multi-level IR compilers. Our progressive raising enables us to lift the entry point of general-purpose languages, thus enabling domain-specific optimizations in a multi-level IR.
We showed how Multi-level Tactics allows expressing patterns and builders concisely based on a *Tensor Comprehensions* inspired syntax, and demonstrated our framework for two raising paths: Affine to Affine or Affine to Linalg. Besides, we illustrated a case for progressive raising by implementing a transformation on top of the Linalg dialect by reordering chains of matrix multiplications.
To the best of our knowledge, we are the first to provide an infrastructure and a demonstration to achieve progressive raising in a multi-level IR compiler like MLIR.

5

Programming Novel Architectures

With the end of Dennard's scaling and the diminishment of Moore's law, the semiconductor industry is moving towards specialization, which surfaced as a promising approach to deliver orders of magnitude performance and energy benefits compared to general-purpose computing. Unfortunately, the adoption of heterogeneity in the software stack cannot keep up with the hardware evolution. If we don't want to lose our ability to sustain progress in a fundamental area like deep learning, we need to evolve and retarget our software stack at a comparable high pace, without incurring high software development and maintenance costs every time a new accelerator is launched.
This Chapter shows how relying on a multi-level intermediate representation compiler enables the fast adoption of heterogeneity by modeling the right abstraction level and how abstraction raising enables heterogeneity on legacy code by automatically and transparently offload computational idioms on specialized accelerators. We will focus on deep-learning applications and novel in-memory accelerators, albeit our idea is applicable to any accelerator that exposes an API.

5.1 Why Computing-In-Memory?

Three horsemen have driven recent developments in Artificial Intelligence: availability of vast amounts of data, continuous growth in computing performance and algorithm innovation. But, one of them is getting to a halt. Von-Neumann-based architecture can no longer satisfy the energy and the compute requirements requested by today's Deep Learning applications. Recent analysis shows that the demand for computing power has increased by 300000× since 2012 [5]. Current estimations predict an additional doubling in compute performance every 3.4 months—a much higher and faster rate than what Moore's scaling has provided us till today. The increase in the amount of data to be processed highlighted another criticality of Von-Neumann-based architectures: the energy cost of moving the data to and from the main memory. Moving a word from the DRAM main memory to the on-chip local SRAM is 3000× more expensive then to perform a multiplication operation on the same word. To give a simple example, on a 45 nm CMOS technology, moving a single word from a low-power DRAM to a faster SRAM costs 640pJ; moving from a

This Chapter is based on: Vadivel et al., "TDO-CIM: transparent detection and offloading for computation in-memory" DATE, 2020, Drebes et al., "TC-CIM: Empowering Tensor Comprehensions for Computing In-Memory" IMPACT, 2020 and Siemieniuk et al., "OCC: The Open CIM Compiler" under submission at TCAD.

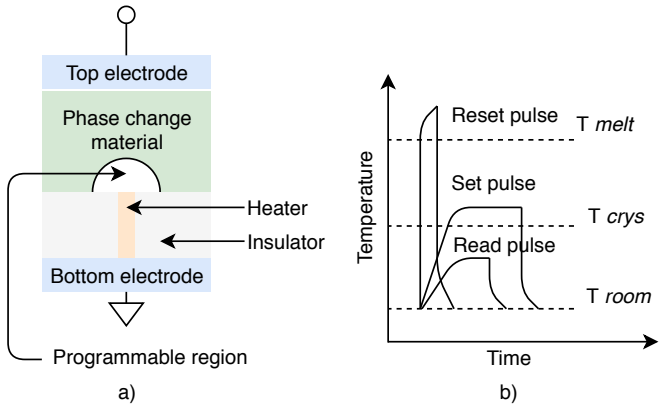


Figure 5.1: Cross section of a PCM device (a) and its programming pulses (b).

faster SRAM to the register file accounts for an additional 55pJ. Contrary, an 8-bit multiplication requires only 0.2pJ.
Consequently, if we want to keep increasing compute performance and energy efficiency we need to depart from the traditional Von-Neumann architecture where compute and storage are physically separated (compute-centric) to a more data-centric approach where compute and storage happens on the same physical device. Computing-In-Memory (CIM) is a promising approach to deliver the next leap in energy and compute efficiency requested by today's DL applications.

5.1.1 Memristor Basics

Computing-In-Memory relies on the use of memristive devices. Memristive devices are build using different technologies and materials, in this Chapter we use phase change material (PCM) devices. PCM is a type of non-volatile memory that stores information by changing the cell resistance, switching between amorphous and crystalline states. The transition between the two states happens as a consequence of the application of external voltages that exceed the threshold voltage of a device. Figure 5.1 (a) shows a cross-section of a PCM device. The phase change material is sandwiched between two electrodes, and current is applied through the heater in order to change the material state. A short, but intense, pulse—known as reset pulse—is used to bring the material in the amorphous phase (high-resistance). Contrarily, to switch back to low resistance the set pulse—a lower and longer pulse—is applied. To read the device, an even lower pulse (read pulse) is used (Figure 5.1 (b)). Thanks to the excellent scaling capabilities of PCM devices—which allows increasing main memory capacity in a cost-effective and power-efficient way—it is expected that PCM will play a significant role in future memory architectures [122]. Assembling memristive devices into crossbar tiles allows for the in-place computation of fixed-size tensor operations in constant time. For example, the dot product of two fixed-size vectors v_1 and v_2 can be accomplished by applying voltages corresponding to the values of v_1 to a column of memristive devices whose conductance correspond to the values v_2 and by measuring the resulting current for the entire column (Figure 5.2). This can be extended to fixed-size matrix-vector in constant time by adding one column of memristive devices for each row of the input matrix

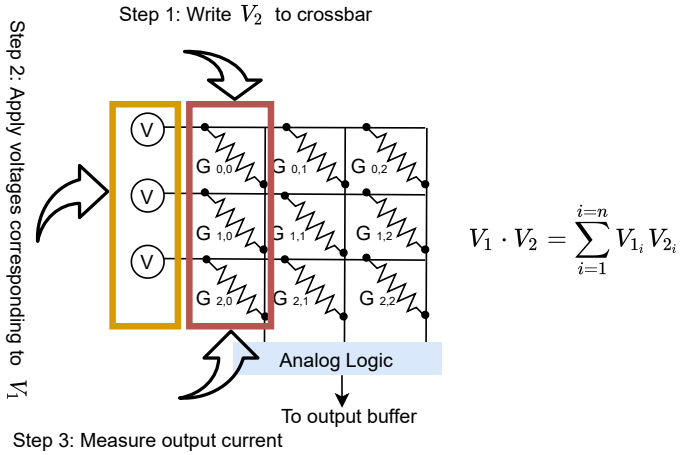


Figure 5.2: Mapping a dot product on a CIM crossbar.

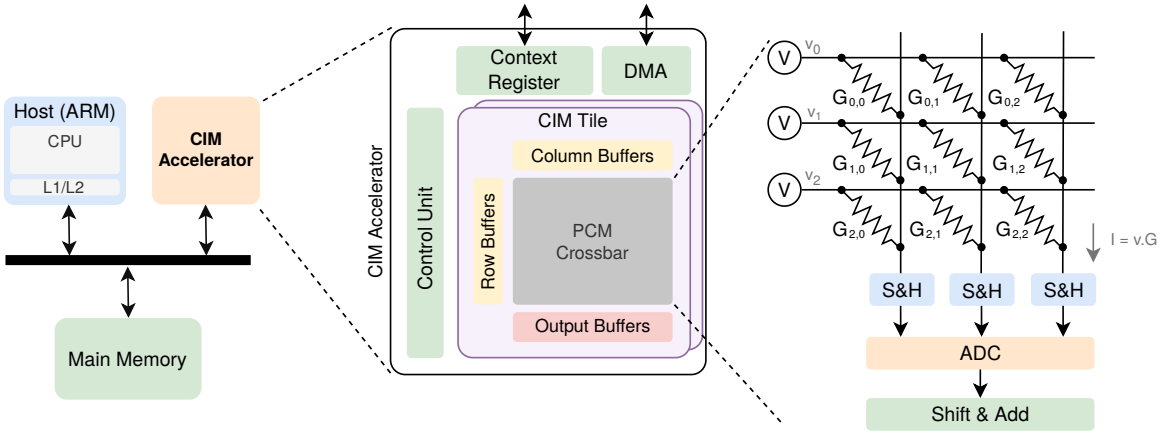
and measuring each column's current.

5.1.2 PCM-based Architecture

The compilation flows presented in the next section assume a System-on-Chip (SoC) where the PCM-based accelerator acts as a co-processor. Figure 5.3 shows the overview of our emulated SoC in Gem5. We use the full-system simulation environment to simulate a bare-metal machine. The SoC consists of a single high-performance in-order (HPI) ARM core with on-chip instruction and data caches (32 kB and 64 kB respectively) and a unified L2-cache of size 2 MB. The core operates at a clock frequency of 2 GHz, uses a main-memory of size 4 GB (single channel DRAM DDR3) and is connected to the CIM accelerator via the system bus. For the memory system, we use the classic memory model in Gem5. The CIM accelerator acts as a co-processor, is memory-mapped, and accesses the shared main-memory using DMA operations.
The accelerator's brain is the Control Unit responsible for orchestrating and steering the internal circuitry. The execution pipeline is represented by an array of CIM tiles. Within a CIM-tile, the memristor crossbar executes the analog vector-matrix multiplication. We model a 4-tile PCM crossbar with a tile size of 64×64 and 8-bit precision using 4-bit PCM. To accomplish 8-bit precision, we rely on the bit-slicing technique, which allows increasing accuracy by combining modules of smaller bit width [123]. To be precise, we implement bit-slicing by distributing the computation over multiple columns, where each column represents a single bit slice. The bits are then weighted at the column output using a shift and add block. The physical properties of each PCM device, such as write and read latency, are taken from the literature [124, 125], and reported in Table 5.1. Additional CMOS peripheral logic is required to interface the PCM crossbar with the rest of the CIM-tile. Sample-and-Holds and ADC converters serve such purpose. Finally, row, columns input buffers, and output buffer in each CIM-tile follow the purpose of storing temporary data that will be read (output) or written (row and column buffer).
In a typical offloading scenario, the host prepares the data on shared memory and

Table 5.1: SOC configuration for CIM.

CIM Parameter	**Value**
Technology(64×64 @8-bit)	IBM PCM 4×(64×64 @8-bit)
Compute and write latency/8-bit	1 µs and 2.5 µs
Compute/Read energy / 8-bit	200 fJ (2×100 fJ / 8-bit PCM)
Write energy / 8-bit	200 pJ (2×100 pJ / 8-bit PCM)
Cell endurance	3.2×10^7
Mixed signal circuit energy	3.9 nJ (@1.2 GHz)
Input/Output buffer energy (1.5 kB)	5.4 pJ/byte-access
Digital logic energy	40 pJ / GEMV for weighted sum and 2.11 pJ / extra ALU operation
DMA and control unit energy	<0.78 nJ
Host CPU Spec	**Value**
ARM-A53, 28 nm	2 GHz
L1-I, L1-D	32 kB, 64 kB
L2, Main-memory	2 MB, 4 GB (DDR3_1600_8×8)



(a) Overview of the architecture **(b)** Architecture of the CIM accelerator **(c)** Memristive crossbar array

Figure 5.3: Overview of the emulated SoC.

triggers the accelerator execution by writing to special memory-mapped registers. The CIM accelerator then reads the data in the shared-memory via DMA transactions. Once done, the accelerator writes back the results in the shared memory. The host monitors the status of CIM execution by polling the status register, and upon completion, it can safely resume execution. Cache coherency is ensured by flushing the CPU cache before acceleration. All steps necessary for offloading (i.e., flushing the CPU cache) are encapsulated by a system library that exposes high-level BLAS-like API. To reduce simulation time in our experiments, we used a lightweight, bare-metal implementation of the library directly interacting with the hardware instead of a full-system simulation with a complete operating system kernel and user-space.

We now present two flows that allow us to target our CISC accelerator from low-level code such as C or a higher-level domain-specific language like *Tensor Comprehensions*. Figure 5.4 illustrates the abstraction level of the CIM accelerator and how the two flows raise or lower to it.

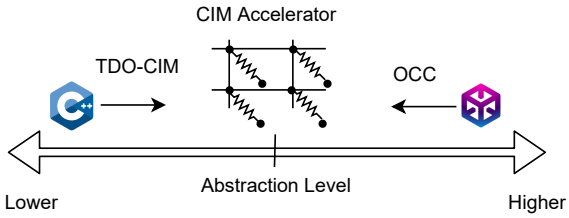


Figure 5.4: Abstraction gap between the CISC-like CIM accelerator, a low-level programming like C, and a domain-specific language like *Tensor Comprehensions*. TC-CIM enables raising to such an abstraction while OCC lowering to it.

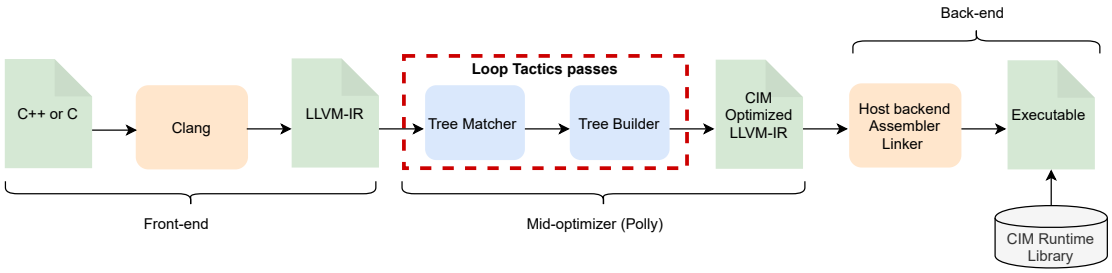


Figure 5.5: TDO-CIM an LLVM-based compilation flow developed for the CIM accelerator. TDO-CIM can transparently detect and offload computational motifs amendable for in-memory execution detected on legacy code.

5.2 Detection and Offloading on Legacy Code

To enable CIM acceleration on legacy code, we use the techniques developed in Chapter 3, which leads to the TDO-CIM compilation flow. A compiler to enable transparent detection and offloading of computational motifs amendable for CIM acceleration on legacy C or C++ code.

Figure 5.5 shows a bird's eye view of TDO-CIM. The entry point is low-level code written in a general-purpose language like C or C++. We use the Clang frontend to lower the code to LLVM IR. At the LLVM-IR level, we rely on the polyhedral optimizer Polly to detect, extract and model compute kernels. Internally, Polly represents each detected kernel's schedule as a tree, which we refer to as a schedule tree. *Loop Tactics* works as additional passes in Polly to detect and replace CIM-friendly computational motifs to function calls to a CIM runtime library.

We evaluate *Loop Tactics* on simple kernels composed of a single tensor operation, we used *mv* (matrix-vector multiplication), *mm* (matrix-matrix multiplication), and *batchmm* (batched matrix-matrix multiplication). As a representative for kernels with multiple tensor operations, we have added *3mm* (two matrix-matrix multiplications and the multiplication of their results) and *4cmm*, which performs four consecutive, independent matrix-matrix multiplications.

As a more advanced use case, we experimented with the multi-layer perceptron component of a production model reported in the *Tensor Comprehensions* paper [126]. This model has trailing fully-connected layers followed by non-linearities, which corresponds to matrix-matrix multiplications and point-wise operations. In *Tensor Comprehensions*, these layers were split into two parts: MLP1 and MLP3, with

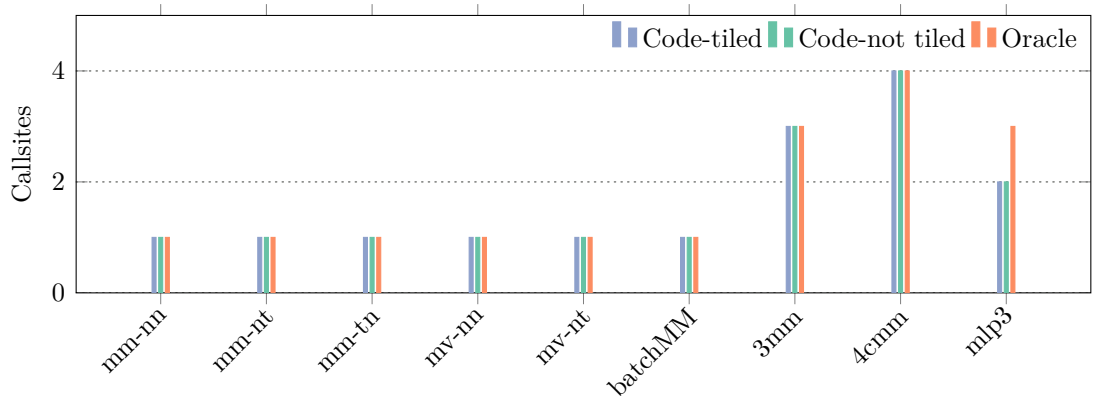


Figure 5.6: Number of callsites for CIM library functions inserted by *Loop Tactics* without (Code-not tiled) and with prior tiling (Code-tiled) compared to perfect matching (Oracle).

the former containing a single matrix multiplication and the latter containing three matrix multiplications using each other's result. Since the single matrix multiplication of MLP1 is already covered by *mm*, we have only added *mlp3* to the evaluation.

In order to match the element size of the CIM accelerator, we used tensor elements with a size of 8 bit in all benchmarks.

