# Generation of Compiler Backends from Formal Models of Hardware

Gus Henry Smith

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington

2024

Reading Commitee:

Zachary Tatlock, Chair

Luis Ceze

Michael Taylor

Program Authorized to Offer Degree:
Computer Science & Engineering

University of Washington

**Abstract**

Generation of Compiler Backends from Formal Models of Hardware

Gus Henry Smith

Chair of the Supervisory Committee:
Zachary Tatlock
Computer Science & Engineering

Compilers convert between representations—usually, from higher-level, human writable code to lower-level, machine-readable code. A compiler backend is the portion of the compiler containing optimizations and code generation routines for a specific hardware target. In this dissertation, I advocate for a specific way of building compiler backends: namely, by automatically generating them from explicit, formal models of hardware using automated reasoning algorithms. I describe how automatically generating compilers from formal models of hardware leads to increased optimization ability, stronger correctness guarantees, and reduced development time for compiler backends. As evidence, I present two case studies: first, Glenside, which uses equality saturation to increase the 3LA compiler's ability to offload operations to machine learning accelerators, and second, Lakeroad, a technology mapper for FPGAs which uses program synthesis and semantics extracted from Verilog to map hardware designs to complex, programmable hardware primitives.

# Contents

# Acknowledgments

First, allow me to thank everyone in my life *not* mentioned in the following paragraphs. My PhD was far more than just a collection of projects; for the past six years, it was my life. Anyone who has contributed to my life has contributed to this dissertation.

Thanks to my early research mentors. To Vijay Narayanan and Jack Sampson at Penn State. Without their early mentorship, I would have never considered the PhD a viable option. To my initial research mentors in SAMPL: Thierry Moreau, Tianqi Chen, Jared Roesch, and Luis Vega. Thank you for bringing me on to TVM and introducing me to compilers research. To my advisor Luis Ceze, who hired me on at UW and changed the course of my entire life. I could never thank Luis enough for the impact that single decision had on me.

Thanks to everyone I've worked with in industry. To Aart Bik, Mangpo Phothilimthana, and Penporn Koanantakool at Google for their mentorship throughout my extended 2021 internship. To Aart and Penporn for bringing me onto the MLIR team, and to Mangpo for her patient tutelage in applying machine learning to compilers. To Noah Evans and everyone at Sandia, for bringing me on and being interested in my research. To Nina, Jannis, Claire, and the entire YosysHQ team for their support. To Jin Yang and Jeremy Casas at Intel.

Thanks to every research team I've worked with. To the Glenside team: Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Zach Tatlock, and Luis Ceze. To the 3LA team: Bo-Yuan Huang, Steven, Yi Li, Mike He, Thierry Tambe, Akash Gaonkar, Vishal

Canumalla, Andrew Cheung, Gu-Yeon Wei, Aarti Gupta, Zach, and Sharad Malik. To the Lakeroad team: Ben Kushigian, Vishal, Andrew, Steven, Sorawee Porncharoenwase, René Just, Gilbert Bernstein, and Zach. To the Churchroad team: Dan Petrisko, Colin Knizek, Chandra Nandi, Zach, Jonathan Balkind, Zach Sisco and Thanawat Techaumnuaiwit.

Thanks to those who volunteered to be on my committee—to Mike Taylor, Scott Hauck, and Gilbert Bernstein. Thanks especially to Gilbert, who invested time and effort into making me a sharper, more precise PL researcher in my last year at UW. Gilbert was brave enough to attempt to teach me about coinduction—and persistent enough to (mostly) succeed. I will always appreciate his willingness to take the time and effort required to explain difficult concepts.

Thanks to the MISL lab for letting me hang around all these years, despite my utter lack of any formal training in biology. Thanks to Chris Thachuk and Lancelot Wathieu for taking the Fridge Compiler from half-working course project to published paper. Thanks to Jeff Nivala, Gwen Roote, Carina Imburgia, and the whole FPCA team for letting me tag along on the project. A more down-to-earth lab I may never find.

To the entire Cornell Capra lab. To Rachit Nigam, who served as my model of success in programming languages and hardware design research. To Adrian Sampson, who I can always look to for advice. To Priya Srikumar, whose brief presence in PLSE in the summer of 2023 was a blessing. Priya's passing was felt throughout the Programming Languages research world, into their communities in Ithaca and Seattle, and beyond. I will always remember Priya sharing a Tom Wayman poem with me, titled "Did I Miss Anything?", an excerpt of which will always remind me of them:

> Nothing. When you are not present
> how could something significant occur?

To the PLSE lab. It is harder to imagine a better lab exists anywhere on the planet. The energy of the lab can be felt even before stepping through the door. Whether it's laughter, or spirited research discussion, or someone playing one of the lab's many instruments, or even just the peaceful silence of face-in-laptop research progress, the energy of the lab is one of a kind. Without realizing it, the lab has become my family; leaving it has been one of the most challenging things I've ever done. I can only hope to start something half as vibrant, half as connected, half as supportive, wherever I end up.

Thanks to my mentee Hannah Leung, whose unyielding positivity and bravery in the face of challenges the likes of which I'll never myself experience was a constant source of inspiration. Though she may not know it, I've learned more from her than she has from me.

To my mentees Vishal Canumalla and Andrew Cheung. Thanks for trusting me to mentor you in research and in life, and thanks for your friendship along the way. I am so proud of you guys, and cannot wait to see what you do next. Furthermore, thanks to Vishal's family, the Canumallas, who brought me in. Thank you Anu and Sridhar for making me feel at home and making sure I'm well fed.

Thanks to Race Condition Running for the miles and miles of fun over the years. Through my best and worst times in the PhD and in my personal life, the Saturday run and brunch were there for me as a consistent source of companionship and connection. Thanks to Zach, Nick Walker, Ellis, Chandra, Max, and everyone else who kept the club going. Thanks to everyone else who kept me active during my PhD—thanks to Eric Zeng and Chien-Yu Lin for getting me out in the backcountry when the snow was good and to Gwen Roote for keeping me humble on the BJJ mats.

Thanks to the lab jam crew—Ben, Carina, Rohan, and everyone else—for providing a safe space to be a musical beginner.

To Ben Kushigian—a wonderful coauthor and friend.

To Nick and Danielle. To Holly and Peter. To Jared.

To Anjali, for all of the time you've given me.

To `#binary-blobs`, `#spicelords`, and associated friends: Pratyush Patel, Chien-Yu Lin, Jacob Van Geffen, Dan Petrisko, Luzdary Ruelas, Ellis Michael, Max Ruttenberg, and Katie Lim. Thanks to the countless hours of Twilight Imperium.

To Max Willsey and Sami Davies. Without Max's urging, I may have never joined UW; even after I joined UW, without Max's urging, I may have never joined PLSE. Max's quiet, confident intelligence served as a model for the entire PLSE lab. I was lucky to witness a golden age of PLSE research, with Max's `egg` at the forefront. Max showed me that a new project, an empty buffer, a clean slate, was nothing to be scared of if you face it with confidence and excitement. And to Sami, whose intimidating intelligence is softened by her infectious smile and sincere desire for connection.

To Steven Lyubomirsky—*Lex Lyubomiricus* will forever guide my research. Thank you for your constant mentorship at every stage of my research career. Steven helped me present my first Programming Languages Reading Group paper in 2018—"Build Systems à la Carte" by Mokhov et al.—before I had any formal Programming Languages research under my belt. Six years later, in 2024, Steven helped me publish the final first-author publication of my PhD, the Lakeroad paper.

To Chandra Nandi—I could not have asked for a better research mentor and friend. Chandra helped in every single one of my paper pushes in grad school, simply out of the love of camaraderie. She taught me how to feel the unabashed joy that comes from working hard on cool problems with your closest friends.

To Jeff's sons, Christian, Nick, and Aaron. You all are so much like your father. Thanks for bucking the Seattle Freeze and making me feel at home here in my first few years. Here's to many, many more rounds of pitch and putt.

To Eric, Rick, Zach, and the other men in the Montlake Men's Group. Your support and mentorship helped me navigate the most challenging chapters of my life to date.

To my friends from the previous chapters of my life that I've been lucky enough to bring into this one: Spencer Norris, Imaz Athar, Anthony Marucci, Tim Lagnese, Jacob Brunette, and Drew Abbott. Thank you to my oldest and best friends, Spencer and Imaz. There is something bittersweet about how, more than almost anything else, it's simply *time* that is the key ingredient for a deep relationship—I'm so lucky to have invested my time in you guys. You two are and will always be my brothers. It has been an honor and a joy to grow with you through life's changes, to have you guys help me through my lowest lows and celebrate with me during my highest highs. HSNE, boys. Thanks to Anthony, Tim, and Jacob. I could not have picked a better crew of friends to move to Seattle with, to spend my 20s with. May our caps always fly true. To Drew and Em—we've rarely lived in the same city, but you two are family to me.

Boundless love and thanks to my older brother Simon. In all my 29 years, he has never been far from my mind.

Thank you to Michaela, whose five years of love and loyalty transformed me into the man I am. Her support carried me through the single continuous paper push that was the two years of my PhD. Her patience with the all-consuming nature of my PhD was more than I could have asked for. Her genuine care for everyone around her showed me that there is nothing more to life than the people in it. There is no one more deserving of happiness. Furthermore, thanks to the entire Brock and Lowe family, who brought me in without question, who made me a part of their family. To Lisa Lowe, whose unhesitating love from the moment we met taught me the meaning of "unconditional". To Ray Brock, who was a role model for men in a world where they are severely lacking. I am immensely lucky to have known him. It is through the Lowes that I came to understand the meaning of family—a gift I could never repay.

Finally, to my advisor, Zach Tatlock. Zach is the kind of researcher all should strive to be. Intelligent, hardworking, and selfless to a fault. Zach brought me on when he didn't need to. He has a steady supply of top-tier PL students. Bringing me on, as a student with zero formal

programming languages research experience, was in many ways a questionable decision. But, like Luis's decision to admit me, that single decision altered the course of my entire life. Everything I know of research I learned first from Zach. But Zach is far more than just a good researcher. Zach is, in fact, the only thing I really strive to be—a good man. I've had very few role models in life, and Zach is first among them. Thank you, Zach—none of this could have happened without you.

# Glossary

**accelerator** Custom hardware, specialized to a specific task or set of tasks. In this dissertation, we are most frequently referring to accelerators for machine learning kernels. 11, 14, 15, 31, 33, 81

**ASIC** A hardware chip produced for a specific purpose. In contrast to FPGAs, which are reprogrammable, ASICs are static. As a consequence, the design of ASICs is an intensive process involving significant validation and verification to ensure hardware correctness. 10

**automated reasoning** A term to capture algorithms such as SMT or equality saturation. In general, automated reasoning algorithms are able to solve complex constraint or optimization problems by applying mathematical and logical rules. Automated reasoning algorithms are generally applicable to many tasks in computer science, as long as the can be captured in a way the algorithm can understand. 12, 19, 25, 27, 133, 135

**compiler** A tool which converts between representations. 9, 12, 13

**compiler backend** The portion of a compiler that concerns the target, e.g., target-specific optimizations and code generation. 13

**DSL** A programming language designed for a narrow domain, generally for a specific purpose. DSLs are generally smaller than general-purpose programming languages, with fewer

complicated features like general control flow. This makes them easier to reason about and optimize. 14, 16, 24

**equality saturation** A term rewriting algorithm [174, 175, 195] which uses a specific data structure—the egraph—to capture a large space of equivalent programs. 9, 20, 27, 31, 37, 40, 49, 81, 127

**FPGA** Field Programmable Gate Arrays are a reprogrammable hardware platform allowing users to design hardware without fabricating an ASIC. FPGAs are pre-fabricated chips composed of programmable primitives. The FPGA can be reprogrammed to configure and connect the various primitives, producing different hardware designs. 14, 15, 19, 27, 71, 85, 131

**hardware synthesis** Another term for hardware compilation. The process of converting a hardware design captured in a high-level language to a low-level target implementation, for example, a *netlist* which can be programmed onto an FPGA or a geometry file to be made into an ASIC. 12, 14, 19, 85, 134, 135

**HLS** A set of tools [43] for compiling high-level implementations of algorithms (often written in C) into hardware designs. HLS is often contrasted against RTL, written in languages like Verilog or VHDL, which is a comparatively lower level of abstraction. 11, 26, 134

**instruction selection** A core compiler algorithm in which higher-level, hardware-independent operations are lowered to hardware-specific instructions [20]. 14

**machine learning kernel** A core subroutine used within many machine learning workloads—for example, matrix multiplication. Machine learning kernels are often highly optimized, often by building custom hardware to implement them efficiently. 9, 14, 51

**netlist** Much like RTL or HLS, netlists are a way to capture a hardware specification. Netlists are lower-level than RTL or HLS, however. As their name implies, netlists are simply lists of "nets" (or wires), plus, importantly, the modules they are connected to. 10, 11

**primitive** The atomic units of a language or representation. In the context of this dissertation, we often mean either accelerator primitives or hardware primitives. Accelerator primitives are operations implemented by accelerators which are, from the view of software, black box operations which cannot be split into smaller sets of instructions. They represent the smallest unit a compiler can target. Hardware primitives are the modules provided for use by a hardware platform—e.g. the gates and devices available for use on an FPGA. A netlist is composed of primitive instantiations. 10, 27, 85, 136

**program synthesis** A broad term capturing a set of techniques for generating programs, often using automated reasoning tools [71]. In this dissertation, we are generally referring to solver-aided synthesis, which formulates program generation as a constraint solving problem and applies solvers (e.g. SMT solvers). Specifically, we are generally referring to *sketch-guided* program synthesis, in which the final compiled program is captured as a sketch, with holes to be filled in by the solver [166]. 20, 27, 85, 99, 131, 136

**RTL** A specific level of abstraction for a hardware design specification, in which registers (hardware modules which hold state) are explicit in the design, but computation is still modeled at a high level. This is in contrast to a higher level representation like HLS, which does not include timing, or a lower, gate-level or netlist representation which may use specific gates or other hardware primitives. 10, 11, 134

**SMT** A class of constraint problem. SMT is simply SAT—Boolean satisfiability—with extra "theories" added in, somewhat like libraries. For example, the theory of fixed-width bitvectors,

which includes operations over groups of bits. SMT solvers are automated reasoning algorithms which solve SMT problems. 9, 11, 25, 99, 135

**target** The platform which a compiler generates code for. 9, 13

**technology mapping** A core compiler algorithm in hardware synthesis tools in which hardware-independent components of a design (e.g. a Verilog multiply operator *) are lowered to hardware-specific primitive instantiations (e.g. a Xilinx DSP48E2 DSP). 15, 19, 85, 131

**tensorization** An algorithm within machine learning compilers which uncovers places to invoke primitives operating over tensors—often in the form of accelerator invocations [182]. Tensorization is analogous to vectorization in standard compilers, in which groups of instructions are translated into a single vector instruction to improve performance. 14, 19, 81

**validation** The process of sanity-checking a program or hardware design using a limited, finite set of inputs, and judging the correctness of the outputs. Also referred to as *testing*. Validation should specifically be distinguished from verification. Confusingly, in the world of hardware design, validation is often referred to as verification, while verification (by our definition) is referred to as *formal* verification. 9, 12, 34, 74

**verification** The process of mathematically proving correctness about a program or hardware design. While the proofs of correctness themselves can have limitations, verification is generally more thorough than validation or *testing*. In the world of hardware design, "verification" generally refers to what we call validation or testing, while "formal verification" refers to our verification. 9, 12, 135

# Chapter 1

# Introduction

A *compiler* is a tool which converts from one representation to another—usually, from a higher-level, human-writeable representation to a lower-level, machine-readable representation. A classic example is the clang compiler from the LLVM suite [104], which compiles programs written in processor-independent C code into processor-specific machine code, which the hardware understands how to execute:

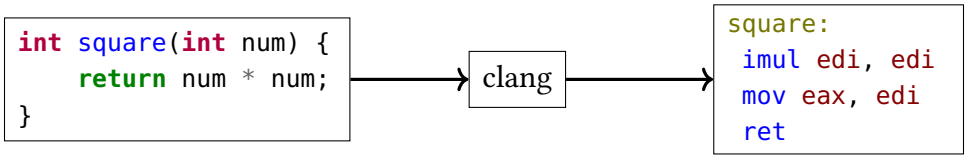



Figure 1.1: clang compiling high-level C code to target-specific x86 assembly.

In this case, clang is generating code for an x86 processor; we refer to the platform which the compiler is compiling for as the *target*. Though compilers perform many target-agnostic transformations and optimizations (modifications of the high-level code which are useful regardless of the target), a compiler's fundamental purpose is to produce a program in the target's language. The portion of the compiler which handles target-specific optimizations and code generation is called the *compiler backend*—for example, it is clang's x86 backend which is responsible for

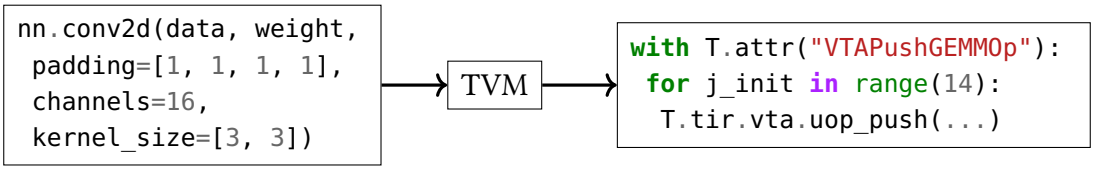



Figure 1.2: Tensorizing a 2D convolution to VTA [121] accelerator calls.

performing x86-specific optimizations and eventually producing x86 assembly code. Compiler backends will be the focus of this dissertation; we will largely ignore other compiler components (i.e. frontends and target-independent optimizations).

Throughout this intro, we will use three different compilers as our running examples. As we have already seen, we will consider the general-purpose C compiler clang as our "gold standard" example of a commonly-known and understood compiler. However, we will also consider two compilers from more specialized domains, which will be key to part I and part II of this dissertation, respectively. First, the TVM compiler [35], which will be a main focus of part I, is a compiler for deep learning programs which compiles code written in a high-level Python *domain-specific language (DSL)* into optimized code for CPUs, GPUs, and custom *accelerators*. Second, the open-source *hardware synthesis* tool Yosys is a compiler for hardware designs supporting hardware targets such as *FPGAs* and will be a focus of part II.

Compiler backends are composed of multiple stages, and each stage is implemented with one or a number of core **algorithms.** For example, a key stage in clang's x86 backend when compiling our example in fig. 1.1 is *instruction selection*, in which clang decides how to implement each operation in the C program using actual instructions provided by the processor [110]. It is in this stage where clang decides to implement C's * operator using x86's `imul` instruction.

Our other two compiler examples, TVM and Yosys, also rely on a few core algorithms to implement their backends. TVM implements a step called *tensorization* [182], which, among other things, maps high-level *machine learning kernels* to target-specific implementations, including

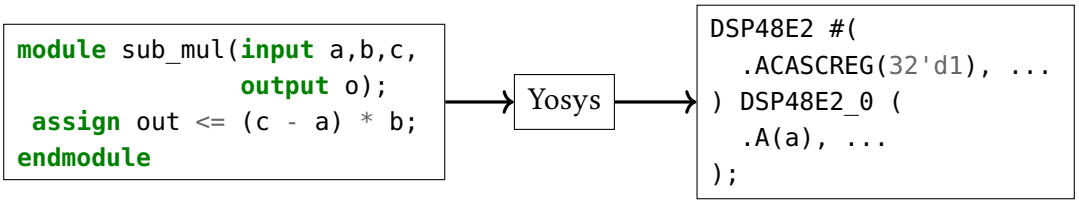



Figure 1.3: Technology mapping a high-level hardware design to an instantiation of a specific hardware primitive.

invocations of specialized hardware accelerators. Figure 1.2 shows an example of tensorizing a 2-dimensional convolution to general matrix multiplication (GEMM) instructions for a specific accelerator backend, VTA. Similarly, a core step in hardware compilation for Yosys and other hardware synthesis tools is *technology mapping* in which the tool determines how to implement the high-level hardware design using the hardware primitives available on the hardware platform. An example of technology mapping is shown in fig. 1.3, in which a high-level, architecture-independent hardware module is implemented using an architecture-specific hardware primitive (in this case, a DSP48E2 primitive present on Xilinx FPGAs). Both tensorization and technology mapping will be key focuses of this dissertation, in parts I and II respectively.

To implement their core algorithms, compiler backends employ **models** of hardware. Following our running examples, clang's instruction selection algorithm directly utilizes a model of the x86 architecture's instructions to determine what instructions are available for use. The model is explicit, built into clang's x86 backend itself. The following snippet is taken from the x86 instruction model, and is the declaration of the `imul` instruction used to implement our `square` function above:

```
defm IMUL : Mul<0xF7, "imul", MRM5r, MRM5m, null_frag>;
```

Figure 1.4: Declaration of x86's `imul` instruction in LLVM's x86 backend [111].

This instruction declaration tells clang's instruction selector that there is an instruction, `imul`,

available for use; other parts of the model (omitted) describe the functionality of the instruction, which helps the instruction selector decide when to use the `imul` instruction.

Not all models within compilers are made the same, however. To contrast the explicit model in fig. 1.4, consider this snippet from Yosys's technology mapper for Xilinx FPGAs:

```
subpattern in_dffe
arg argQ clock
code
  dff = nullptr;
  if (argQ.empty())
    reject;
  for (const auto &c : argQ.chunks()) {
    if (!c.wire)
      reject;
    ...
```

Figure 1.5: Snippet of code from Yosys's pmgen framework [206] attempting to map hardware designs to specific FPGA hardware primitives.

This code is an imperative pattern matching algorithm written in Yosys's pmgen DSL which searches for a specific pattern in the hardware design. Unlike fig. 1.4, which is an explicit hardware model *used by* clang's instruction selector algorithm, the above example is *both* algorithm *and* model: encoded implicitly within this algorithm is a model of the underlying hardware.[1] Soon, I will argue why this method of entwining algorithm and model is disadvantageous; before I do that, however, I will introduce some terminology to make it easier to discuss the properties of algorithms and models.

To better discuss compiler backends' algorithms and the hardware models on which they depend, I introduce two terms: *model explicitness* and *algorithm adaptability.*

[1] In fact, it would be quite difficult to build a compiler *without* encoding some kind of model of the underlying hardware. That is, any compiler which generates code for a hardware target will encode facts about the hardware which amount to a model of the hardware. The less those facts are explicitly separated out, the more implicit the model.

**Model explicitness.** Model explicitness captures how overtly a model is encoded into a compiler backend. For example, fig. 1.4 presents an overt, explicit model of hardware in the form of a list of instructions implemented on x86. In contrast, fig. 1.5 presents an implicit model embedded within a pattern matching algorithm. We consider fig. 1.4 more explicit as the model is easier to identify and interpret. Model explicitness is also highly correlated to whether or not the model is captured in a non-executable, *declarative* form such as the static list of instructions in fig. 1.4, or in an executable, *imperative* form such as the pattern matching algorithm in fig. 1.5.

**Algorithm adaptability.** Algorithm adaptability captures the ability of a particular compiler backend algorithm (e.g. an instruction selection algorithm or a technology mapping algorithm) to adapt to new hardware with minimal modification. For example, because clang's instruction selection algorithm reads its instructions from declarative models such as the one presented in fig. 1.4, it can easily adapt to new instructions and hardware targets by simply being supplied a new list of instructions [110]. The snippet of Yosys's technology mapping algorithm presented in fig. 1.5, on the other hand, encodes a model of the target FPGA implicitly within the algorithm; thus, adapting it to a new FPGA would involve entirely rewriting the algorithm.

Now that we've introduced these terms, let's reconsider the two examples we have discussed so far: clang's instruction selector and Yosys's technology mapper. clang's instruction selector is powered by an *explicit* model of the x86 ISA, part of which is presented in fig. 1.4. Furthermore, its underlying algorithm is *adaptable* to new models; the user simply needs to update the model, and the algorithm will adapt. On the other hand, Yosys's technology mapper utilizes *implicit* models, and its algorithm is *inflexible*. To visualize this, we we plot Yosys's techology mapper and clang's instruction selector on a 2D plane with model explicitness on the horizontal axis and algorithm adaptability on the vertical axis, shown in fig. 1.6. Yosys's technology mapper,

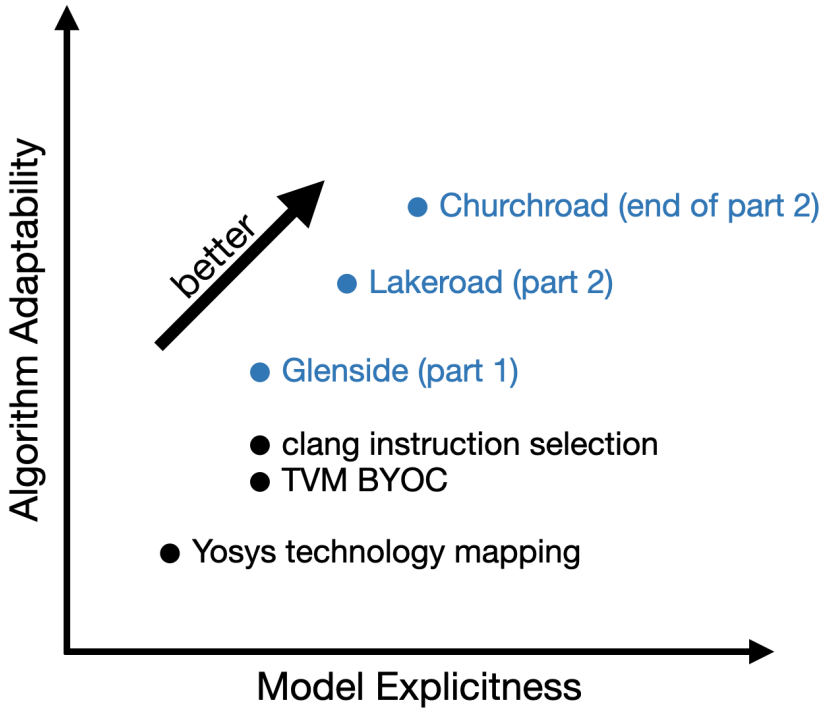



Figure 1.6: A visualization of where various compiler backend components fall on the model explicitness–algorithm automation spectrum. Glenside, Lakeroad, and Churchroad are the contributions of this dissertation.

being built upon implicit models and inflexible algorithms, is towards the bottom left. Meanwhile, clang's instruction selection is further up and to the right.

**The core claim of this dissertation, put informally, is that pushing up and to the right on our model explicitness–algorithm adaptability spectrum (fig. 1.6) produces better compiler backends.** By "pushing up", I mean using more adaptable *automated reasoning* algorithms to automatically generate compiler backends from explicit, formal models of hardware, as opposed to hardcoding backends using inflexible algorithms and implicit models. By "better compiler backends", I focus on three primary classes of improvements: correctness, optimization, and development time improvements.

Next, I present my formal thesis statement. Afterwards, I will break down the statement and discuss each part.

**Thesis Statement**

Automatically generating compiler backends from explicit formal models of the hardware they target enables optimizations, improves correctness, and reduces development time.

I will now explain and define the individual components of this thesis. I will use a set of five keywords to refer back to to the primary components of my thesis statement: its two "inputs", **Algorithms** and **Models**, and its three "outputs", **Optimizations**, **Correctness**, and **Devtime**.

**Automatically generating compiler backends.** At the highest level, my thesis advocates for automatically generating compiler backends. By this, I mean utilizing automated reasoning techniques to automatically implement compiler backend tasks like tensorization in ML compilers and technology mapping in FPGA hardware synthesis tools. At the highest level, automatically

generating compiler backends takes the form of specializing an automated reasoning algorithm to a specific compilation task by feeding it a hardware model. Thus, I break down the automated generation of compiler backends into two primary design decisions: choosing an automated reasoning algorithm (hence referred to with the keyword **ALGORITHMS**) and choosing the formal hardware models to feed into the algorithms (hence referred to with the keyword **MODELS**). **ALGORITHMS** refers to the automated reasoning algorithms we use to automatically generate our backends. **MODELS** refers to the hardware models which the automated reasoning algorithms consume to generate the compiler backend. I thus consider the **ALGORITHMS** and **MODELS** as "inputs" or the independent variables in my research. In parts I and II, I will show how different choices of **ALGORITHMS** and **MODELS** produce different results. In part I, we focus on an algorithm called *equality saturation*, and our models take the form of program rewrites capturing the high-level functional behavior of hardware accelerators. In part II, we focus on an algorithm called *program synthesis*, and we directly utilize vendor-provided Verilog simulation models as our models of hardware.

In my thesis statement, I claim that automatically generating compiler backends benefits compiler **OPTIMIZATIONS**, **CORRECTNESS**, and **DEVTIME**. We consider these the "outputs" or dependent variables in our experiments in parts I and II. I now describe each in detail.

**OPTIMIZATIONS.** A key task of a compiler backend is to utilize the target hardware efficiently to produce optimized programs. Compiler backends which rely on poor, implicitly encoded models and inflexible algorithms leave key optimizations on the table. In part I, we demonstrate how inflexible algorithms lead to missed accelerator mapping opportunities in deep learning compilers. In part II, we show how inflexible algorithms and implicit models lead to poor utilization of specialized FPGA primitives during FPGA compilation. Both of these cases correspond to missed

opportunities for optimization.

**Correctness.** Compiler backends are expected to produce correct code. However, backends which are built on implicit models of hardware which are deeply integrated into the algorithms themselves can have hard-to-find bugs. A cleaner division between hardware model and algorithm makes it easier to find bugs on either side. Furthermore, many of the more adaptable algorithms I advocate for in this dissertation (equality saturation, program synthesis) have open-source, highly-used, well-tested implementations which are likely more trustworthy than a hand-implemented algorithm used only within a single compiler backend. In part I, I demonstrate how the difficulty of building compilers leads to a lack of validation in machine learning accelerators. I show how automatically generating compilers for accelerators aids in rapid testing and validation, and directly leads to uncovering bugs in real hardware designs. In part II, I demonstrate how utilizing automated reasoning methods, we can generate correctness guarantees stronger than the guarantees provided by any existing hardware synthesis tool.

