

TensorLift: Automatic Extraction of Tensor-Level ISA Semantics from Accelerator RTL via MLIR Semantic Lifting

Ruijie Gao¹ Haoran Jin¹ Jirong Yang^{2*} Nathaniel Bleier¹

¹Computer Science and Engineering, University of Michigan, Ann Arbor, MI

²Computer Science, University of Texas at Austin, Austin, TX

{ruijieg, allenjin, nbleier}@umich.edu polarisyjr@utexas.edu

Abstract

Most proposed tensor accelerators lack well-documented ISAs and compiler backends, and are exercised only through hand-written kernels covering a handful of operators. Recent work (TAIDL, ACT) shows that a tensor-level ISA specification is enough to generate complete software stacks automatically. Writing that specification, however, remains a manual, expert-driven process.

We present TensorLift, the first end-to-end MLIR-based pipeline that lifts RTL-extracted accelerator semantics to TAIDL-like tensor ISA specifications. Building on prior architecture-level model extraction that yields bit-level IR, an 8-pass MLIR pipeline progressively recovers tensor structure—MAC idioms, saturation semantics, multi-dimensional buffer organizations, and layout transformations—and emits specifications the ACT ecosystem consumes directly.

On Gemmini and VTA, TensorLift captures every hardware instruction the two designs decode across 156 MLIR files, reducing the extracted bit-level MLIR by 24.8% and 41.2% overall and by up to 92.9% on a processing element, with controller modules retaining irreducible control logic. It recovers hardware features the hand-written reference omits—multi-bank DMA configuration, pooling, and im2col—and lifts VTA unmodified. Core compute and data-movement semantics are proven equivalent to the RTL-extracted model by Z3 SMT, the remainder validated against golden simulator data. Fed into ACT, the extracted specification yields a compiler backend at parity with hand-written Gemmini kernels (1.014× geometric mean), giving an automated path from RTL to a working software stack.

CCS Concepts

• **Hardware** → **Emerging architectures**; *Hardware description languages and compilation*; **Emerging languages and compilers**.

Keywords

Tensor Accelerators, ISA Extraction, MLIR, Semantic Lifting

ACM Reference Format:

Ruijie Gao, Haoran Jin, Jirong Yang, and Nathaniel Bleier. 2026. TensorLift: Automatic Extraction of Tensor-Level ISA Semantics from Accelerator RTL via MLIR Semantic Lifting. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '26)*, November 08–12, 2026, San Jose, CA, USA. ACM, New York, NY, USA, 9 pages. <https://doi.org/10.1145/3831252.3833946>



This work is licensed under a Creative Commons Attribution 4.0 International License. ICCAD '26, San Jose, CA, USA

© 2026 Copyright held by the owner/author(s).

ACM ISBN 979-8-4007-2873-0/2026/11

<https://doi.org/10.1145/3831252.3833946>

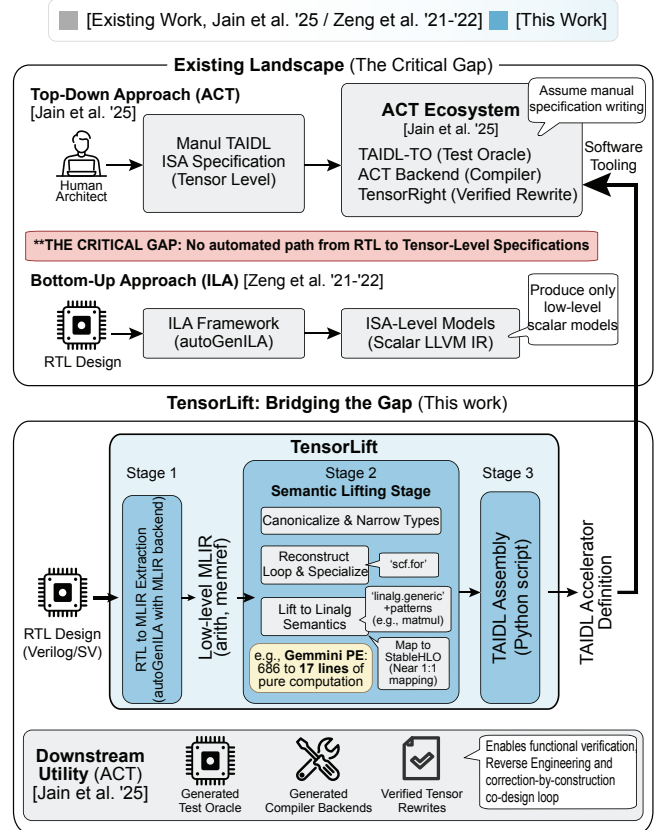


Figure 1: Overview of the existing landscape. TensorLift bridges the gap between bottom-up RTL extraction (scalar-level models) and top-down compiler automation (which assumes a hand-written TAIDL specification).

1 Introduction

The rapid growth of deep learning workloads has driven a proliferation of specialized tensor accelerators. Designs such as Gemmini [12], MAERI [23], SIGMA [33], Eyeriss [5], NVDLA [7], and FEATHER [38] explore diverse microarchitectural innovations—systolic arrays, reconfigurable interconnects, flexible dataflow engines—for orders-of-magnitude gains in performance and energy efficiency over general-purpose processors. Yet only a handful of vendor-backed platforms—Google’s TPU [22] with XLA [34], Intel AMX [18] with oneDNN [32], and NVIDIA GPUs with

*Work done while at the University of Michigan.

Table 1: Software stack readiness of representative open-source tensor/DNN accelerators (● available, ○ partial, ○ unavailable). TensorLift targets accelerators with open RTL but no ISA specification.

Accelerator	ISA Spec	Compiler Backend	Open RTL	Test Oracle
MAERI [23]	○	○	●	○
SIGMA [33]	○	○	●	○
Eyeriss [5]	○	○	●	○
MAGNet [41]	○	○	●	○
DNNBuilder [45]	○	○	●	○
FlexASR [36]	○	○	●	○
SPAGHETTI [13]	○	○	●	○
NVDLA [7]	●	●	●	●
Gemmini [12]	●	●	●	●
VTA [30]	●	●	●	●
DnnWeaver [35]	○	●	●	○
Google TPU [22]	●	●	○	●
Intel AMX [18]	●	●	○	●

CUDA/cuDNN [6]—enjoy mature compiler toolchains connecting them to frameworks like PyTorch, JAX [3], and TensorFlow. As Table 1 summarizes, accelerators with end-to-end compiler support number fewer than five, while over a dozen open-source designs lack any usable backend and can be exercised only through hand-written kernel libraries that cover a narrow set of operators and miss cross-layer fusion—leaving most proposed hardware unable to run real ML models end-to-end.

The root cause lies upstream of the compiler: the ISA semantics of most accelerators have never been formally defined. Where the CPU ISA is the foundational hardware/software contract, accelerator ISA information is scattered across RTL source, informal documentation, or undocumented designer knowledge [16, 17]. Figure 1 summarizes the landscape: the top-down ACT ecosystem assumes a hand-written TAIDL specification, while the bottom-up ILA framework produces only scalar-level models, leaving no automated path from RTL to tensor-level semantics.

