Application-Level Validation of Accelerator Designs Using a Formal Software/Hardware Interface

BO-YUAN HUANG*, Intel Corporation, USA
STEVEN LYUBOMIRSKY*, OctoML, USA
YI LI and MIKE HE, Princeton University, USA
GUS HENRY SMITH, University of Washington, USA
THIERRY TAMBE, Harvard University, USA
AKASH GAONKAR, Princeton University, USA
VISHAL CANUMALLA and ANDREW CHEUNG, University of Washington, USA
GU-YEON WEI, Harvard University, USA
AARTI GUPTA, Princeton University, USA
ZACHARY TATLOCK, University of Washington, USA
SHARAD MALIK, Princeton University, USA

Ideally, accelerator development should be as easy as software development. Several recent design languages/-tools are working toward this goal, but actually testing early designs on real applications end-to-end remains prohibitively difficult due to the costs of building specialized compiler and simulator support. We propose a new first-in-class, mostly automated methodology termed "3LA" to enable end-to-end testing of prototype accelerator designs on unmodified source applications. A key contribution of 3LA is the use of a formal software/hardware interface that specifies an accelerator's operations and their semantics. Specifically, we leverage the Instruction-Level Abstraction (ILA) formal specification for accelerators that has been successfully used thus far for accelerator implementation verification. We show how the ILA for accelerators serves as a software/hardware interface, similar to the Instruction Set Architecture (ISA) for processors, that can be used for automated development of compilers and instruction-level simulators. Another key contribution of this work is to show how ILA-based accelerator semantics enables extending recent work on equality saturation to auto-generate basic compiler support for prototype accelerators in a technique we term "flexible matching." By combining flexible matching with simulators auto-generated from ILA specifications, our approach enables end-to-end evaluation with modest engineering effort. We detail several case studies of 3LA, which uncovered an unknown flaw in a recently published accelerator and facilitated its fix.

CCS Concepts: • **Hardware → Application-specific VLSI designs**; *Functional verification*; • **Software and its engineering → Compilers**; • **Computer systems organization** → *Heterogeneous (hybrid) systems.*

*Both authors contributed equally to this research; their work was carried out whilst at Princeton University and University of Washington, respectively.

Authors' addresses: Bo-Yuan Huang, Intel Corporation, Santa Clara, CA, USA; Steven Lyubomirsky, OctoML, Seattle, WA, USA; Yi Li; Mike He, Princeton University, Princeton, NJ, USA; Gus Henry Smith, University of Washington, Seattle, WA, USA; Thierry Tambe, Harvard University, Cambridge, MA, USA; Akash Gaonkar, Princeton University, Princeton, NJ, USA; Vishal Canumalla; Andrew Cheung, University of Washington, Seattle, WA, USA; Gu-Yeon Wei, Harvard University, Cambridge, MA, USA; Aarti Gupta, Princeton University, Princeton, NJ, USA; Zachary Tatlock, University of Washington, Seattle, WA, USA; Sharad Malik, Princeton University, Princeton, NJ, USA.

XXXX-XXXX/2023/12-ART $15.00
https://doi.org/XXXXXXX.XXXXXXX

Additional Key Words and Phrases: Accelerator, domain-specific language, compilation, validation, software/hardware interface

ACM Reference Format:
Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Gus Henry Smith, Thierry Tambe, Akash Gaonkar, Vishal Canumalla, Andrew Cheung, Gu-Yeon Wei, Aarti Gupta, Zachary Tatlock, and Sharad Malik. 2023. Application-Level Validation of Accelerator Designs Using a Formal Software/Hardware Interface. 1, 1 (December 2023), 25 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Hardware specialization is the main technique for improving power-performance efficiency in emerging compute platforms. By customizing compute engines, memory hierarchies, and data representations [11, 22, 41], hardware accelerators provide efficient computation in various application domains like artificial intelligence, image processing, and graph analysis [15, 25–27, 63, 89]. However, despite significant recent progress in design languages and tools for custom accelerators [40, 57], many difficulties remain in developing domain-specific accelerators.

A particularly challenging aspect of accelerator development is validating early design prototypes on real applications. Such validation is critical, as errors can arise from some of the techniques used to achieve maximum power-performance efficiency in accelerators, such as the use of custom numeric representations or reformulated operators. In domains like deep learning (DL), signal processing or graphics, an application-level result (like a DL-based classification) can remain within acceptable range even if the numerical results of individual operations change slightly, presenting an opportunity to trade numerical accuracy for efficiency. However, these changes need to be carefully validated at the application level—even small changes in numerical accuracy of individual operators have the potential to cascade throughout an application, making the application-level results unacceptable [90]. Early end-to-end application level validation is essential for avoiding expensive and complex late stage hardware design changes.

1.1 Challenges and Goals for Application-level Validation

Testing accelerators under development on complete applications requires two critical components: compiler support and application-level testing support.

- **Custom compiler support:** An application (likely written in a domain-specific language, or DSL) must be adapted to offload computations to an accelerator, which entails writing DSL compiler passes or manual modification of the source program. In common practice, invoking accelerator operations from software requires engineering effort, such as developing custom drivers to invoke accelerators via memory-mapped I/O (MMIO) interfaces. Such drivers are opaque to the compiler, difficult to debug, and often rely on low-level architectural details. The compilation tasks would be simplified through *greater automation* in: (1) identifying acceleration offload opportunities in the application, and (2) generating the low-level code that invokes the requisite accelerator operations.
- **Application-level testing support:** This goal poses several difficulties with existing techniques. Register-transfer level (RTL) designs (and thus RTL simulation) are not available in the early stages when the proposed end-to-end-testing is most useful. Even when prototype RTL designs are available, RTL simulation is only practically feasible for individual operations, being too slow for full applications. FPGA-based emulation requires significant engineering effort and is typically not done until late design stages. Faster high-level simulation (e.g., using SystemC) is feasible, but requires manually writing detailed simulation models and verifying later that they are sound with respect to the RTL implementation. The ideal for

application-level testing is to *automatically generate a sound high-level simulation model* for the accelerator that can be co-simulated with an application.

Note that the support components outlined above are specialized to a particular accelerator, and need to be updated every time an accelerator design is modified. In current practice, large industrial teams invest substantial resources to develop bespoke infrastructure [36, 37], while smaller teams often do not pursue end-to-end evaluation, as illustrated by our literature survey in Fig. 1.

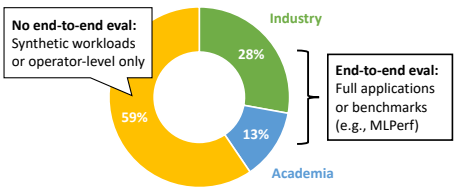

Fig. 1. **Gap in end-to-end evaluation of accelerators for neural network applications:** Our survey of 79 papers in recent conferences (ISCA, MICRO, VLSI, and ISSCC in 2021 and ICCAD, DAC in 2020) that introduced new DL accelerator designs/methodologies, comparing how the accelerators were evaluated. 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.

1.2 Novel Contributions of our 3LA Approach

We present a *first-in-class* methodology that supports end-to-end evaluation of accelerators *on unmodified full applications*, which includes the ability to compile to and run simulations of accelerator designs still in flux. As a practical capability, *this provides hardware designers with a feedback loop similar to that of software debugging and testing.*

Our methodology, termed "3LA," aims to reduce the manual engineering required for this feedback loop by effectively treating accelerator operations as extensions of processor instructions. A novel contribution of 3LA is the use of a *formal software/hardware (SW/HW) interface* that specifies an accelerator's operations and their semantics. Specifically, we leverage the Instruction-Level Abstraction (ILA), a formal specification for accelerators, that has been successfully used thus far for accelerator implementation verification [32] but not for compilation. In this work, we show how the ILA for accelerators serves as a SW/HW interface, similar to the Instruction Set Architecture (ISA) for processors, effectively serving as a "single source of truth" to drive various tasks required for compilation and end-to-end application testing. (The same high-level ILA model can then be used in a late-stage ILA/RTL validation step using existing techniques.) While the ISA has wide applications in computer architecture/compilers, there is no existing framework that uses an ISA-like formal SW/HW interface for accelerators: 3LA provides *an existence proof* that this is feasible both conceptually and as a practical framework.