**Devtime.** And lastly, more explicit models and more adaptable algorithms can ease compiler development. Implicit models, especially models deeply integrated into the algorithms themselves (such as the Yosys example in fig. 1.5) are generally harder to comprehend and thus harder to update. Understanding implicit, imperative models is often more challenging than understanding explicit, declarative models. More flexible algorithms reduce development time in a number of ways. First, the algorithms this dissertation promotes all have free-to-use open source implementations, thus alleviating the need for the compiler engineer to write their own algorithm by hand. Second, the greater flexibility of the algorithms allows them to adapt to new hardware with less engineering effort. In both parts I and II, we demonstrate how compiler backends generated with automated reasoning algorithms are more easily extensible. In both cases, to target a new hardware platform,

users simply need to provide models of the target hardware. In part I, these models come in the form of rewrites capturing accelerator functionality. In part II, we use simulation models of FPGA primitives provided by the FPGA vendors.

I will demonstrate this thesis in two parts. These parts are visualized on our model explicitness–algorithm adaptability spectrum in fig. 1.6. In part I, I introduce Glenside [162] and 3LA [81], which demonstrate how a more adaptable algorithm can increase a compiler's ability to offload operations to machine learning accelerators. As is shown in fig. 1.6, part I only pushes along one axis of our spectrum: namely, algorithm adaptability. In part II, I more fully realize my thesis statement via Lakeroad [161]: a technology mapper for FPGAs which utilizes both more adaptable algorithms and more explicit models. In the end of part II, I also describe Churchroad [163], which seeks to extend the power of Lakeroad to larger hardware designs. Glenside, Lakeroad, and Churchroad demonstrate how, by automatically generating portions of compiler backends using more adaptable algorithms and more explicit models of hardware, we we improve their optimization ability and correctness, while easing development effort.

Before jumping into the content of this dissertation, though, let me first take the time to situate this work in the existing literature and explain my novel contributions.

## Situating this Dissertation: What's New?

The automatic generation of compilers is not a new idea; in fact, it has been somewhat of a holy grail for decades. So what does this dissertation bring to the table? Before elaborating on that question, I will briefly chronicle related work from the past five decades related to the top-level topic of compiler generation. Then, I will discuss the novel insights of this dissertation: namely, (1) taking advantage of the new wave of powerful, off-the-shelf automated reasoning

tools, (2) constraining ourselves to domain-specific tasks to limit the size of the problem, and (3) directly generating compiler backends from externally-supplied (i.e., not written by us) models of hardware.

The earliest citations for automated compiler generation go back to the late 1960s and early 1970s. However, though it was often referred to as *compiler* generation, much of this work focused solely on compiler *frontends:* parsers and lexers [127, 123, 92, 156, 164, 61]. A common task was, given a grammar for a new programming language, could you generate the parser and lexer for that language. This work culminated in industry-standard tools like Yacc and GNU Bison. Even just Yacc's name, which stands for "yet another compiler–compiler" shows how our terminology may have changed; while Yacc might have been considered a compiler–compiler in the past, now it only handles the very frontend of compiler tasks: the parser and lexer.

However, not all of the initial wave of research focused on compiler frontends. There was also work on generating components of compiler *backends*—often referred to as "code generation." In a 1977 survey of code generators from R. G. Cattell [1], he states:

> Traditionally, compiler-generation systems have been weak on automating the later stages of compilation, specifically code generation. But as the formal methods and grammars applied have become better understood and more powerful, their scope has gradually been evolving towards the later stages of compilation.

Certainly this entire dissertation can simply be seen as one more step in this gradual evolution. As we will discuss later, this dissertation benefits from the nearly five decades of formal methods and automated reasoning research since the time of Cattell's writing.

Nonetheless, researchers did approach the topic of backend generation [165, 65]. Interestingly, the spectrum I identify earlier in this chapter—the model explicitness–algorithm adaptability spectrum—seems to apply even in the early years of automated backend generation research. In

his 1977 survey, Cattell describes a very similar spectrum in methods of automatically producing code generators:

> In general, there have been two kinds of approaches to more automatic production of code generators. The first is the development of a specialized language for code generators, with built-in machinery for dealing with common details of the process. The second extreme is the development of a program to build a code generator for a language from a purely structural and behavioral machine description. Rather than being mutually exclusive, these procedural and descriptive language approaches, respectively, represent points in a continuum of degrees of automatic programming.

The two extremes on Cattell's spectrum have analogous points on my model explicitness–algorithm adaptability spectrum. At one extreme on his spectrum, he is essentially describing code generator DSLs: specialized languages for building procedural (in his words) or imperative (in mine) code generators. We've already seen a modern example of this—Yosys's pmgen DSL in fig. 1.5. At the other extreme of his spectrum are programs which produce code generators from descriptive/structural (in his words) or declarative (in mine) machine descriptions and hardware models. Again, we have seen a modern example of this in fig. 1.4: clang's instruction selector and the declarative model of the ISA which it consumes.

Much of the early work on the automated generation of code generators (i.e. compiler backends) was based on intermediate representations. To generate compilers, these works introduce an intermediate representation; then, those wishing to generate a compiler would then specify how to compile that intermediate representation to their target machine. A prime example of this was Perry Miller's DMACS system [118] where DMACS stood for "Descriptive MACro code generating System." This system introduced macros, which were machine-independent operations which could be implemented differently for each target machine. Much of this work can be seen as

the predecessors of the now-standard method of building compilers using one or a number of intermediate representations, most recently made popular by the foundational LLVM compiler toolchain [104].

While it is further from the work of this dissertation, it is also worth mentioning work on partial evaluation [44, 66]. Futamura projections specifically are of interest, as they describe how compilers can be viewed as partial evaluations of interpreters. I do not use any partial evaluation techniques in this dissertation.

In the ensuing five decades, the topic of automated backend generation has seen steady interest [29, 53, 25, 46, 107, 26]. In the next few paragraphs, I highlight notable trends and important projects.

A common pattern in recent research is the application of more and more powerful automated reasoning algorithms, especially those employed in this dissertation: term rewriting [151, 52, 58, 51, 47] and synthesis based on *SMT (SAT modulo theories)* [47]. One of the most notable projects in this space is SPIRAL [64]. Historically, SPIRAL is an umbrella over many related grants, researchers, and projects, but at its core is the SPIRAL compiler. The SPIRAL compiler's goal is to enable performance portability of specialized kernels across a wide range of architectures. They use many techniques also used in this dissertation, such as capturing programs in a high-level, backend-nonspecific language and utilizing automated reasoning (in their case, term rewriting systems) to adapt their compiler to different backends.

Another point worthy of note is the work of Norman Ramsey and his student João Dias, whose work on generating instruction selectors (among other compiler components) is very reminiscent of the work in this dissertation [143, 144, 53]. As is the general pattern in the work in this area, Ramsey and Dias focus on utilizing automated reasoning algorithms and high-level machine descriptions to implement compiler backend components, saving compiler development time.

If this dissertation focuses on generating software (i.e. compilers) from hardware, it is im-

portant to also mention the parallel line of research which attempts to generate hardware from software. Programs are a literal description of what needs to be computed; why not use them to determine what hardware we should make? A perfect example of this is the concept of *High-Level Synthesis (HLS)* [43, 42] which allows hardware designers to produce hardware from software algorithms written in high-level languages like C.[2] There is also an entire literature on hardware–software codesign [176, 97, 155, 198, 72], which, while rich and varied, generally centers on the idea of exploring the hardware design space using representative software workloads as a starting place. In fact, Glenside (presented in part I) was originally a hardware–software codesign tool which aimed to use equality saturation to rewrite machine learning models into potential accelerator designs.

The ideas presented in this dissertation are complementary to the ideas of hardware–software codesign, and I believe we should pursue both directions. The core idea presented in this dissertation—generating compiler backends from formal models of hardware—targets the lower layers of the compiler stack. In general, this dissertation answers the question of, given low-level primitives, how do we find places to use those primitives in programs or hardware designs? Hardware–software codesign, on the other hand, targets higher levels of the stack. Codesign seeks to answer the question, what hardware *should* we design, given the software we need to run? Thus, the methods presented here will only benefit codesign; better technology mapping from Lakeroad, for example, will only benefit the designs produced by HLS tools.

I claim there are three specific features of this dissertation which set it apart from existing literature. First is our focus on using *off-the-shelf* automated reasoning tools, rather than developing our own. Second is constraining ourselves to domain-specific tasks to limit the size of the generation problem. And last is the generation of compiler backends from *externally-supplied* (i.e., not written by us) models of hardware. I will now discuss each of these in detail.

[2]Note that the reality of modern HLS is a little messier than described, but this is the intention.

First, since the inception of automated compiler backend generation as a research topic, there have been significant advances in automated reasoning techniques, e.g. equational reasoning via equality saturation [174, 195], program synthesis [166, 178], and machine learning for program generation [4, 10]. Many of these advances have made automated reasoning techniques accessible to a broader audience. Whereas previous work often may need to build automated reasoning techniques by hand, in this dissertation I show how off-the-shelf tools have become powerful enough to use for tasks of this magnitude. Specifically, I utilize the equality saturation library `egg` [195] in part I and the program synthesis library Rosette [178, 179] in part II.

Second, we focus on compiler generation for specialized hardware backends. Compiler backend generation projects of the past often focused on generating compiler components for general-purpose processors [60, 107, 25, 24, 26, 143, 144, 53]. As hardware becomes more heterogeneous, however, the variety of hardware targets needing compilers has increased. In this dissertation, we focus instead on building compiler components for new, specialized platforms—machine learning accelerators in part I and specialized FPGA *primitives* like DSPs in part II. Not only are compilers needed for these new specialized targets, but their specialization also constrains the search space, making automated reasoning far more tractable.

Lastly, while previous works have often leaned on machine descriptions [144] when generating compiler components, these machine descriptions generally must be handwritten by the compiler engineer utilizing the compiler generation framework. In part II we introduce a new technique—semantics extraction from Verilog—which directly leverages vendor-supplied Verilog. Many of these models are built to be used with automated reasoning tools, but are currently only utilized for *post-compilation verification,* rather than in compilation itself [158, 159]. I demonstrate how these models can be used to generate more correct, more complete compilers for specialized hardware.

# Part I

# Compilation to Machine Learning Accelerators

# Part I Abstract

In part I, I describe an application of my underlying thesis to the generation of compilers for machine learning accelerators. Specialized hardware, especially for high-performance fields such as machine learning, have only grown in importance over the last decade. Despite this increased importance, it remains surprisingly difficult to build compilers for specialized hardware. Existing approaches require significant developer effort (**Devtime**), and often leave **Optimizations** on the table. Because building a satisfactory compiler remains so challenging, many hardware designers choose not to. Without a compiler, designers are unable to test their hardware on full workloads, leaving crucial bugs undiscovered (**Correctness**). In the following chapters, I describe how, responding to the lack of testing in the accelerator design community, we applied my thesis to automatically generate compiler backends targeting machine learning accelerators. Specifically, we utilized equality saturation (**Algorithms**) driven by rewrites capturing the functional behavior of accelerators (**Models**) to produce a backend which requires little developer effort to use (**Devtime**) but finds more mapping to accelerators than existing work (**Optimizations**) and enables hardware designers to run crucial end-to-end testing (**Correctness**). This is wrapped up inside a language and tool called Glenside; Glenside was then integrated into a larger compiler called 3LA. Part I draws from both the Glenside paper, "Pure Tensor Rewriting via Access Patterns" [162], and the 3LA paper, "Application-level Validation of Accelerator Designs Using a Formal Software/Hardware Interface" [81].

# Chapter 2

# Introduction and Motivation

Hardware acceleration has powered significant advances in subfields like artificial intelligence, image processing, and graph analysis [77, 37, 147, 207, 76, 75, 95, 99, 150]. This trend has highlighted the need for flexible accelerator support in domain-specific compilers like Halide [141], TVM [35], TensorFlow/MLIR [3, 105], and PyTorch [137].

Despite our increasing dependence on accelerators, building compilers for custom accelerators remains a daunting task. Developing a compiler for a custom accelerator requires significant **Devtime**. Current frameworks for compiler generation generally require significant compilers expertise, and the sheer amount of effort required to build a compiler from the ground up generally limits bespoke compiler construction to teams at large companies, e.g. the TensorFlow stack [3] for Google's TPU [95, 94]. Though projects such as MLIR [105, 106, 57] and Exo [85] have begun to prescribe a general framework for structuring a compiler, these tools are built for domain experts, and require significant time investment. Even once a compiler is constructed for a piece of custom hardware, it may still miss crucial **Optimizations**—in this case, taking the form of accelerator mapping opportunities.

Furthermore, the difficulty in building compilers for accelerators has a negative effect on

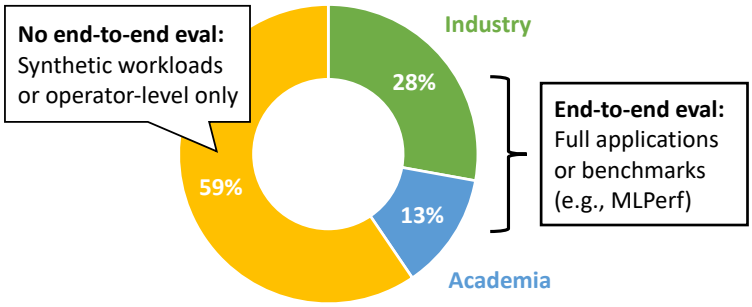



Figure 2.1: *This figure is reproduced from Huang et al. [81].* **Gap in end-to-end evaluation of accelerators for neural network applications:** Survey of papers from ISCA, MICRO, VLSI, and ISSCC in 2021 and ICCAD, DAC in 2020. The authors surveyed 79 papers introducing new DL accelerator designs/methodologies and determined how each accelerator was evaluated. Only 41% of the works reported end-to-end evaluation on non-synthetic applications, of which 68% (28% of the total) were from industrial teams.

accelerator **CORRECTNESS**. Core to ensuring correctness of accelerators is the process of end-to-end hardware *validation*—that is, testing the hardware design on full applications—and thorough hardware validation requires a working compiler. Some accelerator bugs will only be caught during full-application validation, especially bugs in the microarchitectural optimizations common in accelerator design [31, 59, 102]. For example, a deep learning accelerator may use a custom numeric format tuned to be storage-efficient, while still providing enough precision to support effective inference. Testing the numeric format on a single layer of the deep learning model may produce results well within the designer's error bounds. However, if the designer neglects to test *all* layers of the model, they may fail to discover that, even though individual layers behave well, the error may accumulate across layers, producing inaccurate application-level results [209]. Early end-to-end application level validation is thus essential for avoiding expensive and complex late stage hardware design changes.

The difficulty in building compilers is reflected in the literature. Figure 2.1 presents a figure originally published in Huang et al. [81], which shows how few new accelerator designs were evaluated on end-to-end applications. As we will see, this has practical consequences, as bugs in

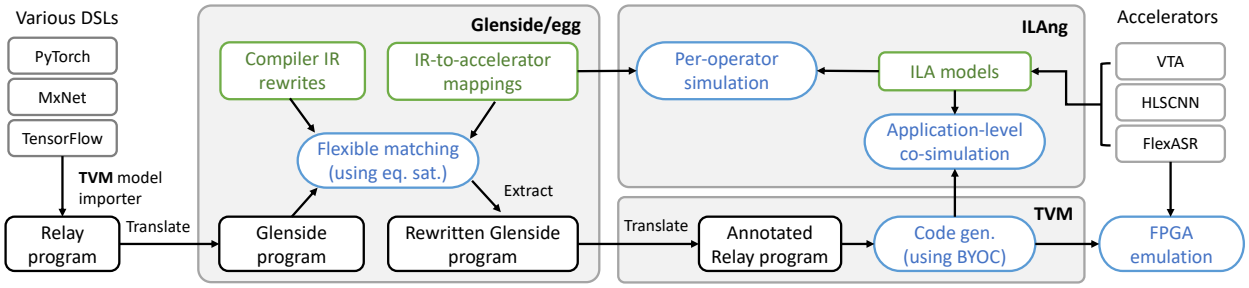



Figure 2.2: *This figure is reproduced from Huang et al. [81].* Diagram of the 3LA prototype's flow. Note that only the "Glenside/egg" portion of 3LA is considered a contribution of this dissertation; for more information on the other components, please see the original paper.

these accelerators may not be apparent until full end-to-end validation.

In response to the challenges of hardware validation to the common designer, Huang et al. (including the author of this dissertation) developed 3LA: a methodology to make testing easier for new accelerator designs [82, 81]. The 3LA flow is shown in fig. 2.2. The primary contribution of 3LA is a methodology to end-to-end evaluate accelerators on unmodified, full applications. While 3LA as a whole is not a contribution of this dissertation (see the dissertations of Bo-Yuan Huang [82] and Steven Lyubomirsky [113]), the motivations behind 3LA and the story of its development are important for part I of this dissertation.

3LA solves two major problems, the first being that generating simulators for hardware is difficult and time consuming. 3LA solves this problem using the Instruction-Level Abstraction (ILA) [84, 83]. ILA is a specification system originally designed for system-on-chip verification purposes. The authors of 3LA repurposed the ILA as a tool for capturing and simulating machine learning accelerator behavior, giving accelerator developers a framework for generating fast, high-level simulators for their accelerators. The first contribution of 3LA we do not consider a contribution of this dissertation; again, please refer to the dissertations of the coauthors for more information about 3LA as a whole.

The second challenge 3LA solves is that, even given a simulator, compiling full programs to the

simulator is difficult. At the time of developing 3LA, there were few open-source, flexible compiler frameworks for targeting custom hardware. One open-source framework supporting mapping to custom accelerators is TVM's Bring Your Own Codegen (BYOC) [38, 39]. Initially, the 3LA authors attempted to use BYOC to address their second issue. To use BYOC, the user provides syntactic patterns which BYOC then searches for in the workloads of interest. However, exact syntactic pattern matching (which we refer to simply as "exact matching") faces difficulties as there is often no canonical way to represent an operation, necessitating either the addition of more patterns or manual modifications to the input program to match the expected patterns. Application code can vary greatly in structure, particularly in the case of compiler IRs, which may be produced after several iterations of program transformations. Code variations are especially prevalent in machine learning compilers, where workloads are often imported from other languages via importers. Even for the same machine learning model, importers from different languages can produce wildly different (but equivalent) imported programs. Consider, for example, the compiler IR pattern for a linear layer in LSTM-WLM:

```
(bias_add (nn_dense %a %b) %c).
```

However, in another model (ResNet-20) the linear layers are equivalently expressed as:

```
(add (reshape (nn_dense %a %b) %s) %c)
```

when `%c` is a vector, for certain shapes `%s`. Though they are *semantically* equivalent, they are not *syntactically* equivalent, and thus we cannot match both of these operations with a single syntactic pattern. With an inflexible system like BYOC, we would be forced to add a new pattern for each possible way of writing the operator of interest. To put this in the framing of my dissertation, BYOC is near the bottom-left corner on our model explicitness–algorithm adaptability spectrum in

fig. 1.6. BYOC's **Models** of hardware are explicit: rewrites, capturing the functioning of hardware. However, its **Algorithms**—the underlying exact matching algorithm—is inflexible, unable to find semantically equivalent but syntactically different matches.

This is where I was able to apply my dissertation. When existing **Algorithms** (exact matching) proved inflexible, we introduced a new algorithm (equality saturation) whose increased flexibility made it easier to find opportunities to invoke accelerators. We developed a language called Glenside which allowed for the application of equality saturation to the task of accelerator mapping. We then integrated Glenside in 3LA to solve the second challenge listed above.

Note that we do not improve upon model explicitness in part I; notice that the point for Glenside is above, but not further to the right of, the point for BYOC on fig. 1.6. Both Glenside and BYOC use similar patterns to capture the functioning of hardware. In part II, we will show how we can increase both algorithm adaptability *and* model explicitness.

The rest of part I proceeds as follows. In chapter 3, we introduce Glenside. In chapter 4, we evaluate Glenside, primarily by demonstrating what Glenside is able to achieve when integrated into 3LA. In chapter 5, we present related work.

# Chapter 3

# Glenside

*This chapter is derived from Smith et al. [162].*

In the previous chapter, we laid out our need for tooling which can allow us to reason about tensor programs flexibly. While developing the 3LA methodology [81], we needed a tool which would allow us to find opportunities to invoke accelerators within machine learning workloads. TVM's Bring Your Own Codegen (BYOC) [39] framework was ostensibly designed for this purpose, but as we saw, BYOC was not flexible enough for the workloads we cared about.

To increase the likelihood of finding matches, pattern matching often relies on additional transformations to canonicalize intermediate representations (IRs) and massage data layouts into formats matching accelerator requirements [134, 126, 74]. Put in terms of our thesis, these transformations increase the *flexibility* of our underlying matching **Algorithms**, pushing us further up in our algorithm adaptability–model explicitness spectrum (fig. 1.6). Once we start modifying the source program to find matches, the problem of mapping to accelerators becomes a *term rewriting* problem, and thus we should be able to take advantage of the wealth of existing knowledge on term rewriting techniques [11].

Term rewriting is a well-known technique for program transformations, with some compiler optimizations being implemented as term-rewriting systems [50, 12, 20, 116]. Given a set of syntactic rewrite rules ($\ell \rightarrow r$) that also preserve semantic equality, a term-rewriting system rewrites instances of pattern $\ell$ in the input program with semantically equivalent pattern $r$ where applicable.

One term rewriting approach of particular interest is equality saturation. In traditional term rewriting, applying one rewrite rule may prevent using other, potentially profitable, rewrite rules; this is referred to as the phase-ordering problem [193]. Equality saturation avoids phase-ordering issues by searching over many equivalent rewritings of the same program [175, 93]. Given an input program $p$, equality saturation repeatedly applies the given rewrite rules to explore all equivalent ways to express $p$ using an *e-graph* data structure to efficiently represent an exponentially large set of equivalent program expressions [125, 131]. Upon reaching a fixed point, i.e., when no application of any rewrite rule can introduce a new program expression, or upon hitting a predetermined resource limit, the optimal rewritten program can be extracted from an e-graph according to a given cost function.

To increase flexibility of accelerator mapping **ALGORITHMS**, 3LA sought to employ equality saturation; unfortunately, existing IRs in compilers for array/tensor programming DSLs are not compatible with equality saturation. Equality saturation is most easily applied in *pure* (side effect–free) IRs that support equational reasoning. Due to their purity, these IRs tend to be high-level. However, mapping to accelerators requires considering low-level hardware details like data layout. Existing pure IRs for ML frameworks are used primarily for high-level transformations (e.g., type elaboration and inlining) and do not expose low-level data layout details [152]. On the other hand, IRs used for crucial lower-level optimizations like operator fusion must support precise reasoning about memory use, and therefore are typically impure, hampering term rewriting. In summary, for our purposes, existing IRs are either pure but too high-level, or low-level enough but impure.

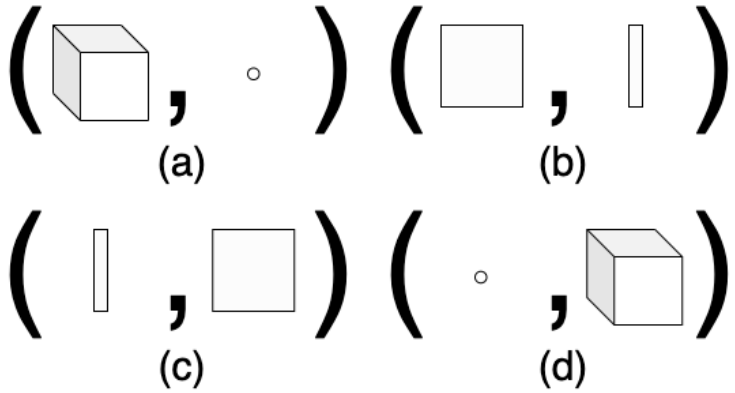



Figure 3.1: Four access patterns, representing different ways a tensor program (or *kernel*) might access the same 3D tensor. For example, (c) represents accessing a 3D tensor as a vector of 2D matrices.

To help mitigate such impedance mismatches, we present *Glenside*,[1] a pure tensor program IR that enables hardware-level term rewriting. Glenside is based on a simple *access pattern* abstraction that supports expressing and reasoning about data layout transformations via syntactic rewrite rules. When combined with standard arithmetic rewrites for per-tensor-element computations, access patterns enable implementing complex transformations for accelerator support as compositions of simple rewrites.

Tensors are traditionally characterized by their *shape*, an $n$-tuple of positive integers indicating the size of each of a tensor's dimensions. Access patterns instead characterize each tensor with two shapes, e.g., $((x), (y, z))$, separating the dimensions which are *iterated over* from the dimensions which are *computed on.* Figure 3.1(c) depicts an example where a 3D tensor's first dimension is iterated over and some computation applied to each corresponding 2D matrix.

In the rest of this chapter, we will first walk through an example demonstrating the difficulty in developing a pure tensor IR section 3.1. We will then describe the implementation of Glenside in section 3.2. Lastly, we describe how Glenside was incorporated into 3LA in section 3.3.

[1]Publicly available at `https://github.com/gussmith23/glenside`.

## 3.1 From Pure `matMul` to IR Design Goals

Applying functional techniques and term rewriting to tensor IRs requires careful design. For example, we must ensure that operators be compositional with respect to tensor shapes and that the representation support generic rules within the target rewrite engine. To highlight such constraints and motivate access patterns in Glenside, this section illustrates potential pitfalls with a simple matrix multiplication example.

### 3.1.1 Pure Matrix Multiplication

We write `f64` for the type of 64-bit floats and `[A]` for vectors over type `A`. Using this notation, we can specify operators like dot product and 2D matrix transpose as:

```
dotProd :  [f64] * [f64] -> f64
 trans2 :  [[f64]] -> [[f64]]
```

Implementing 2D matrix multiplication on inputs $P$ and $Q$ requires computing an output matrix $R$ where $R_{ij} = \Sigma_k P_{ik}\, Q_{kj} = P_i \cdot Q_j^T$. The need to compute `dotProd` for every pair of a row from $P$ and a column from $Q$ suggests map and Cartesian product operators which we might specify with:

```
    map :  (A -> B) * [A] -> [B]
cartProd :  [A] * [B] -> [A * B]
```

Naively, we can almost implement matrix multiplication as:

```
matMul(P, Q) :=
   map(dotProd, cartProd(P, trans2(Q)))
```

However, the result type will have been flattened to just `[f64]`, making it impossible to compose with other matrix operators that expect `[[f64]]` inputs.

Our first problem is that the `cartProd` specification above "forgets" the shape of its arguments. We could change this specification to arrange the output as a matrix:

```
cartProd2D : [A] * [B] -> [[A * B]]
```

But this result type prevents directly mapping `dotProd`.[2] Now the problem is that `map` only applies a computation by iterating over the first (outermost) dimension of a tensor. If we specialize `map` to iterate over the second dimension:

```
mapAt2 :  (A -> B) * [[A]] -> [[B]]
```

then we can implement a compositional `matMul` operator that correctly produces results of type `[[f64]]` as:

```
matMul(P, Q) :=
   mapAt2(dotProd, cartProd2D(P, trans2(Q)))
```

[2]This simple type does not specify how `cartProd2D` orders its output relative to its input vectors. We assume the order expected for matrix multiplication.

While this gets us close to our goal of a pure, functional IR for tensor programs, as we'll see, this style also has its issues.

### 3.1.2 Glenside Design Constraints and Goals

This style of pure, higher-order functional program representation enables term rewriting and equational reasoning via rules like:

$$\texttt{dotProd(P, Q)} \leftrightsquigarrow \texttt{dotProd(Q, P)}$$

$$\texttt{trans2(trans2(P))} \leftrightsquigarrow P$$

$$\texttt{map(f, map(g, P))} \leftrightsquigarrow \texttt{map(f} \circ \texttt{g, P)}$$

$$\texttt{mapAt2(f, trans2(P))} \leftrightsquigarrow \texttt{trans2(mapAt2(f, P))}$$

However, some of these rules depend on the shapes of dimension-specific operators aligning. What happens when we need to support higher-dimensional tensors? Without a mechanism to abstract which dimensions of a tensor are being iterated as opposed to computed over, we would have to generate versions of each rule for every combination of dimensions. Worse, these problems do not only affect rewrite rules; they also lead to code blowup just to specify all the variants of tensor kernels that arise in practice—e.g. we would eventually need `mapAt3`, `mapAt4`, and so on.

One strategy to address these challenges is adding support for anonymous functions ("lambdas"), currying, and closures to the tensor program representation. These features can provide sufficient flexibility to handle shape alignment issues that otherwise may require dimension-specific operators like `cartProd2D` and `mapAt2` above. For example, given curried versions of

`dotProd` and `map`, we could have used such features to implement a curried `matMul` as:

```
matMul' P Q :=
   map' (λ r => map' (dotProd' r) (trans2 Q)) P
```

Alternatively, some IRs rely on index notation for even pithier implementations like:

```
matMul(P,Q)[i,j] := dotProd(P[i], trans2(Q)[j])
```

Unfortunately, these approaches all rely on some form of *name binding* which can significantly complicate term rewriting. Rewriting under binders, whether explicitly in the form of lambdas or implicitly with index notation, requires additionally analyzing the potential *contexts* (what names are bound to) of every subexpression. While it is still technically possible to apply state-of-the-art rewrite engines like `egg` [195] via explicit variable substitution rules and free variable analyses, we have found the additional complexity and rewrite search space blow up substantially eliminate the potential advantages of term rewriting in such IR designs.