Recent work closes this gap from the ISA specification downward: TAIDL [20] introduced a domain-specific language for defining tensor accelerator ISAs, and ACT [21] showed that compiler backends can be generated automatically from such specifications (§2.2). The prerequisite is critical, though: someone must first write the TAIDL specification by hand, which requires deep understanding of the accelerator’s architectural state, instruction encoding, and tensor-level semantics.

This bottleneck is well-documented: writing formal accelerator models has been called “tedious and error-prone” [44] and “a significant burden for accelerators which may not have the volume or life-span compared to processors” [43]. RTL designs iterate rapidly, while TAIDL and ACT automate everything downstream of the ISA spec but cannot function without one. Prior efforts such as ILA [16] and 3LA [15] formalized accelerator interfaces but still require hand-authored models, and LLM-based EDA agents [26, 47] that depend on machine-readable hardware semantics [14, 42] cannot operate when such semantics do not exist.

We present TensorLift, the first end-to-end MLIR-based pipeline that lifts RTL-extracted accelerator semantics to TAIDL-like tensor ISA specifications. Building on prior architecture-level model extraction [43, 44] that produces bit-level MLIR from flattened RTL, TensorLift employs an 8-pass MLIR pipeline that progressively recovers high-level tensor structure—from bit-manipulation

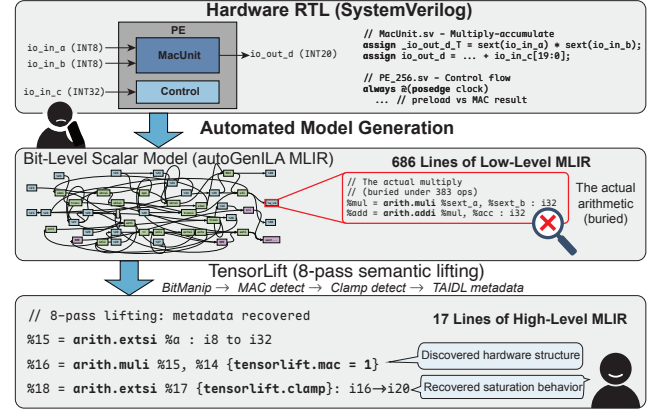


Figure 2: TensorLift’s semantic lifting on a Gemini PE: from RTL source (top) through 686 lines of bit-level MLIR (middle) to 49 lines of lifted MLIR (bottom, 92.9% reduction).

canonicalization through idiom detection and loop reconstruction to structured metadata emission—connecting gate-level hardware to the tensor-level semantics tools like TAIDL and ACT require (Figure 2).

We evaluate TensorLift on two open-source accelerators: Berkeley’s Gemmini (systolic array, 16×16, INT8) and TVM’s VTA. Lifting reduces the extracted bit-level MLIR by 24.8% on Gemmini and 41.2% on VTA overall, rising to 92.9% on compute-dominated modules; it discovers hardware features omitted from hand-written references and generalizes across both accelerators without modification. The core compute and data-movement semantics are proven equivalent to the RTL-extracted model via Z3 [8] SMT, and the remaining lifted operations are validated against golden functional-simulator data. Fed into ACT, the extracted specification yields a compiler backend that matches hand-written Gemmini kernels (1.014× geometric mean) on workloads including ResNet-50 and MobileNet, with no hand-written specification anywhere in the path.

Filling this first gap in the RTL-to-framework pipeline (Figure 1) removes the step that has required the most manual effort, and completes an automated path from RTL through ACT to an XLA-compatible backend.

In summary, this paper makes the following contributions:

- We propose the first end-to-end MLIR-based pipeline that lifts RTL-extracted accelerator semantics to TAIDL-like tensor ISA specifications, addressing the manual specification bottleneck that blocks adoption of automated compiler generation tools such as TAIDL and ACT.
- We design an 8-pass MLIR pipeline that recovers high-level tensor structure from bit-level arithmetic in four phases: canonicalization, idiom detection, loop reconstruction, and metadata emission.
- We evaluate TensorLift on Gemmini and VTA, reducing extracted bit-level MLIR by 24.8% and 41.2% overall and by up to 92.9% on compute modules, discovering hardware features absent from hand-written references, and formally verifying the core compute and data-movement semantics via Z3 SMT proofs. The same pipeline generalizes across both accelerators without modification.

- We complete the path from RTL to ML framework, contributing HLO frontend support, convolution pattern matching, and multi-layer memory allocation to ACT, so the extracted specification compiles end-to-end to binaries at parity with hand-written Gemmini kernels (1.014× geometric mean) on ResNet-50, MobileNet, and others.

2 Background

2.1 Hardware-Software Interfaces in Tensor Accelerators

Tensor accelerators occupy a different point in the design space than scalar or vector processors: rather than executing fine-grained instruction streams over flat register files and cache-coherent memory, they expose *coarse-grained* operations over multi-dimensional tensors and replace implicit caches with *explicitly managed* on-chip memories (scratchpads, accumulator banks). Gemmini [12], Berkeley’s open-source systolic-array generator, is representative: it attaches to a Rocket core [2] via the Rocket Custom Coprocessor (RoCC) interface [1] with non-standard RISC-V custom instructions; its programmer-visible state is a row-addressed scratchpad and an accumulator; and its ISA exposes only configuration, DMA data-movement, and matrix-multiply execution instructions—every operation simultaneously specifies *what* to compute, *where* operands live in a private memory hierarchy, and *how* data should be staged.

This shifts substantial work onto the compiler. ML frameworks feed tensor IR (StableHLO, Linalg, TOSA) into backends such as XLA [34], TVM [4], or IREE [37], which must tile and fuse loop nests to fit scratchpad capacity, schedule explicit DMA transfers, allocate scratchpad regions, and orchestrate double-buffering—all presupposing a precise model of what each instruction does to which piece of tensor state. Lowering a single XLA dot $C=A \cdot B$ beyond Gemmini’s 16×16 tile, for instance, requires emitting `config_ex/config_ld/config_st` to set dataflow and stride parameters, `mvin/mvin2` to stage tiles of A and B , `preload` followed by `compute_preloaded` (or `compute_accumulated`) to fire the systolic array per output tile, and `mvout` to drain results—which in turn requires knowing, for every instruction, which tensor tile it touches, which scratchpad region it occupies, and how its output composes with what follows.

Most accelerators fall short here: such a mapping presupposes a *machine-readable* ISA specifying both the binary encoding and the *tensor-level semantics* of each instruction. For Gemmini, semantics live informally in prose documentation, the Chisel RTL, and a hand-written Spike functional model; most open-source accelerators offer only RTL and perhaps a driver, so building a backend today still means *reverse-engineering the RTL by hand*.