5.2.1 Kernel Experiments: Static Impact

For each combination of benchmarks and workloads, we have generated specialized code with *Loop Tactics* by setting the parameters corresponding to the workload sizes statically. As a metric for the success of the matching, we have determined the number of callsites for CIM library functions statically from the source code and compared it to the maximum number of callsites expected for perfect matching (`Oracle`). Figure 5.6 shows the number of callsites for each of the benchmarks, under two different conditions: in `Code-not tiled`, we carried out the matching right after running the affine scheduler, while for `Code-tiled`, we additionally forced tiling on the three outermost loops with tile sizes of 32 before the matching. The suffixes for *mm* and *mv* indicate whether the operands are transposed (*nn*: no transposition, *tn*: first operand transposed, *nt*: second operand transposed).

For the non-tiled version, *Loop Tactics* detects all but one matrix multiplication in *mlp3*. This multiplication is missed, as it does not match the structure expected by the matcher: the matcher expects the initialization statement and the core-computation statement to be filter nodes under the same sequence, while in this case, the filters are children of different sequences.

For the tiled version, *Loop Tactics* is still capable of detecting all operations for all benchmarks. Indeed, tiling does not affect detection at all since *Loop Tactics* runs a canonicalization pass which squashes together point-loop and tile-loop bands.

At this stage, *Loop Tactics* reliably detects the relevant patterns for most cases, even in the presence of prior tiling transformations and for transposed accesses. The remaining mismatches motivate further work to better coordinate affine transformations and matchers/builders.

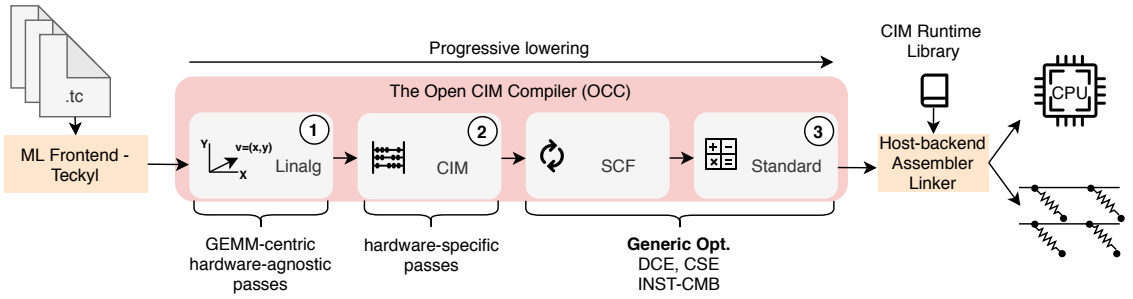


Figure 5.7: The different building blocks of the Open CIM Compiler.

5.3 The Open CIM Compiler (OCC)

The Open CIM Compiler (OCC) is a fully automatic end-to-end compilation framework based on multi-level IR rewriting, which allows reliable mapping of computational motifs to the crossbar in a transparent way, without any user intervention from the *Tensor Comprehensions* domain-specific language. OCC performs two kinds of optimizations: (1) a set of high-level hardware agnostic passes that rewrite each computational motifs using the matrix-matrix product as a basic building block. (2) A set of low-level hardware-specific optimizations to ensure the computation fits the CIM crossbar and to reduce the number of write operations, increasing the crossbar's lifetime. OCC comes with a frontend to express tensor computations arising in the ML domain based on *Tensor Comprehensions* (Teckyl).

5.3.1 The OCC Lowering Pipeline

In OCC compilation flow (Figure 5.7), the entry point is a collection of computational motifs defined in a productive-oriented language for tensor operations, *Tensor Comprehensions*. We use Teckyl [99] to enter the OCC lowering pipeline at the Linalg abstraction. At the Linalg level, OCC performs a set of hardware-agnostic passes to rewrite computational motifs for a CIM-friendly execution. All the passes serve to rewrite each motif (e.g., contraction) using the matrix-matrix multiplication as a basic building block to execute them efficiently on the crossbar. The Linalg dialect is then lowered to CIM. The CIM dialect acts as an interface to our accelerator. It performs hardware-specific optimizations to ensure the computation fits in the crossbar array and to reduce the number of write operations to increase the crossbar lifetime. Besides, it orchestrates the data movement to and from the device. During lowering from CIM to SCF and then to Standard, the operations amenable for CIM execution are mapped to function calls to our accelerator runtime library. The remaining operations that are not being offloaded to CIM follow the route toward the CPU code generation path.

5.3.2 OCC Transformations

We distinguish between three types of transformations that work in symbiosis across our entire lowering pipeline: hardware-agnostic rewriting passes ① in Figure 5.7,

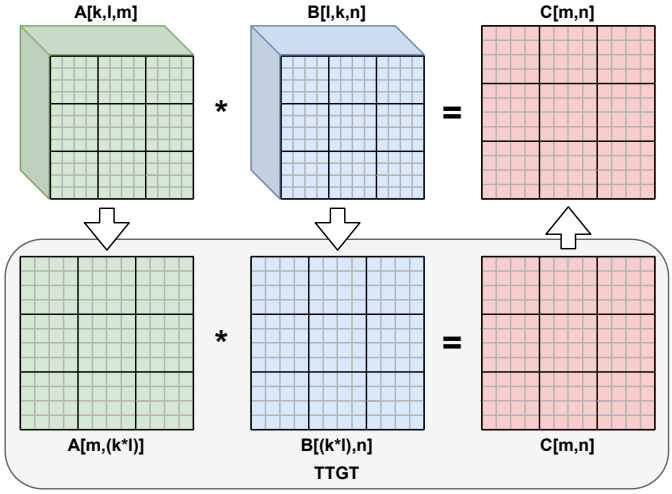


Figure 5.8: TTGT enables to execute contractions as GEMM operations.

```
def contr(int16(K,L,M) A, int16(L,K,N) B) -> (int16(M,N) C)
{
  C(m,n) += A(k,l,m) * B(l,k,n)
}
                ⇓ lowers to
%0 = linalg.transpose(%A, {2, 0, 1})
%1 = linalg.transpose(%B, {1, 0, 2})
%2 = linalg.reshape(%0, {0, {1, 2}})
%3 = linalg.reshape(%1, {{0, 1}, 2})
// eligible for offloading to CIM
linalg.matmul(%2, %3, %C)
```

Listing 30: TTGT reduces each contraction to a sequence of transpose, reshape, and GEMM operations.

hardware-specific passes to adapt the computational motif to the hardware features ②, and the actual lowering to CIM library ③.

Hardware-agnostic Rewriting Passes All hardware-agnostic passes work at the Linalg level and aim at rewriting motifs using the matrix-matrix multiplication as building block. OCC supports two rewriting passes: TTGT [127] for contractions and Im2Col [128] for convolutions.
The **TTGT pass** automatically detects and applies the TTGT transformation scheme on each Linalg contractions (Figure 5.8). TTGT stands for Transpose Transpose Gemm Transpose and enables rewriting a contraction as a composition of transpose, reshape, and GEMM operations. More specifically, the main idea is first to flatten the tensors into matrices via direct tensor transposition and reshape operations, then execute a single GEMM operation, and finally, fold back the resulting matrix into the original tensor layout. Only the GEMM operation will be offloaded to the CIM accelerator, while the rest of the operations will follow the CPU code generation path. More concretely, consider how the TTGT pass rewrites the contraction in Listing 30 as a sequence of Linalg operations. Two `linalg.transpose` operations

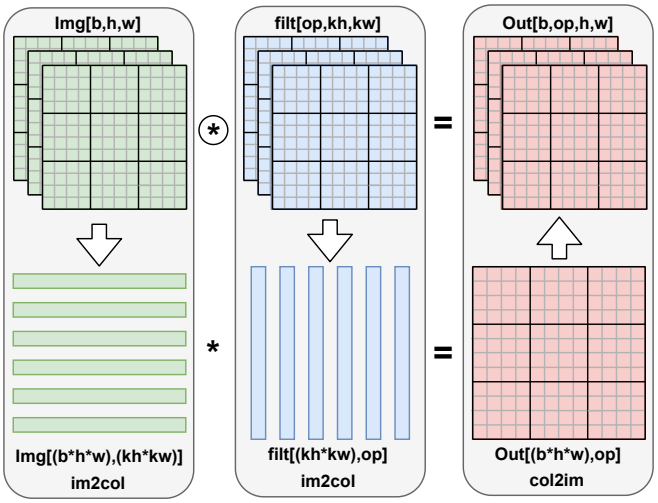


Figure 5.9: Im2Col enables execution of convolutions a GEMM operations.

```
def conv2d(int16(B,I_P,H,W) Img, int16(O_P,I_P,K_H,K_W) Filt)
    -> (int16(B,O_P,H,W) Out)
{
  Out(b,o_p,h,w) +=! Img(b,i_p,h+k_h,w+kw) * Filt(o_p,i_p,k_h,k_w)
}
                ⇓ lowers to
%0 = linalg.im2Col(%Img)
%1 = linalg.im2Col(%Filt)
// eligible for offloading to CIM
linalg.matmul(%0, %1, %OutTmp)
linalg.Col2Im(%OutTmp, %Out)
```

Listing 31: Im2Col transforms each detected 2-dimensional contraction in a GEMM operation. GEMM operations will be executed efficiently on the CIM crossbar.

are emitted to re-arrange the dimensions for tensor `A` and `B` (e.g., $(k, l, m) \rightarrow (m, k, l)$ for `A`). The transposes' outputs are then fed to `linalg.reshape` operations that collapse the second and third dimensions for `A` and the first and second dimensions for `B`, reshaping the tensors to matrices. Finally, a `linalg.matmul` is emitted. For our running example no additional reshape and transpose operations are needed for the output tensor `C`.

Complementary the **Im2Col pass** automatically detects and applies the Im2Col transformation to convolutions (Figure 5.9). Only spatial convolutions using N_C data format are supported [129]. The Im2Col transformation is based on the idea that convolution is no more than a dot-product between the kernel filters and the moving window's local regions. If we take each window and stack them in a column-wise order and we do the same for the filters, but in a row-wise order, we obtain two matrices, and the output of multiplication between them will be equivalent to the original convolution's output. Similarly to the TTGT transformation, only the GEMM computation will be offloaded to the CIM device, while the remaining operations will follow the CPU code-generation path. Listing 31 shows how a 2D-

Operation	Datatype	Target	Description
cim.write(%id, %matB)	integer, memref	CIM	Transfer and write the 2D buffer %matB to the crossbar of the CIM tile %id, synchronously (or blocking).
cim.matmul(%id, %matA, %matC)	integer, memref, memref	CIM	Transfer the 2D buffer %matA to the CIM tile %id, perform GEMM and store the results in the 2D buffer %matC, asynchronously (or non-blocking).
cim.barrier(%id)	integer	HOST	Wait for work completion on the CIM tile %id.
%tile = cim.copyTile(%mat, %row, %col)	memref, memref, index, index	HOST	Allocate a contiguous 2D buffer %tile and copy a tile at the position (%row, %col) from the 2D buffer %mat.
cim.storeTile(%tile, %mat, %row, %col)	memref, memref, index, index	HOST	Copy the tile %tile to the 2D buffer %mat at the position (%row, %col).
cim.accumulate(%lhs, %rhs)	memref, memref	HOST	Add corresponding elements of the %rhs to the %lhs.
%output = cim.allocDuplicate(%input)	memref, memref	HOST	Allocate an empty buffer %output which has the same dimensions as the input buffer %input.

Table 5.2: Some of the CIM dialect operations.

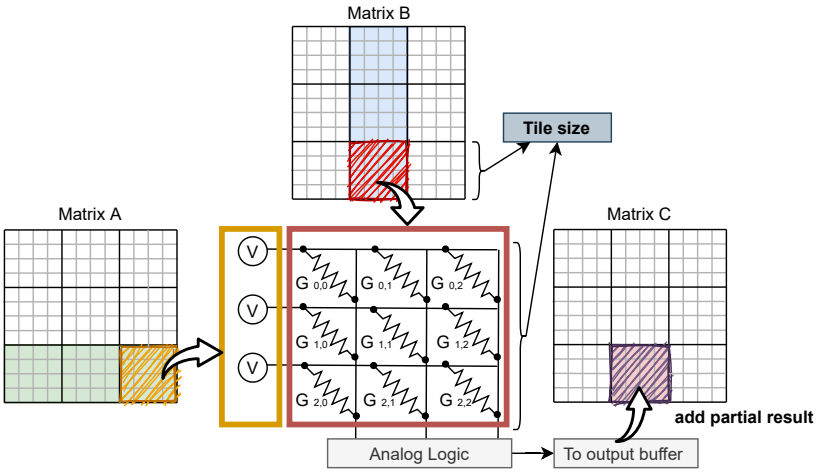


Figure 5.10: Loop tiling used to fit the computation on the CIM tiles.

convolution is rewritten at the Linalg level by applying the Im2Col transformation.

Hardware-specific Passes The CIM dialect focuses on interfacing high-level Linalg GEMM operations with the underlying accelerator hardware. Table 5.2 shows a subset of operations exposed by the CIM dialect. The optimizations performed at this level focus primarily on maximizing hardware utilization, and enabling the computation on the crossbar. The latter aims to extend the crossbar lifetime.
The **tiling pass** (Figure 5.10), the output of which is shown in Listing 32, allows performing matrix multiplication on data that exceeds a CIM device's capacity. The data is split into multiple smaller GEMM computations. The input buffers are divided into tiled rows (`tiledRows`) and columns (`tiledCols`) equal to $\lceil \frac{M}{tileSize} \rceil$ and $\lceil \frac{N}{tileSize} \rceil$ where the $tileSize$ depends on the size of a crossbar tile. Along the inner dimension K, the number of tiles (`numTiles`) is defined as $\lceil \frac{K}{tileSize} \rceil$. At the buffer boundaries, the tile sizes are trimmed to stay within the matrix dimensions. Because the host and the accelerator communicate through DMA, the transferred

```
// GEMM in the Linalg dialect
linalg.matmul(%A, %B, %C)
                ⇓ lowers to
// tiled GEMM in the CIM dialect
%c0 = constant 0 : i32
%c1 = constant 1 : i32
%id = constant 0 : i32 // tile id
scf.for %i = %c0 to %tiledRows step %c1 {
  scf.for %j = %c0 to %tiledCols step %c1 {
    %tileC = cim.copyTile(%C, %i, %j)
    %tempTile = cim.allocDuplicate(%tileC)
    scf.for %k = %c0 to %numTiles step %c1 {
      %tileA = cim.copyTile(%A, %i, %k)
      %tileB = cim.copyTile(%B, %k, %j)
      cim.write(%id, %tileB)
      cim.matmul(%id, %tileA, %tempTile)
      cim.barrier(%id)
      // tileC += tempTile
      cim.accumulate(%tileC, %tempTile)
    }
    cim.storeTile(%tileC, %C, %i, %j)
  }
}
```

Listing 32: Lowering from a Linalg GEMM to a tiled version that fits into a CIM accelerator crossbar. Bold operations are executed on the CIM accelerator.

```
// original tiled GEMM
scf.for %i = %c0 to %tiledRows step %c1 {
  scf.for %j = %c0 to %tiledCols step %c1 {
    %tileC = cim.copyTile(%C, %i, %j)
    scf.for %k = %c0 to %numTiles step %c1 {
      ...
      %tileB = cim.copyTile(%B, %k, %j)
      cim.write(%id, %tileB)
      ...
    }
    cim.storeTile(%tileC, %C, %i, %j)
  }
}
                    ⇓ transforms to
// loop interchanged GEMM
scf.for %k = %c0 to %numTiles step %c1 {
  scf.for %j = %c0 to %tiledCols step %c1 {
    %tileB = cim.copyTile(%B, %k, %j)
    cim.write(%id, %tileB)
    scf.for %i = %c0 to %tiledRows step %c1 {
      %tileC = cim.copyTile(%C, %i, %j)
      ...
      cim.storeTile(%tileC, %C, %i, %j)
    }
  }
}
```

Listing 33: Loop interchange to reduce the number of writes to the CIM crossbar during a tiled GEMM computation.

data must be contiguous in the memory. This is ensured by `cim.copyTile` operation that copies elements from an input matrix that resides on a specific tile to a temporary buffer. Then a partial result is computed and placed into a temporary, tile-sized output buffer allocated using `cim.allocDuplicate`. These buffers are then accumulated on the host by performing element-wise addition, represented by `cim.accumulate`. Finally, the tile is written back to the provided output matrix C by a `cim.storeTile` operation.

loop interchange pass, which reduces the number of crossbars writes performed during a tiled GEMM computation. In the original tiled GEMM computation the `cim.copyTile` and the `cim.write` operations in the innermost k-loop depend only on two out of the three iterators: `k` and `j`. By interchanging the innermost loop `k` with the outermost one `i`, the `cim.copyTile` and the `cim.write` operations can be moved one level higher. Thus, reducing the number of crossbars writes by a factor of `tiledRows` (Listing 33).