All the above constraints inform Glenside's key design goal: providing an IR that flexibly supports specifying and composing higher-order tensor operators[3] over arbitrary dimensions while still enabling high-performance term rewriting techniques like equality saturation. In the rest of part I, we show how *access patterns* enable achieving these goals with a focus on applications to mapping application fragments down to specialized hardware accelerators.

Table 3.1: Glenside's access pattern transformers.

| Transformer | Input(s) | Output Shape |
|---|---|---|
| `access` | $((a_0, \ldots), (\ldots, a_n))$ and non-negative integer $i$ | $((a_0, \ldots, a_{i-1}), (a_i, \ldots, a_n))$ |
| `transpose` | $((a_0, \ldots), (\ldots, a_n))$, $\ell$ (a permutation of $(0, \ldots, n-1)$) | $((a_{\ell_0}, \ldots), (\ldots, a_{\ell_n}))$ |
| `cartProd` | $((a_0, \ldots, a_n), (c_0, \ldots, c_p))$, $((b_0, \ldots, b_m), (c_0, \ldots, c_p))$ | $((a_0, \ldots, a_n, b_0, \ldots, b_m), (2, c_0, \ldots, c_p))$ |
| `windows` | $((a_0, \ldots, a_m), (b_0, \ldots, b_n))$, window shape $(w_0, \ldots, w_n)$, strides $(s_0, \ldots, s_n)$ | $((a_0, \ldots, a_m, b'_0, \ldots, b'_n), (w_0, \ldots, w_n))$, where $b'_i = \lceil (b_i - (k_i - 1))/s_i \rceil$ |
| `slice` | $((a_0, \ldots), (\ldots, a_n))$, dimension index $d$, bounds $[l, h)$ | $((a'_0, \ldots), (\ldots, a'_n))$ with $a'_i = a_i$ except $a'_d = h - l$ |
| `squeeze` | $((a_0, \ldots), (\ldots, a_n))$, index $d$ where $a_d = 1$ | $((a_0, \ldots), (\ldots, a_n))$ with $a_d$ removed |
| `flatten` | $((a_0, \ldots, a_m), (b_0, \ldots, b_n))$ | $((a_0 \cdots a_m), (b_0 \cdots b_n))$ |
| `reshape` | $((a_0, \ldots, a_m), (b_0, \ldots, b_n))$, access pattern shape literal $((c_0, \ldots, c_p), (d_0, \ldots, d_q))$ | $((c_0, \ldots, c_p), (d_0, \ldots, d_q))$, if $a_0 \cdots a_m = c_0 \cdots c_p$ and $b_0 \cdots b_n = d_0 \cdots d_q$ |
| `pair` | two access patterns of shape $((a_0, \ldots), (\ldots, a_n))$ | $((a_0, \ldots), (2, \ldots, a_n))$ |

## 3.2 Glenside

This section details Glenside's implementation, focusing on its core abstraction, *access patterns*. We use Section 3.1's matrix multiplication as a running example throughout.

### 3.2.1 Access Patterns

Access patterns encode common tensor IR patterns where some tensor dimensions are *iterated over* (accessed) while others are *computed on*.[4] Section 3.1's `matMul` example *iterates over* dimension 0 of input $P$, while *computing on* dimension 1, effectively viewing $P$ as a 1D vector of 1D vectors.

Access patterns are specified by their *shape* — a pair of tuples of positive integers $(S_A, S_C)$. An

[3] As `map` and `mapAt2` in Section 3.1.1 illustrate, an IR can support higher-order operators without necessarily providing lambdas, currying, or closures.

[4] This is similar to NumPy's concept of *universal functions*.

access pattern of shape $(S_A, S_C)$ is, in turn, a tensor $T$ whose shape is given by the concatenation of the access pattern shape tuples $S_A$ ++ $S_C$; we refer to $S_A$ and $S_C$ as the *access* and *compute* dimensions of $T$, respectively.

Access patterns represent the view of an $(|S_A| + |S_C|)$–dimensional tensor as a tensor of shape $S_A$, each of whose elements has shape $S_C$. For an access pattern $T$ of shape $(S_A, S_C)$ where $|S_A| = n_A$, we use the syntax `(access` $T$ $n_A$`)` to represent $T$ in Glenside. For example, if a 2D matrix $T$ has shape $(m, n)$, then the Glenside expression `(access` $T$ `1)` yields an access pattern of shape $((m), (n))$.

The matrix multiplication example from Section 3.1 directly accesses the rows of $P$, but uses `trans2` to iterate over the columns of $Q$. Instead of requiring an explicit transpose operator, Glenside provides access pattern *transformers*.

### 3.2.2 Access Pattern Transformers

Access pattern transformers manipulate one or more access patterns to produce a new access pattern, allowing Glenside to support more complex patterns like slicing, transposing, and interleaving. Table 3.1 lists Glenside's transformers.

To produce an access pattern representing the columns of $Q$ for matrix multiplication, we employ the `transpose` transformer. It takes an access pattern and a list of dimension indices, and rearranges the dimensions of the access pattern in the order specified by the indices. If $Q$ has shape $(N, O)$, `(transpose (access` $Q$ `1) (list 1 0))` produces an access pattern of shape $((O), (N))$.

The `cartProd` transformer takes access patterns of shapes $((a_0, \ldots, a_n), (c_0, \ldots, c_p))$ and $((b_0, \ldots, b_m), (c_0, \ldots, c_p))$ respectively, and produces an access pattern of the shape $((a_0, \ldots, a_n, b_0, \ldots, b_m), (2, c_0, \ldots, c_p))$, where $(2, c_0, \ldots, c_p)$ represents a 2-tuple of the input access patterns' compute di-

Table 3.2: Glenside's access pattern operators.

| Operator | Type | Description |
|---|---|---|
| `reduceSum` | $(\dots) \to ()$ | sum values |
| `reduceMax` | $(\dots) \to ()$ | max of all values |
| `dotProd` | $(t, s_0, \dots, s_n) \to ()$ | eltwise mul; sum |

mensions. The access dimensions of the input access patterns are simply concatenated. In the matrix multiplication example, the Cartesian product of the rows of $P$ with the columns of $Q$ is an access pattern of shape $((M, O), (2, N))$, where the second shape represents a 2-tuple of a row from $P$ with a column from $Q$.

We have nearly re-implemented matrix multiplication example in Glenside. The final step is to implement the dot product, for which Glenside uses access pattern *operators*.

### 3.2.3 Access Pattern Operators

*Operators* are the only Glenside constructs which perform computation. They are invoked only in `compute` expressions, which map the operator over the compute dimensions of an access pattern. For an input access pattern $A$ of shape $((s_0, \dots, s_{m-1}), (s_m, \dots, s_n))$, and an operator $f$ with type $(s_m, \dots, s_n) \to (s'_{m'}, \dots, s'_{n'})$, the result of `(compute` $f$ $A$`)` will have shape $((s_0, \dots, s_{m-1}), (s'_{m'}, \dots, s'_{n'}))$; that is, a `compute` expression cannot change the access dimensions of the input access pattern. Table 3.2 lists the operators in Glenside.

Recall where we are in converting our matrix multiplication example: we have accessed the rows of $P$ and the columns of $Q$ and taken their Cartesian product, resulting in an access pattern of shape $((M, O), (2, N))$, and we need now to compute the dot product of these row-column pairs. In Glenside, the `dotProd` operator (see Table 3.2) does just that. To compute the dot product over our row-column pairs, we need only to apply `compute dotProd` to our access pattern, to produce an access pattern with final shape $((M, N), ())$. The entire Glenside specification of

```
(transpose                ;  ((N, O, H', W'), ())
 (squeeze                 ;  ((N, H', W', O), ())
  (compute dotProd        ;  ((N, 1, H', W', O), ())
   (cartProd              ;  ((N, 1, H', W', O), (2, C, Kh, Kw))
    (windows              ;  ((N, 1, H', W'), (C, Kh, Kw))
     (access activations 1) ; ((N), (C, H, W))
     (shape C Kh Kw)
     (shape 1 Sh Sw))
    (access weights 1)))  ;  ((O), (C, Kh, Kw))
  1)
 (list 0 3 1 2))
```

(a) 2D convolution.

```
(compute dotProd        ;  ((M, O), ())
 (cartProd              ;  ((M, O), (2, N))
  (access activations 1) ; ((M), (N))
  (transpose            ;  ((O), (N))
   (access weights 1)   ;  ((N), (O))
   (list 1 0))))
```

(b) Matrix multiplication.

```
(compute reduceMax      ; ((N, C, H', W'), ())
 (windows               ; ((N, C, H', W'), (Kh, Kw))
  (access activations 2) ; ((N, C), (H, W))
  (shape Kh Kw)
  (shape Sh Sw)))
```

(c) Max pooling.

Figure 3.2: Common tensor kernels from machine learning expressed in Glenside. Lines containing access patterns are annotated with their access pattern shape. $N$ is batch size; $H/W$ are spatial dimension sizes; $C/O$ are input/output channel count; $K_h/K_w$ are filter height/width; $S_h/S_w$ are strides.

matrix multiplication is shown in Figure 3.2b.

## 3.3 Glenside in 3LA

Now that we have presented Glenside, we will now briefly describe how Glenside was used within 3LA.

Operations which can be offloaded to accelerators can be captured via patterns, as discussed with TVM's Bring Your Own Codegen framework. Rather than attempt to enumerate all semantically equivalent patterns (a task that is tedious, error-prone, and likely to result in an incomplete enumeration), or expect users to modify their application code to expose expected patterns (demanding knowledge of the model and patterns as well as engineering effort), 3LA increases the flexibility of compiler backend algorithms by utilizing term rewriting and equality saturation

techniques to transform programs to expose the most matching opportunities for accelerator operation selection. It is in this task that 3LA uses Glenside.

Flexible matching uses two kinds of rewrite rules, both expressed in Glenside:

- Compiler IR rewrite rules: These are general-purpose Glenside-to-Glenside rules, independent of the accelerator, and are reusable and composable for various applications. We have developed a general set in 3LA including rules for, e.g., merging/splitting tensors, commutativity, associativity, and identities for common operators.

- IR-to-accelerator mapping rules: These rewrite rules are accelerator-specific, and translate from Glenside to black-box accelerator calls. When targeting new accelerators, accelerator designers are expected to provide these mappings.

All rewrites in 3LA are polymorphic over tensor size, which requires specifying relationships between the input and output sizes for operations that merge, split, or broadcast over tensors. This also makes a given IR-to-accelerator mapping more general and provides support for applications using different block sizes, strides, etc., without changing any rules.

In the extraction phase of equality saturation, the rewritten program optimizing the cost function is chosen. This provides flexibility in the criteria for selection among functionally equivalent candidates for accelerator offloads. In our evaluations where we focused on end-to-end functional testing, we used a simple cost function that maximizes the number of accelerator invocations. More sophisticated cost functions can incorporate information about performance or data movement costs, and thereby result in different offloads.

*Compiler IR rewrite rules.* Here, we describe three examples of compiler IR rewrite rules to

show different types of opportunities that can be exposed.

```
(compute dot-product (reshape %x %s))  →  (reshape (compute dot-product %x) %s)      (3.1)
(add (reshape (dense %a %b) %s) %c)  →  (reshape (bias_add (dense %a %b) %c) %s)  (3.2)
                                %x  →  (reshape (flatten %x) (shape-of %x))       (3.3)
```

The `reshape` operator takes a tensor and a shape vector as input and re-arranges the layout of the tensor to the given shape, and the `dot-product` operator takes a tensor as input and computes the inner product of vectors under the given axis [162]. Rule 3.1 exploits the properties of the two operators and shows that rearranging the application order of `reshape` and `dot-product` operators preserves the semantics. Rule 3.2 shows that linear layer machine learning kernels can be expressed using different arrangement and combinations of operators, e.g., `bias_add` (broadcasting) or the elementwise `add`. Rule 3.3 shows that de-simplifying a computation (e.g., flattening then unflattening) could expose more opportunities for matching rewrites. Moreover, combining these individual rewrite rules together enables more sophisticated rewrites. For example, combining Rule 3.1 and Rule 3.3 allows for the emerging `im2col` transformations for convolution kernels, without needing to specify the transformation as a new rewrite rule.

*IR-to-accelerator mapping rules.* Similar to compiler IR rewrite rules, we specify IR-to-accelerator mapping rules in Glenside. These rules are accelerator-specific, mapping supported operations to accelerator invocations. We now describe three examples of IR-to-accelerator mapping rules.

```
(compute dot-product (cartesian-product ?x ?w))  →  (vta-dense ?x ?w)                          (3.4)
(conv2d ?input ?kernel ?group ...)  →  (hlscnn-conv2d ?input ?kernel ?group)      (3.5)
{{LSTM Relay Pattern}}  →  (flexasr-lstm ?input ?hidden_0 ...)          (3.6)
```

Rule 3.4 maps tensor-level computation of dense matrix multiplication to VTA's `dense` operation. This allows matching decomposed coarse-grained operators and mapping to fine-grained accelerator operations. Rule 3.5 maps kernel-level computation of a 2D convolution to HLSCNN's `conv2d`

operation—a common accelerator offloading for deep learning kernels. Rule 3.6 maps an LSTM computation to FlexASR's `lstm` operation. Note that the LSTM computation (left-hand side) is specified using a pattern compiled from a Relay program; this Glenside feature helps express complex operations.

3LA utilizes equality saturation and the two types of rewrite rules to transform programs, aiming to expose the most matching opportunities for accelerator operation selection. It was not clear *a priori* whether flexible matching would be performant for accelerators with complex IR-to-accelerator mapping rules needed for available accelerator designs. Our evaluation results in the next chapter show that compiler IR rewrites can be combined effectively with a few IR-to-accelerator mapping rules in flexible matching, which finds more matches than exact matching in a reasonable time.

# Chapter 4

# Evaluation

Thus far, we have described how we have applied the thesis of this dissertation—that compiler backends should be generated from formal models of hardware—in the realm of deep learning accelerators. We first described the difficulties in developing compilers for deep learning accelerators. We then described how these difficulties motivated the creation of 3LA: a mostly-automated, end-to-end methodology for accelerator development. (Note again that 3LA itself is not a contribution of this dissertation.) We identified a specific problem within this domain as a problem of interest: mapping applications to accelerators. To address this problem, this dissertation introduced Glenside, a tensor language which enables powerful rewriting techniques. We then integrated Glenside into 3LA, to bring the power of equality saturation to bear on the task of mapping to accelerators.

This evaluation will first demonstrate the utility of Glenside via a number of case studies in section 4.1. Then, in section 4.2 we will evaluate the specific claims of our thesis (improved **Optimizations**, **Correctness**, and **Devtime**) by evaluating the components of 3LA to which Glenside was essential.

## 4.1 Case Studies

To demonstrate Glenside's utility, we first show how it enables concise specifications of several critical ML kernels (Section 4.1.1). We then show how Glenside's pure, binder-free representation enables mapping kernels to an example accelerator via direct application of generic rewrite rules (Section 4.1.2). Finally, we highlight how Glenside enables the flexible mapping of larger, more diverse kernels to our accelerator, utilizing the power of equality saturation to automatically discover a variety of program transformations. Specifically, we show how Glenside can automatically map convolutions to matrix multiplications (Section 4.1.3) and automatically map large matrix multiplications into a sequence of smaller matrix multiplications (Section 4.1.4). This portion of the evaluation is drawn from Smith et al. [162].

### 4.1.1 Representation of Common ML Kernels

Figure 3.2 lists the Glenside specifications of three common ML kernels: 2D convolution, matrix multiplication, and max pooling. Below, we discuss the specifications of 2D convolution and max pooling; see Section 3.2 for a description of matrix multiplication.

#### 2D Convolution

2D convolution (`conv2d`) is a core kernel in deep learning, defined element-by-element over tensors storing activations $A$, strides $S$, and weights $W$ as:

$$\text{out}[n, o, x, y] = \sum_{dx,dy,c} (A[n, c, S[0] \cdot x + dx, S[1] \cdot y + dy] \cdot W[o, c, dx, dy])$$

where $n$ indexes the output batch, $o$ indexes output channels, $x/y$ index spatial dimensions, $dx/dy$ index the convolutional window spatial dimensions, and $c$ indexes input channels. 2D convolution slides each of the $o$ filters of shape $(c, dx, dy)$ through each possible $(c, dx, dy)$–shaped window of the input images. At each of these locations, an elementwise multiplication and reduction sum is computed.

The Glenside specification of `conv2d` is shown in Figure 3.2a. We access the `weights` as a vector of $O$ filters and the `activations` as a vector of $N$ images. We leave the filters as they are, but form windows of shape $(C, K_h, K_w)$ over the activations using the `windows` access pattern transformer (Table 3.1). This produces an access pattern of shape $((N, 1, H', W'), (C, K_h, K_w))$, i.e., a batch of "images" of new spatial shape $(H', W')$, where every location is a window of the original input. Finally, we take the Cartesian product of the filters and the windows, compute their dot product, and `squeeze` and `transpose` the output into the correct layout.

### Max Pooling

Max pooling, commonly used in ML to condense intermediate activations ("act" below), is defined as:

$$\text{out}[n, c, x, y] = \max_{dx,dy}(\text{act}[n, c, \text{strides}[0] \cdot x + dx, \text{strides}[1] \cdot y + dy])$$

Max pooling slides a window of shape $(dx, dy)$ over all possible locations within the spatial (i.e., $x$ and $y$) dimensions. At each window location, it reduces the window to a scalar with the max operator. The Glenside specification merely applies `reduceMax` over each two-dimensional window.

```
(compute dotProd (cartProd ?a0 ?a1))
  ⟹ (systolicArray ?rows ?cols ?a0 (access (transpose ?a1 (list 1 0)) 0))
```
where `?a0` is of shape ((`?batch`), (`?rows`)) and `?a1` is of shape ((`?cols`), (`?rows`))

Figure 4.1: Our rewrite rewriting matrix multiplication to a systolic array invocation.

**Discussion**

Glenside separates the *computation* from the *data access patterns* in these kernels while exposing the simplicity of their computation—and the relative complexity of their data access. In all three kernels, the computation can be described with a single operator; most of the specification entails setting up the data access pattern.

Furthermore, Glenside exposes similar structure between kernels; for example, both `conv2d` and matrix multiplication feature the expression `(compute dotProd (cartProd ...))`.

At their core, these kernels are performing the same computation, but with different patterns of data access. In Section 4.1.3, we exploit this similarity in structure when mapping kernels to hardware.

These kernels highlight the expressive power of access patterns. Consider the use of `windows` in `conv2d` and max pooling. Both kernels form windows differently: `conv2d` forms three-dimensional windows over the channels, height, and width dimensions, while max pooling forms two-dimensional windows over the height and width. Rather than passing configuration parameters to `windows`, Glenside attaches this information to the tensors themselves.

### 4.1.2 Mapping `matMul` to Accelerators

Glenside can be used to uncover opportunities to invoke accelerator components—indeed, this is exactly how Glenside is used within 3LA. Consider a weight-stationary systolic array, a common

```
?a ⟹ (reshape (flatten ?a) ?shape)

(cartProd (reshape ?a0 ?shape0) (reshape ?a1 ?shape1))
  ⟹ (reshape (cartProd ?a0 ?a1) ?newShape)

(compute dotProd (reshape ?a ?shape))
  ⟹ (reshape (compute dotProd ?a) ?newShape)
```

Figure 4.2: Rewrites enabling the discovery of the `im2col` transformation.

matrix multiplication architecture. A weight-stationary systolic array with $r$ rows and $c$ columns takes two lists of length-$r$ vectors (the activations and weights, respectively), pairing each vector from one list with each vector from the other, and computes a dot product over each pair. The second list contains $c$ vectors, while the first can be of any length.

As discussed in section 3.3, Glenside's purity allows us to implement this hardware mapping task using a term rewriting system, in which we rewrite a matching program pattern to an invocation of our systolic array. Our rewrite is shown in Figure 4.1, mimicking `egg`'s rewrite syntax. Tokens starting with a question mark (such as `?a0` in Figure 4.1) are variables in the pattern, bound by the left-hand side (LHS), and then used on the right-hand side (RHS). `egg` also allows for conditions on rewrites, which we print below our rewrites.

To design our rewrite, we first must design the LHS to match program patterns that resemble the data access pattern and compute pattern of our systolic array. Glenside is eminently suitable for this task, as it can express exactly the data access and computation pattern we described for the systolic array. Pairing all vectors from one list with all vectors from another and computing the dot product of the pairs is represented as `(compute dotProd (cartProd ?a0 ?a1))`, binding `?a0` and `?a1` to the input access patterns. We encode the systolic array's constraints on the input shapes as a condition on the rewrite. Patterns which match the LHS are mapped to the RHS; in this case, we introduce a new `systolicArray` construct to represent the functioning of our

```
(transpose
 (squeeze
  (reshape           ;  ((N, 1, H', W', O), ())
   (compute dotProd  ;  ((N · 1 · H' · W', O), ())
    (cartProd
     (flatten        ;  ((N · 1 · H' · W'), (C · Kh · Kw))
      (windows (access activations 1)
               (shape C Kh Kw) (shape 1 Sh Sw)))
     (flatten        ;  ((O), (C · Kh · Kw))
      (access weights 1))))
   ?shape) 1) (list 0 3 1 2))
```

Figure 4.3: An `im2col`-transformed `conv2d`, after the application of the rewrites in Figure 4.2 and just before the application of the systolic array rewrite.

systolic array. The shape of the systolic array is given by the `?rows` and `?cols` parameters, and the inputs are given as access patterns. Note how we also transform the second access pattern to more accurately convey how the actual systolic array hardware accesses the weight tensor: it reads it all at once (hence, `(access ... 0)`), and expects it to be laid out in transposed form in memory. This added information—enabled by Glenside's access patterns—provides richer data layout information, potentially helping future rewrites or code generation steps.

### 4.1.3 Flexible Mapping: Discovering `im2col`

The `im2col` transformation is a data layout optimization which enables computing `conv2d` on matrix multiplication hardware. The transformation involves instantiating the convolutional windows over the input activations directly in memory [33]. This leads to data duplication, but the resulting speedup more than offsets that overhead. In this case study, we show how a few general rewrites within Glenside lead to the *automatic rederivation* of the `im2col` transformation.

Glenside's representation underscores the structural similarity between `conv2d` and matrix multiplication, reflected also by the shared `(compute dotProd (cartProd ...))` between

`conv2d` and the LHS of the systolic array rewrite in Figure 4.1. Using this rewrite on `conv2d` would permit mapping it to the systolic array; however, the restrictions on the shape of `?a0` and `?a1` prevent its application. The systolic array has an activation access pattern of shape $((a), (b))$ and a weight access pattern of shape $((c), (d))$, while `conv2d` operates over access patterns of shape $((N, 1, H', W'), (C, K_h, K_w))$ and of $((O), (C, K_h, K_w))$, respectively. Transforming the access pattern into a lower-dimensional form would enable the systolic array rewrite.

Figure 4.2 shows the rewrites which enable this transformation. We call the first rewrite an *exploratory* rewrite as it optimistically matches any access pattern expression. It flattens an access pattern and immediately reshapes it back to its original shape, thus preserving equality (see Table 3.1 for formal definitions). This exploratory rewrite introduces the flattening necessary to convert the higher-dimensional access patterns of `conv2d` into the access patterns matched by the systolic array rewrite. However, the `reshape` operator will still need to be moved before we can fire Figure 4.1's systolic array rewrite. The second and third rewrites in Figure 4.2 take care of this; they implement *composition commutativity* of `reshape` with `cartProd` and `compute dotProd`, which "bubble" `reshape` operators up and out of expressions. These rewrites express general properties of these operators and are not specific to this task.

These three rewrites work in concert to map `conv2d` to a systolic array. First,[1] the exploratory rewrite flattens and reshapes all access pattern expressions. This includes the inputs to `conv2d`'s `cartProd` subexpression, which are flattened to shapes $((N \cdot 1 \cdot H' \cdot W'), (C \cdot K_h \cdot K_w))$ and $((O), (C \cdot K_h \cdot K_w))$ and reshaped back to their original shapes. Next, the composition commutativity rewrites for `cartProd` and `compute dotProd` fire in sequence, bubbling the `reshape` up through the `cartProd` and `dotProd` expressions (shown in Figure 4.3). Finally, the systolic array rewrite completes the `im2col` transform. Glenside's equality saturation based rewrite engine discovers

[1] Since equality saturation explores rewrites non-destructively, the rewriting order here is purely for explanatory purposes.

```
?a ⟹ (concat (slice ?a ?dim ?b0 ?b1) (slice ?a ?dim ?b1 ?b2) ?dim)

(cartProd ?a (concat ?b0 ?b1 ?dim))
  ⟹ (concat (cartProd ?a ?b0) (cartProd ?a ?b1) ?newDim)
if ?dim is an access dimension

(cartProd (concat ?a0 ?a1 ?dim0) (concat ?a2 ?a3 ?dim1))
  ⟹ (concat (cartProd ?a0 ?a2) (cartProd ?a1 ?a3) ?newDim)
if ?dim0 and ?dim1 are the same shape dimension

(compute dotProd (concat ?a0 ?a1 ?dim))
  ⟹ (concat (compute dotProd ?a0) (compute dotProd ?a1) ?dim)
if ?dim is an access dimension

(compute dotProd (concat ?a0 ?a1 ?dim))
  ⟹ (compute reduceSum (pair (compute dotProd ?a0) (compute dotProd ?a1)))
if ?dim is a shape dimension
```

Figure 4.4: Rewrites for blocking `matMul`.

these rewrites as the exploratory rewrite fires on every term and no rewrites are missed due to the absence of phase ordering.

This example highlights how, *with straightforward, generally applicable rewrites defined over Glenside*, equality saturation can emergently discover useful transformations that previously required expert insights to apply.

### 4.1.4 Flexible Mapping: `matMul` Blocking

Equality saturation can also be used with Glenside to emergently discover a matrix multiplication blocking scheme. Matrix multiplication blocking is the common strategy of breaking up a single, large matrix multiplication into smaller multiplications, by multiplying subsets of the input matrices and assembling the results to form the output matrix. This is essential in practice, as systolic arrays are small (often between $16 \times 16$ and $256 \times 256$) while matrices in ML and HPC applications can be much larger.

```
(concat                          ;  ((32, 32), ())
 (concat                         ;  ((16, 32), ())
  (compute reduceSum             ;  ((16, 16), ())
   (pair                         ;  ((16, 16), (2))
    (compute dotProd             ;  ((16, 16), ())
     (cartProd                   ;  ((16, 16), (2, 16))
      (slice                     ;  ((16), (16))
       (slice (access activations 1) 0 0 16) 1 0 16)
      (transpose                 ;  ((16), (16))
       (slice
        (slice (access weights 1) 0 0 16) 1 0 16)
       (list 1 0))))
    (compute dotProd             ;  ((16, 16), ())
     (cartProd                   ;  ((16, 16), (2, 16))
      (slice                     ;  ((16), (16))
       (slice (access activations 1) 0 16 32) 1 0 16)
      (transpose                 ;  ((16), (16))
       (slice                    ;  ((16), (16))
        (slice (access weights 1) 0 16 32) 1 0 16)
       (list 1 0)))))) ...
```

Figure 4.5: A $32 \times 32$ `matMul` blocked into $16 \times 16$ `matMul`s. Only two of the eight total multiplications are shown.

As in Section 4.1.3, this transformation follows from an exploratory rewrite and associated "cleanup" rewrites. The exploratory rewrite used for blocking is shown at the top of Figure 4.4. Given an access pattern, this rewrite slices the access pattern into two pieces along a dimension and then concatenates them back together. The dimension as well as the division strategy are configurable. For this example, we assume for simplicity that we run this rewrite on every available dimension, that we divide each dimension perfectly in half, and that all dimensions are powers of 2 in size. Figure 4.4 gives rewrites for bubbling the introduced `concat` operators up through the expression, namely the compositional commutativity of `concat` with `cartProd` and `compute dotProd`. Starting from the matrix multiplication in Figure 3.2b, assuming input shapes of $(32, 32)$, the exploratory rewrite first slices and concatenates the access patterns at the input of `cartProd`. Then, using the commutativity rewrites, the resulting `concat`s are bubbled up to produce the final expression in Figure 4.5. The effect of these rewrites is that the single $32 \times 32$ `matMul` becomes

eight separate $16 \times 16$ `matMul`s, which are summed and concatenated to form the full output matrix. This case study demonstrates yet again that Glenside's expressiveness allows a small set of rewrites to produce interesting and useful emergent transformations.

## 4.2 Evaluation as Part of 3LA

Next, we evaluate Glenside by way of evaluating 3LA. We first begin by describing the evaluation setup of 3LA: the accelerators we compile to in section 4.2.1, and the workloads we compile in section 4.2.2. This section is drawn from Huang et al. [81].

### 4.2.1 Target Accelerators

We added support for three deep learning accelerators that provide hardware operators at different levels of granularity:

1. **FlexASR** is an accelerator for speech and natural language processing (NLP) tasks that supports various RNNs [169]. It uses a custom numeric data type called *AdaptivFloat* for boosting the accuracy of quantized computations [170].

2. **HLSCNN** is an accelerator optimized for 2D convolutions [192]. It operates on mixed 8/16-bit fixed point data (8 bits for storing weights and 16 bits for computations).

3. **VTA** is a parameterizable accelerator for tensor operations featuring a processor-like design [122]. It supports element-wise arithmetic operations as well as generalized matrix multiplication, operating on 8-bit integer data.

