Idiomatic Code Acceleration for Heterogeneous Systems

Philip Ginsbach

Doctor of Philosophy
Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh
2019

Abstract

This doctoral thesis will present the results of my work into the reanimation of lifeless human tissues.

Acknowledgements

First and foremost I want to thank Mike for his ...

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers:

- Philip Ginsbach and Michael F. P. O'Boyle.
  Discovery and Exploitation of General Reductions: A Constraint Based Approach.
  Proceedings of the 15th Annual International Symposium on Code Generation and Optimization (CGO), 2017

- Philip Ginsbach, Lewis Crawford and Michael F. P. O'Boyle.
  CAnDL: A Domain Specific Language for Compiler Analysis.
  Proceedings of the 27th International Conference on Compiler Construction (CC), 2018

- Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach and Michael F. P. O'Boyle.
  Automatic Matching of Legacy Code to Heterogeneous APIs: An Idiomatic Approach.
  Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2018

- Philip Ginsbach, Bruce Collie and Michael F. P. O'Boyle.
  Automatically Harnessing Sparse Acceleration Libraries.
  In: *???, 2019*

(*Philip Ginsbach*)

Table of Contents

Chapter 1

Introduction

The end of Moores Law and the end of Dennard Scaling require new approaches in hardware.

Heterogeneous computing platforms are the natural reaction to this.

However, with heterogeneous hardware becoming an essential component of computing capabilities, thinking of it as library accelerators becomes wrong.

We need a new hardware software contract instead and heterogeneous hardware should become a responsibility of the compiler.

While compilers have lagged behind the developments in the hardware domain, we can profit from experience with multi-core processors.

Auto-parallelizing compilers have failed to solve the problems and have only had major success for specific kernels and using auto-tuning.

Heterogeneous computing is a superset of parallel computing and so an approach to fully automatically optimize code is unlikely.

At the same time, a library and DSL based approach has been successful, however fails to become mainstream due to adoption cost.

What is promising therefore is a combination of hand-optimized and -parallelized libraries together with compiler automatisms.

This is what we consider an idiomatic approach.

1.1 Structure of the Thesis

The thesis will be structures as follows. Firstly, there will be a literature survey investigating related work.

We will then establish a theory for constraint programming on static single assignment compiler intermediate representation code. Using this theory, a constraint programming language will be presented and an implementation provided: The Compiler Analysis Description Language (CAnDL).

Using this language, the thesis will derive computational idioms and show how they can be used to get heterogeneous performance. Lastly, we build a fully integrated system.

Chapter 2

Literature Survey

2.1 Computational Idioms

The concept of computational idioms has been observed in different contexts and remains a rather vague concept. While terms such as `reduction`, `stencil` and `linear algebra` are commonly used, the concrete concepts can be surprisingly vague, although previous work has established several formal approaches. We will not try to create our own formal definitions in this chapter, but instead want to give an overview of the literature that sheds different perspectives on this topic.

The basic observation is that software programs don't cover the space of possible programs evenly, instead, they tend to be structured among certain design principles. The same is true algorithmically and particularly for performance intensive applications.

2.2 Literature Survey

2.2.1 Algorithmic Skeletons

2.2.2 Berkeley Parallel Dwarves

2.2.3 Computational Patterns

2.3 Idiom Specific Optimizations

2.3.1 Important Approaches

2.3.2 Ways of Encapsulating Expertise

2.3.3 Constraint Programming for Program Analysis

There is a large body of work that uses constraints for program analysis. Constraint systems have long been used in program analysis for classical purposes such as dataflow analysis and

type inference [1]. Gulwani et al. [36] showed how constraints can be used to for some existing analysis problems. It can also be used to assist generation of programs from specifications [87].

There is considerable work on formally verifying existing compiler transformations using SMT solvers and theorem provers that operate on IR code, among them [107]. Recent domain specific language for formally verifying compiler optimizations, such as Alive [57] operate on single static assignment compiler IR as well. However, it has no support for control flow and is limited to simple peephole optimizations.

Further approaches to generating compiler optimizations that focus on formal verification instead of programmer productivity include Rhodium [55], PEC [51] and Gospel [100]. The CompCert tool by Mullen et al. [67] verifies peephole optimizations directly on x86 assembly code and LifeJacket [69] proves the correctness of floating point optimizations in LLVM, as does Alive-FP [62], which also generates C++ code. The authors of [96] investigate the automatic generation of optimization transformations from examples. None of the approaches however, tackle the issue of efficiently writing arbitrary compiler analysis passes.

Martin [58] present a specification language for program analysis functionality called PAG that is based on abstract interpretation. Domain specific languages for the conception of optimization passes have also been studied before using tree rewrites, among them Olmos and Visser [74]. Lipps et al. [56] propose the domain specific language OPTRAN for matching patterns in attributed abstract syntax trees of Pascal programs. These patterns can then be automatically replaced by semantically equivalent, more efficient implementations. Generic programming concepts can also be used to generate optimization passes, as demonstrated by [101]. These schemes however are not able to work at the IR level essential for compiler implementation and do not scale beyond simple functions.

As opposed to code transformation techniques based on the LLVM ASTMatcher library and LibTooling [103], we work on compiler intermediate representation and are independent from the compiler frontend. This makes us integrate well with the existing optimization infrastructure, allows us to be language independent and makes our approach robust to shallow syntactic changes.

Another language for implementing optimizations that emphasizes developer productivity is OPTIMIX [7, 8], based on graph rewrite rules. OPTIMIX programs are compiled into C code that performs the specified transformation. A domain specific language for the generation of optimization transformations was also used in the CoSy compiler [3]. These are simple rewrite engines and have no knowledge of global program constraints.

Other work has investigated the use of constraint solvers for detecting structure in LLMV IR. Ginsbach and O'Boyle [32] use a solver to detect histograms and scalar reductions in order to automatically parallelize code. A different important approach to detecting structure in intermediate code is the polyhedral model. Compilers using this model, such as Grosser

et al. [35], Baskaran et al. [12] and Verdoolaege et al. [97], capture well behaved loops with affine array accesses and are able to perform advanced loop optimizations. Recent work has investigated performant reduction computations on GPUs with the polyhedral model [21]. However this approach is not easily extensible to other program structures and captures only a very specific class of well behaved programs.

In [106], the authors propose the use of the Web Ontology Language (WOL) for the description of software architectures. This representation enables automatic reasoning and analysis about the interoperability of software architectures.

Chapter 3

Contraint Programming on Static Single Assignment Code

The research contributions of this thesis are based on a novel constraint programming approach on compiler intermediate representation. This chapter introduces the theoretical underpinnings of this approach, developing a method for formally describing and recognising algorithmic structures in programs during compilation. Based on a mathematical model of static single assignment form, concepts that are typically applied to syntactic representations of programs are transferred onto more semtantically simple compiler internal representaitons.

Traditional pattern matching approaches here turn into more subtle constraint problems. This requires careful consideration to handle search space explosion but at the same time powers much more powerful detection capabilities based on higher level algorithmic structures that are impossible to express syntactically. Later chapters will evaluate these claims in details by building a complete constraint programming language on top of the theory derived in this chapter and by then using it to express real compiler analysis problems.

After defining a mathematical foundation of SSA form and giving basic definitions of constraint problems and solutions, several standard definitions from compiler theory are then reformulated in the the framework. This includes common graph properties and the domination concepts as well as control flow structues such as single entry single exit blocks. All of this lies the foundations for the succeeding chapter.

As a last part, a practical application of constraint programming to compiler intermediate representation requires efficient solver techniques and a section in the chapter will discuss strategies for limiting compile time explosions and explore analogies of the described model to standard Satisfiability Modulo Theory (SMT) problems. This will give further insights into performance improvements and put the work into a wider context.

3.1 Introduction

During the different compilation stages, modern optimizing compilers for proceedural languages such as C/C++, Fortran or JavaScript typically use a range of different representations of the user program. Static single assignment (SSA) form has emerged as a suitable representation in the mid end for applying complex optimizing transformations. It abstracts away the complexities of both the source language and the target architecture, enabling reliable analysis and platform independent reasoning.

Prominent examples of compilers that utilize static single assignement representations for the bulk of their optimisation passes are **clang/clang++** (LLVM IR), **gcc** (GIMPLE), **v8 Crankshaft** (Hydrogen) and **SpiderMonkey** (IonMonkey/MIR).

The precise instruction set, syntax and type systems of the different static single assignment form intermediate representations vary depending on the requirements of the source languages (static or dynamic) and the operating constraints (JIT or AOF). However, they share the same fundamental paradigmes and we can mostly abstract away the differences as implementation specific details in this chapter. Fundamentally, a program in single static assignment form is made up of functions that are represented as sequences of instructions that are grouped into basic blocks and that operate on virtual registers. These virtual registers can be assigned only once and the place of assignment can be statically determined. In order to support control flow divergence, SSA form utilized PHI nodes that enapsulate the non-SSA behaviour.

This chapter derives a methodology to recognise structures in SSA code via constraint programming. For this to work, a mathematical model of the textual representation is required.

3.1.1 Static Single Assignment Form

Static single assignemnt is a property that applies to compiler intermediate representations. Functions in SSA form are represented as sequences of instructions that operate on an abstract machine and that are grouped into basic blocks. The abstract machine provides an unlimited number of registers and a well defined instruction set. Each instruction has a finite amount of input arguments and an opcode that refers to a specific operation to be performed. instruction arguments can be registers or constants and instructions can write their result into a single output register. Control flow is expressed as branch instructions that can redirect control conditionally or unconditionally to the start of other basic blocks. Branch instructions signify the end of a basic block. Instructions and registers may be statically typed.

The static single assignemnt property stipulates that within a function, each register is only written at a single static location. This implies that the data dependencies between the instructions are explicit and registers can be identified directly with the instructions that write to them. The registers themselves can therefore be considered implicit, with only the data flow

between instructions required to recover them.

In the presence of dynamic control flow behaviour in the program, most simply in the case of a conditional branch, the static single assignment property can only be maintained using phi instructions. These are particular pseudo-instructions that encapsulate assignments that can only dynamically be determined.

3.2 Deriving a Mathematical Model

This section derives a mathematical model of programs in single static assignemnt form. Using this model as a sound basis and mathematical notation, common compiler analysis problems are then reformulated. Later chapters will use this to define computational idioms formally, and to implement automatic compiler tools. It is not the aim here to introduce a formal operational semantics, or more generally to derive a model for the execution of SSA programs. Instead, the section will investigate the static structure, focusing on clear notation of the commonalities of existing SSA intermediate representations.

The previous section showed how programs in SSA representations can be disected into several components, including control flow, data flow and instruction specifications. Figure 3.1 shows how this can be taken further in order to extract a more mathematical approach. At the top of the figure, different textual representations of a simple vector dot product are shown. The version of the top right is in an SSA intermediate representation: LLVM IR as generated by the clang compiler.

In the middle column of Figure 3.1, the information contained in the SSA representation is split into three components: Firstly, the we need a set of per-instruction properties, such as instruction opcodes, types and the values of constants that are used. Secondly, we capture the control flow graph. Thirdly, we capture the detailed data flow graph of the program. This data structure contains information about how the results of previous instructions are used as arguments to successive instructions. As we discussed in the previous section, the SSA property ensures that this is enough information to make the concrete register usage implicit.

We can see this Figure 3.1. The single static assignment property implies that it is statically knows at each point, which instruction produced the value in each register. Therefore, we can immediately identify registers with their producting instruction. Therefore we can entirely capture the interaction between instructions in two graphs: the control flow graph and the data flow graph. This is shown in in middle section of the figure. The data flow graph entirely replaces the concept of registers and instead models directly how the results from instructions are used as arguments in succeeding instructions. The control flow graph models the possible paths through the program.

Note that we entirely removed the concepts of basic blocks and registers here. However,

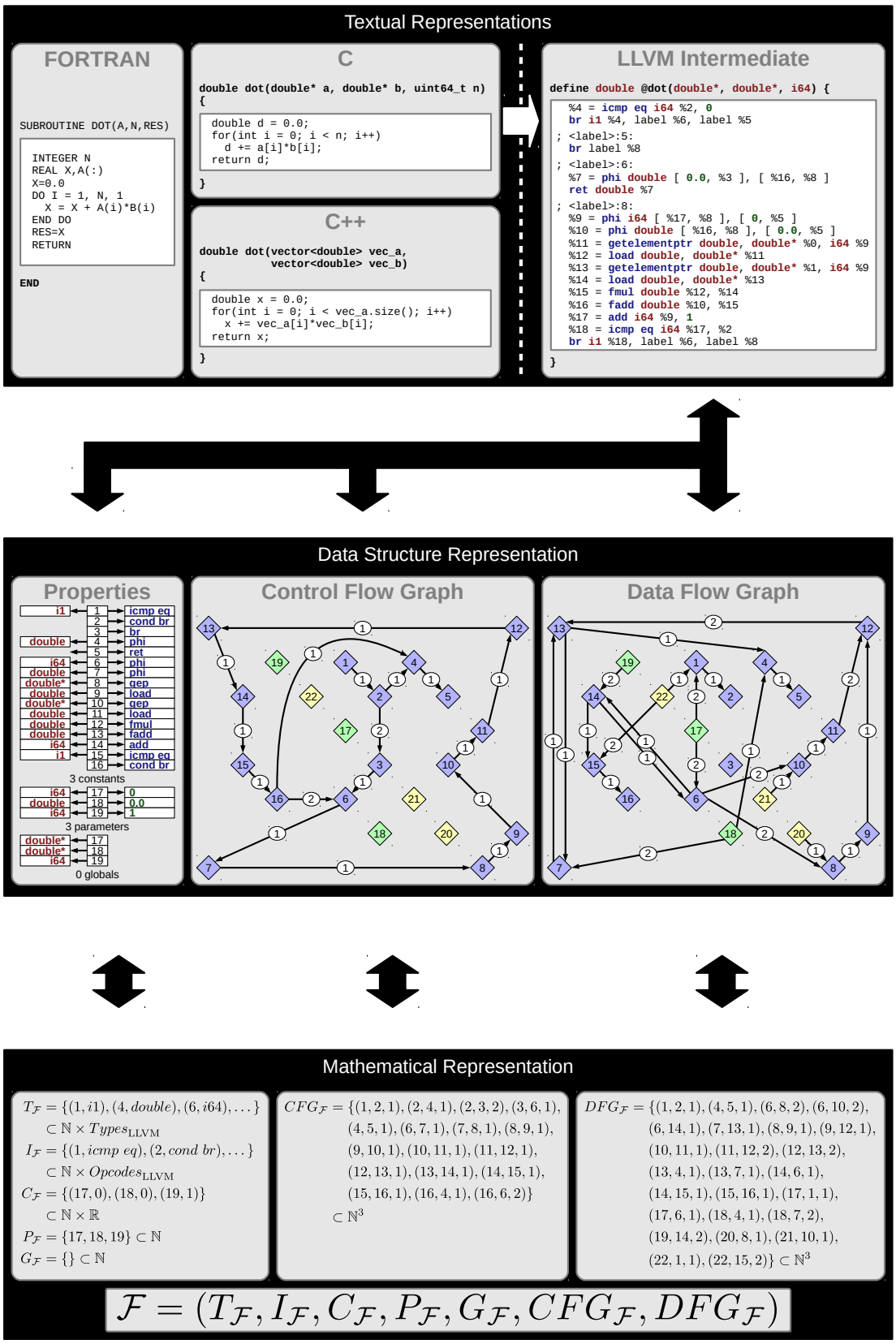


Figure 3.1: Compiler-generated SSA code is first decomposed into data flow, control flow and per-value attributes. Mathematical representations of the data are then introduced with notation.

both can be recovered easily from the graph representations and hence we lost no information going from the textual representation towards the graph representations.

Finally, we can model this entirely mathematical. We can see at the bottom of the figure, how we can model the entire function body as a tuple

$$\mathscr{F} = (T_{\mathscr{F}}, I_{\mathscr{F}}, C_{\mathscr{F}}, P_{\mathscr{F}}, G_{\mathscr{F}}, CDG_{\mathscr{F}}, DFG_{\mathscr{F}}).$$

Note that the same model can be used abstractly for other single static assignment forms, only the type and opcode information as encoded via $Types_{\text{LLVM}}$ and $Opcodes_{\text{LLVM}}$ are specific to LLVM and need to be replaced.

3.2.1 Important Graph Properties

With our established notation, we can now transfer standard compiler analysis problems into this more formal language. Most of these are based on graph theoretic considerations, so we will firstly need to recapitulate some graph theory basics. Firstly, there is the notion of *cuts* of graphs, that we will introduce here in a hybrid version of edge based and vertex based modelling.

Definition 3.1: Connections and Cuts

Consider an adjacency set $E \subset \mathbb{N} \times \mathbb{N}$ of a directed graph and let $a, b \in \mathbb{N}$.
A *connection* between a and b in E is a subset $A \subset \mathbb{N}$ such that a finite sequence $c_1, \ldots, c_n$ exists with

$$a = c_1 \qquad c_2, \ldots, c_{n-1} \in A \qquad b = c_n$$
$$(c_k, c_{k+1}) \in E \quad \text{for all} \quad k = 1, \ldots, n-1.$$

A *cut* between a and b in E is a subset $B \subset E$ such that no *connection* between a and b in $E \setminus B$ exists. We define the *set of cuts* between a and b in E as

$$\text{Cuts}_E(a,b) := \{B \subset E \mid B \text{ is } \textit{cut} \text{ between } a \text{ and } b \text{ in } E\}$$

These notions are quite intuitive, two vertices in a graph have a connection if one can reach the other via the available edges and by "cutting" these edges, they are no longer connected.

These definitions are very useful in order to identify crucial properties of data and control flow graphs. Most standard is the the definition of a dominator in the control flow graph: An instruction d is said to dominate another instruction n if every path from the entry node to n through the control flow graph must go through d. In our model this is of course equivalent to the following:

Definition 3.2: Dominator

Consider an instruction n in a function $\mathscr{F}$. A *dominator* of n in $\mathscr{F}$ is an instruction d such that $\{(d,m) \mid (d,m) \in CFG^*_{\mathscr{F}}\}$ is a *cut* between 1 and n in $CFG^*_{\mathscr{F}}$.

Another important definition is the concept of control dependence. Control dependence models the behaviour of conditional control flow. Instructions that are executed only in some control flow paths are control dependent on the conditional branches that preceed them.

UNDERFUL VBOX.

UNDERFUL VBOX.

UNDERFUL VBOX.

UNDERFUL VBOX.

Definition 3.3: Control Dependence

Consider instructions a, b. We say that an b is control dependent on a if a instructions c, c' exist such that $(a,c),(a,c') \in CFG^*_{\mathscr{F}}$ and

$$\{(a,c)\} \in \text{Cuts}_E(a,b)$$
$$\{(a,c')\} \notin \text{Cuts}_E(a,b).$$

We define the *control dependence graph* as follows

$$CDG_{\mathscr{F}} := \{(a,b) \in \mathbb{N}^2 \mid b \text{ control dependent on } a\}$$

3.2.2 Control Dependence Example

The control dependence graph is a function of the control flow graph, as is directly apparent from Definition 3.3. We can see how an example control dependence graph is computed in Figure 3.2, from the control flow graph of the `dot` function in Figure 3.1. From the definition it is immediately obvious that we need to only consider conditional branches as origins of control dependence.

We can consider the two conditional branches 2 and 16 independently. On the right, we consider only 2. We check the defining property: On the top of the figure, all the instructions that are not reachable from 2 without the edge $(2,4)$ are in grey. Below this, all instructions not reachable from 2 without $(2,3)$ are grey. We see that $4,5$ are always reachable and 1 is never reachable, these are therefore not control dependent on 2. All the other instructions are control dependent on 2.

Once we have computed this for all conditional branches, we take the union on graphs and get the complete control dependence graph of the function. Note what this graph represents: Once the loop in the function has been unrolled, it contains a conditional and a loop. Eveything within the body of the conditional is control dependent on 2. Everythig within the loop as well as everything afterwards is control dependent on 16.

UNDERFUL VBOX.

UNDERFUL VBOX.

UNDERFUL VBOX.

3.2.3 Phi Dependence Graph

Phi nodes are fundamental in single static assignment form and need special care. The value that a phi node takes depends on from where a phi node was reached. We need to encapsulate this in a graph.

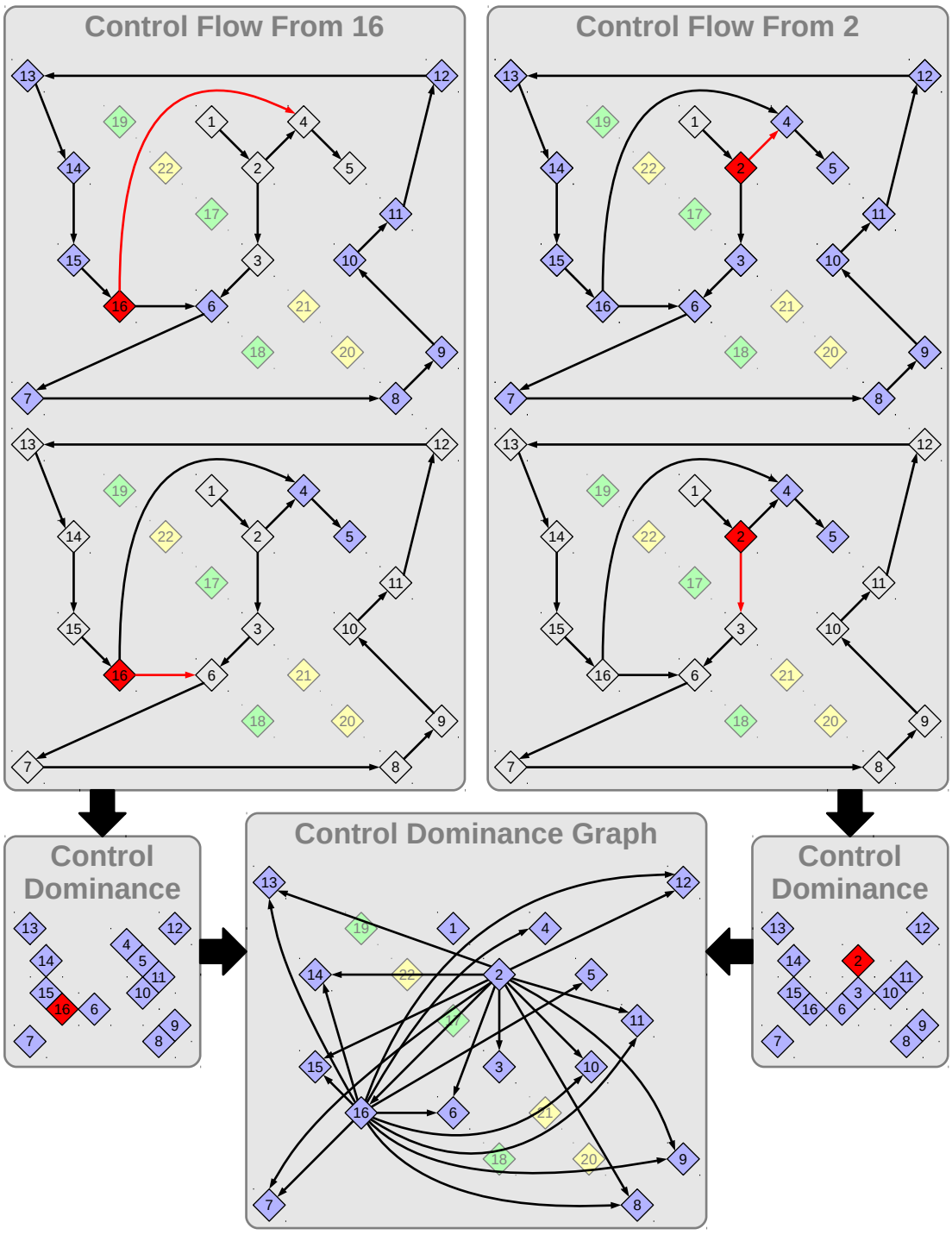


Figure 3.2: Computation of the control dependence graph.

Definition 3.4: Phi Dependence Graph

Let p a phi node and c a conditional branch instruction. We say that the outcome of p depends on c if there is a branch instruction b that reaches p such that b is control dependent on c.
This defines the *phi dependence graph* $\Phi DG_{\mathscr{F}}$.

3.2.4 Program Dependence Graph

After the control flow, data flow and control dependence graph, we lastly introduce the *program dependence graph*. It is the most exhaustive tool that we have to describe how values depend on each other.

Definition 3.5: Program Dependence Graph

The *program dependence graph* is defined as the union of data flow and control dependence graphs.

$$PDG_{\mathscr{F}} := DFG^*_{\mathscr{F}} \cup CDG^*_{\mathscr{F}} \cup \Phi DG_{\mathscr{F}}.$$

With the program dependence graph, we can now define subsections of the program that are self-contained and can be separated into their own function. This works even if they contain complicated control flow. Firstly, we need a definition of an interface.

Definition 3.6: Interface

Let $a \in CFG^*_{\mathscr{F}}$ and $b_1, \ldots, b_n \in DFG^*_{\mathscr{F}}$. Furthermore let $A \subset \mathbb{N}$ a set of instructions.
We say that $(b_1, \ldots, b_n)$ is an interface to A if it is a cut between o and A in $PDG_{\mathscr{F}}$ for any of the following o:

- o is a paramter
- o is impure

3.2.5 Interface Example

We will now consider a non-trivial example. Consider this snppet of C code, implementing a function that performs a simple square root approximation on each element in an array of double precision floating point values.

```
1  void map_sqrt(size_t length, double* array)
2  {
3      for(int i = 0; i < length; i++)
4      {
5          double root = 1.0;
6          for(int i = 0; i < 10; i++)
7              root = 0.5*(root+array[i]/root);
8
9          array[i] = root;
10     }
11 }
```

Figure 3.3: **map**∘**sqrt**: Apply an appriximate sqare root function to each element in a vector.

Coneptually, we should be able to disentangle the square root function from the control flow of the outer loop. This is possible with the preceeding definition of *interfaces*.

In single static assignment form, this code looks as follows:

```
define void @map_sqrt(i64, double*) {
  %3 = icmp eq i64 %0, 0
  br i1 %3, label %5, label %4

; <label>:4:
  br label %6

; <label>:5:
  ret void

; <label>:6:
  %7 = phi i64 [ %10, %8 ], [ 0, %4 ]
  br label %12

; <label>:8:
  %9 = getelementptr double, double* %1, i64 %7
  store double %19, double* %9
  %10 = add nuw i64 %7, 1
  %11 = icmp eq i64 %10, %0
  br i1 %11, label %5, label %6

; <label>:12:
  %13 = phi i64 [ 0, %6 ], [ %20, %12 ]
  %14 = phi double [ 1.0, %6 ], [ %19, %12 ]
  %15 = getelementptr inbounds double, double* %1, i64 %13
  %16 = load double, double* %15
  %17 = fdiv double %16, %14
  %18 = fadd double %14, %17
  %19 = fmul double %18, 5.0
  %20 = add nuw nsw i64 %13, 1
  %21 = icmp eq i64 %20, 10
  br i1 %21, label %8, label %12
}
```

In this example, the set $\{\%9\}$ is an interface to $\{(\%19, \%store)\}$.