Figure 5.11 shows the **loop unrolling pass** which parallelizes computation across multiple CIM tiles by unrolling the inner dimension loop `k` of the tiled GEMM. The crossbars are first populated with data and then all the computations are performed in parallel. Once the first partial result is available, it is accumulated before waiting

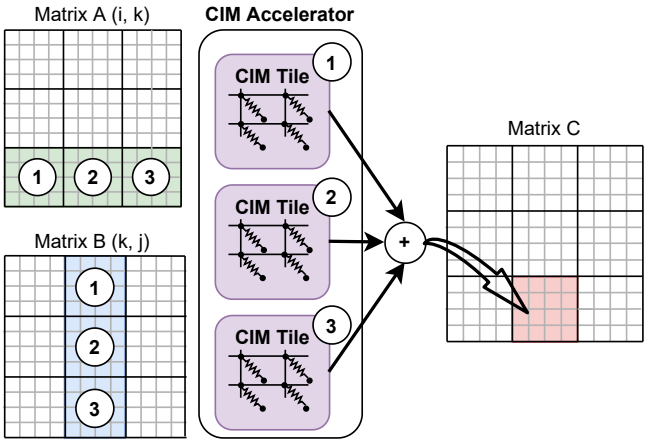


Figure 5.11: Loop unrolling used to parallelize the GEMM computation across multiple CIM tiles.

```
// high-level operations in the CIM dialect
%id = constant 0 : i32 // tile id
cim.write(%id, %B)
cim.matmul(%id, %A, %C)
cim.barrier(%id)
                    ↓
// function call to the CIM runtime library
%id = constant 0 : i32 // tile id
call cimWrite(%id, %B) : (i32, memref<?,?,i16>) -> ()
call cimGemm(%id, %A, %C) : (i32, memref<?,?,i16>, memref<?,?,i16> -> ()
call cimBarrier(%id) : (i32) -> ()
```

Listing 34: Lowering from high-level operations in the CIM dialect to function calls exposed by the CIM runtime library.

for the next CIM tile which helps improving overall latency.

Lowering to CIM Library Calls Similarly to existing works, our CIM accelerator exposes an Application Program Interface (API). Thus the last step in our compilation flow is to lower high-level CIM operations to function calls exposed by the CIM run-time library. Each CIM-dialect operation (Table 5.2) that is accelerator specific (`cim.write`, `cim.matmul`, `cim.barrier`) has a one-to-one mapping with a function call exposed by the CIM runtime library (Listing 34). The rest of the operations get lowered to the other existing MLIR dialects.

5.4 Evaluation

In this section, we demonstrate the efficacy of OCC in transparently detecting and offloading matrix-matrix and matrix-vector multiplications. We first explain our experimental setup and the set of benchmarks used, and then show the impact of OCC offloading and transformations on both performance and energy consumption.

We evaluate the following configurations:

- *arm*: benchmarks compiled with the LLVM static compiler after MLIR code generation (LLVM git commit fc2199d), with no-parallelization enabled. The kernels are executed on the host (ARM) processor with no-call to the CIM accelerator. This provides a baseline for comparison.
- *tile*: program generated by the OCC where the compute kernel is tiled and offloaded to the accelerator. Tiling is required to fit the kernel on the crossbar if the kernel's size exceeds the crossbar's size.
- *tile+interchange*: program generated by the OCC where the compute kernel is tiled and the tiled loops are interchanged to reduce write operations to the crossbar. The CIM-optimized code is offloaded to the accelerator.
- *tile+parallel*: program generated by the OCC where the compute kernel is tiled, parallelized and offloaded across multiple tiles by unrolling the inner loop dimension.
- *tile+interchange+parallel*: all optimizations enabled.

The selection of the ARM core as a baseline is justified by the fact that we focus on developing a new compiler infrastructure for CIM. The evaluation thereof stress demonstrating the effectiveness of OCC and not the CIM device itself. Comparison of the CIM computational paradigm to state-of-the-art multi- and many-cores machines as well as hardware accelerators has already shown in previous works [123, 130, 124].

As for workloads, we use kernels from the ML domain and kernels from previous studies on tensor contractions [103, 126]. For the ML domain, we include applications ranging from simple matrix multiplication kernels (e.g., mm) to a full WaveNet cell [131]. While for the contractions, we included tensors with different dimensionality used in coupled-cluster methods [104] and chemistry calculations [105]. All the workloads are expressed in 8-bit integers to match the precision of the CIM crossbar.

Matrix Multiplication in the context of deep learning is ubiquitous. We consider different flavours of matrix multiplications: `mm` a single matrix multiplication, `2mm` two consecutive independent matrix multiplications, and `3mm` two matrix-matrix multiplications and the multiplication of their results. We consider also a transposed version with `tmm`.

Contraction generalizes matrix-multiplication to N-dimensional tensors. We select three relevant contractions. For example, `abcd-aebf-dfce` is widely used in chemistry computation performing a reduction over the `ef` dimensions and resulting in an output tensor `abcd`. Another use of contractions is in Kronecker Recurrent Units which reduces the size of neural network models [3]. We consider an example of a Kronecker product of three matrices with `kronecker3`.

Convolution is essential in modern neural network-based visual processing. For the evaluation, we consider three representative shapes: `conv1d` convolution between 1D data such as two signal time series, `conv2d` convolution between two 2D inputs commonly used with images and feature extraction kernels, and `conv3d` convolution between data such as video and 3D kernels.

Fused Multi-Layer Perceptron or `MLP3` is a component of a production model which consists of 3 MLP layers that feed into the binary classifier. Each MLP layer consists of matrix multiplication followed by a point-wise operation.

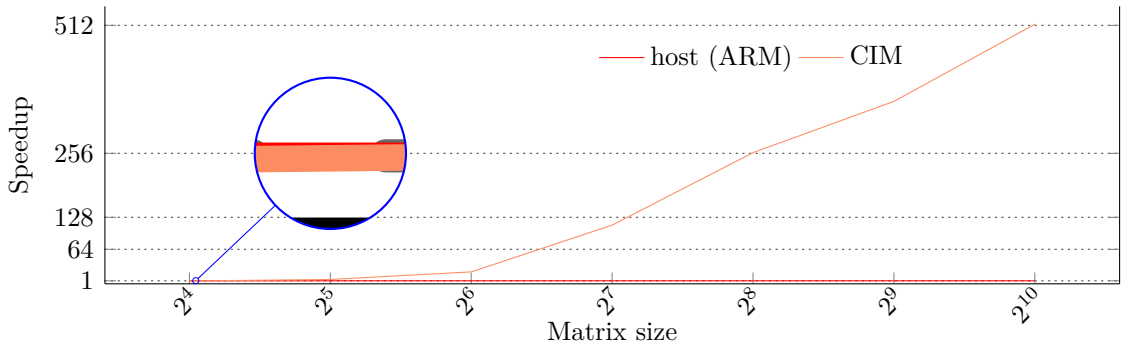


Figure 5.12: Effect of matrix size on the CIM accelerator performance. The results are normalized to the host processor performance. The matrices always fit in the crossbar.

Long Short-Term Memory (`lstm`) is widely used in recurrent neural network that uses temporal dependencies in data sequence (i.e., speech recognition) [132].
WaveNet is a full neural network model that generates raw audio waveforms. The model consists of causal and dilated convolutions, gated activation units, and residual and skip connections. The latter two components are represented by tensor contractions that can benefit from CIM acceleration.
Unsupported Kernels specific workloads are not offloaded due to limitations in the current operation set and mapping pipeline. Batch matrix multiplication and group convolution are not processed as they contain extra dimensions (batch and group dimension, respectively) which require additional, currently unsupported, preprocessing steps to map them into CIM primitives. In case of 2D moments computation, a potential candidate for offloading is omitted as it consists of an input matrix `I` being multiplied with itself `I`×`I`. This computation does not fit the current contraction detection model which assumes that all not reduced dimensions are unique. Finally, some workloads, e.g., a gather operation, are ineligible for CIM acceleration as they cannot be rewritten into equivalent GEMM representation.

5.4.1 Effect of Kernel Size on CIM Performance

Figure 5.12 shows the effect of varying the data size for the mm kernel. For smaller data sizes, programming the crossbar dominates the benefits in the execution time, resulting in 32% better performance for the baseline. But, as the data size increases, the CIM accelerator outperforms the host (ARM) processor by as much as 512×. The performance speedup mainly comes from the parallel nature of the accelerator and the slow down of the ARM processor due to limited data reuse in the small caches as well as its relatively longer execution time.
For the results in the following sections, we fix the size for majority of kernels to 256 to show the impact of tiling and loop interchange transformations. A few exceptions are made to retain representative workload shapes. The convolutions are performed on a single dataset with 3 input feature maps using the default size of 256 in every dimension. The inputs are convolved with 3 output feature maps using a dimension size of 3. The `waveNet` uses batch size of 1, 32 residual and dilation channels, and

64 skip channels. The `lstm` is configured with 64 hidden states. Another exception is made for the two largest contractions which dimensions are reduced to fit them in the main memory. The `abcd-aebf-dfce` is limited to a dimension size of 128 and the `kronecker3` uses size of 64×128 for its weights.

5.4.2 Effect on Performance

Figure 5.13 (top) shows the performance comparison of the ARM core and the CIM accelerator for various workloads. We gather Gem5 statistics only for our region of interest, which consists of the offloaded kernel accounting for its execution and the data transfer time to and from the device. The latter is negligible and less than 3%. We neglect initialization statements as they are always executed on the CPU side in all the considered configurations.

Overall, the CIM accelerator improves performance in the majority of benchmarks and configurations. The tile configuration improves performance by 6.6×. Adding the interchange pass on top of the tile pass gives us extra performance benefits by reducing the number of write operations to the crossbar (7.4×). By parallelizing, we obtain the highest gains 15.5× for the `tile+parallel` configuration and 17× for the `tile+parallel+interchange` configuration.

The effective utilization of the crossbar also has a significant impact on the benchmarks' performance gain. Once mapped, the GEMM-like kernels utilize the CIM accelerator 100%. On the contrary, the kernel size in convolution benchmarks is much smaller compared to the tile size and utilizes only around 2% of the crossbar accelerator. In WaveNet, the utilization of the CIM accelerator is around 6.5%. Similarly, a portion of the computations still has to be performed on the host (Figure 5.14), which further limits the maximal achievable speedup.

Kernels with the highest potential for reuse benefit the most from the CIM accelerator (i.e., `mm`). Contrary kernels with low data reuse (i.e., `mv`, `lstm`) achieve better performance only in the parallelized configuration. This is reasonable as in the non-parallelized configurations, the overhead introduced by the expensive write operations is not amortized due to the low-data reuse, which makes the ARM core faster than the CIM. The convolution kernels get correctly detected and offloaded by the OCC. However, as the obtained results are similar in all three cases, only the most common ML kernel `conv2d` is shown. The convolution uses a box moving window of size 3, thus the workload does not fully utilize the CIM crossbars nor can it be parallelized. The execution time is dominated by the Im2Col transformation overhead, which limits the potential acceleration to the 34% of the kernel computations.

Similarly, the offloaded contractions in `waveNet` perform reduction over the dilation channels of size 32 which underutilize the crossbar and prevent parallelization. The overall speedup is also limited by the significant part of the kernel which remains on the ARM CPU. For comparison, in `waveNet` only 50% of the total computations can be offloaded, while this fraction increases to 99.5% for `mlp3`. Looking at the contractions the `abcd-aebf-dfce` achieves higher speedup than `ab-acd-dbc` and `abc-acd-db` as the baseline performance drops for `abcd-aebf-dfce` due to the high control-flow overhead and high-stride accesses.

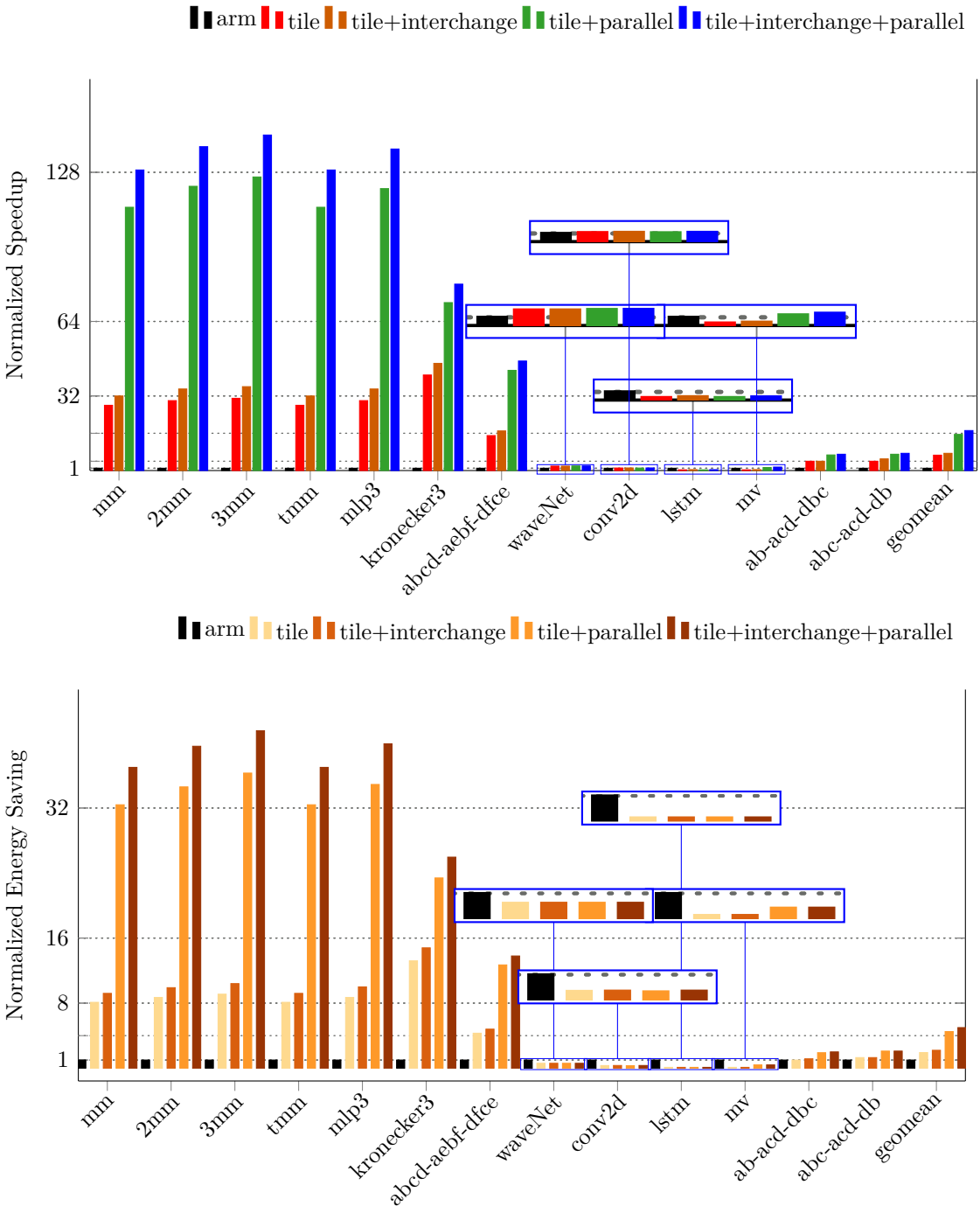


Figure 5.13: Performance (top) and energy results (bottom). All results are normalized to the baseline ARM configuration.

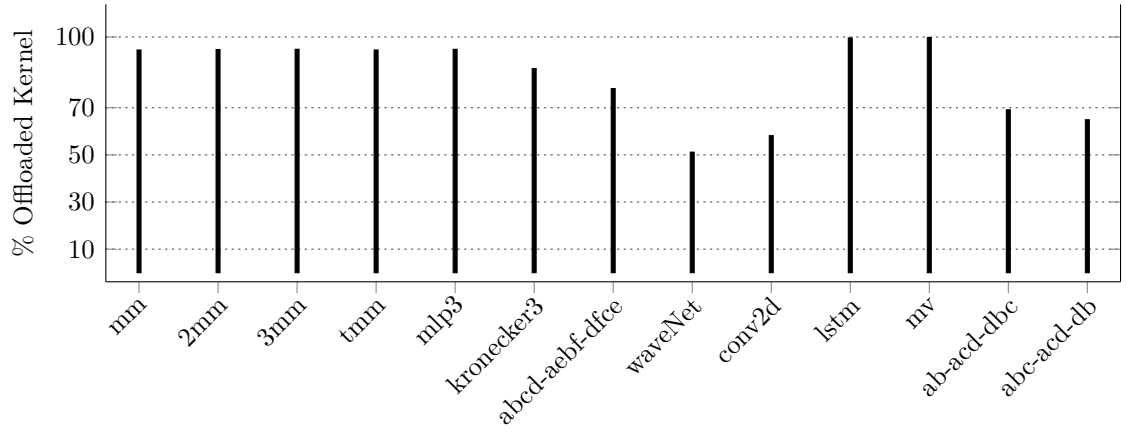


Figure 5.14: Percentage of kernel workload performed on the CIM accelerator.

5.4.3 Effect on Energy Consumption

To measure the energy consumption, we feed McPact [133] with the Gem5 statistics. The energy estimate of the CIM module is not integrated into McPAT and is computed independently and added to the total energy based on the parameters provided in Table 5.1.

The CIM accelerator reduces the energy consumption by an average (geomean) of 1.9×—5.0× (Figure 5.13). While the memory and compute operation in the accelerator are extremely cheap, the control unit i.e., in/out/buffer registers, DAC/ADC and other peripheral circuitry contribute a significant amount to the total energy consumption of the accelerator. The reduction in energy consumption of the CIM accelerator is mainly attributed to the lower compute energy of the device and reduced data movement, compared to the ARM processor.

Compared to the baseline ARM, the highest energy reduction by geomean of 4.5× and 5.0× is achieved for `tile+parallel` and `tile+parallel+interchange` configurations respectively. They are also 2.6× and 2.3× better compared to the `tile` and `tile+interchange` configurations, respectively. The reduction in energy consumption comes from the shorter runtime attained by parallelizing the computations over multiple tiles.