For each accelerator, we defined an ILA model and a set of IR-to-accelerator mapping rules such as those described in sections 3.3 and 4.1.2. The ILA models for FlexASR, HLSCNN, and VTA

are approximately 5600, 1600, and 2100 lines of ILAng code (C++), respectively. The high-level synthesis (HLS) implementations of the accelerators are about 9300 (SystemC), 5100 (SystemC), and 6900 (Chisel) LoC, respectively; the ILA specifications are thus of modest size, compared even to the relatively compact HLS implementations. For each IR-to-accelerator mapping rule, we represent the compiler side in Glenside IR, and the accelerator side as a program composed of ILA instructions (in a Python-embedded DSL). The total size of mapping rules (both the compiler and accelerator sides) for FlexASR (5 mappings), HLSCNN (1 mapping), and VTA (1 mapping) was 186, 22, and 49 LoC, respectively. Recall that these mappings are polymorphic over tensor size on both sides, leading to general and compact representations. Additionally, the BYOC-based code generators and runtimes for these accelerators are approximately 450, 300, and 900 LoC of C++, respectively. These indicate the implementation of the code generation module in our prototype, as well as reusable utilities for data movement, handling custom numerics, and emitting the low-level MMIO code for each selected accelerator offload for end-to-end simulation of the application. This reduction in total lines of code between Glenside and BYOC mapping rules represents a significant decrease in **Devtime** for those seeking to build a compiler for their accelerator.

### 4.2.2 Target Applications

We considered six DL applications corresponding to common neural network models for language and vision tasks that contain operators supported by the three target accelerators. We selected applications with reasonable size for human inspection and in-depth analysis.

1. **EfficientNet** is a recent convolutional neural network (CNN) designed for image classification [171]. It has convolutions that are supported by VTA and HLSCNN.

2. **LSTM-WLM** is a text generation application [196] implemented using an LSTM recurrent

Table 4.1: **End-to-end compilation statistics.** The total number of Relay operators (row 3) is given as a proxy for program complexity. In rows 4-6, we include rewrites for only one accelerator at a time; we do not offload to multiple accelerators at once like in §4.2.5. Flexible matching identifies significantly more offloads than exact matching. Abbreviations: MN: MobileNet, Trans.: Transformer, and TF: TensorFlow.

| Application Statistics | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | Application | EfficientNet | LSTM-WLM | MN-V2 | ResMLP | Trans. | ResNet-20 | ResNet-50 | | |
| 2 | Source DSL | MxNet | PyTorch | PyTorch | PyTorch | PyTorch | MxNet | PyTorch | ONNX | TF |
| 3 | #Relay Ops | 232 | 578 | 757 | 343 | 872 | 494 | 709 | 194 | 609 |
| Number of Static Accelerator Offloads Identified Using Exact Matching/Flexible Matching | | | | | | | | | | |
| 4 | FlexASR | 0/35 | 1/1 | 0/41 | 0/38 | 0/66 | 2/22 | 0/54 | 0/54 | 0/54 |
| 5 | HLSCNN | 35/35 | 0/0 | 40/40 | 0/0 | 0/0 | 21/21 | 53/53 | 53/53 | 0/53 |
| 6 | VTA | 0/35 | 36/36 | 1/41 | 38/38 | 66/66 | 0/22 | 0/24 | 0/24 | 0/24 |

neural network architecture [70]. The LSTM layer in this model is supported by FlexASR.

3. **MobileNet-V2** is a common CNN designed for mobile applications [80, 154]. We chose MobileNet-V2 due to its wide use, especially on embedded devices.

4. **ResMLP** is a recent residual network for image classification, comprised only of multi-layer perceptrons [180]. Its linear layers could be accelerated by VTA and FlexASR.

5. **Transformer** is an NLP model comprised primarily of attention mechanisms [187]. We chose Transformer as a representative of recent popular NLP models.

6. **ResNet** is a popular CNN designed for image classification [78]. Besides ResNet-20, which we use in most of the evaluation, in §4.2.3, we additionally compare various implementations of ResNet-50 from MLPerf [120] for its availability of different reference implementations.

All applications were mapped to accelerators *without any manual modifications*.

### 4.2.3 Identifying Acceleration Opportunities

We took the six DL applications, developed by different teams in different DSLs, and compiled them for the three target accelerators. Our compiler successfully generated code that exploits the

accelerators for supported computations—thus improving over BYOC along the axis of **Optimizations**.

Table 4.1 shows the compilation statistics of using exact matching and flexible matching. Note that some accelerator operators correspond to multiple Relay operators; in particular, the LSTM RNN in LSTM-WLM corresponds to 566 Relay operators and maps to *one* FlexASR operator, which shows 3LA effectively overcoming a dramatic granularity mismatch between the compiler IR and accelerator operators.

Our results demonstrate 3LA's viability across a range of DL applications and accelerators with the successful identification of acceleration opportunities and provide evidence for the utility of flexible matching. For example, the linear layer rewrite described in section 3.3 resulted in 66 invocations of FlexASR's linear layer in Transformer and 38 in ResMLP, in comparison to exact matching that produced no match. Furthermore, the `im2col` transformation described in section 4.1.3 rewrites 2D convolutions into matrix multiplications; for VTA, this resulted in *additional* 35 invocations in EfficientNet, 22 in ResNet-20, and 40 in MobileNet-V2. Hence, flexible matching allowed us to support 2D convolutions on VTA even when there is no IR-to-accelerator mapping that maps 2D convolutions to VTA instructions. Another rewrite that turns lone matrix multiplications into linear layers (by a zero-vector bias) works in concert with the `im2col` rewrites, resulting in offloads of 2D convolutions onto FlexASR in EfficientNet, MobileNet-V2, and ResNet-20—thus allowing an accelerator for NLP applications to also accelerate vision applications. Note that these additional acceleration opportunities were identified automatically and are examples of *emergent effects* resulting from simple, reusable (accelerator-agnostic) compiler IR rewrite rules.

We additionally evaluate the robustness of flexible matching by comparing the three implementations of ResNet-50 from MLPerf [148] in Table 4.1, right. Their Relay representations differed in subtle ways (such as in reshaping operators)[2] and are reflected in the difference in results of exact

[2]For example, the TensorFlow implementation takes data in NHWC format rather than NCHW; Glenside can

Table 4.2: **Simulation-based validation results for checking IR-to-accelerator mappings.** The average relative error (Avg. Err.) and the standard deviation (Std. Dev.) of errors are measured over 100 test inputs. For VTA, there was no error because the host supports 8-bit integer operations.

| | Accel. | Operation | Avg.Err. | Std.Dev. |
|---|---|---|---|---|
| 1 | VTA | All ops | 0.00% | 0.00% |
| 2 | HLSCNN | Conv2D | 1.78% | 0.16% |
| 3 | FlexASR | LinearLayer | 0.84% | 0.29% |
| 4 | FlexASR | LSTM | 1.21% | 0.19% |
| 5 | FlexASR | LayerNorm | 0.27% | 0.20% |
| 6 | FlexASR | MaxPool | 0.00% | 0.0% |
| 7 | FlexASR | MeanPool | 1.79% | 0.28% |
| 8 | FlexASR | Attention | 4.22% | 0.09% |

matching. Flexible matching found the same (increased) number of matches for each accelerator, regardless of its source DSL. All of these results speak to the increased **OPTIMIZATIONS** found by utilizing Glenside.

### 4.2.4 Per-Operator Evaluation

Having now evaluated **DEVTIME** and **OPTIMIZATIONS**, we will evaluate **CORRECTNESS**. We do so by demonstrating how Glenside, via 3LA, enables testing of accelerators. We begin by evaluating single operators. Although evaluating individual operators does not suffice to characterize how an accelerator performs on a full application, it is a basic first step and provides insights on the identified acceleration opportunities. Here, we discuss both functional validation and performance evaluation at the operator level.

#### Functional Validation

The 3LA methodology readily enables operator-level validation through auto-generated ILA simulators. In our experiments, we compared the outputs of the accelerator ILA simulator and

rewrite convolutions to use NCHW.

those of TVM's runtime on host. The accelerator ILA simulators precisely model the data types used by the accelerators. For the reference results (TVM's runtime), we use 8-bit integer for comparing against VTA and 32-bit floating point for the other accelerators, as these are the closest host processor data types to those used by the accelerators. We measure the relative errors by using the standard Frobenius Norm [6] for the tensors based on the reference and accelerator generated output values as follows: $Error = \|Out_{ref} - Out_{acc}\|_F / \|Out_{ref}\|_F$.

We validated all types of operators supported by the target accelerators. Table 4.2 shows a representative subset of the validation results: four IR-to-accelerator mappings (Rows 1-4) that are used in the full application compilation (Table 4.1) and four additional mappings for non-trivial operations (Rows 5-8). Note that some mappings introduce no numerical differences; e.g., the TVM runtime supports 8-bit integer execution, so the results for VTA match perfectly. For other mappings, we see deviations caused by the custom numerics, especially for complex operators such as the attention operator on FlexASR. Such deviations should be carefully assessed in the context of application-level validation, as even small deviations could accumulate and affect the final accuracy.

**Performance Evaluation**

We also evaluated the performance gain of offloading operations from the host to accelerators using cycle counts as the performance metric, since we did not have clock frequencies for an SoC containing the host and accelerators. For accelerators, we derived the cycle counts based on their cycle-accurate models (VTA's Chisel model and FlexASR's and HLSCNN's SystemC models). For the host, we measured averaged cycle counts (1000 random inputs) in TVM's runtime on one pinned EPYC-7532 core.

Fig. 4.6 shows the performance gains (ratio of host to accelerator cycles) of all identified acceleration opportunities in ResNet-20 and MobileNet-V2 when operations are offloaded from

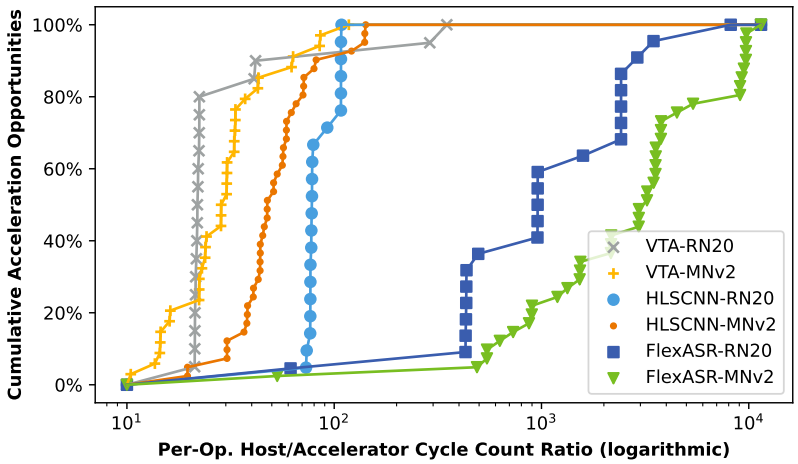



Figure 4.6: **Cumulative distribution of per-operator performance gains of all identified acceleration opportunities** in ResNet-20 (RN20) and MobileNet-V2 (MNv2) on the three accelerators. Each point represents an operation offloaded from the host to the accelerator (as identified by flexible matching, Table 4.1). The $x$-axis shows the host-to-accelerator cycle count ratio of each offloaded operation and the $y$-axis shows the cumulative distribution of offloaded operations. Points and plots more to the right are better; e.g., coarse-grained operators, supported with higher parallelism in FlexASR, offer greater speedup compared to the fine-grained operators in VTA.

the host to VTA, HLSCNN, and FlexASR, respectively. Overall, as expected, all offloads resulted in performance gains relative to the host; we also see that accelerators providing coarser-grained operators (e.g., FlexASR), supported with higher parallelism, achieve higher performance gain per operator compared to finer-grained accelerators like VTA.

### 4.2.5 Application-Level Validation Through Co-Simulation

We performed application-level co-simulation by using the ILAng-generated simulators for accelerator computations and the host CPU for the rest of the computation. We considered three applications, which between them provide opportunities to use each of the three accelerators: (1) LSTM-WLM, where we accelerate linear layer and LSTM operations on FlexASR; (2) ResNet-20, where we accelerate convolutions on HLSCNN and linear layers on FlexASR; and (3) MobileNet-V2, where we accelerate convolutions and linear layers as in ResNet-20 and additionally accelerate both these operations on VTA (due to the `im2col` rewrites). In ResNet-20 and MobileNet-V2, we were able to *explore using HLSCNN and FlexASR together and separately*, simply by varying which IR-to-accelerator mappings we included in flexible matching. Note again that this validation would not have been possible without the mapping power of Glenside.

We trained and validated the LSTM-WLM model using the WikiText-2 dataset [117]. The image classification models (MobileNet-V2 and ResNet-20) were trained and validated using the CIFAR-10 dataset [41]. We additionally trained and validated a MobileNet-V2 model optimized for ImageNet using the ImageNet dataset [49].

Table 4.3 shows the application-level co-simulation results. For LSTM-WLM, the application-level results using the accelerators did not differ greatly from the reference results. In the case of FlexASR, this was the *first time* it had been run end-to-end on a full application—this provided validation for its AdaptivFloat data type. For VTA on MobileNet-V2, there was a small decrease in

Table 4.3: **Application-level co-simulation results.** In each test, we evaluated 2000 CIFAR-10 images (for vision tasks) or 100 WikiText-2 sentences (for text generation) that were evenly sampled from the corresponding dataset. The reference results were obtained by running tasks in the original frameworks (MxNet for ResNet-20, PyTorch for the rest). The results without numerical tuning are for the initial accelerator designs, modeled in ILA. The result with numerical tuning, where provided, were obtained by updating the ILA specifications according to design revisions suggested by the accelerator developers. We measured the accuracy for image classification tasks (ResNet-20, MobileNet-V2) and perplexity for text generation (LSTM-WLM).

| Application | Processing Platform | Reference Result* | Result without Numerical Tuning | Result with Numerical Tuning |
|---|---|---|---|---|
| LSTM-WLM | FlexASR | 122.15 | 121.97 | N/A |
| ResNet-20 | FlexASR | 91.55% | 91.50% | N/A |
| | HLSCNN | 91.55% | 29.75% | 92.10% |
| | FlexASR & HLSCNN | 91.55% | 29.15% | 91.85% |
| MobileNet-V2 | VTA | 92.40% | 89.40% | N/A |
| | FlexASR | 92.40% | 92.30% | N/A |
| | HLSCNN | 92.40% | 10.35% | 91.50% |
| | FlexASR & HLSCNN | 92.40% | 10.35% | 91.20% |

∗ The reference result does not represent the best achievable accuracy/perplexity of the model on the given dataset. This table is intended for comparing the application-level results on different processing platforms.
† Average simulation time of running one data point (e.g., an image or a sentence) on an AMD EPYC-7532 core.

accuracy that may be attributed to quantization error.[3]

However, the initial results for ResNet-20 and MobileNet-V2 (both CIFAR-10 and ImageNet) using HLSCNN revealed a large loss in accuracy, as shown in Column 4 "Results without Numerics Tuning" in Table 4.3. We noticed that the linear layers accelerated by FlexASR did not impact the final accuracy, suggesting the issue stemmed from HLSCNN (for which this was also the first time it was run in an end-to-end application). We then instrumented our 3LA prototype to record additional information for each accelerator invocation, such as input and output ranges. This helped the accelerator developers determine that the loss of accuracy was due to a lack of

[3]We apply a form of uniform quantization [87], which involves scaling the results based on the floating point reference.

dynamic range in the data type: weight data values in HLSCNN's 2D convolutional layers were heavily quantized due to the narrow value range of their 8-bit fixed point representation. After we updated the ILA specification (a much easier task than modifying the RTL implementation) based on the developers' suggestion to expand the fixed point representation to 16 bits and adjust the binary points in inputs' and accumulators' fixed point data types, the accuracy recovered. This is shown in Column 5 "Results with Numerics Tuning" in Table 4.3. This case study readily demonstrates how the 3LA methodology, enabled by Glenside, *facilitates debugging and improving accelerator designs with rapid turnaround,* and thus improving overall **Correctness**.

The overall results in Table 4.3 reaffirm the need for application-level validation, especially for accelerators utilizing custom numerics. Thanks to formal ILA models and flexible compilation via Glenside, 3LA provides quick design space exploration and numerics tuning without hardware engineering overhead in each design iteration. Further, it provides handy debugging information and efficient simulation—for FlexASR, the ILA simulator yields a 30× speedup on average compared to RTL simulation.

### 4.2.6 System Deployment and FPGA Emulation

As an additional demonstration of 3LA and Glenside, we explored their use in compiling workloads to a real hardware platform. Specifically, we used our prototype to compile workloads to an FPGA emulation of FlexASR.[4] We configured our prototype to lower FlexASR ILA instructions to the corresponding MMIO commands for FlexASR, passing them to the FPGA using the Xilinx SDK [202]. Next, we compiled and executed synthetic workloads in which LSTM layers and linear layers were offloaded to the FlexASR accelerator. The results matched those of the ILAng-generated

[4]We synthesized and placed-and-routed the FlexASR accelerator on a Xilinx Zynq ZCU102 FPGA, which consumed 86% of the available LUT resources. Due to the significant engineering overhead of FPGA emulation, FlexASR is the only accelerator we deployed on an FPGA.

simulator bit for bit, providing validation for the custom numerics. This is a proof of concept for utilizing the 3LA methodology for an actual deployment, above and beyond simulation-based testing.

# Chapter 5

# Background and Related Work

In part I of this dissertation, we have described an application of our thesis to the problem of generating compilers for deep learning accelerators. We have described the difficulties in building compilers for custom accelerators. First, developing a compiler requires much developer effort and expertise (**Devtime**). Second, even once a compiler is built, optimizations still often get left on the table (**Optimizations**). And lastly, the difficulties in building compilers for accelerators often makes the process of validation of accelerators hard if not impossible for accelerator designers (**Correctness**). We describe how we applied our thesis to produce Glenside, a domain-specific language which can be used to automatically compile workloads to accelerators. We used Glenside to build 3LA, a new methodology for accelerator design.

We now present background and related work on the various topics involved in part I of this dissertation. We start from the basics, discussing existing machine learning accelerators (section 5.1), whose popularity is the core motivation behind building 3LA and Glenside.

## 5.1 Machine Learning Accelerators

A variety of accelerators [95, 37, 121, 115, 133, 69] have been developed to provide efficient implementations of tensor operators for ML applications. These devices accelerate tensor operators through hardware parallelism, simultaneously applying related operations across many tensors in the accelerator's memory (which are often laid out according to custom rules that facilitate hardware optimization). Tensor program compilers must translate expensive application code fragments down to accelerator invocations that adhere to these layout rules, which often involves both (a) higher-level transformations like tensor reshaping to match accelerator size bounds and loop unrolling to expose optimization opportunities, and (b) lower-level transformations like operator fusion and `im2col` to match accelerator calling conventions and even implement different operations using the same accelerator, e.g., on systolic arrays [33, 91].

## 5.2 Validating Hardware Designs

Validation of hardware designs is an incredibly important task—so important, in fact, that it often represents a majority of the cost for a new hardware design. Validating a hardware design is the process of ensuring that it behaves as intended. In the world of hardware design, this process is more commonly referred to as *verification*—however, in this dissertation, I prefer to use the definitions of *validation* and *verification* from the field of Programming Languages, in which *validation* refers to non-formal-methods-based, (usually) non-exhaustive sanity checking, while *verification* refers to formal, mathematical proving of properties (like correctness). Hardware designers and validation engineers perform validation at many points in the hardware design process. For example, they might validate that their initial prototype of the design, written in e.g. Python or C++, behaves identically to their initial Verilog implementation using a simulator

such as Verilator [189]. Later in the design process, they might formally verify the equivalence of their initial Verilog implementation against a lower-level, backend-specific version of the Verilog using an equivalence checker like Cadence's Jasper [90], or similar tools from Mentor Graphics or Synopsys.

Simulation tools can operate at different levels of specificity. Tools like Verilator [189] and Cuttlesim [139] enable *cycle-accurate RTL simulation*: i.e. they simulate exactly what the hardware is doing at each clock cycle. However, cycle-accurate simulation is slow, as it requires simulating every component within the hardware design. Often, it is useful to run *application-level co-simulation,* in which a high-level software program (e.g. a deep learning model) is simulated simultaneously with the hardware. Co-simulation is integral to the 3LA methodology—by running entire applications via co-simulation, 3LA helps designers find bugs in their hardware designs. In general, cycle-accurate simulators are too slow to run full applications in any realistic amount of time, making co-simulation infeasible. In these cases, higher-level, non-cycle-accurate simulation can enable fast simulation. SystemC [9] and ILA [84], are common tools for implementing these high-level models. The 3LA framework relies on ILA, which, unlike SystemC, provides a clear formal verification path to RTL.

## 5.3 Hardware–Software Co-Design

*Hardware–software co-design* refers to a loosely grouped set of techniques and ideas centered around one primary idea: rather than the well-defined separation between software and hardware design which existed in the past, hardware and software should instead be designed *together*. That is, to maximize the performance of hardware, hardware designers should be able to suggest changes to the software such that it will be more amenable to acceleration; similarly, software designers should be able to suggest hardware changes to enable new algorithms. Glenside and the

3LA methodology enable hardware–software co-design through the simple fact that they make hardware design iterations quicker. By making it quicker and easier to simulate full end-to-end applications on prototype hardware, 3LA and Glenside enable designers to see simulation results more quickly and adjust both their design and the software running on the hardware.

There exists much other work on hardware–software codesign for deep learning. Work on accelerator generation and integration [14, 181] has explored adding support in the Halide [142] compiler flow for specialized Coarse-Grained Reconfigurable Array (CGRA) accelerators. That work composes an impressive array of custom tools to generate and verify specialized CGRA accelerators and also map Halide program fragments down to accelerator invocations. HeteroCL [101] also provides a similar custom flow.

## 5.4 Tensor IRs and Compilers

Glenside, the primary contribution of part I of this dissertation, is fundamentally an intermediate representation (IR) for tensor programs. On top of the Glenside IR, we build the 3LA methodology, which includes a compiler utilizing Glenside to map deep learning workloads to accelerators. In this section, we describe other IRs and compilers for tensor programs.

Machine learning workloads are generally viewed as a sequence of of *tensor kernel* invocations, where tensor kernels are large operations over multidimensional arrays (tensors) such as 2D convolution or dense matrix multiplication. Machine learning frameworks (such as TensorFlow [3] and PyTorch [137]) and machine learning compilers (such as TVM [35]) can optimize machine learning workloads at a number of levels, which often result in each framework having a number of different IRs. TVM, for example, can optimize machine learning workloads at a high-level using its high-level IR Relax [100] (and its former high-level IR Relay [152]), but low-level optimizations such as loop blocking and reordering must be done in its lower-level IRs.

Tensor compilers for ML and HPC applications strive to balance clear, high-level operator semantics and support for the low-level optimizations necessary to target specialized accelerators. Halide [141] achieves this balance by separating operator *specifications* (what is computed) from *schedules* (how, when, and where each output element is generated). This style of separation has proven highly effective across both application domains and hardware targets; numerous compilers including TVM [35], FireIron [73], LIFT [167], and Accelerate [30] follow variations of this strategy.

The specification/schedule separation approach allows the same high-level program (specification) to be flexibly optimized for and mapped to different hardware targets by applying different schedules. From this perspective, schedules represent different rewriting strategies to explore various loop ordering and memory layouts; in LIFT and Accelerate these take the form of functional combinators closely related to Glenside's approach. As in classic term rewriting, experts must often carefully craft schedules for each target to achieve the best performance and mitigate phase ordering challenges [194], though recent projects have produced promising results in automatic scheduling [36, 208, 7].

Other tensor IRs like TACO [96], Keops [32], and COMET [177] rely on *index notation*[1] to concisely express tensor operators and simplify optimization by uniformly representing per-output-element computations. These approaches also rely on rewriting passes to generate kernel implementations specialized to tensor sparsity/density, operator combinations arising in the source application, and details of the target hardware. In Section 3.1 we discuss some of the tradeoffs of these approaches with respect to other rewriting strategies.

Polyhedral compilers [157] like Tensor Comprehensions [186] and Tiramisu [13] optimize loop-filled programs by modeling loop nests as polyhedra and applying geometric transformations. The polyhedral approach exploits regular loop structure, but is also restricted to geometrically

[1]Index notation is closely related to "Einstein notation", in which reduction indices are implicit.

affine transformations. In contrast, term rewriting is neither guided nor restricted by geometric constraints, making these approaches broadly complementary.

Another key piece of background to note when it comes to deep learning compiler frameworks is interoperability of frontends. Machine learning models can be expressed in many frontend languages, including MxNet [34], PyTorch [136], TensorFlow [3], ONNX [108], and CoreML [45], and generally, compilers should strive to support models expressed in all of these frontends. As it is built on top of TVM, which supports importing from many frontend languages, our 3LA prototype supports all of these frontend languages.

### 5.4.1 Term Rewriting and Equality Saturation

Term rewriting is a classic program optimization technique [11] that relies on iteratively applying rewrite rules of the form $\ell \longrightarrow r$: when part of a program matches the pattern $\ell$ under substitution $\sigma$, it is rewritten into $\sigma(r)$. This approach is ubiquitous, frequently used in both mainstream and DSL compilers to implement features including preprocessing, strength reductions, and peephole optimizations [68].

Classic term rewriting systems where rewrites are applied destructively suffer phase ordering problems [194]: the order in which rewrites are applied can enhance or severely diminish performance. Recent work has shown how program synthesis can help address this challenge in peephole optimizers like Halide's scalar expression rewriter [126].

Advances in alternate rewriting techniques like equality saturation [174] also mitigate phase ordering by exploiting the e-graph data structure from SMT solvers to repeatedly apply all rewrites simultaneously, thus obviating rule ordering considerations. In particular, the `egg` library [195] has been used to develop new synthesis and optimization tools across diverse domains [135, 124, 190], including DSP compiler vectorization [184] and tensor computation graphs [203].

Glenside provides the first tensor IR amenable to equality saturation by introducing access patterns to provide pure, higher order tensor kernel combinators that support rank-polymorphism without the need for binding structures like anonymous functions or index notation.

### 5.4.2 Pattern Matching Accelerator Calls

The most closely related work to flexible matching is from TVM BYOC [39], which only provides exact syntactic matching. Past work has also explored rewrite-based techniques for automatically inferring instruction selection passes between ISAs [144, 53] and in the context of superoptimization [15, 16]. Rewriting in 3LA instead operates on a high-level IR to expose opportunities to invoke code generators, rather than performing low-level code generation directly. Equality saturation has been used in the context of ML and DSP compilers for optimization [203, 185, 98]. There has also been significant work on ML and HPC compiler frameworks with varying degrees of support for targeting custom accelerators [142, 109, 35, 122, 106]. To the best of our knowledge, none of these frameworks provides support for testing prototype accelerators designs end-to-end on unmodified source applications.

# Part I Conclusion

In part I of this dissertation, I presented the first of two case studies giving evidence to my thesis. In chapter 2, noting the gap in compiler and simulation tools for machine learning accelerators and how it was potentially affecting accelerator **Correctness**, the authors of 3LA (including the author of this dissertation) set to developing a methodology for compiling to and testing designs. As part of this flow, we needed a tool for finding places in machine learning workloads where we could invoke accelerators. Existing tools missed mapping opportunities and were cumbersome to use, and thus poor with regards to **Optimizations** and **Devtime**. In response, in section 3.2 we introduced Glenside: a pure, binder-free tensor language which allowing for the use of more flexible **Algorithms**—namely, equality saturation—over machine learning workloads. We incorporated Glenside into 3LA to automatically generate tensorization routines in our compiler backend. In chapter 4, we demonstrated how 3LA, with the power of Glenside, finds more accelerator mappings (greater **Optimizations**) with less developer input (reduced **Devtime**). Furthermore, we showed how we used 3LA to find and fix bugs in real accelerators (aiding in **Correctness**).

Note that, while we improve upon the state of the art in algorithm flexibility (**Algorithms**), part I does not improve upon the state of the art in model explicitness (**Models**). In part II, I will more fully realize my thesis by utilizing *both* more adaptable algorithms and more explicit models to automatically generate backends for FPGA compilers.

# Part II

# Compilation to FPGAs

# Part II Abstract

In part II of this dissertation, I apply my thesis to a fully separate domain: hardware synthesis tools. FPGA technology mapping is the process of implementing a hardware design expressed in high-level HDL (hardware design language) code using the low-level, architecture-specific primitives of the target FPGA. As FPGAs become increasingly heterogeneous, achieving high performance requires hardware synthesis tools that better support mapping to complex, highly configurable primitives like digital signal processors (DSPs). Current tools support DSP mapping via handwritten special-case mapping rules, which are laborious to write (poor **Devtime**), error-prone (poor **Correctness**), and often overlook mapping opportunities (poor **Optimizations**). In part II of this dissertation, we introduce Lakeroad, a principled approach to technology mapping via sketch-guided program synthesis (**Algorithms**). A primary insight of Lakeroad is to utilize vendor-provided simulation models (**Models**) to generate the semantics needed by program synthesis. Lakeroad provides more extensible (**Devtime**) technology mapping with stronger correctness guarantees (**Correctness**) and higher coverage of mapping opportunities (**Optimizations**) than state-of-the-art tools. Across representative microbenchmarks, Lakeroad produces 1.4–3.6× the number of optimal mappings compared to proprietary state-of-the-art tools and 6–30× the number of optimal mappings compared to popular open-source tools, while also providing correctness guarantees not given by any other tool.

# Chapter 6

# Introduction and Motivation

*Part II is adapted from "FPGA Technology Mapping Using Sketch-Guided Program Synthesis" by Smith, et al. [161].*

Given a high-level hardware design specification (e.g., expressed in behavioral Verilog), FPGA technology mappers search for an equivalent low-level implementation in terms of the target FPGA's primitives. See fig. 6.1 for an example, where the high-level, behavioral `sub_mul` module ("input 1") is converted into FPGA-specific implementations ("their output" and "our output") using the Xilinx-specific `DSP48E2` primitive.