3.3 Formulating Constraint Problems

We now use these mathematical background deliberations to derive constraint programming on top of LLVM code.

UNDERFUL VBOX.

UNDERFUL VBOX.

UNDERFUL VBOX.

Consider the following definitions of simple binary predicates:

$$is\_branch\_inst(\mathscr{F},n) := (n,\mathbf{br}) \in T_{\mathscr{F}}$$
$$is\_control\_edge(\mathscr{F},n,m) := (n,m) \in CFG^{*}_{\mathscr{F}}$$
$$is\_control\_dom(\mathscr{F},n,m) :=$$

We can then use these predicates to define more complex constructs, such as single entry, single exit (SESE) regions.

Definition 3.7

A single entry single exit region is a tuple $a,b,c,d \in \mathscr{N}$ such that the following properties hold:

$$is\_control\_edge(\mathscr{F},a,b)$$
$$is\_control\_edge(\mathscr{F},c,d)$$
$$is\_control\_dom(\mathscr{F},c,d)$$
$$is\_control\_postdom(\mathscr{F},d,c)$$

Chapter 4

Introducing the Compiler Analysis Description Language

In the preceding chapter, a mathematical theory for contraint programming on static single assignment compiler intermediate representation was derived. This chapter will now build on top of this theoretical framework, developing a domain specific contraint programming language and presenting an implementation of this language in the production quality LLVM compiler infrastructure.

The Compiler Analysis Description Language (CAnDL) makes it possible to automatically generate compiler analysis functions that would otherwise have to be implemented manually. Such a programming language fits an important niche in compiler development: Optimizing compilers utilise elaborate program transformations to exploit increasingly complex hardware, and implementing the appropriate analysis functionality for such optimizations to be safely applied is a time consuming and error prone activity. For example, tens of thousands of lines of code are required in the LLVM code base to detect the appropriate places to apply peephole optimizations and bugs in this functionality can result in corrupted user programs. This is a barrier to the rapid prototyping and evaluation of new optimizations.

CAnDL enables a workflow that is much more amenable to efficient compiler development: The compiler implementer only has to write CAnDL programs, which are then compiled by the CAnDL compiler fully automatically into C++ LLVM passes. These passes automatically spot code sections that adhere the specified structure, leaving only the transformation functionality to be implemented by hand. This provides a uniform manner in which to describe compiler analysis, reducing code length and complexity.

The approach crucially scales to a wide range of compiler analysis tasks, ranging from the detection of small scale local optimization opportunities over domain specific graphics optimizations to fully capturing polyhedral static control flow regions. In all cases, these tasks can be expressed more briefly than with competing approaches.

4.1 Introduction

Compilers are complex pieces of software responsible that proces source code in several stages to produce an efficient binary program. In order to generate fast programs, compilers rely on a complex mid end, in which program optimisations are applied to the user program after parsing and before code genreation. In this mid end, user code is typically represented in a static single assignment intermediate representation and is optimised by successively applying a wide range of optimization passes. There is generally obvious upper bound to the complexity of optimisations that are considered, instead compilers are limited by their increasing code complexity and by the necessity to reason at each stage about the soundness of the performed operations. It is crucial that optimisations retain the semantics of the original program, as otherwise the resulting binary might be compromised.

Implementations of compiler optimizations generally require two components: analysis and transformation. Analysis routines find candidate sections in user programs that could profit from specific transformations and verify that all conditions are satisfied in order to legally apply them. Transformations are then applied to the analysis results, often after consulting heuristic cost models to gauge the effect on runtime, code size and other metrics.

The analysis often contains much of the code complexity. For example, simple peephole optimizations in the LLVM `instcombine` pass contain approximately 30000 lines of complex C++ code, despite the transformations being simple. This impedes the implementation of new compiler passes, preventing the rapid prototyping of new ideas. Furthermore, this is an important source of bugs and bugs in this stage of the compiler are particularly pernicious, as they tamper with the user programs. Ideally, we would like a simpler way of describing such analysis that reduces boiler-plate code and opens the way for new compiler optimization innovation in a controlled fashion.

This chapter presents CAnDL, a domain specific language for compiler analysis. It is a constraint programming language that operates on the static single assignment intermediate representation of the LLVM compiler infrastructure (LLVM IR). Instead of writing compiler analysis code inside the main codebase of the compiler infrastructure, it enables compiler writers to specify optimization functionality external to the main C++ code base. The CAnDL compiler then generates C++ functions that are linked together with the clang compiler binary and that implement LLVM passs. The formulation of optimizing transformations in CAnDL is faster, simpler and less error prone than writing them in C++. It has a strong emphasis on modularity, which enables debugging and the formulation of highly readable code.

The implementation of CAnDL is based on the constraint programming methodology that was introduced in chapter 3, using a solver that is integrated directly into the LLVM code base. On top of this solver, a complete working system is provided, including a full parser and code generator for CAnDL as a full-fledged programming langauge.

4.2 Motivating Example

As an example of the CAnDL workflow, consider Equation 4.1. This basic algebraic equation can be interpreted as a recipe for a compiler optimisation: Assuming an environment without the particularities of floating point arithmetic (i.e. assuming the `-ffast-math` flag is active), the compiler could use this equality to eliminate some square root invocations in user code. This is desirable, as the square root has to be approximated with relatively expensive numerical methods, whereas computing the absolute value is computationally cheap.

$$\forall a \in \mathbb{R}: \ \sqrt{a*a} = |a| \tag{4.1}$$

The compiler should use the equation left to right: It should analyse the user code in order to find representations of the left side of the equation and then transform all those occurences analogous to the right side of the equation. The compiler therefor must detect occurrences of $\sqrt{a*a}$ in the LLVM IR code and replace them with a call to the `abs` function. The generation of the new function call is trivial, but the detection of even a simple pattern like $\sqrt{a*a}$ requires some care when implementing it manually in a complex code base such as LLVM.

The established approach would be to implement it as part of the `instcombine` pass, which applies a collection of such peephole optimisations and already extends to almost 30000 lines of C++ code. This code makes heavy use of raw pointers and dynamic type casts and has been identified as a frequent source of bugs by Yang et al. [104], Menendez and Nagarakatte [61]. This is impractical and an impediment to compiler development.

Instead, CAnDL allows a declarative description of the analysis problem, which is easier to follow, has no interaction with other optimisations and is much less verbose. This is shown in Figure 4.1. The specification is given the identifier `SqrtOfSquare` in the first line of the program and then defined by the interaction of seven atomic constraint statements that have to hold simultaneously hold on the values of `sqrt_call`, `sqrt_fn`, `square` and `a`. The lines 2-8 each stipulate one of these constraints and they are joined together with the logical conjunction operator "∧".

Figure 4.1: Simple Declarative CAnDL Specificaiton

```
1 Constraint SqrtOfSquare
2 ( opcode{sqrt_call} = call
3 ∧ {sqrt_call}.args[0] = {sqrt_fn}
4 ∧ function_name{sqrt_fn} = sqrt
5 ∧ {sqrt_call}.args[1]  = {square}
6 ∧ opcode{square} = fmul
7 ∧ {square}.args[0] = {a}
8 ∧ {square}.args[1] = {a})
9 End
```

The CAnDL compiler translates this declarative program into a C++ analysis function that is used as an LLVM optimisation pass. This is demonstrated in Figure 4.2, which shows the analysis function generated from the CAnDL specification in Figure 4.1 applied to a user program (**a**-**d**). The input program (**a**) is a simple C program that calls the `sqrt` function twice with squares of floating point values. It is translated using the clang compiler into LLVM IR code (**b**), applying standard optimisation passes during the process. This results in the expressions from the user program being sequentialised, with the two occurences of the `SqrtOfSquare` idiom clearly visible: The two `fmul` instructions compute squares via a floating point multiplication and these are then used as arguments to `sqrt` function invocations.

The analysis function detects two opportunities to apply the transformation, shown as first solution (**d**) and second solution (**e**). Each of the two solutions assigns values from within the LLVM IR code to each of the variables in the CAnDL program such that all the specified constraints are fulfilled. The validity of these solutions is demonstrated in the middle row of the figure (**e**-**f**). Subsituting the variables in the CAnDL programs with the concrete instantiations from the solutions, we can verify the individual atomic constraints one by one:

- `%4` and `%6` are function calls and their first argument (the function to be called) is `@sqrt`.
- `@sqrt` is the square root function. Note that we have no choice but to identify it by name.
- The second argument of the function call instruction (and hence the first actual function argument) are `%3` and `%5` respectively.
- `%3` and `%5` are square values, i.e. a floating point multiplication of a value with itself.

With the solutions identified by the CAnDL system, performing the transformation is now simple. The result of the analysis for each solution is provided in the form of a C++ dictionary `std::map<std::string,llvm::Value*>`, which contains the information required to apply appropriate code transformations (**g**). A new function call to `abs` is generated, with `a` as the only argument. This instruction then replaces the original call instruction that was captured in `sqrt_call`. After post processing with standard dead code elimination, this results in the optimized code shown at the bottom of the figure (**h**).

Although this is only a small example, it illustrates the main steps of the CAnDL scheme. In practice, the strength of the system is to scale to complex specifications, culminating in a full polyhedral analysis, which will be demonstrated towards the end of the chapter. The next sections give a specification of the CAnDL language and outline how it is implemented on top of the constraint programming methodology devised in chapter 3.

(a) C program code:

```
double example(double a, double b) {return sqrt(a*a) + sqrt(b*b); }
```

(b) Resulting LLVM IR:

```
1  define double @example(
2   double %0,
3   double %1) {
4   %3 = fmul double %0, %0
5   %4 = call double @sqrt(%3)
6   %5 = fmul double %1, %1
7   %6 = call double @sqrt(%5)
8   %7 = fadd double %4, %6
9   ret double %7 }
10 declare double @sqrt(double)
```

(c) First solution:

```
a = %0

square = %3
sqrt_call = %4

sqrt_fn = @sqrt
```

(d) Second solution:

```
a = %1

square = %5
sqrt_call = %6

sqrt_fn = @sqrt
```

(e) Verification of first solution:

```
( opcode{%4} = call
∧ {%4}.args[0] = {@sqrt}
∧ function_name{@sqrt} = sqrt
∧ {%4}.args[1]  = {%3}
∧ opcode{%3} = fmul
∧ {%3}.args[0] = {%0}
∧ {%3}.args[1] = {%0})
```

(f) Verification of second solution:

```
( opcode{%6} = call
∧ {%6}.args[0] = {@sqrt}
∧ function_name{@sqrt} = sqrt
∧ {%4}.args[1]  = {%5}
∧ opcode{%5} = fmul
∧ {%5}.args[0] = {%1}
∧ {%5}.args[1] = {%1})
```

(g) C++ transformation code:

```
1 using namespace std;
2 using namespace llvm;
3 void transform(map<string,Value*> solution, Function* abs) {
4     ReplaceInstWithInst(
5        dyn_cast<Instruction>(solution["sqrt_call"]),
6        CallInst::Create(abs, {solution["a"]}));
7 }
```

(h) Transformed LLVM IR after dead code elimination:

```
1 define double @example(double %0, double %1) {
2  %3 = call double @abs(double %0)
3  %4 = call double @abs(double %1)
4  %5 = fadd double %3, %4
5  ret double %5 }
```

Figure 4.2: Demonstration of CAnDL specification in Figure 4.1 on an example C program (**a**): In the generated LLVM IR code (**b**), two instances (**c**,**d**) of `SqrtOfSquare` are detected that fulfill all the constraints (**e**, **f**). Applying a transformation is simple (**g**) and results in efficient code (**h**).

4.3 Language Specification

The Compiler Analysis Description Language (CAnDL) is a domain specific programming language for the specification of compiler analysis problems. Individual CAnDL programs define computational structures that can be exploited by code transformations in optimising compilers. These structures are specified as constraint programs on static single assignment (SSA) form representation of user code, the most common intermediate representation in modern compilers.

The expressed structures can scale from simple instruction patterns that are suited for peephole optimizations over basic control flow structures such as loops to complex algorithmic concepts such as stencil codes with arbitrary kernel functions or code regions suitable for polyhedral analysis.

The basic building blocks of CAnDL programs are well known compiler analysis tools, such as constraints on data and control flow, data types and instruction opcodes. On top of these low level constraints, CAnDL employs powerful mechanisms for modularity and encapsulation that allow the construction of complex programs.

Like most constraint programming languages, CAnDL programs have two fundamental features: **variables** and **constraints**. These are composed with several higher level language features and, with a system of modularity and extensability on top. This section will introduce the language features from the top down, starting from the high level program structure.

4.3.1 High Level Structure of CAnDL Programs

An individual CAnDL program contains constraint formulas that are bound to identifiers. As we already saw in Figure 5.13 (a), the syntax for this is as follows:

Constraint ⟨**s**⟩ *formula* **End**

For the description of CAnDL syntax we use these notational conventions: terminal symbols are **bold**, non-terminals are *italic*, ⟨**s**⟩ is an identifier (alphanumeric string) and ⟨**n**⟩ is an integer literal.

We already saw in Figure 5.13 (a) that logical conjunctions can be used to combine *formula*s. More generally, a *formula* can be any of the following:

$$\begin{aligned} &\mathit{atomic} \mid \mathit{conjunction} \mid \mathit{disjunction} \\ \mid &\mathit{foreach} \mid \mathit{forany} \mid \mathit{include} \mid \mathit{collect} \end{aligned}$$

The basis of every CAnDL program are *atomic* constraints. For example in Figure 5.13 (a), lines 2-8 each specify individual atomic constraints. Atomic constraints are bound together by logical connectives $\wedge$ and $\vee$ (*conjunction* and *disjunction*) and other higher level constructs.

These include two kinds of loop structures (*foreach*, *forany*), as well as a system for modularity (`include`). Lastly, the *collect* construct allows for the formulation of more complex constraints that require the $\forall$ quantifier.

4.3.2 Atomic Constraints

The first type of constraint is an *atomic* constraint based on *variable*s. *Variable*s in CAnDL correspond to instructions and values in LLVM IR. Given some IR code, all occurring values can be assigned to the *variable*s of a given constraint formula. This includes instructions, globals, constants and function parameters. Syntactically, a variable is simply an identifier in curly brackets.

CAnDL uses the following atomic constraints:

data_type *variable* **= int**

data_type *variable* **= float**

This restricts the data type to integer or floating point.

ir_type *variable* **= literal**

ir_type *variable* **= argument**

ir_type *variable* **= instruction**

This restricts the type of IR node that is allowed to compile time constants, function arguments or instructions.

opcode *variable* **=** $\langle$**s**$\rangle$

This restricts the value instructions of the specified opcode.

function_name *variable* **=** $\langle$**s**$\rangle$

This restricts the variable to be a specified standard function (i.e. the `sqrt` function).

variable **=** variable

variable **!=** variable

This enforces two variables to have the same/not the same value. This is a shallow comparison, i.e. it compares whether two variables represent the same IR node.

control_origin *variable*

data_origin *variable*

The value is an origin of control (function entry) or data (function argument, `load` instruction, inpure function call).

$$\textit{variable}\textbf{.arg[}\,\langle\textbf{n}\rangle\,\textbf{]} = \textit{variable}$$
$$\textit{variable} \in \textit{variable}\textbf{.args}$$

There data flow from one value to the next.

$$\textit{variable}\ \textbf{->}\ \textit{variable}\ \Phi\ \textit{variable}$$

The left value has to reach the right value, which is a phi node, via the middle value, which is a jump instruction.

$$\textbf{domination(}\textit{variable}\textbf{,}\textit{variable}\textbf{)}$$
$$\textbf{strict\_domination(}\textit{variable}\textbf{,}\textit{variable}\textbf{)}$$

Both values have to be instructions and the first dominates the second in the control flow graph.

$$\textbf{calculated\_from(}\textit{varlist}\textbf{,}\textit{varlist}\textbf{,}\textit{variable}\textbf{)}$$

Varlist is a set of one or multiple *variables*. Any path from one of the entries in the first *varlist* to the single *variable* argument has to pass through at least one of the entries in the second *varlist*. All paths in the union of the data flow and control dependence graph are considered. We will see later how this is useful to specify kernel functions for e.g. stencil calculations.

There are some *atomic*s that we omit for space reasons. The set of *atomic*s that CAnDL supports is easily extensible. Possible additions include constraints on function attributes, value constraints on literals etc.

4.3.3 Range Constraints

Building on top of the basic conjunction and disjunction constructs, there are range based versions that operate on arrays of variables.

$$\textit{formula}\ \textbf{foreach}\ \langle\textbf{s}\rangle\ \textbf{=}\ \textit{index}\ \textbf{..}\ \textit{index}$$
$$\textit{formula}\ \textbf{forany}\ \langle\textbf{s}\rangle\ \textbf{=}\ \textit{index}\ \textbf{..}\ \textit{index}$$

These constructs allow the repeated application of a formula according to some range of indices. This is demonstrated by Figure 4.3, which shows two equivalent CAnDL programs, one formulated with `foreach` and one without. In both cases, the program specifies an array of five variables with data flow from each element to the next. We can see how the `foreach` loop can be expanded similar to loop unrolling.

UNDERFUL VBOX.

UNDERFUL VBOX.

```
Constraint ValueChain
 {element[i] ∈ {element[i+1]}.args foreach i=0..4
End
```

```
Constraint ValueChain
( {element[0]} ∈ {element[1]}.args
∧ {element[1]} ∈ {element[2]}.args
∧ {element[2]} ∈ {element[3]}.args
∧ {element[3]} ∈ {element[4]}.args)
End
```

Figure 4.3: Expansion of range constraints in CAnDL

4.3.3.1 Modularity

Modularity is central to the CAnDL programming language, and it is achieved using the *include* construct.

include ⟨**s**⟩
[(*variable* **->** *variable* {**,** *variable* **->** *variable*})]
[@ *variable*]

Note that the syntax in square brackets is optional and the syntax in curly brackets can be repeated. The basic version of *include*, without the optional structures, is simple. It copies the *formula* that corresponds to the identifier verbatim into another *formula*. If [@ *variable*] is specified, then all the variable names of the inserted constraint formula are prefixed with the given variable name, separated with a dot. The other optional syntax is used to rename individual *variable*s in the included *formula*.

Figure 4.4 illustrates this with two equivalent programs. Both programs specify an addition of four values, first adding pairwise and then adding the intermediate results. We can see in the first listing that a *formula* for the addition of two values is bound to the name `Sum`. This is then included three times in another *formula* names `SumOfSums`. Using the optional grammatical constructs, the formula operates on a different set of *variable*s each time such that the third addition takes the results of the previous two as input.

4.3.3.2 Collect

The *collect* construct is used to capture all possible solutions of a given formula. It is used to implement constraints that require the logical ∀ quantifier. For example, it can be used to guarantee that all memory accesses in a loop use affine index computations. The grammar is simple but the semantics require some elaboration.

collect ⟨**s**⟩ *index formula*

```
Constraint Sum
( opcode{out} = add
∧ {out}.args[0] = {in1}
∧ {out}.args[1] = {in2})
End
Constraint SumOfSums
( include Sum@{sum1}
∧ include Sum@{sum2}
∧ include Sum({sum1.out}->{in1},{sum2.out}->{in2}))
End
```

```
Constraint SumOfSums
( opcode{sum1.out} = add
∧ {sum1.out}.args[0] = {sum1.in1}
∧ {sum1.out}.args[1] = {sum1.in2}
∧ opcode{sum2.out} = add
∧ {sum2.out}.args[0] = {sum2.in1}
∧ {sum2.out}.args[1] = {sum2.in2}
∧ opcode{out} = add
∧ {out}.args[0] = {sum1.out}
∧ {out}.args[1] = {sum2.out})
End
```

Figure 4.4: Expansion of Inheritance in CAnDL

In Figure 4.5, the variables `arg[0],...,arg[N-1]` are constraint to contain all of the data dependences of `ins`. The first argument of *collect* specifies the name of an index variable that is used to detect which variables belong to the collected set. In this example we want all solutions of `arg[i]` for a given value of `ins`. The second argument gives an upper bound to the amount of collected variables, in this case we leave it unspecified by using the symbol `N`.

```
Constraint CollectArguments
( ir_type{ins} = instruction
∧ collect i N ({arg[i]} ∈ {ins}.args))
End
```

Figure 4.5: Simple collect example in CAnDL

We can now extend this example to show how *collect* can be used to implement quantifiers. Consider that we want to detect instructions with only floating point data dependences. This involves the $\forall$ quantifier, as it is equivalent to the following equation.

$$\forall v\colon\ v \in I.\text{args} \implies \text{data\_type}(v) = \text{float} \tag{4.2}$$

We can rewrite this to an equivalent formulation on sets.

$$S_1 := \{v \mid v \in I.\text{args}\} \subset S_2 := \{v \mid \text{data\_type}(v) = \text{float}\}$$

Now we can apply the following equivalences:

$$S_1 \subset S_2 \Leftrightarrow S_1 = S_1 \cap S_2$$
$$\Leftrightarrow \exists S\colon S = S_1 \wedge S = S_1 \cap S_2$$

This means that if we constraint a set S to be equal to both S_1 and $S_1 \cap S_2$ at the same time, the constraints are satisfiable if and only if the implication in Equation 4.2 holds.

This condition can be expressed in CAnDL, as is shown in Figure 4.6. With the first *collect* statement in line 3, we constrain the set `arg` to be equal to S_1 and with the second one in lines 4-5 we constrain it to be $S_1 \cap S_2$ as well. Note that we were from the onset only interested in the values that qualify for `ins`. The set `arg` was only introduced to further constraint `ins`, not because we actually wanted to know the values that it contains.

```
1 Constraint FloatingPointInstruction
2 ( ir_type{ins} = instruction
3 ∧ collect i N ( {ins} ∈ {arg[i]}.args)
4 ∧ collect i N ( {ins} ∈ {arg[i]}.args
5                ∧ data_type{arg[i]} = float))
6 End
```

Figure 4.6: Collect Example in CAnDL

The exact same approach can be used to e.g. restrict all array accesses in a loop to be affine in the loop iterators. This can be achieved by first *collect*ing all memory accesses (i.e. all `load` and `store` instrutions) and then using another *collect* statement to stipulate affine calculations for the indices.

4.3.4 Expressing Larger Structures

The modularity of CAnDL allows the creation of a library of building blocks that are shared by multiple CAnDL programs. We will now give an overview about how these can be defined with CAnDL.

Important building blocks include control flow structures such as single entry single exit regions and loops. These are standard in compiler analysis and the implementation in CAnDL is straightforward. A for loop involves a comparison of the loop iterator with the end of the iteration space. In order to be valid, this value has to be determined before the loop is entered, it isn't allowed to change from loop iteration to iteration. This leaves it to be either a function argument, an actual constant or an instruction that strictly dominates the loop entry. This is expressed in Figure 4.7. Note that this formula is to be included into larger CAnDL programs, as the `begin` variable is under specified otherwise.

Another class of important building blocks are different categories of memory access. These form a hierarchy of restrictiveness and include multidimensional array access and array

```
Constraint LocalConst
( ir_type{value} = literal
∨ ir_type{value} = argument
∨ strict_domination({value}, {begin}))
End
```

Figure 4.7: LocalConst in CAnDL

access that is affine in some loop iterators. LLVM strictly separates memory access from pointer computations, which means that CAnDL only has to concern itself with plain pointer computations here. In general it is required that the base pointer is `LocalConst` in order to avoid pointer chases. The index computation can then be described using the data flow and instruction opcode restrictions.

To enable the capture of higher order functions such as stencils or reduction operations, we need to handle arbitrary kernel functions. Kernel functions are sections of code that are side-effect free and can be separated out. Identifying side-effect free code is useful in many types of compiler optimization. It can be expressed in CAnDL, as shown in Figure 4.8.

```
Constraint KernelFunction
( collect i N
∧ ( include LocalConst({outer}->{begin}*@*{closure[i]}
  ∧ ir_type{closure[i].value} != literal
  ∧ {closure[i].use} ∈ {closure[i].value}.args
  ∧ domination({inner}, {closure[i].use}))
∧ collect i N data_origin{tainted1[i]}
∧ collect i N
∧ ( domination({outer}, {tainted2[i]})
  ∧ strict_domination({tainted2[i]}, {inner}))
∧ calculated_from(
    {tainted1[0..N],tainted2[0..N]},
    {origin[0..N],closure[0..N].value,input[0..N]},
    {output}))
End
```

Figure 4.8: CAnDL Formulation of Kernel Functions

Essentially, this set of contraints captures the computation of a single `output` value from a set of specified `input` values as well as a set of automatically captured `closure` variables. The computation for `output` should be such that it can be replaced with a function call that is free of side effects and takes only the `input` and `closure` variables as arguments.

In addition to these variables, there are two non-obvious additional variables involved: `outer` and `inner`. These set boundaries in the control flow as follows: Closure values have to be computed before `outer` and the computation that results in outer is performed after `inner`. Usually these will be derived from loops nests, where `outer` is the entry to the outermost loop

and `inner` is the entry to the innermost loop. The actual core constraint is then a generalized dominance relationship in the program dependence graph.

The use of the sets `tainted1` and `tainted2` makes sure that no impure functions are used in computing `output` and the entire computation is performed after `inner` (ruling out e.g. loop carried variabls from outer loops).

UNDERFUL VBOX.

UNDERFUL VBOX.

UNDERFUL VBOX.

4.4 Implementation

CAnDL interacts with the LLVM framework, as shown in Figure 4.9. CAnDL programs are read by the CAnDL compiler, which then generates C++ source code to implement the specified LLVM analysis functionality. This code depends on a generic backtracking solver, which is incorporated into the main LLVM code base. We will see in the evaluation section this this solver adds little compile-time overhead in practice. The generated code is compiled and linked together with the existing LLVM libraries to make LLVM optimization passes available in the clang compiler.

The generated analysis passes use the solver to search for the specified computational structures and output the found instances into report files, as well as making them available to ensuing transformation passes.