5.4.4 Effect on Endurance

One of the main challenges in PCM-based architecture is the limited write endurance, currently projected between 10^7 and 10^9 writes due to device wear-out [134, 135]. Once the endurance limit is reached the PCM cell loses its ability to transition between state, potentially giving data errors. Multiple works provide wear-leveling techniques as hardware mechanisms [134]; OCC does that on the software side. OCC tackles the problem by maximizing the reuse of written tiles to the crossbar to reduce the overall number of write operations. We compare our default `tile` strategy with the `tile+interchange` where tile reuse is maximized by interchanging the

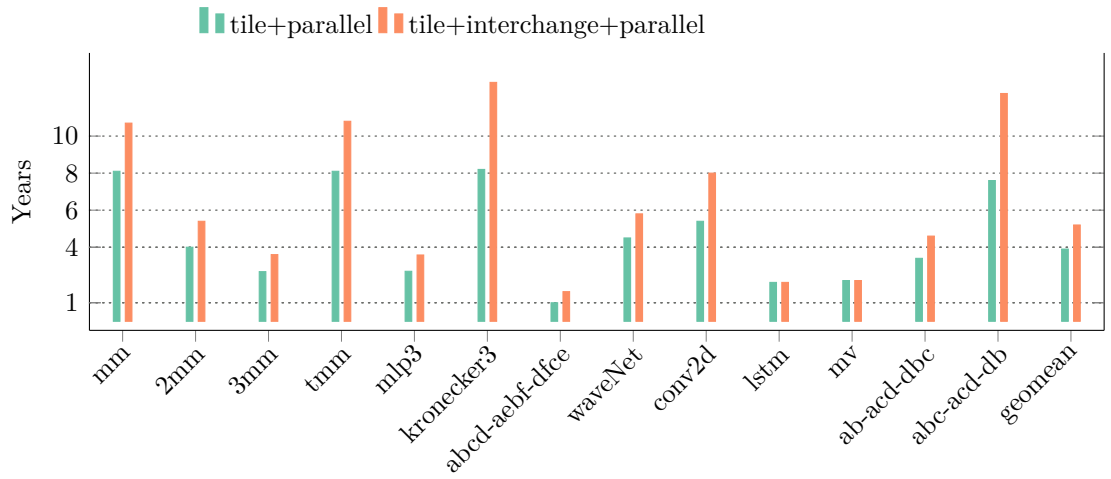


Figure 5.15: OCC increases the crossbar's lifetime by minimizing the number of write operations to the crossbar.

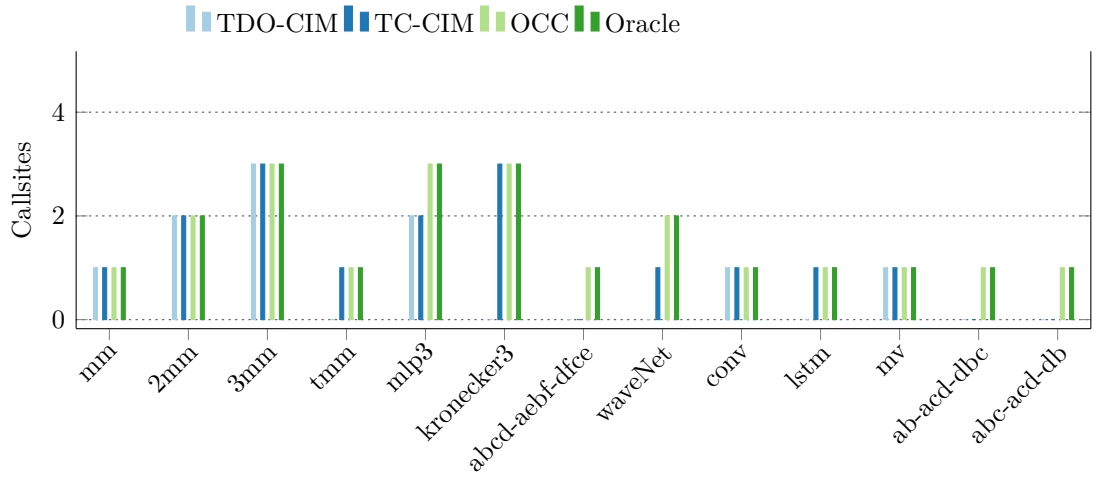


Figure 5.16: Number of callsites for CIM library functions inserted by OCC compared to previous works and a perfect mapping (Oracle).

point loops. To estimate the system lifetime, we adopt the formula below [134]:

$$SystemLifeTime = \frac{CellEndurance * S}{B} \tag{5.1}$$

where S is the crossbar's size ($4 \times (64 \times 64)$), while B is the write traffic in GB/s estimated with Gem5. We assume a cell endurance of 3.2×10^7, as in [134], and a uniform write distribution on the crossbar. Figure 5.15 shows the expected lifetime for the two mapping strategies. By exploiting data reuse OCC increases the device's lifetime by almost 2 years (from 3.9 years to 5.2).

5.4.5 Comparison with previous CIM Compilers

To show OCC's ability to identify and extract matrix and matrix-vector multiplications from multiple benchmarks and its robustness, we compare it with the work of

Vadivel et al. [136] (TDO-CIM) and Drebes et al. [137] (TC-CIM). To the best of our knowledge, these are the only works on CIM compilation that aims at providing an end-to-end compilation for automatically and transparently invoke CIM acceleration. All the other works we are aware of require the users to rewrite the application to explicit CIM acceleration, reducing application readiness. As a metric for success, we count the number of identified matrix-matrix or matrix-vector operations identified by inspecting the generated code and comparing it with the maximum number of callsites expected for perfect matching. Figure 5.16 shows the number of callsites for each benchmark. `conv` refers to either a 1d, 2d or 3d convolution.
OCC behaves like the Oracle for the considered benchmarks, enabling to map each kernel on the crossbar efficiently. TDO-CIM, and TC-CIM, on the other hand, miss some opportunities. Both frameworks are not able to identify and map contractions. Additionally, in `mlp3` and `waveNet` both frameworks miss to identify one matrix-vector multiplication. The main culprit is how both TDO-CIM and TC-CIM operate. Both frameworks rely on Loop Tactics to identify computational motifs in low-level code. Loop Tactics relies on canonicalization passes to provide robust detection; however, sometimes, it may fail. On the other hand, OCC relies on the progressive lowering preserving semantic information till when they are needed.

5.5 Related Work

Let us illustrate what other compiler-based approach exists today for computing-in-memory and near-memory computing, and how they relate to what this Chapter proposed. Let's then discuss how our approach in identifying computational motifs can be extended beyond CIM and how it compares with general-purpose polyhedral accelerator mappers.

Computing-In-Memory Fujiki et. al. proposed a compiler framework that lowers Google's TensorFlow DFG into simpler instructions supported by the in-memory accelerator [138]. During lowering, complex instructions (e.g., division, exponents, and transcendental functions) are broken down into a set of LUTs, additions, and multiplications that can be executed on the crossbar array. The approach also includes a set of scheduling optimizations to expose instruction and block-level parallelism. Software pipelining is used to overlap computation and storage in the CIM crossbar. Ambrosi et. al. proposed end-to-end software stack to support hybrid analog accelerators for ML that are ISA programmable [139]. The software stack includes an ONNX [140] back-end for importing models from popular deep-learning frameworks and a compiler which lower the ONNX description to a custom ISA. During the lowering phase, the compiler performs a set of optimizations, such as graph partitioning and tile placement. The authors present a limited evaluation of their compiler. Building upon the work of Ambrosi et. al., Ankit et. al. developed a runtime compiler implemented as C++ library. The programmer needs to write the application using the library provided constructs, and the compiler will lower the high-level code to assembly targeting their own custom ISA [130]. Contrary to previously mentioned works [138, 139, 130] we decided not implement an ISA for

our accelerator, the main reasons are: (1) the high non-recurring engineering cost of developing an ISA, (2) the loss in crossbar density due to the addition of an hardware decoder. Crossbar density is one of the workhorse for PCM-based devices and we want to preserve it [125].
Similar to our adopted approach, other works expose an API for their accelerator [141, 142, 143, 144]. In the compilation stage, the API is lowered to data-path configurations, as well as executing commands with data dependencies and control flow. However, this approach requires the programmer to write the entire application using the proposed API, hence reducing application readiness. On the other hand, in our approach the API invocation is handled transparently in the compiler; thus, no changes in the application are needed.

Near-memory computing Another computational paradigm that is promising to overcome the memory wall problem is *near-memory computing*, which aims at processing close to where the data resides. One of the main challenges in near-memory compilation is to decide which code portion should be offloaded to the accelerator. Our pattern matching can be seen as an explicit way of performing code offloading. Other works propose cost-based analysis. Hsieh et. al. proposed to statically identify code portions with the highest potential in bandwidth saving using simple cost functions [145]. Pattanik et. al. proposed an affinity prediction model relying on memory-related metrics [146]. Hadidi et. al. identified code region to be offloaded in the context of the Hybrid Memory Cube using a cache profiler, a compile-time analysis phase and benefit analysis models [147]. Differentiating from previous work, Nair et. al. rely on the user to offload code regions that should be marked with OpenMP 4.0 directives [148]. Cost-based analysis can be easily embedded in our approach as callback functions during matching. One of the future works will be the integration of a fast analytical model for associative caches [68] such that we can have a more sophisticated analysis for offloading profitability.

Broader applications of CIM-tile Clearly, there exist a wide CIM design space exploring different technological and architecture tradeoffs. All of them have in common the need to automatically decompose the computation, data, and communication patterns to suit the hardware constraints. Our approach offers a portable abstraction to program any such device, providing automatic tiling and enabling loop transformations specifically suited to the target. Coupling affine transformations with Tactics provides additional performance and specialization through the embedding of target-specific software or hardware blocks. Moreover, many CIM designs—including all finite state and analog-based ones—have in common to expose a limited-functionality API rather than a programmable architecture. In addition to enabling transformations, such architectures strongly depend on Tactics to lower a portable tensor programming layer to target-specific API calls.
Beyond CIM, and thanks to its declarative nature, the matcher-based approach extends easily to other kinds of emerging accelerators with matrix- or tensor-level operations (e.g., GPU tensor cores). Such devices are often programmable through driver library calls, featuring the same interface as numerical libraries, e.g., BLAS. Our approach facilitates the porting of standard tool flows and programming models

to new hardware accelerators, and also helps improving performance on existing hardware when aggressively optimized library implementations are available.

General-purpose (polyhedral) accelerator mapping There are a variety of approaches for automatic accelerator programming. At the source level, Par4All [149] uses a non-polyhedral approach based on abstract interpretation which enables powerful inter-procedural analyses. Polyhedral compilation techniques have first been used for GPU code generation by Baskaran [150] and have later been improved as part of the R-Stream compiler [151]. An alternative mapping approach that relies on the counting of integer points to tightly fill shared memory caches has been proposed by Baghdadi et. al. [152], but the resulting memory accesses have been shown to be too costly in practice. With CUDA-CHiLL[153] generating GPU codes based on user provided scripts has been proposed. The state-of-the-art in general-purpose polyhedral source-to-source compilation is ppcg [154, 155], which provides effective GPU mappings that exploit shared and private memory. The main focus of these tools is the generation of GPU kernel code from code following strict programming rules [156]. Offloading from within a compiler has been first proposed by GRAPHITE-OpenCL [40] which allowed for the static mapping of parallel loops, but did not considering inter SCoP data reuse. In the context of Polly [157], Kernelgen [158] proposed a new approach in which it aims to push as much execution as possible on the GPU, using the CPU only for system calls and other program parts not suitable for the GPU. The final executables are shipped with a sophisticated run-time system that supports just-in-time accelerator mapping, parameter specialization and provides a page-locking based run-time system to move data between devices. Damschen et. al. [159] introduce a client-server system to automatically offload compute kernels to a Xeon-Phi system. These approaches are based on an early version of Polly (or GRAPHITE), without support for non-affine subregions, modulo expressions, schedule trees or delinearization and are consequently limited in the kind of SCoPs they can detect. Finally, with Hexe [160] a modular data management and kernel offloading system was proposed which does to our understanding not take advantage of polyhedral device mapping strategies. The presented approaches aim for general-purpose accelerator mapping and do not consider the identification and transformation of algorithm specific constructs.

5.6 Summary and Conclusion

We presented the Open CIM Compiler and TDO-CIM. The former enables to program novel in-memory devices starting from a productivity-oriented language and exploits progressive lowering to have a reliable mapping. OCC, however, is not directly applicable to legacy code. Thus, we propose TDO-CIM, enabling CIM acceleration by transparently detecting and offloading computational motifs on general-purpose code.
In the future, we would like to have a common infrastructure that allows both progressive lowering and progressive raising. Furthermore, we would like to extend OCC to (1) detect and offload a broader set of operations (both logical and arithmetic)

that can be accelerated using the in-memory programming model, (2) to provide, over time, a catalog of portable optimizations to benefit multiple in-memory technologies.

6
MLIR Building Blocks

In this part of the thesis, we present a set of MLIR building blocks used to enter the MLIR's lowering pipeline starting from general-purpose C and C++ languages. We start by introducing PET-to-MLIR, a tool based on *pet* and *isl*; we then provide a more general and robust solution based on the Clang frontend: Polygeist.

6.1 PET-to-MLIR

PET-to-MLIR, also known as MET, is a simple MLIR code generator that enables entering the Affine dialect from C or C++ code. It relies on *pet* and *isl* library. The former enables the extraction of polyhedral snippets from C code fragments. The extracted code fragment is called a static-control part or *SCoP* for short. The latter is a mathematical library to model and manipulate piece-wise quasi-affine expressions and conditions.
For each *SCoP*, PET-to-MLIR emits an `mlir::FuncOp` with a void signature. The inputs of the function match with the input and output of the *SCoP*. To identify the input, we use the *SCoP*'s array list. The array list keeps track, for each array reference, of the following information: (1) the extent (e.g., the size of the array), (2) element type (e.g., float or double). (3) The set of constraints on the array parameters to ensure that it has a valid size. (4) Two additional flags: declared and exposed. The former tells us if an array is declared within the *SCoP*. The latter if the array is visible outside the *SCoP*. If an array is marked as exposed, it will be inserted as an input parameter to the function. Whereas, if an array is marked as declared, it will be allocated and deallocated within the function.
A *pet SCoP* also contains a context and a list of statements with line locations. The context defines the parameter values for which the *SCoP* is executed. We do not allow any symbolic constant in the context; if any, the tool will bail out with a warning. The list of statements keeps track of statement information. Specifically, each statement consists of a line number, a domain, a schedule, and a parse tree. The latter reflects the structure of the C statement. In the parse tree, each node corresponds to `pet_expr`. A `pet_expr` carries the type of operation and the arguments to be modeled. An argument can be an access to an array (e.g., a read or write access) or another `pet_expr`. PET-to-MLIR, builds each statement by recursively walking each parse tree and creating the corresponding operation using the builders exposed by the affine dialect.

This Chapter is based on: K. Komisarczyk et al., "PET-to-MLIR: A polyhedral front-end for MLIR" DSD, 2020 and WS. Moses et al., "Polygeist; Affine C in MLIR" IMPACT, 2021.

To emit control flow operations, PET-to-MLIR walks the *isl* AST. For each node, PET-to-MLIR emits the corresponding operation in the affine dialect. In more details, for each `isl_ast_node_for` PET-to-MLIR emits an `Affine::ForOp`. Currently, we can handle loops that count upward, downward and triangular ones. The for loop needs to be in the form `for (int i = init(n); condition(n, i); i+=s)` where `n` and `s` are numbers. For each `isl_ast_node_user` PET-to-MLIR generates a statement. Specifically, the statement id is extracted from the AST node and the statement look-up in the statement list. We emit the statement operations by walking the parse tree. `isl_ast_node_block` are handled in the same way as `isl_ast_node_user` with the difference that they contains multiple statements. Finally, *isl* AST nodes of type `isl_ast_node_if` are not yet handled.

Limitations Although PET-to-MLIR is already able to handle the majority of the Polybench benchmarks suite, it is still relatively new and under active development. At the time of this writing (git commit `d38f9e6`), the tool comes with limitations that, however, do not preclude its usage. These limitations are: (1) If and else construct are not yet handled in the code generation. Currently, when an if condition is detected in the code fragment to be translated, the tool exists with a warning. (2) Symbolic bounds are not yet handled, and for now, we require all the loop bounds to be statically known. (3) External function calls in the code fragment are not allowed, and the tool bails-out if a call is detected. (4) Other operations such as division, as well as, constant accesses to arrays are not handled yet. (5) Line locations are not tracked.
Additionally, using *pet*'s representation limits the usability as PET-to-MLIR cannot interface with non-polyhedral code such as initialization, verification, or printing routines. These limitations have been overcome in Polygeist.

6.2 Polygeist

Polygeist is a new compilation flow that connects MLIR infrastructure to cutting edge polyhedral optimization tools. It consists of a C and C++ frontend capable of converting a broad range of existing codes into MLIR suitable for polyhedral transformation and a bi-directional conversion between MLIR and OpenScop exchange format.
Polygeist's pipeline consits of four main components (Figure 6.1):

- frontend that allows entering MLIR at the SCF loops level from C or C++ code (Section 6.2.1);
- preprocessing step within MLIR that raises to the Affine dialect (Section 6.2.2);
- polyhedral scheduler of the Affine parts of the program *via* a round-trip to and from OpenSCoP and running Pluto transformations, controlled by the new statement splitting heuristic;
- backend that runs postprocessing MLIR optimizations (section 6.2.3) and final lowering to an executable.