2.2 Automated Compiler Generation and the Specification Bottleneck

The main barrier to end-to-end software support is thus not only backend engineering effort but the lack of a precise, machine-readable Instruction Set Architecture (ISA) specification. TAIDL [20] (Tensor Accelerator ISA Definition Language) addresses this with a domain-specific language for describing accelerator data models and instruction semantics at the tensor

level, expressing instruction behavior with operators close to the XLA High Level Optimizer (XLA-HLO) [34]—reshape, transpose, convert, dot, add. This abstraction is high enough to capture programmer-visible tensor semantics compactly, yet close enough to tensor compiler intermediate representations to support downstream automation, so a representative specification reads less like microarchitectural pseudocode than like a tensor transformation:

```
# Data model: scratchpad and accumulator
acc.add_data_model("sp", "1024×DIM", f"{DIM}xs8")
acc.add_data_model("bias", f"{DIM}", f"{DIM}xs32")

# Instruction with register operands
instr = acc.add_instruction("matmul8_compute",
    ["rs1", "rs2"])
instr.add_semantics(f"""
%A.8: {DIM}x{DIM}xs8 ← sp[ea.rs1:..., 0:{DIM}];
%B.8: {DIM}x{DIM}xs8 ← sp[ea.rs2:..., 0:{DIM}];
%0.8: {DIM}x{DIM}xs8 ← sp[es.rs1:..., 0:{DIM}];
%A.32: {DIM}x{DIM}xs32 = convert(%A.8);
%B.32: {DIM}x{DIM}xs32 = convert(%B.8);
%0.32: {DIM}x{DIM}xs32 = convert(%0.8);
%dot: {DIM}x{DIM}xs32 = dot(%A.32, %B.32),
      lhs_contracting_dims={{1}},
      rhs_contracting_dims={{0}};
%C.32: {DIM}x{DIM}xs32 = add(%dot, %0.32);
%C.cl: {DIM}x{DIM}xs32 = clamp(-128, %C.32, 127);
%0.8: {DIM}x{DIM}xs8 = convert(%C.cl);
%0.8 → sp[es.rs2, 0];
""")
```

Listing 1: Simplified TAIDL specification for a Gemmini multiply-accumulate instruction (adapted from [19]); the full specification additionally includes dataflow routing and state constraint logic.

The author must specify the tensor buffers the accelerator exposes, the instruction operands, and the exact tensor-level effect of the instruction—but not the underlying Register-Transfer Level (RTL) control or datapath implementation. Published examples [19] model even Intel Advanced Matrix Extensions (Intel AMX) [18] tdpbusd in the same style, composing layout transformations (reshape, transpose), type conversion, tensor contraction, accumulation, and write-back.

Built on such specifications, ACT [21] (Accelerator Compiler Toolkit) automates everything after the ISA description, generating compiler backends by combining equality saturation for instruction selection with constraint-programming-based memory allocation—an ISA-driven flow in which much of the backend logic is synthesized rather than hand-written. This is especially attractive for tensor accelerators, whose parameterized tensor tiles and explicitly managed on-chip memories are difficult to support with conventional backend-generation techniques.

The automation boundary is also the key limitation: TAIDL and ACT automate everything downstream of the ISA specification but not its creation—someone must still write the TAIDL model by hand. This front-end burden is the gap TensorLift addresses.

2.3 RTL Extraction and MLIR-Based Semantic Lifting

The natural baseline for recovering ISA semantics from hardware is architecture-level model extraction from RTL, and the relevant starting point is autoGenILA [43, 44]: it identifies the architectural state variables (ASVs) that persist across instructions, then derives each instruction’s state update functions directly from RTL, removing the need to hand-write an Instruction-Level Abstraction (ILA) [16]

model and yielding an interface consistent with the hardware by construction. That is its role in TensorLift: the extraction backend that converts accelerator RTL into per-instruction semantic functions.

These extracted models, however, are still too low-level for tensor compiler generation: autoGenILA captures hardware behavior faithfully, but its output is a flattened, bit-level scalar model in which tensor intent is obscured by the structure of the datapath [43, 44]. As Figure 2 illustrates, a single Gemini processing element may initially appear as hundreds of primitive operations—dominated by sign-extension chains, truncations, bitwise composition, and scalar arithmetic—before later passes recover the underlying dot + add + clamp semantics. Such an output suits bit-accurate reasoning but not TAIDL [20], which requires tensor-visible semantics: buffer reads and writes, structured layout transformations, multiply-accumulate patterns, reductions, and saturation behavior. autoGenILA thus solves the *extraction* problem, but not the *semantic abstraction* problem.

MLIR (Multi-Level Intermediate Representation) is what makes the missing step practical. MLIR offers no hardware-specific frontend that would help here. What it offers is multi-level, progressive abstraction within one compiler framework [25], together with the right semantic stepping stones between bit-level arithmetic and tensor-level intent: `arith` for scalar operations, `memref` for explicit memory accesses, `scf` for structured control flow, and `linalg` for tensor computations [27–29]. TensorLift materializes the extracted semantics as low-level `arith`/`memref` MLIR and raises them through structured, rewrite-driven and partially legalized passes [25, 29] rather than jumping from RTL-level behavior to tensor semantics in one step (Section 3).

3 TensorLift Pipeline

TensorLift transforms an unannotated RTL design into a TAIDL specification through three stages (Figures 1 and 3): **Stage 1 (§3.1)** symbolically extracts per-instruction behavior from the RTL into bit-level MLIR; **Stage 2 (§3.2)** progressively lifts that bit-level representation to tensor-level semantics via an eight-pass rewrite pipeline; **Stage 3 (§3.3)** assembles the lifted MLIR into a TAIDL specification with XLA-HLO-style semantics. The entire pipeline takes a directory of SystemVerilog files and a top-module name as input, and produces a complete TAIDL specification consumable by the ACT ecosystem.

3.1 Stage 1: RTL-to-MLIR Extraction

Stage 1 builds on autoGenILA [43, 44], which symbolically unrolls a Verilog design for each (instruction, architectural state variable) pair into a function mapping current state and inputs to next state—provably bit-equivalent to the RTL by construction. Its LLVM IR backend, however, destroys two structural properties that tensor-level reasoning requires: conditional updates are lowered into branches and phi nodes, erasing the if/else structure that encodes hardware mode selection, and RTL signal names—which identify whether an input carries activations, weights, or an accumulator—are discarded.

TensorLift replaces that backend with an MLIR code generator preserving both: conditional updates become `scf.if` regions,