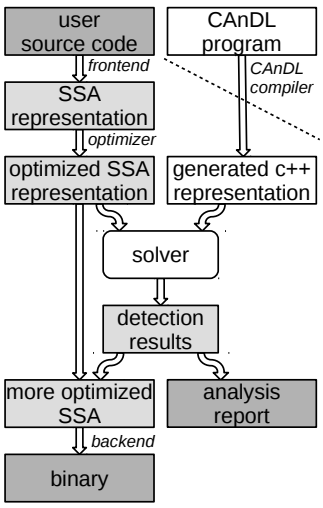


Figure 4.9: CAnDL in the LLVM/clang build system

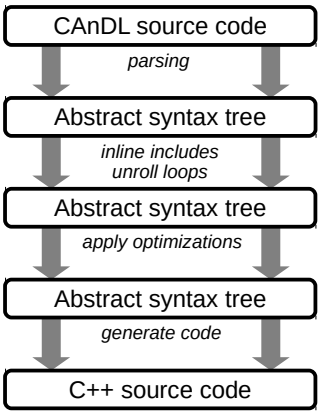


Figure 4.10: CAnDL compiler flow

4.4.1 The CAnDL Compiler

The CAnDL compiler is responsible for generating C++ code from CAnDL programs. An overview of its flow is shown in Figure 4.10. The frontend reads in CAnDL source code and builds an abstract syntax tree. This syntax tree is simplified in two steps to eliminate some of the higher order constructs of CAnDL. The `inheritance` clauses are replaced using standard function inlining after the contained variables have been transformed accordingly. Also, `foreach` and `forany` statements are lowered to conjunction and disjunction constructs by duplicating the contained constraint code and then renaming the contained variable names appropriately for each iteration. This is equivalent to complete loop unrolling. From this point onwards, variable names are treated as flat strings. The remaining core language now consists only of atomics, conjunctions, disjunctions and collections.

The CAnDL compiler applies a set of basic optimizations to speed up the solving process using the later generated C++ code. For example, nested conjunctions and disjunctions are flattened wherever possible.

Finally, the compiler generates the C++ source code. This essentially involves constructing the constraint problem as a graph structure that is accessible to the solver.

4.4.1.1 C++ Code Generation

We demonstrate the code generation process in Figure 4.11 with an example. Each of the atomic constraints results in a line of C++ code that constructs an object of a corresponding class: In this case, the three involved atomic constraints are implemented by `AddInstruction`, `FirstArgument` and `SecondArgument`. For constraints that involve more than one variable, these objects are instantiated as shared pointers.

The compiler then generates similar objects for the `conjunction`, `disjunction` and the `collect` structures. In our example, this only affects the variable `addition`, which is part

of a `conjunction` clause. This results in an additional object construction that instantiates the `And` class corresponding to the ∧ operator in CAnDL. The `select` function is used here to specify which variable of a constraint is being operated on. In this case, `addition` is the second variable in lines 3-4 of the CAnDL program, so we use `select<1>`.

Finally, the generated objects are inserted into a vector together with the corresponding variable names. This vector is then passed to the solver.

```
1  Constraint SimpleAddition
2  ( opcode{addition} = add
3  ∧ {addition}.args[0] = {left}
4  ∧ {addition}.args[1] = {right})
5  End
```

```
1  auto constr0 = AddInstruction(context);
2  auto constr1 = make_shared<FirstArgument>(context);
3  auto constr2 = make_shared<SecondArgument>(context);
4  auto constr3 = And(constr0, select<1>(constr1),
5                              select<1>(constr2));
6
7  vector<pair<string,SolverAtomContainer>> result(3);
8  result.emplace_back("addition", constr3);
9  result.emplace_back("left", select<0>(constr1));
10 result.emplace_back("right", select<0>(constr2));
```

Figure 4.11: C++ source code generation

4.4.2 The Solver

The solver takes LLVM IR code and a graph representation of the constraint problem as constructed by the generated code. We saw in Figure 4.11 that this representation comes in the form of a vector of labeled instances of a class called `SolverAtomContainer`. This class wraps around the class `SolverAtom` that is defined in Figure 4.12.

```
1  class SolverAtom {
2  public:
3    virtual SkipResult skip_invalid(unsigned& c)const;
4
5    virtual void begin();
6    virtual void fixate(unsigned c);
7    virtual void resume();
8  };
```

Figure 4.12: The SolverAtom interface.

The motivation for this interface is as follows: The solver operates on unsigned integers,

```
1  i := 0
2  while i >= 0:
3      result := atom[i].skip_invalid(solution[i])
4
5      if result = SkipResult::PASS:
6          atom[i].fixate(solution[i])
7
8          if i+1 = N:
9              return solution
10         else:
11             i := i+1
12             solver_atom[i].begin()
13             solution[i] := 0
14
15     if result = SkipResult::FAIL:
16         i := i-1
17         atom[i]->resume()
18         solution[i] := solution[i]+1
```

Figure 4.13: Pseudocode of the backtracking solver

using the `skip_invalid` method to search for partial solutions. The corresponding LLVM values are numbered consecutively and the unsigned integers simply represent indices into that enumeration. When `skip_invalid` returns `FAIL`, the solver backtracks. The other member functions `begin`, `fixate` and `resume` allow implementations of the `SolverAtom` interface to do bookkeeping.

Pseudocode for the backtracking constraint solver is shown in Figure 4.13. The array of `SolverAtom`s is named `atom`, `solution` is an array of integers that is incrementally filled with a solution to the constraint problem. In line 3, we can see that the `skip_invalid` method is used to find the next candidate solution for the `i`th element of the solution, taking into account all the previously established elements `solution[0..i-1]`. The candidate solution is directly stored in `solution[i]`, which is passed by reference as shown in Figure 4.12. If the step was successful, then the solver either returns the solution if it is complete or it moves on the the next element (line 11), otherwise it backtracks (line 16). The member function `fixate`, `begin` and `resume` are called appropriately in lines 6, 12 and 17.

We can see that the solver is generic and requires no knowledge of LLVM or the structure of variable names in CAnDL. Furthermore, the solver is unaware not only of the underlying LLVM structure and the corresponding atomic constraints, but also of the `conjunction` and `disjunction`, as well as the `collect` constructs.

The complexity of the solving process lies almost entirely in the implementation of the `SolverAtom` interface for the different atomic constraints and their interactions. For simple constraints like the **is an add instruction** statement, this is straightforward and `skip_invalid`

only has to test whether the value at index n in the function is an add instruction or not. This can be implemented as shown in Figure 4.14.

```
1  class AddInstruction : public SolverAtom {
2  public:
3    AddInstruction(Context&) { ... }
4
5    SkipResult skip_invalid(unsigned& c) const
6    {
7        if(c >= value_list->size())
8          return SkipResult::FAIL;
9
10       if(auto inst = dyn_cast<Instruction>
11                        ((*value_list)[c]))
12       {
13         if(inst->getOpcode() == Instruction::Add)
14           return SkipResult::PASS;
15       }
16
17       c=c+1;
18       return SkipResult::CHANGE;
19   }
20
21   void begin() {}
22   void fixate(unsigned c) {}
23   void resume() {}
24
25 private:
26   shared_ptr<vector<Value*>> value_list;
27 };
```

Figure 4.14: SolverAtom for additions

4.4.3 Developer Tools

CAnDL makes writing compiler analysis passes easier, but reasoning about the semantics of compiler IR still remains difficult and the correctness of CAnDL programs can only be guaranteed with thorough testing. It is important to keep in mind that CAnDL is targeted at expert compiler developers.

In order to make debugging of CAnDL programs more feasible, we developed supporting tools. Most importantly, this includes an interactive gui, where developers can test out corner cases of CAnDL programs to find false positives and false negatives. This gui is shown in Figure 4.15, the example is from one of the use cases presented in a section 4.5.

In the left column, we can see part of a C program from the PolyBench benchmark suite, which implements a two-dimensional Jacobi stencil. The gui is configured to look for static

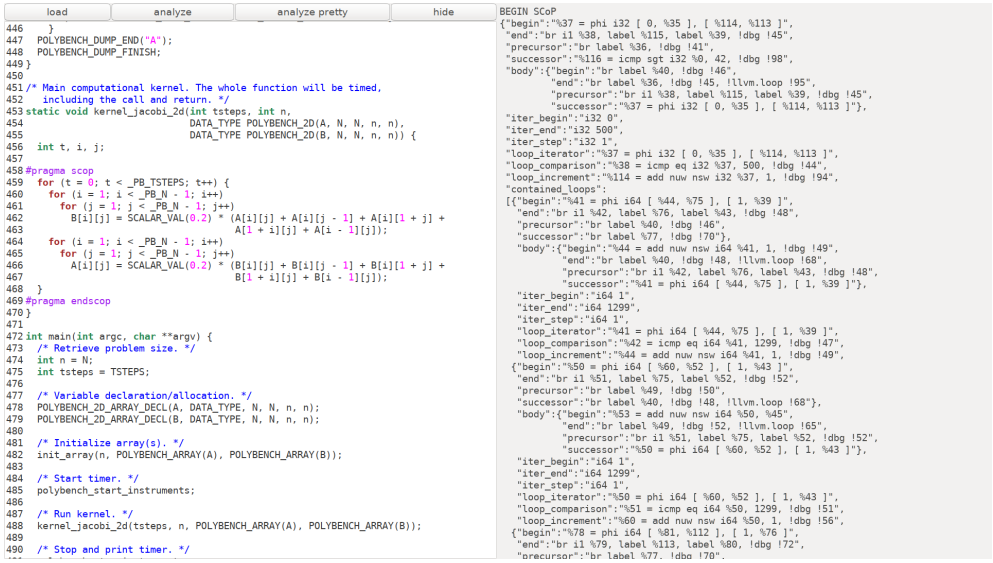


Figure 4.15: Interactive CAnDL test tool

Left hand panel shows a SCoP in Polybench jacobi-2d, the right hand panel show the corresponding constraint solution

control flow regions (SCoPs), as described later in one of the use cases. The user has clicked the "analyze" button, which triggered the analysis to run and prints the results in the right column.

The solver found a SCoP in the IR code (corresponding to lines 459-468 of the C program). The text on the right shows the hierarchical structure of the solution, with IR values assigned to every variable. The corresponding C entities can be recovered using the debug information that is contained in the generated LLVM IR code. By modifying the C code, the developer can now test the detection and e.g. verify that no SCoP is detected if irregular control flow is introduced.

4.5 Case Studies

We evaluate the usefulness of CAnDL on three real life use cases. We first use it for simple peephole optimizations. We then apply CAnDL to graphics shader code optimization. Finally, we demonstrate the detection of static control flow parts (SCoPs) for polyhedral code analysis. Where possible, we compare the number of lines of CAnDL code, the achieved program coverage and performance against prior approaches.

4.5.1 Simple Optimizations

Arithmetic simplifications in LLVM are implemented in the `instcombine` pass. One example of this is the standard factorization optimization that uses the law of distributivity to simplify integer calculations as shown in Equation 4.3. It is implemented in 203 lines of code and

```
Constraint ComplexFactorization
( opcode{value} = add
∧ {value}.args[0] = {sum1.value}
∧ {value}.args[1] = {sum2.value}
∧ include SumChain at {sum1}
∧ {mul1.value} = {sum1.last_factor}
∧ include MulChain at {mul1}
∧ {mul1.last_factor} = {mul2.last_factor}
∧ include SumChain at {sum2}
∧ {mul2.value} = {sum2.last_factor}
∧ include MulChain at {mul2})
End
```

Figure 4.16: ComplexFactorization in CAnDL

furthermore uses supporting functionality provided by `instcombine`.

$$a*b+a*c \rightarrow a*(b+c) \tag{4.3}$$

This analysis problem can be formulated in CAnDL as shown in Figure 4.16. The CAnDL program even captures a much larger class of opportunities for factorization than `instcombine`, which require to first apply associative and commutative laws to reorder the values. This is for example needed in the case of Equation 4.4 and only partially supported by LLVM with the additional `reassociate` pass.

$$a*b+c+d*a*e- > a*(b+d*e)+c \tag{4.4}$$

We evaluated the program in Figure 4.16 against the default factorization optimization in LLVM's `instcombine` on three benchmark suites: the sequential C versions of NPB, the C versions of Parboil and the OpenMP C/C++ versions of Rodinia. We extended the existing LLVM `instcombine` pass such that it automatically reports every time that it successfully applies the `tryFactorization` function.

We compiled all the individual benchmark programs in the three benchmark suites, which consist of 94915 lines of code in total. For each benchmark suite, we added up all factorizations that were reported. We also measured LLVM's total compilation time.

We then disabled the standard LLVM optimization and instead used the CAnDL generated detection functionality. We compiled the same application programs reporting the number of factorizations found and again measured the total compilation time this time using CAnDL. Note that this compilation time includes all the other passes within LLVM as well as the CAnDL generated path.

	LLVM	CAnDL
Lines of Code	203	12
Detected in NPB	1	1 + 2
Detected in Parboil	0	0 + 1
Detected in	24	24 + 4
Total Compilation time	152.2s	152.2s+7.8s

Figure 4.17: Factorizations LLVM vs CAnDL

4.5.1.1 Results

The results are shown in Figure 4.17. In general, the performance impact of peephole optimizations is small and in two of the benchmark sets we find only very small numbers. LLVM was unable to perform any factorization in the entire Parboil suite. However, the Rodinia suite contains more opportunities, mostly in the Particlefilter and Mummergpu programs.

In all three benchmarks suites, our scheme finds the same factorization opportunities as the `instcombine` pass plus an additional 7 cases. With only 12 lines of CAnDL code, we were able to capture more factorization opportunities than LLVM did using two hundred lines of code.

Using CAnDL on large complex benchmark suites only increased total compilation time by 5%, demonstrating its use as a prototyping tool.

4.5.2 Graphics Shader Optimizations

Graphics computations often involve arithmetic on vectors of single precision floating point values that represent either vertex positions in space or color values. Common graphics shader compilers utilize the LLVM framework using the LunarGLASS project.

For real shader code, LLVM misses an opportunity for the associative reordering of floating point computations. Although such reordering is problematic in general, it is applicable in the domain of graphics processing.

There are often products of multiple floating point vectors, where several of the factors are actually scalars that were hoisted to vectors. By reordering the factors and delaying the hoisting to vectors, some of the vector products can be simplified to scalar products, as shown in the following equation.

$$\begin{aligned}\vec{x} &= \vec{a} *_v \vec{b} *_v \text{vec3}(c) *_v \vec{d} *_v \text{vec3}(e) \\ &= \text{vec3}(c * e) * \vec{a} *_v \vec{b} *_v \vec{d}\end{aligned}$$

We implemented the required analysis functionality for this optimization with CAnDL, as shown in Figure 4.18. The included `VectorMulChain` program discovers chains of floating

point vector multiplications in the IR code and uses the variables `factors` and `partials` such that

$$\texttt{partials}[0] = \texttt{factors}[0]$$
$$\texttt{partials}[i+1] = \texttt{partials}[i] \times \texttt{factors}[i+1].$$

The `VectorMulChain` program furthermore guarantees that this is a chain of maximal length by checking that neither of the first two factors are multiplications and the last factor is not used in any multiplication. `ScalarHoist` detects the hoisting of scalars to vectors and this is used to collect all hoisted factors into the array `hoisted`. In a last step, all other factors are collected into the array `nonhoisted`.

```
1  Constraint FloatingPointAssociativeReorder
2  ( include VectorMulChain and
3  ∧ collect j N
4  ∧ ( {hoisted[k]} = {factors[i]} forsome i=0..N
5    ∧ include ScalarHoist({hoisted[j]}->{out},
6                          {scalar[j]}->{in})@{hoist[j]})
7  ∧ collect j N
8    ( {nonhoisted[j]}  = {factors[i]} forsome i=0..N
9    ∧ {nonhoisted[j]} != {hoisted[i]} forall  i=0..N))
10 End
```

Figure 4.18: CAnDL code for vectorized multiplication chains

The corresponding transformation pass simply generates all the appropriate scalar and vector multiplications and replaces the old result with this newly generated one. We evaluated the performance impact on the CFXBench 4.0 on the Qualcomm Adreno 530 GPU.

4.5.2.1 Results

The optimization was relevant to 8 of the fragment shaders in GFXBench 4.0. The number of lines of code needed and the resulting performance impact are shown in Figure 4.19 and Figure 4.20. A total of 19 opportunities for the optimization to be applied were detected. Although the performance impact was moderate with $1-4\%$ speedup on eight of the fragment shaders, it shows how new analysis can be rapidly prototyped and evaluated with only a few lines of code.

4.5.3 Polyhedral SCoPs

The polyhedral model allows compilers to utilize powerful mathematical reasoning to detect parallelism opportunities in sequential code and to implement code transformations for well structured nested loops. It is currently applicable to Static Control Flow Parts (SCoPs) with

	LLVM	CAnDL
Lines of Code	Not implemented	10
Detected in GFX 4.0	-	19

Figure 4.19: Shader optimization LLVM vs CAnDL

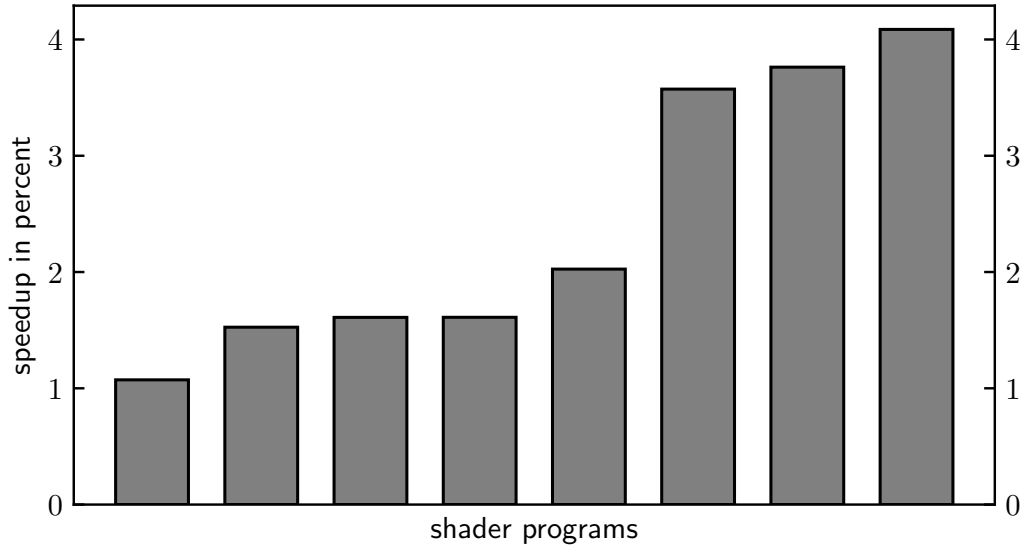


Figure 4.20: Speedup on Qualcomm Adreno 530

affine data dependencies. Detecting SCoPs is fundamental and a necessary first step for any later polyhedral optimization.

Implementations of the polyhedral model may differ in their precise definition of SCoPs. We implemented SCoP detection functionality in CAnDL and compared against the Polly implementation in LLVM. We rely on the definition of Semantic SCoPs in Grosser et al. [35]. The constraints for SCoPs can be broken into several components:

UNDERFUL VBOX.

4.5.3.0.1 Structured Control Flow SCoPs require well structured control flow. Technically speaking, this means that every conditional jump within the corresponding piece of IR is accounted for by for loops and conditionals. We enforce this with the `collect` statement as demonstrated in Figure 4.6. We use it in CAnDL programs `ForLoop` and `Conditional` that describe the control flow of for loops and conditionals and extract the involved conditional jump instructions. We then use another `collect` to verify that these are indeed all conditional jumps within the potential SCoP.

Once we have established the control flow, we can use the iterators that are involved in the loops to define affine integer computations in the loop. This is done in a brute force fashion with a recursive constraint program. Using this analysis we then check that the iteration domain

of all the for loops is well behaved, i.e. the boundaries are affine in the loop iterators.

4.5.3.0.2 Affine Memory Access We want to make sure that all memory access in the SCoP is affine. For this to be true we have to verify that for each load and store instruction, the base pointer is loop invariant and the index is calculated affinely. The loop invariant base pointer is easily guaranteed using the `LocalConst` program from Figure 4.7.

Checking the index calculations is more involved and is again based on the `collect` method that was demonstrated in Figure 4.6. We use the `collect` construct to find all of the affine memory accesses in all the loop nests. We then use collect all `load` and `store` instructions and verify that both collections are identical.

4.5.3.0.3 We evaluated our detection of SCoPs on the PolyBench suite. For both our method as well as for Polly, we counted how many of the computational kernels contained in the benchmark suite are captured by the analysis.

4.5.3.1 Results

As is visible in Figure 4.21, we capture all the SCoPs that Polly was able to detect. There is some postprocessing of the generated constraint solutions required to achieve this. Firstly, our results are not in the jscop format that Polly uses but contain the raw constraint solution as shown on the right side of Figure 4.15. Also, our CAnDL implementation does not merge consecutive outer level loops into SCoPs of maximum size. To compare, we extracted the detected loops from our report files, manually grouped them into maximum size SCoPs and verified that they fully cover the SCoPs detected by Polly.

To measure lines of code, we compared our version with the amount of code in Polly's `ScopDetection.cpp` pass. We are able to detect the same number of SCoPs with much fewer lines of code. Note that the LoC count that we give for our SCoP program does not include all CAnDL code involved in the detection of polyhedral regions. We consider the code that is not specific to this idiom (such as loop structures) to be part of the CAnDL standard library. In the same way we did not account for e.g the ScalarEvolution pass when counting the lines for Polly.

By having a high level representation of SCoPs, we allow polyhedral compiler researchers to explore the impact of relaxing or tightening the exact definition of SCoPs in a straightforward manner, enabling rapid prototyping.

	Polly	CAnDL
Lines of Code	1903	45
Detected in datamining	2	2
Detected in Linear-algebra	19	19
Detected in medley	3	3
Detected in stencils	6	6

Figure 4.21: SCoPs detected Polly vs CAnDL

4.6 Conclusion

Optimizing compilers require sophisticated program analysis in order to generate performant code. The current way of implementing this functionality manually in programming languages such as C++ is not satisfactory.

The domain specific Compiler Analysis Description Language (CAnDL) provides a more efficient approach. CAnDL programs can be used to automatically generate compiler analysis passes. They are easier to write and reduce the code size and complexity when comparing against manual C++ implementations.

Although CAnDL is based on a constraint programming paradigm and uses a backtracking solver to analyze LLVM IR code, the use of CAnDL results in only moderate compile time increases. Many compiler analysis tasks are suitable for CAnDL, from peephole optimizations to sophisticated program analysis with the polyhedral model.

Future work will investigate how formal verification methods can be applied to CAnDL in order to guarantee the correctness of compiler optimizations. Also, there is some overlap between what can be formulated in CAnDL and what is provided by LLVM in the form of the ScalarEvolution pass. We are interested as to whether we could use this existing functionality to speed up solving times.

Chapter 5

Formalizing Computational Idioms for Heterogeneous Acceleration

In the previous chapter, the CAnDL constraint programming language was introduced, which can generate compiler analysis functionality from terse and easily scalable, declarative problem specifications. This chapter takes a broader view of what automatic compiler analysis can entail with such a powerful tool at hand. Being freed from having to implement complex optimisation passes manually, it explores the complexity to which CAnDL can be driven, capturing broader computational concepts than were demonstrated previously.

The goal of this chapter is to take abstract algorithmic concepts that are conventionally explored outside the context of compiler analysis – computational idioms – and to formalize them as CAnDL specifications, enabling detection and manipulation in optimising compilers. Heterogeneous acceleration will serve as the motivation for this effort. As many scientific codes are structured around idiomatic performance bottlenecks, efforts that focus on computational idioms can greatly improve performance, especially with accelerators that were designed with similar computations in mind. The focus is therefore on calculations that are well supported by accelerators and their software ecosystems: sparse and dense linear algebra, stencil codes and generalised reductions and histograms.

The CAnDL specifications are used to build a prototype compiler that automatically detects the idioms and uses them to circumvent the code generator with libraries and domain specific languages: BLAS implementations, cuSPARSE, clSPARSE, Halide and Lift. The funcionality is directly accessible in a modified version of the widely used clang C/C++ compilers. The evaluation can therefore be performed on the well established benchmark suites NAS and Parboil, where 60 idiom instances are detected. In those cases where idioms are a significant part of the sequential execution time, the generated heterogeneous code achieves 1.26× to over 20× speedup on integrated and external GPUs.

5.1 Specification of Idioms in CAnDL

The specification of computational idioms in CAnDL requires a careful handling of the arising complexity, using the modularity functionality that CAnDL provides.

Control flow constructs, memory access patters as well as basic data flow patterns are specified independently and then combined together in order to define the complete constraint specifications.

5.2 Basic Control Flow Structures

The most basic control flow structure that is needed for expressing computational idioms is the single entry single exit (SESE) region. Figure 5.1 shows the specification, which contains some non-obvious considerations.

```
1  Constraint SESE
2  ( {precursor} is branch instruction and
3    {precursor} has control flow to {begin} and
4    {end} is branch instruction and
5    {end} has control flow to {successor} and
6    {begin} control flow dominates {end} and
7    {end} control flow post dominates {begin} and
8    {precursor} strictly control flow dominates {begin} and
9    {successor} strictly control flow post dominates {end} and
10   all control flow from {begin} to {precursor} passes through {end} and
11   all control flow from {successor} to {end} passes through {begin})
12 End
```

Figure 5.1: Single Entry Single Exit Regions

The necessity for `begin` and `end` is clear, these are the instructions that bound the SESE region. The other two variables are equally important, as they define the two crucial edges in the control flow. The edge from `precursor` to `begin` is the single entry and the edge from `end` to `sucessor` is the single exit of the SESE region.

Lines 6 and 7 ensure that `begin` and `end` acually span a region of code in the sense that the control flow will always reach `begin` before `end` and will always eventually reach `end` after passing through `begin`. This is complemented by lines 10 and 11, which guarantee that the region can only be left once after it is entered and can only be entered once before it has to be left. Lines 3 and 8 ensure that `precursor` is in fact the only instruction that reaches `begin` from outside the region, lines 4 and 9 check the corresponding property for `sucessor`. Lines 2 and 4 add as an additional constraint that the SESE region has to be aligned to basic block boundaries.