Historically, FPGAs consisted of relatively simple primitives, such as lookup tables (LUTs) and carry chains. Tools like ABC [149, 119, 28] *automatically* map to these basic primitives by translating designs to a library of simple logic gates and then packing those gates into LUTs.

However, FPGAs are becoming increasingly heterogeneous via the inclusion of specialized and diverse primitives such as digital signal processors (DSPs). Utilizing these specialized primitives effectively is now crucial for achieving high performance [188]. These specialized primitives make FPGA technology mapping far more challenging since technology mappers must now explore a

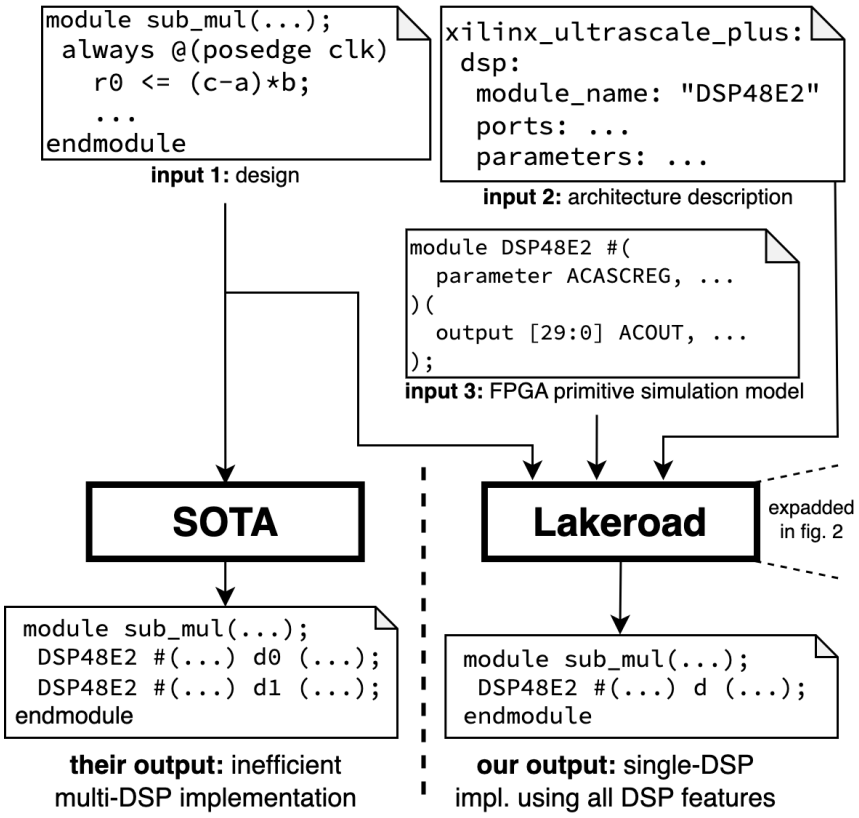



Figure 6.1: Even given a simple input design (input 1), the state-of-the-art (SOTA) hardware synthesis tool for Xilinx FPGAs frequently fails to efficiently use programmable primitives like DSPs. Lakeroad, on the other hand, can utilize all features of programmable primitives given just a short description of an FPGA architecture (input 2) and the vendor-provided simulation models of the primitive (input 3).

much larger search space while also satisfying each primitive's complex set of restrictions and dependencies. For example, Xilinx's DSP48E2 is a multifunction DSP with nearly 100 ports and parameters, whose numerous configurations enable support for a large variety of computations. The manual for the DSP48E2 alone is 75 pages long, where considerable text details the complex restrictions between the settings of the nearly 100 ports and parameters.

Existing technology mapping tools frequently fail to map designs to specialized primitives like DSPs, resulting in less-performant designs (that is, poor **Optimizations**) and requiring manual work for the hardware designer to recover the performance of their design (that is, increased **Devtime**). While existing toolchains have the ability to automatically infer locations where specialized primitives can be used in large designs, inference often fails [200, 201, 86]. In these cases, the designer can either accept lower performance and higher resource utilization, or they can perform what we call *partial design mapping*. During partial design mapping, the designer manually identifies and separates out the module that should be mapped to a DSP. They can attempt to re-run technology mapping on that module alone, in the hopes that mapping succeeds. Yet existing toolchains often fail *even in the partial design mapping case:* fig. 6.1 shows a simple module `sub_mul` which *should* fit on a single DSP48E2 according to the DSP's manual, but is instead mapped to two DSPs by current state-of-the-art tools[1]—a 100% increase in resource utilization! In the worst case, hardware designers are forced to manually instantiate complex primitives by hand, e.g., by looking through the 75-page DSP48E2 user manual to manually configure the DSP's dozens of ports and parameters.

Current state-of-the-art technology mappers are implemented via ad hoc, handwritten pattern matching procedures, which fall short in three primary ways. First, as we saw above, they are **incomplete:** they miss many mapping opportunities, even across microbenchmarks based on

[1]Licensing restrictions forbid naming the specific proprietary tools, but they are familiar, standard packages used by many hardware designers.

vendor documentation (**OPTIMIZATIONS**). Second, they do not provide strong **CORRECTNESS** guarantees: recent work highlights the significant number of bugs found across all major hardware synthesis tools [79]. Third, they are difficult to extend: *each* new complex primitive requires supporting detailed semantics and adding hundreds of new, special-case syntactic pattern matching rules [197] (hence increasing **DEVTIME**).

This chapter's key observation is that technology mapping is well-suited for the application of automated reasoning procedures (**ALGORITHMS**)—specifically, *program synthesis* [71]. We observe that the configuration space of a programmable FPGA primitive is essentially a small, bespoke programming language, and that program synthesis could be applied to automatically generate primitive configurations. We explore how program synthesis can simplify the design and implementation of FPGA technology mappers while providing **correct** (**CORRECTNESS**), **extensible** (**DEVTIME**), and **more complete** (**OPTIMIZATIONS**) support for mapping to diverse, highly configurable primitives like DSPs. Program synthesis techniques rely on automated theorem provers like SAT and SMT solvers [48, 17] to automatically generate programs satisfying a given specification. We demonstrate how *sketch-guided program synthesis* [166] can be adapted for FPGA technology mapping, leveraging the Rosette [179] program synthesis framework.

Sketch-guided program synthesis requires encoding the *semantics* of the target language: in our case, a machine-readable, mathematical model specifying the behavior of each FPGA-specific primitive being mapped to. In a typical synthesis tool, which generates programs for a single target language, this is a one-time cost. However, in our setting, each new FPGA primitive introduces yet another new target language, which in turn requires extending the tool to encode yet another formal semantics.

To support correct, extensible, and more complete technology mapping, we propose automating this process with **semantics extraction from HDL**, adapted from past work [46], to automatically extract complete primitive semantics from vendor-published HDL **MODELS** (fig. 6.1, "input 3").

Traditionally, such models have been used only for simulation and validation *after* technology mapping; we show that using the semantics to *implement* technology mapping with a program-synthesis-based approach yields substantially more complete FPGA technology mapping.

Sketch-guided program synthesis also requires *sketches*, which are partially complete programs with "holes" to be filled in by the solver. Sketches primarily serve to scale synthesis by constraining the set of programs that solvers explore when searching for one that satisfies the given specification, i.e., performance at the cost of completeness. In our setting, sketches correspond to arrangements of primitives, using holes as placeholders for some of the primitives' ports and parameters. This could be a single DSP with holes for its ports and parameters (as in the example in section 6.1.2), or a number of LUTs with holes for their LUT memories, or even a mixture of LUTs, DSPs, and carry chains. The synthesizer "fills in the holes" as necessary for the low-level FPGA-specific primitive to implement a given high-level behavioral design fragment. Unfortunately, developing effective sketches still requires synthesis expertise [21, 183]. Naïvely, our approach would also require new sketches for each new FPGA primitive we target.

To address these challenges, we introduce **architecture-independent sketch templates.** Hardware designs are often implemented using high-level blueprints that are similar across most FPGA architectures—sketch templates capture these blueprints and make them reusable across architectures. Therefore, by using sketch templates, we greatly reduce the overhead of supporting new architectures and diverse primitives. Typically, when adding support for a new primitive or FPGA architecture in Lakeroad, the hardware designer need not write or modify any sketch templates.

We leverage semantics extraction from HDL and architecture-independent sketch templates to build Lakeroad,[2] a new FPGA technology mapper based on program synthesis.

Lakeroad's prototype implementation automatically imports semantics for the LUTs, arith-

[2]Lakeroad is publicly available at `https://github.com/uwsampl/lakeroad`.

metic carry chains, and DSPs of the Xilinx UltraScale+, Lattice ECP5, Intel Cyclone 10 LP, and SOFA [173] FPGA architectures. The only additional user input to Lakeroad is a short architecture description that lists the target FPGA's primitives (fig. 6.1, "input 2"). Architecture descriptions only need to be written once per architecture, and Lakeroad pre-supplies architecture descriptions for the aforementioned architectures. With the automatically extracted primitive semantics and the user-provided architecture description, we demonstrate that Lakeroad is more complete than proprietary tools on a variety of microbenchmarks that are representative of program fragments implemented with DSPs during partial design mapping. In particular, Lakeroad maps up to 3.6× more microbenchmarks than state-of-the-art tools for Xilinx, Lattice, and Intel, and up to 30× more microbenchmarks than Yosys.

Part II of this dissertation makes the following key contributions:

- The novel application of program synthesis to produce a technology mapper—Lakeroad—that is more **correct, complete,** and **extensible** than state-of-the-art tools.
- A technique for applying **semantics extraction from HDL** to automatically generate models of hardware usable by formal automated reasoning tools.
- The concept of **architecture-independent sketch templates,** which capture common patterns in hardware design in an architecture-independent way, plus **primitive interfaces** and **architecture descriptions**, the abstractions underlying these templates.
- A formalization of the Lakeroad toolchain and an argument for its correctness and sketch-completeness.
- The first notion of **technology mapping completeness** for FPGA compilers.
- **Empirical comparisons** of Lakeroad and existing hardware synthesis tools to evaluate both their relative completeness and ease of extensibility.

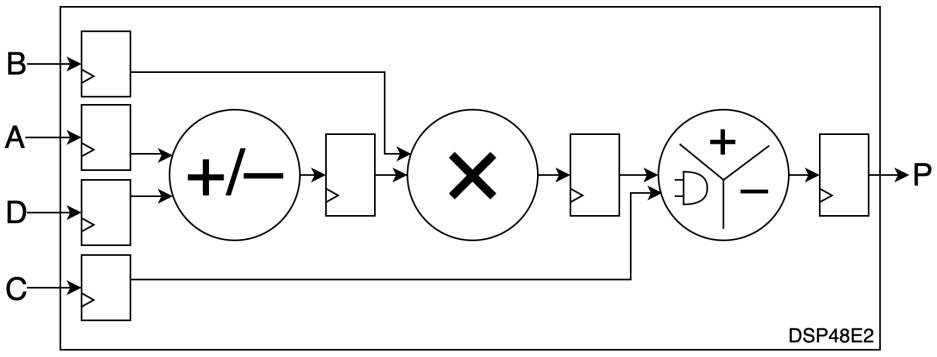
In the following sections, we walk through a real-world example using both existing tools and Lakeroad and highlight Lakeroad's design and key features (section 6.1); formalize Lakeroad and demonstrate its correctness (section 7.1); describe Lakeroad's implementation (section 7.2); and evaluate Lakeroad on its completeness of mapping, extensibility, and expressiveness (chapter 8) . chapter 10 discusses related work, and chapter 10 concludes.

## 6.1 Overview

We now walk through an example of how current FPGA technology mapping tools can fail a hardware designer (section 6.1.1) and how Lakeroad overcomes these limitations (section 6.1.2). In the process, we provide a high-level overview of Lakeroad's main components.

### 6.1.1 Compiling a Design to a DSP with Existing Tools

Consider the following scenario: A hardware designer is designing a large hardware block for the Xilinx UltraScale+ family of FPGAs. The designer is specifically aiming to use the UltraScale+'s specialized DSP48E2 units, which can implement combined multiplication, arithmetic, and logic operations, as captured in this simplified block diagram [199]:




The designer's hardware block involves the computation `(c-a)*b`, which the manual states is

implementable with a single DSP. In particular, suppose the design consists of four separate instances of the following computation:

```
for(i=0; i<4; i++) begin
 r[i] <= (c[i] - a[i]) * b[i];
end
```

It would be reasonable for the designer to expect the design to use a total of four DSPs.

***Current tools fail.*** After compiling the design with existing tools, the designer is frustrated to find that the compiler returns a design that uses more resources than anticipated. Instead of using the expected four DSPs, it uses eight—a 100% increase in resource consumption! **The compiler has thus failed to fully utilize the DSP**—it has not configured a DSP48E2 to implement `(c-a)*b` but has instead overflowed the computation onto an extra DSP. The designer now faces a choice: either accept the result or attempt to coax the compiler into returning a more optimal design.

***Coaxing the compiler, to no avail.*** Though many may choose to accept a less optimal result, this tenacious designer[3] tries to coax the compiler into giving the expected results by placing the computation of interest into a separate module:

```
module sub_mul(input clk, input [15:0] a, b, c,
                  output reg [15:0] out);
 assign out = (c-a)*b;
endmodule
```

This allows the designer to apply specific optimizations while mapping the module—a process we call *partial design mapping*. They attempt various strategies, including annotating the module with Xilinx's `use_dsp` Verilog attribute (to force the compiler to use a DSP where possible) and using a different synthesis directive (to apply a more resource-intensive synthesis procedure). **Despite these efforts, the compiler still cannot map the design to a single DSP,** instead using two

[3]This may not be purely a personal preference. For example, a hardware design simply may not fit on an FPGA without manual optimizations!

DSPs for the partial design. Again, the designer must decide: give up and accept suboptimal results, or press on?

***Manual compilation.*** The hardware designer presses on and now has only one option remaining: manually instantiating a DSP48E2 with the desired behavior. Skimming through the daunting 75-page DSP48E2's online user manual, the designer quickly discovers that configuring even the "pre-sub" `c-a` requires correctly setting multiple ports and parameters (`INMODE`, `AMULTSEL`, `BMULTSEL`, and `PREADDINSEL`), whose descriptions span 10+ pages and multiple tables. Correctly configuring the subsequent multiplier and logic unit proves even more difficult and time-consuming. After configuring the computational units, the designer must still manually ensure correct pipelining of the 10+ pipeline registers. After hours of frustration, a configuration is found that seems to work, which the designer inserts into the design. Precious time has been wasted, most of which will need to be repeated to configure the DSP again. Making matters worse, **the designer has no formal guarantees about the correctness of this DSP configuration.** It may work in a few simulated test cases, but are there corner cases that have been missed?

### 6.1.2 Compiling a Design to a DSP with Lakeroad

Lakeroad can save hardware designers the great effort involved in manual DSP configuration while also providing correctness guarantees. Let us imagine how the designer in this example, frustrated by conventional tools, can instead proceed using Lakeroad during partial design mapping. After putting `sub_mul` into its own module, the designer calls Lakeroad from the command line:

```
$ lakeroad --template dsp \
           --arch-desc xilinx-ultrascale-plus.yml \
           sub_mul.v
```

The `lakeroad` command outputs `sub_mul_impl`, an implementation of `sub_mul` that uses a single UltraScale+ DSP48E2:

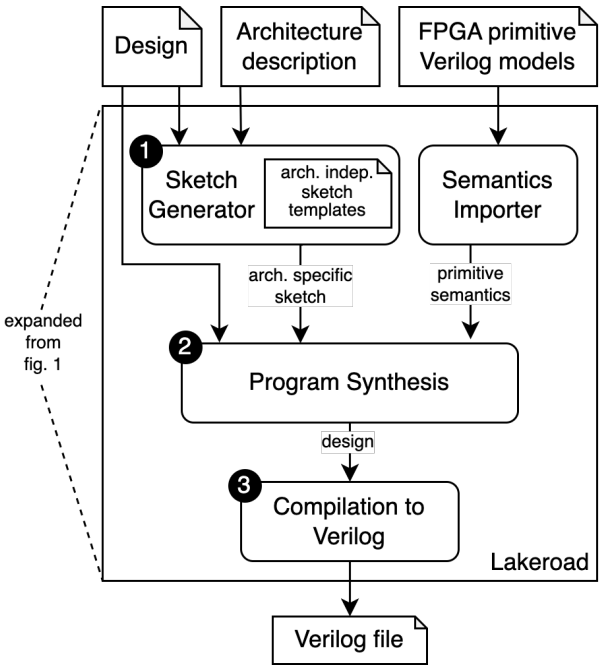



Figure 6.2: The components within Lakeroad.

```
module sub_mul_impl(input clk, input [15:0] a, b, c, output [15:0] out);
 DSP48E2 #(
  .ACASCREG(32'd0), .ADREG(32'd0), .ALUMODEREG(32'd0),
  .AMULTSEL("AD"), .AREG(32'd0), .AUTORESET_PATDET("NO_RESET"),
  // ...plus 30+ more parameters
 ) DSP48E2_0 (
   .A({ 14'h0000, a }), .ACIN(30'h00000000), .ALUMODE(4'hc),
   .B({ 2'h0, b }), .BCIN(18'h00000), .C({ 32'h00000000, c }),
   .CARRYCASCIN(1'h0), .CARRYIN(1'h0), .CARRYINSEL(3'h6),
   // ...plus 30+ more ports
 );
endmodule
```

Unlike current compilers, Lakeroad has produced an implementation using a single DSP48E2 by utilizing more of the DSP's features. Importantly, this compiled design is also formally guaranteed to implement the input `sub_mul` design.

How does Lakeroad provide *verified, more complete* support for the DSP48E2 over existing tools? At the core of Lakeroad's correctness and completeness is *sketch-guided program synthesis*,

a technique that begins with a program *sketch*, which captures a rough outline of a program and uses automated reasoning tools (e.g., SMT solvers) to fill in the sketch's *holes*. As shown in fig. 6.2, Lakeroad uses the following three-step process to generate an efficient and correct DSP48E2 implementation of the `sub_mul` design.

***Step 1: Generating a Sketch.*** In the `sub_mul` example, Lakeroad generates the following sketch, which we refer to as `sketch`:[4]

```
module sketch(input clk, input [15:0] a, b, c, output [15:0] out);
 DSP48E2 #(
  .ACASCREG(??), .ADREG(??), .ALUMODEREG(??), .AMULTSEL(??),
  .AREG(??), .AUTORESET_PATDET(??), ...
 ) DSP48E2_0 (
   .A({ 14'h0000, a }), .ACIN(??), .ALUMODE(??),
   .B({ 2'h0, b }), .BCIN(??), .C({ 32'h00000000, c }),
   .CARRYCASCIN(??),  .CARRYIN(??), .CARRYINSEL(??), ...
 );
endmodule
```

This sketch consists of a single DSP48E2 instance with *holes* (represented by `??`) serving as placeholders for most of its ports and parameters. It is easy to see the parallels between `sketch` and `sub_mul_impl`; `sketch` is simply `sub_mul_impl` with holes. But how does Lakeroad generate `sketch` in the first place?

To maximize portability across architectures, Lakeroad does not store sketches like `sketch` directly; instead, it *generates* sketches from architecture-independent **sketch templates.** Instead of storing the preceding UltraScale+–specific sketch, Lakeroad generates the sketch from the DSP sketch template, which the designer has chosen to use with the `--template dsp` flag. A simplified form of this template looks like the following:

```
module dsp_sketch_template(input clk,
                           input [n-1:0] a, b, c,
```

[4]Though this example is presented in a Verilog-like language, Lakeroad's sketches are actually encoded in a Racket DSL that resembles structural Verilog.

```
                              output [n-1:0] out);
 DSP dsp_instance(.clk(clk), .A(a), .B(b), .C(c), .D(d), .out(out));
endmodule
```

Sketch templates capture hardware design patterns that are common across FPGA architectures in an architecture-independent way. `dsp_sketch_template`, for example, captures a basic pattern, i.e., instantiating a single DSP. Lakeroad includes sketch templates of varying complexity, from the simplicity of the one above to the complexity of LUT-based multipliers. Though new sketch templates can be added easily, in most cases (as in this example) users can simply apply Lakeroad's provided templates.

To specialize `dsp_sketch_template` into `sketch`, Lakeroad translates the sketch template's generic `DSP` **primitive interface** into an UltraScale+–specific DSP48E2 using the UltraScale+ **architecture description.** The generic `DSP` module is an instance of a **primitive interface:** a Lakeroad-introduced abstraction that captures the similarities between primitives across diverse FPGA architectures. For example, Lakeroad's DSP primitive interface captures the facts that DSPs on all FPGA platforms generally have two to four data inputs (captured by `A`–`D`; note that our example doesn't use input `D`) and a clock input (captured by `clk`). To convert the sketch template's DSP primitive interface instance into a DSP48E2, Lakeroad utilizes the Xilinx UltraScale+ architecture description, which the designer has pointed to with the `--arch-desc xilinx-ultrascale-plus.yml` flag. An **architecture description** specifies how Lakeroad's various primitive interfaces are implemented for a given architecture. The following simplified snippet of the UltraScale+ architecture description, for example, tells Lakeroad that, when generating a sketch for UltraScale+, instances of the DSP primitive interface should be implemented with a DSP48E2:

```
- interface: {name: DSP, params: { out-width: 48, a-width: 30, ...}}
  holes: [?ACASCREG, ?ADREG, ?ALUMODEREG, ?AREG, ...]
  implementation:
```

```
module: DSP48E2
ports: [{ name: A, bitwidth: 30, value: A }, ...]
parameters: [{ name: ACASCREG, value: ?ACASCREG }, ...]
outputs: { O : P }
```

Thus, while converting `dsp_sketch_template` into `sketch`, Lakeroad reads this architecture description and converts the single DSP instance into a DSP48E2, filling the ports and parameters with the concrete values and holes contained in the architecture description. Architecture descriptions are usually short (100-400 LoC) and written only once per FPGA architecture; Lakeroad already contains such descriptions for Xilinx UltraScale+, Xilinx 7-series, Lattice ECP5, Intel Cyclone 10 LP, and the open-source FPGA SOFA [173].

To generate a sketch, Lakeroad takes an architecture-independent sketch template and specializes it using an architecture description. Once the sketch is ready, the designer can move on to synthesis.

***Step 2: Program Synthesis.*** The next step fills in the holes to generate a complete, correct hardware design, which is done automatically using a technique called program synthesis. *Program synthesis* is the process of using automated reasoning tools (like SMT solvers) to generate correct programs by encoding program generation as a constraint solving problem. In our `sub_mul` example, Lakeroad, aided by Rosette [178, 179], generates a query like the following:[5]

$$\exists\ \text{ACASCREG}, \text{ADREG}, \ldots\ .\ \forall\ \text{inputs}.\ \texttt{sub\_mul}(\text{inputs}) = \texttt{sketch}(\text{inputs}, \text{ACASCREG}, \text{ADREG}, \ldots)$$

The query asks: are there concrete values for `ACASCREG`, `ADREG`, etc., that will make our sketch's behavior equivalent to the input design's behavior on all inputs? If the solver finds such values, Lakeroad can use them to fill the holes in the sketch and produce a compiled design. However, if Lakeroad tries to pass the preceding formula to an SMT solver, the solver will throw an error since the query is not expressed at a level it understands, viz., as equalities between bitvector

[5] We formalize this synthesis query and explain it precisely in section 7.1.

expressions, using simple Boolean or arithmetic operations. While it is conceivable that `sub_mul` could be converted to a bitvector expression since its core computation is already expressed as `(c-a)*b`, it is unclear how to express `sketch` as an expression over bitvectors. In particular, Lakeroad must express bitvector-level semantics for Xilinx's DSP48E2 primitive.

To generate bitvector-level semantics for complex FPGA primitives, Lakeroad introduces the concept of **semantics extraction**. Rather than requiring manual effort to encode the semantics of the underlying hardware, which is notoriously difficult even for experts [19], Lakeroad's key insight is that these challenges can be avoided altogether by extracting low-level semantics directly from vendor-supplied simulation and verification models. Lakeroad builds on internal passes in Yosys [197] to automatically extract solver-ready semantics from these vendor-provided HDL models, which we detail in section 7.2.4. For the `sub_mul` example, the DSP48E2's semantics have already been imported into Lakeroad. Semantics need to be imported only when adding support for a new architecture, i.e., about as infrequently as writing a new architecture description. In most cases, Lakeroad users can rely on already-imported semantics.

With the sketch generated and the DSP48E2's semantics imported, program synthesis can begin. Lakeroad utilizes Rosette to drive program synthesis, as detailed in section 7.1. In our example, Rosette returns a configuration for the DSP48E2. The last step, then, is to convert the compiled design to Verilog.

***Step 3: Compilation to Verilog.*** Compiling Lakeroad's internal representation into Verilog is a purely one-to-one syntactic mapping; no optimizations are done at this stage, reducing the likelihood that bugs could be inserted. In our example, the final Verilog produced results in the `sub_mul_impl` we saw at the start of section 6.1.2.

***In summary.*** Lakeroad delivered an implementation of the designer's `sub_mul` module, improving upon both state-of-the-art compilers and manual approaches in multiple ways. Lakeroad's implementation is significantly more resource-efficient than the state-of-the-art compiler's. Lakeroad

delivered its implementation in mere seconds, compared to the hours to days of work that manually instantiating a DSP might take. Lastly, Lakeroad's implementation is formally guaranteed to be correct. Meanwhile, Lakeroad did all of this while requiring no input from the user other than the Verilog to be compiled.

In the next chapter, we provide details on the concepts introduced in this overview. We begin by formalizing Lakeroad's compilation flow in section 7.1, covering sketches and program synthesis in detail. Then, in section 7.2, we give implementation details for sketch templates, primitive interfaces, semantics importing, and compilation to Verilog.

# Chapter 7

# Lakeroad

Now that we have motivated the need for a tool like Lakeroad, and have shown how it can improve a hardware designer's life in a simple example, in this chapter we provide the detailed information needed to reimplement Lakeroad. We begin by formalizing Lakeroad in section 7.1, providing a rigorous mathematical description of the entire compilation flow. We then fill in implementation details not covered by formalization in section 7.2.

## 7.1 Formalization

$$
\begin{array}{lll}
\text{Prog} & ::= \langle\ \text{Id}, \langle \text{Id}, \text{Node}\rangle{*}\rangle & \\
\text{Node} & ::= \text{BV } b \mid \text{Var } x & \\
& \mid \text{OP } op\ \text{Id}^{*} & \\
& \mid \textbf{Reg}\ \text{Id}\ (\text{BV } b) & \\
& \mid \textbf{Prim}\ \text{binds Prog} & \\
& \mid \blacksquare_x &
\end{array}
$$

| | | | |
|---|---|---|---|
| Id | $id \in \mathbb{N}$ | | |
| Bitvectors | $b \in \mathbb{BV}$ | | |
| Variables | $x \in \text{LegalVarNames}$ | Wire op | $\text{OP}_w = \{\texttt{concat}, \texttt{extract}, \ldots\}$ |
| Operators | $op \in \text{OP}_{bv} \cup \text{OP}_w$ | Non-wire op | $\text{OP}_{bv} = \{+, -, \times, \ldots\}$ |
| binds | $bs \in (\text{Variables} \rightharpoonup \text{Id})$ | | |

Figure 7.1: Syntax of $\mathcal{L}_{\text{LR}}$. $\blacksquare_x$ is a syntactic hole, labeled with variable $x$. $A \rightharpoonup B$ denotes the set of partial functions from $A$ to $B$.

We claim that, through the use of program synthesis, we can improve the **CORRECTNESS** of FPGA compilers. This section presents our core argument towards this point by formalizing Lakeroad's semantics and arguing for its correctness.

We now formalize Lakeroad with functions $f_{\text{LR}}$ and $f^*_{\text{LR}}$, and use these models to argue for the correctness and partial completeness of Lakeroad. We first define $f_{\text{LR}}$ (section 7.1.1) and then motivate and define the language $\mathcal{L}_{\text{LR}}$, specify its syntax and semantics, and define behavioral ($\mathcal{L}_{\text{BEH}}$), structural ($\mathcal{L}_{\text{STRUCT}}$), and sketch ($\mathcal{L}_{\text{SKETCH}}$) sublanguages (section 7.1.2). We next explain the underlying queries Lakeroad uses to synthesize hardware programs that meet the desired specification (section 7.1.3). We demonstrate the correctness and partial completeness of $f_{\text{LR}}$, enumerate our Trusted Computing Base (section 7.1.4) and extend $f_{\text{LR}}$ to $f^*_{\text{LR}}$, which ensures the generated program's semantics matches the design over multiple timesteps (section 7.1.5). Finally, we highlight potential future applications that could be built on this section's formalization (section 7.1.6).