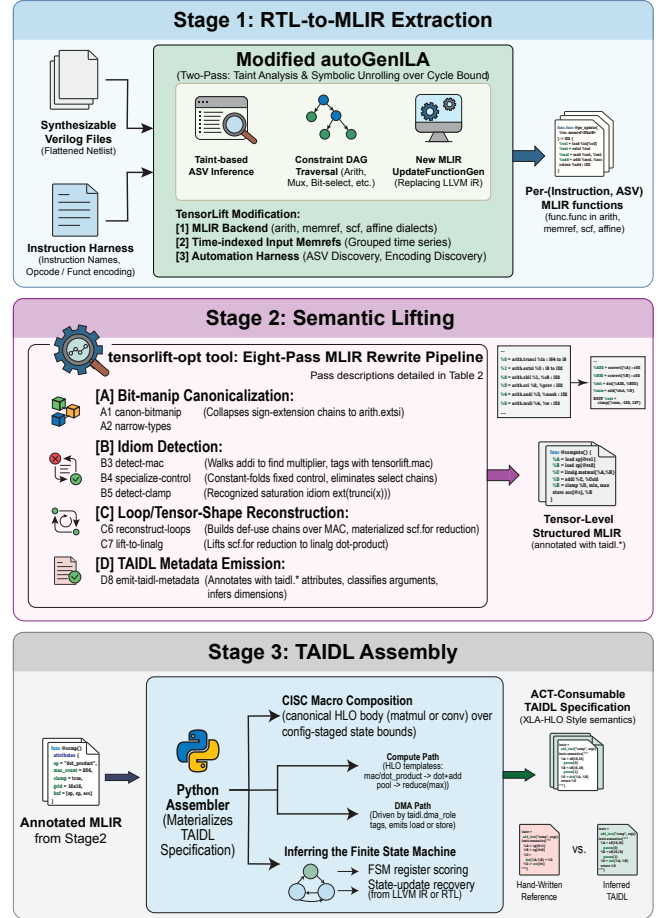


Figure 3: TensorLift’s three-stage pipeline: RTL-to-MLIR extraction (Stage 1), eight-pass semantic lifting (Stage 2), and TAIDL assembly with compute/DMA paths and FSM recovery (Stage 3).

memory accesses remain indexed `memref` loads, and RTL signal names are attached as structured metadata that downstream passes consult to classify arguments semantically. A second design choice packs the time series of each input signal across unroll cycles into a single `memref` argument rather than hundreds of independent scalars—the grouping that later lets loop reconstruction (§3.2) recover MAC chains as indexable reductions. An automation layer generates flattening scripts, enumerates architectural state variables, and discovers instruction encodings from the RTL decoder, reducing per-design manual input to two items; the original LLVM IR backend is retained unchanged as a verification oracle (§4).

The output is a corpus of MLIR files, one per (instruction, ASV) pair, provably bit-equivalent to the RTL. As Figure 2 illustrates, even a single Gemini PE produces 686 lines at this stage, its core multiply-accumulate buried under hundreds of sign-extension and bit-packing operations.

Table 2: TensorLift’s eight-pass semantic-lifting pipeline; each pass is named with its `tensorlift-opt` flag.

Phase / Pass	What it does
A1 canon-bitmanip	Collapses the bit-by-bit sign-extension chains <code>autoGenLLA</code> emits for Verilog <code>\$signed</code> contexts into a single <code>arith.extsi</code> ; the dominant source of code reduction on PEs.
A2 narrow-types	Folds redundant <code>trunci/ext</code> round trips left after canonicalization, preserving the <code>extsi(trunci())</code> idioms that Pass B5 recovers as saturation.
B3 detect-mac	Walks each <code>addi</code> back through width casts to a multiplier and recovers its pre-extension inputs; tags the pair <code>tensorlift.mac</code> when operand widths are hardware-realistic, filtering bit-packing artifacts.
B4 specialize-control	Constant-folds loads of the instruction’s fixed control inputs (from the descriptor that drove Stage 1), letting canonicalization eliminate the select chains between hardware modes.
B5 detect-clamp	Recognizes the hardware fixed-point saturation idiom <code>ext(trunci(x))</code> and annotates the surviving extension with the recovered clamp range and signedness.
C6 reconstruct-loops	Builds a use-def chain over MAC-annotated additions whose accumulator input is the previous MAC’s output; for chains of length ≥ 2 , materializes it as an <code>scf.for</code> for reduction with one <code>iter_arg</code> .
C7 lift-to-linalg	Verifies that a reconstructed <code>scf.for</code> matches the canonical dot-product shape (single <code>iter_arg</code> , two <code>memref</code> loads at the induction variable, multiply-add-yield) and tags it with <code>linalg.op = "dot_product"</code> .
D8 emit-taidl-metadata	Classifies each <code>memref</code> argument by its load/store footprint, labels scalar arguments as control attributes, infers grid dimensions from coordinate suffixes in target ASV names, and emits the closed set of <code>taidl.*</code> attributes Stage 3 consumes.

3.2 Stage 2: Semantic Lifting

Stage 2 is an eight-pass MLIR pipeline that progressively recovers tensor-level semantics from the bit-level Stage 1 output. The passes fall into four phases (Table 2).

Phase A (canonicalization) collapses the bit-by-bit sign-extension chains arising from Verilog `$signed` contexts and folds redundant width round-trips—the dominant source of code reduction. **Phase B (idiom detection)** recovers multiply-accumulate by tracing additions back through width casts to a multiplier, dead hardware modes by constant-folding the instruction’s fixed control inputs, and fixed-point saturation from the `ext(trunci(x))` idiom. **Phase C (loop reconstruction)** chains MAC-annotated operations into `scf.for` for reductions and tags those matching the canonical dot-product shape. **Phase D (metadata emission)** classifies each `memref` argument by access pattern, labels scalar arguments as control attributes, and emits the `taidl.*` attributes Stage 3 consumes.

Two properties follow from this design. First, with the exception of loop reconstruction, passes *annotate* rather than *rewrite*: they attach attributes to existing operations, and downstream passes key off those attributes. An instruction that fails to match a template simply receives no annotation, and Stage 3 falls back to opaque semantics rather than producing incorrect TAIDL. Second, the pipeline is accelerator-agnostic: the structural cues it relies on—sign-extension chains, MAC fan-in, fixed-point clamp idioms, accumulator def-use chains—are properties of common RTL synthesis patterns, not of any particular ISA. The same binary lifts Gemmini’s PE and DMA controllers, and VTA’s TensorGemm and TensorAlu without modification.

On the running Gemmini PE example, the eight passes reduce 686 lines of bit-level MLIR to 49, of which only 17 encode the computation core `clamp(dot(%A,%B)+%C)`.

Scope and limitations. These cues are artifacts of how RTL synthesizes arithmetic, so they do not depend on array topology: one binary lifts both a systolic mesh (Gemmini) and an adder-tree GEMM

engine (VTA). Extending that reasoning to classes we do not evaluate: a reconfigurable-interconnect design [23, 33, 38] still exposes MAC fan-in and clamp idioms in its multiply and reduction nodes, but its routing fabric is switch and credit logic with no numeric output, so Phase B finds nothing to annotate and the fabric lifts as control state rather than a tensor operation; a sparse design splits the same way, its value path retaining MAC structure while the coordinate machinery (index merge, run-length decode, descriptor sequencing) stays at the bit level. In both cases the pipeline emits opaque semantics instead of incorrect TAIDL: passes only annotate, so an unmatched instruction simply receives no annotation. Synthesis-optimized RTL is the clearest limitation: retiming, register fusion, and technology mapping can redistribute the sign-extension chains and accumulator def-use edges Phases A and C depend on, and we do not claim coverage of designs delivered in that form.

3.3 Stage 3: TAIDL Assembly

Stage 3 assembles the annotated MLIR into a complete TAIDL specification (Listing 1), merging the per-(instruction, ASV) functions back into per-instruction groups and dispatching on the recognized tensor operation through two paths. The *compute path* maps each tensor op to an XLA-HLO template—`dot_product` to `convert + dot + add` (with optional clamp), `pool` to `reduce(max)`, `im2col_matmul` to `reshape + dot + add`—inserting the corresponding HLO operation when the RTL encodes an activation function (e.g., `maximum(result, 0)` for ReLU). The *DMA path* classifies memory-port roles (DRAM vs. scratchpad address) from the annotated metadata and emits a load or store body accordingly.