5.3 Loop Structures

Building on top of the SESE region, Figure 5.2 presents the CAnDL specification of a basic loop structure. Using the CAnDL description of a SESE region, this is a very simple definition: A Loop is simply a SESE region with a backedge.

```
1  Constraint Loop
2  ( inherits SESE and
3    {end} has control flow to {begin})
4  End
```

Figure 5.2: Loop in CAnDL

More interesting are for loops with well behaved interation spaces. They are distinguished by a specific kind of breaking condition that is based on an induction variable reaching a limit value, which can be expressed in CAnDL using a set of further constraints shown in Figure 5.3.

```
1  Constraint ForLoop
2  ( inherits Loop at {loop} and
3    inherits InductionVar
4        with {iterator}  as {old_ind}
5         and {increment} as {new_ind} at {loop} and
6    {iter_begin} has data flow to {iterator} and
7    inherits LocalConst
8        with {iter_begin} as {value}
9         and {loop.begin} as {begin} and
10   inherits OffsetAdd
11       with {increment} as {value}
12        and {iterator}  as {base}
13        and {iter_step} as {offset} at {loop} and
14   {comparison} is first argument of {loop.end} and
15   inherits RangeCheck
16       with {comparison} as {value}
17        and {increment}  as {input}
18        and {iter_end}   as {limit} at {loop} and
19 End
```

Figure 5.3: For Loop in CAnDL

As opposed to the previous CAnDL programs, this one makes extensive use of the modularity and simply binds together a range of simpler construct into a high level definition. Several additional variables are introduced: `increment` is instruction that increased the value of the `iterator` for the next iteration using the offset `iter_step`.

With the definition of IDL, we can now specify idioms. The complete set of idioms used in this paper comprises of ≈500 lines of code. Due to space restrictions, we first show a simple

constraint that we rely on – single entry, single exit regions – and then describe the top level constraints for each idiom.

5.3.1 Building Blocks

Before any algorithmic idiom can be specified, we need some basic control flow constructs. The most fundamental is the single entry single exit region (SESE) [45] which is used amongst other things to determine loop bodies. A SESE region is a part of code spanned by two instructions A and B such that A dominates B, B postdominates A and every cycle containing either A or B also contains the other. It is defined in Figure 5.5.

Using simple building blocks such as SESE, we can define more complex control structures such as loops and important memory access patterns such as matrix reads. From this we build powerful idiom definitions that capture complex computational patterns that can include arbitrary control flow.

5.3.2 Full Idiom Definition

The generalized matrix multiplication idiom is described in Figure 5.6. The control flow is captured by three nested for loops. Inside these loops, the memory access is characterized by three matrix accesses, each with a different subset of the loop iterators. The corresponding `MatrixRead` and `MatrixWrite` idioms model generic access to matrices allowing strides, transposed matrices etc. The actual computation is encapsulated by the `DotProductLoop` idiom. This also contains the linear combination with factors `alpha` and `beta` that is part of the generalized matrix multiplication.

Figure 5.7 shows the generalized histogram idiom. It is contained in a for loop and the basic memory access pattern is a read-modify-write to a bin array. This memory access can be conditional as long as the condition is well behaved, which is guaranteed by the later `KernelFunction` idiom. The histogram uses input data that is read from input arrays using the loop iterator as a base index (that can be strided, offset etc.). Finally there are two well behaved kernel functions in a histogram, one to compute the access index and one to compute the updated value.

The sparse matrix vector multiplication defined in Figure 5.8 is different to the other idioms in that the control flow of the skeleton of the idiom does not consist of perfectly nested for loops. Instead, the iteration space of the inner loop is read from an array using the `ReadRange` idiom. The actual computation that SPMV performs is a dot product and thus it uses the same `DotProductLoop` idiom as `GEMM` but the memory access pattern is different, with indirect memory access in `indir_read`.

Figure 5.9 shows the basic stencil idiom. Stencils consist of a loop nest with a multi-dimensional memory access to store the updated cell value. This updated value is computed

with a kernel function using a number of values that are constraint by the `StencilRead` idiom, which specifies multidimensional array access with only constant offsets in all dimensions.

The scalar reduction idiom is specified in Figure 5.10. We can see that its structure is similar to the histogram idiom, but instead of a read-modify-write memory access it operates on an induction variable that is implemented with the `InductionVar` idiom.

5.3.3 Not Syntactic Pattern Matching

The idiom descriptions may at first appear to be shallow syntactic pattern matching. In fact, because it operates on the IR level, it can detect idioms that are written in superficially distinct style but are semantically equivalent. For example, there are two syntactically distinct programs in Figure 5.4, which in fact are both implementations of general matrix multiplication. The IDL in Figure 5.6 discovers they are both instances of GEMM and they can both be replaced with an API call to GEMM.

```
for (int mm = 0; mm < m; ++mm) {
  for (int nn = 0; nn < n; ++nn) {
    float c = 0.0f;
    for (int i = 0; i < k; ++i) {
      float a = A[mm + i * lda];
      float b = B[nn + i * ldb];
      c += a * b;
    }
    C[mm+nn*ldc] =
        C[mm+nn*ldc] * beta + alpha * c;
  }
}
```

```
for(int i = 0; i < 1000; i++)
    for(int j = 0; j < 1000; j++) {
        M3[i][j] = 0.0f;
        for(int k = 0; k < 1000; k++)
           M3[i][j]+=M1[i][k]*M2[k][j]; }
```

Figure 5.4: Two matching instances of GEMM

There are limitations to this semantic matching. In particular, the use of low level manual optimizations that circumvent the usual IR representation, *e.g.* SIMD compiler intrinsics, would distort the algorithms beyond recognition by our system. In practice, this is rarely encountered.

```
Constraint SESE
( {precursor} is branch instruction and
  {precursor} has control flow to {begin} and
  {end} is branch instruction and
  {end} has control flow to {successor} and
  {begin} control flow dominates {end} and
  {end} control flow post dominates {begin} and
  {precursor} strictly control flow dominates
      {begin} and
  {successor} strictly control flow post dominates
      {end} and
  all control flow from {begin} to {precursor}
      passes through {end} and
  all control flow from {successor} to {end}
      passes through {begin})
End
```

Figure 5.5: IDL specification of SESE region

```
Constraint GEMM
( inherits ForNest(N=3) and
  inherits MatrixStore
      with {iterator[0]} as {col}
       and {iterator[1]} as {row}
       and {begin} as {begin} at {output} and
  inherits MatrixRead
      with {iterator[0]} as {col}
       and {iterator[2]} as {row}
       and {begin} as {begin} at {input1} and
  inherits MatrixRead
      with {iterator[1]} as {col}
       and {iterator[2]} as {row}
       and {begin} as {begin} at {input2} and
  inherits DotProductLoop
      with {loop[2]}        as {loop}
       and {input1.value}   as {src1}
       and {input2.value}   as {src2}
       and {output.address} as {update_address})
End
```

Figure 5.6: IDL specification of GEMM

```
Constraint Histogram
( inherits For and
  inherits ConditionalReadModifyWrite
      with {indexkernel.output} as {address}
       and {kernel.output} as {value} and
  collect i
  ( inherits VectorRead
       with {read_value[i]} as {value}
        and {iterator} as {idx}
        and {begin} as {begin} at {read[i]}) and
  inherits Concat
      with {read_value}   as {in1}
       and {old_value}    as {in2}
       and {kernel.input} as {out} and
  inherits KernelFunction
      with {begin} as {outer}
       and {body.begin} as {inner} at {kernel} and
  inherits KernelFunction
      with {read_value} as {input}
       and {begin} as {outer}
       and {body.begin} as {inner} at {indexkernel})
End
```

Figure 5.7: IDL specification of generalized histogram

```
Constraint SPMV
( inherits For and
  inherits VectorStore
      with {iterator} as {idx}
       and {begin} as {begin} at {output} and
  inherits ReadRange
      with {iterator} as {idx}
       and {inner.iter_begin} as {range_begin}
       and {inner.iter_end}   as {range_end} and
  inherits For at {inner} and
  inherits VectorRead
      with {inner.iterator} as {idx}
       and {begin} as {begin} at {idx_read} and
  inherits VectorRead
      with {idx_read.value} as {idx}
       and {begin} as {begin} at {indir_read} and
  inherits VectorRead
      with {inner.iterator} as {idx}
       and {begin} as {begin} at {seq_read} and
  inherits DotProductLoop
      with {inner}            as {loop}
       and {indir_read.value} as {src1}
       and {seq_read.value}   as {src2}
       and {output.address}   as {update_address})
End
```

Figure 5.8: IDL specification of SPMV

```
Constraint Stencil
( inherits ForNest and
  inherits PermMultidStore
      with {iterator} as {input}
       and {begin} as {begin} at {write} and
  collect i
  ( inherits StencilRead
      with {write.input_index} as {input}
       and {kernel.input[i]} as {value}
       and {begin} as {begin} at {reads[i]}) and
  {kernel.output} is first argument of {write.store} and
  inherits KernelFunction
      with {begin}      as {outer}
       and {body.begin} as {inner} at {kernel})
End
```

Figure 5.9: IDL specification of simple stencil

```
Constraint Reduction
( inherits For and
  collect i
  ( inherits VectorRead
        with {iterator}         as {idx}
         and {read_value[i]}    as {value}
         and {begin} as {begin} at {read[i]}) and
  inherits InductionVar
       with {old_value}       as {old_ind}
        and {kernel.output}   as {new_ind} and
  {old_value} is not the same as {iterator} and
  inherits Concat
      with {read_value}   as {in1}
       and {old_value}    as {in2}
       and {kernel.input} as {out} and
  inherits KernelFunction
      with {begin}      as {outer}
       and {body.begin} as {inner} at {kernel})
End
```

Figure 5.10: IDL specification of scalar reductions

5.4 Introduction

Heterogeneous accelerators provide the potential for great performance, but achieving that potential is difficult and requires significant effort. In order to target heterogeneous systems, programms need to be partially rewritten, making use of a diverse range of broad and narrow interfaces that are quickly evolving. General purpose languages such as OpenCL [72] provide some portability across heterogeneous devices, but the achieved performance often disappoints [54], effectively requiring rewrites despite the functional portability. Optimised numerical libraries provide more reliable performance, but they are more specialised and often provided by hardware vendors without portability in mind [4, 70, 71, 5]. More narrow domain specific languages (DSLs) [31, 84] have been proposed, attempting to deliver both portability and performance [79]. However this class is quickly evolving and most of them are academic projects with unclear long term support.

Hardware becomes increasingly heterogeneous, (*e.g.* TPU [46]). This means library or DSL based programming is likely to become far more common. The problems with this trend that arise due to the aforementioned tradeoffs are clear: Firstly, application developers have to learn multiple specialized DSLs and vendor-specific libraries if they want good performance. Secondly, they will have to rewrite their existing applications to use them. Thirdly, this ties their code into an ecosystem with unclear future support that might deprecate quickly. This situation is a severe impediment to the wide-spread efficient exploitation of heterogeneous hardware.

Ideally, we would like a compiler that automatically maps existing code to heterogeneous hardware, with full performance and requiring no directions from the application programmer. While this is unrealistic in general, the system that is presented in this chapter achieves a good approximation of such a general purpose system by utilising the knowhow that is already available, encapsulated in the existing interfaces as an intermediary. Instead of implementing code generation for several heterogeneous accelerators, it maps user code to heterogeneous hardware using the existing libraries and domain specific languages, effectively outsorcing the code genration to hardware specialists.

The approach is based on detecting specific *computational idioms* in application code that correspond to the functionality of existing APIs for heterogeneous acceleration. The covered idioms are largely a reflection of the existing libraries and DSLs, with a focus on sparse and dense linear algebra, stencils and generalized reductions and histograms. The idioms are formulated using CAnDL from chapter 4, which allows the concise formulation and fast detection in optimised LLVM IR code.

Once detected, the idioms are mechanically translated into the appropriate DSL or replaced with a library call. This optimized code is then linked into the original program. As backends, the libraries cuSparse, clSparse, cuBLAS, clBLAS for sparse and dense linear algebra and target the DSL Halide [79] for stencil computations are used in the evaluation. In addition

to this, the Lift language [89] is used - a data parallel language that supports generalized reductions as well as stencils and linear algebra. This allows the freedom to target many APIs for the same idiom and pick the implementation that best suits the target platform.

5.5 Overview

Our approach is automatic and has been implemented inside the LLVM compiler infrastructure. It takes arbitrary sequential C/C++ programs as input. Using the clang compiler, the input source code is compiled into a Single Static Assignment (SSA) intermediate representation. We then search this representation for particular idioms which are replaced with calls to specific APIs. Finally, the code generated by the LLVM compiler and the output of the idiom specific code generators/libraries are linked together into a binary, producing an optimized program. LLVM was chosen as it is the best supported SSA-based compiler; the methodology could easily be transferred to other infrastructures such as gcc.

5.5.1 Compiler Flow

The structure of our approach is described in more detail in Figure 5.11. Our compiler takes two programs as inputs: the first is the user's program source code, the second describes the idioms we wish to detect using our idiom description language (section 3). The same idioms, of course, can be detected across many user programs, so the IDL program does not have to change from one run to the next.

The program source code is compiled to optimized LLVM intermediate representation code and the idiom description is translated into constraints and represented internally as a C++ object. The C++ representation of the constraints and the user program LLVM IR code are then passed as inputs to a backtracking solver [32], which detects all cases where the idioms can be found in the LLVR IR.

The recognized idioms and the LLVM IR code are then passed on to the transformation phase of our system. The sections of code corresponding to computational idioms are extracted and reformulated for the appropriate heterogeneous APIs. For libraries this means replacing the code covered by the idiom with a library call. For domain specific language interfaces, things are a little more involved. As before, we first extract the code associated with the idiom and replace it with a function call. This extracted code is now translated into the appropriate DSL and then passed on to the external DSL compiler which optimizes and generates code. The generated code is then linked with the object code from the main program.

Determining the best heterogeneous APIs to use for a given platform and the best idioms to exploit will become a major issue as the number of idioms and APIs grows. Currently, in this paper, we just try all applicable libraries and DSLs and pick the best executing code.

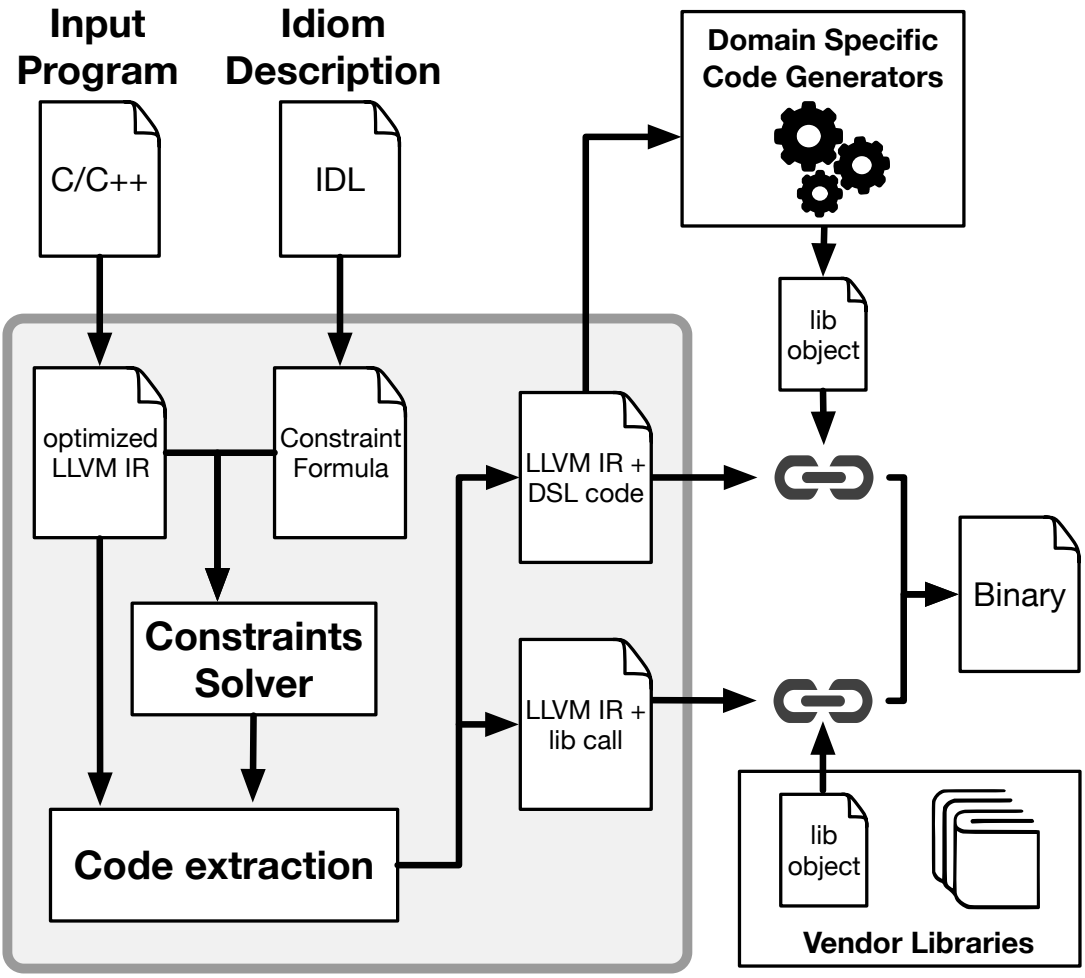


Figure 5.11: Workflow of our system

Determining the best option is future work.

5.5.2 IDL Example

At the core of our approach is IDL, which is described in section 5.6. A fundamental part of its design is the ability to detect complex idioms. Here we first focus on a simple example to show how IDL works. Consider the standard factorizing optimization that applies the algebraic rule of distributivity

$$(x*y)+(x*z)=x*(y+z)$$

to simplify calculations by reducing the number of multiplications in an expression. The established way of implementing such an optimization is to hard code a detection compiler pass. In LLVM, this is 47 lines of code inside the `instcombine` pass.

With IDL, we can formulate this in only a few lines of an easily understandable program (Figure 5.12). For this example, the underlying constraint problem is immediately visible: There are four variables `sum`, `left_addend`, `right_addend`, `factor` and nine individual constraints

```
Constraint FactorizationOpportunity
( {sum} is add instruction and
  {left_addend} is first argument of {sum} and
  {left_addend} is mul instruction and
  {right_addend} is second augment of {sum} and
  {right_addend} is mul instruction and
  ( {factor} is first argument of {left_addend} or
    {factor} is second argument of {left_addend}) and
  ( {factor} is first argument of {right_addend} or
    {factor} is second argument of {right_addend}))
End
```

Figure 5.12: IDL formulation of (x*y)+(x*z) pattern

that are combined with boolean operators.

From the formulation in Figure 5.12, the IDL compiler builds an internal representation of the underlying constraint problem that is passed to a constraint solver. For a given section of user code, this solver returns the set of factorization opportunities, each containing four entries `sum`, `left_addend`, `right_addend`, `factor` that refer to values inside the user code. Figure 5.13 shows a simple example. The incoming C code is translated to optimized LLVM IR. The solver then finds a single solution to the constraint problem.

Original C code:

```
int example(int a, int b, int c) {
    int d = a;
    return (a*b) + (c*d);
}
```

Resulting LLVM IR:

```
define i32 @example(i32 %a, i32 %b, i32 %c) {
  %1 = mul i32 %a, %b
  %2 = mul i32 %c, %a
  %3 = add i32 %1, %2
  ret i32 %3
}
```

Detected factorization opportunities:

```
{ "sum" :          %3,
  "left_addend" :  %1,
  "right_addend" : %2,
  "factor" :       %a }
```

Figure 5.13: Demonstration of simple idiom detection

In this case, the variable `sum` is matched to the value `%3`, an add instruction, while the

variables `left_addend` and `right_addend` match the left and right operands `%1` and `%2` of this instruction.

Lines 7 and 8 of Figure 5.12 say that `factor` can either be the first argument OR the second argument of `left_addend`. As the `left_addend` is `%1`, then `factor` can be either `%a` or `%b`. Similarly lines 9 and 10 of Figure 5.12 say that `factor` can be be the first argument OR the second argument of the `right_addend`. As the `right_addend` is `%1`, then `factor` can be either `%c` or `%a`. The two disjunctions in Lines 7-10 are connected by AND, so they must both hold.

$$\begin{aligned}&((factor = a) \vee (factor = b))\\ \wedge&((factor = c) \vee (factor = a))\\ &\Longrightarrow factor = a\end{aligned}$$

The only value of `factor` that satisfies this condition is `factor = %a`. Therefore, the solution at the bottom of Figure 5.13 is the only factorization opportunity in the code.

5.5.3 Sparse Linear Algebra in IDL

Although the previous example illustrated how constraints can be applied to program analysis, we want to detect much more complex idioms and map them to existing APIs.

The C code in Figure 5.14 shows the performance bottleneck of the NAS *Conjugate Gradient (GC)* benchmark, as well as the corresponding LLVM IR code. It implements a standard operation from sparse linear algebra, namely a multiplication of a sparse matrix in Compressed Sparse Row (CSR) format with a dense vector.

This code contains several features that make it unsuitable for most established compiler optimizations: The iteration domain of the nested loop is memory dependent (line 3) and there is indirect memory access (line 4). This makes the iteration domain of the loop nest non-polyhedral and the access structure to memory non-affine. Under these conditions, simple data dependence models, but also sophisticated analysis based on the polyhedral model, would fail.

We can express this idiom in IDL (section 5.1, Figure 5.8). The IR code, together with the IDL program, is fed into a constraint solver, which outputs a constraint solution as shown in Figure 5.15. We can see that different parts of the IR have been assigned to IDL variables.

Figure 5.16 shows how this solution is used to generate a call to a cuSPARSE procedure. The solution variables are inserted into the `cusparseDcsrmv` code template as function arguments. The original code is then cut out and replaced with this function call. The cuSPARSE library is then linked with the object code produced by the LLVM compiler, resulting in a speedup of 17× on a GPU as described in section 6.6.

Central to our approach is the ability to detect idioms. In the next section we introduce a powerful description language that is capable of expressing a wide class of idioms that are

```
for (j = 0; j < m; j++) {
  d = 0.0;
  for (k = rowstr[j]; k < rowstr[j+1]; k++)
    d = d + a[k]*z[colidx[k]];
  r[j] = d; }
```

```
; <label>:2:
  %j = phi i64 [ %j_next, %12 ], [ 0, %1 ]
  %j_cond = icmp slt i64 %j, %m
  br i1 %j_cond, label %3, label %13
; <label>:3:
%4 = getelementptr i32, i32* %rowstr, i64 %j
  %5 = load i32, i32* %4
  %j_next = add nuw nsw i64 %j, 1
  %6 = getelementptr i32, i32* %rowstr, i64 %j_next
  %7 = load i32, i32* %6
  %k_begin = sext i32 %5 to i64
  %k_end = sext i32 %7 to i64
  br label %8
; <label>:8:
  %k = phi i64 [ %k_next, %9 ], [ %k_begin, %dnext ]
  %d = phi double [ 0.0, %3 ], [ %d_next, %9 ]
  %k_cond = icmp slt i64 %iv, %k_end
  br i1 %k_cond, label %9, label %12
; <label>:9:
  %a_addr = getelementptr double, double* %a, i64 %k
  %a_load = load double, double* %a_addr
  %cix_addr = getelementptr i32, i32* %colidx, i64 %k
  %cix_load = load i32, i32* %cix_addr
  %10 = sext i32 %cix_load to i64
  %z_addr = getelementptr double, double* %z, i64 %10
  %z_load = load double, double* %z_addr
  %11 = fmul double %a_load, %z_load
  %d_next = fadd double %d, %11
  %k_next = add nsw i64 %k, 1
  br label %8
; <label>:12:
  %r_addr = getelementptr double, double* %r, i64 %j
  store double %d, double* %r_addr
  br label %2
```

Figure 5.14: Sparse linear algebra in C and LLVM IR

↓

Idiom detection with IDL program in Figure 5.8

↓

Variable Name	Assigned IR Value
iterator	%j
inner.iter_begin	%k_begin
inner.iter_end	%k_end
inner.iterator	%k
idx_read.value	%cix_load
indir_read.value	%a_load
seq_read.value	%z
output.address	%r_addr
iter_begin	0
iter_end	%**m**
idx_read.base_pointer	%**colidx**
seq_read.base_pointer	%**a**
indir_read.base_pointer	%**z**
...	...

Figure 5.15: Constraint solution for sparse mv

↓

Code generation: insert arguments, replace code

↓

```
cusparseDcsrmv(context,
    CUSPARSE_OPERATION_NON_TRANSPOSE, m, n,
    rowstr[m+1]-rowstr[0], &gpu_1, descr, gpu_a,
    gpu_rowstr, gpu_colidx, gpu_z, &gpu_0, gpu_r);
```

suitable for acceleration by heterogeneous hardware.

5.6 Idiom Description Language

Any detection method needs to be robust and work on real code. It should work in the presence of complex language features, such as the standard library containers, operator overloading and class hierarchies in C++, as well as the myriad different ways users can write the same, common algorithms.

This rules out a syntactic approach. To allow robust detection of complex idioms, we devised IDL, a domain specific constraint language that operates on the SSA based LLVM IR. In IDL, idioms are specified in a modular fashion, exploiting standard compiler primitives such as types and data and control flow analysis.

IDL was developed with the aim of enabling analysis routines that are too complex to directly implement by hand. However, it is still targeted at compiler experts. Writing and debugging IDL code is challenging, but the modularity mechanisms make it very suitable for unit testing. The full syntax specification of IDL in BNF notation is shown in Figure 5.17.

5.6.0.0.1 Terminals The symbols ⟨**s**⟩ and ⟨**n**⟩ in the grammar correspond to arbitrary strings and positive integer literals respectively, while the ⟨`specification`⟩ top level construct of the language binds an idiom definition to a name. The significant part of the language specification is everything covered by ⟨`constraint`⟩.

5.6.0.0.2 Atomic Constraints All idiom definitions are eventually built up by combining atomic constraints. These correspond to basic boolean predicates that may hold for one or more values in the IR. The atomic constraints describe standard properties within the IR. Control flow in our model is evaluated on the granularity of instructions. This is to reduce the size of the language, there is no notion of basic blocks. For phi nodes, the incoming basic blocks are identified with their terminating branch instruction.

5.6.0.0.3 Higher Level Constructs Atomic constraints can be combined with many higher level language constructs. The semantics of ⟨`conjunction`⟩ and ⟨`disjunction`⟩ correspond to AND, OR. The ⟨`inheritance`⟩ inserts another idiom description into the current one. Idiom definitions can be parameterized in a way that is inspired by C++ templates with integers, allowing more concise descriptions. The ⟨`if`⟩ constraint has the standard meaning.

The ⟨`forall`⟩ and ⟨`forsome`⟩ constructs provide range based versions of conjunction and disjunction. The contained constraint formula is duplicated for each index in the provided range and the contained variable names are modified according to the index (i.e. if the index

occurs in a variable name, it is substituted with the current iteration value). The duplicated formulas are then combined with conjunctions or disjunctions respectively.