Our work makes the following novel contributions:

- Use of a formal SW/HW interface for the accelerator: We use the ILA accelerator specification to automate key tasks required for compilation and instruction-level simulation (§3.1). Thus far, the ILA had only been used for accelerator implementation and firmware verification.
- Design of "flexible matching" (§3.3): This new semantics-guided term rewriting technique specialized to accelerators adds custom rewrite rules for accelerators (§3.2), and uses them in combination with generic compiler intermediate representation (IR) rewrites. This allows identifying, for the first time, semantically equivalent accelerator operations *even without a direct syntactic match* and significantly automates sophisticated operation offloading to accelerators without manually rewriting applications.
- 3LA methodology and prototype: Combining the above techniques to achieve end-to-end mapping of unmodified applications to accelerators is another contribution. No existing tool

(e.g., MLIR/CIRCT, PyMTL; see §1.3) has attempted, much less achieved, the capabilities offered by the 3LA prototype (§4) at such a level of automation. Our evaluation (§5) demonstrates automatic identification of multiple acceleration opportunities in off-the-shelf DL models imported from publicly available implementations and benchmarks. We evaluated these models end-to-end in simulation for three different accelerators; this was the *first time* that full applications were evaluated for two of the accelerators, and the tests exposed a flaw in one design related to numerical representations, which the developers were able to correct.

The 3LA methodology requires two main inputs from the user: (1) accelerator ILA models: a formal ILA specification for accelerator operations (which can be reused for separate RTL verification), and (2) IR-to-accelerator mapping rules: rewrite rules from the compiler IR to the accelerator operations ("mappings," for short). Note that both of these are *one-time efforts* per accelerator. Furthermore, this approach greatly lowers the effort after hardware design revisions, as it requires modifying only the accelerator operation specifications and rewrite rules, if necessary, and obviates the need for certain additional work, like updating the high-level simulators, which are generated automatically in 3LA.

1.3 Comparison with Existing Approaches and Tools

Although there have been many efforts in compiler flows to support accelerators [5, 14, 16, 33, 40, 42, 47, 53, 61, 77], none of them provides automated support for end-to-end testing of unmodified applications at the same level as 3LA. We start by providing a high-level comparison summarized in Table 1 and provide a detailed comparison with specific tools at the end.

1.3.1 Task-based Comparison. Existing approaches use different techniques for three critical tasks: accelerator operation selection, code generation, and software-hardware co-simulation. We compare them against 3LA for each task.

Task 1: Accelerator Operation Selection. A common practice is for the software developer to manually insert API calls for accelerator invocations, or to use syntactic patterns to identify possible matches (e.g., using BYOC [16]). Bespoke compilation efforts, possibly built on top of tools like BYOC and frameworks like MLIR [42] or Exo [33], make sophisticated compilation passes to identify operations for offloading, but require compiler expertise. In contrast, 3LA overcomes the limitations of purely syntactic matching to find *semantically-equivalent matches* in an automated way, without requiring the expertise and cost of bespoke compiler infrastructure.

Task 2: Code Generation. This task involves emitting the actual instructions, i.e., the MMIO loads/stores, from the application program to invoke accelerator operations. A common practice is to emit MMIO code in implementations of the API calls that invoke accelerator operations, often referred to as the "device driver" for the accelerator. However, these API calls are opaque, in that a compiler has no built-in knowledge of the semantics of the accelerator operations or the MMIO code. Alternatively, in bespoke compilation efforts, this knowledge is built into the compiler but requires significant expertise and does not use a formal hardware semantics. In contrast, in 3LA this code generation task is trivial following the operation selection step: each ILA instruction in the IR-to-accelerator mapping corresponds one-to-one with an MMIO instruction and is replaced accordingly. Moreover, the compiler has complete knowledge of their semantics via the formal SW/HW interface.

Task 3: Software-Hardware Co-Simulation. Co-simulation is needed to validate the results of the computation being done by the accelerator offloads (the hardware) and the host processor (the software) for application-level testing.

A common practice is to use frameworks like QEMU [9], which integrate RTL simulation calls (e.g., via Verilator [81]) with host processor execution. However, this is too slow for full-application testing. Higher-level system models in languages like SystemC [3] provide faster simulation but require significant effort for creating simulation models, and these models are difficult to validate against the RTL design. In contrast, the ILA model in 3LA supports *automatic generation of instruction-level simulation models* using the ILAng toolchain [31]. The ILA also allows separate verification against the RTL implementation, thereby ensuring soundness between the high-level simulation and the RTL implementation.

Table 1. **Comparison of the 3LA methodology against existing approaches for three critical tasks.** Approaches discussed: TVM with BYOC [16], Glenside [69], MLIR with various dialects [42], Halide with various extensions [24, 38, 44, 48, 60, 67, 70, 82], Exo [33], Verilator [81] with QEMU [9], SystemC [3] with QEMU, PyMTL [8], and Catapult HLS [68].

Approach	Pros and Cons	Related Works
Task 1: Accelerator operation selection		
Manual selection	Simple, but tedious and error-prone	common practice
Syntactic pattern matching	Simple, but may miss accelerator offloads	BYOC, MLIR
Custom flow	Flexible, but high effort to design (e.g., schedules of rewrites)	BYOC, MLIR, Halide, Exo
3LA: ILA and mapping rules	One-time effort (ILA + mapping rules) for Task 1-3	Glenside (rewrite rules)
Task 2: Code generation		
High-level API	No formal SW/HW interface, error-prone	common practice, BYOC
Bespoke codegen	No formal SW/HW interface, error-prone, high effort	BYOC, MLIR, Halide, Exo
3LA: auto-gen. MMIO code	Formal SW/HW interface, verifiable against RTL	
Task 3: Software-hardware co-simulation		
RTL simulation	Late-stage, very slow, operation-level only	Verilator with QEMU
High-level software model	Early-stage, end-to-end, not validated w.r.t. RTL	SystemC, PyMTL, Catapult
3LA: auto-gen. ILA simulator	Early-stage, end-to-end, validated w.r.t. RTL	

1.3.2 *Detailed comparison with closely related tools.*

We discuss details of some specific tools.

MLIR [42]. MLIR is a framework for building compiler IRs (as "dialects") in a structured, reusable manner. Some MLIR dialects address tasks related to hardware design; these include CIRCT, which supports high-level synthesis (HLS) and hardware simulation but *not compilation of applications to accelerators*. To the best of our knowledge, there is no SW/HW interface in MLIR that enables compiling applications to accelerators via CIRCT. Other dialects are intended to interface with specific accelerators (e.g., the TPU) and deep learning frameworks (e.g., ONNX). However, compilation using these dialects still entails mapping between IRs at different granularities and other challenges, which are addressed by 3LA.

HLS tools (e.g., Catapult [68]) and PyMTL [8]. These tools/frameworks allow for describing hardware designs with a high-level software-like interface, which is useful for designing accelerators and high-level hardware simulation. However, the hardware design specifications in these frameworks do not address accelerator operation selection and code generation during compilation—two of the three tasks in Table 1, which are significantly automated by 3LA.

Exo [33]. Exo also addresses the problem of compiling applications to accelerators. It uses a notion similar to high-level instructions for interfacing with accelerators, but, unlike 3LA, relies on *manually specified sequences of rewrites* and other bespoke compiler passes, to expose instances of those instructions and compile them to the devices using *exact syntactic matching*. 3LA introduces flexible matching to *automatically discover* accelerator offload opportunities in the applications, given a few rules relating the behavior of accelerator operations to the compiler IR (as in Exo). Additionally, Exo does not provide a *formal SW/HW interface* or any means to verify its accelerator instructions against the RTL implementation.

1.4 Paper organization

We first provide the background (§2) on ILA [32] and equality saturation [35, 75]. The techniques in 3LA are described in detail next (§3), followed by a description of the 3LA prototype[1] (§4). We present detailed evaluation using our prototype (§5), and end with a discussion on more broadly related work (§6) and conclusions (§7).