For CISC-style loop macros that orchestrate primitive-instruction sequences behind one invocation (e.g., Gemmini’s `loop_ws`), TensorLift recovers the primitive vocabulary and loop-bound configuration registers from the controller RTL, then composes the primitive tensor operation over those bounds into a whole-program HLO body. Ordering constraints—that `compute_preloaded` may fire only after `preload`—are recovered by analyzing FSM register updates across instruction groups and folded in as structured metadata.

4 Evaluation

We evaluate TensorLift along four axes: how effectively the pipeline compresses bit-level MLIR into tensor-level semantics (§4.2); whether the lifted semantics are provably equivalent to the RTL (§4.3); how the extracted specification compares to a hand-written reference (§4.4); and whether it can drive end-to-end compiler backend generation (§4.5).

4.1 Experimental Setup

We target two open-source tensor accelerators. **Gemmini** [12] is a systolic-array accelerator (16×16, INT8 MAC) generated via Chipyard [1] in the `GemminiRocketConfig` configuration, exposing three hardware controllers (ExecuteController, LoadController, StoreController) with 11 hardware instructions across configuration, DMA data movement, and matrix execution. **VTA** [30] is TVM’s Versatile Tensor Accelerator in the `DefaultDe10Config` configuration (16-element GEMM engine, ALU, INT8), with four

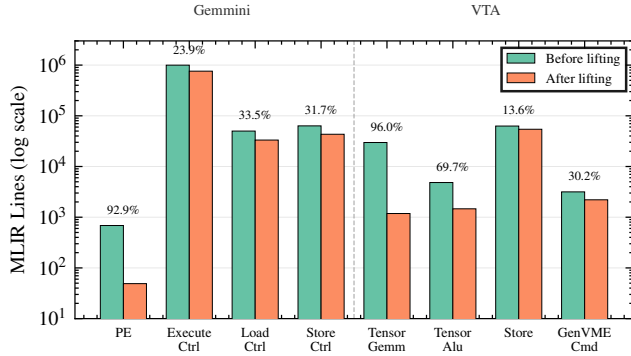


Figure 4: MLIR line counts before and after 8-pass lifting, across all Gemmini and VTA modules; percentages indicate reduction.

Table 3: Cross-accelerator lifting summary. “Before” and “After” report total MLIR lines across all per-(instruction, ASV) files in each module.

Accelerator	Module	Files	Before	After	Reduction
Gemmini	PE (TileWithReset)	1	686	49	92.9%
	ExecuteController	80	997,760	759,360	23.9%
	LoadController	23	50,004	33,244	33.5%
	StoreController	23	63,313	43,269	31.7%
	Total	127	1,111,763	835,922	24.8%
VTA	TensorGemm	9	29,745	1,183	96.0%
	TensorAlu	10	4,832	1,465	69.7%
	Store	5	62,807	54,294	13.6%
	GenVMECmd	5	3,160	2,205	30.2%
	Total	29	100,544	59,147	41.2%
Combined		156	1,212,307	895,069	26.2%

datapath modules (TensorGemm, TensorAlu, Store, GenVMECmd). All formal verification uses Z3 [8] bitvector SMT solving, cycle-level validation the Gemmini Spike functional simulator [40], and the hand-written reference is the TAIDL MICRO 2025 artifact [19].

4.2 Semantic Lifting Effectiveness

Table 3 and Figure 4 report the results: TensorLift processes 156 MLIR files comprising 1,212,307 lines of bit-level MLIR, reducing them to 895,069 lines (26.2% overall).

The reduction varies sharply with module function. **Compute-dominated modules** compress most: the Gemmini PE collapses from 686 to 49 lines (92.9%), its computation core reduced to 17 lines encoding `clamp(dot(%A,%B)+%C)` once 383 bit-by-bit sign-extension operations from Verilog `$signed` contexts fold into single `arith.extsi` casts under Phase A; VTA’s TensorGemm goes further, its MAC register targets shrinking from 3,156 to 8 lines each (99.7%) and the module from 29,745 to 1,183 lines (96.0%). **Control-heavy modules** reduce moderately: Gemmini’s LoadController (33.5%) and StoreController (31.7%) retain address computation, FSM transitions, and instruction muxing—irreducible control logic—while VTA’s TensorAlu (69.7%) falls in between, its opcode muxing across five operations (MIN, MAX, ADD, SHR, SHL) surviving even as the cast chains around each collapse. **DMA engines** reduce least: VTA’s Store module (13.6%) and Gemmini’s ExecuteController (23.9%) are dominated by large instruction decoders, multi-level address computation, and deep control logic with little bit-manipulation to canonicalize. VTA’s higher overall

Table 4: Formal verification results. Each row is a Z3 bitvector SMT proof establishing equivalence between the lifted MLIR and the bit-level scalar model.

Accelerator	Proof Target	Method	Scope
Gemmini	PE MAC semantics	Z3 bitvector	All 2^{60} inputs
	WS dataflow mux	Z3 SMT	Control specialization
	DMA copy semantics	Z3 + arrays	All addresses/values
VTA	13 datapath targets	Z3 bitvector	4 modules

reduction (41.2% vs. 24.8%) reflects which module dominates each total—TensorGemm for VTA, ExecuteController for Gemmini (998K lines across 80 files for 4 instructions $\times$ 20 ASVs).

Pipeline cost. Extraction and lifting are a one-time, per-accelerator cost rather than a per-compilation overhead: the extracted MLIR is reused by every subsequent compile. On Gemmini at DIM=16, end-to-end wall-clock is 1.3 s for the PE, 6.5 s for the LoadController, and 7.2 s for the StoreController; the ExecuteController takes 5,250 s, decomposing into 48 s flattening, 4,540 s autoGenILA extraction, 619 s lifting, and 43 s TAIDL assembly. Extraction dominates, and its per-PE cost grows super-linearly with array dimension because the symbolic unroll depth scales as $2 \cdot \text{DIM} + 4$ and the dependency cone deepens with it; both extraction and lifting are embarrassingly parallel across PEs, so wall-clock scales down with available cores. What this cost replaces is hand-authoring the equivalent TAIDL model, which takes an expert weeks.

4.3 Correctness Verification

We verify that TensorLift’s lifted semantics are equivalent to the bit-level scalar model they derive from. Since autoGenILA’s extraction is provably bit-equivalent to the RTL by construction [43, 44], equivalence between the lifted MLIR and autoGenILA’s output transitively proves **RTL behavior $\equiv$ TensorLift semantics**. This machine-checked guarantee covers the core compute and data-movement primitives enumerated in Table 4; the remaining lifted operations—pooling, im2col convolution, and the composed `mvin/mvout` sequences—are validated differentially against Spike golden data; we do not claim SMT proofs for them.