### 7.1.1 The Lakeroad Function $f_{\text{LR}}$

We model the execution of Lakeroad with the partial function

$$f_{\text{LR}} : \textsc{Sketch} \times \mathcal{L}_{\text{BEH}} \times \mathsf{Time} \rightharpoonup \mathcal{L}_{\text{STRUCT}},$$

where $f_{\text{LR}}(\Psi, d, t)$ invokes Rosette to synthesize a $t$-cycle implementation of behavioral design $d$ using sketch $\Psi$ to guide the search, where a $t$-cycle implementation of $d$ is a program that is equivalent to $d$ at clock cycle $t$. By not requiring program equivalence before clock cycle $t$ we allow the synthesized program's semantics to differ from the design during an initialization period (e.g., as the pipeline is being filled). To get guarantees beyond a single point in time $t$, we generalize $f_{\text{LR}}$ to $f^*_{\text{LR}}$, which synthesizes a program that is equivalent to the design from time $t$ to

$t+n$. We formalize a sketch $\Psi \in$ Sketch as a tuple $(\psi, h)$, where $\psi$ is a program in $\mathcal{L}_{\text{SKETCH}}$ and $h$ is a map from the holes in $\psi$ to a finite set of valid hole-free nodes in $\mathcal{L}_{\text{STRUCT}}$ that can be used to fill the mapped hole. This mapping $h$ is handled implicitly by Rosette's `choose` and `hole` constructs and need not be explicitly specified by the sketch writer. We write $f_{\text{LR}}(\Psi, d, t) = p$ to indicate that synthesis succeeded and produced Lakeroad program $p$. However, it is possible that sketch $\Psi$ cannot implement $d$, in which case the synthesis fails (i.e., returns UNSAT) and $f_{\text{LR}}$ does not return anything. Design $d$ belongs to $\mathcal{L}_{\text{LR}}$'s behavioral fragment, $\mathcal{L}_{\text{BEH}}$ (see section 7.1.2). When $t = 0$, $f_{\text{LR}}$ synthesizes a *combinational design*; when $t > 0$, $f_{\text{LR}}$ synthesizes a *sequential design* over $t$ clock cycles. The rest of this section considers sequential design synthesis since its combinational counterpart is a special case covered by our general approach.

### 7.1.2 Defining $\mathcal{L}_{\text{LR}}$

Lakeroad uses the $\mathcal{L}_{\text{LR}}$ language to translate behavioral HDL programs to structural, hardware-specific HDL programs. To facilitate this translation, we designed $\mathcal{L}_{\text{LR}}$ to satisfy the following properties:

**P1**. *Easy translation to/from HDLs:* we must be able to translate designs from a behavioral HDL to $\mathcal{L}_{\text{LR}}$ and translate synthesized implementations to a structural HDL.

**P2**. *Support parallel stateful execution:* FPGA designs consist of potentially stateful elements executing in parallel. $\mathcal{L}_{\text{LR}}$ must allow unambiguous parallel execution.

**P3**. *Support graph-like program structures:* An FPGA component's outputs can be wired to multiple other components, including back to itself. This means that FPGA programs can form arbitrary graphs, and $\mathcal{L}_{\text{LR}}$ must be able to express this.

**P4**. *Support for sequential designs:* $\mathcal{L}_{\text{LR}}$ must handle designs that run over multiple clock cycles.

**P5**. *Support for different architectures:* $\mathcal{L}_{\text{LR}}$ must handle FPGA components from different architectures.

We describe how $\mathcal{L}_{\text{LR}}$ satisfies P1-P5 when we define its syntax and semantics in the following subsections.

### $\mathcal{L}_{\text{LR}}$'s Syntax

Figure 7.1 shows the $\mathcal{L}_{\text{LR}}$ syntax. An $\mathcal{L}_{\text{LR}}$ program Prog consists of a root node ID and a graph of nodes, each of which is referred to by its ID. A *node* can be a constant bitvector, input variable, combinational (pure) operator, sequential (stateful) register, primitive, or hole. Given a program $p = (r, \langle id_1, node_1\rangle \ldots \langle id_n, node_n\rangle)$, we use the notation $p.root = r$, $p.ids = \{id_1, \ldots, id_n\}$, and $p[id_i] = node_i$. We define the free variables of a program $p.fv = \{x_i\}$ as the set of variable names occurring in $p$'s nodes of the form (Var $x_i$).[1] Finally, we use the notation $p.all\_ids$ for $p.ids$ together with $p'.all\_ids$ of any subprogram $p'$ of $p$ ($p'$ is a subprogram of $p$ if $\exists j, node_j =$ **Prim** $bs\ p'$).

Given a node $n$, we specify its inputs with the following function:

$$\textsc{inputs}(\text{BV } b) = \{\},$$

$$\textsc{inputs}(\text{Var } x) = \{\},$$

$$\textsc{inputs}(\text{OP } op\ i_1 \ldots i_n) = \{i_1, \ldots, i_n\}$$

$$\textsc{inputs}(\textbf{Reg}\ i\ b_{init}) = \{i\}$$

$$\textsc{inputs}(\textbf{Prim}\ bs\ p') = \{bs[x] \mid x \in \text{domain}(bs)\}$$

Note that we use $A \rightharpoonup B$ to denote the set of partial functions from $A$ to $B$; given $bs \in A \rightharpoonup B$, we write domain($bs$) to denote the set of $x \in A$ s.t. $bs[x]$ is defined.

A program $p$ is well-formed if and only if all the following conditions hold:

[1] Note that this does not include variables of sub-programs occurring recursively inside of **Prim** nodes.

**W1.** $p.root \in p.ids$;

**W2.** All ids are unique and distinct. (i.e. for any sub-program $p'$, $p.ids$ and $p'.all\_ids$ are disjoint, and for any two sub-programs $p'$ and $p''$, $p'.all\_ids$ is disjoint from $p''.all\_ids$.)

**W3.** The inputs of all nodes in $p$ are ids of other nodes in $p$: $\forall id \in p.ids$, $\text{inputs}(p[id]) \subseteq p.ids$;

**W4.** All primitive nodes contain well-formed programs;

**W5.** All primitive nodes bind exactly their free variables; i.e., for **`Prim`** $bs$ $p'$, $\text{domain}(bs) = p'.fv$; and

**W6.** Program $p$ is free of combinational loops (formalized below in property 1).

**Property 1 (Free of Combinational Loops)** *Formally, a program $p$ is free of combinational loops if there exists a function $w : p.all\_ids \to \mathbb{N}$, that satisfies the following properties (collectively "monotonicity"):*

1. *If $p[id] =$ **`Reg`** _ _, then $w(id) = 0$;*
2. *If $p[id] =$ **`Prim`** $bs$ $p'$, then $w(id) > w(p'.root)$;*
3. *if $p[id] =$ **`Prim`** $bs$ $p'$ and $p'[id'] = Var\ x$,*
   *then $w(id') > w(bs[x])$; and*
4. *Otherwise (e.g., $p[id] = \text{OP}\ op\ ids^*$),*
   *if $id' \in$ INPUTS$(p[id])$, then $w(id) > w(id')$.*

The function $w$ acts as a witness to the absence of combinational loops because it is impossible to define a strictly monotonic function without acyclicity. We consider only well-formed $\mathcal{L}_{\text{LR}}$ programs.

BV, Var, and OP nodes encode bitvectors, variables, and operators.

**Reg** $i_{data}$ $b_{init}$ nodes let $\mathcal{L}_{\text{LR}}$ implement sequential designs (P4). $i_{data}$ is the register's data input, which updates the stored value at the positive edge of each clock cycle, and $b_{init}$ is the register's initialization value.

**Prim** *bs* $p$ nodes let $\mathcal{L}_{\text{LR}}$ programs use hardware-specific components from different architectures (P5). The *bs* component is a *variable map*, mapping Vars to input Ids. The $p$ component is an $\mathcal{L}_{\text{LR}}$ program that defines the semantics of the hardware primitive. A **Prim** node also carries some metadata used during compilation to a structural HDL, which we omit for clarity.

$\mathcal{L}_{\text{BEH}}$ is the concrete *behavioral* fragment of $\mathcal{L}_{\text{LR}}$ used for writing specifications; it is formed by excluding **Prim** nodes and holes from $\mathcal{L}_{\text{LR}}$.

$\mathcal{L}_{\text{STRUCT}}$ is the concrete *structural* fragment of $\mathcal{L}_{\text{LR}}$ used for lowering $\mathcal{L}_{\text{LR}}$ to structural HDLs; it is formed by excluding **Reg** nodes, OP nodes, and holes from $\mathcal{L}_{\text{LR}}$, with the following exception: the $p$ term in **Prim** *bs* $p$ must always be from the $\mathcal{L}_{\text{BEH}}$ since it is used to specify the semantics of the **Prim** node to the synthesis engine. The behavioral node $p$ is not used during compilation to HDL, and this behavioral expression does not propagate to the structural HDL output.

$\mathcal{L}_{\text{SKETCH}}$ is another sublanguage of $\mathcal{L}_{\text{LR}}$ that is $\mathcal{L}_{\text{STRUCT}}$ but also including holes. Let $s$ be a program in $\mathcal{L}_{\text{SKETCH}}$ with holes $\blacksquare_{x_1}, \ldots, \blacksquare_{x_k}$. These holes can be *filled* with nodes $n_1, \ldots, n_k$ in $\mathcal{L}_{\text{STRUCT}}$ by replacing each hole $\blacksquare_{x_i}$ with its corresponding node $n_i$ to obtain a complete $\mathcal{L}_{\text{STRUCT}}$ program, denoted by $s[\blacksquare_{x_1} \mapsto n_1, \ldots]$.

The simplicity of this syntax makes translating to and from HDLs straightforward (P1). section 7.2 describes how Lakeroad implements the translations to and from HDLs.

### $\mathcal{L}_{\text{LR}}$'s Semantics

Before discussing the formal semantics of $\mathcal{L}_{\text{LR}}$, we present key definitions. We assume a *bitvector type* and, for simplicity, we elide bitvector widths. We represent *time* as a natural number. A

Time $t \in \mathbb{N}$ Env $e \in (\text{Var} \rightharpoonup \text{Time} \to \text{BV})$

$$
\begin{aligned}
&\textsc{Interp} : \text{Prog} \to \text{Env} \to \text{Time} \to \text{Node} \to \text{BV} \\
&\textsc{Interp}\ p\ e\ t\ (\text{BV}\ b) = b \\
&\textsc{Interp}\ p\ e\ t\ (\text{Var}\ x) = e\ x\ t \\
&\textsc{Interp}\ p\ e\ 0\ (\textbf{Reg}\ \_\ init) = init \\
&\textsc{Interp}\ p\ e\ (t+1)\ (\textbf{Reg}\ id\ \_) = \textsc{Interp}\ p\ e\ t\ p[id] \\
&\textsc{Interp}\ p\ e\ t\ (\text{OP op}\ ids) = [\![op]\!]\ (\text{map}\ (\lambda id\ .\ \textsc{Interp}\ p\ e\ t\ p[id])\ ids) \\
&\textsc{Interp}\ p\ e\ t\ (\textbf{Prim}\ bs\ p') = \\
&\quad \text{let } e' = \lambda x, t'\ .\ \textsc{Interp}\ p\ e\ t'\ (p[bs\ x]) \text{ in} \\
&\quad \textsc{Interp}\ p'\ e'\ t\ p'[p'.root]
\end{aligned}
$$

Figure 7.2: Lakeroad's semantics as pseudocode.

*stream* is a function from Time to bitvectors. An *environment* is a map from variable names to streams.

We give the semantics for $\mathcal{L}_{\text{LR}}$ as an interpreter in fig. 7.2. We define the function Interp to interpret a program $p$ in environment $e$ at time $t$ and node $n$. We do not define semantics for holes, as they are intended to be replaced by other constructs with well-defined semantics.

Most of the rules are straightforward. A bitvector BV $b$ evaluates to its backing bitvector value $b$. A variable node Var $x$ in an environment $e$ at time $t$ evaluates to the value returned by the stream associated with $x$ in $e$ at time $t$; using function notation, this is denoted by $e\ x\ t$. A $k$-ary operator node OP $op\ i_1 \ldots i_k$ recursively interprets each operand in the current environment at the current time and then applies $op$'s semantics, denoted $[\![op]\!]$, to the resulting values. A register **Reg** $id\ b_{init}$ has two cases depending on the current time: at time $t = 0$, a register evaluates to its initial bitvector value $b_{init}$; at nonzero times $t + 1$, a register evaluates to the value produced by the input $i$ at the *previous* timestep $t$. A primitive **Prim** $bs\ p'$ in environment $e$ at time $t$ is evaluated by interpreting the program $p'$ under the fresh environment $e'$ formed by the binding map $bs$.

### 7.1.3 Program Synthesis

$f_{\text{LR}}$ performs sketch-based program synthesis [166]. Operationally, we implement the INTERP function from Figure 7.2 in Rosette, a solver-aided host language [179]. Let sketch $\Psi = (\psi, h) \in$ SKETCH, where $\psi \in \mathcal{L}_{\text{SKETCH}}$ has holes $\blacksquare_{x_i}$ and $h$ maps $\psi$'s holes to the set of structural nodes that can legally fill the mapped hole. Given a design $d$, we query Rosette if there are nodes $n_1, n_2, \ldots n_k$ such that $n_i \in h[\blacksquare_{x_i}]$ and $p = \Psi[\blacksquare_{x_1} \mapsto n_1, \ldots]$ is well-formed and equivalent to $d$ (i.e., we ask Rosette to fill each hole with a node associated with the node in $h$). Program equivalence between well-formed programs $p$ and $d$ at time $t$, written $p \cong_t d$, is defined as

$$
\begin{aligned}
&p.fv = d.fv \wedge \\
&\forall e \; s.t. \; \text{domain}(e) = p.fv, \\
&\quad \textsc{Interp}\ p\ e\ t\ p.root = \textsc{Interp}\ d\ e\ t\ d.root.
\end{aligned}
$$

In section 7.1.5, we use bounded model checking to extend $f_{\text{LR}}$'s guarantees beyond the single timestep at clock cycle $t$.

### 7.1.4 Correctness and Completeness of $f_{\text{LR}}$

Recall that the synthesis function $f_{\text{LR}}$ is partial. We say that $f_{\text{LR}}$ is *correct* if it returns a program $f_{\text{LR}}(\Psi, d, t) = p$ where $p$ is a well-formed completion of $\Psi = (\psi, h)$, meaning $p = \Psi[\blacksquare_{x_1} \mapsto n_1, \ldots]$ such that $n_i \in h[\blacksquare_i]$ for all $i$ and $p \cong_t d$.

Furthermore, we say that $f_{\text{LR}}$ is *sketch-complete* if $f_{\text{LR}}(\Psi, d, t)$ is defined whenever there exists a well-formed completion $p$ of $\Psi$ such that $p \cong_t d$. That is, synthesis is correct if it never returns an erroneous result and sketch-complete if it returns a correct result whenever one exists.

We have implemented $f_{\text{LR}}$ with Rosette (see section 7.1.3), which guarantees our system is

correct and complete under the following assumptions:

1. Correctness of Rosette and underlying SMT solvers;
2. That our encoding of Lakeroad is bug-free;
3. That the lowering of Interp to SMT formulas by Rosette always terminates. This is possible when partial evaluation of Interp on arguments $p$, $t$ and $n$ terminates (independently of the value of $e$).

**Lemma 7.1** *Let $p$ be a well-formed program, $e$ an environment, $t$ a Time, and $n$ be a node belonging to $p$. Then Interp is primitive recursive (i.e. terminates) in the arguments $p$, $t$, and $n$.*

**Proof 7.1 (Proof of Lemma 7.1)** *Recall that a function $f(x, y, z)$ is primitive recursive in arguments $x$ and $y$ (under a lexicographic ordering) if in the definition of $f$ every recursive call $f(x', y', z')$ is made with values $(x', y')$ such that $x' < x$ or $x' = x \wedge y' < y$. If $x$ and $y$ are drawn from the natural numbers (or another well-ordered set), then the recursion is guaranteed to terminate.*

*Under what order is Interp primitive recursive? Because our program is well-formed, it must be free of combinational loops (see property 1). Formally, this means we have an acyclicity witness function $w : p.all\_ids \to \mathbb{N}$ that monotonically increases in the direction of dataflow in our circuit. Each node $n$ argument passed to Interp has an Id that is unique and distinct from the Ids used in $p$ or any of $p$'s subprograms (**W2**); we denote this Id as $id_n$. We can associate each $n$ argument to a recursive call of Interp with a number $w(id_n)$. We claim that Interp is primitive recursive under the lexicographic ordering on $(t, w(id_n))$.*

*To prove this claim we need to demonstrate that if Interp with time and node arguments $t'$ and $n'$ makes a recursive call to Interp with time and node arguments $t''$ and $n''$, then the following condition holds:*

$$t'' < t' \vee (t'' = t' \wedge w(id_{n''}) < w(id_{n'}))\,. \tag{7.1}$$

*To do this it suffices to examine each case of* INTERP*'s definition.*

*When $n'$ is a* BV *constant,* INTERP *makes no recursive calls, and the condition in eq.* (7.1) *holds vacuously.*

*When $n'$ is a* **`Reg`** *node* INTERP *either terminates (when $t' = 0$) or makes a recursive call with time value $t'' = t' - 1$, maintaining the condition in eq.* (7.1)*.*

*When $n'$ is an operator node,* INTERP *recursively interprets the operands with time arguments $t'' = t'$. However, each operand's id $id''$ belongs to* INPUTS*$(n')$, and, by property 1, $w(id_{n'}) > w(id'')$, so our condition holds.*

*This leaves us with the less obvious cases in which $n'$ is either a* **`Prim`** *or* Var*, which work together in tandem. When $n' =$* **`Prim`** *bs $p'$,* INTERP *makes a recursive call with node argument $p'.root$ and time argument $t$. By property 1, $w(p'.root) < w(id_{n'})$, and the condition in eq.* (7.1) *holds.* INTERP *also defines a new environment for execution of $p'$ via $\lambda$-abstraction, and this in turn will recursively invoke* INTERP*. These environments are only invoked by the rule for variables, which we handle presently.*

*When $n' =$* Var *$x$, the environment is invoked on variable $x$. Here, there are two possible cases. First, we are interpreting the top-level program $p$. As this is the initial, top-level environment, there is no further recursion. Second, we are interpreting a sub-program $p'$ and $e'\ x\ t =$* INTERP *$p\ e\ t\ (p[bs\ x])$ is actually a recursive call into the program $p$ one level up, with its environment $e$. In this latter case, note that $w$ is defined such that $w(id_{p[bs\ x]}) = w(bs\ x) < w(id_{\text{Var}\ x})$ (item 3 of property 1), satisfying our property. All cases are complete.*

From this, we conclude that all possible substitutions for $\Psi$ are attempted, and $f_{\text{LR}}$ is sketch-complete.

**Trusted Computing Base.** The *trusted computing base* (TCB) of a system is the set of components it assumes to be correct [114]. A bug anywhere in the TCB could cause the guarantees made

by that system to be violated. Lakeroad's TCB includes: Rosette and the underlying SAT/SMT solvers that Rosette queries (Bitwuzla, cvc5, Yices2, and STP); the internal Yosys passes Lakeroad uses to extract primitive semantics and translate design specifications from behavioral Verilog into $\mathcal{L}_{\text{BEH}}$; the semantics for $\mathcal{L}_{\text{LR}}$, which we assume conservatively models non-cyclic (DAG) designs; our code to translate from the $\mathcal{L}_{\text{STRUCT}}$ to structural Verilog; and the vendor-provided Verilog simulation models for FPGA primitives. Each TCB component has also been thoroughly tested, as described in chapter 8. Importantly, sketches and sketch generation are *not* in Lakeroad's TCB: even if there were a bug in Lakeroad's sketch-related components, it would not violate Lakeroad's correctness guarantees.

### 7.1.5 Multiple Clock Cycle Guarantees with $f^*_{\text{LR}}$

The preceding completeness and correctness properties for $f_{\text{LR}}$ guarantee that running the synthesized program $p$ and the design $d$ for $t$ clock cycles produces the same output. To extend this guarantee, Lakeroad supports a form of bounded model checking, where synthesis ensures that $p$ is semantically equivalent to $d$ for $c$ additional clock cycles starting at time $t$. We formalize this with the function $f^*_{\text{LR}}$, which takes a sketch $\Psi$, a behavioral design $d$, a number of clock cycles $t$, and a model checking time bound $c \geq 0$ and returns an implementation $p \in \mathcal{L}_{\text{STRUCT}}$ that is equivalent to $d$ at time steps $t, t+1, \ldots, t+c$.

Our correctness and completeness guarantees are similar to those for $f_{\text{LR}}$:

$$
\begin{aligned}
&p.fv = d.fv \wedge \\
&\forall e \; s.t. \; \text{domain}(e) = p.fv, \\
&\quad \bigwedge_{i=t}^{i=t+c} \textsc{Interp}\ p\ e\ i\ p.root = \textsc{Interp}\ d\ e\ i\ d.root.
\end{aligned}
$$

### 7.1.6 Beyond Lakeroad

$\mathcal{L}_{\text{LR}}$, its semantics, and the synthesis approach we describe here are useful for applying program synthesis to other hardware design problems. For example, the synthesis problem detailed above could be "flipped" to decompile structural designs back to higher-level behavorial designs, i.e., synthesizing from $\mathcal{L}_{\text{STRUCT}}$ to an expression in $\mathcal{L}_{\text{BEH}}$. Such decompilation has seen recent interest for recovering equivalent but faster-to-simulate models and for porting models across different architectures [160]. As another example, the synthesis approach could be adapted to help port designs by synthesizing expressions in $\mathcal{L}_{\text{STRUCT}}$ that use one set of primitives on one architecture from other designs in $\mathcal{L}_{\text{STRUCT}}$ that use a different set of primitives from a different architecture. Thus, the formalization in this section transcends the particular challenges of FPGA technology and provides a reusable foundation for exploring a much broader range of hardware design challenges from a program synthesis perspective.

## 7.2 Implementation

Lakeroad is composed of approximately 13K lines of Racket plus approximately 58K lines of Racket automatically generated from vendor-supplied Verilog. Vendor-supplied Verilog was obtained from Lattice Diamond, Intel Quartus, and Xilinx Vivado sources. We used Vivado version v2023.1, Quartus 22.1std.1 Build 917 02/14/2023 SC Lite Edition, Diamond version 3.12, Yosys version 0.36+42 (commit `70d3531`), the cvc5 [17] and Yices2 [55, 54] solvers included in the 2023-08-06 release of `oss-cad-suite` from YosysHQ, the Bitwuzla solver at commit `b655bc0` [129], the STP solver at commit `0510509a`, Racket version 8.9 [62, 63], and Rosette version 4.1 [140].

```
implementations:
  - interface: { name: LUT, num_inputs: 4 }
    internal_data: { sram: 16 }
    modules:
      - module_name: frac_lut4
        filepath: SOFA/frac_lut4.v
        ports:
          - { name: in, direction: in, width: 4,
              value: (concat I3 I2 I1 I0) }
          - { name: mode, direction: in,
              width: 1, value: (bv 0 1) }
          - { name: lut4_out, direction: out,
              width: 1 }
        parameters: [{ name: sram, value: sram }]
    outputs: { O: lut4_out }
```

Figure 7.3: SOFA architecture description.

### 7.2.1 Primitive Interfaces

As described in section 6.1, *primitive interfaces* describe abstract versions of common FPGA primitives, which allow sketch templates to be architecture-independent. To date, Lakeroad declares primitive interfaces for $n$-input LUTs, $w$-width carry chains, $n$-input muxes, and DSPs with up to four data inputs and one clock input. The next section includes a concrete example of Lakeroad's LUT4 primitive interface.

### 7.2.2 Architecture Descriptions

As described in section 6.1, *architecture descriptions* convey the information required to convert each instance of a primitive interface into the corresponding architecture-specific module, which occurs while converting sketch templates into sketches. The architecture description is the only additional input that may be required from a user to support a new architecture; it is a one-time effort that is reusable for any designs in an architecture. Architecture descriptions are simply lists (provided as YAML files) of the primitive interfaces that an architecture implements, but, crucially,

also include architecture-specific port and parameter values in a map called `internal_data`. Values in this map become symbolic values solvable by the SMT solver. Additional constraints can also be specified in the architecture description to rule out invalid configurations and minimize the solver's search space.

As an example, fig. 7.3 shows the architecture description for the SOFA [173] FPGA architecture. The description contains a single primitive interface implementation, i.e., LUT4. Lakeroad's LUT4 primitive interface standardizes the names of a LUT4's inputs and outputs, naming the inputs `I0` through `I3` and the output `O`. The SOFA implementation of the LUT4 primitive interface uses the SOFA-specific `frac_lut4` primitive. Primitive interface inputs `I0` through `I3` are mapped to the actual input port of the `frac_lut4`, named `in`. Likewise, the `frac_lut4` output `lut4_out` is mapped to the primitive interface output `O`. The `internal_data` field declares `sram`, the LUT's 16-bit internal memory, as an architecture-specific detail to be solved during synthesis.

If a sketch template uses a primitive interface not included in the architecture description (e.g., SOFA does not implement carries), Lakeroad may still be able to implement the primitive interface based on primitive interfaces the architecture *does* implement. To date, Lakeroad can implement any mux with LUTs, a larger LUT from smaller LUTs, a smaller LUT from a larger LUT, a carry from LUTs, and a smaller DSP from a larger DSP; it handles these conversions during sketch generation.

### 7.2.3 Sketch Templates, Sketches, and Sketch Generation

As described in section 6.1, Lakeroad captures common FPGA implementation patterns in reusable, architecture-independent *sketch templates.* Thus far, we have described only the relatively simple `dsp` sketch template, which instantiates a DSP. As a more complex example of capturing common FPGA implementation patterns, consider the `bitwise-with-carry` sketch template, which uses

*n* LUTs and a carry chain to implement designs such as addition or subtraction. Lakeroad currently provides 5 sketch templates: `dsp`, `bitwise`, `bitwise-with-carry`, `comparison` (LUT- and carry-based arithmetic comparison), and `multiplication` (LUT-based multiplication).

The process of converting sketch templates to sketches is implemented as described in section 6.1 and section 7.2.2. Lakeroad iterates over every primitive interface instance in the sketch and replaces it with the concrete primitive in accordance with the architecture's architecture description. If the architecture description does not implement the requested primitive interface, Lakeroad checks whether it can implement the primitive interface with other implemented interfaces (e.g., implementing a smaller LUT with a larger LUT) and raises an error otherwise.

Sketch templates and sketches alike are written in a domain-specific language (DSL) embedded into Rosette, whose implementation closely mirrors the syntax and semantics of $\mathcal{L}_{\text{LR}}$. The only significant difference is that the interpreter implementation does not use bitvector streams natively. Instead, each invocation of the interpreter represents a single timestep, and all intermediate values from the previous timestep are taken as input. Streams are then built up using multiple invocations of the interpreter.

### 7.2.4 Importing Semantics from Verilog Modules

Lakeroad uses Yosys [197] to convert Verilog modules into the `btor2` format [130] and then converts the resulting `btor2` to Rosette/Racket code.

Due to the semantics of the Verilog language and the internal implementation of Yosys, extracting semantics from Verilog modules may require the following manual modifications to accommodate semantics extraction and synthesis:

- As Yosys converts parameters from variables to constant values immediately upon module import, module parameters should be converted to ports to ensure they remain variables (and

thus solvable by the SMT solver). Note that not all parameters can always be converted to ports, meaning some parameters cannot be solved for.

- Strings should be converted to bitvectors.
- All registers should be initialized.
- All instances of `x` and `z` values should be converted to 2-state logic (0 or 1).

Note that these caveats apply only to our prototype implementation, not the general technique of semantics extraction from HDL. Once these manual modifications are made, the following series of Yosys passes can be used to convert the Verilog into suitable `btor2`: `prep; flatten; pmuxtree; opt_muxtree; clk2fflogic; prep; write_btor`.

We implement the translation from `btor2` to Rosette bitvector expressions as a 1:1 translation since both languages are simply operations over bitvectors.

### 7.2.5 Program Synthesis and Compilation to Verilog

We implement the synthesis procedure defined in section 7.1.4 with Rosette. Multiple clock cycle guarantees, as described in section 7.1.5, are implemented simply by making $c + 1$ total assertions, asserting the output of the input design and the sketch are equal after each of the $c + 1$ timesteps. We use a portfolio solving method, running Bitwuzla [128], cvc5 [17], Yices2 [55, 54], and STP [168] in parallel and using results from the first solver to terminate. To produce Verilog, Lakeroad compiles the program from its internal DSL to the JSON format defined by Yosys using a straightforward translation and then uses Yosys to output Verilog.

# Chapter 8

# Evaluation

We now evaluate Lakeroad in terms of completeness (related to **Optimizations**) and extensibility (related to **Devtime**). Note that we consider our formalization in section 7.1 as evidence for our **Correctness** claim, and thus we don't include an evaluation of correctness in this chapter. In the following experiments, we target four FPGA architectures: **Xilinx UltraScale+**, commonly used for large, high-performance workloads; **Lattice ECP5**, commonly used in low-power, low-cost scenarios; **Intel Cyclone 10 LP**, an FPGA designed for low-cost, high-volume use cases, and **SOFA** [173], a recent, open-source FPGA developed by the research community. We compare Lakeroad to existing technology mappers. For Xilinx Ultrascale+, Lattice ECP5, and Intel Cyclone 10 LP, we compare Lakeroad against both the open source toolchain Yosys [197] and the state-of-the-art, proprietary, closed source toolchains for each architecture.[1]The experiments were conducted on a system running Ubuntu 20.04.3 with an AMD EPYC 7702P 64-Core CPU. The resident set size of a single Lakeroad process did not exceed 300MB while running our evaluation. We use the software versions listed in section 7.2.

[1]Again, licensing restrictions prevent our naming the specific proprietary tools, but they are familiar, standard packages used by many hardware designers.

## 8.1 Lakeroad Completeness

The reliance of many technology mappers, including state-of-the-art tools, on hand-written patterns leads them to fail when attempting to map many workloads that *should* be mapped to a single DSP. In particular, the process of partial design mapping (illustrated in section 6.1) becomes a laborious endeavor because of this incompleteness: hardware designers hand-instantiate DSPs rather than rely on substandard automated tooling, repeating the work each time they identify a potential opportunity to use a DSP. Lakeroad's greater mapping completeness significantly reduces the burden on hardware designers during partial design mapping and marks the first step in automated mapping for full designs. We next evaluate how Lakeroad's program synthesis approach enables it to achieve greater completeness for these program fragments. In the context of this dissertation, this corresponds to providing evidence for our **Optimizations** claim: namely, through the use of more adaptable **Algorithms** (program synthesis) and more explicit **Models** (vendor-supplied simulation models), we can build a compiler, Lakeroad, which finds more **Optimizations** in the form of mappings to specialized primitives.