2 BACKGROUND

2.1 ILA Software/Hardware Interface Specification

The ILA is an ISA-like formal model for specifying the functional behavior of accelerators. It generalizes the ISA to accelerators, where each instruction of an accelerator ILA corresponds to a command at the accelerator interface, i.e., an MMIO load or store from a host processor. Like processor ISAs, the ILA captures a formal semantics of the accelerator behavior, by specifying how each instruction reads or updates software-visible (viz., architectural) state variables in the accelerator, while abstracting out implementation details.

Fig. 2 shows an example ILA specification for one of the instructions of FlexASR [72] (one of the three accelerators used in our evaluation studies). The ILA models are written in ILAng, a DSL embedded in C++. The figure caption points out the per-instruction modular specification, where each instruction is specified by defining its decode condition (i.e., when the instruction is triggered) and state update functions (i.e., how it updates the architectural state variables).

Thus far, the ILA has been used only for accelerator implementation verification and co-verification of firmware [30, 32]. In this work, we use ILA as a formal SW/HW interface that drives the key tasks in compilation and application-level testing for accelerators.

2.2 Term Rewriting with Equality Saturation

Term rewriting is a well-known technique for program transformations, with some compiler optimizations being implemented as term-rewriting systems [4, 10, 20, 50]. Given a set of syntactic rewrite rules ($\ell \rightarrow r$) that also preserve semantic equality, a term-rewriting system rewrites instances of pattern ℓ in the input program with semantically equivalent pattern r where applicable.

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 [84]. Equality saturation avoids phase-ordering issues by searching over many equivalent rewritings of the same program [35, 75]. 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 [54, 56]. 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.

In 3LA, we extend equality saturation to support accelerators. Specifically, we create custom rewrite rules for accelerators (§3.2), and specialize the cost function to maximize the number of accelerator offloads (§3.3) or consider cost of data movement (§3.4). Our prototype uses the `egg` library [85] for its efficient implementation of equality saturation.

3 3LA METHODOLOGY

We now describe technical details of the 3LA methodology, along with illustrative examples that demonstrate its various components through our prototype implemention for the deep learning domain. The important steps in the 3LA flow are shown in Fig. 3, and described in the related

[1] Our prototype, benchmarks, and evaluation infrastructure will be open-sourced under a permissive license.

Fig. 2. **ILA model (snippet) for FlexASR accelerator.** Lines 3-11 define the inputs and architectural state variables. Lines 14-22 show an example ILA instruction "pe_0_cfg_mngr." Its decode condition (lines 16-17) specifies that this instruction is triggered when there is a write command to the processing element's (PE) management configuration register. Its state update functions (lines 19-22) specify that this instruction stores arguments from the interface inputs into the corresponding configuration registers. The state update functions of instructions such as for linear layer (elided here) encode their operational semantics. This snippet highlights: (1) similarity of ILA with ISA, and (2) the formal semantics based on how each instruction reads/writes the architectural state variables.

```
1  auto m = ilang::Ila("flexasr-ila");
2  // declare inputs at the interface
3  auto wr = m.NewBvInput("top_if_wr", TOP_IF_WR_BITS);
4  auto rd = m.NewBvInput("top_if_rd", TOP_IF_RD_BITS);
5  auto addr = m.NewBvInput("top_addr_in", TOP_ADDR_IN_BITS);
6  auto data = m.NewBvInput("top_data_in", TOP_DATA_IN_BITS);
7  // declare architectural states
8  m.NewBvState("pe_0_is_valid", PE_VALID_BITS);
9  m.NewBvState("pe_0_is_bias", PE_IS_BIAS_BITS);
10 m.NewMemState("gb_large_buffer", TOP_ADDR_IN_BITS, TOP_DATA_IN_BITS);
11 // ... (some code)
12 // ILA instruction for configuring pe_cfg_mngr
13 auto instr = m.NewInstr("pe_0_cfg_mngr");
14 // define decode condition for this instruction
15 auto is_write = (wr == 1) & (rd == 0);
16 instr.SetDecode(is_write & (addr == PE_0_CFG_MNGR_ADDR));
17 // define state update functions for this instruction
18 auto is_valid = ilang::SelectBit(data, PE_IS_VALID_BIT_IDX);
19 instr.SetUpdate(m.state("pe_0_is_valid"), is_valid);
20 auto is_bias = ilang::SelectBit(data, PE_IS_BIAS_BIT_IDX);
21 instr.SetUpdate(m.state("pe_0_is_bias"), is_bias);
22 // ... (more code)
```

subsections. Our examples and prototype (§4) pertain to deep learning, but the 3LA techniques based on a formal SW/HW interface and flexible matching are general and can apply to other domains. For instance, besides tensor operations in deep learning, digital signal processing and cryptography are two domains that both ILA and equality saturation have been shown effective in modeling the accelerators [32, 71] and optimizing the workloads [43, 78], respectively. We chose deep learning since it has the highest availability of quality open-source accelerator designs and compiler frameworks.

We consider the following two examples: (1) an LSTM word language model (LSTM-WLM), a text generation application consisting of an LSTM recurrent neural network (RNN) followed by a linear layer [23, 86], and (2) ResNet-20, a widely used image classification model featuring 2D convolutions and residual connections [28]. Since the LSTM RNN comprises most of the computation in LSTM-WLM, it is desirable to accelerate this application using FlexASR [72], a natural language processing accelerator that includes support for both LSTM RNNs and linear layers in hardware. Thus, FlexASR can also be used to accelerate the linear layers in ResNet-20. In addition, ResNet-20 can also be accelerated with HLSCNN [83], an accelerator for 2D convolutions exclusively; hence, the two accelerators can be used in concert.

However, compiling DL applications like these from a high-level DSL (e.g., PyTorch for LSTM-WLM) to coarse-grained accelerators like FlexASR and HLSCNN poses several challenges highlighted below:

(1) **Specialized accelerator interfaces.** These accelerators, like many others, are invoked using MMIO instructions over AXI interfaces, to configure the accelerator's state and signal when to begin operations, thus requiring a thorough knowledge of both the accelerator architecture and functions of the MMIO instructions.

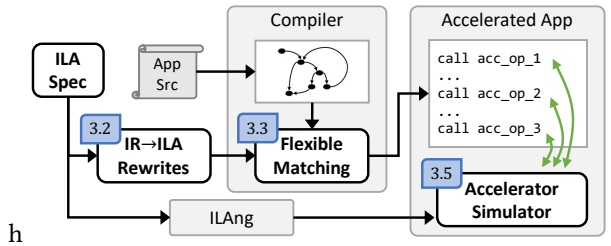

Fig. 3. **3LA methodology flow.** §3.2: the developer provides one *IR-to-accelerator mapping* for each accelerator operator, §3.3: a compiler extended with *flexible matching* automatically maps unmodified source application fragments to accelerator calls, §3.5: the ILAng platform generates fast simulators, enabling end-to-end co-simulation. These features enable a mostly automated workflow for end-to-end testing of prototype accelerator designs on unmodified applications.

(2) **Granularity mismatch.** The compiler must relate the accelerator operations' coarse-grained semantics (e.g., an LSTM RNN) with the possibly fine-grained corresponding representations in the compiler IR.
(3) **Numerical representations.** Accelerators often use specialized numerical representations for improved performance or reduced hardware costs. For example, FlexASR uses a custom format called AdaptivFloat [73] and HLSCNN uses a mixed 8/16-bit fixed-point representation. One must check that these data types do not cause inaccuracies at the full application level, particularly when values are cast between data types.

In the descriptions of various 3LA techniques in this section, we will emphasize how they address these challenges.

3.1 The ILA as a Formal SW/HW Interface in 3LA

The first step in the 3LA methodology is to develop the ILA formal models for accelerators (§2.1). We follow the techniques proposed in prior work [31, 32], where each instruction of an accelerator ILA corresponds to a command at the accelerator interface. Some instructions are simple instructions that configure the accelerator, while others may trigger complex operations, e.g., FlexASR's linear layer operator.

The ILA for accelerators serves as a formal software/hardware interface in 3LA, similar to the Instruction Set Architecture (ISA) for processors, and effectively drives the following tasks required for compilation and end-to-end application testing with accelerators.