Table 4 summarizes the proofs. For Gemmini we verify the PE’s multiply-accumulate core (`sxt(a*b + trunc(c))`) against autoGenILA’s LLVM IR output by Z3 bitvector reasoning over all 2^{60} input combinations—universal equivalence rather than test vectors—and similarly the weight-stationary dataflow mux (control specialization, Pass B4) and DMA copy semantics including strided access. For VTA, 13 targets across all four datapath modules are proven equivalent to the LLVM IR output, covering both compute and control semantics. Lifting also exposes a structural symmetry: VTA’s input and weight index generators produce identical 146-line MLIR after lifting, consistent with the symmetric roles of these buffers in the GEMM engine. Verification cost is independent of array size: the PE proof discharges in ≈ 25 ms, and by systolic homogeneity that single proof certifies all DIM² processing elements, so the cost stays flat as DIM grows.

Guarantees for Stage-3 assembly. The proofs above establish Stage-2 lifting; Stage-3 assembly is not itself SMT-proven, and three structural properties bound its risk. First, HLO templating is a fixed,

accelerator-agnostic map from a closed set of Stage-2 annotations to HLO—`dot_product` to convert `+ dot + add` (the add present only when an accumulator read was recovered), `mul` to multiply with optional `clamp`, add to a reduction—with no per-instruction or per-accelerator logic, so dispatch depends only on the recovered annotation. Second, CISC macro composition (`loop_ws`, `loop_conv_ws`) reuses per-primitive MAC semantics already proven equivalent in Stage 2: the macro body is that primitive wrapped in shape plumbing, iterated over a loop nest whose bound *registers* are identified from the controller RTL (the values being runtime configuration fields). Third, FSM ordering constraints are read off explicit state-comparison logic in the extracted RTL, and every recovered ordering edge corresponds to an instruction that demonstrably writes the guarding control register; the complementary idle-state guard is inferred under a binary-FSM assumption.

As an independent cross-check, all extracted VTA instruction behaviors—GEMM tensor contraction, ALU element-wise operations with five opcodes, Store DMA command generation, and Load burst addressing—align with the operational semantics documented in the VTA specification [30] (§6–9), which the pipeline never reads.

4.4 Specification Completeness

TensorLift extracts TAIDL specifications for every hardware instruction Gemmini’s three controllers decode, and no instruction on either accelerator falls back to opaque semantics. Of Gemmini’s 11 decoded instructions, seven carry tensor semantics (`compute_preloaded`, `compute_accumulated`, `preload` operand staging, `mvin/mvin2/mvin3`, `mvout`) and four are pure configuration writes producing no numeric output (`config_ex/_ld/_st/_norm`); VTA splits 3/1 the same way across its four instructions. Control instructions have no tensor semantics to recover in the first place, so the opaque fallback never applies to them; their bit-level state updates are preserved verbatim. VTA’s adder-tree GEMM stresses the templates hardest: its reduction does not match the single dense dot-product shape of Pass C7, so Stage 3 emits it as a composition of primitives with its value semantics intact.

Comparison with hand-written reference. Compared against the TAIDL MICRO 2025 artifact [19], a hand-written Gemmini specification, TensorLift captures strictly richer semantics in three areas:

- **Multi-bank DMA configuration.** The hand-written reference models each DMA load variant (`mvin_spad`, `mvin_acc`, etc.) with a single `@a.stride` parameter, implicitly assuming all three load engines share one configuration. Gemmini’s LoadController in fact maintains *three independent banks*, each with its own `stride`, `scale`, `shrink`, `block_stride`, and `pixel_repeat` registers (5 parameters $\times$ 3 banks), the active bank selected by the `state_id` field (`rs1[4:3]`) of the encoding. The hand-written TAIDL therefore cannot correctly simulate programs using `mvin` and `mvin2` with different strides simultaneously—a real case in tiled matrix multiplication, where A- and B-tiles load from DRAM with different access patterns. TensorLift sees all three banks automatically because autoGenLLA traces every register independently and the assembler groups them by indexed base name (`strides_0`, `strides_1`, `strides_2`).

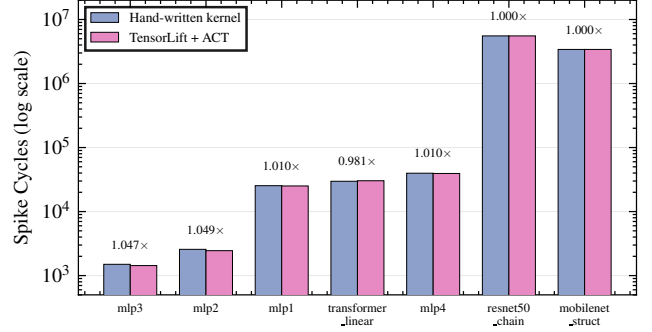


Figure 5: Cycle-level latency of hand-written Gemmini kernels vs. the TensorLift+ACT backend on Spike; labels show speedup.

Table 5: Latency in Spike simulator cycles; speedup = hand-written / TensorLift+ACT.

Benchmark	Hand-written	TensorLift+ACT	Speedup
mlp3	1,501	1,433	1.047×
mlp2	2,571	2,451	1.049×
mlp1	25,364	25,123	1.010×
transformer_linear	29,732	30,302	0.981×
mlp4	39,742	39,362	1.010×
resnet50_chain	5,544,077	5,544,303	1.000×
mobilenet_struct	3,408,414	3,408,581	1.000×
Geometric mean			1.014×

- **Pooling engine semantics.** TensorLift extracts 12 pooling configuration registers from the StoreController (`pool_size`, `pool_stride`, `pool_upad`, `pool_lpad`, `pool_orows`, `pool_ocols`, etc.) and generates `reduce(max)` + `clamp` semantics; the hand-written reference does not model the pooling engine at all.
- **Im2col hardware support.** TensorLift extracts 9 `im2col` output ports from the ExecuteController and generates `reshape` + `dot` semantics, capturing the hardware’s on-the-fly `im2col` transformation during convolution.

The hand-written reference also includes four entries—`softmax`, `GELU`, `layernorm`, and fused `add-with-transpose`—that are *not* hardware ISA primitives but software-composed sequences using the host CPU’s FPU for operations such as `exp()` and `divide()`; no Gemmini RTL module implements them. TensorLift identifies this hardware/software boundary correctly by extracting only instructions with actual RTL decoder entries.

We do not directly compare backends generated from the two specifications because the MICRO 2025 artifact targets a *functional simulator*: it describes how individual instructions move data between scratchpad and accumulator at per-tile (16 $\times$ 16) granularity—what a simulator needs to replay an instruction trace. A backend instead needs per-*kernel* granularity that maps an entire HLO dot or convolution onto the hardware, and the artifact’s per-instruction patterns are too granular for ACT’s instruction selection, which expects CISC-level macro-instructions (`loop_ws` for a whole matrix multiply, `loop_conv_ws` for a whole convolution). TensorLift’s specification supplies that macro-level abstraction (§3.3), enabling the backend evaluated next.

4.5 End-to-End Compiler Backend Generation

To validate that the extracted specification suffices for practical use, we feed it into ACT [21] to generate a Gemmini compiler backend and compare the generated code against hand-written kernels on realistic workloads.