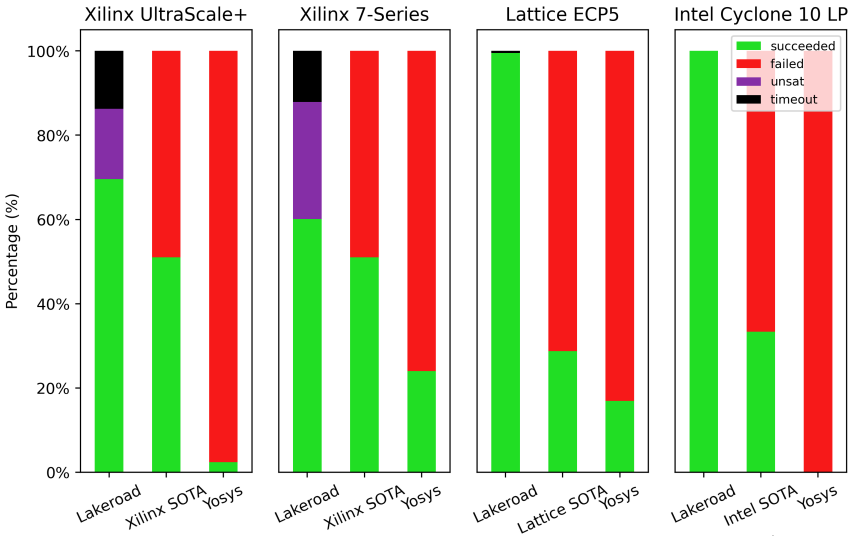
***Evaluation Setup.*** We highlight four particularly complex DSPs for the Xilinx Ultrascale+, Xilinx 7-series, Lattice ECP5, and Intel Cyclone 10 LP architectures: the Xilinx DSP48E2, Xilinx DSP48E1, Lattice ALU54A–MULT18X18C (a single DSP composed of two primitives), and Intel cyclone10lp_mac_mult. SOFA provides no DSP, and is not included in this part of the evaluation. For each architecture's DSP, we enumerate a large subset of the designs theoretically mappable to a single DSP according to its configuration manual. This microbenchmark set aims to capture the real-world designs which hardware designers would attempt to map to a platform's DSP. For each architecture, we compare Lakeroad to both the corresponding state-of-the-art toolchain for the architecture as well as to Yosys. For Xilinx Ultrascale+ and 7-series, the DSP48E2 (resp. DSP48E1)

configuration manual details the structure of designs mappable to the primitive. Our designs for Xilinx include all permutations of the design form $((a \pm b) * c) \pm d$, as well as designs of the forms $(a * b)$, $(a \pm b) * c$ and $((a * b) \pm c)$. We pipeline each of these workloads from zero to three stages and use bitwidths from 8 to 18 bits. For the DSP on Lattice, we similarly enumerate all designs of the form $(a * b) \odot c$, where $\odot \in \{\&, |, \oplus, \pm\}$, and of the form $(a * b)$. For each of these designs, we use zero to two stages and bitwidths from 8 to 18 bits. This results in 792 microbenchmarks for Xilinx UltraScale+, 396 for Lattice ECP5, and 66 for Intel Cyclone 10 LP. Though Lakeroad's output is correct by construction, we further validate its output by simulating each Lakeroad-compiled design over thousands of consecutive cycles using Verilator.

***Comparison to Existing Toolchains.*** As demonstrated in Figure 8.1 (top), Lakeroad maps 29× more designs than Yosys and 1.4× more designs than the proprietary, state-of-the-art toolchain on Xilinx Ultrascale+. On Lattice ECP5, Lakeroad maps 6.0× more designs than Yosys and 3.6× more designs than the proprietary, state-of-the-art toolchain. On Intel Cyclone 10 LP, Lakeroad successfully maps all designs: 3× more designs than the proprietary, state-of-the-art toolchain for Intel. Yosys fails to map a single design on Intel. State-of-the-art toolchains for all architectures fail to map more than half of the queried designs. Lakeroad times out on less than 20% of designs.[2] Note that Lakeroad returns "UNSAT" on a number of designs on Xilinx UltraScale+ and 7-series, i.e., Lakeroad claims there is *no* possible mapping to a DSP48E2/DSP48E1 for the requested workload. In all of these cases, both Xilinx SOTA and Yosys agree with Lakeroad and do not map the designs to a single DSP. We conclude that the set of designs we presented in *Evaluation Setup* must be overly broad; though the documentation implies that all of these designs are mappable to a single DSP, all three Xilinx synthesis tools surveyed indicate that they are indeed not mappable.

For timing, we compared the mapping time for each of the tools and report the results in

[2]We restricted Rosette synthesis time to 120 seconds, 40 seconds, and 20 seconds for Xilinx, Lattice, and Intel respectively, and marked failure past that (though bitvector synthesis problems are decidable).




| Tool | Median Time (s) | Min / Max Time (s) | |
|---|---|---|---|
| **Xilinx UltraScale+** | | | |
| Lakeroad | 14.99 | 2.99 | 127.70 |
| SOTA Xilinx | 261.61 | 227.82 | 598.67 |
| Yosys | 14.97 | 6.66 | 21.10 |
| **Xilinx 7-series** | | | |
| Lakeroad | 5.63 | 3.13 | 62.68 |
| SOTA Xilinx | 94.50 | 89.32 | 111.61 |
| Yosys | 6.99 | 5.64 | 9.39 |
| **Lattice ECP5** | | | |
| Lakeroad | 9.49 | 6.70 | 55.23 |
| SOTA Lattice | 2.32 | 0.95 | 4.52 |
| Yosys | 2.31 | 0.90 | 4.01 |
| **Intel Cyclone 10 LP** | | | |
| Lakeroad | 2.92 | 2.12 | 4.13 |
| SOTA Intel | 38.73 | 19.11 | 43.49 |
| Yosys | 0.96 | 0.48 | 1.88 |

Figure 8.1: Results of the completeness experiments described in section 8.1, measuring the completeness of technology mapping tools for DSPs on Xilinx UltraScale+, Xilinx 7-series, Lattice ECP5, and Intel Cyclone 10 LP, plus timing information. A single bar in the bar chart communicates, for a given FPGA architecture and technology mapper, the proportion of the microbenchmarks that the given technology mapper could map to a single DSP. In Lakeroad's case, experiments can either succeed (Lakeroad maps the microbenchmark to a single DSP), timeout, or return UNSAT. For the other tools, experiments can either succeed or fail (i.e., the tool returns a mapping, but the mapping uses more than a single DSP). There are a total of 792 experiments/microbenchmarks for Xilinx, 396 for Lattice, and 66 for Intel.

Figure 8.1 (bottom). The wide ranges for Lakeroad show that solver time for different program synthesis queries is highly variable. This is explored more deeply in fig. 8.2, which shows that most synthesis queries terminate quickly, with a long tail of slower queries. Note that the state-of-the-art technology mapper for Ultrascale+/7-series has a slow running time due to its long start-up process.

Regarding which solvers in the portfolio were most useful, of all terminating (success or UNSAT) Lakeroad experiments, Bitwuzla was the first to complete for 806 of them, STP for 595, Yices2 for 538, and cvc5 for 54.

***Discussion.*** Compared to Yosys, it is clear that Lakeroad provides more complete support for programmable DSPs. However, Lakeroad's greater completeness over Yosys is perhaps not surprising since Yosys is an open-source tool still under active development. Part of the appeal of the Yosys toolchain is the diversity of backends it can target; these results show that, if incorporated into Yosys, Lakeroad would further increase Yosys's flexibility and generality. Perhaps most surprising is that Lakeroad is more complete than specialized proprietary toolchains. Even the UNSAT results Lakeroad produces can be useful to designers since they indicate potential flaws in the documentation or vendor-provided semantics. In the context of a larger synthesis tool, Lakeroad would provide stronger guarantees for mapping modules of larger designs.

## 8.2 Lakeroad Extensibility and Expressiveness

In addition to being correct by construction (section 7.1) and more complete than existing FPGA technology mappers (section 8.1), Lakeroad can also easily extend to new FPGA architectures. Furthermore, automatic primitive semantics extraction from vendor-provided HDL simulation models enables Lakeroad to support diverse, highly configurable FPGA primitives. In the context of this dissertation, this corresponds to providing evidence for our **Devtime** claim: namely,

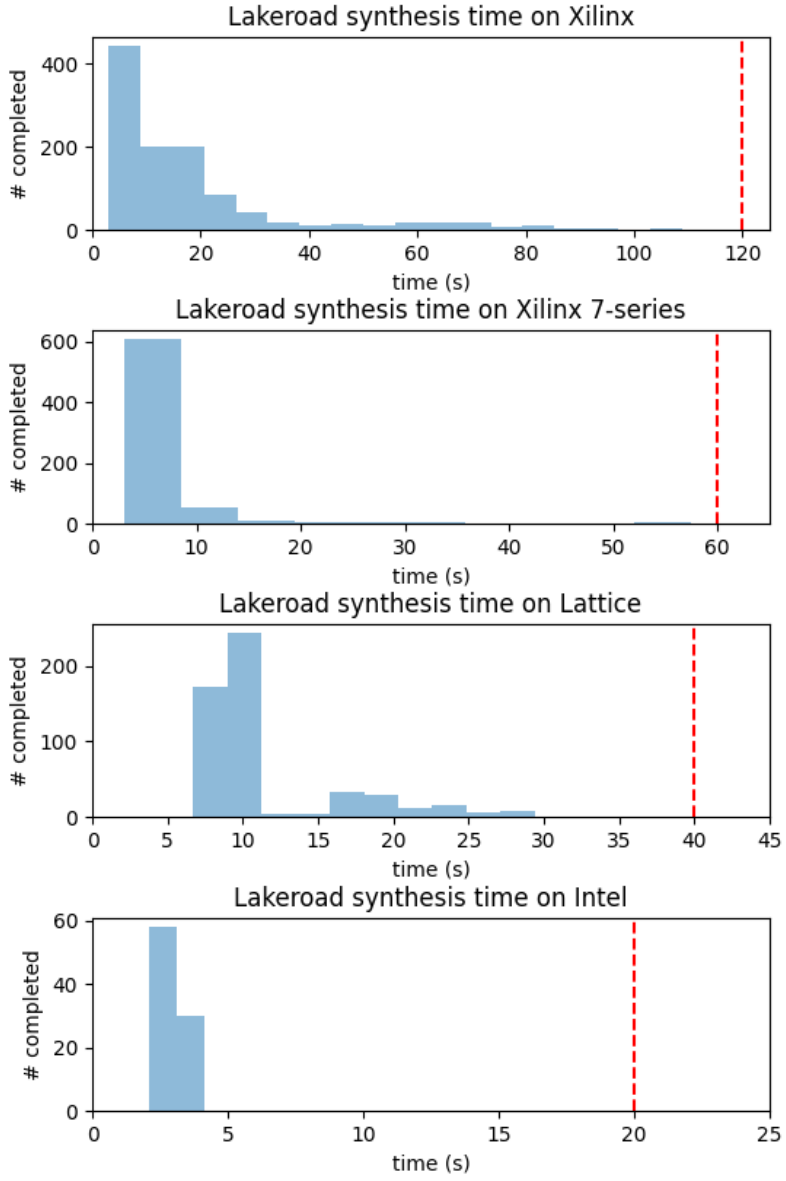



Figure 8.2: Histograms of Lakeroad program synthesis runtime for all terminating (success or UNSAT) Lakeroad experiments described in section 8.1, with timeout thresholds indicated with a vertical dotted red line.

through the use of more adaptable **Algorithms** (program synthesis) and more explicit **Models** (vendor-supplied simulation models), supporting new FPGA architectures in Lakeroad requires less **Devtime** compared to other tools.

The architecture descriptions vary in length from 20 to 240 source lines of code (SLoC). SOFA (20 SLoC) is the simplest, shown in full in fig. 7.3. The descriptions for Xilinx UltraScale+ (185 SLoC), Xilinx 7-series (174), Lattice (240 SLoC), and Intel (178 SLoC) are longer since those FPGA architectures provide a wider range of configurable primitives.

As a point of comparison, the open-source Yosys toolchain, which has roughly 200 contributors on GitHub, provides technology mapping for Xilinx UltraScale+ across over a dozen complex Verilog, C++, and Python files (about 1300 lines of code). We cannot provide similar numbers for state-of-the-art proprietary tools, but a developer of one such technology mapper shared that extending their tool to support new FPGA architectures was extremely difficult since it "spans millions of lines of low-level C." This is not surprising; Yosys aims to target a variety of vendor architectures, while proprietary tools have teams of engineers to extract better mapping (evident by Yosys' limitations in section 8.1). By contrast, Lakeroad supports diverse architectures and is easy to extend. Even if a user wants to target a completely new architecture that Lakeroad does not support, architecture-independent sketch templates allow reuse of previously implemented mapping strategies, and the user is only required to provide a few lines of high-level configuration for each primitive in the architecture description.

Table 8.1 further highlights Lakeroad's expressiveness, i.e., its ability to support a diverse range of configurable primitives by automatically extracting semantics from vendor-provided HDL simulation models. Lakeroad can import the semantics of large configurable primitives, such as the UltraScale+ DSP (896 lines of Verilog) or Lattice ECP5's ALU and multiplier units (1642 and 795 lines of Verilog, respectively). It is difficult and error-prone to manually formalize the full semantics for these primitives; partial support by ad hoc search procedures that rely on syntactic

Table 8.1: FPGA primitives imported automatically by Lakeroad from vendor-provided Verilog models, with number of source lines of code (excluding comments and empty lines) of the original Verilog models.

| FPGA | Primitive | Verilog SLoC |
|---|---|---|
| Xilinx Ultrascale+ | LUT6 | 88 |
| | CARRY8 | 23 |
| | DSP48E2 | 896 |
| Xilinx 7-series | DSP48E1 | 1129 |
| Lattice ECP5 | LUT2 | 5 |
| | LUT4 | 7 |
| | CCU2C | 60 |
| | ALU54A | 1642 |
| | MULT18X18C | 795 |
| Intel Cyclone 10 LP | cyclone10lp_mac_mult | 319 |
| SOFA | frac_lut4 | 69 |

pattern matching leads to missing many mapping opportunities, as shown in section 8.1.

# Chapter 9

# Future Work: Churchroad

I wanted to briefly mention an ongoing extension to Lakeroad. A fundamental limitation of solver-aided approaches like Lakeroad is problem size: the larger a constraint problem, the more likely solvers are to time out while trying to find a solution. In Lakeroad, this means that compiling larger hardware designs will present issues; we already saw in chapter 8 that Lakeroad's underlying solvers time out on some benchmarks. Furthermore, even some small tasks are notably difficult for solvers—namely, reasoning about multiplication [145, 23]. Scaling to large designs and compiling multipliers are core requirements for a hardware compiler, however. So how do we ensure that we can use Lakeroad on large designs?

In response, we have been developing Churchroad [163, section 4], which combines many of the techniques from this dissertation. Namely, we employ equality saturation—our algorithm of choice in part I. Using equality saturation, we can capture an entire design in an egraph. Once we have captured a full design, we can use equational reasoning via rewrites to achieve tasks that would otherwise be challenging for SMT solvers—for example, splitting a wide multiplier into multiple multipliers and adders. Similarly, we can use rewrites to discover places in the design where we might want to invoke Lakeroad. Consider the benchmarks we generated to evaluate

DSP mapping in in chapter 8—we could similarly generate patterns to identify potential places where DSPs could be used in larger designs. Once these locations are identified within the design, we can call Lakeroad as a subroutine, and if Lakeroad finds a mapping, we can then import it back into the egraph.

Churchroad has already shown promise as a method of scaling solver-aided hardware compilation. Beyond that, however, Churchroad also presents new strategies for leveraging the strengths of of SMT solvers and equality saturation.

# Chapter 10

# Background and Related Work

To the best of our knowledge, Lakeroad is the first work to apply the technique of program synthesis to FPGA technology mapping. Indeed, as noted by Sisco *et al.* [159], program synthesis has seldom been applied in the domain of hardware design although its underlying formal methods techniques are frequently used for the *formal verification* of hardware designs rather than compilation, as in Bluespec SystemVerilog [132], Kôika [22], and Kami [40]. Sisco *et al.* cite two examples of works that use program synthesis for hardware design, Verisketch [8] and Sketchilog [18], both of which apply program synthesis to produce HDL implementations from high-level designs. Other works use program synthesis to generate *software* that runs on low-powered hardware, like Chlorophyll [138], which targets extremely memory-constrained power-efficient processors, Chipmunk [67], which targets programmable network switches, and Diospyros [184],[1] which generates vectorized programs for standalone digital signal processors (more powerful and general-purpose devices than the DSP units in FPGAs). These works demonstrate the utility of program

[1]Diospyros uses symbolic evaluation, which is related to program synthesis, to lift imperative programs for digital signal processors into a high-level mathematical representation that can then be used with the technique of equality saturation [175] to generate optimized code for the target devices. This is also distinct from the program synthesis techniques referenced elsewhere in this dissertation.

synthesis for generating code that handles specific wrinkles in hardware designs, as does the use of program synthesis in Lakeroad to harness the programmability of FPGA DSPs. Note that other types of solvers (beyond the SMT solvers used in Lakeroad) can be used within compilers, e.g. using partitioned boolean quadratic problem (PBQP) solvers for instruction selection in LLVM [56]

Lakeroad is also related to past work in FPGA compilation and techmapping, much of which does not entreaty to support programmable DSPs with as much generality. ODIN [89] and ODIN-II [88] are used in *hard-block synthesis* for FPGAs, which is the task of mapping portions of hardware designs to specialized units (*hard blocks*) like multipliers. They operate purely over syntax (e.g., mapping * to a multiplier) and so are greatly limited in their ability to handle programmable DSPs. The ABC [28] logic synthesis tool is used to lower hardware designs into LUT and carry-chain configurations; it is related to Lakeroad in that it also uses constraint solvers to find configurations, though it is not general enough to handle a wide variety of programmable DSPs, unlike the program synthesis techniques used in Lakeroad. Note also that the use of configuration files in Lakeroad to abstract away details of the FPGA architecture was inspired by past work in FPGA compilation, including OpenFPGA [172] and the Verilog-to-Routing project (VTR) [153], both of which use abstract architecture descriptions to facilitate portability across designs, though these projects are limited in their support for DSPs. Library-Parameterized Models [2, 5] define generic interfaces for common primitives and are also similar to Lakeroad's primitive interfaces, though they are limited in their ability to represent configurable units like DSPs.

Virtual FPGA overlays [112, 27, 103] are another approach to improving the mapping of hardware designs to hardware. Overlays present a "virtual" FPGA architecture; each actual architecture must then define a mapping from virtual to actual primitives. This required translation is similar to Lakeroad's requirement on users to implement primitive interfaces in an architecture description, though it requires more user effort. The translation from virtual to actual architecture often comes with a steep resource and performance overhead.

# Part II Conclusion

Part II presented Lakeroad, a novel approach to FPGA technology mapping. Lakeroad utilizes both more adaptable **Algorithms**—sketch-guided program synthesis—and more explicit **Models**—vendor-supplied simulation models—to provide greater **Correctness**, completeness (i.e. **Optimizations**), and extensibility (i.e. reduced **Devtime**) over state-of-the-art tools. Because program synthesis tools can efficiently explore large search spaces, Lakeroad can find mappings of hardware designs to FPGA DSPs in more cases than state-of-the-art tools, often finding more efficient implementations in the process. With our techniques of semantics extraction from HDL and architecture-independent sketch templates, users must expend little manual effort to apply Lakeroad to a given FPGA architecture and extend it to handle further primitives. Moreover, our formalization of Lakeroad fosters greater confidence in its correctness. Lakeroad hence enables the extensible, efficient, and correct lowering of hardware designs to FPGAs, highlighting the effectiveness of program synthesis for FPGA technology mapping.

Lakeroad cleanly and completely realizes my thesis set out at the start of this dissertation: utilize explicit, vendor-supplied **Models**, apply state-of-the-art automated reasoning **Algorithms**, and you will produce a powerful compiler backend when measured along the axes of **Optimizations**, **Correctness**, and **Devtime**.

# Chapter 11

# Conclusion and Broader Thoughts

In this dissertation, I have presented an argument for a certain method of building compiler backends: namely, automatically generating them from explicit **Models** of hardware using automated reasoning **Algorithms**. I demonstrated how this method of backend generation leads to improved **Optimizations**, greater **Correctness**, and reduced **Devtime** in two case studies. In part I, I introduced Glenside, a language and tool enabling the use of equality saturation on machine learning workloads. When incorporated into the 3LA methodology, Glenside enabled more flexible mapping of machine learning workloads to accelerators, ultimately enabling easier developer testing of hardware designs. In part II, I introduced Lakeroad, which utilizes sketch-guided program synthesis and semantics extracted from vendor-supplied simulation models to generate more correct, more complete, and more extensible FPGA technology mappers. While the examples presented in this dissertation are specific—applying equality saturation to accelerator mapping, applying sketch-guided synthesis to technology mapping—the underlying recipe is portable. In the end, I hope it is this recipe you take away: to automatically generate better compiler backends, apply automated reasoning **Algorithms** to explicit, formal **Models** of the target hardware.

Though this dissertation may come to represent a substantial amount of my life's research output, I also consider it just a piece of a larger vision. To conclude this dissertation, I will attempt to capture the the higher level thoughts, ideas, and inspirations underlying the projects in my dissertation.

This dissertation is simply two useful, testable implementations (Glenside and Lakeroad) of a larger idea that's been bugging me since I was a Master's student at Penn State. At Penn State, I worked on the problem of computing with emerging devices—in our case, transistor which provided computing primitives other than just Boolean logic [146, 204, 205]. This is where the nagging feeling started: namely, the feeling that *the hardware tells us what it does.* Let me explain what I mean.

Models of hardware are ubiquitous. Any representation of hardware that isn't the literal silicon itself—from a low-level GDSII file, to a mid-level *Register Transfer Level (RTL)* representation, to a high-level HLS implementation or Python simulator—is a *model* of hardware. All hardware in use likely has at least one model behind it, if not more. Furthermore, models come in all shapes and sizes, and capture far more than just computational functionality, but also things like timing, area, and power. The SkyWater PDK (recently open sourced by Google) or the ASAP7 PDK from Arizona State University are great examples of open-source process development kits, rich with models (simulation, timing, power, and area) of real, fabricatable hardware platforms.

There is currently a directionality associated with hardware models. For example, the simulation models of an FPGA's primitives packaged inside hardware synthesis tools are meant to be used to simulate a design after it has been compiled from its higher-level specification. Alternatively, hardware designs intended for synthesis of an FPGA or ASIC are intended to be run through a synthesis tool and lowered to a netlist (to eventually become an ASIC or FPGA bitstream). In both cases, the hardware model is only *lowered,* such as when a synthesis-ready design is compiled, or *lowered to,* such as when a design is compiled to simulation models. It is not often the case that

these models are used in the opposite direction—i.e. *lifting* models to higher levels of abstraction, to generate higher-level outputs (like compiler backends).

It is surprising that models are only used in the lowering direction, as many of them are prime for lifting. Models written in Verilog, for example, are readily usable by automated reasoning tools like SMT solvers. For the most part, automated reasoning tools are only used to do post-lowering *verification*, but there is no reason that they can't be run "in the other direction"—in fact, this is exactly what Lakeroad does.

With all of that in mind, let's return to the idea that *the hardware tells us what it does.* Rephrasing this idea in the terms presented in the paragraphs above, we are overflowing with rich, varied hardware models that are prime for use with automated reasoning tools, ready to have interesting, higher-level semantics lifted from them. It's for these reasons that, for years, it's felt like hardware has been practically screaming at us, *directly* providing all the information we need to understand how to compile to it optimally. Yet we ignore it, choosing instead to use the models *indirectly.*

When I say that tools use models "indirectly", I'm referring to the current optimization loop standard in many optimization tasks. Consider hardware synthesis tools. During synthesis, the tool will make some guesses during compilation about what optimizations will generate optimal output (often informed, indirectly, by the underlying hardware models). The designer then simulates the results using the hardware models, to check whether the tool's guesses were correct. The odd part of this loop, to me, is the *indirect* use of the models to inform compiler construction, and the *direct* use of the models only after the fact, to check compilation results. Why not shorten this loop, and use the models *directly* during compiler construction? Hardware tells us what it does—so why not listen?

Lakeroad was the most concrete instantiation of this idea that I was able to achieve in the span of my PhD. Rather than encoding DSP mapping rules as patterns (informed, indirectly, from models and DSP documentation), Lakeroad determines how to map to primitives by ingesting

models directly, utilizing program synthesis to map to primitives based on the ingested models. However, Lakeroad only scratches the surface of the broader idea I'm discussing here. Lakeroad only maps based on functional behavior alone; it does not take advantage of, for example, the timing information also available in the models.

Furthermore, these ideas go far beyond digital hardware. I've been lucky enough to apply these ideas in non-boolean, analog computing [204, 205, 146] as well as DNA circuit design [191]. It isn't just digital hardware that is telling us what it does. Any computing substrate has primitives; most primitives have models. These models can be similarly used to generate compilers.

With that, I will conclude this dissertation. If nothing else, I hope you take with you the idea that *hardware is telling us what it does*—it's on us to listen!

# Bibliography


[1] A survey and critique of some models of code generation. 1977.

[2] Eia-is-103 : Library of parameterized modules (lpm).

[3] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI '16, pages 265–283, USA, 2016. USENIX Association.

[4] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

[5] Altera. Lpm quick reference guide.

[6] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney, and D. Sorensen. *LAPACK Users' Guide (Third Ed.)*. Society for Industrial and Applied Mathematics, USA, 1999.

[7] Luke Anderson, Andrew Adams, Karima Ma, Tzu-Mao Li, and Jonathan Ragan-Kelley. Learning to schedule halide pipelines for the gpu, 2020.

[8] Armaiti Ardeshiricham, Yoshiki Takashima, Sicun Gao, and Ryan Kastner. Verisketch: Synthesizing secure hardware designs with timing-sensitive information flow properties. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 1623–1638, New York, NY, USA, 2019. Association for Computing Machinery.

[9] Guido Arnout. SystemC standard. In *Proceedings of the Design Automation Conference*, pages 573–577, USA, 2000. IEEE.

[10] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[11] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998.

[12] Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, USA, 1998.

[13] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, Feb 2019.

[14] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. Creating an agile hardware design flow. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20, New York, NY, USA, 2020. IEEE Press.

[15] Sorav Bansal and Alex Aiken. Automatic generation of peephole superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 394–403, New York, NY, USA, 2006. Association for Computing Machinery.

[16] Sorav Bansal and Alex Aiken. Binary translation using peephole superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI '08, pages 177–192, USA, 2008. USENIX Association.

[17] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

[18] Andrew Becker, David Novo, and Paolo Ienne. Sketchilog: Sketching combinational circuits. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–4, 2014.

[19] Gilbert Louis Bernstein and Jonathan Ragan-Kelley. What are the semantics of hardware? In *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE)*, 2021.

[20] Gabriel Hjort Blindell. *Instruction Selection - Principles, Methods, and Applications*. Springer, Berlin, Heidelberg, 2016.

[21] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. *SIGPLAN Not.*, 52(6):467–481, jun 2017.

[22] Thomas Bourgeat, Clément Pit-Claudel, and Adam Chlipala. The essence of bluespec: a core language for rule-based hardware design. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 243–257, 2020.

[23] Martin Brain. Further steps down the wrong path: Improving the bit-blasting of multiplication. In *SMT*, pages 23–31, 2021.

[24] Florian Brandner. Automatic tool generation from structural processor descriptions. In *KPS'09: Proceedings of the 15th Biennial Workshop on Programmiersprachen und Grundlagen der Programmierung*, 2009.

[25] Florian Brandner, Dietmar Ebner, and Andreas Krall. Compiler generation from structural architecture descriptions. In *Proceedings of the 2007 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 13–22, 2007.

[26] Florian Brandner, Viktor Pavlu, and Andreas Krall. Automatic generation of compiler backends. *Software: Practice and Experience*, 43(2):207–240, 2013.

[27] Alexander Brant and Guy GF Lemieux. Zuma: An open fpga overlay architecture. In *2012 IEEE 20th international symposium on field-programmable custom computing machines*, pages 93–96. IEEE, 2012.

[28] Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pages 24–40. Springer, 2010.

[29] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization*, pages 300–313, 2018.

[30] Manuel M T Chakravarty, Gabriele Keller, Sean Lee, Trevor L. McDonell, and Vinod Grover. Accelerating Haskell array codes with multicore GPUs. In *DAMP '11: The 6th workshop on Declarative Aspects of Multicore Programming*. ACM, January 2011.

[31] Wei-Ting Jonas Chan, Andrew B. Kahng, Siddhartha Nath, and Ichiro Yamamoto. The ITRS MPU and SOC system drivers: Calibration and implications for design-based equivalent scaling in the roadmap. In *Proceedings of the 32nd IEEE International Conference on Computer Design*, ICCD '14, pages 153–160, New York, NY, USA, 2014. IEEE Computer Society.

[32] Benjamin Charlier, Jean Feydy, Joan Alexis Glaunès, François-David Collin, and Ghislain Durif. Kernel operations on the gpu, with autodiff, without memory overflows. *Journal of Machine Learning Research*, 22(74):1–6, 2021.

[33] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High Performance Convolutional Neural Networks for Document Processing. In Guy Lorette, editor, *Tenth International Workshop on Frontiers in Handwriting Recognition*, La Baule (France), October 2006. Université de Rennes 1, Suvisoft. http://www.suvisoft.com.