- *Accelerator Operation Selection:* A formal instruction representation for accelerator operations enables adapting existing instruction selection techniques to identify acceleration opportunities and output the ILA instructions that correspond to functionally equivalent parts of the compiler IR representation.
- *Code Generation:* The ILA instructions correspond one-to-one with the MMIO commands that operate the accelerator. Thus, the selected ILA instructions can be directly lowered to MMIO commands for invoking accelerator operations from the application code running on a host processor.
- *SW/HW Co-Simulation*: The ILAng toolchain [31] can automatically generate a functional simulator given an ILA specification—this can be done in the early design stages, even without an RTL implementation.

Further, note that a revision to the accelerator design can easily be tested simply by making changes to the comparatively high-level ILA specification.

The ILA model size is about 10-20% of the size of the RTL implementation [32]. Although writing the ILA model can be a significant one-time effort, it carries benefits beyond the context of the 3LA methodology. The ILA specification has been used for checking the accelerator RTL implementation, both using formal verification [32] and simulation-based validation [31]. Thus, ILA-based accelerator simulations in 3LA can be made sound with respect to RTL.

3.2 Compiler IR-to-Accelerator Mapping

To support compilation to accelerators, we require some means of mapping from the compiler IR to the ILA instructions that specify the accelerator operations. This is accomplished by specifying an *IR-to-accelerator mapping rule* ("mapping," in short). In general, this is a many-to-many mapping, i.e., where a program fragment with many instructions in the compiler IR are rewritten to a program fragment with many ILA instructions on the accelerator side. This provides a general way to handle different granularities in compiler IR intrinsics (e.g., dot products and convolutions) and in accelerator operations (e.g., fine-grained operations in VTA [53] and coarse-grained operations in HLSCNN, FlexASR). Furthermore, the ILA instructions in the mapping provide a verifiable abstraction of the hardware accelerator operation in terms of updates to software-visible architectural state.

Writing the IR-to-accelerator mapping rule is a one-time effort per accelerator operation and is reusable across applications. Further, the mapping rule can be validated by comparing the results of the compiler IR fragment on the host device and the ILA-based simulation results.

Examples. In our prototype, we use the Relay IR [65] in the TVM DL compiler stack [14] as the compiler IR. TVM supports importing models from other DL frameworks by converting them into Relay, thus allowing our prototype to support these front-ends as well. Further, it enables leveraging the Bring Your Own Codegen (BYOC) [16] library for code generation (discussed later). For LSTM-WLM on FlexASR, we provide a mapping from an LSTM RNN (a large construct in Relay if "unrolled") to a short sequence of FlexASR ILA instructions, and another mapping for a linear layer, which we illustrate in Fig. 4. ResNet-20 also uses the same mapping for linear layer and a straightforward mapping from a single 2D convolution operator to a sequence of HLSCNN ILA instructions for performing a convolution.

3.3 Flexible Matching for Accelerator Operator Selection

Given compiler IR-to-accelerator mapping rules, we can identify all potential offloads to an accelerator by finding portions of an application that match the given compiler IR fragments, syntactically or semantically.

3.3.1 Difficulties due to syntactic matching. Searching the application for exact syntactic matches for the given compiler IR fragments (referred to as "exact matching") is simple to implement (e.g., this is done by the BYOC library in TVM [16]). However, 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. Developing a canonicalization for each given IR may be possible, but would require careful design per IR and further effort to prove that the program transformations preserve the canonicalization [55]. Application code can vary greatly in structure, particularly in the case of compiler IRs, which may be produced after several iterations of program transformations (as with TVM, its model importers may translate equivalent expressions from various frameworks into different Relay expressions).

Examples. In our LSTM-WLM, the compiler IR pattern for a linear layer is (as an S-expression [49]):

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

```
/*------------------------------------ (a) Compiler IR fragment ----------------------------*/
%1 = nn.dense(%data, %weight)
%2 = nn.bias_add(%1, %bias)

/*------------------------------------ (b) FlexASR ILA program fragment ---------------------------------*/
// configure accelerator states
FlexASR_ILA.pe_cfg_rnn_layer      %is_zero %is_cluster %is_bias %num_mngr %num_v_out
FlexASR_ILA.pe_cfg_mngr           %mngr_idx %is_zero %bias_w %bias_b %bias_i %num_v_in %base_w ...
FlexASR_ILA.pe_cfg_act_mngr       %is_zero %bias %num_insn %num_v_out %buf_base %out_base
FlexASR_ILA.pe_cfg_act_v          %is_zero %insn_0 %insn_1
FlexASR_ILA.gb_cfg_mmngr_gb_large %base_0 %num_v_0 %base_1 %num_v_1
FlexASR_ILA.gb_cfg_gb_control     %mode %is_rnn %mem_id_in %mem_id_out %num_v_in %num_v_out %num_ts
// trigger accelerator function
FlexASR_ILA.fn_start              %fn_id

/*------------------------------------ (c) FlexASR MMIO commands ----------------------------------------*/
// configure accelerator states
Write, Addr=0xA4400010, Data=0x0010101000001
Write, Addr=0xA4400020, Data=0x00000000100000000102020200
// ...
// trigger accelerator function
Write, Addr=0xA3000010, Data=0x1
```

Fig. 4. **An example IR-to-accelerator mapping for the FlexASR linear layer operation.** The compiler IR fragment (a) is mapped to a sequence of FlexASR ILA instructions (b) that configure the accelerator states and trigger the computation. The ILA instructions correspond one-to-one to the accelerator's MMIO commands (c). This example illustrates how the ILA instructions are used in mapping rules and code generation.

However, in ResNet-20, which was imported from MxNet, linear layers are equivalently expressed as:

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

when `%c` is a vector, for certain shapes `%s`. The former pattern would fail to match it, thus missing an opportunity to invoke FlexASR's linear layer operation.

3.3.2 Semantic matching via term rewriting. 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 aims to maximize the degree of automation by utilizing term-rewriting and equality saturation techniques to transform programs *to expose the most matching opportunities for accelerator operation selection.* We call this process "flexible matching", and describe how it is specialized for accelerators.

Flexible matching uses two kinds of rewrite rules:

- Compiler IR rewrite rules: These are general-purpose 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. Recall (§3.2) that these mappings are many-to-many, providing a general way to handle different granularities in compiler IR intrinsics and accelerator operations. When targeting new accelerators, accelerator designers are expected to provide these mappings. For our evaluation, we created these mappings for the operations supported by the accelerators.

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.

One benefit of separating the two kinds of rewrite rules in flexible matching is that this allows the compiler IR rewrites to use purely functional IRs, without requiring bespoke compilation steps for state/effect analysis during those rewrites, while stateful effects are limited to mappings, where they are formally specified by ILA instructions. Another benefit is that mappings for multiple accelerators can be *simultaneously* included, thereby searching over all opportunities to invoke all available accelerators in concert.

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.

Examples. In our prototype, we compile programs in Relay into another IR called Glenside [69], which uses the `egg` library [85] to implement equality saturation for tensor programs. Glenside provides a set of general-purpose rewrite rules for common deep learning operations, such as tensor shape transformations and algebraic manipulations of combinators. These rules allow the term-rewriting system to conclude that different variations of an expression (like the linear layer examples above) are, in fact, equivalent. In our examples, including those general-purpose rules (as compiler IR rewrites) exposed acceleration opportunities in both the LSTM-WLM and ResNet-20 programs by specifying *only a single* IR-to-accelerator mapping rule for each accelerator operator.

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)          (1)
(add (reshape (nn_dense %a %b) %s) %c)  ⟶  (reshape (bias_add (nn_dense %a %b) %c) %s)   (2)
                                    %x  ⟶  (reshape (flatten %x) (shape-of %x))          (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 [69]. Rule 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 2 shows that linear layer kernels can be expressed using different arrangement and combinations of operators, e.g., `bias_add` (broadcasting) or the elementwise `add`. Rule 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 1 and Rule 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, one for each target accelerator that we evaluated (§5).

```
(compute dot-product (cartesian-product ?x ?w))  ⟶  (vta-dense ?x ?w)                    (4)
                (conv2d ?input ?kernel ?group ...)  ⟶  (hlscnn-conv2d ?input ?kernel ?group)  (5)
                                {{LSTM Relay Pattern}}  ⟶  (flexasr-lstm ?input ?hidden_0 ...)  (6)
```

Rule 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 5 maps kernel-level computation of a 2D convolution to HLSCNN's `conv2d` operation—a common accelerator offloading for deep learning kernels. Rule 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 (§5) show that powerful 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 reasonable time.