Methodology. We pass TensorLift’s extracted TAIDL specification to ACT, which generates an XLA-compatible backend using equality saturation for instruction selection and constraint programming for scratchpad allocation. Completing the path to an executable binary required engineering contributions to ACT: HLO frontend support for JAX-produced operations (e.g., convolution, reduce_max), enode-level preconditions for correct convolution dispatch, and memory allocator support for multi-layer chains. We reimplement the benchmark suite from gemmini-rocc-tests [39]—the Gemmini team’s official bare-metal test suite—in JAX [3] and compile it through the generated backend, comparing each benchmark’s Spike cycle count against the corresponding hand-written C kernel under identical workload parameters (layer shapes, weight-stationary dataflow, activation functions, scaling, batch sizes). The suite spans MLP, convolutional, and transformer workloads from 2-layer networks (1,501 cycles) to full ResNet-50 (5,544,077 cycles) and MobileNet (3,408,414 cycles).

Results. Table 5 and Figure 5 report the results. Across all seven benchmarks, the automatically generated backend lands within $\pm 2\%$ of the hand-written kernels ($1.014\times$ geometric mean, per-benchmark range $0.981\times$ – $1.049\times$). The small MLP benchmarks (mlp2, mlp3) run slightly faster (1.047 – $1.049\times$) from minor tiling differences; the transformer benchmark is the only slowdown ($0.981\times$), from per-tiling configuration overhead in the generated sequence; and on the large workloads that dominate real deployment—ResNet-50 (49 conv layers) and MobileNet (52 layers)—the ratio is $1.000\times$.

TensorLift combined with ACT therefore produces a functioning Gemmini backend, at parity with hand-written kernels across the suite, without any hand-written specification.

5 Related Work

Formal accelerator interfaces and model extraction. Instruction-Level Abstraction (ILA) [16] established a software-visible specification framework for accelerators and processors; autoGenILA [43, 44] showed how architecture-level models can be derived automatically from RTL, and 3LA [15] how such interfaces support application-level validation. TensorLift builds on this line but differs in target abstraction: rather than stopping at scalar architectural state updates or validation-oriented interfaces, it lifts extracted semantics to tensor-visible instruction behavior suitable for software-stack generation.

Specification and backend automation. TAIDL [20] introduced a domain-specific language for expressing tensor-accelerator ISA semantics, and ACT [21] showed that compiler backends can be generated automatically from such specifications. These efforts automate the downstream path from ISA description to tooling but assume that a tensor-level specification already exists. TensorLift addresses this missing front-end step by recovering TAIDL-like semantics from RTL-derived low-level models.

MLIR and hardware IR semantics. Our work is enabled by MLIR [25] as a progressive abstraction framework and is adjacent to hardware-oriented MLIR efforts such as CIRCT [10]; K-CIRCT [46] and first-class verification dialects for MLIR [11] further highlight the value of giving hardware and compiler IR dialects explicit semantics. TensorLift differs in emphasis: rather than defining semantics for an existing IR or hardware dialect, it uses MLIR as the staging ground for lifting extracted hardware behavior to tensor-level ISA descriptions.

Top-down accelerator programming and generation. TensorLift should be distinguished from top-down frameworks such as HeteroCL [24], Calyx [31], and Exo [17], which help users design, program, or compile for accelerators from higher-level descriptions; TensorLift instead starts from an existing RTL implementation and reconstructs the software-visible tensor interface. It has some affinity with semantic lifting and raising in binary analysis [9], but its goal is not generic decompilation to compiler IR—it is recovery of structured tensor semantics consumable by accelerator-specific tooling.

6 Conclusion

We presented TensorLift, the first end-to-end pipeline that automatically lifts RTL-extracted accelerator semantics to tensor-level ISA specifications, addressing the manual specification bottleneck that has kept automated compiler generation from reaching most tensor accelerator designs. Its 8-pass MLIR pipeline—bit-manipulation canonicalization, MAC pattern detection, control-flow specialization, saturation recovery, loop reconstruction, linalg lifting, and structured metadata emission—connects gate-level hardware behavior to the tensor-level semantics languages like TAIDL require.

Evaluated on Gemmini and VTA, TensorLift processes 156 MLIR files totaling over 1.2 million lines of bit-level IR, reducing them by 24.8% and 41.2% respectively—up to 92.9% on compute-dominated modules—and recovering hardware features absent from hand-written references (multi-bank DMA configuration, pooling engine semantics, im2col support). Core compute and data-movement semantics are proven equivalent to the RTL-extracted model via Z3 SMT, the remainder validated against golden simulator data, and the pipeline generalizes across both architectures unmodified. Fed into ACT, the extracted specification yields a backend at parity with hand-written kernels ($1.014\times$ geometric mean) up to ResNet-50 and MobileNet, with no hand-written specification anywhere in the path. TensorLift thereby closes the first gap in the RTL-to-framework pipeline, the step from hardware design to machine-readable ISA semantics, making the full automated path from RTL through TAIDL and ACT to XLA possible.

References

- [1] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, et al. 2020. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *Ieee Micro* 40, 4 (2020), 10–21.
- [2] Krste Asanovic, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The rocket chip generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016), 6–2.
- [3] James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Yash Katariya, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. 2018. *JAX: composable transformations of Python+NumPy programs*. <http://github.com/google/jax>