[34] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, 2015.

[35] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Meghan Cowan, Haichen Shen, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 579–594, USA, 2018. USENIX Association.

[36] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 3393–3404, USA, 2018. USENIX Association.

[37] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE J. Solid State Circuits*, 52(1):127–138, 2017.

[38] Zhi Chen and Cody Yu. How to bring your own codegen to tvm. `https://tvm.apache.org/2020/07/15/how-to-bring-your-own-codegen-to-tvm`, 2020.

[39] Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. Bring your own codegen to deep learning compiler, 2021.

[40] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, and Arvind. Kami: A platform for high-level parametric hardware specification and its modular verification. *Proc. ACM Program. Lang.*, 1(ICFP), August 2017.

[41] The cifar-10 dataset. `http://www.cs.toronto.edu/~kriz/cifar.html`, 2009. Accessed Nov. 15, 2021.

[42] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. Fpga hls today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(4):1–42, 2022.

[43] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for fpgas: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4):473–491, 2011.

[44] Charles Consel and Olivier Danvy. Tutorial notes on partial evaluation. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 493–501, 1993.

[45] CoreML: Integrate Machine Learning Models Into Your App. `https://developer.apple.com/documentation/coreml`, 2022.

[46] Ross Daly, Caleb Donovick, Jackson Melchert, Rajsekhar Setaluri, Nestan Tsiskaridze Bullock, Priyanka Raina, Clark Barrett, and Pat Hanrahan. Synthesizing instruction selection rewrite rules from rtl using smt. In *CONFERENCE ON FORMAL METHODS IN COMPUTER-AIDED DESIGN–FMCAD 2022*, page 139, 2022.

[47] Ross G. Daly, Caleb Donovick, Caleb Terrill, Jackson Melchert, Priyanka Raina, Clark Barrett, and Pat Hanrahan. Efficiently synthesizing lowest cost rewrite rules for instruction selection. *ArXiv*, abs/2405.06127, 2024.

[48] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[49] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. Imagenet: A large-scale hierarchical image database. In *Proceedings of the 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, CVPR '09, pages 248–255, New York, NY, USA, 2009. IEEE.

[50] Nachum Dershowitz. A taste of rewrite systems. In Peter E. Lauer, editor, *Functional Programming, Concurrency, Simulation and Automated Reasoning*, volume 693 of *Lecture Notes in Computer Science*, pages 199–228, Berlin, Heidelberg, 1993. Springer.

[51] Annie Despland, Monique Mazaud, and Raymond Rakotozafy. Pagode: A back end generator using attribute abstract syntaxes and term rewritings. In *International Conference on Compiler Construction*, 1990.

[52] Annie Despland, Monique Mazaud, and Raymond Rakotozafy. Using rewriting techniques to produce code generators and proving them correct. *Sci. Comput. Program.*, 15:15–54, 1990.

[53] João Dias and Norman Ramsey. Automatically generating instruction selectors using declarative machine descriptions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 403–416, New York, NY, USA, 2010. Association for Computing Machinery.

[54] Bruno Dutertre. Yices 2.2. In *International Conference on Computer Aided Verification*, pages 737–744. Springer, 2014.

[55] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[56] Dietmar Ebner, Florian Brandner, Bernhard Scholz, Andreas Krall, Peter Wiedermann, and Albrecht Kadlec. Generalized instruction selection using ssa-graphs. In *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 31–40, 2008.

[57] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzhenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. Mlir as hardware compiler infrastructure. In *Workshop on Open-Source EDA Technology (WOSET)*, 2021.

[58] Helmut Emmelmann. Code selection by regularly controlled term rewriting. In *Code Generation*, 1991.

[59] Zhenman Fang, Farnoosh Javadi, Jason Cong, and Glenn Reinman. Understanding performance gains of accelerator-rich architectures. In *Proceedings of the 30th IEEE International Conference on Application-specific Systems, Architectures and Processors*, ASAP '19, pages 239–246, New York, NY, USA, 2019. IEEE.

[60] Andreas Fauth, Johan Van Praet, and Markus Freericks. Describing instruction set processors using nml. In *Proceedings the European Design and Test Conference. ED&TC 1995*, pages 503–507. IEEE, 1995.

[61] J. Feldman and David Gries. Translator writing systems. *Communications of the ACM*, 11:77 – 113, 1968.

[62] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. A programmable programming language. *Communications of the ACM*, 61(3):62–71, March 2018.

[63] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Design Inc., 2010.

[64] Franz Franchetti, Tze Meng Low, Doru Thom Popovici, Richard M Veras, Daniele G Spampinato, Jeremy R Johnson, Markus Püschel, James C Hoe, and José MF Moura. Spiral: Extreme performance portability. *Proceedings of the IEEE*, 106(11):1935–1968, 2018.

[65] Christopher W Fraser and Alan L Wendt. Automatic generation of fast optimizing code generators. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 79–84, 1988.

[66] Yoshihiko Futamura. Partial evaluation of computation process–an approach to a compiler-compiler. *Higher-Order and Symbolic Computation*, 12:381–391, 1999.

[67] Xiangyu Gao, Taegyun Kim, Aatish Kishan Varma, Anirudh Sivaraman, and Srinivas Narayana. Autogenerating fast packet-processing code using program synthesis. In *Proceedings of the 18th ACM Workshop on Hot Topics in Networks*, HotNets '19, page 150–160, New York, NY, USA, 2019. Association for Computing Machinery.

[68] Hubert Garavel, Mohammad-Ali Tabikh, and Imad-Seddik Arrada. Benchmarking Implementations of Term Rewriting and Pattern Matching in Algebraic, Functional, and Object-Oriented Languages - The 4th Rewrite Engines Competition. In *Proceedings of the 12th International Workshop on Rewriting Logic and its Applications (WRLA'18)*, Thessaloniki, Greece, 2018.

[69] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*, pages 769–774. IEEE, 2021.

[70] Alex Graves and Navdeep Jaitly. Towards end-to-end speech recognition with recurrent neural networks. In Eric P. Xing and Tony Jebara, editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 1764–1772, Bejing, China, 22–24 Jun 2014. PMLR.

[71] Sumit Gulwani, Alex Polozov, and Rishabh Singh. *Program Synthesis*, volume 4. NOW, August 2017.

[72] Rajesh K. Gupta and Giovanni De Micheli. Hardware-software cosynthesis for digital systems. *IEEE Design & Test of Computers*, 10:29–41, 1993.

[73] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. Fireiron: A scheduling language for high-performance linear algebra on gpus, 2020.

[74] Bastian Hagedorn, Johannes Lenfers, Thomas Kundefinedhler, Xueying Qin, Sergei Gorlatch, and Michel Steuwer. Achieving high-performance the functional way: A functional pearl

on expressing high-performance optimizations as rewrite strategies. *Proc. ACM Program. Lang.*, 4(ICFP), August 2020.

[75] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator for graph analytics. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-49, New York, NY, USA, 2016. IEEE Press.

[76] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 37–47, New York, NY, USA, 2010. Association for Computing Machinery.

[77] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. Eie: Efficient inference engine on compressed deep neural network. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 243–254, New York, NY, USA, 2016. IEEE Press.

[78] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, pages 770–778, New York, NY, USA, 2016. IEEE Computer Society.

[79] Yann Herklotz and John Wickerson. Finding and understanding bugs in fpga synthesis tools. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 277–287, 2020.

[80] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[81] Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Gus Henry Smith, Thierry Tambe, Akash Gaonkar, Vishal Canumalla, Andrew Cheung, Gu-Yeon Wei, et al. Application-level validation of accelerator designs using a formal software/hardware interface. *ACM Transactions on Design Automation of Electronic Systems*, 29(2):1–25, 2024.

[82] Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Thierry Tambe, Gus Henry Smith, Akash Gaonkar, Vishal Canumalla, Gu-Yeon Wei, Aarti Gupta, et al. Specialized accelerators and compiler flows: Replacing accelerator apis with a formal software/hardware interface. *arXiv preprint arXiv:2203.00218*, 2022.

[83] Bo-Yuan Huang, Sayak Ray, Aarti Gupta, Jason M. Fung, and Sharad Malik. Formal security verification of concurrent firmware in socs using instruction-level abstraction for hardware.

In *Proceedings of the 55th Annual Design Automation Conference*, DAC '18, New York, NY, USA, 2018. Association for Computing Machinery.

[84] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.*, 24(1), December 2018.

[85] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI '22, pages 703–718, USA, 2022. ACM.

[86] Inferring SIMD accumulator with Xilinx DSP48e2. `https://old.reddit.com/r/FPGA/comments/tr9vzn/inferring_simd_accumulator_with_xilinx_dsp48e2/`. Accessed: 2023-12-07.

[87] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference, 2017.

[88] Peter Jamieson, Kenneth B Kent, Farnaz Gharibian, and Lesley Shannon. Odin ii-an open-source verilog hdl synthesis tool for cad research. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 149–156. IEEE, 2010.

[89] Peter Jamieson and Jonathan Rose. A verilog rtl synthesis tool for heterogeneous fpgas. In *International Conference on Field Programmable Logic and Applications, 2005.*, pages 305–310. IEEE, 2005.

[90] `https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification.html`. Accessed April 3rd, 2024.

[91] Yangqing Jia. *Learning Semantic Image Representations at a Large Scale*. PhD thesis, EECS Department, University of California, Berkeley, May 2014.

[92] Neil D. Jones and David A. Schmidt. Compiler generation from denotational semantics. In *Semantics-Directed Compiler Generation*, 1980.

[93] Rajeev Joshi, Greg Nelson, and Keith Randall. Denali: A goal-directed superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 304–314, New York, NY, USA, 2002. Association for Computing Machinery.

[94] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. A domain-specific supercomputer for training deep neural networks. *Commun. ACM*, 63(7):67–78, jun 2020.

[95] 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. *SIGARCH Comput. Archit. News*, 45(2):1–12, June 2017.

[96] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017.

[97] J Kokila, N Ramasubramanian, and S Indrajeet. A survey of hardware and software co-design issues for system on chip design. In *Advanced Computing and Communication Technologies: Proceedings of the 9th ICACCT, 2015*, pages 41–49. Springer, 2016.

[98] Smail Kourta, Adel Abderahmane Namani, Fatima Benbouzid-Si Tayeb, Kim Hazelwood, Chris Cummins, Hugh Leather, and Riyadh Baghdadi. Caviar: An e-graph based trs for automatic code optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, CC '22, pages 54–64, New York, NY, USA, 2022. Association for Computing Machinery.

[99] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, 2012.

[100] Ruihang Lai, Junru Shao, Siyuan Feng, Steven S. Lyubomirsky, Bohan Hou, Wuwei Lin, Zihao Ye, Hongyi Jin, Yuchen Jin, Jiawei Liu, Lesheng Jin, Yaxing Cai, Ziheng Jiang, Yong Wu, Sunghyun Park, Prakalp Srivastava, Jared G. Roesch, Todd C. Mowry, and Tianqi Chen. Relax: Composable abstractions for end-to-end dynamic machine learning, 2023.

[101] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '19, pages 242–251, New York, NY, USA, 2019. Association for Computing Machinery.

[102] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. Programming and synthesis for software-defined FPGA acceleration: Status and future prospects. *ACM Trans. Reconfigurable Technol. Syst.*, 14(4), September 2021.

[103] Joshua Landgraf, Tiffany Yang, Will Lin, Christopher J Rossbach, and Eric Schkufza. Compiler-driven fpga virtualization with synergy. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 818–831, 2021.

[104] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International symposium on code generation and optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

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

[106] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In Jae W. Lee, Mary Lou Soffa, and Ayal Zaks, editors, *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '21, pages 2–14, New York, NY, USA, 2021. IEEE.

[107] Rainer Leupers and Peter Marwedel. Retargetable generation of code selectors from hdl processor models. In *Proceedings European Design and Test Conference. ED & TC 97*, pages 140–144. IEEE, 1997.

[108] Onnx: Open neural network exchange. `https://onnx.ai/`, 2019.

[109] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. Verified tensor-program optimization via high-level scheduling rewrites. *Proc. ACM Program. Lang.*, 6(POPL), jan 2022.

[110] `https://llvm.org/docs/CodeGenerator.html#instruction-selection-section`. Accessed April 18th, 2024.

[111] `https://github.com/llvm/llvm-project/blob/0cee89431d77d0bb0809fd9b2c9d21da2a2783aa/llvm/lib/Target/X86/X86InstrArithmetic.td#L130-L131`. Accessed April 17th, 2024.

[112] Roman L Lysecky, Kris Miller, Frank Vahid, and Kees A Vissers. Firm-core virtual fpga for just-in-time fpga compilation. In *FPGA*, page 271, 2005.

[113] Steven Solomon Lyubomirsky. *Compiler and Runtime Techniques for Optimizing Deep Learning Applications*. University of Washington, 2022.

[114] Donald MacKenzie. *Mechanizing Proof: Computing, Risk, and Trust*. The MIT Press, 09 2001.

[115] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S. Vetter. Nvidia tensor core programmability, performance & precision. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2018.

[116] Jedidiah McClurg, Miles Claver, Jackson Garner, Jake Vossen, Jordan Schmerge, and Mehmet E. Belviranli. Optimizing regular expressions via rewrite-guided synthesis. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques*, PACT '22, pages 426–438, New York, NY, USA, 2023. Association for Computing Machinery.

[117] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[118] Perry L Miller. Automatic creation of a code generator from a machine description. 1971.

[119] Alan Mishchenko, Satrajit Chatterjee, and Robert K. Brayton. Improvements to technology mapping for lut-based fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):240–253, 2007.

[120] Mlperf benchmarks. `https://mlcommons.org`, n. d. Accessed Oct. 21, 2021.

[121] T. Moreau, T. Chen, L. Vega, J. Roesch, L. Zheng, E. Yan, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy. A hardware-software blueprint for flexible deep learning specialization. *IEEE Micro*, pages 1–1, 2019.

[122] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Q. Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A hardware-software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019.

[123] Peter Mosses. *Mathematical semantics and compiler generation*. PhD thesis, University of Oxford, 1975.

[124] Chandrakana Nandi, Max Willsey, Adam Anderson, James R. Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 31–44, New York, NY, USA, 2020. Association for Computing Machinery.

[125] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, April 1980.

[126] Julie L. Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. Verifying and improving halide's term rewriting system with program synthesis. *Proc. ACM Program. Lang.*, 4(OOPSLA), November 2020.

[127] Joseph M. Newcomer. Machine-independent generation of optimal local code. 1975.

[128] Aina Niemetz and Mathias Preiner. Bitwuzla at the smt-comp 2020. *arXiv preprint arXiv:2006.01621*, 2020.

[129] Aina Niemetz and Mathias Preiner. Bitwuzla. In Constantin Enea and Akash Lal, editors, *Computer Aided Verification - 35th International Conference, CAV 2023, Paris, France, July 17-22, 2023, Proceedings, Part II*, volume 13965 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2023.

[130] Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2 , btormc and boolector 3.0. In *CAV (1)*, volume 10981 of *Lecture Notes in Computer Science*, pages 587–595. Springer, 2018.

[131] Robert Nieuwenhuis and Albert Oliveras. Proof-producing congruence closure. In Jürgen Giesl, editor, *Proceedings of the 16th International Conference on Term Rewriting and Applications (RTA 2005)*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468, Berlin, Heidelberg, 2005. Springer.

[132] Rishiyur Nikhil. Bluespec system verilog: Efficient, correct rtl from high level specifications. In *Proceedings of the Second ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, MEMOCODE '04, pages 69–70, USA, 2004. IEEE Computer Society.

[133] The nvidia deep learning accelerator (nvdla). `http://nvdla.org/`, 2018. Accessed Apr. 23, 2021.

[134] NVIDIA. Convolutional layers user guide. `https://docs.nvidia.com/deeplearning/performance/dl-performance-convolutional/index.html`, 2020.

[135] Pavel Panchekha, Alex Sanchez-Stern, James R. Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *SIGPLAN Not.*, 50(6):1–11, June 2015.

[136] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché Buc, Emily B. Fox, and Roman Garnett, editors, *Proceedings of the Annual Conference on Advances in Neural Information Processing Systems*, NeurIPS '19, pages 8024–8035, USA, 2019. Curran Associates, Inc.

[137] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy,

Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[138] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. *ACM SIGPLAN Notices*, 49(6):396–407, 2014.

[139] Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Arvind, and Adam Chlipala. Effective simulation and debugging for a high-level hardware language using software compilers. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 789–803, New York, NY, USA, 2021. Association for Computing Machinery.

[140] Sorawee Porncharoenwase, Luke Nelson, Xi Wang, and Emina Torlak. A formal foundation for symbolic evaluation with merging. *Proc. ACM Program. Lang.*, 6(POPL), January 2022.

[141] 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.

[142] 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. Association for Computing Machinery.

[143] Norman Ramsey. Pragmatic aspects of reusable program generators. *Journal of Functional Programming*, 13(3):601–646, 2003.

[144] Norman Ramsey and João Dias. Resourceable, retargetable, modular instruction selection using a machine-independent, type-based tiling of low-level intermediate code. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 575–586, New York, NY, USA, 2011. Association for Computing Machinery.

[145] Jakob Rath, Clemens Eisenhofer, Daniela Kaufmann, Nikolaj Bjørner, and Laura Kovács. Polysat: Word-level bit-vector reasoning in z3. *arXiv preprint arXiv:2406.04696*, 2024.

[146] Arijit Raychowdhury, Abhinav Parihar, Gus Henry Smith, Vijaykrishnan Narayanan, György Csaba, Matthew Jerry, Wolfgang Porod, and Suman Datta. Computing with networks of oscillatory dynamical systems. *Proceedings of the IEEE*, 107(1):73–89, 2018.

[147] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, ISCA '16, pages 267–278, New York, NY, USA, 2016. IEEE Press.

[148] Vijay Janapa Reddi, Christine Cheng, David Kanter, Peter Mattson, Guenther Schmuelling, Carole-Jean Wu, Brian Anderson, Maximilien Breughe, Mark Charlebois, William Chou, Ramesh Chukka, Cody Coleman, Sam Davis, Pan Deng, Greg Diamos, Jared Duke, Dave Fick, J. Scott Gardner, Itay Hubara, Sachin Idgunji, Thomas B. Jablin, Jeff Jiao, Tom St. John, Pankaj Kanwar, David Lee, Jeffery Liao, Anton Lokhmotov, Francisco Massa, Peng Meng, Paulius Micikevicius, Colin Osborne, Gennady Pekhimenko, Arun Tejusve Raghunath Rajan, Dilip Sequeira, Ashish Sirasao, Fei Sun, Hanlin Tang, Michael Thomson, Frank Wei, Ephrem Wu, Lingjie Xu, Koichi Yamada, Bing Yu, George Yuan, Aaron Zhong, Peizhao Zhang, and Yuchen Zhou. Mlperf inference benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture*, ISCA '20, page 446–459, New York, NY, USA, 2020. IEEE Press.

[149] Berkeley Logic Synthesis and Verification Group. ABC: A system for sequential synthesis and verification, 2005. `http://www.eecs.berkeley.edu/~alanmi/abc`.

[150] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey and benchmarking of machine learning accelerators. *2019 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep 2019.

[151] Timothy Richards. Verification of code generators using term rewriting systems. 2006.

[152] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Josh Pollock, Logan Weber, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. Relay: A high-level IR for deep learning. *CoRR*, abs/1904.08368, 2019.

[153] Jonathan Rose, Jason Luu, Chi Wai Yu, Opal Densmore, Jeffrey Goeders, Andrew Somerville, Kenneth B Kent, Peter Jamieson, and Jason Anderson. The vtr project: architecture and cad for fpgas from verilog to routing. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, pages 77–86, 2012.

[154] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks, 2019.

[155] Patrick R Schaumont. *A practical introduction to hardware/software codesign*. Springer Science & Business Media, 2012.

[156] Ravi Sethi. Circular expressions: Elimination of static environments. In *Science of Computer Programming*, 1981.

[157] A. Simbürger, S. Apel, A. Größlinger, and C. Lengauer. The potential of polyhedral optimization: An empirical study. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 508–518, 2013.

[158] Zachary D Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. A position on program synthesis for processor development. In *Workshop on Languages, Tools, and Techniques for Accelerator Design–LATTE*, volume 2022, 2022.

[159] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. A position on program synthesis for processor development. In *Workshop on Languages, Tools, and Techniques for Accelerator Design–LATTE 2022*, 2022.

[160] Zachary D. Sisco, Jonathan Balkind, Timothy Sherwood, and Ben Hardekopf. Loop rerolling for hardware decompilation. *Proc. ACM Program. Lang.*, 7(PLDI), jun 2023.

[161] Gus Henry Smith, Benjamin Kushigian, Vishal Canumalla, Andrew Cheung, Steven Lyubomirsky, Sorawee Porncharoenwase, René Just, Gilbert Louis Bernstein, and Zachary Tatlock. Fpga technology mapping using sketch-guided program synthesis. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 416–432, 2024.

[162] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. Pure tensor program rewriting via access patterns (representation pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, MAPS '21, pages 21–31, New York, NY, USA, 2021. Association for Computing Machinery.

[163] Gus Henry Smith, Zachary D Sisco, Thanawat Techaumnuaiwit, Jingtao Xia, Vishal Canumalla, Andrew Cheung, Zachary Tatlock, Chandrakana Nandi, and Jonathan Balkind. There and back again: A netlist's tale with much egraphin'. *arXiv preprint arXiv:2404.00786*, 2024.

[164] Michael J. A. Smith. Semantics-directed compiler generation. 2005.

[165] Alan Snyder. A portable compiler for the language c. 1975.

[166] Armando Solar-Lezama. *Program synthesis by sketching*. University of California, Berkeley, 2008.

[167] Michel Steuwer, Toomas Remmelg, and Christophe Dubach. Lift: A functional data-parallel ir for high-performance gpu code generation. In *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, CGO '17, page 74–85. IEEE Press, 2017.

[168] The simple theorem prover.

[169] Thierry Tambe, En-Yu Yang, Glenn G. Ko, Yuji Chai, Coleman Hooper, Marco Donato, Paul N. Whatmough, Alexander M. Rush, David Brooks, and Gu-Yeon Wei. 9.8 a 25mm$^2$ soc for iot devices with 18ms noise-robust speech-to-text latency via bayesian speech denoising and attention-based sequence-to-sequence dnn speech recognition in 16nm finfet. In *Proceedings of the IEEE International Solid-State Circuits Conference*, ISSCC '21, pages 158–160, New York, NY, USA, 2021. IEEE.

[170] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. Algorithm-hardware co-design of adaptive floating-point encodings for resilient deep learning inference. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference*, DAC '20, USA, 2020. IEEE Press.

[171] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning*, volume 97 of *ICML '19*, pages 6105–6114, Atlanta, Georgia, USA, 2019. PMLR.

[172] Xifan Tang, Edouard Giacomin, Aurélien Alacchi, Baudouin Chauviere, and Pierre-Emmanuel Gaillardon. Openfpga: An opensource framework enabling rapid prototyping of customizable fpgas. In *2019 29th International Conference on Field Programmable Logic and Applications (FPL)*, pages 367–374. IEEE, 2019.

[173] Xifan Tang, Ganesh Gore, Grant Brown, and Pierre-Emmanuel Gaillardon. Taping out an fpga in 24 hours with openfpga: The sofa project. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*, pages 400–400. IEEE, 2021.

[174] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM Symposium on Principles of Programming Languages*, POPL '09, page 264–276, 2009.

[175] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: A new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 264–276, New York, NY, USA, 2009. Association for Computing Machinery.

[176] Jürgen Teich. Hardware/software codesign: The past, the present, and predicting the future. *Proceedings of the IEEE*, 100(Special Centennial Issue):1411–1430, 2012.

[177] Ruiqin Tian, Luanzheng Guo, Jiajia Li, Bin Ren, and Gokcen Kestor. A high-performance sparse tensor algebra compiler in multi-level ir, 2021.

[178] Emina Torlak and Rastislav Bodik. Growing solver-aided languages with rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*, pages 135–152, 2013.

[179] Emina Torlak and Rastislav Bodik. A lightweight symbolic virtual machine for solver-aided host languages. *ACM SIGPLAN Notices*, 49(6):530–541, 2014.

[180] Hugo Touvron, Piotr Bojanowski, Mathilde Caron, Matthieu Cord, Alaaeldin El-Nouby, Edouard Grave, Armand Joulin, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jégou. Resmlp: Feedforward networks for image classification with data-efficient training, 2021.

[181] Lenny Truong, Steven Herbst, Rajsekhar Setaluri, Makai Mann, Ross G. Daly, Keyi Zhang, Caleb Donovick, Daniel Stanley, Mark Horowitz, Clark W. Barrett, and Pat Hanrahan. fault: A python embedded domain-specific language for metaprogramming portable hardware verification components. In Shuvendu K. Lahiri and Chao Wang, editors, *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV 2020)*, volume 12224 of *Lecture Notes in Computer Science*, pages 403–414, Berlin, Heidelberg, 2020. Springer.

[182] `https://tvm.apache.org/docs/how_to/work_with_schedules/tensorize.html`. Accessed May 4th, 2024.

[183] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. Synthesizing jit compilers for in-kernel dsls. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*, page 564–586, Berlin, Heidelberg, 2020. Springer-Verlag.

[184] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 874–886, 2021.

[185] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, pages 874–886, New York, NY, USA, 2021. Association for Computing Machinery.

[186] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730*, 2018.

[187] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[188] Luis Vega, Joseph McMahan, Adrian Sampson, Dan Grossman, and Luis Ceze. Reticle: a virtual machine for programming modern fpgas. In *Proceedings of the 42nd ACM SIGPLAN*

*International Conference on Programming Language Design and Implementation*, pages 756–771, 2021.

[189] Verilator. `https://www.veripool.org/verilator/`, n. d.

[190] Yisu Remy Wang, Shana Hutchison, Dan Suciu, Bill Howe, and Jonathan Leang. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.*, 13(11):1919–1932, 2020.

[191] Lancelot Wathieu, Gus Smith, Luis Ceze, and Chris Thachuk. Fridge compiler: Optimal circuits from molecular inventories. In *International Conference on Computational Methods in Systems Biology*, pages 236–252. Springer, 2023.

[192] Paul N. Whatmough, Sae Kyu Lee, Marco Donato, Hsea-Ching Hsueh, Sam Likun Xi, Udit Gupta, Lillian Pentecost, Glenn G. Ko, David M. Brooks, and Gu-Yeon Wei. A 16nm 25mm$^2$ soc with a 54.5x flexibility-efficiency range from dual-core arm cortex-a53 to efpga and cache-coherent accelerators. In *Proceedings of the 2019 Symposium on VLSI Circuits*, page 34, New York, NY, USA, 2019. IEEE.

[193] 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.

[194] 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.

[195] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.*, 5(POPL), January 2021.

[196] Word-level language modeling rnn. `https://github.com/pytorch/examples/tree/master/word_language_model`, 2020. Accessed Nov. 18, 2021.

[197] Clifford Wolf, Johann Glaser, and Johannes Kepler. Yosys-a free verilog synthesis suite. In *Proceedings of the 21st Austrian Workshop on Microelectronics (Austrochip)*, 2013.

[198] Wayne Wolf. A decade of hardware/software codesign. *Computer*, 36(4):38–43, 2003.

[199] Xilinx. Ultrascale architecture DSP slice user guide, 2021. `https://docs.xilinx.com/v/u/en-US/ug579-ultrascale-dsp`.

[200] DSP48E2 inference for convolution/multiplication of 8-bit operands. `https://support.xilinx.com/s/question/0D52E00006hpnGVSAY/dsp48e2-inference-for-convolutionmultiplication-of-8bit-operands?language=en_US`. Accessed: 2023-12-07.

[201] Can not correctly infer "A*B+C" to DSP48E2. https://support.xilinx.com/s/question/0D54U00006AqPXFSA3/can-not-correctly-infer-abc-to-dsp48e2?language=en_US. Accessed: 2023-12-07.

[202] The xilinx software development kit (xsdk). https://www.xilinx.com/products/design-tools/embedded-software/sdk.html, n. d. Accessed Apr. 24, 2021.

[203] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. In *Proceedings of Machine Learning and Systems*, volume 3, pages 255–268, Virtual, 2021. MLSys.org.

[204] Insik Yoon, Muya Chang, Kai Ni, Matthew Jerry, Samantak Gangopadhyay, Gus Smith, Tomer Hamam, Vijayakrishan Narayanan, Justin Romberg, Shih-Lien Lu, et al. A fefet based processing-in-memory architecture for solving distributed least-square optimizations. In *2018 76th Device Research Conference (DRC)*, pages 1–2. IEEE, 2018.

[205] Insik Yoon, Muya Chang, Kai Ni, Matthew Jerry, Samantak Gangopadhyay, Gus Henry Smith, Tomer Hamam, Justin Romberg, Vijaykrishnan Narayanan, Asif Khan, et al. A ferrofet-based in-memory processor for solving distributed and iterative optimizations via least-squares method. *IEEE Journal on Exploratory Solid-State Computational Devices and Circuits*, 5(2):132–141, 2019.

[206] yosys/passes/pmgen/xilinx_dsp.pmg. https://github.com/YosysHQ/yosys/blob/91fbd58980e87ad5dc0a5d37c049ffaf5ab243dd/passes/pmgen/xilinx_dsp.pmg. Accessed 2024-02-28.

[207] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. Cambricon-x: An accelerator for sparse neural networks. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '49, New York, NY, USA, 2016. IEEE Press.

[208] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, Joseph E. Gonzalez, and Ion Stoica. Ansor: Generating high-performance tensor programs for deep learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 863–879, Banff, Canada, November 2020. USENIX Association.

[209] Bill Zorn. *Rounding*. PhD thesis, University of Washington, 2021.