To allow modularity, complementing the inheritance feature, there are two mechanisms to change the variable names in the contained constraint specification. With ⟨rename⟩, the translation of variable names is done with a simple dictionary, where every variable that is not explicitly mentioned in the dictionary remains unchanged. The ⟨rebase⟩ has the same behaviour for variables in the dictionary, but for every other variable, a prefix is added to the variable name.

The ⟨collect⟩ construct is more powerful. It is used to capture all possible solutions of a given constraint for expressions that require the logical ∀ quantifier. For example, it can be used to collect all affine array accesses in a loop.

5.6.1 Compilation Process and Implementation

Idiom definitions are compiled to C++ functions that perform idiom recognition on LLVM IR. In a first step, the compiler eliminates ⟨inheritance⟩, ⟨forall⟩, ⟨forsome⟩, ⟨if⟩, ⟨rename⟩ and ⟨rebase⟩. They are replaced with simpler ⟨conjunction⟩ and ⟨disjunction⟩ constructs. This also involves removing all parameterizations from the formula and flattening all variable names. Next, variables are collected and ordered to assist constraint solving. The ordering impacts performance, as it determines how well the search space is pruned. For each variable, all the constraints associated with the variable are assembled.

The compiler then emits C++ code which is passed to a generic solver based on [32] to search for idiom instances. This solver is based on standard backtracking. As shown in the results section, this increases compilation time, but the overhead is modest.

5.7 Targeted Heterogeneous APIs

After idiom detection, we must transform the user program to exploit the relevant API. Two types of heterogeneous APIs are currently targeted: libraries and domain specific languages with their optimizing compilers.

5.7.1 Domain Specific Libraries

Libraries provide narrow interfaces but are often highly optimized. For example, the cuBLAS library is only suitable for a limited set of dense linear algebra operations and only works on Nvidia GPUs, but its implementation provides outstanding performance. For sparse linear algebra we use the vendor provided cuSPARSE, clSPARSE, and MKL libraries. For dense BLAS routines cuBLAS, clBLAS, CLBlast, and MKL are used.

5.7.2 Domain Specific Code Generators

Domain Specific Languages provide wider interfaces than libraries and allow problems to be expressed as composition of dedicated language constructs. An optimizing compiler then specializes the program for the target hardware. We currently support Halide and Lift as domain specific code generators.

5.7.2.0.1 Halide [79] is a language and optimizing compiler targeted at image processing applications. Optimized code is generated for CPUs as well as GPUs. Halide separates the functional description of the problem from the description of the implementation which is called a *schedule*. This allows retargeting of Halide programs to different platforms. We translate some of the stencil idioms and linear algebra idioms into Halide. Stencils involving control flow in their computations are not easily expressible in Halide.

5.7.2.0.2 Lift [88, 89, 41] is an optimizing code generator based on rewrite rules. The Lift language consists of functional parallel patterns such as *map* and *reduce* which express a range of parallel applications. For this work we translate stencil idioms, complex reductions and linear algebra idioms to Lift.

5.8 Translating Computational Idioms

This section describes how the detected idioms are mapped to the previously described library APIs domain specific languages. The two types of APIs (library interfaces and domain specific languages) are treated individually.

5.8.1 Library

For library call interfaces, the original code is removed and an appropriate function call is inserted. The solution that is generated by the solver using the IDL program contains both the IR instructions to remove as well as the arguments that are to be used for the function call.

For example, in the case of the `GEMM` program that was shown in Figure 5.6, the original code is removed by deleting the IR instruction at `output.store_instr` explicitly, which captures the store instruction of the `MatrixStore` subprogram. The remaining cleanup is left to the standard dead code elimination pass. The arguments that specify the matrix dimensions are taken from `ForNest` in combination with the stride and offset determined by `MatrixRead` and `MatrixWrite`.

The mapping of solution variables to the arguments of the generated function call needs to be implemented individually for each backend, as we have no way to describe it in IDL itself. Once the code is replaced, LLVM continues with code generation as usual.

5.8.2 DSL

For domain specific languages, the situation is a bit more involved. Reduction, histogram and stencil idioms are higher order functions that contain a kernel function or reduction operator that has to be represented for the DSL.

For each combination of idiom and DSL there is a parameterized skeleton program. This skeleton is then specialized for the appropriate data types and numeric parameters as well as the kernel function or reduction operator.

Numerical parameters are picked from the constraint solution in the same way that was described previously for library call interfaces. Also from the constraint solution, we have the loop body that contains the kernel function or reduction operator, as well as the input values and the result value used. We use this information to cut out the kernel function that is then used to generate code appropriate for the DSL backends:

5.8.2.0.1 Lift expects stencil kernels or reduction operators to be sequential C code with a specific function interface that is used internally by Lift when generating OpenCL code. We therefore implemented a rudimentary LLVM IR to C backend for generating this function.

5.8.2.0.2 Halide is a language embedded in C++, it requires a syntax tree of the kernel functions built using a class hierarchy.

5.8.2.0.3 After code for the DSLs is generated, it is passed to the DSL code generator. Figure 5.18 shows an example of the Lift code generated for GEMM (`gemm_in_lift`). It performs a dot product (expressed in line 8 using the Lift skeletons `zip`, `map`, and `reduce`) for each row of matrix A (`a_row`) and column of matrix B (`b_col`). This code is compiled by Lift into optimized OpenCL code.

Finally, we again replace the idiom code in the user's code with a call to the code generated by the DSL and continue once again with LLVM code generation.

5.8.3 Aliasing

Since idiom detection works statically, we are unable to fully rule out aliasing of pointers, which can make transformations unsound. For dense linear algebra this is easily solved with some basic run time checks for non-overlapping memory. However, for sparse linear algebra this is not as straightforward and in corner cases our approach is unsound. In practice this did not cause problems on any of the benchmark programs, however this means that optimizations based on these techniques will have to provide appropriate feedback to the programmer.

5.9 Experimental Setup

Benchmarks We applied our approach to all of the sequential C/C++ versions of the NAS Parallel Benchmarks. We use the SNU NPB implementation by the Seoul National University, containing the original 8 NAS benchmarks plus two of the newer unstructured components UA and DC. We also evaluated our approach on all Parboil benchmarks, giving 21 programs in total.

Platform and Evaluation We use an AMD A10-7850K APU with a multi-core CPU and an integrated Radeon R7 GPU on the same die using driver version 1912.5, as well as an Nvidia GTX Titan X as an external GPU using driver version 375.66. We report the median runtime of 10 executions for each program.

Alternative detection approaches There are no easily available compilers to compare against that perform idiom detection. Instead, we consider two parallelizing compilers and examine whether they detect idioms as part of their parallelization approach. As their goal is parallelization and not idiom detection, this should be borne in mind in the results section.

Polly [26] is an LLVM based polyhedral compiler capable of finding parallel loops and reductions in static control flow (SCoP) parts of programs. This allows comparison against another approach that uses the same compiler infrastructure. We gathered the SCoPs that Polly detected with the options `-O3 -mllvm -polly -mllvm -polly-export` and manually inspected the reported SCoPs for stencil like parallel loops and reduction operations. When Polly captured such a loop as a SCoP, we counted it as an idiom detection, although Polly itself has no concept of idioms. This gives an optimistic estimate as to what idiom coverage a polyhedral based approach can achieve.

The Intel C++ Compiler (ICC) is a mature industry strength compiler that provides a detection mechanism for parallelizing reduction idioms based on data dependency analysis. We use the `-parallel -qopt-report` command line options and checked in the optimization report files whether the corresponding loop is considered parallelizable.

5.10 Results

We first evaluate how often our approach is able to detect idioms and its compile time cost. We then investigate the runtime coverage of the idioms to see where exploitation might be beneficial. Where runtime coverage is substantial, we report speedups over sequential C code and compare the performance of each of the targets APIs. We also compare against the handwritten OpenMP and OpenCL implementations that are included with the benchmark suites as reference implementations.

	Scalar Reduction	Histogram Reduction	Stencil	Matrix Op.	Sparse Matrix Op.
Polly	3	—	5	—	—
ICC	28	—	—	—	—
IDL	45	5	6	1	3

Table 5.1: Idioms detected by IDL, ICC, Polly

5.10.1 Idiom Detection

Table 5.1 shows the number of idioms found by our approach, Polly, and ICC. Polly finds 3 scalar reductions and 6 stencils while ICC which just considers scalar reduction finds 28. Polly is unable to perform idiom specific optimizations on GEMM. Other approaches do not detect any histograms or sparse matrix operations, because such code involves indirect and thus non-affine memory accesses. This fundamentally contradicts assumptions that these tools rely on and is not merely an implementation artifact. Our IDL approach detects 60 idioms overall with the compile time cost shown in figure Table 5.2. On average, the compilation time is increased by 82%, which can be reduced further by optimizing the solver.

Figure 5.19 shows the different idioms detected across the benchmarks. We detect both scalar and histogram reductions as well as stencils, dense matrix operations and sparse matrix-vector multiplication. Polly and ICC are only capable of detecting simple scalar reductions, but we are able to detect histogram reductions, *e.g.* in the *histo* benchmark as well. For stencils, Polly detects two in *lbm* and *stencil* while our approach detects all the stencils in *lbm*, *stencil* and *MG*. Unlike any existing approach, we detect sparse matrix-vector operations in *C*G and *s*pmv as well as dense matrix operations in *sgemm*. It is worth repeating, however, that both Polly and ICC are parallelizing compilers, not idiom recognition tools.

5.10.2 Runtime Coverage

To determine if the detected idioms are actually important, Figure 5.20 shows the percentage of time spent in the detected computational idiom. This data shows that either the detected idioms have a low runtime contribution or they dominate almost the entire execution. *EP* is the only exception where about 50% of the runtime is spent inside a detected histogram reduction. We focus on the 10 programs which spend a significant amount of time in the detected idioms, as only these can reasonably expect a performance gain using our approach.

5.10.3 Performance Results

Speedup vs. Sequential Figure 5.21 shows the end-to-end speedup obtained by accelerating idioms with heterogeneous APIs on a CPU, an integrated GPU, and an external GPU. All results

	BT	CG	DC	EP	FT	IS	LU	MG	SP	UA	bfs	cutcp
without IDL	1.9	0.5	1.0	0.3	0.6	0.3	1.9	0.8	1.6	2.7	0.4	0.4
with IDL	4.0	0.8	1.6	0.6	1.2	0.5	3.9	4.5	3.2	7.3	0.5	0.6
overhead in %	116	77	57	77	93	62	103	484	97	169	30	65

	histo	lbm	mri-g	mri-q	sad	sgemm	spmv	stencil	tpacf
without IDL	0.2	0.3	0.2	0.2	0.4	0.6	0.3	0.2	0.2
with IDL	0.2	0.6	0.4	0.3	0.6	0.7	0.7	0.2	0.4
overhead in %	35	87	100	52	58	24	115	36	54

Table 5.2: Compile time cost in seconds

include data transfer overhead to and from the GPUs. Here the best performing API is shown; Figure 5.23 provides detailed results for all APIs.

For five benchmarks we obtain moderate speedups from 1.26× for *histo* up to 4.5× for *IS*. All of these benchmarks besides *MG* have a scalar or histogram reduction as their performance bottleneck and are, therefore, not computationally expensive. Interestingly, we can see that for different benchmarks, different hardware is beneficial: for *tpcaf* the CPU is the best platform, beating the GPU for which the data transfer time dominates; for *MG* and *histo* the integrated GPU strikes the right balance between computational power while avoiding the movement of data to the external GPU; and, finally, for *EP* and *IS* the data transfer to the GPU pays off exploiting the high GPU internal memory bandwidth. These results emphasize the significance of heterogeneous code generation flexibility.

For five of the benchmarks we achieve significantly higher performance gains, from 17× for *CG* and up to over 275× for *sgemm*. These benchmarks are computationally expensive and the external GPU is always the fastest architecture by a considerable margin.

The red highlighting in the plot indicates an important runtime optimization: redundant data transfers for the iterative *CG*, *lbm*, *spmv* and *stencil* benchmarks. All of these benchmarks execute computations inside a for loop and do not require access to the data on the CPU between iterations. We manually applied a straightforward lazy copying technique by flagging memory objects to avoid redundant transfers, similar to [44]. As can be seen this runtime optimization is crucial for achieving high performance for these benchmarks.

API performance comparison Figure 5.23 shows a breakdown of the performance of each API on each program and platform. Not all APIs target all platforms, *e.g.* cuSPARSE only targets NVIDIA GPUs and in the case of Halide, the current version that we have access to failed to generate valid GPU code for any of the benchmarks we tried. The best performing API is highlighted in bold in the table entries. The *spmv* benchmark uses an unusual sparse

matrix format, so that we implemented a custom library libSPMV for this benchmark.

On the multicore CPUs, the Intel MKL library gives the best linear algebra performance, outperforming the other libraries and Lift. Halide achieves good performance for the NPB *IS* and Parboil *stencil* benchmarks on the CPU, outperforming Lift due to its more advanced vectorization capabilities. In the programs where scalar reductions dominate, Lift performs well. On the iGPU, clBLAS provides a better matrix-multiplication implementation than either CLBlast or Lift. On the external GPUs, the libraries provide better linear algebra implementations, while Lift performs well on stencils and reductions.

Speedup vs. Handwritten Parallel Implementations Figure 5.22 shows the performance of our approach compared to hand-written reference OpenMP and OpenCL implementations. For some of the benchmarks, the parallel versions are significantly modified using different algorithms beyond the domain of automation. We can see that for benchmarks where the handwritten implementation does not make algorithmic changes (*CG*, *histo*, *lbm*, *sgemm*, *spmv*, *stencil*), we achieve comparable – or better – performance. For four benchmarks (*EP*, *IS*, *MG*, and *tpacf*) it is beneficial to parallelize the entire application – which is beyond the scope of this paper. Future work will examine outer loop parallelism as an idiom to exploit.

For the *sgemm* and *stencil* benchmarks we improved the baseline implementation provided by the benchmarks as these had extremely poor performance. A simple interchange of two loops improved performance by almost 20 times.

Summary 60 idioms were detected across the benchmark suites and significant performance improvements were achieved by targeting different heterogeneous APIs for those benchmarks where idioms dominate execution time.

5.11 Related and Future Work

Domain specific Languages Domain specific languages have received much attention in recent years, ranging from SPIRAL [73], a DSL for Fast Fourier Transforms, over Lift [88, 89, 41] to UFL [2], a DSL for partial differential equations. Stencils in particular have received much attention [65, 41], the best known of which is Halide [79]. DSLs to exploit complex reductions are less studied. Chandan Reddy and Cohen [21] introduce a type of DSL via annotations that allow expression of complex reductions. This was based on the Platform-Neutral Compute Intermediate Language by Baghdadi et al. [10]. The Matrix multiplication idimos is well supported by specific libraries [4, 42, 70].

Generation of Performance Portable Code for Heterogeneous Hardware Recent research has highlighted the challenges of generating code that performs well on different heterogeneous

hardware architectures. PetaBricks [75] is one of the first languages to address this performance portability challenge by encoding algorithmic choices which are then empirically evaluated and automatically taken by the compiler. Similarly [68] explores automatic selection of code variants using machine learning. In a similar spirit, Lift [88] uses rewrite rules to explore optimization choices automatically.

Functional Code Generation Approaches There exist multiple functional approaches for generating code for heterogeneous hardware. Chakravarty et al. [20], McDonell et al. [59] propose Accelerate, a domain specific language embedded in Haskell that generates efficient GPU code. Recently, Collins et al. [23] introduced NOVA, a new functional language targeted at code generation for GPUs, and Copperhead [18], a data parallel language embedded in Python. Delite [17, 19] is a system that enables the creation of domain-specific languages using functional parallel patterns and targets multi-core CPUs or GPUs. In contrast to these approaches, we require no rewriting of legacy programs.

Idiom Detection Idiom based optimization based on AST manipulation, as described by Pinter and Pinter [76], has fallen out of fashion. More systematic approaches based on static single assignment representation [52] and polyhedral representations [15] however were more recently investigated. They were largely based on syntactic pattern matching and not robust in the presence of complex control and dataflow. More recently, work by Andión [6] describes a compiler based parallelization approach for heterogeneous computing, based on an diomatic intermediate representation called KIR. It is not clear how such an approach would work on general C/C++ programs.

Stencils Stencil detection has been driven by the introduction of DSLs such as Halide. Helium [60] tackles the challenging task of detecting stencils in binary code. It relies on dynamic analysis and cannot easily be extended to other idioms. Another closely related paper is [48], which detects stencils in FORTRAN by the verified lifting of code segments to a representation that can be mapped to Halide DSL. It uses syntax guided synthesis to verify translation with added constrains to ensure that it can be mapped to Halide. It however requires nested loops without conditionals in well behaved FORTRAN and in some cases requires user annotations.

Reductions Discovering and exploiting scalar reductions in programs has been studied for many years based on dependence analysis and idiom detection [30, 78, 92]. Alongside this data dependence based approach, there has also been a large body of work exploring mapping of reductions in a polyhedral setting [47, 83] The treatment of more general reduction operations has received less attention. Work has focused on exploitation rather than discovery [38, 39, 40],

examining trade-offs in implementation [105] or exploitation of novel hardware [82, 102]. Recent work [32] shows that more complex reductions can be detected, but this is tied to an ad hoc non-portable code generation phase.

Polyhedral Approaches Polyhedral compilers [12, 97] perform advanced loop optimizations and have been used for the generation of fast GPU kernels. More recently, extensions to the polyhedral framework have been proposed, allowing it to capture reduction computations [22, 37, 90]. Such efforts are described in [26], but they are fragile in the presence of non static control flow.

Future Work Although idioms can be described concisely with IDL, we currently have to implement a separate translation scheme for each API. While much of the translation code is common, it would be preferable to have an API description language similar to IDL that allows automatic generation of API translators. This would allow rapid evaluation of different APIs for the same idiom.

With a growing number of APIs and idioms, profitability heuristics will become necessary to determine the best API to use for each program and platform. Machine learning approaches are an obvious starting point as they easily adapt to changing targets.

This paper restricts its attention to five common idioms. Other idioms such as graph processing can also be described. Given that IDL works on the compiler IR, loop and function parallelism can also be described as idioms. In those cases where user codes do not quite match the platform API and associated idioms, we can apply program transformations to refactor or rejuvenate code to fit.

To be a robust approach to heterogeneous programming, we need to ensure correctness. Syntax guided synthesis is a promising means of verifying the idiom translation.

It would be interesting to see to whether our approach could be used for binary optimization or applied to heavily optimized and complex code bases.

5.12 Conclusion

This paper develops a compiler based approach that automatically detects a wide class of idioms supported by libraries or domain specific languages for heterogeneous processors. This approach is based on a constraint based description language that identifies program subsets that adhere to idiom specifications. Once detected, the idioms are mechanically translated into API calls to external libraries or code generated by DSL compilers.

The approach is robust and was evaluated on C/C++ versions of two well known benchmark suites: NAS and Parboil. We detected more stencils, sparse matrix operations and generalized reductions and histograms than existing approaches and generated fast code.

Future work will extend the constraint formulation to consider other common idioms. As the number of idioms detected and of implementations available grows, a smart profitability analysis will be needed and is the subject of future work.

specification ::= **Constraint** ⟨**s**⟩ ⟨constraint⟩ **End**

constraint ::= ⟨atomic⟩ | ⟨grouping⟩ | ⟨collect⟩ | ⟨rename⟩ | ⟨rebase⟩ | **'('** ⟨constraint⟩ **')'**

grouping ::= ⟨conjunction⟩ | ⟨disjunction⟩ | ⟨inheritance⟩ | ⟨forall⟩ | ⟨forsome⟩ | ⟨forone⟩ | ⟨if⟩

conjunction ::= **'('** ⟨constraint⟩ **and** ⟨constraint⟩ {**and** ⟨constraint⟩} **')'**

disjunction ::= **'('** ⟨constraint⟩ **or** ⟨constraint⟩ {**or** ⟨constraint⟩} **')'**

inheritance ::= **inherits** ⟨**s**⟩ [**'('** ⟨**s**⟩ **'='** ⟨calculation⟩ {**','** ⟨**s**⟩ **'='** ⟨calculation⟩} **')'**]

forall ::= ⟨constraint⟩ **for all** ⟨**s**⟩ **'='** ⟨calculation⟩ **'..'** ⟨calculation⟩

forsome ::= ⟨constraint⟩ **for some** ⟨**s**⟩ **'='** ⟨calculation⟩ **'..'** ⟨calculation⟩

forone ::= ⟨constraint⟩ **for** ⟨**s**⟩ **'='** ⟨calculation⟩

if ::= **if** ⟨calculation⟩ **'='** ⟨calculation⟩ **then** ⟨constraint⟩ **else** ⟨constraint⟩ **endif**

rename ::= ⟨grouping⟩ **with** ⟨var⟩ **as** ⟨var⟩ **and** ⟨var⟩ **as** ⟨var⟩

rebase ::= ⟨grouping⟩ [**with** ⟨var⟩ **as** ⟨var⟩ **and** ⟨var⟩ **as** ⟨var⟩] **at** ⟨var⟩

collect ::= **collect** ⟨**s**⟩ ⟨**n**⟩ ⟨constraint⟩

atomic ::= ⟨var⟩ **is** (**integer** | **float** | **pointer**) [**constant zero**]
| ⟨var⟩ **is** (**unused** | **a constant** | **a compile time value** | **an argument** | **an instruction**)
| ⟨var⟩ **is** (**store** | **load** | **return** | **branch** | **add** | **sub** | **mul** | **fadd** | **fsub** | **fmul** | **fdiv**
| **select** | **gep** | **icmp**) **instruction**
| ⟨var⟩ **is** [**not**] **the same as** ⟨var⟩
| ⟨var⟩ **has** (**data flow** | **control flow** | **control dominance** | **dependence edge**) **to** ⟨var⟩
| ⟨var⟩ **is** (**first** | **second** | **third** | **fourth**) **argument of** ⟨var⟩
| ⟨var⟩ **reaches phi node** ⟨var⟩ **from** ⟨var⟩
| ⟨var⟩ [**does not**] [**strictly**] [(**data flow** | **control flow**)] **dominates** ⟨var⟩
| **all** [(**data** | **control**)] **flow from** ⟨var⟩ **to** ⟨var⟩ **passes through** ⟨var⟩
| **all flow from** ⟨varlist⟩ **to** ⟨varlist⟩ **is killed by** ⟨varlist⟩

varsingle ::= ⟨**s**⟩ | ⟨varsingle⟩ **'.'** ⟨**s**⟩ | ⟨varsingle⟩ **'['** ⟨calculation⟩ **']'**

varmulti ::= ⟨varsingle⟩ | ⟨varmulti⟩ **'['** ⟨calculation⟩ **'..'** ⟨calculation⟩ **']'**

varlist ::= **'{'** ⟨varmulti⟩ **','** {⟨varmulti⟩ **','**} ⟨varmulti⟩ **'}'**

var ::= **'{'** ⟨varsingle⟩ **'}'**

calculation ::= ⟨**s**⟩ | ⟨**n**⟩ | ⟨calculation⟩ (**'+'** | **'-'**) (⟨**s**⟩ | ⟨**n**⟩)

Figure 5.17: BNF notation of IDL syntax

```
1  float mult(float x, float y) { return x*y; }
2  float add(float x, float y) { return x+y; }
3
4  gemm_in_lift(A, B, C, alpha, beta) {
5   map(fun(a_row, c_row) {
6    map(fun(b_col, c) {
7     map(fun(ab){ add(mult(alpha, ab), mult(beta, c))},
8      reduce(add, 0.0f, map(mult, zip(a_row, b_col))))
9    }, zip(transpose(B), c_row))
10  }, zip(A, C))
11 }
```

Figure 5.18: Example of matrix multiplication in Lift.

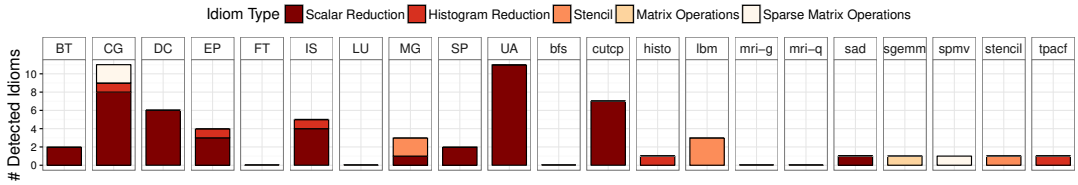


Figure 5.19: The different computational idioms found in all benchmarks.

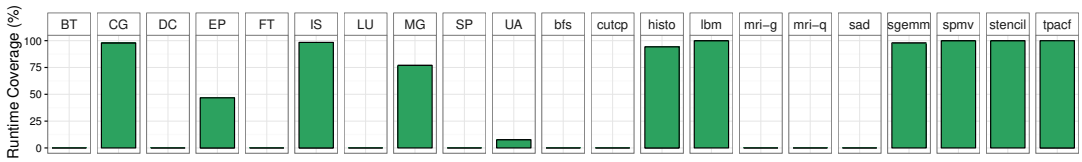


Figure 5.20: Runtime coverage of the detected idioms in all benchmarks.

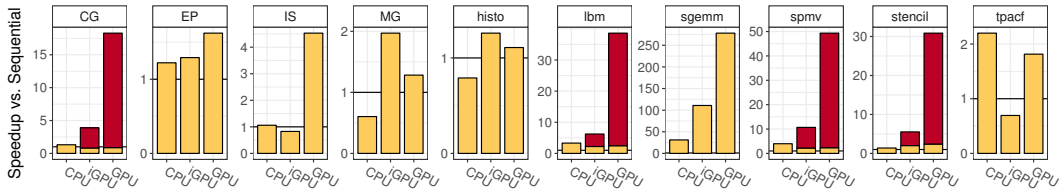


Figure 5.21: Speedup compared to the sequential C program. Results for the best performing heterogeneous API on each device are shown. The red bars indicate a manual runtime optimization for avoiding unnecessary data transfers.

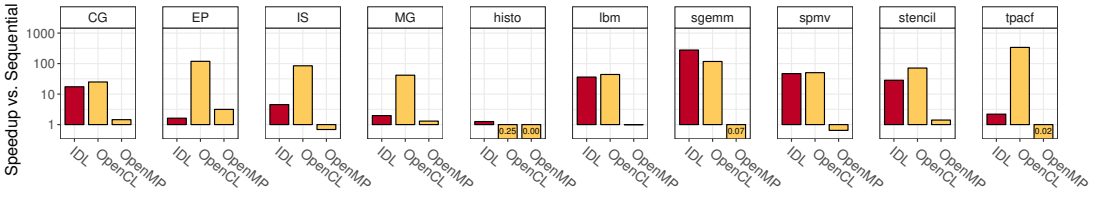


Figure 5.22: Speedup of our constraints based approach (executed on the best hardware and highlighted in red) compared to handwritten parallel OpenCL (executed on the GPU) and OpenMP (executed on the CPU) implementations.