- [4] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. {TVM}: An automated {End-to-End} optimizing compiler for deep learning. In *13th USENIX symposium on operating systems design and implementation (OSDI 18)*. 578–594.
- [5] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.
- [6] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).
- [7] NVIDIA Corporation. 2017. NVDLA: Open Source Deep Learning Accelerator. <http://nvidia.org>. Accessed: 2026-04-05.
- [8] Leonardo De Moura and Nikolaj Björner. 2008. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (Budapest, Hungary) (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340.
- [9] Alessandro Di Federico, Mathias Payer, and Giovanni Agosta. 2017. rev.ng: a unified binary analysis framework to recover CFGs and function boundaries. In *Proceedings of the 26th International Conference on Compiler Construction (Austin, TX, USA) (CC 2017)*. Association for Computing Machinery, New York, NY, USA, 131–141. doi:10.1145/3033019.3033028
- [10] Schuyler Eldridge, Prithayan Barua, Aliaksei Chapyzenka, Adam Izraelevitz, Jack Koenig, Chris Lattner, Andrew Lenharth, George Leontiev, Fabian Schuiki, Ram Sunder, et al. 2021. MLIR as hardware compiler infrastructure. In *Workshop on Open-Source EDA Technology (WOSSET)*, Vol. 3.
- [11] Mathieu Fehr, Yuyou Fan, Hugo Pompougnac, John Regehr, and Tobias Grosser. 2025. First-Class Verification Dialects for MLIR. *Proc. ACM Program. Lang.* 9, PLDI, Article 206 (June 2025), 25 pages. doi:10.1145/3729309
- [12] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vignesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, et al. 2021. Gemmini: Enabling systematic deep-learning architecture evaluation via full-stack integration. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 769–774.
- [13] Reza Hojabr, Ali Sedaghati, Amirali Sharifian, Ahmad Khonsari, and Arrvindh Shriraman. 2021. SPAGHETTI: Streaming Accelerators for Highly Sparse GEMM on FPGAs. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 84–96. doi:10.1109/HPCA51647.2021.00017
- [14] Charles Hong, Sahil Bhatia, Alvin Cheung, and Yakun Sophia Shao. 2025. Autocomp: A Powerful and Portable Code Optimizer for Tensor Accelerators. *arXiv preprint arXiv:2505.18574* (2025).
- [15] 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. 2024. Application-level Validation of Accelerator Designs Using a Formal Software/Hardware Interface. *ACM Trans. Des. Autom. Electron. Syst.* 29, 2, Article 35 (Feb. 2024), 25 pages. doi:10.1145/3639051
- [16] 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. doi:10.1145/3282444
- [17] Yuka Ikarashi, Gilbert Louis Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. 2022. Exocompile for productive programming of hardware accelerators. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (San Diego, CA, USA) (PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. doi:10.1145/3519939.3523446
- [18] Intel Corporation. 2023. *Intel Advanced Matrix Extensions (Intel AMX) Technology Brief*. Technology Brief. Intel Corporation. <https://intel.com>
- [19] Devansh Jain, Marco Frigo, Jai Arora, Akash Pardeshi, Zhihao Wang, Krut Patel, and Charith Mendis. 2025. *Artifact of TAIDL: Tensor Accelerator ISA Definition Language with Auto-generation of Scalable Test Oracles*. doi:10.5281/zenodo.1693475
- [20] Devansh Jain, Marco Frigo, Jai Arora, Akash Pardeshi, Zhihao Wang, Krut Patel, and Charith Mendis. 2025. TAIDL: Tensor Accelerator ISA Definition Language with Auto-generation of Scalable Test Oracles. In *Proceedings of the 2025 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 1316–1333. doi:10.1145/3725843.3756075
- [21] Devansh Jain, Akash Pardeshi, Marco Frigo, Krut Patel, Kaustubh Khulbe, Jai Arora, and Charith Mendis. 2025. ACT: Automatically Generating Compiler Backends from Tensor Accelerator ISA Descriptions. doi:10.48550/arXiv.2510.09932
- [22] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.
- [23] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects. *ACM Sigplan Notices* 53, 2 (2018), 461–475.
- [24] 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*. 242–251.
- [25] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shepsman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling compiler infrastructure for domain specific computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 2–14.
- [26] Mingjie Liu, Teodor-Dumitru Ene, Robert Kirby, Chris Cheng, Nathaniel Pinckney, Rongjian Liang, Jonah Alben, Himyanshu Anand, Sanmitra Banerjee, Ismet Bayraktaroglu, et al. 2024. ChipNeMo: Domain-Adapted LLMs for Chip Design. *arXiv preprint arXiv:2311.00176* (2024).
- [27] LLVM Project. 2026. Chapter 5: Partial Lowering to Lower-Level Dialects for Optimization. <https://mlir.llvm.org/docs/Tutorials/Toy/Ch-5/>. MLIR tutorial documentation. Accessed: 2026-04-09.
- [28] LLVM Project. 2026. Linalg Dialect. <https://mlir.llvm.org/docs/Dialects/Linalg/>. MLIR documentation. Accessed: 2026-04-09.
- [29] LLVM Project. 2026. Pattern Rewriting: Generic DAG-to-DAG Rewriting. <https://mlir.llvm.org/docs/PatternRewriter/>. MLIR documentation. Accessed: 2026-04-09.
- [30] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. 2019. A hardware-software blueprint for flexible deep learning specialization. *IEEE Micro* 39, 5 (2019), 8–16.
- [31] Rachit Nigam, Samuel Thomas, Zhijiang 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. doi:10.1145/3445814.3446712
- [32] oneDNN Contributors. [n. d.]. *oneAPI Deep Neural Network Library (oneDNN)*. <https://github.com/uxfoundation/oneDNN>
- [33] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. 2020. Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 58–70.
- [34] Amit Sabne. 2020. XLA : Compiling Machine Learning for Peak Performance.
- [35] Hardik Sharma, Jongse Park, Divya Mahajan, Emmanuel Amaro, Joon Kyung Kim, Chenkai Shao, Asit Mishra, and Hadi Esmaeilzadeh. 2016. From high-level deep neural models to FPGAs. In *2016 49th Annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE, 1–12.
- [36] 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. A 25mm2 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 *International Solid-State Circuits Conference (ISSCC)*.
- [37] The IREE Authors. 2019. *IREE*. <https://github.com/iree-org/iree>
- [38] Jianming Tong, Anirudh Itagi, Prasanth Chatarasi, and Tushar Krishna. 2024. Feather: A reconfigurable accelerator with data reordering support for low-cost on-chip dataflow switching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 198–214.
- [39] UC Berkeley Architecture Research. 2026. Gemmini RoCC Tests. <https://github.com/ucbar/gemmini-rocc-tests>. Bare-metal test suite for the Gemmini accelerator.
- [40] UC Berkeley Architecture Research. 2026. Gemmini Spike Functional Simulator. <https://github.com/ucbar/libgemmini>. Cycle-level functional model for Gemmini.
- [41] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, et al. 2019. Magnet: A modular accelerator generator for neural networks. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.
- [42] Zhongzhi Yu, Mingjie Liu, Michael Zimmer, Yingyan Celine, Yong Liu, and Haoxing Ren. 2025. Spec2rtl-agent: Automated hardware code generation from complex specifications using llm agent systems. In *2025 IEEE International Conference on LLM-Aided Design (ICLAD)*. IEEE, 37–43.
- [43] Yu Zeng, Aarti Gupta, and Sharad Malik. 2022. Automatic Generation of Architecture-Level Models from RTL Designs for Processors and Accelerators. In *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. 460–465. doi:10.23919/DATE54114.2022.9774527
- [44] Yu Zeng, Bo-Yuan Huang, Hongce Zhang, Aarti Gupta, and Sharad Malik. 2021. Generating Architecture-Level Abstractions from RTL Designs for Processors and Accelerators Part I: Determining Architectural State Variables. In *2021 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*. 1–9. doi:10.1109/ICCAD51958.2021.9643584
- [45] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs. In *Proceedings of the International Conference on Computer-Aided Design (San Diego, California) (ICCAD '18)*. Association for Computing Machinery, New York, NY, USA, Article 56, 8 pages. doi:10.1145/3240765.3240801
- [46] Jianhong Zhao, Jinhui Kang, and Yongwang Zhao. 2024. K-CIRCT: A Layered, Composable, and Executable Formal Semantics for CIRCT Hardware IRs. *arXiv:2404.18756 [cs.SE]* <https://arxiv.org/abs/2404.18756>
- [47] Ruizhe Zhong, Xingbo Du, Shixiong Kai, Zhentao Tang, Siyuan Xu, Hui-Ling Zhen, Jianye Hao, Qiang Xu, Mingxuan Yuan, and Junchi Yan. 2024. LLM4EDA: Emerging Progress in Large Language Models for Electronic Design Automation. *arXiv preprint arXiv:2401.12224* (2024).