3.4 3LA support for additional optimizations

The following 3LA design features provide support for additional optimizations: (1) IR-to-accelerator mapping: Recall that our mappings are polymorphic over tensor size, i.e., we parameterize these mappings with arguments such as sizes of the input/output data. On the accelerator side, the maximum sizes supported by a single accelerator offload are limited by the accelerator-controlled hardware resources (e.g., buffer sizes, memory layout, internal/external memory, etc.). (2) Flexible matching: Our technique provides optimization capabilities by including performance or other criteria in the cost function, which is optimized for instruction selection.

These features have facilitated the development of Shoehorn, a scheduling optimization framework that decomposes oversized DL layers to sequences of accelerator operations [45]. Based on the accelerator-controlled hardware resources and the layer dimensions, Shoehorn generates optimal schedules (loop tiling, loop ordering, memory partitioning across tensors, etc.) to minimize off-chip data movement. The details of this optimization framework are beyond the scope of this paper and we refer the reader to the Shoehorn paper for additional details [45].

In future work, we plan to integrate this scheduler with flexible matching, thereby enabling optimization of data movement or for considering possible tradeoffs with other costs. Another optimization opportunity we have identified is in removing redundant intermediate data transfers in back-to-back offloads to accelerator operations. This can be done in a pass after flexible matching and before code generation.

3.5 Co-Simulation for Application-Level Results

After the acceleration operation selection is done and specific portions of the application are marked as offloaded to an accelerator, we can co-simulate the results at the full-application level rather than for only individual operators. Namely, the portions of the applications that are not marked are directly executed on the host (generally a CPU) and the marked portions converted into their corresponding ILA instruction sequences are simulated via an ILAng-generated simulator. Note that the ILAng-generated simulators faithfully simulate the custom numerics, either using semantics formally modeled in the ILA specification or by accepting trusted software libraries that implement custom numerical data types.

Examples. In §5, we examine the application of the 3LA methodology on several DL applications. In the process, upon identifying a numerical accuracy issue with HLSCNN in ResNet-20 and MobileNet-V2, we rapidly explored revisions to the design by changing the ILA specification—a much simpler task than modifying the RTL.

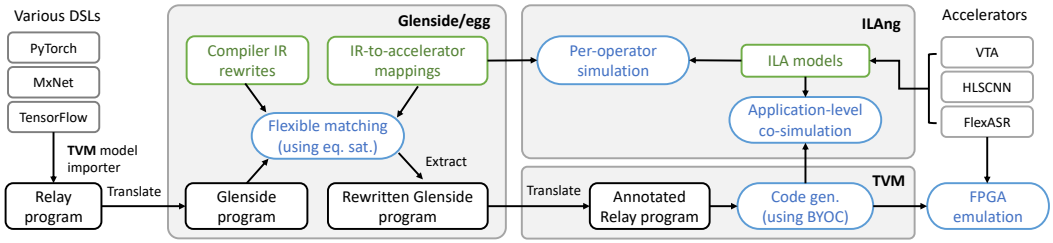

Fig. 5. **Prototype implementation of the 3LA flow.** The green boxes represent additional inputs needed for the 3LA flow. The blue boxes represent the mostly automated capabilities added by the flow. The ILA specification model can also be used to formally verify the accelerator RTL implementation (as demonstrated in prior work [32]). The compiler IR rewrite rules and IR-to-accelerator mapping rules can support other compiler optimization and verification tasks.

4 3LA PROTOTYPE IMPLEMENTATION

As a demonstration of the 3LA methodology, we have implemented an end-to-end compilation and simulation flow for DL applications by integrating with existing compiler frameworks and the ILAng platform [31], as shown in Fig. 5. Specifically, our prototype is integrated into the TVM DL compiler and uses Relay as the representation for DL applications [14, 65]. We convert Relay programs into Glenside, and then perform flexible matching via equality saturation on tensor programs using egg [69, 85]. Finally, we use ILAng for co-simulating the compiled applications.

DSL Front-End. TVM is a compiler framework for DL applications. We make use of TVM's model importer as the front-end for DSL programs. The importer takes programs written in common DL DSLs (e.g., ONNX [46], PyTorch [58], and TensorFlow [1]) and translates them into Relay.

Flexible Matching. As described earlier (§3.3), we implement flexible matching by translating input programs from Relay into Glenside. Given both compiler IR rewrites and IR-to-accelerator mappings via Glenside, egg explores the space of acceleration opportunities using equality saturation.

Code Generation. Once flexible matching completes, the extracted rewritten program is translated back to Relay where accelerator operations are specially annotated. In our prototype, we use TVM's BYOC library [16] to implement code generation (i.e., MMIO instructions and data movement code) for these accelerator operations.

5 EVALUATION

In this section, we evaluate our prototype for end-to-end testing with six applications and three accelerator designs. We focus especially on (1) automated identification of acceleration opportunities, and (2) application-level validation using automated co-simulation. Note that other tools provide very little automated support (if any) for these two capabilities, thus precluding any head-to-head comparisons. We also report on operator-level evaluation (accuracy and performance) and FPGA-based deployment.

5.1 Target Accelerators

We added support for three DL 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 [72]. It uses a custom numeric data type called *AdaptivFloat* for boosting the accuracy of quantized computations [73].
(2) **HLSCNN** is an accelerator optimized for 2D convolutions [83]. 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 [53]. 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. 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.

5.2 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 [74]. It has convolutions that are supported by VTA and HLSCNN.
(2) **LSTM-WLM** is a text generation application [86] implemented using an LSTM recurrent neural network architecture [23]. The LSTM layer in this model is supported by FlexASR.
(3) **MobileNet-V2** is a common CNN designed for mobile applications [29, 66]. 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 [76]. Its linear layers could be accelerated by VTA and FlexASR.
(5) **Transformer** is an NLP model comprised primarily of attention mechanisms [80]. We chose Transformer as a representative of recent popular NLP models.
(6) **ResNet** is a popular CNN designed for image classification [28]. Besides ResNet-20, which we use in most of the evaluation, in §5.3, we additionally compare various implementations of ResNet-50 from MLPerf [52] for its availability of different reference implementations.

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

5.3 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.

Table 2. **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 §5.5. 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

Table 2 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 (§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, certain Glenside rewrites [69] that implement the `im2col` optimization [12] rewrite 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 [64] in Table 2, right. Their Relay representations differed in subtle ways (such as in reshaping operators)[2] and are reflected in the difference in results of exact matching. Flexible matching found the same (increased) number of matches for each accelerator, regardless of its source DSL.

5.4 Per-Operator Evaluation

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 functional validation and performance evaluation at the operator level.

5.4.1 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 those of TVM's runtime on host. The accelerator ILA simulators precisely model the

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

Table 3. **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%

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 [2] 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 3 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 2) 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.

5.4.2 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. 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 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.

5.5 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,

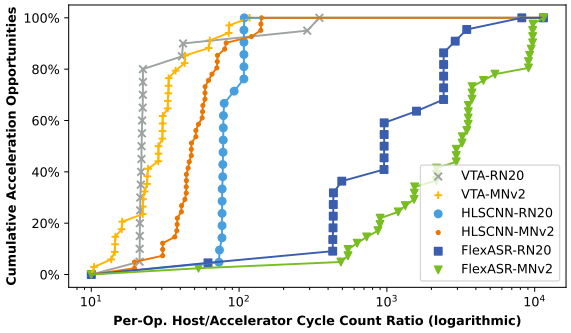

Fig. 6. **Cumulative distribution of per-operator performance gains of all identified acceleration opportunities** in ResNet-20 (RN20) and MobileNet-V2 (MNv2) on the three accelerators. Each point represents an operation offloaded from the host to the accelerator (as identified by flexible matching, Table 2). 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.

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.

