Appendix: Automatically Generating ML Compiler Backends from Tensor Accelerator ISA Descriptions

DEVANSH JAIN, University of Illinois Urbana-Champaign, USA
AKASH PARDESHI, University of Illinois Urbana-Champaign, USA
MARCO FRIGO, University of Illinois Urbana-Champaign, USA
KAUSTUBH KHULBE, University of Illinois Urbana-Champaign, USA
KRUT PATEL*, NVIDIA, USA
SAATVIK LOCHAN, University of Illinois Urbana-Champaign, USA
JAI ARORA, University of Illinois Urbana-Champaign, USA
CHARITH MENDIS, University of Illinois Urbana-Champaign, USA

Machine learning (ML) compilers play a key role in enabling high-performance implementations of ML workloads. These compilers use existing CPU and GPU backends to generate device-specific code. In recent years, many tensor accelerators (or AI accelerators) have been designed to further accelerate these workloads, with commercial products like AWS Trainium publicly available. However, compared to commodity hardware, a majority of tensor accelerators do not have mature ML compiler backends with robust code generation support. Moreover, tensor accelerator designs are subject to fast iteration cycles, making it difficult to manually develop and maintain ML compiler backends. Therefore, to enable faster integration of novel tensor accelerator designs in ML infrastructure, we need to make the compiler backend construction process more agile.

In this paper, we introduce ACT, a compiler backend generator that automatically generates compiler backends for tensor accelerators, given just the instruction set architecture (ISA) descriptions. These backends are integrated with XLA, a production ML compiler. ACT uses a novel ISA-parameterized compilation algorithm to generate a compiler backend with an equality-saturation-based instruction selection phase and a constraint-programming-based memory allocation phase. We generated compiler backends for 6 accelerator platforms from industry (e.g., AWS Trainium, Intel AMX) and academia (e.g., Gemmini). We showed that these generated backends match or outperform commercial compiler backends and expert-written kernel libraries, while maintaining low compilation overheads. Notably, ACT-generated backend for AWS NKI ISA improved the code generation coverage for AWS Trainium by 2.3x compared with AWS's production compiler, `neuronx-cc`.

ACT is part of a larger open-source ecosystem, built around our ISA description language TAIDL, that automatically generates essential software tools, such as test oracles and compiler backends, from ISA descriptions of tensor accelerators. Our tooling has been adopted by multiple academic and industry teams designing novel tensor accelerators. The ecosystem is available at https://github.com/act-compiler/act.

CCS Concepts: • **Software and its engineering → Compilers**; **Retargetable compilers**.

Additional Key Words and Phrases: ML Compilers, AI Accelerators, Automated Compiler Construction

*Work done while at University of Illinois Urbana-Champaign.

Authors' Contact Information: Devansh Jain, University of Illinois Urbana-Champaign, USA, devansh9@illinois.edu; Akash Pardeshi, University of Illinois Urbana-Champaign, USA, pardesh2@illinois.edu; Marco Frigo, University of Illinois Urbana-Champaign, USA, mfrigo3@illinois.edu; Kaustubh Khulbe, University of Illinois Urbana-Champaign, USA, kkhulbe2@illinois.edu; Krut Patel, NVIDIA, USA, krutp@nvidia.com; Saatvik Lochan, University of Illinois Urbana-Champaign, USA, slochan2@illinois.edu; Jai Arora, University of Illinois Urbana-Champaign, USA, jaia3@illinois.edu; Charith Mendis, University of Illinois Urbana-Champaign, USA, charithm@illinois.edu.

Contents

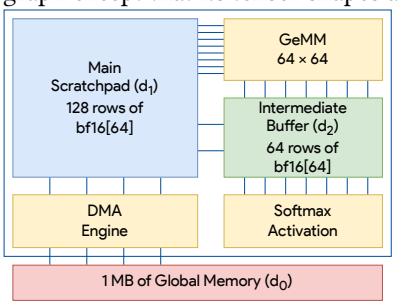
This document is the supplementary appendix to *Automatically Generating ML Compiler Backends from Tensor Accelerator ISA Descriptions*, published in the Proceedings of the ACM on Programming Languages, Volume 10, Issue OOPSLA2, Article 325 (https://doi.org/10.1145/3839457). It is self-contained: definitions, notation and running examples needed to follow the material below are restated here rather than assumed, so the appendix can be read without the paper at hand. Cross-references written as "§*n* of the main paper" or "Fig. *n* of the main paper" point into that article; all other references are internal to this document.