	CPU						iGPU					GPU			
	MKL	libSPMV	Halide	clBLAS	CLBlast	Lift	clSPARSE	libSPMV	clBLAS	CLBlast	Lift	cuSPARSE	libSPMV	cuBLAS	Lift
CG	**1504.21**	—	—	—	—	—	**644.02**	—	—	—	—	**113.51**	—	—	—
EP	—	—	—	—	—	**32762.50**	—	—	—	—	**30983.40**	—	—	—	**24680.70**
IS	—	—	**426.95**	—	—	1765.61	—	—	—	—	**547.28**	—	—	—	**99.95**
MG	—	—	—	—	—	**4699.63**	—	—	—	—	**1439.58**	—	—	—	**2211.56**
histo	—	—	—	—	—	**27.42**	—	—	—	—	**17.20**	—	—	—	**19.54**
lbm	—	—	—	—	—	**6457.93**	—	—	—	—	**5335.09**	—	—	—	**590.60**
sgemm	**53.50**	—	—	1661.75	660.44	1339.15	—	—	**14.73**	19.03	15.04	—	—	**5.99**	7.87
spmv	—	**218.17**	—	—	—	—	—	**102.233**	—	—	—	—	**18.437**	—	—
stencil	—	—	**5760.81**	—	—	21951.80	—	—	—	—	**2261.48**	—	—	—	**279.38**
tpacf	—	—	—	—	—	**19276.40**	—	—	—	—	**61111.90**	—	—	—	**23358.20**

Figure 5.23: Detailed performance results for each heterogeneous API used in milliseconds. Fastest implementations for each benchmark and target hardware are highlighted in bold.

Chapter 6

Building a Fully integrated Idiom Specific Optimization Pipeline

Sparse linear algebra is central to many scientific codes, yet compilers fail to optimize it well. Instead, programmers rely on library implementations that are hand optimized and can utilize accelerator hardware. This comes at a cost, however, as it ties programs into vendor specific software ecosystems and results in non-portable code.

This chapter develops an approach based on the *specification Language for implementers of Linear Algebra Computations* (LiLAC). Rather than requiring application developers to (re)write every program for a given library, the burden is shifted to a *one-off* description for the library implementer. The LiLAC-enabled compiler then uses this to incorporate library routines automatically.

LiLAC provides automatic data marshaling, seamlessly maintaining state between calls as needed and minimizing data transfers. Appropriate places for library replacement are detected at the compiler intermediate representation level, making our approach language independent.

We evaluate on legacy large-scale scientific applications written in FORTRAN; standard benchmarks written in C/C++ and FORTRAN; and C++ graph analytics kernels. Across heterogeneous platforms, applications and data sets we show performance improvements of 1.1x to over 10x without any user intervention.

6.1 Introduction

Linear algebra is an important component of many compute intensive applications. While there are many compiler papers examining automatic optimization of dense linear algebra, sparse codes have received less attention. Sparse algorithms are, however, increasingly important as the basis for graph algorithms and data analytics [49].

Ideally, we would like compilers to automatically map dense and sparse-based codes to

heterogeneous compute platforms efficiently and with no user intervention. However, this has proven difficult.

Instead, we see the wide-scale provision of fast libraries [71, 5, 42], often provided by hardware vendors themselves. They provide excellent performance, but place the burden of code rewriting on the application developer and are rarely portable across platforms. Rewriting legacy applications involves considerable effort, especially when using hardware accelerators that require careful data marshaling. In fact, the difficulty of efficient integration is a key impediment to the wider use of accelerator libraries. Furthermore, source code modifications reduce the portability of the program and require a commitment to specific hardware vendors, resulting in legacy code bases that quickly become obsolete.

In this chapter, we reexamine the way that compilers and libraries are used to tackle the challenge of achieving full performance without requiring any application programmer effort. Highly tuned and platform specific libraries are invariably the fastest implementations available, therefore we use them as our building blocks. We then develop a new specification language for implementers of such libraries, the *specification Language for implementers of Linear Algebra Computations* (LiLAC).

Using LiLAC, library implementers specify with a few lines of code, *what* a library does and *how* it is invoked. Our compiler then determines where the library specification matches the user's code and automatically rewrites the code to utilize library calls.

LiLAC-What is a high level language to describe sparse and dense linear algebra programs. The LiLAC compiler uses it to detect such functionality in user applications at the level of compiler intermediate representation. It is powerful enough to formulate linear algebra, yet remains independent of compiler internals and is easy to understand and program.

LiLAC-How is a language to specify how parts of the LiLAC-What description can be used for library invocations and how state can be retained in between calls. Our compiler uses this to automatically generate library call *harnesses* that efficiently schedule memory transfers, execute setup code and handle hardware context management.

Our approach is broadly applicable. To demonstrate its applicability to legacy FORTRAN codes, we improved and extended the flang frontend to LLVM and evaluated on a large-scale scientific application. It also works on standard C/C++ and FORTRAN benchmark programs and C++ graph analytics kernels. The presented LiLAC system achieves significant performance improvement across heterogeneous platforms, from 1.1x to over 10x, without any user intervention.

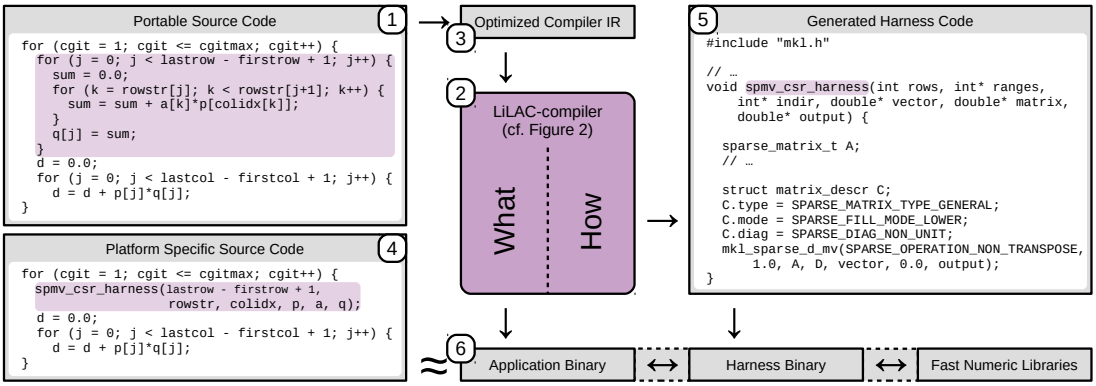


Figure 6.1: LiLAC applied to NPB Conjugate Gradient: Code that matches the LiLAC-What specification is replaced with calls to a harness function. The harness is generated from a LiLAC-How specification to utilize Intel MKL.

6.2 Overview

Consider the example in Figure 6.1. In the top left corner (1), we see unmodified application source code. This is *conjugate gradient* from the NAS-PB suite. The program is written in a straightforward manner and a naively compiled program would fail to exploit the full potential of modern hardware.

In order to achieve good performance on Intel processors, we provide a LiLAC specification of Intel MKL. The LiLAC compiler (2) uses this to detect that the code in the framed loop is a sparse matrix-vector product. Instead of passing it on to the generic compiler backend, it replaces it with a call to an auto-generated *harness* function and captures the parameters of the computation as function call arguments. This is performed on optimized intermediate representation code (3) and results in a program that is equivalent to the source code shown in the bottom left (4).

LiLAC also automatically generates the corresponding harness as shown at the right of the figure (5). The resulting shared library is then linked with the application binary at runtime, interfacing the underlying library implementation (6).

6.2.1 Implementation Overview

Figure 6.2 provides more detail on the internals of LiLAC. On the left we can see the LiLAC specification provided by the library implementer - just 16 lines of code. It consists of a *What* and a *How* part. These two parts are processed by the LiLAC system, eventually resulting in a runtime library and a modified C/C++ compiler.

LiLAC-What specifies the functionality that is provided by a library, in this example *spmv-csr* (cf. Figure 6.1). From this specification, a detection program that finds the computation in normalized LLVM IR code is automatically generated and the harness interface is determined.

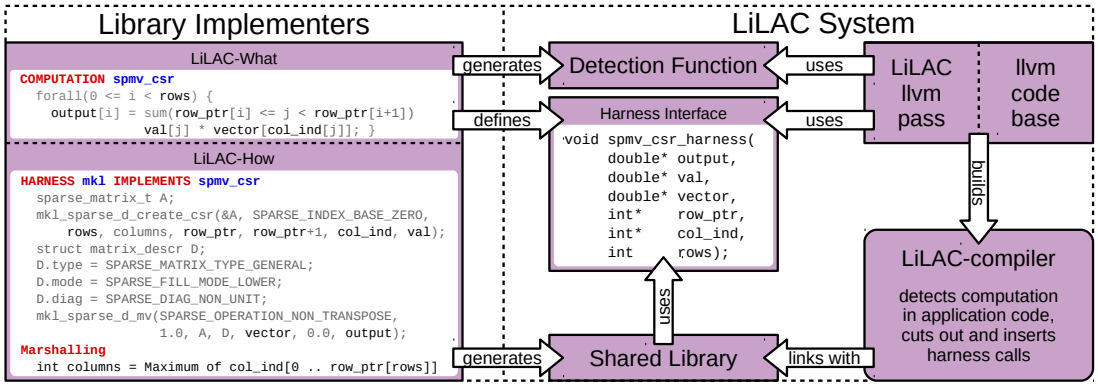


Figure 6.2: Overview of the LiLAC internals: On the left is the entire LiLAC program the library implementer must write. This results in a modified LiLAC-compiler in the bottom right that behaves as the shown in Figure 6.1.

LiLAC-How specifies how the library, in this case Intel MKL, can be invoked to perform the specified calculation. This involves boilerplate code, but also advanced features, such as optimized data transfers to accelerators. In the last two lines, we can see how to efficiently compute the *columns* variable with LiLAC. It is not present in the *LiLAC-What* definition but is required by the MKL library. It cannot be captured statically and has to be computed at runtime using the values in the *row_ptr* array. Using *Marshalling*, LiLAC automatically generates the harness such that this is only recomputed if the values in *row_ptr* change.

On the right of the figure we can see how the components generated from the LiLAC specification are used to build the LiLAC-compiler. Based on the LLVM infrastructure, it implements a transformation pass that is executed after the normal optimization pipeline. Using the generated detection function, it finds instances of the computation and replaces them with calls to the specified harness interface. Application binaries are dynamically linked to the harness library.

6.3 What and How

This section describes in more detail the two components of the LiLAC language. LiLAC-What describes what a specific library does and LiLAC-How describes how user code is bound to the underlying implementation.

6.3.1 LiLAC-What: Functional Description

At the heart of our approach is a simple language to specify sparse and dense linear algebra operations. This serves two purposes in our LiLAC system: Firstly, it is used to generate a detection program for finding the computation in user code. Secondly, it identifies the variables

that are arguments to the library, thus defining the harness interface.

The key challenge in the design of this language was to stay simple enough to allow automatic generation of robust detection functionality, yet to be able to capture interesting functionality. Crucial for sparse linear algebra routines is capturing the many different memory access patterns, the control flow on the other hand is very rigid. This is reflected in the grammar as shown in Figure 6.3.

6.3.2 Sparse Matrix Variations in LiLAC-What

Sparse matrices can be stored in different formats. In this section we introduce two of them and show how LiLAC-What can express the corresponding computations.

6.3.2.1 Compressed Sparse Row

For Compressed Sparse Row (CSR) [86], all non-zero entries are stored in a flat array **val**. The **col_ind** array stores the column position for each value. Finally, the **row_ptr** array stores the beginning of each row of the matrix as an offset into the other two arrays. The number of rows in the matrix is given directly by the length of the **row_ptr** array minus one, however the number of columns is not explicitly stored. In Figure 6.4, a 5x5 matrix is shown represented in this format, the LiLAC-What code is in the top left of Figure 6.2.

6.3.2.2 Jagged Diagonal Storage

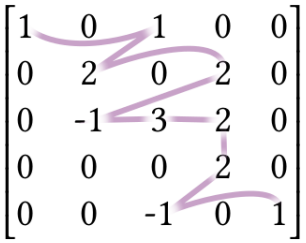
For Jagged Diagonal Storage (JDS) [85], the rows of the matrix are permuted such that the number of nonzeros per row decreases. The permutation is stored in a vector **perm**, the number of nonzeros in **nzcnt**. The nonzero entries are then stored in an array **val** in the following order: The first nonzero entry in each row, then the second nonzero entry in each row etc. The array **col_ind** stores the column for each of the values and **jd_ptr** stores offsets into **val** and **col_idx**. The product of a sparse matrix in JDS format with a dense vector is specified in LiLAC-What at the bottom of Figure 6.5.

6.3.3 LiLAC-How

Where LiLAC-What specifies the computations that are implemented by a library, LiLAC-How describes how precisely library calls are to be be used for performing them. The langauge was designed with important existing libraries such as cuSPARSE, clBLAS and Intel MKL in mind. In order to support the idiosyncrasies of these libraries, the specifications need to be built around boilerplate C++ code that manages the construction of parameter structures, calling conventions etc. However, it is also crucial for the specification language to be as high-level as as possible without sacrificing any performance, and the fine balance between

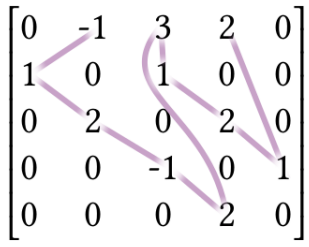
$$
\begin{aligned}
\mathit{program} &::= \textbf{COMPUTATION}\ \langle \mathit{name} \rangle\ \langle \mathit{body} \rangle \\
\mathit{body} &::= \langle \mathit{forall} \rangle \mid \langle \mathit{dotop} \rangle \\
\mathit{range} &::= \textbf{(}\ \langle \mathit{exp} \rangle\ \textbf{<=}\ \langle \mathit{name} \rangle\ \textbf{<}\ \langle \mathit{exp} \rangle\ \textbf{)} \\
\mathit{forall} &::= \textbf{forall}\ \langle \mathit{range} \rangle\ \textbf{\{}\ \langle \mathit{body} \rangle\ \textbf{\}} \\
\mathit{dotp} &::= \langle \mathit{addr} \rangle\ \textbf{=}\ \textbf{dot}\ \langle \mathit{range} \rangle\ \langle \mathit{addr} \rangle\ \textbf{*}\ \langle \mathit{addr} \rangle\ \textbf{;} \\
\mathit{addr} &::= \langle \mathit{name} \rangle\ \{\ \textbf{[}\ \langle \mathit{exp} \rangle\ \textbf{]}\ \} \\
\mathit{add} &::= \langle \mathit{exp} \rangle\ \textbf{+}\ \langle \mathit{exp} \rangle \\
\mathit{mul} &::= \langle \mathit{exp} \rangle\ \textbf{*}\ \langle \mathit{exp} \rangle \\
\mathit{exp} &::= \langle \mathit{name} \rangle \mid \langle \mathit{cnst} \rangle \mid \langle \mathit{addr} \rangle \mid \langle \mathit{add} \rangle \mid \langle \mathit{mul} \rangle
\end{aligned}
$$

Figure 6.3: Grammar of the LiLAC-What language



$$
\begin{aligned}
\textbf{val} &= \begin{bmatrix} 1 & 1 & 2 & 2 & -1 & 3 & 2 & 2 & -1 & 1 \end{bmatrix} \\
\textbf{col\_ind} &= \begin{bmatrix} 0 & 2 & 1 & 3 & 1 & 2 & 3 & 3 & 2 & 4 \end{bmatrix} \\
\textbf{row\_ptr} &= \begin{bmatrix} 0 & 2 & 4 & 7 & 8 & 10 \end{bmatrix}
\end{aligned}
$$

Figure 6.4: Compressed Sparse Row representation as used by the LiLAC-What example in Figure 6.1



$$
\begin{aligned}
\textbf{perm} &= \begin{bmatrix} 1 & 2 & 0 & 4 & 3 \end{bmatrix} \\
\textbf{val} &= \begin{bmatrix} -1 & 1 & 2 & -1 & 2 & 3 & 1 & 2 & 1 & 2 \end{bmatrix} \\
\textbf{col\_ind} &= \begin{bmatrix} 1 & 0 & 1 & 2 & 3 & 2 & 2 & 3 & 4 & 3 \end{bmatrix} \\
\textbf{jd\_ptr} &= \begin{bmatrix} 0 & 5 & 9 & 10 \end{bmatrix} \\
\textbf{nzcnt} &= \begin{bmatrix} 3 & 2 & 2 & 2 & 1 \end{bmatrix}
\end{aligned}
$$

```
COMPUTATION spmv_jds
  forall(0 <= i < rows) {
    output[perm[i]] = sum(0 <= j < nzcnt[i])
      val[jd_ptr[j]+i] * vector[col_ind[jd_ptr[j]+i]]; }
```

Figure 6.5: Jagged Diagonal Storage in LiLAC-What

these requirements drove the design of the language. In particular, LiLAC-How abstracts away memory transfers.

The result is a language that can be further divided into two interacting components. Firstly, it describes boilerplate code that is required for individual library call invocations in a *harness*. Secondly, it is used for data *marshaling* between the core program and the library, which is particularly crucial for heterogeneous compute environments. Figure 6.6 shows the grammar specification of LiLAC-How.

6.3.3.1 Individual Library Invocations

We need to encapsulate the boilerplate code that any given library requires, such as setup code, filling of parameter structures etc. This part of the language is straightforward.

6.3.3.1.1 Harness The harness construct is the central way of telling the LiLAC system how a library can be used to perform a computation that was specified in LiLAC-What. As we can see at the top of Figure 6.6, a harness refers to a LiLAC-What program by name and also has a name itself. It is built around some C++ code, which can use all the variables from the LiLAC-What program to connect with the surrounding program. It also needs to specify the relevant C++ header files that the underlying library requires. Lastly, the harness can incorporate persistent state and utilize data marshaling functionality.

6.3.3.1.2 Persistence Many libraries need setup and cleanup code, which is specified with the keywords *BeforeFirstExecution* and *AfterLastExecution*. These are used in combination with *PersistentVariables*, allowing state to persist between harness invocations, e.g. to retain handlers to hardware accelerators.

6.3.3.1.3 Example In Figure 6.7, we see a trivial LiLAC-What program for implementing `spmv_csr` with the Intel MKL library. The actual call to the relevant library function is in line 16. To prepare for that call, there is boilerplate code in lines 7–14 to fill parameter structures.

Critically, there is an additional parameter required by the library that is not part of the LiLAC-What specification: the number of columns, *cols*, in the sparse matrix. It is determined at runtime, in lines 2–5, leading to reduced performance. We will avoid this with the data marshaling constructs in the next section.

6.3.3.2 Data Marshaling

Heterogeneous accelerators require data transfers to keep memory consistent between host device and accelerator. To achieve the best performance, these have to be minimized.

Importantly, unchanged data should never be copied again. This is not trivial to achieve, as it requires dynamic checks. LiLAC-How uses hardware memory protection to implement this with minimal run time overhead in generated harnesses.

This data *marshaling* comes in two flavors; INPUT and OUTPUT. Both of these are bound to a range of memory in the harness. The LiLAC-generated code then uses hardware memory protection to detect read and write accesses to these memory ranges. In the specification, the underlying array is available using the identifiers `in`, `size`, and `out`.

6.3.3.3 Detailed Example

In Figure 6.8, LiLAC-How functionality is specified using the `cudaMemcpy` function from NVIDIA CUDA. It is used to copy data from the host to the accelerator. Note that for this to work, it first needs to allocate memory of the device using `cudaMalloc`, which is later freed with `cudaFree`. `cudaMemcpy` is executed only when a value in the array changes, resulting in minimal memory transfers.

We can use the same construct to efficiently compute values such as the `cols` variable in Figure 6.7, as shown in Figure 6.9. The optimized implementation is nearly identical to the code in Figure 6.7 lines 2-5, however instead of the concrete variable name the reserved identifiers `in`, `size`, and `out` are used.

We can now use these in Figure 6.10, showing a CSR SPMV LiLAC-How program for the cuSPARSE library. A number of data marshaling variables are introduced in lines 12–17, that automatically optimize both memory transfers and the computation of the `cols` variable. The core of the harness in lines 2–10 is again nothing more than library specific boilerplate C++ code.

6.4 Implementation

The LiLAC system, as shown in Figure 6.2 is entirely integrated into the LLVM build system. When LLVM is compiled, the LiLAC specification is parsed using a Python program. Based on the LiLAC-What and LiLAC-How sections, C++ code is generated that is automatically incorporated into LLVM in further stages of the build process.

The result is an LLVM optimization pass that is available when linking LLVM with the clang C/C++ compiler. This performs the discovery of linear algebra code and the insertion of harness calls. Furthermore, the harness libraries themselves are built at compile time of LLVM, using C++ code emitted from the LiLAC-How sections.

The two crucial implementation details are therefore: Firstly, how automatic detection functionality in C++ can be generated from the abstract LiLAC-What specifications and secondly, how the LiLAC-How sections are used to generate fast C++ implementations of the specified library harnesses.

6.4.1 LiLAC-What

Once the LiLAC-What sections have been parsed, they are turned into C++ functionality that is used by the LiLAC LLVM pass to detect appropriate places for harness calls. Instead of performing this via syntax-level pattern matching, we use optimized intermediate compiler representation. The standard optimizations for `-O2` without vectorization and loop unrolling are used. This results in code in a normal form, which allows us to abstract away many syntactic differences and programming languages.

The effect is demonstrated in Figure 6.11, which shows three implementations of a dot product in different languages: C, C++ and FORTRAN. After translating to LLVM IR and performing optimizations, the dot product is recognized in the LiLAC system using the same LiLAC-What specification.

The exact detection algorithm works in two steps. Firstly, the control flow skeleton is recognized. This is simple, as the control flow that can be expressed in LiLAC-What is limited, and we only need to detect loop nests of a certain depth. After candidate loop nests have been identified, the index and loop range calculations from LiLAC-What are mapped onto the LLVM IR nodes. This is done via a backtracking search procedure that requires some careful

consideration in order to get robust results on implementation details of LLVM, particularly for different ways of array indexing, pointer casts and integer conversions.

6.4.1.1 Backtracking Search Algorithm

In order to match instances of LiLAC-What specifications in user programs, the LLVM IR segments that match the corresponding control flow skeleton are processed with a backtracking search algorithm.

All $\langle exp \rangle$ expressions in the LiLAC-What program are identified. These now have to be assigned instructions or other values from the LLVM IR segment. The top-level $\langle exp \rangle$ expressions that are used as limits or iterators in $\langle range \rangle$ expressions are easily connected with the corresponding loop boundaries in the matched control flow structures.

The remaining expressions are then successively assigned by backtracking. Consider the example in Figure 6.12, which shows a candidate loop from the LLVM IR generated from the C++ dot product code in Figure 6.11. The iteration space is determined by loop analysis and this immediately allows us to assign the LiLAC-What iterator and range in Figure 6.13. The LLVM IR instructions that correspond to `left[i]`, `left`, `right[i]`, `right` and `result` are then searched for one by one. When a partial solution fails, such as at the point when `right` is assigned the same value as `left`, we backtrack. If we cannot determine a solution this way, we discard the control flow candidate.

6.4.1.2 Code Replacement

For each identified loop nest that matches a LiLAC-What specification, the code is replaced with a harness call. To minimize the invasiveness of our pass, this is performed as follows: Firstly, a harness call is inserted directly before the loop. The function call arguments are then selected from the backtracking result and passed to the harness. Secondly, the LLVM instruction that stores the result of the computation or passes it out of the loop as a phi node is removed. The remainder of the loop nest is be removed automatically by dead code elimination.

6.4.2 LiLAC-How

To generate harnesses, LiLAC uses C++ templates. The LiLAC-How syntax elements that take C++ code are used to generate generic functions, and template parameter deduction inserts concrete types later.

Consider Figure 6.14, generated from LiLAC-How code in Figure 6.8. The correspondence between generated C++ and LiLAC-How code is clearly visible in the three functions that contain the source code specified in LiLAC-How. These functions are used to specialize the *ReadObject* class template. It guarantees the following properties using hardware memory

protection:

construct is called before the first invocation and when `in` or `size` change for consecutive harness invocations.

update is called after *construct* and whenever any of the data in the array changes between consecutive harness invocations.

destruct is called in between consecutive *construct* calls and before the program terminates.

6.4.3 FORTRAN

Compatibility with Fortran was a key implementation hurdle. The LLVM frontend under active development, flang, is in an unfinished state and produces unconventional LLVM IR code by default. Significant additional work was required to normalize the IR code. We developed normalization passes in LLVM to overcome the specific shortcomings, enabling Fortran programs to be managed as easily as C/C++.

The problems that we encountered included: the differing indexing conventions required offsetting pointer variables on a byte granularity with untyped pointers; Pointer arguments are passed untyped, obfuscating the intermediate representation types to `i64` pointers, frequently necessitating a pointer type conversion followed by a load from memory; obfuscated loops with additional induction variable that counts down instead of up such that the standard LLVM **indvars** pass is unable to merge the loop iterators.

6.5 Experimental Setup

We wrote short LiLAC programs for a collection of linear algebra libraries. The approach was then applied to a chemical simulation application, two graph analytics applications and a collection of standard benchmark suites.

Libraries We selected four different libraries for sparse linear algebra functions. These were: Intel MKL [42], Nvidia cuSPARSE [71], clSPARSE [5] and SparseX [29]. MKL is a general-purpose mathematical library designed to provide easy-to-integrate performance primitives, while clSPARSE and cuSPARSE are OpenCL and CUDA implementations of sparse linear algebra designed to be executed on the GPU and SparseX uses an auto-tuning model and code generation to optimize sparse operations on particular matrices.

Applications To evaluate the impact of LiLAC in a real world context, the PATHSAMPLE physical chemistry simulation suite was used. This is a large FORTRAN legacy application [98] consisting of over 40,000 lines of code. Recent work shows that applications in this area are amenable to acceleration using sparse linear algebra techniques [95], and PATHSAMPLE provides a useful example of this. We also evaluated two modern C++ graph analytics kernels (BFS and PageRank [25, 14]). PATHSAMPLE was run in two different modes and three different

Name	Hardware	Libraries
Intel-0	2× Intel Xeon E5-2620	MKL
	Nvidia Tesla K20 GPU	cuSPARSE
Intel-1	Intel Core i7-8700K	clSPARSE
	Nvidia GTX 1080 GPU	SparseX
AMD	AMD A10-7850K	cuSPARSE
	AMD Radeon R7 iGPU	clSPARSE ×2
	Nvidia Titan X GPU	SparseX

Table 6.1: Evaluated platforms and library harnesses; AMD-0 supports clSPARSE on both its internal and its external GPU.

levels of pruning, in each case using a system of 38 atoms [28] commonly used to evaluate applications in this domain. The graph kernels were run against 10 matrices from the University of Florida's sparse matrix collection from Davis and Hu [24], with sizes between 300K and 80M non-zero elements.