We trained and validated the LSTM-WLM model using the WikiText-2 dataset [51]. The image classification models (MobileNet-V2 and ResNet-20) were trained and validated using the CIFAR-10 dataset [17]. We additionally trained and validated a MobileNet-V2 model optimized for ImageNet using the ImageNet dataset [19].

Table 4 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 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. 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 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

[3]We apply a form of uniform quantization [34], which involves scaling the results based on the floating point reference results.

Table 4. **Application-level co-simulation results.** We evaluated 100 WikiText-2 sentences (for LSTM-WLM), 2000 CIFAR-10 images (for MobileNet-V2 and ResNet-20), and 2000 ImageNet images (for MobileNet-V2) 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 numerics tuning are for the initial accelerator designs, modeled in ILA. The result with numerics 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 Numerics Tuning	Result with Numerics Tuning	Avg. Sim. Time†
LSTM-WLM	FlexASR	122.15	121.97	N/A	22.4s
ResNet-20 (CIFAR-10)	FlexASR	91.55%	91.50%	N/A	11.6s
	HLSCNN	91.55%	29.75%	92.10%	7min 3s
	FlexASR & HLSCNN	91.55%	29.15%	91.85%	7min 6s
MobileNet-V2 (CIFAR-10)	VTA	92.40%	89.40%	N/A	20min 15s
	FlexASR	92.40%	92.30%	N/A	18.1s
	HLSCNN	92.40%	10.35%	91.50%	20min 33s
	FlexASR & HLSCNN	92.40%	10.35%	91.20%	21min 01s
MobileNet-V2 (ImageNet)	VTA	72.30%	70.55%	N/A	15min 3s
	FlexASR	72.30%	72.10%	N/A	38.1s
	HLSCNN	72.30%	0.10%	68.85%	46min 34s
	FlexASR & HLSCNN	72.30%	0.10%	68.25%	48min 20s

∗ 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.

"Results with Numerics Tuning" in Table 4. This case study readily demonstrates how the 3LA methodology *facilitates debugging and improving accelerator designs with rapid turnaround.*

The overall results in Table 4 reaffirm the need for application-level validation, especially for accelerators utilizing custom numerics. Thanks to formal ILA models, 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.

5.6 System Deployment and FPGA Emulation

As an additional demonstration of 3LA, we explored its 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 [87]. 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 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.

[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.

6 RELATED WORK

Software/Hardware Co-Design. Recent work on accelerator generation and integration [5, 77] has explored adding support in the Halide [61] 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 [40] also provides a similar custom flow. By contrast, the 3LA methodology supports software/hardware co-design by mitigating impedance mismatches between the granularity of high-level DSLs and *near-arbitrary* accelerators; because of the flexibility of the ILA, the 3LA methodology is applicable to a broader class of compilers and accelerators.

Pattern Matching Accelerator Calls. The most closely related work to flexible matching is from (1) TVM BYOC [16], which only provides exact syntactic matching as discussed in §2, and (2) Glenside [69], which, prior to this work, had not been integrated into a compilation pipeline nor used to target custom accelerators. Past work has also explored rewrite-based techniques for automatically inferring instruction selection passes between ISAs [21, 62] and in the context of superoptimization [6, 7]. 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 [39, 79, 88]. There has also been significant work on ML and HPC compiler frameworks with varying degrees of support for targeting custom accelerators [14, 42, 47, 53, 61]. To the best of our knowledge, none of these frameworks provides support for testing prototype accelerators designs end-to-end on unmodified source applications.

Validating and Verifying Accelerator Calls. Tools like Verilator [81] and Cuttlesim [59] enable cycle-accurate RTL simulation, but are too slow to enable application-level co-simulation. Co-simulation using faster high-level SystemC [3] models partially address this gap; however, the SystemC models need to be independently written and, unlike ILA models, do not have a clear formal verification path to RTL. Further, general SystemC models do not target MMIO interfaces and may have arbitrary levels of detail. Other work has targeted formal verification of code generation for accelerators [33, 47], but does not have a path to RTL design verification that is possible with ILAs.

Support for Diverse DSLs. Our 3LA prototype supports importing from various DL frameworks (including MxNet [13], PyTorch [58], TensorFlow [1], ONNX [46], and CoreML [18]) via TVM's importers to the Relay IR. Similar to Relay, MLIR [42] now also supports importing models from various frameworks; the 3LA approach complements such flows as demonstrated in our prototype. In contrast, Exo [33] presently has no support for importing other representations.

7 CONCLUSIONS

In this work we address the key gaps hindering application-level evaluation of accelerator designs, especially during early design stages. We propose the 3LA methodology that contributes (1) the use of a formal software/hardware interface for specifying accelerator operations, which enables identifying acceleration opportunities and automatically generating correct high-level simulators, and (2) compiler rewrites and equality saturation for flexible matching, which facilitates automatically searching through a large space of equivalent programs to find acceleration opportunities. We provide a 3LA prototype implementation for DL applications using the TVM and ILAng frameworks and evaluate it through automated compilation of six applications on three different accelerator platforms.

REFERENCES

[1] 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. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI '16)*. USENIX Association, USA, 265–283.

[2] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and Danny Sorensen. 1999. *LAPACK Users' Guide (Third Ed.)*. Society for Industrial and Applied Mathematics, USA.

[3] Guido Arnout. 2000. SystemC standard. In *Proceedings of the Design Automation Conference*. IEEE, USA, 573–577. https://doi.org/10.1109/ASPDAC.2000.835166

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

[5] 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. 2020. Creating an Agile Hardware Design Flow. In *Proceedings of the 57th ACM/EDAC/IEEE Design Automation Conference* (Virtual Event, USA) *(DAC '20)*. IEEE Press, New York, NY, USA, Article 142, 6 pages. https://doi.org/10.1109/DAC18072.2020.9218553

[6] Sorav Bansal and Alex Aiken. 2006. Automatic Generation of Peephole Superoptimizers. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California, USA) *(ASPLOS XII)*. Association for Computing Machinery, New York, NY, USA, 394–403. https://doi.org/10.1145/1168857.1168906

[7] Sorav Bansal and Alex Aiken. 2008. Binary Translation Using Peephole Superoptimizers. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation* (San Diego, California) *(OSDI '08)*. USENIX Association, USA, 177–192.

[8] Shunning Jiang Christopher Torng Christopher Batten. 2018. An Open-Source Python-Based Hardware Generation, Simulation, and Verification Framework. In *Workshop on Open-Source EDA Technology (WOSET '18)*. N.A., Virtual, 1–5.

[9] Fabrice Bellard. 2005. QEMU, a Fast and Portable Dynamic Translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (Anaheim, CA) *(ATEC '05)*. USENIX Association, USA, 41.

[10] Gabriel Hjort Blindell. 2016. *Instruction Selection - Principles, Methods, and Applications*. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-319-34019-7

[11] Wei-Ting Jonas Chan, Andrew B. Kahng, Siddhartha Nath, and Ichiro Yamamoto. 2014. 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* (Seoul, South Korea) *(ICCD '14)*. IEEE Computer Society, New York, NY, USA, 153–160. https://doi.org/10.1109/ICCD.2014.6974675

[12] Kumar Chellapilla, Sidd Puri, and Patrice Simard. 2006. High Performance Convolutional Neural Networks for Document Processing. In *Proceedings of the Tenth International Workshop on Frontiers in Handwriting Recognition*, Guy Lorette (Ed.). Université de Rennes 1, La Baule (France), 6 pages.

[13] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv:1512.01274 [cs.DC]

[14] 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. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation* (Carlsbad, CA, USA) *(OSDI '18)*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, USA, 579–594.