In this thesis, we mainly focus on the front-end and post-processing transformations in MLIR as these are the main pieces of the infrastructure I contributed the most.

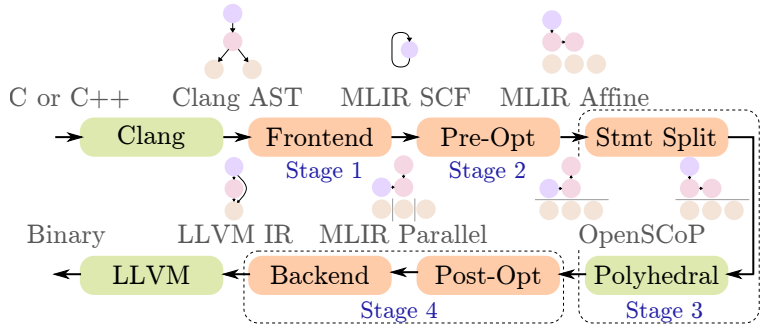


Figure 6.1: Polygeist flow consists of 4 stages. The frontend traverses Clang AST to emit MLIR SCF dialect, which is raised to the Affine dialect and pre-optimized. The IR is then processed by a polyhedral scheduler before post-optimization and parallelization. Finally, it is translated to LLVM IR for further optimization and binary generation by LLVM.

C type	LLVM IR type	MLIR type
`int`	`i32 (on machine X)`	`i32 (on machine X)`
`intNN_t`	`iNN`	`iNN`
`uintNN_t`	`iNN`	`uiNN`
`float`	`float`	`f32`
`double`	`double`	`f64`
`ty *`	`ty *`	`memref<?xty>`
`ty &`	`ty *`	`memref<1xty>`
`ty **`	`ty **`	`memref<memref<?xty>>`
`ty[N][M]`	`[N x [M x ty]]*`	`memref<NxMxty>`

Figure 6.2: Type correspondence between C, LLVM IR and MLIR types.

6.2.1 Frontend

Polygeist relies on the Clang AST to emit MLIR IRs. It thus avoids reimplementing parsing and language-level semantic analysis and handles modern C and C++ features. Polygeist creates a recursive symbol table data structure to look up the correct variable for a given scope. Polygeist lazily registers all global variables and functions found in the AST to its symbol table before generating any code. Polygeist then traverses the call graph from a given entry function (`main` by default), creating and defining MLIR functions as necessary.

Control Flow & High Level Information In contrast to traditional compiler pipelines, targeting a branch-based IR, Polygeist leverages the existing of high-level operations such as `scf.while` (a looping construct) and `scf.if` (a conditional construct) within MLIR's SCF dialect to preserve the high-level structure of the source code. C-level `continue` and `break` constructs are handled by introducing signal variables and checking them before each operation that follows original constructs. Furthermore, within a `#pragma scop`, Polygeist assumes that the program is affine and uses an `affine.for` to represent loops directly.

Types & Polygeist ABI While emitting operations, Polygeist must decide how to represent C or C++ types within MLIR. For primitive types such as `int` or `float`, Polygeist emits an MLIR variant of that type with the same width as would be used within LLVM/Clang. This allows Polygeist to keep the same Application Binary Interface (ABI) as code compiled by a normal C or C++ compiler when calling a function with only primitive types. On the other hand, for pointer, reference and array types, Polygeist uses `memref` type (Figure 6.2). This allows Polygeist to preserve more of the structure available within the original program (e.g., multi-dimensional arrays) and enables interaction with MLIR's high-level memory operations.
This represents a breaking change to the C ABI for any functions with pointer arguments. Polygeist addresses this by providing an attribute for function arguments and allocations to use a C-compatible pointer type rather than `memref`, applied by default to external functions such as `strcmp` and `scanf`. When a pointer-ABI function with a `memref`-ABI argument, Polygeist generates wrapper code that calls the function with a C ABI-compatible pointer and ensures the correct result. Listing 36 shows an example demonstrating how the Polygeist and C ABI may interact for a small program.
When allocating and deallocating memory, this difference in ABI becomes significant. This is because allocating several bytes of an array with `malloc` then casting to a `memref` will not result in legal code (as `memref` itself may not be implemented with a raw pointer). Thus, Polygeist identifies calls to allocation and deallocation functions and replaces them with legal equivalents for `memref`.
Functions and global variables are emitted using the same name used by the C or C++ ABI. This ensures that all external values are loaded correctly, and multi-versioned functions (such as those generated by `C++` templates or overloading) have distinct names and definitions.

Instruction Generation For most instructions, Polygeist directly emits an MLIR operation corresponding to the equivalent C operation (`addi` for integer add, `call` for function call, etc.). For some special instructions such as a call to `pow`, Polygeist chooses to emit a specific MLIR operation in the Math dialect, instead of a call to an external function (defined in libm). This permits such instructions to be better analyzed and optimized within MLIR.
Operations that involve memory or pointer arithmetic require additional handling. MLIR does not have a generic pointer arithmetic instruction; instead, it requires that `load` and `store` operations contain all of the indices being looked up. This presents issues for operations that perform pointer arithmetic. To remedy this, we use a temporary `subindex` operation for `memref`'s that curry the index. A subsequent optimization pass within Polygeist, forwards the indices in a `subindex` to any `load` or `store` which uses them.

Local Variables Local variables are handled by allocating a `memref` on stack at the top of a function. This permits the desired semantics of C or C++ to be implemented with relative ease. However, as many local variables and arguments contain `memref` types, this immediately results in a `memref` of a `memref`—a hindrance for most MLIR optimizations as it is illegal outside of Polygeist. As a remedy,

we implement a heavyweight memory-to-register (mem2reg) transformation pass that eliminates unnecessary loads, stores, and allocations within MLIR constructs. Empirically this eliminates all `memref`s of `memref` in the Polybench suite.

6.2.2 Raising to Affine

The translation from C or C++ to MLIR directly preserves high-level information about loop structure and n-D arrays, but does not generate other Affine operations. Polygeist subsequently raises memory, conditional, and looping operations into their Affine dialect counterparts if it can prove them to be legal affine operations. If the corresponding frontend code was enclosed within `#pragma scop`, Polygeist assumes it is legal to raise all operations within that region. Any operations which are not proven or assumed to be affine remain untouched. We perform simplifications on affine maps to remove loops with zero or one iteration and drop branches of a conditional with a condition known at compile time.

Memory operations and loop bounds To convert an operation, Polygeist replaces its bound and subscript operands with identity affine maps (`affine_map<()[s0]->(s0)>[%bound]`). It then folds the operations computing the map operands, e.g., `addi`, `muli`, into the map itself. Values that are transitively derived from loop induction variables become map dimensions and other values become symbols. For example, `affine_map< ()[s0]->(s0)>[%bound]` with `%bound = addi %N, %i`, where `%i` is an induction variable, is folded into `affine_map<(d0)[s0]->(s0 + d0)>(%i)[%N]`. The process terminates when no operations can be folded or when Affine value categorization rules are satisfied.
Lifting loop to `affine.for` happens as a result of multiple canonicalization passes. Listing 35 shows how a simple sum reduction loop is raised to Affine from the SCF dialect.

Conditionals Conditional operations are emitted by the frontend for two input code patterns: `if` conditions and ternary expressions. The condition is transformed by introducing an integer set and by folding the operands into it similarly to the affine maps, with in addition `and` operations separating set constraints and `not` operations inverting them (`affine.if` only accepts ≥ 0 and $= 0$ constraints). Polygeist processes nested conditionals short-circuit semantics, in which the following conditions are checked within the body of the preceding conditionals, produced to reflect C by hoisting conditions outside the outermost conditional when legal and replacing them with a boolean operation or a `select`. This is always legal within `#pragma scop`.
Conditionals emitted for ternary expressions often involve memory loads in their regions, which prevent hoisting due to side effects. We reuse our mem2reg pass to replace those to equivalent earlier loads when possible to enable hoisting. Empirically, this is sufficient to process all ternary expressions in the Polybench/C suite. Otherwise, ternary expressions would need to be packed into a single statement by the downstream polyhedral pass.

```
int reduction(int A[10]) {
  int sum = 0;
  for(i=0; i<10; i++)
    sum += A[i];
  return sum;
}
```

⇓

```
func @reduction(%arg0: memref<?xi32>) -> i32 {
    %c0_i32 = constant 0 : i32
    %c10_i32 = constant 10 : i32
    %c1_i32 = constant 1 : i32
    %0:2 = scf.while (%arg1 = %c0_i32, %arg2 = %c0_i32) : (i32,
        i32) -> (i32, i32) {
      %1 = cmpi slt, %arg1, %c10_i32 : i32
      scf.condition(%1) %arg2, %arg1 : i32, i32
    } do {
    ^bb0(%arg1: i32, %arg2: i32):  // no predecessors
      %1 = index_cast %arg2 : i32 to index
      %2 = memref.load %arg0[%1] : memref<?xi32>
      %3 = addi %arg1, %2 : i32
      %4 = addi %arg2, %c1_i32 : i32
      scf.yield %4, %3 : i32, i32
    }
    return %0#0 : i32
  }
```

⇓

```
func @reduction(%arg0: memref<?xi32>) -> i32 {
    %c0_i32 = constant 0 : i32
    %c10 = constant 10 : index
    %c0 = constant 0 : index
    %c1 = constant 1 : index
    %0 = scf.for %arg1 = %c0 to %c10 step %c1 iter_args(%arg2 =
        %c0_i32) -> (i32) {
      %1 = memref.load %arg0[%arg1] : memref<?xi32>
      %2 = addi %arg2, %1 : i32
      scf.yield %2 : i32
    }
    return %0 : i32
  }
```

⇓

```
 func @reduction(%arg0: memref<?xi32>) -> i32 {
    %c0_i32 = constant 0 : i32
    %0 = affine.for %arg1 = 0 to 10 iter_args(%arg2 = %c0_i32) ->
        (i32) {
      %1 = affine.load %arg0[%arg1] : memref<?xi32>
      %2 = addi %arg2, %1
      affine.yield %2 : i32
    }
    return %0 : i32
  }
```

Listing 35: Raising loops from the SCF dialect to the Affine one. Starting from a CFG, loops are fist raised to an `scf.while` operations, then `scf.while` are rewritten as `scf.for` and finally to `affine.for`.

```
void setArray(int N, double val, double* array) {...}
int main(int argc, char** argv) {
  ...
  cmp = strcmp(str1, str2)
  ...
  double array[10];
  set_array(10, array)
}
```

```
func @setArray(%N: i32, %val: f64
               %array: memref<?xf64>) {
  %0 = index_cast %N : i32 to index
  affine.for %i = 0 to %0 {
    affine.store %val, %array[%i] : memref<?xf64>
  }
  return
}

func @main(%argc: i32,
           %argv: !llvm.ptr<ptr<i8>>) -> i32 {
  ...
  %cmp = llvm.call @strcmp(%str1, %str2) :
           (!llvm.ptr<i8>, !llvm.ptr<i8>) -> !llvm.i32
  ...
  %array = memref.alloca() : memref<10xf64>
  %arraycst = memref.cast %array : memref<10xf64> to
    memref<?xf64>
  call @setArray(%N, %val, %arraycst) :
           (i32, f64, memref<?xf64>) -> ()
}
```

Listing 36: Example demonstrating Polygeist Application Binary Interface (ABI). For functions expected to be compiled with Polygeist such as `setArray`, pointer arguments are replaced with `memref`'s. For functions that require external calling conventions (such as `main`/`strcmp`), we fall back to using `llvm.ptr` and generating conversion code where appropriate.

6.2.3 Post-Transformations and Backend

Polygeist allows one to operate on both quasi-syntactic and SSA level, enabling analyses and optimizations that are extremely difficult, if not impossible, to perform at either level in isolation. In addition to statement splitting, we propose two techniques that demonstrate the potential of Polygeist. such optimizations.

Loops with Carried Values (Reductions) Polygeist leverages MLIR's first-class support for loop-carried values to detect, express and transform reduction-like loops. This support does not require source code annotations, unlike source-level tools [161] that use annotations to enable detection, nor complex modifications for parallel code emission, unlike Polly [162], which suffers from LLVM missing first-class parallel constructs. We do not modify the polyhedral scheduler either, relying on post-processing for reduction parallelization, including outermost parallel reduction loops.
The overall approach follows the definition proposed in [163] with adaptations to MLIR's region-based IR, and is illustrated in Listing 37. Polygeist identifies memory locations modified on each iteration, i.e. `load`/`store` pairs with loop-invariant subscripts and no interleaving aliasing `store`s, by scanning the single-block body of the loop. These are transformed into *loop-carried values* or secondary induction variables, with the `load`/`store` pair lifted out of the loop and repurposed for reading the initial and storing the final value. Loop-carried values may be updated by a chain of side effect-free operations in the loop body. If this chain is known to be associative and commutative, the loop is a *reduction*. Loop-carried values are detected even in absence of reduction-compatible operations. Loops with such values contribute to mem2reg, decreasing memory footprint, but are not subject to parallelization.

6.2.4 Evaluation

Our evaluation has two goals: (1) we want to demonstrate that the code produced by Polygeist without additional optimizations does not have any inexplicable performance differences than a state-of-the-art compiler like Clang. (2) We explore how Polygeist's internal representation can support a mix of affine and SSA-based transformation in the same compilation flow, and evaluate the potential benefits compared to existing source and compiler-based polyhedral tools.

Experimental Setup We ran our experiments on an AWS `c5.metal` instance with hyper-threading and Turbo Boost disabled. The system used is Ubuntu 20.04 running on an Intel Xeon Platinum 8275CL CPU at 3.0 GHz with 1.5, 48, 71.5 MB L1, L2, L3 cache, respectively, and 256 GB RAM. We ran all 30 benchmarks from PolyBench, using the "EXTRALARGE" dataset. Pluto is unable to extract *SCoP* from the `adi` benchmark. We ran a total of 5 trials for each benchmark, taking the execution time reported by PolyBench; the median result is taken unless stated otherwise. Every measurement or result reported in the following sections refers to double-precision data. All experiments were run on cores 1-8, which ensured that all threads were on the same socket and did not potentially conflict with processes scheduled on core 0.

```
affine.for %i = ... {
  // Reduction into r1[0]
  %1 = affine.load %r1[0]
  %5 = addi %1, %2
  affine.store %5, %r1[0]
  // Loop-dependent load
  %10 = affine.load %r2[%i]
  %15 = addi %10, %2
  // Inteleaving store
  %20 = affine.load %r2[0]
  affine.store %21, %r2[0]
  %25 = addi %20, %2
  // May have side effects
  %30 = affine.load %r3[0]
  call @f(%30, %2)
}
```

⇒

```
%init = affine.load %r1[0]
%red = affine.for %i = ...
 iter_args(%arg = %init) {
  // Reduction accumulation
  %5 = addi %arg, %2
  // Loop-dependent load
  %10 = affine.load %r2[%i]
  %15 = addi %10, %2
  // Inteleaving store
  %20 = affine.load %r2[0]
  affine.store %21, %r2[0]
  %25 = addi %20, %2
  // May have side effects
  %30 = affine.load %r3[0]
  call @f(%30, %2)
  // Yield accumulated
  affine.yield %5
}
affine.store %red, %r1[0]
```

Listing 37: Polygeist detects memory locations accessed in all loop iterations, e.g. reduction accumulators such as `%r1[0]` (left) and transforms them to loop-carried values (secondary induction variables), except when computed with side-effects, interleaved stores or by non-associative/commutative operations (right).

In all cases, we use two-stage compilation: (i) using `clang` at `-O3` excluding unrolling and vectorization; or Polygeist to emit LLVM IR from C; (ii) using `clang` at `-O3` to emit the final binary. As several optimizations are not idempotent, a second round of optimization can potentially significantly boost (and rarely, hinder) performance. This is why we chose to only perform vectorization and unrolling at the last optimization stage. Since Polygeist applies some optimizations at the MLIR level (e.g., mem2reg), we compare against the two-stage compilation pipeline as a more fair baseline (`Clang`). We also evaluate a single-stage compilation to assess the effect of the two-stage flow (`ClangSing`).

Baseline Performance Polygeist must generate code with runtime *as close as possible* to that of existing compilation flows to establish a solid baseline. In other words, Polygeist should *not introduce overhead nor speedup* unless explicitly instructed otherwise, to allow for measuring the effects of additional optimizations. We evaluate this by comparing the runtime of programs produced by Polygeist with those produced by Clang at the same commit (Apr 2021)[1]. Figure 6.3 summarizes the results with the following flows:

- `Clang`: a compilation of the program using Clang, when running two stages of optimization;
- `ClangSing`: a compilation of the program using Clang, when running one stage of optimization;
- `MLIR-Clang`: a compilation flow using the Polygeist frontend and preprocessing optimizations within MLIR, but not running polyhedral scheduling nor

[1]LLVM commit `20d5c42e0ef5d252b434bcb610b04f1cb79fe771`.