For completeness and validation that our LiLAC-generated implementations were correct, we also applied our technique to sparse codes from standard benchmark suites: CG from the NAS parallel benchmarks [11], spmv from Parboil [91] and the Netlib sparse benchmark suites [27]. Each benchmark suite was run using their supplied inputs.

6.5.0.0.1 Platforms We evaluated our approach across 3 different machines with varying hardware performance and software availability. Each one was only compatible with a subset of our LiLAC-generated implementations—a summary of these machines is given in Table 6.1.

6.6 Results

We show that across a variety of hardware and software platforms, LiLAC can speed up real-world applications. We present raw performance impact first, then we analyze two intermediate metrics: the reliability of linear algebra discovery and the effectiveness of memory transfer optimizations.

6.6.1 Performance

The speedups that LiLAC achieves on real applications as well as benchmarks are shown in Figure 6.15. They range from 1.1–3× on the scientific application codes to 12× on well-known sparse benchmark programs.

Platform	Implementation	PFold			NGT			PageRank			
		L0	L1	L2	L0	L1	L2	Erdos	LJ-2008	Road	Er
Intel-0	MKL	2.88	2.46	1.00	1.18	1.18	1.18	1.25	2.93	1.72	2.
	cuSPARSE	0.75	0.60	0.45	0.66	0.66	0.66	1.39	1.00	3.32	0.
	clSPARSE	0.90	0.75	0.46	0.81	0.79	0.78	1.24	0.95	2.24	0.
	SparseX	-	-	-	-	-	-	-	-	-	1.
Intel-1	MKL	2.70	2.43	1.01	1.20	1.20	1.19	1.63	1.03	2.26	1.
	cuSPARSE	0.48	0.41	0.30	0.68	0.69	0.68	1.59	0.87	4.44	1.
	clSPARSE	1.00	1.00	1.00	1.00	1.02	1.00	1.50	0.87	3.46	0.
	SparseX	-	-	-	-	-	-	-	-	-	1.
AMD	cuSPARSE	1.38	1.18	0.67	0.69	0.69	0.70	3.44	1.18	9.97	1.
	clSPARSE (eGPU)	2.17	1.82	1.22	1.16	1.16	1.16	3.08	1.24	6.06	0.
	clSPARSE (iGPU)	2.03	1.78	1.03	0.90	0.90	0.90	3.26	1.31	4.05	0.
	SparseX	-	-	-	-	-	-	-	-	-	1.

Table 6.2: LiLAC speedups on each platform, across different applications and problem sizes. SparseX demonstrated promising performance on some applications, but we were unable to evaluate on every relevant instance due to instability.

On the PATHSAMPLE applications (PFold and NGT), we measured consistent speedups of approximately 50% and 10% respectively across all 3 platforms. For large applications, Amdahl's law is a severe limitation for approaches like ours — other parts of the applications dominate execution times when linear algebra is accelerated.

PageRank requires a large number of SPMV calls using the same input matrix to iterate until convergence. The GPU implementations running on AMD and Intel-1 are able to take advantage of data remaining in memory. The larger number of CPU cores and slower GPU available on Intel-0 make MKL its best-performing implementation. CPU implementations perform best on BFS by avoiding memory copies entirely — on AMD, SparseX outperforms GPU implementations.

On standard sparse linear algebra benchmarks, LiLAC achieves speedups of up to 12×. The impact is independent of the source language, as the C and FORTRAN versions of the Netlib benchmark demonstrate. LiLAC is able to achieve consistent, useful speedups across a variety of hardware configurations.

Figure 6.16 demonstrates that achieving good performance independent of the hardware platform is no small feat. It shows the distribution of speedups on the NPB-CG benchmark across different machines, problem sizes and linear algebra implementation. More detailed performance data is in Table 6.2. The best-performing implementation varies considerably, depending on characteristics of the problem in question. No accelerator library performs well reliably, each harness outperforms any other harness on some combination of data set and platform. clSPARSE on the AMD iGPU is the only harness that is consistently outperformed.

We verify that LiLAC can reach near-optimal performance, despite using general and portable harnesses, by comparing the achieved speedups with the performance of reference

implementations. This was possible on the NPB and Parboil benchmarks, where expert hand crafted version are available that achieve close to peak performance.

Figure 6.17 shows our results from these experiments. For context, we also measure application programmer effort for peak performance, which is substantial, requiring hundreds of lines of carefully crafted high performance OpenCL code. On the NPB-CG benchmark, we achieve $\sim 3\times$ speedup, while the expert implementation is approximately $3\times$ faster than LiLAC. This is partly due to Amdahl's law — the benchmark also performs operations other than sparse matrix-vector multiplication and the expert implementation is a complete rewrite that performs all these operations on the GPU. On Parboil SPMV, the difference between an expert and LiLAC is only $1.07\times$.

After moving the burden of targeting accelerators from the user to the library vendor with LiLAC, we would also like to automate the selection of which library to use. This may change over time and per application, as we saw in Figure 6.16. We have experimented with simple classifiers for each platform, that dynamically select the best library given the data input. This was straightforward to achieve for our problem space, but more data is needed for a meaningful evaluation, and we leave it to future work.

6.6.2 Effectiveness of Data Marshaling

Our implementation of LiLAC relies on a non-trivial data marshaling system that prevents redundant computations and memory transfers. We present performance results that show the importance and effectiveness of this system.

We repeated our experiments, using the best-performing implementations from Figure 6.15. Instead of using our data marshaling scheme, we transfer memory naively for each accelerator invocation. We then measured the performance advantage that our harness system gives us over this naive approach, results are in Figure 6.18. The GPU implementations received speedups of 1.4–3.5$\times$ and SparseX was sped up by over $25\times$ (because it performs an internal matrix tuning phase that is far more expensive than a memory copy). These speedups are large enough that in each case, the application would in fact have performed worse than their sequential baselines without performing data marshaling.

6.6.3 Reliability of Discovery

To achieve performance impact, LiLAC needs to reliably detect linear algebra computations, independent of coding style and source programming language. The results so far demonstrated that this works reliably and the data in Table 6.3 reiterates this. Established approaches, like the polyhedral model, on the otehr hand, are unable to model sparse linear algebra. We verified this with the Polly compiler and found that no SCoPs captured sparse linear algebra. This is unsurprising, as the basic assumptions of affine memory access are contradicted by the

indirection that is inherent in sparsity. Similarly, the Intel icc and ifort compilers fail to auto-parallelize, as they cannot reason about sparsity and have to assume additional dependencies where sparse accesses occur.

6.7 Related Work

Compiler centric linear algebra optimization Compiler management of indirect memory accesses was first studied using an inspector-executor model for distributed-memory machines in Baxter et al. [13]. Here the location of read data was discovered at runtime and appropriate communication automatically inserted. Later work by Pottenger and Eigenmann [78], Fisher and Ghuloum [30], Rauchwerger and Padua [81], Suganuma et al. [92] was focused on efficient runtime dependence analysis and the parallelization of more general programs. However, the performance achieved by these approaches is modest due to runtime overhead and falls well short of library performance.

Polyhedral compiler approaches The polyhedral model is a well established approach for modeling data dependencies in compilers, with relevant publications by Redon and Feautrier [83], Jouvelot and Dehbonei [47], Chi-Chung et al. [22], Gupta and Rajopadhye [37], Stock et al. [90]. Polyhedral optimizers have been implemented in mainstream C/C++ compilers, notably are the Polly extensions to LLVM described in [26]. Recent work by Baghdadi et al. [9] has extended the polyhedral model beyond affine programs to some forms of sparsity with the PENCIL extensions. These can be used to model important features of sparse linear algebra, such as counted loops [108], meaning loops with dynamic, memory dependent bounds but statically known strides. Such loops are central to sparse linear algebra. It uses the PPCG compiler [97] to automatically detect relevant code regions, but it relies on well behaved C code with all arrays declared in variable-length C99 array syntax. Unfortunately such restrictions prevent application to real world programs; none of our benchmarks or data sets have this

Table 6.3: Sparsity does not fit the polyhedral model, Intel compilers fail to parallelize and Polly is not available for FORTRAN. Only LiLAC detects sparse linear algebra reliably.

Benchmark	LiLAC	Polly	Intel icc/ifort
PFold	CSR	-	parallel dependence
NGT	CSR	-	parallel dependence
Parboil-SPMV	JDS	no SCoP	parallel dependence
BFS	CSR	no SCoP	parallel dependence
NPB-CG	CSR	-	parallel dependence
PageRank	CSR	no SCoP	parallel dependence
Netlib C	CSR	no SCoP	parallel dependence
Netlib Fortran	CSR	-	parallel dependence

structure.

Compiler detection There has been previous work on detecting code structures in compilers using constraint programming. Early work was based on abstract computation graphs [77], but more recent approaches have used real compiler intermediate code and made connections to the polyhedral model [33].

In [34] they implement a method that operates on SSA intermediate representation. It uses a general-purpose low level constraint programming language aimed at compiler engineers. The paper focuses on code detection, with library calls inserted by hand. It is performed naively without the use of harnesses and the results show significant performance degradation due to the redundant memory transfers that have to be amended manually. Other advanced approaches to extracting higher level structures from assembly and well structured FORTRAN code involve temporal logic [60, 48]. These approaches tend to focus on a more restricted set of computations (dense memory access). While this allows formal reasoning about correctness, is too restrictive to model sparse linear algebra.

Domain Specific Languages There have been multiple domain specific libraries proposed to formulate linear algebra computations. Many of these contain some degree of autotuning functionality to achieve good performance across different platforms [93]. Halide [80] was designed for image processing. [94]. Its core design decision is the scheduling model that allows the separation of the computation schedule and the actual computation. There has been work on automatically tuning the schedules [66] but in general the computational burden is put on the application programmer. The MILK programming model [50] extends C++ with pragmas to annotate indirect memory accesses. This allows low level optimizations that are applicable to sparse linear algebra. However, the approach is unable to utilize the much greater potential of heterogeneous compute and requires detailed programmer intervention.

Libraries The most established way of encapsulating fast linear algebra is via dedicated library implementations, generally based on the BLAS interface specification [16]. These are generally very fast on specific hardware platforms, but require application programmer effort and offer little performance portability. Implementations of dense linear algebra are available for most suitable hardware platforms, such as cuBLAS [70] for NVIDIA GPUs, clBLAS [4] for AMD GPUs and the Intel MKL library [42] for Intel CPUs and accelerators.

There are fewer implementations of sparse linear algebra, but they exist for the most important platforms, including cuSPARSE [71] for NVIDIA GPUs and clSPARSE [5] built on top of OpenCL. Several BLAS implementations attempt platform independent acceleration and heterogeneous compute, among them Wang et al. [99], Moreton-Fernandez et al. [64, 63].

CPU-GPU data transfer optimizations Data transfers between CPU and GPU have been studied extensively as an important bottleneck for parallelization efforts. Previous work [43] established a system for automatic management of CPU-GPU communication. The authors of [53] implemented a system to move OpenMP code to GPUs, optimizing data transfers using data flow analysis. However, this approach performs a direct translation, not optimizing the code for the specific performance characteristics of GPUs.

6.8 Conclusion

This paper has presented LiLAC, a language and compiler that enables existing code bases to exploit sparse (and dense) linear algebra accelerator libraries. No effort is required from the application programmer. Instead, the library implementer provides a few lines of specification, which LiLAC uses to automatically and efficiently matches user code to underlying high performance libraries.

We demonstrated this approach on C, C++ and FORTRAN benchmarks as well as legacy applications and shown significant performance improvement across platforms and data sets. Many language features of the LiLAC language could be repurposed for libraries beyond linear algebra. In future work, we will investigate how our framework could be adapted to other application domains, enabling effort free access to an even larger set of accelerator libraries.

$$
\begin{aligned}
\mathit{harness} ::= {} & \textbf{HARNESS}\ \langle \mathit{name} \rangle\ \textbf{IMPLEMENTS}\ \langle \mathit{name} \rangle \\
& \langle C{+}{+} \rangle \\
& [\ \langle \mathit{marshalling} \rangle\ ]\ [\ \langle \mathit{persistence} \rangle] \\
& [\ \textbf{CppHeaderFiles}\ \{\ \langle \mathit{name} \rangle\ \}\ ] \\
\mathit{persistence} ::= {} & \textbf{PersistentVariables}\ \{\ \langle \mathit{name} \rangle\ \langle \mathit{name} \rangle\ \} \\
& [\ \textbf{BeforeFirstExecution}\ \langle C{+}{+} \rangle\ ] \\
& [\ \textbf{AfterLastExecution}\ \langle C{+}{+} \rangle\ ] \\
\hline
\mathit{marshalling} ::= {} & \textbf{Marshalling} \\
& \{\ \langle \mathit{type} \rangle\ \langle \mathit{name} \rangle\ \textbf{=}\ \langle \mathit{name} \rangle\ \textbf{of} \\
& \langle \mathit{name} \rangle\ [\ \textbf{0}\ \textbf{..}\ \langle \mathit{exp} \rangle\ ]\ \} \\
\mathit{input} ::= {} & \textbf{INPUT}\ \langle \mathit{name} \rangle\ \langle C{+}{+} \rangle \\
& [\ \textbf{BeforeFirstExecution}\ \langle C{+}{+} \rangle\ ] \\
& [\ \textbf{AfterLastExecution}\ \langle C{+}{+} \rangle\ ] \\
\mathit{output} ::= {} & \textbf{OUTPUT}\ \langle \mathit{name} \rangle\ \langle C{+}{+} \rangle \\
& [\ \textbf{BeforeFirstExecution}\ \langle C{+}{+} \rangle\ ] \\
& [\ \textbf{AfterLastExecution}\ \langle C{+}{+} \rangle\ ]
\end{aligned}
$$

Figure 6.6: Grammar of LiLAC-How

```
1  HARNESS mkl IMPLEMENTS spmv_csr
2    int cols = 0;
3    for(int i = 1; i < rowstr[rows]; i++)
4      cols = colidx[i]>cols?colidx[i]:cols;
5    cols = cols+1;
6
7    sparse_matrix_t A;
8    mkl_sparse_d_create_csr(&A, SPARSE_INDEX_BASE_ZERO,
9                            rows, cols, rowstr,
10                           rowstr+1, colidx, a);
11   struct matrix_descr dscr;
12   dscr.type = SPARSE_MATRIX_TYPE_GENERAL;
13   dscr.mode = SPARSE_FILL_MODE_LOWER;
14   dscr.diag = SPARSE_DIAG_NON_UNIT;
15
16   mkl_sparse_d_mv(SPARSE_OPERATION_NON_TRANSPOSE,
17                   1.0, A, dscr, iv, 0.0, ov);
18 PersistentVariables
19   "mkl.h"
```

Figure 6.7: This LiLAC-What code implements CSR SPMV naïvely with Intel MKL. Performance is degraded because of lines 2–5. Figure 6.9 will present a solution to this bottleneck.

```
INPUT CudaRead
  cudaMemcpy(out, in, sizeof(type_in)*size,
             cudaMemcpyHostToDevice);
BeforeFirstExecution
  cudaMalloc(&out, sizeof(type_in)*size);
BeforeFirstExecution
  cudaFree(out);
```

Figure 6.8: LiLAC-How code to provide efficient automatic data marshaling between host and CUDA accelerator.

```
INPUT Maximum
  out = in[0];
  for(int i = 1; i < size; i++)
    out = in[i]>out?in[i]:out;
  out = out+1;
```

Figure 6.9: *INPUT* can also be used to specify data dependent computations that are only recalculated when required.

```
HARNESS cuda IMPLEMENTS spmv_csr
  double alpha = 1.0;
  double beta  = 0.0;
  cusparseMatDescr_t descrA;
  cusparseCreateMatDescr(&descrA);
  cusparseDcsrmv(handle,
                 CUSPARSE_OPERATION_NON_TRANSPOSE,
                 rows, cols, ranges[rows], &alpha,
                 descrA, d_mat, d_ranges, d_indir,
                 d_vec, &beta, d_out);
Marshalling
  int      cols = Maximum of indir[0..ranges[rows]]
  double* d_mat = CudaRead of matrix[0..ranges[rows]]
  double* d_vec = CudaRead of vector[0..cols]
  int* d_ranges = CudaRead of ranges[0..rows+1]
  Int*  d_indir = CudaRead of indir[0..rowstr[rows]]
  double* d_out = CudaWrite of output[0..rows]
PersistentVariables
  cusparseHandle_t handle
BeforeFirstExecution
  cusparseCreate(&handle);
AfterLastExecution
  cusparseDestroy(handle);
CppHeaderFiles
  <cuda_runtime.h> "cusparse_v2.h"
```

Figure 6.10: This LiLAC-What specification implements an efficient SPMV harness using cuSPARSE in only 25 lines of code.

```
COMPUTATION dotproduct
    result = sum(0 <= i < length) left[i] * right[i];
```

```
int i = 0;
while(i < N) {
    x += (*(A+i))*(*(B+i));
    i+=1; }
```

```
for(int i = 0; i < vec_a.size(); i++)
    x += vec_a[i]*vec_b[i];
```

```
DO I = 1, N, 1
    X = X + A(i)*B(i)
END DO
```

Figure 6.11: Syntactically different computations in C, C++ or FORTRAN are captured by one LiLAC-What specification.

```
; <label>:17:
  %18 = phi i64 [ 0, %10 ], [ %26, %17 ]
  %19 = phi double [ 0.0, %10 ], [ %25, %17 ]
  %20 = getelementptr double, double* %9, i64 %18
  %21 = load double, double* %20
  %22 = getelementptr double, double* %12
  %23 = load double, double* %22
  %24 = fmul double %21, %23
  %25 = fadd double %19, %24
  %26 = add nuw i64 %18, 1
  %27 = icmp ugt i64 %14, %26
  br i1 %27, label %17, label %15
```

Figure 6.12: Optimizations remove language specific features, the result is normalized LLVM IR. Control flow candidates for a match are easily determined with standard loop analysis.

			left[i]	←	%21	
i	←	%18	left	←	%9	
length	←	%14	right[i]	←	%21	%23
			right	←	fail!	%12
			result	←		%19

Figure 6.13: Control flow candidates contain iterator ranges directly (left), backtracking identifies the dot product by assigning the other LiLAC-What entities one by one (right).

```
template<typename type_in, typename type_out>
void CudaRead_update(type_in* in, int size,
                     type_out& out) {
    cudaMemcpy(out, in, sizeof(type_in)*size,
               cudaMemcpyHostToDevice);
}
template<typename type_in, typename type_out>
void CudaRead_construct(int size, type_out& out) {
    cudaMalloc(&out, sizeof(type_in)*size);
}
template<typename type_in, typename type_out>
void CudaRead_destruct(int size, type_out& out) {
    cudaFree(out);
}
template<typename type_in, typename type_out>
using CudaRead = ReadObject<type_in, type_out,
    CudaRead_update<type_in,type_out>,
    CudaRead_construct<type_in,type_out>,
    CudaRead_destruct<type_in,type_out>>;
```

Figure 6.14: LiLAC uses code from Figure 6.8 to define three functions that specialize the ReadObject template, which uses `mprotect` for memory protection internally.

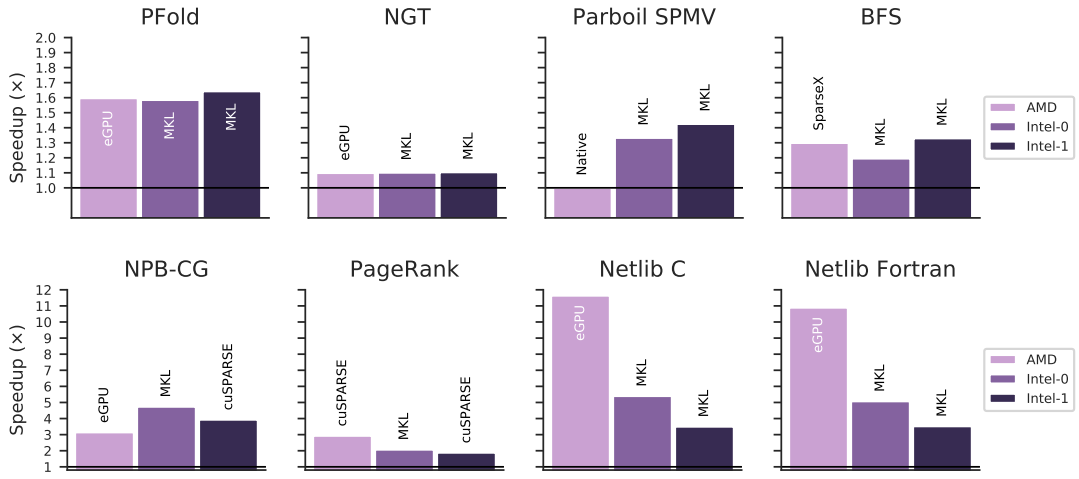


Figure 6.15: Evaluation on real-world application code and well-known benchmark suites: Bars show geometric mean speedup of best-performing LiLAC harness across the set of input examples for each benchmark and platform.

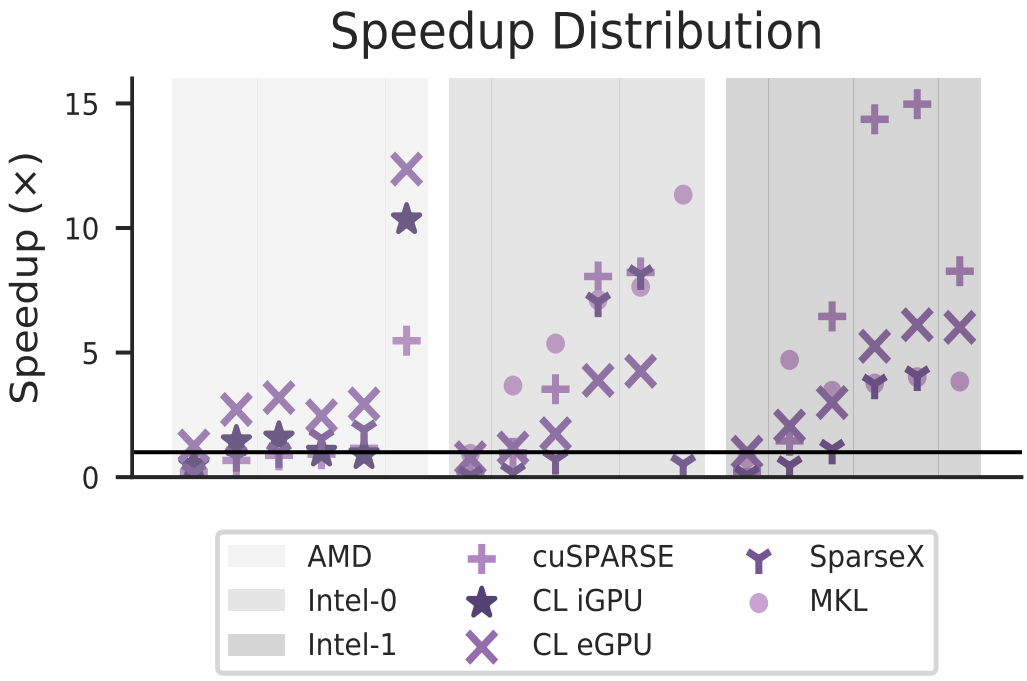
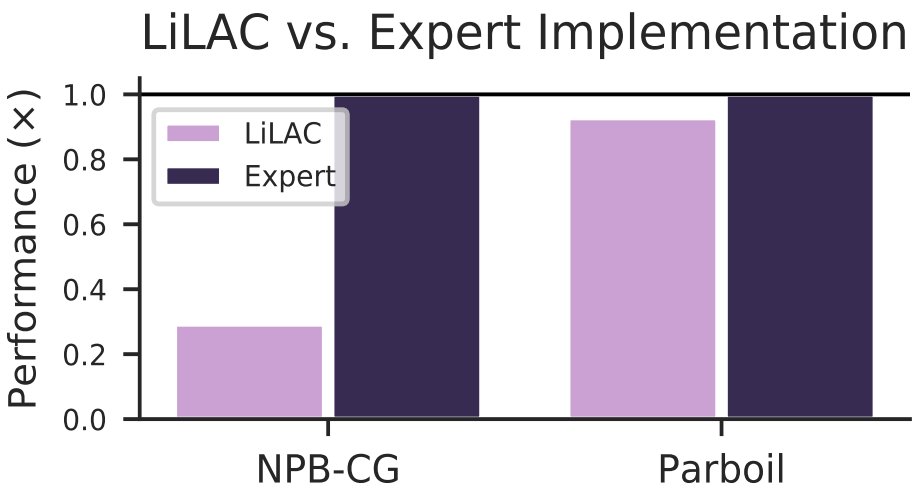


Figure 6.16: Distribution of speedups achieved by LiLAC on the NPB-CG benchmark. Each column of points shows the speedup achieved by each implementation available on a platform for a particular problem size. Data is sorted along the x-axis by the best speedup in that column.



Benchmark	Modified LoC	
	LiLAC	Expert
NPB	0 (44)	1948
Parboil SPMV	0 (44)	261

Figure 6.17: LiLAC vs Expert: We achieve good performance with no application programmer effort (measured as required LoC change). LiLAC code to be written by library developers is in parentheses. Amdahl's Law limits our impact on NPB.