[15] Yu-Hsin Chen, Tushar Krishna, Joel S. Emer, and Vivienne Sze. 2017. Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks. *IEEE J. Solid State Circuits* 52, 1 (2017), 127–138. https://doi.org/10.1109/JSSC.2016.2616357

[16] Zhi Chen, Cody Hao Yu, Trevor Morris, Jorn Tuyls, Yi-Hsiang Lai, Jared Roesch, Elliott Delaye, Vin Sharma, and Yida Wang. 2021. Bring Your Own Codegen to Deep Learning Compiler. arXiv:2105.03215 [cs.LG]

[17] Canadian Institute for Advanced Research 2009. *The CIFAR-10 dataset*. Canadian Institute for Advanced Research. http://www.cs.toronto.edu/~kriz/cifar.html

[18] Apple Inc. 2022. *CoreML: Integrate Machine Learning Models Into Your App*. Apple Inc. https://developer.apple.com/documentation/coreml

[19] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Fei-Fei Li. 2009. ImageNet: A large-scale hierarchical image database. In *Proceedings of the 2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Miami, FL, USA) *(CVPR '09)*. IEEE, New York, NY, USA, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[20] Nachum Dershowitz. 1993. A Taste of Rewrite Systems. In *Functional Programming, Concurrency, Simulation and Automated Reasoning (Lecture Notes in Computer Science, Vol. 693)*, Peter E. Lauer (Ed.). Springer, Berlin, Heidelberg, 199–228. https://doi.org/10.1007/3-540-56883-2_11

[21] João Dias and Norman Ramsey. 2010. Automatically Generating Instruction Selectors Using Declarative Machine Descriptions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 403–416. https://doi.org/10.1145/1706299.1706346

[22] Zhenman Fang, Farnoosh Javadi, Jason Cong, and Glenn Reinman. 2019. Understanding Performance Gains of Accelerator-Rich Architectures. In *Proceedings of the 30th IEEE International Conference on Application-specific Systems, Architectures and Processors* (New York, NY, USA) *(ASAP '19)*. IEEE, New York, NY, USA, 239–246. https://doi.org/10.1109/ASAP.2019.00013

[23] Alex Graves and Navdeep Jaitly. 2014. Towards End-To-End Speech Recognition with Recurrent Neural Networks. In *Proceedings of the 31st International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 32)*, Eric P. Xing and Tony Jebara (Eds.). PMLR, Bejing, China, 1764–1772. https://proceedings.mlr.press/v32/graves14.html

[24] Peng Gu, Xinfeng Xie, Yufei Ding, Guoyang Chen, Weifeng Zhang, Dimin Niu, and Yuan Xie. 2020. IPIM: Programmable in-Memory Image Processing Accelerator Using near-Bank Architecture. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, USA, 804–817. https://doi.org/10.1109/ISCA45697.2020.00071

[25] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. 2016. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO-49)*. IEEE Press, New York, NY, USA, Article 56, 13 pages.

[26] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding Sources of Inefficiency in General-Purpose Chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture* (Saint-Malo, France) *(ISCA '10)*. Association for Computing Machinery, New York, NY, USA, 37–47. https://doi.org/10.1145/1815961.1815968

[27] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, New York, NY, USA, 243–254. https://doi.org/10.1109/ISCA.2016.30

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *Proceedings of the 2016 IEEE Conference on Computer Vision and Pattern Recognition* (Las Vegas, NV, USA) *(CVPR)*. IEEE Computer Society, New York, NY, USA, 770–778. https://doi.org/10.1109/CVPR.2016.90

[29] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861 [cs.CV]

[30] Bo-Yuan Huang, Sayak Ray, Aarti Gupta, Jason M. Fung, and Sharad Malik. 2018. Formal Security Verification of Concurrent Firmware in SoCs Using Instruction-Level Abstraction for Hardware. In *Proceedings of the 55th Annual Design Automation Conference* (San Francisco, California) *(DAC '18)*. Association for Computing Machinery, New York, NY, USA, Article 91, 6 pages. https://doi.org/10.1145/3195970.3196055

[31] Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. 2019. ILAng: A Modeling and Verification Platform for SoCs Using Instruction-Level Abstractions. In *Proceedings of the 25th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2019)* (Prague, Czech Republic) *(Lecture Notes in Computer Science, Vol. 11427)*, Tomás Vojnar and Lijun Zhang (Eds.). Springer, Berlin, Heidelberg, 351–357. https://doi.org/10.1007/978-3-030-17462-0_21

[32] Bo-Yuan Huang, Hongce Zhang, Pramod Subramanyan, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Instruction-Level Abstraction (ILA): A Uniform Specification for System-on-Chip (SoC) Verification. *ACM Trans. Des. Autom. Electron. Syst.* 24, 1, Article 10 (Dec. 2018), 24 pages. https://doi.org/10.1145/3282444

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

[34] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2017. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. arXiv:1712.05877 [cs.LG]

[35] Rajeev Joshi, Greg Nelson, and Keith Randall. 2002. Denali: A Goal-Directed Superoptimizer. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation* (Berlin, Germany) *(PLDI '02)*. Association for Computing Machinery, New York, NY, USA, 304–314. https://doi.org/10.1145/512529.512566

[36] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A Domain-Specific Supercomputer for Training Deep Neural Networks. *Commun. ACM* 63, 7 (jun 2020), 67–78. https://doi.org/10.1145/3360307

[37] 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. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/3079856.3080246

[38] Ville Korhonen, Pekka Jääskeläinen, Matias Koskela, Timo Viitanen, and Jarmo Takala. 2015. Rapid Customization of Image Processors Using Halide. In *Proceedings of the 2015 IEEE Global Conference on Signal and Information Processing (GlobalSIP '15)*. IEEE, USA, 629–633.

[39] Smail Kourta, Adel Abderahmane Namani, Fatima Benbouzid-Si Tayeb, Kim Hazelwood, Chris Cummins, Hugh Leather, and Riyadh Baghdadi. 2022. Caviar: An e-Graph Based TRS for Automatic Code Optimization. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC '22)*. Association for Computing Machinery, New York, NY, USA, 54–64. https://doi.org/10.1145/3497776.3517781

[40] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. 2019. 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* (Seaside, CA, USA) *(FPGA '19)*. Association for Computing Machinery, New York, NY, USA, 242–251. https://doi.org/10.1145/3289602.3293910

[41] Yi-Hsiang Lai, Ecenur Ustun, Shaojie Xiang, Zhenman Fang, Hongbo Rong, and Zhiru Zhang. 2021. Programming and Synthesis for Software-Defined FPGA Acceleration: Status and Future Prospects. *ACM Trans. Reconfigurable Technol. Syst.* 14, 4, Article 17 (Sept. 2021), 39 pages. https://doi.org/10.1145/3469660

[42] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques A. Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization* (Seoul, South Korea) *(CGO '21)*, Jae W. Lee, Mary Lou Soffa, and Ayal Zaks (Eds.). IEEE, New York, NY, USA, 2–14. https://doi.org/10.1109/CGO51591.2021.9370308

[43] Dongkwon Lee, Woosuk Lee, Hakjoo Oh, and Kwangkeun Yi. 2023. Optimizing Homomorphic Evaluation Circuits by Program Synthesis and Time-Bounded Exhaustive Search. *ACM Trans. Program. Lang. Syst.* 45, 3, Article 16 (sep 2023), 37 pages. https://doi.org/10.1145/3591622

[44] Jiajie Li, Yuze Chi, and Jason Cong. 2020. HeteroHalide: From image processing DSL to efficient FPGA acceleration. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, USA, 51–57.

[45] Yi Li, Aarti Gupta, and Sharad Malik. 2024. Exact Scheduling to Minimize Off-Chip Data Movement for Deep Learning Accelerators. In *Proceedings of the 29th Asia and South Pacific Design Automation Conference (ASP-DAC '24)*. IEEE. To appear.