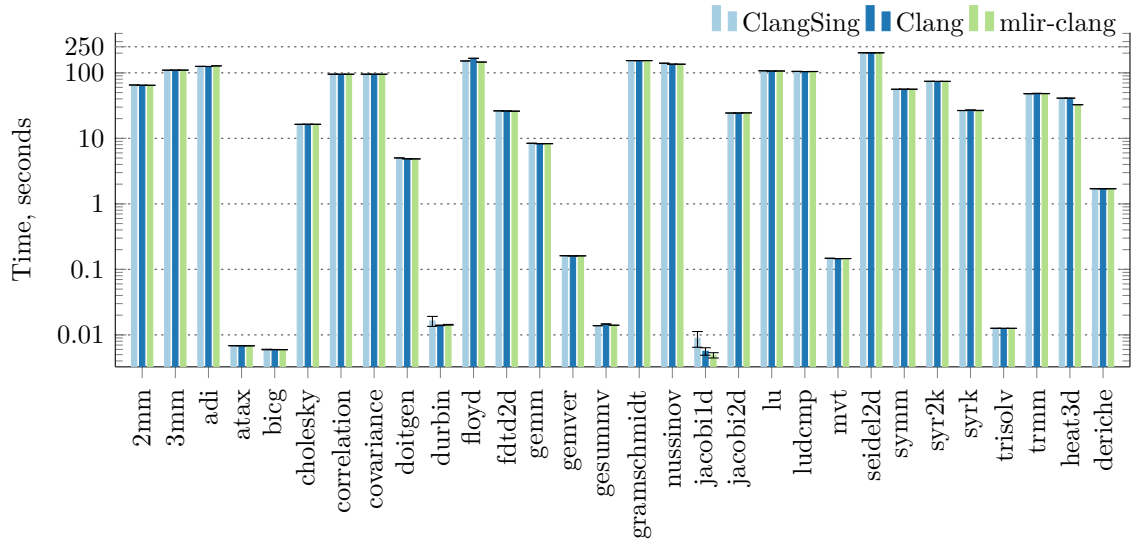


Figure 6.3: Mean and 95% confidence intervals (log scale) of program run time across 5 runs of Polybench in `Clang`, `ClangSing` and `MLIR-Clang` configurations, lower is better. The run times of code produced by Polygeist without optimization is comparable to that of Clang. No significant variation is observed between single and double optimization. Short-running `jacobi-1d` shows high intra-group variation.

postprocessing.
No significant variation are observed between `Clang`, `ClangSing` and `MLIR-Clang`.

Reduction Detection and Parallelization Polygeist uses its reduction detection pass to identify and parallelize reductions. We demonstrate this transformation using the Durbin benchmarks as case study. For the relatively small input run by default, $N = 4000$ iterations inside another sequential loop with N iterations, the overall performance decreases. We hypothesize that the cost of creating parallel threads and synchronizing them outweighs the benefit of the additional parallelism and test our hypothesis by increasing N. Considering the results in Figure 6.4, one observes that Polygeist starts yielding speedups (> 1) for $N \geq 16000$ whereas Polly only does so at $N \geq 224000$, and to a much lesser extent: $6.62\times$ vs $1.01\times$. Without reduction parallelization, Polygeist follows the same trajectory as Polly. Pluto fails to parallelize any innermost loop and shows no speedup. This evidences in favor of our hypothesis and highlights the importance of being able to parallelize reductions.

6.3 Related Work

PET-to-MLIR is a MLIR frontend, while Polygeist can both serve as an MLIR frontend or polyhedral exporter/importer to and from the OpenScop representation. Let's then discuss current approaches.

Polyhedral extractors The polyhedral model has been on the cutting edge of compiler research for several decades, resulting in the creation of many tools [164].

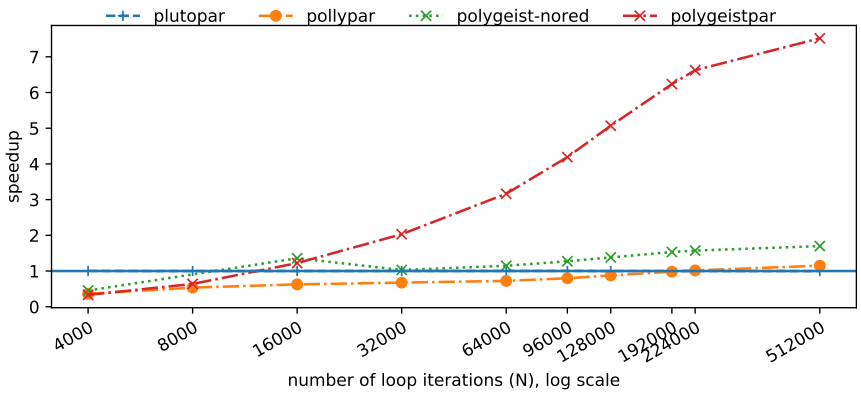


Figure 6.4: Reduction parallelization allows `PolygeistPar` to produce larger speedups and at smaller sizes than `PollyPar` and `PolygeistPar` without reduction support (`Polygeist-nored`). `PlutoPar` fails to parallelize leading to no speedup.

Polly [22] and Graphite [21] enable polyhedral optimizations in LLVM and GCC, respectively, by raising from the low-level IR to higher and richer polyhedral representations. Other proprietary compilers, such as IBM XL [38] and R-Stream [39], use polyhedral techniques and thus rely on extractor tool, but being proprietary few documentation is available. However, extracting the polyhedral model from low-level IR is not be the best approach for source-to-source optimizers such as Pluto [42] and PoCC [43], since it is difficult or even impossible to relate low-level code to input program. Source-level parsers such as Clan [41] and *pet* [53] aim at providing a convenient way to extract polyhedral representation directly from the source code. Polygeist falls into this category and aims to enable MLIR to leverage the decades of research in the polyhedral model by lifting C code to the Affine dialect. Besides, we sometimes need polyhedral optimization in MLIR before LLVM IR is produced to adopt MLIR-specific passes, e.g., GPU mapping, which further justifies the necessity of Polygeist even Polly exists.

MLIR Frontends Since the adoption of MLIR under the LLVM umbrella, several frontends have been created for generating MLIR from domain-specific languages. Teckyl [99] brings *Tensor Comprehensions* [3], a productivity-orientated language to express computation between tensors, to MLIR's Linalg dialect. Flang—the LLVM's Fortran frontend — enables models Fortran specific constructs (i.e., dispatch table) using the FIR dialect [165]. COMET, a domain-specific compiler, for chemistry compilation enters the MLIR lowering pipeline using a domain-specific frontend from a tensor-based language [166]. NPComp aims at providing the necessary infrastructure to compile numerical Python programs taking advantage of the MLIR infrastructure. Work is on progress to provide a PyTorch frontend [167]. PET-to-MLIR converts a subset of polyhedral C code to MLIR's Affine dialect by parsing *pet*'s internal representation. In addition to currently not handling specific constructs (ifs, symbolic bounds, and external function calls), parsing *pet*'s representation limits the frontend's usability as it cannot interface with non-polyhedral code such as initialization, verification, or printing routines [168]. In contrast, Polygeist generates MLIR from non-polyhedral code as well (though not necessarily in the

Affine dialect). CIRCT is a new project under the LLVM umbrella that aims to apply MLIR development methodology to the electronic design automation industry [169].

6.4 Summary and Conclusion

We started by introducing PET-to-MLIR, an MLIR code generator based on *isl* and *pet*, which allows entering the Affine dialect starting from C and C++ code. Albeit PET-to-MLIR is capable of handling most of the Polybench benchmark suite, in retrospect, the choice of building the tool on *pet* was a limiting one, but we learn from our failure. To overcome PET-to-MLIR limitations, we drop *pet* and *isl* dependencies, and we emit MLIR constructs by walking the Clan AST directly. This solution (Polygeist) enables us to handle also non-affine constructs. A subsequent raising pass enables us to enter the Affine dialect from the SCF for loops that respect Affine constraints. Polygeist does not only allow us to enter the MLIR lowering pipeline but allows us to reconnect decades of research in the polyhedral model with the novel MLIR infrastructure.

7

Conclusion and Future Work

Previous chapters demonstrated the need to raise the level of abstractions when targeting modern and heterogeneous hardware. Today's solution relies on introducing a new domain-specific language with an accompanying compiler or using vendor-optimized libraries. We believe that both solutions are not optimal in the long term. Domain-specific languages require programmers to retarget their application in a time-consuming process with no guarantees that the language will support new hardware in the future. Vendor-optimized libraries, on their part, can just be restricted to few fundamental basic-building blocks. To sidestep this problem, we envisioned a compiler-based solution, where it is the task of the compiler to match new (and legacy) software to new hardware. Consequently, rather than asking programmers to rewrite their code in a new DSL or using a new API, it is now up to the compiler's job to perform this task.

7.1 Thoughts on What We Achieved

We now discuss how each contribution of this thesis pushes toward our vision of having a mechanism that allows matching new hardware to existing and future software without the need of developing a new compiler or language every time.

Bringing Domain-specific Compilation in General-purpose Flows Chapter 3 and Chapter 4 introduced Loop Tactics and Multi-level Tactics, respectively. With Loop Tactics we brought domain-specific optimization directly in general-purpose flows by providing a way to express composable program transformations and detect computational motifs based on an internal tree-like program representation. We applied Loop Tactics on two critical domains: Linear-algebra and Stencils, demonstrating the benefit of such a framework in terms of performance (e.g., faster code) and code size reductions. As a result, with Loop Tactics, we reduced the need to design a domain-specific compiler, enabled the optimization of multi-domain applications, and the transparent use of vendor-optimized libraries.

Limitations: Loop Tactics inherits the power and limitations of the polyhedral representation. The mathematical nature of the model allows us to precisely identify complex access patterns, which goes beyond what similar approaches on SSA-based representation can detect. But on the other side Loop Tactics is limited to what can be represented in the model. Writing the matchers and builders further requires knowledge on the schedule tree representation. The GUI proposed in Chapter 3 is a first step in making the framework more accessible to non-polyhedral experts.

The advent of multi-level IR representation promises to lower the design and cost for domain-specific compilers by providing a reusable, extensible, and non-opinioned framework for expressing domain-specific and high-level abstractions directly in the IR. But, while such frameworks support the progressive lowering of high-level representations to low-level IR, they do not raise in the opposite direction. Thus, the entry point into the compilation pipeline defines the highest level of abstraction for all subsequent transformations, limiting the set of applicable optimizations, particularly for general-purpose languages that are not semantically rich enough to model the required abstractions. Thus general-purpose languages are left behind and cannot benefit from more aggressive optimizations at higher-level of abstractions.
With Multi-level Tactics, we want to establish progressive raising as a complementary approach to the progressive lowering in multi-level IR, thus enabling a path from low-level abstractions to higher-level ones. We demonstrated three raising paths at different abstractions: Within the Affine dialect, from the Affine dialect to the Linalg one, and a case for a more progressive raising by lifting to Linalg and then apply another raising pass to detect chains of matrix multiplications. With abstraction raising, we enable to raise the abstraction levels of languages and compilers without leaving behind general-purpose software or forcing developers to rewrite their applications.

Limitations: Multi-level Tactics is a first step in bringing progressive raising in multi-level IR compilers. Only some rising paths are available today, and more research needs to be done in defining a specification language for matchers and builders that generalize over multiple domains. Furthermore, Multi-level Tactics (and Loop Tactics) rely on proceeding optimizations passes to canonicalize the intermediate code. The relation between normalization and optimization poses interesting research questions that were resolved pragmatically in his research. For example, by running the affine scheduler to normalize the schedule tree or run pre-canonicalization passes in MLIR (e.g., loop distributions) to simplify subsequent matching.

Overall, both Loop Tactics and Multi-level Tactics enhance a general-purpose compiler with a mechanism to detect computational motifs that may belong to different domains and provide builders to reuse existing optimizations or implementing new ones. We thus reduce the need for new DSLs and provide a compiler-based solution to match software to new hardware.

Programming Novel Hetereogenous Devices The silicon industry is moving toward specialization to counterbalance the end of Moore's law and the diminishing of the Dennard scaling. However, such a diverse landscape reduces the effectiveness of compilers; consequently, the absolute maximum performance on novel accelerators is obtained by provided libraries. Unfortunately, the usage of such libraries is left onto the programmer, who needs to map manually specific kernels on the accelerator, reducing application readiness and limiting the widespread of these novel accelerators. With OCC and TDO-CIM, we enable an automatic compilation flow. The former exploits progressive lowering in a multi-level compiler; the latter allows transparent acceleration on legacy code.

7.2 Thinking On the Next Steps

Directions for future work include the specification of more computational idioms, usability improvements of the specification languages. Moreover, matcher specifications could eventually be generated from examples, eliminating the difficulty of writing them manually.

Phase-ordering for Canonicalization, and Transformation Effects Both Loop Tactics and Multi-level Tactics rely on preceding optimizations passes to normalize the intermediate code generated from the user programs. The relation between normalization and optimizations in the compiler poses interesting research questions. For example, Loop Tactics run after the affine scheduler. The affine scheduler allows us to canonicalize the tree by discovering permutability and parallelism properties, coalescing input bands into a single band whenever possible. While this approach was effective in practice, the affine scheduler was not designed for this purpose. Another downside is that the scheduling algorithm is unaware of the higher-level information extracted by the matchers.
Interaction with existing optimizations also requires more thinking. In particular, the effect of rescheduling after matching to re-evaluate fusion decisions or using the matched primitives to guide the scheduler.

Generating Specifications from Examples The macher specification requires knowledge of the underneath abstraction. For example, in Loop Tactics, a programmer needs to be aware of the schedule tree structure and the different node types when writing a matcher. We can adopt techniques developed for code refactoring (e.g., clang-query or clang-tidy). For example, we can bootstrap the development of matchers by inferring them automatically by providing a "before" and "after", written in a general-purpose language, for instance. Perfect fidelity inference is unnecessary: Producing an over-specified set of matchers can still boost productivity, allowing construction of final matchers by deletion rather than creation.
On the builder's side, an interesting direction would be to use machine learning techniques to adapt the builder parameters (e.g., tile size) to the target architecture to extract the best possible performance.

Higher-level Specification Languages Multi-level Tactics introduced a language for the specification of matchers and builders based on *Tensor Comprehensions*. While *Tensor Comprehensions* is a perfect fit for machine-learning, or more general, linear-algebra domain benchmarks, it may not be expressive for other domains (e.g., stencils). Future work could investigate how to generalize the specification language over multiple different domains.
Furthermore, knowledge of the compiler internal is still required to write correct specifications as matcher and replacement are not verified for equivalence in any way.

Runtime Instead of Compile Time Today, Loop Tactics and Multi-level Tactics require recompilation; a more dynamic environment would be preferrable. The

Pattern Description Language in MLIR would be a nice abstraction to provide our matchers and builders infrastructure.

Bibliography

[1] W. contributors. (2021) M1 chip. [Online]. Available: https://en.wikipedia.org/wiki/Apple_M1

[2] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*. USA: Addison-Wesley Longman Publishing Co., Inc., 1986.

[3] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Oct. 2019. [Online]. Available: https://doi.org/10.1145/3355606

[4] T. Grosser, "A decoupled approach to high-level loop optimization : tile shapes, polyhedral building blocks and low-level compilers," Theses, Université Pierre et Marie Curie - Paris VI, Oct. 2014. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01144563

[5] D. Amodei and D. Hernandez. (2018) Ai and compute. [Online]. Available: https://openai.com/blog/ai-and-compute/

[6] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers *et al.*, "In-datacenter performance analysis of a tensor processing unit," in *Proceedings of the 44th annual international symposium on computer architecture*, 2017, pp. 1–12.

[7] M. O'Boyle. Rethinking the role of the compiler in a heterogeneous world. https://www.sigarch.org/rethinking-the-role-of-the-compiler-in-a-heterogeneous-world/. [Online; accessed 05-04-2021].

[8] A. K. Sujeeth, K. J. Brown, H. Lee, T. Rompf, H. Chafi, M. Odersky, and K. Olukotun, "Delite: A compiler architecture for performance-oriented embedded domain-specific languages," *ACM Trans. Embed. Comput. Syst.*, vol. 13, no. 4s, pp. 134:1–134:25, Apr. 2014. [Online]. Available: http://doi.acm.org/10.1145/2584665

[9] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," in *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 1989, pp. 25–35.

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

Association for Computing Machinery, 2013, p. 519–530. [Online]. Available: https://doi.org/10.1145/2491956.2462176

[11] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, p. 323–337, Dec. 1992. [Online]. Available: https://doi.org/10.1145/161494.161501

[12] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/

[13] T. Developers. Tensorrt. [Online]. Available: https://github.com/NVIDIA/TensorRT

[14] nGraph developers. ngraph. [Online]. Available: https://github.com/NervanaSystems/ngraph

[15] C. M. developers. Core ml. [Online]. Available: https://github.com/apple/coremltools

[16] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "Mlir: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2021, pp. 2–14.

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

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

[19] P. Feautrier and C. Lengauer, *Polyhedron Model*. Boston, MA: Springer US, 2011, pp. 1581–1592. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_502

[20] U. Bondhugula, S. Dash, O. Gunluk, and L. Renganarayanan, "A model for fusion and code motion in an automatic parallelizing compiler," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sept 2010, pp. 343–352.

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

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

[23] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "Tensor

comprehensions: Framework-agnostic high-performance machine learning abstractions," *CoRR*, vol. abs/1802.04730, 2018. [Online]. Available: http://arxiv.org/abs/1802.04730

[24] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 429–443, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694364

[25] M.-W. Benabderrahmane, L.-N. Pouchet, A. Cohen, and C. Bastoul, "The polyhedral model is more widely applicable than you think," in *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, ser. CC'10/ETAPS'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 283–303. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-11970-5_16

[26] W. Pugh, "The omega test: A fast and practical integer programming algorithm for dependence analysis," in *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Nov 1991, pp. 4–13.