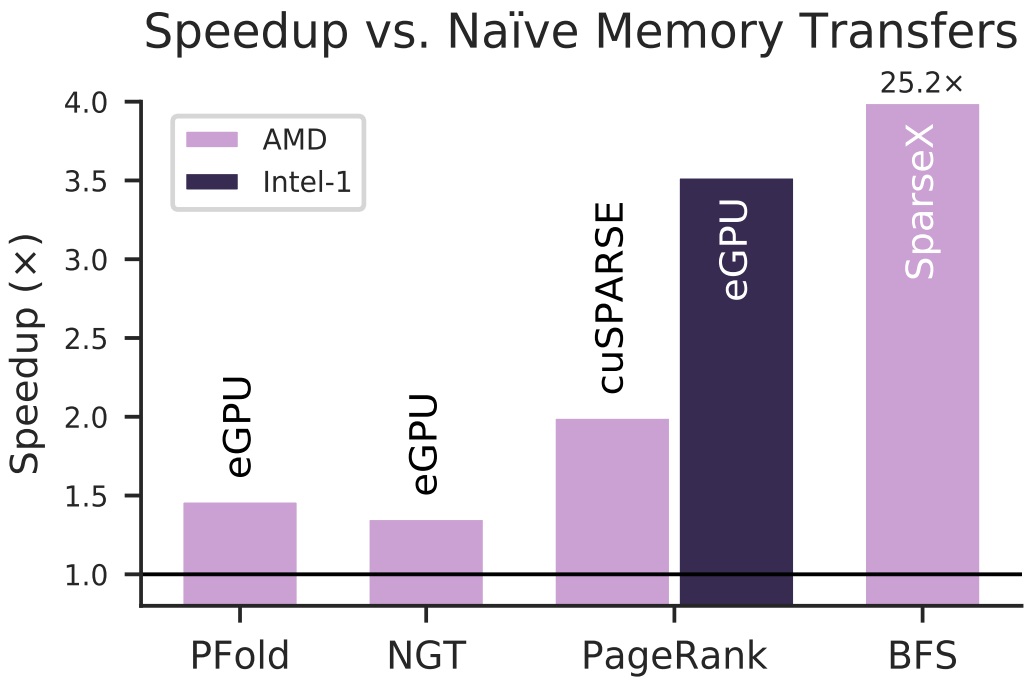


Figure 6.18: LiLAC vs. naïve library calls without harnesses. Avoiding memory transfers is necessary for performance.

Chapter 7

Conclusion

Bibliography

[1] Alexander Aiken. Introduction to set constraint-based program analysis. *Sci. Comput. Program.*, 35(2-3):79–111, November 1999. doi: 10.1016/S0167-6423(99)00007-6. URL http://dx.doi.org/10.1016/S0167-6423(99)00007-6.

[2] Martin S. Alnæs, Anders Logg, Kristian B. Olgaard, Marie E. Rognes, and Garth N. Wells. Unified form language: A domain-specific language for weak formulations of partial differential equations. *ACM Trans. Math. Softw.*, 40(2):9:1–9:37, March 2014. ISSN 0098-3500. doi: 10.1145/2566630. URL http://doi.acm.org/10.1145/2566630.

[3] Martin Alt, Uwe Aßmann, and Hans van Someren. *Cosy compiler phase embedding with the CoSy compiler model*, pages 278–293. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994. ISBN 978-3-540-48371-7. doi: 10.1007/3-540-57877-3_19. URL https://doi.org/10.1007/3-540-57877-3_19.

[4] AMD. clBLAS. https://github.com/clMathLibraries/clBLAS, 2013.

[5] AMD. clSPARSE. https://github.com/clMathLibraries/clSPARSE, 2015.

[6] José M. Andión. *Compilation Techniques for Automatic Extraction of Parallelism and Locality in Heterogeneous Architectures*. PhD thesis, University of A Coruña, 2015.

[7] Uwe Aßmann. *How to uniformly specify program analysis and transformation with graph rewrite systems*, pages 121–135. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. ISBN 978-3-540-49939-8. doi: 10.1007/3-540-61053-7_57. URL https://doi.org/10.1007/3-540-61053-7_57.

[8] Uwe Assmann. Optimix - a tool for rewriting and optimizing programs. In *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools*, pages 307–318. World Scientific, 1998.

[9] Riyadh Baghdadi, Ulysse Beaugnon, Albert Cohen, Tobias Grosser, Michael Kruse, Chandan Reddy, Sven Verdoolaege, Adam Betts, Alastair F. Donaldson, Jeroen Ketema,

Javed Absar, Sven van Haastregt, Alexey Kravets, Anton Lokhmotov, Robert David, and Elnar Hajiyev. Pencil: A platform-neutral compute intermediate language for accelerator programming. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 138–149, Oct 2015. doi: 10.1109/PACT.2015.17.

[10] Riyadh Baghdadi, Albert Cohen, Tobias Grosser, Sven Verdoolaege, Anton Lokhmotov, Javed Absar, Sven Van Haastregt, Alexey Kravets, and Alastair Donaldson. PENCIL Language Specification. Research Report RR-8706, INRIA, May 2015. URL `https://hal.inria.fr/hal-01154812`.

[11] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga. The NAS parallel benchmarks. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing '91, pages 158–165, New York, NY, USA, 1991. ACM. ISBN 0-89791-459-7. doi: 10.1145/125826.125925. URL `http://doi.acm.org/10.1145/125826.125925`.

[12] Muthu Manikandan Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Proceedings of the 19th Joint European Conference on Theory and Practice of Software, International Conference on Compiler Construction*, CC'10/ETAPS'10, pages 244–263, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-11969-7, 978-3-642-11969-9. doi: 10.1007/978-3-642-11970-5_14. URL `http://dx.doi.org/10.1007/978-3-642-11970-5_14`.

[13] D. Baxter, R. Mirchandaney, and J. H. Saltz. Run-time parallelization and scheduling of loops. In *Proceedings of the First Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '89, pages 303–312, New York, NY, USA, 1989. ACM. ISBN 0-89791-323-X. doi: 10.1145/72935.72967. URL `http://doi.acm.org/10.1145/72935.72967`.

[14] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. 08 2015.

[15] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. The polyhedral model is more widely applicable than you think. In *International Conference on Compiler Construction*, pages 283–303. Springer, 2010.

[16] L. Susan Blackford, James Demmel, Jack Dongarra, Iain Duff, Sven Hammarling, Greg Henry, Michael Heroux, Linda Kaufman, Andrew Lumsdaine, Antoine Petitet, Roland Pozo, Karin Remington, and R. Clint Whaley. An updated set of basic linear algebra subprograms (blas). *ACM Trans. Math. Softw.*, 28(2):135–151, June 2002. ISSN 0098-

3500. doi: 10.1145/567806.567807. URL http://doi.acm.org/10.1145/567806.567807.

[17] Kevin J. Brown, Arvind K. Sujeeth, Hyouk Joong Lee, Tiark Rompf, Hassan Chafi, Martin Odersky, and Kunle Olukotun. A heterogeneous parallel framework for domain-specific languages. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques*, PACT '11, pages 89–100, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4566-0. doi: 10.1109/PACT.2011.15. URL http://dx.doi.org/10.1109/PACT.2011.15.

[18] Bryan Catanzaro, Michael Garland, and Kurt Keutzer. Copperhead: Compiling an embedded data parallel language. Technical Report UCB/EECS-2010-124, EECS Department, University of California, Berkeley, Sep 2010. URL http://www2.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-124.html.

[19] Hassan Chafi, Arvind K. Sujeeth, Kevin J. Brown, HyoukJoong Lee, Anand R. Atreya, and Kunle Olukotun. A domain-specific approach to heterogeneous parallelism. *SIGPLAN Not.*, 46(8):35–46, February 2011. ISSN 0362-1340. doi: 10.1145/2038037.1941561. URL http://doi.acm.org/10.1145/2038037.1941561.

[20] Manuel M.T. Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating haskell array codes with multicore GPUs. In *Proceedings of the Sixth Workshop on Declarative Aspects of Multicore Programming*, DAMP '11, pages 3–14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0486-3. doi: 10.1145/1926354.1926358. URL http://doi.acm.org/10.1145/1926354.1926358.

[21] Michael Kruse Chandan Reddy and Albert Cohen. Reduction drawing: Language constructs and polyhedral compilation for reductions on GPUs. In *Proceedings of the 25rd International Conference on Parallel Architectures and Compilation*, PACT '16, 2016.

[22] Lam Chi-Chung, P Sadayappan, and Rephael Wenger. On optimizing a class of multi-dimensional loops with reduction for parallel execution. *Parallel Processing Letters*, 7 (02):157–168, 1997.

[23] Alexander Collins, Dominik Grewe, Vinod Grover, Sean Lee, and Adriana Susnea. NOVA: A functional language for data parallelism. In *Proceedings of ACM SIGPLAN International Workshop on Libraries, Languages, and Compilers for Array Programming*, ARRAY'14, pages 8:8–8:13, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2937-8. doi: 10.1145/2627373.2627375. URL http://doi.acm.org/10.1145/2627373.2627375.

[24] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011. ISSN 0098-3500. doi: 10.1145/2049662.2049663. URL http://doi.acm.org/10.1145/2049662.2049663.

[25] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74. American Mathematical Soc.

[26] Johannes Doerfert, Kevin Streit, Sebastian Hack, and Zino Benaissa. Polly's polyhedral scheduling in the presence of reductions. *CoRR*, abs/1505.07716, 2015. URL http://arxiv.org/abs/1505.07716.

[27] Jack Dongarra, Victor Eijkhout, and Henk van der Vorst. An Iterative Solver Benchmark. *Sci. Program.*, 9(4):223–231, December 2001. ISSN 1058-9244. doi: 10.1155/2001/527931. URL https://doi.org/10.1155/2001/527931.

[28] Jonathan P. K. Doye, Mark A. Miller, and David J. Wales. The double-funnel energy landscape of the 38-atom lennard-jones cluster. *The Journal of Chemical Physics*, 110 (14):6896–6906, 1999. doi: 10.1063/1.478595.

[29] Athena Elafrou, Vasileios Karakasis, Theodoros Gkountouvas, Kornilios Kourtis, Georgios Goumas, and Nectarios Koziris. SparseX: A Library for High-Performance Sparse Matrix-Vector Multiplication on Multicore Platforms. *ACM Trans. Math. Softw.*, 44(3):26:1–26:32, January 2018. ISSN 0098-3500. doi: 10.1145/3134442. URL http://doi.acm.org/10.1145/3134442.

[30] Allan L Fisher and Anwar M Ghuloum. Parallelizing complex scans and reductions. In *ACM SIGPLAN Notices*, volume 29, pages 135–146. ACM, 1994.

[31] Franz Franchetti, Frédéric de Mesmay, Daniel McFarlin, and Markus Püschel. Operator language: A program generation framework for fast kernels. In *IFIP TC 2 Working Conference on Domain-Specific Languages*. Springer, 2009.

[32] Philip Ginsbach and Michael F. P. O'Boyle. Discovery and exploitation of general reductions: a constraint based approach. In *CGO*, pages 269–280. ACM, 2017.

[33] Philip Ginsbach, Lewis Crawford, and Michael F. P. O'Boyle. Candl: A domain specific language for compiler analysis. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 151–162, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5644-2. doi: 10.1145/3178372.3179515. URL http://doi.acm.org/10.1145/3178372.3179515.

[34] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. Automatic matching of legacy code to heterogeneous apis: An idiomatic approach. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 139–153, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173182. URL http://doi.acm.org/10.1145/3173162.3173182.

[35] Tobias Grosser, Armin Größlinger, and Christian Lengauer. Polly - performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters*, 22(4), 2012. URL http://dblp.uni-trier.de/db/journals/ppl/ppl22.html#GrosserGL12.

[36] Sumit Gulwani, Saurabh Srivastava, and Ramarathnam Venkatesan. Program analysis as constraint solving. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 281–292, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375616. URL http://doi.acm.org/10.1145/1375581.1375616.

[37] Gautam Gupta and Sanjay V Rajopadhye. Simplifying reductions. In *POPL*, volume 6, pages 30–41, 2006.

[38] E. Gutiérrez, O. Plata, and E. L. Zapata. A compiler method for the parallel execution of irregular reductions in scalable shared memory multiprocessors. In *Proceedings of the 14th International Conference on Supercomputing*, ICS '00, pages 78–87, New York, NY, USA, 2000. ACM. ISBN 1-58113-270-0. doi: 10.1145/335231.335239. URL http://doi.acm.org/10.1145/335231.335239.

[39] Eladio Gutiérrez, O Plata, and Emilio L Zapata. Optimization techniques for parallel irregular reductions. *Journal of systems architecture*, 49(3):63–74, 2003.

[40] Eladio Gutiérrez, Oscar Plata, and Emilio L Zapata. An analytical model of locality-based parallel irregular reductions. *Parallel Computing*, 34(3):133–157, 2008.

[41] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with Lift. In *CGO*. ACM, 2018.

[42] *Math Kernel Library*. Intel, 2003. URL https://software.intel.com/en-us/intel-mkl.

[43] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic cpu-gpu communication management and optimization.

SIGPLAN Not., 46(6):142–151, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993516. URL http://doi.acm.org/10.1145/1993316.1993516.

[44] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. Automatic CPU-GPU communication management and optimization. In *PLDI*, 2011.

[45] Richard Johnson, David Pearson, and Keshav Pingali. The program structure tree: Computing control regions in linear time. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, PLDI '94, pages 171–185, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: 10.1145/178243.178258. URL http://doi.acm.org/10.1145/178243.178258.

[46] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*, pages 1–12, 2017. doi: 10.1145/3079856.3080246. URL http://doi.acm.org/10.1145/3079856.3080246.

[47] Pierre Jouvelot and Babak Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Proceedings of the 3rd international conference on Supercomputing*, pages 186–194. ACM, 1989.

[48] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 711–726,

New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908117. URL http://doi.acm.org/10.1145/2908080.2908117.

[49] Jeremy Kepner, David A. Bader, Aydin Buluç, John R. Gilbert, Timothy G. Mattson, and Henning Meyerhenke. Graphs, matrices, and the graphblas: Seven good reasons. In *ICCS*, 2015.

[50] Vladimir Kiriansky, Yunming Zhang, and Saman Amarasinghe. Optimizing indirect memory references with milk. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, PACT '16, pages 299–312, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4121-9. doi: 10.1145/2967938.2967948. URL http://doi.acm.org/10.1145/2967938.2967948.

[51] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. *SIGPLAN Not.*, 44(6):327–337, June 2009. ISSN 0362-1340. doi: 10.1145/1543135.1542513. URL http://doi.acm.org/10.1145/1543135.1542513.

[52] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[53] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. Openmp to gpgpu: A compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101–110, February 2009. ISSN 0362-1340. doi: 10.1145/1594835.1504194. URL http://doi.acm.org/10.1145/1594835.1504194.

[54] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101–110, February 2009. ISSN 0362-1340. doi: 10.1145/1594835.1504194. URL http://doi.acm.org/10.1145/1594835.1504194.

[55] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *Proceedings of the 32Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 364–377, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040335. URL http://doi.acm.org/10.1145/1040305.1040335.

[56] Peter Lipps, Ulrich Möncke, and Reinhard Wilhelm. *OPTRAN - A language/system for the specification of program transformations: System overview and experiences*,

pages 52–65. Springer Berlin Heidelberg, Berlin, Heidelberg, 1989. ISBN 978-3-540-46200-2. doi: 10.1007/3-540-51364-7_4. URL https://doi.org/10.1007/3-540-51364-7_4.

[57] Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 22–32, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737965. URL http://doi.acm.org/10.1145/2737924.2737965.

[58] Florian Martin. Pag – an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, Nov 1998. ISSN 1433-2779. doi: 10.1007/s100090050017. URL https://doi.org/10.1007/s100090050017.

[59] Trevor L. McDonell, Manuel M.T. Chakravarty, Gabriele Keller, and Ben Lippmeier. Optimising purely functional GPU programs. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 49–60, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2326-0. doi: 10.1145/2500365.2500595. URL http://doi.acm.org/10.1145/2500365.2500595.

[60] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: Lifting high-performance stencil kernels from stripped x86 binaries to Halide DSL code. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '15, pages 391–402, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737974. URL http://doi.acm.org/10.1145/2737924.2737974.

[61] David Menendez and Santosh Nagarakatte. Alive-infer: Data-driven precondition inference for peephole optimizations in llvm. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 49–63, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062372. URL http://doi.acm.org/10.1145/3062341.3062372.

[62] David Menendez, Santosh Nagarakatte, and Aarti Gupta. *Alive-FP: Automated Verification of Floating Point Based Peephole Optimizations in LLVM*, pages 317–337. Springer Berlin Heidelberg, Berlin, Heidelberg, 2016. ISBN 978-3-662-53413-7. doi: 10.1007/978-3-662-53413-7_16. URL https://doi.org/10.1007/978-3-662-53413-7_16.

[63] Ana Moreton-Fernandez, Arturo Gonzalez-Escribano, and Diego Ferraris. Multi-device controllers: A library to simplify parallel heterogeneous programming. 12 2017.

[64] Ana Moreton-Fernandez, Eduardo Rodriguez-Gutiez, Arturo Gonzalez-Escribano, and Diego R. Llanos. Supporting the xeon phi coprocessor in a heterogeneous programming model. In Francisco F. Rivera, Tomás F. Pena, and José C. Cabaleiro, editors, *Euro-Par 2017: Parallel Processing*, pages 457–469, Cham, 2017. Springer International Publishing. ISBN 978-3-319-64203-1.

[65] Ravi Teja Mullapudi, Vinay Vasista, and Uday Bondhugula. PolyMage: Automatic optimization for image processing pipelines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 429–443, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-2835-7. doi: 10.1145/2694344.2694364. URL http://doi.acm.org/10.1145/2694344.2694364.

[66] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Trans. Graph.*, 35(4):83:1–83:11, July 2016. ISSN 0730-0301. doi: 10.1145/2897824.2925952. URL http://doi.acm.org/10.1145/2897824.2925952.

[67] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for compcert. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '16, pages 448–461, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4261-2. doi: 10.1145/2908080.2908109. URL http://doi.acm.org/10.1145/2908080.2908109.

[68] Saurav Muralidharan, Amit Roy, Mary W. Hall, Michael Garland, and Piyush Rai. Architecture-adaptive code variant tuning. In *ASPLOS*, pages 325–338. ACM, 2016.

[69] Andres Nötzli and Fraser Brown. LifeJacket: Verifying precise floating-point optimizations in llvm. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2016, pages 24–29, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4385-5. doi: 10.1145/2931021.2931024. URL http://doi.acm.org/10.1145/2931021.2931024.

[70] *cuBLAS*. Nvidia. URL http://developer.nvidia.com/cuBLAS.

[71] NVIDIA. NVIDIA CUDA Sparse Matrix library (cuSPARSE). https://developer.nvidia.com/cusparse.

[72] *Nvidia OpenCL Best Practices Guide*. Nvidia, 2011.

[73] Georg Ofenbeck, Tiark Rompf, Alen Stojanov, Martin Odersky, and Markus Püschel. SPIRAL in Scala: Towards the systematic construction of generators for performance libraries. In *International Conference on Generative Programming: Concepts & Experiences*, GPCE, 2013.

[74] Karina Olmos and Eelco Visser. Composing source-to-source data-flow transformations with rewriting strategies and dependent dynamic rewrite rules. In *Proceedings of the 14th International Conference on Compiler Construction*, CC'05, pages 204–220, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-25411-0, 978-3-540-25411-9. doi: 10.1007/978-3-540-31985-6_14. URL http://dx.doi.org/10.1007/978-3-540-31985-6_14.

[75] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman P. Amarasinghe. Portable performance on heterogeneous architectures. In *ASPLOS*, pages 431–444. ACM, 2013.

[76] Shlomit S. Pinter and Ron Y. Pinter. Program optimization and parallelization using idioms. *ACM Trans. Program. Lang. Syst.*, 16(3):305–327, May 1994. ISSN 0164-0925. doi: 10.1145/177492.177494. URL http://doi.acm.org/10.1145/177492.177494.

[77] Shlomit S Pinter and Ron Y Pinter. Program optimization and parallelization using idioms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 16(3): 305–327, 1994.

[78] Bill Pottenger and Rudolf Eigenmann. Idiom recognition in the polaris parallelizing compiler. In *Proceedings of the 9th international conference on Supercomputing*, pages 444–448. ACM, 1995.

[79] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman 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*, PLDI '13, pages 519–530, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462176. URL http://doi.acm.org/10.1145/2491956.2462176.

[80] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *SIGPLAN Not.*, 48(6):519–530, June 2013. ISSN 0362-1340. doi: 10.1145/2499370.2462176. URL http://doi.acm.org/10.1145/2499370.2462176.

[81] Lawrence Rauchwerger and David A Padua. The LRPD test: Speculative run-time parallelization of loops with privatization and reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 10(2):160–180, 1999.

[82] Vignesh T Ravi, Wenjing Ma, David Chiu, and Gagan Agrawal. Compiler and runtime support for enabling generalized reduction computations on heterogeneous parallel configurations. In *Proceedings of the 24th ACM international conference on supercomputing*, pages 137–146. ACM, 2010.

[83] Xavier Redon and Paul Feautrier. Scheduling reductions. In *Proceedings of the 8th international conference on Supercomputing*, pages 117–125. ACM, 1994.

[84] Tiark Rompf and Martin Odersky. Lightweight modular staging: A pragmatic approach to runtime code generation and compiled DSLs. *Commun. ACM*, 55(6), 2012.

[85] Y. Saad. Krylov subspace methods on supercomputers. *SIAM Journal on Scientific and Statistical Computing*, 10(6):1200–1232, 1989. doi: 10.1137/0910073. URL `https://doi.org/10.1137/0910073`.

[86] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition, 2003. doi: 10.1137/1.9780898718003. URL `https://epubs.siam.org/doi/abs/10.1137/1.9780898718003`.

[87] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. *SIGPLAN Not.*, 45(1):313–326, January 2010. ISSN 0362-1340. doi: 10.1145/1707801.1706337. URL `http://doi.acm.org/10.1145/1707801.1706337`.

[88] Michel Steuwer, Christian Fensch, Sam Lindley, and Christophe Dubach. Generating performance portable code using rewrite rules: From high-level functional expressions to high-performance OpenCL code. *SIGPLAN Not.*, 50(9):205–217, August 2015. ISSN 0362-1340. doi: 10.1145/2858949.2784754. URL `http://doi.acm.org/10.1145/2858949.2784754`.

[89] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: a functional data-parallel IR for high-performance GPU code generation. In *CGO*, pages 74–85. ACM, 2017.

[90] Kevin Stock, Martin Kong, Tobias Grosser, Louis-Noël Pouchet, Fabrice Rastello, J. Ramanujam, and P. Sadayappan. A framework for enhancing data reuse via associative reordering. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 65–76, New York, NY, USA,

2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594342. URL http://doi.acm.org/10.1145/2594291.2594342.

[91] John Stratton, Christopher Rodrigues, I-Jui Sung, Nady Obeid, Li-Wen Chang, Nasser Anssari, Daniel Geng, Wen-Mei Liu, and Wen-mei Hwu. Parboil: A Revised Benchmark Suite for Scientific and Commercial Throughput Computing. November 2018.

[92] Toshio Suganuma, Hideaki Komatsu, and Toshio Nakatani. Detection and global optimization of reduction operations for distributed parallel machines. In *Proceedings of the 10th international conference on Supercomputing*, pages 18–25. ACM, 1996.

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

[94] Patricia Suriana, Andrew Adams, and Shoaib Kamil. Parallel associative reductions in halide. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, pages 281–291, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5090-4931-8. URL http://dl.acm.org/citation.cfm?id=3049832.3049863.

[95] Kyle H. Sutherland-Cash, Rosemary G. Mantell, and David J. Wales. Exploiting sparsity in free energy basin-hopping. *Chemical Physics Letters*, 685:288 – 293, 2017. ISSN 0009-2614. doi: https://doi.org/10.1016/j.cplett.2017.07.081. URL http://www.sciencedirect.com/science/article/pii/S0009261417307583.

[96] Ross Tate, Michael Stepp, and Sorin Lerner. Generating compiler optimizations from proofs. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 389–402, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-479-9. doi: 10.1145/1706299.1706345. URL http://doi.acm.org/10.1145/1706299.1706345.

[97] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, José Ignacio Gómez, Christian Tenllado, and Francky Catthoor. Polyhedral parallel code generation for cuda. *ACM Trans. Archit. Code Optim.*, 9(4):54:1–54:23, January 2013. ISSN 1544-3566. doi: 10.1145/2400682.2400713. URL http://doi.acm.org/10.1145/2400682.2400713.

[98] David J. Wales. Discrete path sampling. *Molecular Physics*, 100(20):3285–3305, 2002. doi: 10.1080/00268970210162691.

[99] Linnan Wang, Wei Wu, Zenglin Xu, Jianxiong Xiao, and Yi Yang. Blasx: A high performance level-3 blas library for heterogeneous multi-gpu computing. In

Proceedings of the 2016 International Conference on Supercomputing, ICS '16, pages 20:1–20:11, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4361-9. doi: 10.1145/2925426.2926256. URL http://doi.acm.org/10.1145/2925426.2926256.

[100] Deborah L. Whitfield and Mary Lou Soffa. An approach for exploring code improving transformations. *ACM Trans. Program. Lang. Syst.*, 19(6):1053–1084, November 1997. ISSN 0164-0925. doi: 10.1145/267959.267960. URL http://doi.acm.org/10.1145/267959.267960.

[101] Jeremiah James Willcock, Andrew Lumsdaine, and Daniel J. Quinlan. Reusable, generic program analyses and transformations. In *Proceedings of the Eighth International Conference on Generative Programming and Component Engineering*, GPCE '09, pages 5–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-494-2. doi: 10.1145/1621607.1621611. URL http://doi.acm.org/10.1145/1621607.1621611.

[102] Huo X., Ravi V., and Agrawal G. Porting irregular reductions on heterogeneous CPU-GPU configurations. In *Proceedings of the 18th IEEE International Conference on High Performance Computing*, 2011.

[103] Vanya Yaneva, Ajitha Rajan, and Christophe Dubach. *Compiler-Assisted Test Acceleration on GPUs for Embedded Software*, pages 35–45. ACM, 7 2017. ISBN 978-1-4503-5076-1. doi: 10.1145/3092703.3092720.

[104] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. *SIGPLAN Not.*, 46(6):283–294, June 2011. ISSN 0362-1340. doi: 10.1145/1993316.1993532. URL http://doi.acm.org/10.1145/1993316.1993532.

[105] Hao Yu and Lawrence Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 17 (10):1084–1096, 2006.

[106] Eric Yuan. Towards ontology-based software architecture representations. In *Proceedings of the 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering*, ECASE '17, pages 21–27, Piscataway, NJ, USA, 2017. IEEE Press. ISBN 978-1-5386-0417-5. doi: 10.1109/ECASE.2017..5. URL https://doi.org/10.1109/ECASE.2017..5.

[107] Jianzhou Zhao, Santosh Nagarakatte, Milo M. K. Martin, and Steve Zdancewic. Formalizing the llvm intermediate representation for verified program transformations. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 427–440, New York, NY, USA, 2012.

ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103709. URL http://doi.acm.org/10.1145/2103656.2103709.

[108] Jie Zhao, Michael Kruse, and Albert Cohen. A polyhedral compilation framework for loops with dynamic data-dependent bounds. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 14–24, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5644-2. doi: 10.1145/3178372.3179509. URL http://doi.acm.org/10.1145/3178372.3179509.