[46] The Linux Foundation 2019. *ONNX: Open Neural Network Exchange*. The Linux Foundation. https://onnx.ai/

[47] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified Tensor-Program Optimization via High-Level Scheduling Rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (jan 2022), 28 pages. https://doi.org/10.1145/3498717

[48] Qiaoyi Liu, Dillon Huff, Jeff Setter, Maxwell Strange, Kathleen Feng, Kavya Sreedhar, Ziheng Wang, Keyi Zhang, Mark Horowitz, Priyanka Raina, et al. 2021. Compiling Halide Programs to Push-Memory Accelerators. arXiv:2105.12858 [cs.LG]

[49] John McCarthy. 1960. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. *Commun. ACM* 3, 4 (apr 1960), 184–195. https://doi.org/10.1145/367177.367199

[50] Jedidiah McClurg, Miles Claver, Jackson Garner, Jake Vossen, Jordan Schmerge, and Mehmet E. Belviranli. 2023. Optimizing Regular Expressions via Rewrite-Guided Synthesis. In *Proceedings of the International Conference on Parallel*

Architectures and Compilation Techniques (Chicago, Illinois) *(PACT '22)*. Association for Computing Machinery, New York, NY, USA, 426–438. https://doi.org/10.1145/3559009.3569664

[51] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. arXiv:1609.07843 [cs.CL]

[52] MLCommons n. d.. *MLPerf Benchmarks*. MLCommons. https://mlcommons.org

[53] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Q. Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2019. A Hardware-Software Blueprint for Flexible Deep Learning Specialization. *IEEE Micro* 39, 5 (2019), 8–16. https://doi.org/10.1109/MM.2019.2928962

[54] Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (April 1980), 356–364. https://doi.org/10.1145/322186.322198

[55] Julie L Newcomb, Andrew Adams, Steven Johnson, Rastislav Bodik, and Shoaib Kamil. 2020. Verifying and Improving Halide's Term Rewriting System with Program Synthesis. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[56] Robert Nieuwenhuis and Albert Oliveras. 2005. Proof-Producing Congruence Closure. In *Proceedings of the 16th International Conference on Term Rewriting and Applications (RTA 2005)* (Nara, Japan) *(Lecture Notes in Computer Science, Vol. 3467)*, Jürgen Giesl (Ed.). Springer, Berlin, Heidelberg, 453–468. https://doi.org/10.1007/978-3-540-32033-3_33

[57] Rachit Nigam, Samuel Thomas, Zhijing Li, and Adrian Sampson. 2021. A Compiler Infrastructure for Accelerator Generators. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 804–817. https://doi.org/10.1145/3445814.3446712

[58] 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. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Proceedings of the Annual Conference on Advances in Neural Information Processing Systems* (Vancouver, BC, Canada) *(NeurIPS '19)*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché Buc, Emily B. Fox, and Roman Garnett (Eds.). Curran Associates, Inc., USA, 8024–8035.

[59] Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Arvind, and Adam Chlipala. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 789–803. https://doi.org/10.1145/3445814.3446720

[60] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. 2017. Programming Heterogeneous Systems from an Image Processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)* 14, 3 (2017), 1–25.

[61] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. 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* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 519–530. https://doi.org/10.1145/2491956.2462176

[62] Norman Ramsey and João Dias. 2011. 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* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 575–586. https://doi.org/10.1145/1926385.1926451

[63] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David Brooks. 2016. Minerva: Enabling Low-Power, Highly-Accurate Deep Neural Network Accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA '16)*. IEEE Press, New York, NY, USA, 267–278. https://doi.org/10.1109/ISCA.2016.32

[64] 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. 2020. MLPerf Inference Benchmark. In *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture* (Virtual Event) *(ISCA '20)*. IEEE Press, New York, NY, USA, 446–459. https://doi.org/10.1109/ISCA45697.2020.00045

[65] Jared Roesch, Steven Lyubomirsky, Marisa Kirisame, Logan Weber, Josh Pollock, Luis Vega, Ziheng Jiang, Tianqi Chen, Thierry Moreau, and Zachary Tatlock. 2019. Relay: A High-Level Compiler for Deep Learning. arXiv:1904.08368 [cs.LG]

[66] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2019. MobileNetV2: Inverted Residuals and Linear Bottlenecks. arXiv:1801.04381 [cs.CV]

[67] Ruud Schellekens. 2020. *Automatically Scheduling Halide-HLS.* Master's thesis. Eindhoven University of Technology.

[68] Siemens n. d.. *Catapult High-Level Synthesis and Verification.* Siemens. https://eda.sw.siemens.com/en-US/ic/catapult-high-level-synthesis

[69] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. 2021. Pure Tensor Program Rewriting via Access Patterns (Representation Pearl). In *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming* (Virtual, Canada) *(MAPS '21)*. Association for Computing Machinery, New York, NY, USA, 21–31. https://doi.org/10.1145/3460945.3464953

[70] John A. Stratton, Jyothi Krishna Viswakaran Sreelatha, Rajiv Ravindran, Sachin Sudhakar Dake, and Jeevitha Palanisamy. 2020. Optimizing Halide for Digital Signal Processors. In *Proceedings of the 2020 IEEE Workshop on Signal Processing Systems (SiPS '20)*. IEEE, USA, 1–6. https://doi.org/10.1109/SiPS50750.2020.9195237

[71] Pramod Subramanyan, Bo-Yuan Huang, Yakir Vizel, Aarti Gupta, and Sharad Malik. 2018. Template-Based Parameterized Synthesis of Uniform Instruction-Level Abstractions for SoC Verification. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 37, 8 (2018), 1692–1705. https://doi.org/10.1109/TCAD.2017.2764482

[72] 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. 2021. 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)*. IEEE, New York, NY, USA, 158–160. https://doi.org/10.1109/ISSCC42613.2021.9366062

[73] Thierry Tambe, En-Yu Yang, Zishen Wan, Yuntian Deng, Vijay Janapa Reddi, Alexander Rush, David Brooks, and Gu-Yeon Wei. 2020. 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* (Virtual Event, USA) *(DAC '20)*. IEEE Press, USA, Article 51, 6 pages.

[74] Mingxing Tan and Quoc V. Le. 2019. EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning* (Long Beach, California, USA) *(ICML '19, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Atlanta, Georgia, USA, 6105–6114. http://proceedings.mlr.press/v97/tan19a.html

[75] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

[76] Hugo Touvron, Piotr Bojanowski, Mathilde Caron, Matthieu Cord, Alaaeldin El-Nouby, Edouard Grave, Armand Joulin, Gabriel Synnaeve, Jakob Verbeek, and Hervé Jégou. 2021. ResMLP: Feedforward networks for image classification with data-efficient training. arXiv:2105.03404 [cs.CV]

[77] 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. 2020. fault: A Python Embedded Domain-Specific Language for Metaprogramming Portable Hardware Verification Components. In *Proceedings of the 32nd International Conference on Computer Aided Verification (CAV 2020)* (Los Angeles, CA, USA) *(Lecture Notes in Computer Science, Vol. 12224)*, Shuvendu K. Lahiri and Chao Wang (Eds.). Springer, Berlin, Heidelberg, 403–414. https://doi.org/10.1007/978-3-030-53288-8_19

[78] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

[79] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2021. 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* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 874–886. https://doi.org/10.1145/3445814.3446707

[80] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[81] Veripool n. d.. *Verilator.* Veripool. https://www.veripool.org/verilator/

[82] Sander Vocke, Henk Corporaal, Roel Jordans, Rosilde Corvino, and Rick Nas. 2017. Extending Halide to Improve Software Development for Imaging DSPs. *ACM Trans. Archit. Code Optim.* 14, 3, Article 21 (aug 2017), 25 pages. https://doi.org/10.1145/3106343

[83] 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. 2019. 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* (Kyoto, Japan). IEEE, New York, NY, USA, 34. https://doi.org/10.23919/VLSIC.2019.8778002

[84] Deborah L. Whitfield and Mary Lou Soffa. 1997. An Approach for Exploring Code Improving Transformations. *ACM Trans. Program. Lang. Syst.* 19, 6 (Nov. 1997), 1053–1084. https://doi.org/10.1145/267959.267960

[85] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. https://doi.org/10.1145/3434304

[86] PyTorch Team 2020. *Word-level language modeling RNN.* PyTorch Team. https://github.com/pytorch/examples/tree/master/word_language_model

[87] Xilinx Inc. n. d.. *The Xilinx Software Development Kit (XSDK).* Xilinx Inc. https://www.xilinx.com/products/design-tools/embedded-software/sdk.html

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

[89] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-x: An Accelerator for Sparse Neural Networks. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture* (Taipei, Taiwan) *(MICRO '49)*. IEEE Press, New York, NY, USA, Article 20, 12 pages. https://doi.org/10.1109/MICRO.2016.7783723

[90] Bill Zorn. 2021. *Rounding.* Ph. D. Dissertation. University of Washington.