[27] M. M. Strout, L. Carter, and J. Ferrante, "Compile-time composition of run-time data and iteration reorderings," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 91–102. [Online]. Available: http://doi.acm.org/10.1145/781131.781142

[28] S. Verdoolaege, "isl: An integer set library for the polyhedral model," in *Mathematical Software – ICMS 2010*, K. Fukuda, J. v. d. Hoeven, M. Joswig, and N. Takayama, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 299–302.

[29] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott, "The omega library interface guide," USA, Tech. Rep., 1995.

[30] V. Loechner, "Polylib: A library for manipulating parameterized polyhedra," 1999.

[31] R. Bagnara, P. M. Hill, and E. Zaffanella, "The parma polyhedra library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems," *Science of Computer Programming*, vol. 72, no. 1, pp. 3–21, 2008, special Issue on Second issue of experimental software and toolkits (EST). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0167642308000415

[32] S. Verdoolaege, O. Zinenko, M. Kudlur, R. Estrin, T. Sun, and H. Kamepalli, "A templated c++ interface for isl," 2021.

[33] S. Verdoolaege, "Presburger formulas and polyhedral compilation," 2016. [Online]. Available: $$Uhttps://lirias.kuleuven.be/retrieve/361209$$Dtutorial[freelyavailable]

[34] R. M. Karp, R. E. Miller, and S. Winograd, "The organization of computations for uniform recurrence equations," *J. ACM*, vol. 14, no. 3, p. 563–590, Jul. 1967. [Online]. Available: https://doi.org/10.1145/321406.321418

[35] L. Lamport, "The parallel execution of do loops," *Commun. ACM*, vol. 17, no. 2, p. 83–93, Feb. 1974. [Online]. Available: https://doi.org/10.1145/360827.360844

[36] C. Ancourt and F. Irigoin, "Scanning polyhedra with do loops," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP '91. New York, NY, USA: Association for Computing Machinery, 1991, p. 39–50. [Online]. Available: https://doi.org/10.1145/109625.109631

[37] O. Zinenko, "Interactive Program Restructuring," Theses, Université Paris Saclay (COmUE), Nov. 2016. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01414770

[38] U. Bondhugula, S. Dash, O. Gunluk, and L. Renganarayanan, "A model for fusion and code motion in an automatic parallelizing compiler," in *2010 19th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2010, pp. 343–352.

[39] B. Meister, N. Vasilache, D. Wohlford, M. M. Baskaran, A. Leung, and R. Lethin, *R-Stream Compiler*. Boston, MA: Springer US, 2011, pp. 1756–1765. [Online]. Available: https://doi.org/10.1007/978-0-387-09766-4_515

[40] A. Kravets, A. Monakov, and A. Belevantsev, "Graphite-opencl: Automatic parallelization of some loops in polyhedra representation," *GCC Developers' Summit, GCC Developers' Summit*, 2010.

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

[42] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A Practical Automatic Polyhedral Parallelizer and Locality Optimizer," *ACM SIGPLAN Notices*, vol. 43, no. 6, pp. 101–113, 2008.

[43] PoCC, "The polyhedral compiler collection," 2020, Online; accessed on December 2020. [Online]. Available: https://sourceforge.net/projects/pocc/

[44] M. Griebl and C. Lengauer, "The loop parallelizer loopo—announcement," in *Languages and Compilers for Parallel Computing*, D. Sehr, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 603–604.

[45] S. Verdoolaege, "Counting affine calculator and applications," in *First International Workshop on Polyhedral Compilation Techniques (IMPACT 2011), Chamonix, France*, 2011.

[46] T. Grosser, S. Verdoolaege, and A. Cohen, "Polyhedral AST generation is more than scanning polyhedra," *ACM Trans. Program. Lang. Syst.*, vol. 37, no. 4, pp. 12:1–12:50, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2743016

[47] S. Verdoolaege, S. Guelton, T. Grosser, and A. Cohen, "Schedule Trees," in *4th Workshop on Polyhedral Compilation Techniques (IMPACT, Associated with HiPEAC)*, Vienna, Austria, Jan. 2014, p. 9.

[48] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "A practical and fully automatic polyhedral program optimization system," in *ACM SIGPLAN PLDI*, 2008.

[49] W. Pugh and D. Wonnacott, "Static analysis of upper and lower bounds on dependences and parallelism," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 4, pp. 1248–1278, 1994.

[50] L. Bagnères, O. Zinenko, S. Huot, and C. Bastoul, "Opening polyhedral compiler's black box," in *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, ser. CGO '16. New York, NY, USA: ACM, 2016, pp. 128–138. [Online]. Available: http://doi.acm.org/10.1145/2854038.2854048

[51] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly—performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[52] S. Verdoolaege and G. Janssens, "Scheduling for PPCG," Department of Computer Science, KU Leuven, Leuven, Belgium, Report CW 706, Jun. 2017.

[53] S. Verdoolaege and T. Grosser, "Polyhedral extraction tool," in *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, Jan. 2012.

[54] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Trans. Math. Softw.*, vol. 43, no. 2, pp. 12:1–12:18, Aug. 2016. [Online]. Available: http://doi.acm.org/10.1145/2925987

[55] R. Gareev, T. Grosser, and M. Kruse, "High-performance generalized tensor operations: A compiler-oriented approach," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, pp. 34:1–34:27, Sep. 2018. [Online]. Available: http://doi.acm.org/10.1145/3235029

[56] G. D. Smith, *Numerical solution of partial differential equations: finite difference methods*. Oxford university press, 1985.

[57] A. Taflove and S. C. Hagness, *Computational electrodynamics: the finite-difference time-domain method*. Artech house, 2005.

[58] S. Verdoolaege and A. Isoard, "Consecutivity in the isl polyhedral scheduler," 2017.

[59] O. Zinenko, S. Verdoolaege, C. Reddy, J. Shirako, T. Grosser, V. Sarkar, and A. Cohen, "Modeling the conflicting demands of parallelism and temporal/spatial locality in affine scheduling," in *Proceedings of the 27th International Conference on Compiler Construction*. ACM, Feb. 2018, pp. 3–13.

[60] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for gpus," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO '14. New York, NY, USA: ACM, 2014, pp. 66:66–66:75. [Online]. Available: http://doi.acm.org/10.1145/2544137.2544160

[61] T. Henretty, K. Stock, L.-N. Pouchet, F. Franchetti, J. Ramanujam, and P. Sadayappan, "Data layout transformation for stencil computations on short-vector simd architectures," in *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*, ser. CC'11/ETAPS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 225–245. [Online]. Available: http://dl.acm.org/citation.cfm?id=1987237.1987255

[62] S. Markidis, S. W. D. Chien, E. Laure, I. B. Peng, and J. S. Vetter, "Nvidia tensor core programmability, performance precision," in *2018 IEEE Interna-*

tional Parallel and Distributed Processing Symposium Workshops (IPDPSW), May 2018, pp. 522–531.

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

[64] "TVM: An automated end-to-end optimizing compiler for deep learning."

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

[66] M. Kong and L.-N. Pouchet, "Model-driven transformations for multi-and many-core cpus," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2019, pp. 469–484.

[67] V. Agrawal, A. Dabral, T. Palit, Y. Shen, and M. Ferdman, "Architectural support for dynamic linking," in *ACM SIGARCH Computer Architecture News*, vol. 43, no. 1. ACM, 2015, pp. 691–702.

[68] T. Gysi, T. Grosser, L. Brandner, and T. Hoefler, "A fast analytical model of fully associative caches," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 816–829. [Online]. Available: https://doi.org/10.1145/3314221.3314606

[69] W. Kelly and W. Pugh, "A framework for unifying reordering transformations," University of Maryland, Tech. Rep., 1992.

[70] S. Girbal, N. Vasilache, C. Bastoul, A. Cohen, D. Parello, M. Sigler, and O. Temam, "Semi-automatic composition of loop transformations for deep parallelism and memory hierarchies," *Int. J. Parallel Program.*, vol. 34, no. 3, pp. 261–317, Jun. 2006. [Online]. Available: http://dx.doi.org/10.1007/s10766-006-0012-3

[71] T. Yuki, G. Gupta, D. Kim, T. Pathan, and S. Rajopadhye, "Alphaz: A system for design space exploration in the polyhedral model," in *International Workshop on Languages and Compilers for Parallel Computing*. Springer, 2012, pp. 17–31.

[72] Q. Yi, "Poet: A scripting language for applying parameterized source-to-source program transformations," *Softw. Pract. Exper.*, vol. 42, no. 6, pp. 675–706, Jun. 2012. [Online]. Available: http://dx.doi.org/10.1002/spe.1089

[73] S. Donadio, J. Brodman, T. Roeder, K. Yotov, D. Barthou, A. Cohen, M. J. Garzarán, D. Padua, and K. Pingali, "A language for the compact representation of multiple program versions," in *Languages and Compilers for Parallel Computing*, E. Ayguadé, G. Baumgartner, J. Ramanujam, and P. Sadayappan, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 136–151.

[74] C. Chen, J. Chame, and M. Hall, "Chill: A framework for composing high-level loop transformations," USC Computer Science, Tech. Rep., June 2008.

[75] G. Rudy, "Cuda-chill: A programming language interface for gpgpu optimizations and code generation," Ph.D. dissertation, School of Computing, University of Utah, 2010.

[76] M. Khan, P. Basu, G. Rudy, M. Hall, C. Chen, and J. Chame, "A script-based autotuning compiler system to generate high-performance cuda code," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 31:1–31:25, Jan. 2013. [Online]. Available: http://doi.acm.org/10.1145/2400682.2400690

[77] T. S. F. X. Teixeira, C. Ancourt, D. Padua, and W. Gropp, "Locus: A system and a language for program optimization," in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019. Piscataway, NJ, USA: IEEE Press, 2019, pp. 217–228. [Online]. Available: http://dl.acm.org/citation.cfm?id=3314872.3314898

[78] M. Kruse and H. Finkel, "A proposal for loop-transformation pragmas," in *Evolving OpenMP for Evolving Architectures*, B. R. de Supinski, P. Valero-Lara, X. Martorell, S. Mateo Bellido, and J. Labarta, Eds. Cham: Springer International Publishing, 2018, pp. 37–52.

[79] R. Metzger and Z. Wen, *Automatic algorithm recognition and replacement: a new approach to program optimization*. MIT Press, 2000.

[80] C. W. Kessler, "Pattern-driven automatic parallelization," *Sci. Program.*, vol. 5, no. 3, pp. 251–274, Aug. 1996. [Online]. Available: http://dx.doi.org/10.1155/1996/406379

[81] C. W. Kessler and C. H. Smith, "The sparamat approach to automatic comprehension of sparse matrix computations," in *Proceedings Seventh International Workshop on Program Comprehension*, May 1999, pp. 200–207.

[82] B. Di Martino and G. Iannello, "Pap recognizer: a tool for automatic recognition of parallelizable patterns," in *WPC '96. 4th Workshop on Program Comprehension*, March 1996, pp. 164–174.

[83] W. Kozaczynski, J. Ning, and A. Engberts, "Program concept recognition and transformation," *IEEE Transactions on Software Engineering*, vol. 18, no. 12, pp. 1065–1075, Dec 1992.

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

[85] B. Pottenger and R. Eigenmann, "Idiom recognition in the polaris parallelizing compiler," in *Proceedings of the 9th International Conference on Supercomputing*, ser. ICS '95. New York, NY, USA: ACM, 1995, pp. 444–448. [Online]. Available: http://doi.acm.org/10.1145/224538.224655

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

[87] A. Shafiee Sarvestani, E. Hansson, and C. Kessler, "Extensible recognition of algorithmic patterns in dsp programs for automatic parallelization,"

International Journal of Parallel Programming, vol. 41, no. 6, pp. 806–824, Dec 2013. [Online]. Available: https://doi.org/10.1007/s10766-012-0229-2

[88] M. Bravenboer, K. T. Kalleberg, R. Vermaas, and E. Visser, "Stratego/xt 0.17. a language and toolset for program transformation," *Science of Computer Programming*, vol. 72, no. 1, pp. 52 – 70, 2008, special Issue on Second issue of experimental software and toolkits (EST). [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167642308000452

[89] T. Grosser, J. Ramanujam, L.-N. Pouchet, P. Sadayappan, and S. Pop, "Optimistic delinearization of parametrically sized arrays," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: ACM, 2015, pp. 351–360. [Online]. Available: http://doi.acm.org/10.1145/2751205.2751248

[90] M. Arenaz, J. Touriño, and R. Doallo, "Xark: An extensible framework for automatic recognition of computational kernels," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 6, p. 32, 2008.

[91] T. Suganuma, H. Komatsu, and T. Nakatani, "Detection and global optimization of reduction operations for distributed parallel machines," in *Proceedings of the 10th International Conference on Supercomputing*, ser. ICS '96. New York, NY, USA: ACM, 1996, pp. 18–25. [Online]. Available: http://doi.acm.org/10.1145/237578.237581

[92] M. P. Gerlek, E. Stoltz, and M. Wolfe, "Beyond induction variables: Detecting and classifying sequences using a demand-driven ssa form," *ACM Trans. Program. Lang. Syst.*, vol. 17, no. 1, pp. 85–122, Jan. 1995. [Online]. Available: http://doi.acm.org/10.1145/200994.201003

[93] S. S. Pinter and R. Y. Pinter, "Program optimization and parallelization using idioms," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 3, pp. 305–327, May 1994. [Online]. Available: http://doi.acm.org/10.1145/177492.177494

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

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

[96] R. Gareev, T. Grosser, and M. Kruse, "High-performance generalized tensor operations: A compiler-oriented approach," *ACM Trans. Archit. Code Optim.*, vol. 15, no. 3, Sep. 2018. [Online]. Available: https://doi.org/10.1145/3235029

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

[98] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Orti, "Analytical modeling is enough for high-performance blis," *ACM Trans. Math. Softw.*, vol. 43, no. 2, Aug. 2016. [Online]. Available: https://doi.org/10.1145/2925987

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

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

[101] T. Grosser, J. Ramanujam, L.-N. Pouchet, P. Sadayappan, and S. Pop, "Optimistic delinearization of parametrically sized arrays," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 351–360. [Online]. Available: https://doi.org/10.1145/2751205.2751248

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

[103] P. Springer and P. Bientinesi, "Design of a high-performance gemm-like tensor–tensor multiplication," *ACM Trans. Math. Softw.*, vol. 44, no. 3, Jan. 2018. [Online]. Available: https://doi.org/10.1145/3157733

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

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

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

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

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

[109] P. Ginsbach, T. Remmelg, M. Steuwer, B. Bodin, C. Dubach, and M. F. P. O'Boyle, "Automatic matching of legacy code to heterogeneous apis: An idiomatic approach," *SIGPLAN Not.*, vol. 53, no. 2, p. 139–153, Mar. 2018. [Online]. Available: https://doi.org/10.1145/3296957.3173182

[110] P. Ginsbach, B. Collie, and M. F. P. O'Boyle, "Automatically harnessing sparse acceleration," in *Proceedings of the 29th International Conference on Compiler Construction*, ser. CC 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 179–190. [Online]. Available: https://doi.org/10.1145/3377555.3377893

[111] M. Arenaz, J. Touriño, and R. Doallo, "XARK: An Extensible Framework for Automatic Recognition of Computational Kernels," *ACM Trans. Program. Lang. Syst.*, vol. 30, no. 6, Oct. 2008. [Online]. Available: https://doi.org/10.1145/1391956.1391959

[112] L. Chelini, O. Zinenko, T. Grosser, and H. Corporaal, "Declarative loop tactics for domain-specific optimization," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, Dec. 2019. [Online]. Available: https://doi.org/10.1145/3372266

[113] M. Felleisen, R. B. Findler, M. Flatt, S. Krishnamurthi, E. Barzilay, J. McCarthy, and S. Tobin-Hochstadt, "A programmable programming

language," *Commun. ACM*, vol. 61, no. 3, p. 62–71, Feb. 2018. [Online]. Available: https://doi.org/10.1145/3127323

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

[115] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen, "Languages as libraries," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 132–141. [Online]. Available: https://doi.org/10.1145/1993498.1993514

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

[117] K. Asanovic, R. Bodik, J. Demmel, T. Keaveny, K. Keutzer, J. Kubiatowicz, N. Morgan, D. Patterson, K. Sen, J. Wawrzynek, D. Wessel, and K. Yelick, "A view of the parallel computing landscape," *Commun. ACM*, vol. 52, no. 10, p. 56–67, Oct. 2009. [Online]. Available: https://doi.org/10.1145/1562764.1562783

[118] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 12–25. [Online]. Available: https://doi.org/10.1145/1993498.1993501

[119] M. I. Cole, *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[120] C. Nugteren and H. Corporaal, "Bones: An automatic skeleton-based c-to-cuda compiler for gpus," *ACM Trans. Archit. Code Optim.*, vol. 11, no. 4, Dec. 2014. [Online]. Available: https://doi.org/10.1145/2665079

[121] J. Enmyren and C. W. Kessler, "Skepu: A multi-backend skeleton programming library for multi-gpu systems," in *Proceedings of the Fourth International Workshop on High-Level Parallel Programming and Applications*, ser. HLPP '10. New York, NY, USA: Association for Computing Machinery, 2010, p. 5–14. [Online]. Available: https://doi.org/10.1145/1863482.1863487

[122] S. Raoux, F. Xiong, M. Wuttig, and E. Pop, "Phase change materials and phase change memory," *MRS bulletin*, vol. 39, no. 8, pp. 703–710, 2014.

[123] A. Sebastian, M. Le Gallo, R. Khaddam-Aljameh, and E. Eleftheriou, "Memory devices and applications for in-memory computing," *Nature Nanotechnology*, pp. 1–16, 2020.