A Detailed Problem Formulation

This appendix formalizes the problem statement of a compiler backend generator that produces sound and complete compiler backends from an accelerator ISA description alone.

We proceed in three steps. First, we formalize the tensor accelerator ISA description (§A.1–§A.3) written in TAIDL, using existing formal models of tensors and tensor operators [2, 11, 12, 17]. The formalism is driven by TAIDL's key observation that tensor accelerators support coarse-grained instructions operating on multi-dimensional data, commonly represented as tensors. Second, we define the equivalence condition (§A.4) between a tiled kernel and an accelerator-specific assembly code. Third, we use that equivalence to define soundness and completeness of a compiler backend, completing the problem formulation (§A.5) of a compiler backend generator that addresses the challenges and goals of §1.2 of the main paper.

Throughout, the object a compiler backend consumes is a *tiled kernel*: the single-tile version of a fused sub-graph produced by the XLA middle-end, syntactically an ordinary tensor computation graph except that its tensor shapes are those of one tile rather than of the whole operand.



(a)

Instruction	Description
load_rm	$y = \text{bitcvt}(\text{reshape}(x_1))$
load_cm	$y = \text{transpose}(\text{bitcvt}(\text{reshape}(x_1)))$
store_rm	$y = \text{reshape}(\text{bitcvt}(x_1))$
store_cm	$y = \text{transpose}(\text{reshape}(\text{bitcvt}(x_1)))$
mov	$y = \text{copy}(x_1)$
gemm	$y = \text{dot}(x_1, x_2)$
softmax	$y = \text{div}(\text{exp}(x_1), \text{broadcast}(\text{reduce}(\text{exp}(x_1))))$

(b)

Fig. 1. Running example. (a) High-level accelerator design for a hypothetical accelerator H_{QKV}. (b) Brief description of its instruction set $\Theta^{H_{QKV}}$. Tensor variables are colored based on their storage unit (d_0, d_1, d_2).

A.1 Tensor Accelerator ISA Description ISA^H

An ISA description comprises (1) the software-managed storage units, collectively termed the *data model*, such as scratchpads; and (2) the semantics of instructions that perform computations, such as data movement and compute instructions, on those storage units.

Running example. Fig. 1 reproduces the running example of §3.2 of the main paper (Fig. 5 of the main paper), so that this appendix can be read on its own. H_{QKV} is a hypothetical accelerator with two on-chip storage units — a 16 KB scratchpad and an 8 KB intermediate buffer — and access to global memory through a DMA engine. Its two compute units are a general matrix multiply (gemm) unit and a softmax activation unit. It supports five data movement instructions and two compute instructions, described in Fig. 1 (b).

A.2 Data Model D^H

The software-managed storage units, like the scratchpads on an accelerator, are abstractly represented as *tensor buffers*. A tensor buffer is a tensor with a subset of its dimensions representing *access dimensions* and the remaining dimensions representing the granularity of data access. Accelerator instructions modify sub-tensors of these tensor buffers, called *data slices*. We assume that an accelerator has a byte-addressable 1-D global memory (HBM) accessible by the host processor, denoted d_0. We now define these terms using tensor notation. Definitions are numbered in one series throughout this appendix; the boxed ones fix the *objects* an ISA description denotes, and the unboxed ones that follow in §A.4 onwards define functions and relations over those objects.

Def A.1: Tensor Buffer d

A **tensor buffer** d is a tensor of type $\mathbb{E}[S_0, S_1]$ in which the first $\dim(S_0)$ dimensions are *access dimensions* (S_0) and the remaining dimensions are the *granularity of data access* ($\mathbb{E}[S_1]$). Equivalently, d is a multi-dimensional storage unit of shape S_0 whose elements are tensors of uniform type $\mathbb{E}[S_1]$; we write its type as $(S_0, \mathbb{E}[S_1])$ when the access structure is what matters.

Def A.2: Data Slice $d[I_s{:}I_e]$

A **data slice** $d[I_s{:}I_e]$ of tensor buffer d is a tensor of type $\mathbb{E}[(I_e - I_s), S_1]$ where $I_s, I_e \in \mathbb{N}_0^{|S_0|}$ and $I_s < I_e \leq S_0$. I_s and I_e are the start (inclusive) and end (exclusive) addresses of the data slice under zero-based multi-dimensional addressing.

Def A.3: Memory State M

A **memory state** $M = (d_0, d_1, \dots)$ is a tuple of tensors, one for each tensor buffer.