[124] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. P. Strachan, M. Hu, R. S. Williams, and V. Srikumar, "Isaac: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 14–26.

[125] M. Le Gallo, A. Sebastian, G. Cherubini, H. Giefers, and E. Eleftheriou, "Compressed sensing with approximate message passing using in-memory computing," *IEEE Transactions on Electron Devices*, vol. 65, no. 10, pp. 4304–4312, Oct 2018.

[126] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. Devito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, "The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated gpu kernels, automatically," *ACM Trans. Archit. Code Optim.*, vol. 16, no. 4, pp. 38:1–38:26, Oct. 2019. [Online]. Available: http://doi.acm.org/10.1145/3355606

[127] P. Springer and P. Bientinesi, "Design of a high-performance gemm-like tensor–tensor multiplication," *ACM Transactions on Mathematical Software (TOMS)*, vol. 44, no. 3, pp. 1–29, 2018.

[128] A. Vasudevan, A. Anderson, and D. Gregg, "Parallel multi channel convolution using general matrix multiplication," in *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2017, pp. 19–24.

[129] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cuDNN: Efficient primitives for deep learning," *arXiv preprint arXiv:1410.0759*, 2014.

[130] A. Ankit, I. E. Hajj, S. R. Chalamalasetti, G. Ndu, M. Foltin, R. S. Williams, P. Faraboschi, W.-m. W. Hwu, J. P. Strachan, K. Roy, and D. S. Milojicic, "Puma: A programmable ultra-efficient memristor-based accelerator for machine learning inference," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: ACM, 2019, pp. 715–731. [Online]. Available: http://doi.acm.org/10.1145/3297858.3304049

[131] A. v. d. Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *arXiv preprint arXiv:1609.03499*, 2016.

[132] M. Zhang, S. Rajbhandari, W. Wang, and Y. He, "Deepcpu: Serving rnn-based deep learning models 10x faster," 07 2018.

[133] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: An integrated power, area, and timing modeling framework for multicore and manycore architectures," in *MICRO 42: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, 2009, pp. 469–480.

[134] M. K. Qureshi, J. Karidis, M. Franceschini, V. Srinivasan, L. Lastras, and B. Abali, "Enhancing lifetime and security of pcm-based main memory with start-gap wear leveling," in *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 14–23.

[135] G. W. Burr, M. J. Breitwisch, M. Franceschini, D. Garetto, K. Gopalakrishnan, B. Jackson, B. Kurdi, C. Lam, L. A. Lastras, A. Padilla *et al.*, "Phase change memory technology," *Journal of Vacuum Science & Technology B,*

Nanotechnology and Microelectronics: Materials, Processing, Measurement, and Phenomena, vol. 28, no. 2, pp. 223–262, 2010.

[136] K. Vadivel, L. Chelini, A. BanaGozar, G. Singh, S. Corda, R. Jordans, and H. Corporaal, "Tdo-cim: transparent detection and offloading for computation in-memory," in *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2020, pp. 1602–1605.

[137] A. Drebes, L. Chelini, O. Zinenko, A. Cohen, H. Corporaal, T. Grosser, K. Vadivel, and N. Vasilache, "TC-CIM: Empowering Tensor Comprehensions for Computing-In-Memory," IMPACT 2020 - 10th International Workshop on Polyhedral Compilation Techniques, Jan. 2020. [Online]. Available: https://hal.inria.fr/hal-02441163

[138] D. Fujiki, S. Mahlke, and R. Das, "In-memory data parallel processor," in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '18. New York, NY, USA: ACM, 2018, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/3173162.3173171

[139] J. Ambrosi, A. Ankit, R. Antunes, S. R. Chalamalasetti, S. Chatterjee, I. E. Hajj, G. Fachini, P. Faraboschi, M. Foltin, S. Huang, W. Hwu, G. Knuppe, S. V. Lakshminarasimha, D. Milojicic, M. Parthasarathy, F. Ribeiro, L. Rosa, K. Roy, P. Silveira, and J. P. Strachan, "Hardware-software co-design for an analog-digital accelerator for machine learning," in *2018 IEEE International Conference on Rebooting Computing (ICRC)*, Nov 2018, pp. 1–13.

[140] ONN, "Open Neural Network Exchange Home Page," `https://onnx.ai/` (Nov. 2019).

[141] P. Chi, S. Li, C. Xu, T. Zhang, J. Zhao, Y. Liu, Y. Wang, and Y. Xie, "Prime: A novel processing-in-memory architecture for neural network computation in reram-based main memory," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 27–39.

[142] L. Song, X. Qian, H. Li, and Y. Chen, "Pipelayer: A pipelined reram-based accelerator for deep learning," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Feb 2017, pp. 541–552.

[143] M. N. Bojnordi and E. Ipek, "Memristive boltzmann machine: A hardware accelerator for combinatorial optimization and deep learning," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, March 2016, pp. 1–13.

[144] S. Li, C. Xu, Q. Zou, J. Zhao, Y. Lu, and Y. Xie, "Pinatubo: A processing-in-memory architecture for bulk bitwise operations in emerging non-volatile memories," in *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, June 2016, pp. 1–6.

[145] K. Hsieh, E. Ebrahim, G. Kim, N. Chatterjee, M. O'Connor, N. Vijaykumar, O. Mutlu, and S. W. Keckler, "Transparent offloading and mapping (tom): Enabling programmer-transparent near-data processing in gpu systems," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, June 2016, pp. 204–216.

[146] A. Pattnaik, X. Tang, A. Jog, O. Kayiran, A. K. Mishra, M. T. Kandemir, O. Mutlu, and C. R. Das, "Scheduling techniques for gpu architectures with

processing-in-memory capabilities," in *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*, Sept 2016, pp. 31–44.

[147] R. Hadidi, L. Nai, H. Kim, and H. Kim, "Cairo: A compiler-assisted technique for enabling instruction-level offloading of processing-in-memory," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 4, pp. 48:1–48:25, Dec. 2017. [Online]. Available: http://doi.acm.org/10.1145/3155287

[148] R. Nair, S. F. Antao, C. Bertolli, P. Bose, J. R. Brunheroto, T. Chen, C. . Cher, C. H. A. Costa, J. Doi, C. Evangelinos, B. M. Fleischer, T. W. Fox, D. S. Gallo, L. Grinberg, J. A. Gunnels, A. C. Jacob, P. Jacob, H. M. Jacobson, T. Karkhanis, C. Kim, J. H. Moreno, J. K. O'Brien, M. Ohmacht, Y. Park, D. A. Prener, B. S. Rosenburg, K. D. Ryu, O. Sallenave, M. J. Serrano, P. D. M. Siegl, K. Sugavanam, and Z. Sura, "Active memory cube: A processing-in-memory architecture for exascale systems," *IBM Journal of Research and Development*, vol. 59, no. 2/3, pp. 17:1–17:14, March 2015.

[149] M. Amini, B. Creusillet, S. Even, R. Keryell, O. Goubier, S. Guelton, J. O. McMahon, F.-X. Pasquier, G. Péan, and P. Villalon, "Par4all: From convex array regions to heterogeneous computing," in *IMPACT: Second Int. Workshop on Polyhedral Compilation Techniques HiPEAC 2012*.

[150] M. M. Baskaran, J. Ramanujam, and P. Sadayappan, "Automatic c-to-cuda code generation for affine programs," in *Compiler Construction*. Springer, 2010.

[151] A. Leung, N. Vasilache, B. Meister, M. Baskaran, D. Wohlford, C. Bastoul, and R. Lethin, "A mapping path for multi-gpgpu accelerated computers from a portable high level programming abstraction," in *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, 2010, pp. 51–61. [Online]. Available: http://doi.acm.org/10.1145/1735688.1735698

[152] S. Baghdadi, A. Größlinger, and A. Cohen, "Putting Automatic Polyhedral Compilation for GPGPU to Work," in *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC'10)*, Vienna, Austria, Jul. 2010. [Online]. Available: https://hal.inria.fr/inria-00551517

[153] G. Rudy, M. Khan, M. Hall, C. Chen, and J. Chame, "A programming language interface to describe transformations and code generation," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, K. Cooper, J. Mellor-Crummey, and V. Sarkar, Eds. Springer Berlin Heidelberg, 2011, vol. 6548, pp. 136–150. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19595-2_10

[154] S. Verdoolaege, J. C. Juega, A. Cohen, J. I. Gómez, C. Tenllado, and F. Catthoor, "Polyhedral parallel code generation for CUDA," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 54:1–54:23, Jan. 2013.

[155] S. Verdoolaege, "PENCIL support in pet and PPCG," INRIA Paris-Rocquencourt, Technical Report RT-0457, Mar. 2015. [Online]. Available: https://hal.inria.fr/hal-01133962

[156] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. Van Haastregt,

A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, "PENCIL: a platform-neutral compute intermediate language for accelerator programming," in *International Conference on Parallel Architectures and Compilation Techniques*, San Francisco, US, 2015.

[157] T. Grosser, A. Groesslinger, and C. Lengauer, "Polly – performing polyhedral optimizations on a low-level intermediate representation," *Parallel Processing Letters*, vol. 22, no. 04, p. 1250010, 2012.

[158] D. Mikushin, N. Likhogrud, Z. E. Zhang, and C. Bergstrom, "Kernelgen – the design and implementation of a next generation compiler platform for accelerating numerical models on GPUs," in *Parallel & Distributed Processing Symposium Workshops (IPDPSW)*, 2014.

[159] M. Damschen, H. Riebler, G. Vaz, and C. Plessl, "Transparent offloading of computational hotspots from binary code to Xeon Phi," in *Proc. of the 2015 Design, Automation & Test in Europe Conf. & Exh*, ser. DATE '15. EDA Consortium, 2015, pp. 1078–1083. [Online]. Available: http://dl.acm.org/citation.cfm?id=2757012.2757063

[160] C. Margiolas, "A heterogeneous execution engine for llvm," *LLVM Developers Meeting*, 2015.

[161] C. Reddy, M. Kruse, and A. Cohen, "Reduction drawing: Language constructs and polyhedral compilation for reductions on gpu," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 87–97. [Online]. Available: https://doi.org/10.1145/2967938.2967950

[162] J. Doerfert, K. Streit, S. Hack, and Z. Benaissa, "Polly's polyhedral scheduling in the presence of reductions," *CoRR*, vol. abs/1505.07716, 2015. [Online]. Available: http://arxiv.org/abs/1505.07716

[163] P. Jouvelot and B. Dehbonei, "A unified semantic approach for the vectorization and parallelization of generalized reductions," in *Proceedings of the 3rd International Conference on Supercomputing*, ser. ICS '89. New York, NY, USA: Association for Computing Machinery, 1989, p. 186–194. [Online]. Available: https://doi.org/10.1145/318789.318810

[164] P. Feautrier and C. Lengauer, "Polyhedron Model," in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Springer, 2011, pp. 1581–1592.

[165] E. Schweitz, "An mlir dialect for high-level optimization of Fortran," in *2019 LLVM Developers Meeting*, 2019.

[166] E. Mutlu, R. Tian, B. Ren, S. Krishnamoorthy, R. Gioiosa, J. Pienaar, and G. Kestor, "Comet: A domain-specific compilation of high-performance computational chemistry," in *The 33rd Workshop on Languages and Compilers for Parallel Computing*, 2020.

[167] npcomp developers. (2020) Mlir npcomp. [Online]. Available: https://github.com/llvm/mlir-npcomp

[168] 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)*, 2020, pp. 551–556.

[169] C. Developers. (2020) Circt charter. [Online]. Available: https://github.com/llvm/circt/blob/master/docs/Charter.md

A

Acknowledgements

Here I am writing this last part of my thesis. It has been a long hike, but I enjoyed every moment of it. So many colleagues, friends, and family members walked with me during this four-year-long journey and made it a wonderful experience that formed me personally and professionally. First, I would like to thank you, Henk and Tobias, who guided me throughout these years of studies, leaving me the freedom to choose the direction I want to follow but still encouraging me when I needed guidance. Without you, I would not be here writing this very end of my thesis.
Tobias, you had a tremendous impact on who I am today. As a fresh graduate student with close to zero knowledge of doing research, you taught me everything from thinking critically to creating visual PowerPoint slides. Thank you, Tobias.
Oleksandr "Alex" Zinenko, working with you has been (and still is) fantastic. I still remember our brainstorming meetings and coding sessions when you visited ETH Zurich and our adventure in Bologna to find fresh homemade pasta. If Tobias taught me how to be a better researcher, you made me a better compiler developer. Today, I am enjoying our discussion and work together.
A big thank you to all the fantastic people I met who helped me shape part of this thesis: Albert Cohen, Nicolas Vasilache, Tobias Gysi, Andi Drebes, Asif Ali Khan, William Moses, Ruizhe Zhao, Martin Kong, Roel Jordans, and the other two members of the NeMeCo team: Stefano Corda and Gagandeep Singh. Gagan, we had many adventures, from Tiger-Tiger in Manchester to a great trip in Croatia. I could not have asked for a better colleague, flatmate, and a special friend.
Finally, thanks to my parents Cristina and Michele and my brother Edoardo, who supported me during this journey and made all of this possible, and the woman on my side, Maria, who I met during my studies and who is having a lot of patience while I am writing this paragraph on our vacation in Italy.

B

List of Publications

The following is a list of publications I wrote or contributed to while working towards my doctoral degree. Ideas, figures, listings, and text of some of these works have been incorporated into this thesis.

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

- William S. Moses, Lorenzo Chelini, Ruizhe Zhao, Oleksandr Zinenko. **Polygeist: Raising C to Polyhedral MLIR.** PACT 2021
  coming soon
- Adam Siemieniuk, Lorenzo Chelini, Asif Ali Khan, Jeronimo Castrillon, Andi Drebes, Henk Corporaal, Tobias Grosser, Martin Kong. **OCC: An Automated End-to-End Machine Learning Optimizing Compiler for Computing-In-Memory** TCAD 2021
  publisher paper
- William S. Moses, Lorenzo Chelini, Ruizhe Zhao, Oleksandr Zinenko. **Polygeist: Affine C in MLIR.** IMPACT 2021
  publisher paper
- Lorenzo Chelini, Andi Debres, Oleksandr Zinenko, Albert Cohen, Henk Corporaal, et al. **Progressive Raising in Multi-level IR.** CGO 2021
  publisher paper
- Lorenzo Chelini, Martin Kong, Tobias Grosser, Henk Corporaal. **LoopOpt: Declarative Transformations Made Easy.** SCOPES 2021
  coming soon
- Lorenzo Chelini, Tobias Gysi, Tobias Grosser, Martin Kong, and Henk Corporaal. **Automatic Generation of Multi-Objective Polyhedral Compiler Transformations.** PACT 2020
  publisher paper slides
- Lorenzo Chelini, Andi Debres, Oleksandr Zinenko, Albert Cohen, Henk Corporaal, et al. **Multi-Level Tactics: Lifting Loops in MLIR.** EuroLLVM 2020
  publisher poster paper
- Andi Drebes, Lorenzo Chelini, Oleksandr Zinenko, Albert Cohen, Henk Corporaal, et al. **TC-CIM: Empowering Tensor Comprehensions for Computing-In-Memory.** IMPACT 2020
  publisher paper slides

- Konrad Komisarczyk, Lorenzo Chelini, Roel Jordans, Henk Corporaal. **PET-to-MLIR: A polyhedral front-end for MLIR.** DSD 2020
  publisher paper
- Kanishkan Vadivel, Lorenzo Chelini, Ali BanaGozar, Gagandeep Singh, Stefano Corda, et al. **TDO-CIM: Transparent Detection and Offloading for Computation In-memory.** DATE 2020
  publisher paper
- Lorenzo Chelini, Oleksandr Zinenko, Tobias Grosser, and Henk Corporaal. **Declarative Loop Tactics for Domain-specific Optimization.** ACM TACO 2020
  publisher paper slides
- Oleksandr Zinenko, Lorenzo Chelini, Tobias Grosser. **Declarative Transformations in the Polyhedral Model** INRIA Tech Report 2019
  publisher
- Jan van Lunteren, Ronald Luijten, Dionysios Diamantopoulos, Florian Auernhammer, Christoph Hagleitner, Lorenzo Chelini, et al. **Coherently Attached Programmable Near-Memory Acceleration Platform and its application to Stencil Processing**
  publisher
- Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, et al. **Near-memory computing: Past, present, and future.** MICPRO 2019
  publisher paper
- Gagandeep Singh, Lorenzo Chelini, Stefano Corda, Ahsan Javed Awan, Sander Stuijk, et al. **A Review of Near Memory Computing Architectures Opportunities and Challenges.** DSD 2019
  publisher paper

C

CV

Lorenzo Chelini was born on 23-12-1993 in Lucca, Italy. After finishing Computer Engineering in 2015 at the University of Pisa, Italy, he studied Embedded System Design at Politecnico di Torino (Torino, Italy) and Chalmers University of Technology (Göteborg, Sweden). In 2017 he graduated in the Electronic Design Automation Group with the thesis "Power Estimation for DSP Components for Fiber-Optic Communication Systems". He joined the Electronic Systems group of the Eindhoven University of Technology in September 2017 as a Ph.D. candidate. During his Ph.D., Lorenzo Chelini visited IBM Research Lab in Zurich, ETHZ under the supervision of Dr. Tobias Grosser and The University of Edinburgh, again under the supervision of Dr. Tobias Grosser. The results of this Ph.D. project are presented in this thesis.