Tensor buffer representations for the software-managed storage units of H_{QKV} are annotated in Fig. 1. The 16 KB scratchpad (d_1) consists of 128 rows of 64-length bf16 vectors, and instructions read or modify it at row granularity. It is therefore represented as $d_1 = ([128], \mathrm{bf16}[64])$.

A.3 Instruction Set Θ^H

The instruction set Θ^H is a set of *abstract instructions* that modify the tensor buffers in D^H and are parameterized by instruction attributes. This is analogous to x86 instructions such as mov and add, which modify registers or memory and are parameterized by immediate values, register names, and memory addresses. A *concrete instruction* is an instantiation of an abstract instruction with specific values for those attributes, and is a *deterministic* modification of the tensor buffers. Fig. 1 (b) briefly describes the instruction set $\Theta^{H_{QKV}}$ of H_{QKV}.

Consider the abstract instruction load_rm in $\Theta^{H_{QKV}}$, parameterized by three attributes: $\mathsf{n} \in [1, 128]$, $\mathsf{addr}_{\mathsf{in}} \in [0, 2^{20})$, and $\mathsf{addr}_{\mathsf{out}} \in [0, 128)$. It is a data movement instruction that loads n rows of data (n × 128 bytes) from HBM (d_0) starting at address $\mathsf{addr}_{\mathsf{in}}$, into the main scratchpad (d_1) starting at row $\mathsf{addr}_{\mathsf{out}}$. The concrete instruction $\mathsf{load\_rm}(\mathsf{n} = 4, \mathsf{addr}_{\mathsf{in}} = 0, \mathsf{addr}_{\mathsf{out}} = 2)$ reads 512 bytes from data slice $d_0[0{:}512]$ of tensor-type $\mathrm{u8}[512]$ into data slice $d_1[2{:}6]$ of tensor-type $\mathrm{bf16}[4, 64]$. Each bf16 value (2 bytes) is a bitwise cast of 2 contiguous u8 values. This transformation

is a concrete tensor computation from u8[512] to bf16[4, 64] built from the tensor operators of Table 2.

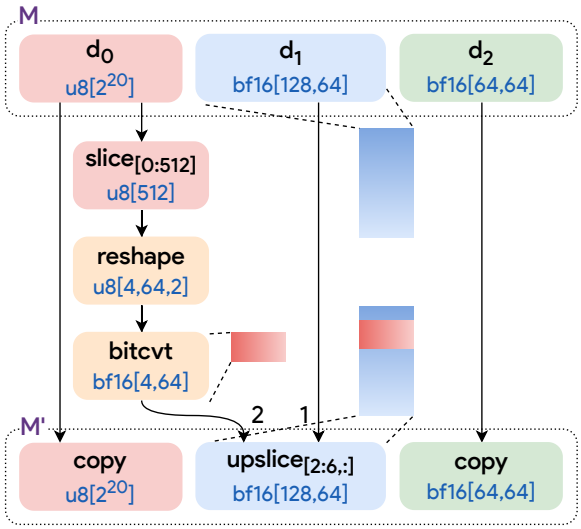


Fig. 2. Execution of the concrete instruction load_rm(n = 4, $\text{addr}_{\text{in}} = 0$, $\text{addr}_{\text{out}} = 2$) as a concrete tensor computation graph.

More generally, the semantics of load_rm can be written as an abstract tensor computation $y = \text{bitcvt}(\text{reshape}(x_1))$ whose input and output tensors are data slices of d_0 and d_1 respectively. The addresses of these data slices are a function of the attributes addr_{in} and addr_{out}, while the number of rows loaded is a function of the attribute n. This observation is what ACT builds on: we *extend* TAIDL descriptions by classifying the instruction attributes into two sets, $\alpha = \{\text{n}\}$ and $\beta = \{\text{addr}_{\text{in}}, \text{addr}_{\text{out}}\}$, termed *computational* and *addressing* attributes respectively. The computational attributes determine the core tensor computation; every remaining attribute is an addressing attribute. This split is not present in TAIDL itself, and it is what makes the two phases of ACT's algorithm separable — Phase 1 reasons about α and Phase 2 about β.

Generalized representation for instruction set semantics. An abstract instruction is parameterized by two sets of integer attributes α and β. Its semantics are represented as (1) an abstract tensor computation concretized by the set of *computational attributes* $\boldsymbol{\alpha}$, where (2) the data slices for the input and output tensors of that computation lie on the tensor buffers, and their addresses are parameterized by the set of *addressing attributes* $\boldsymbol{\beta}$.

Kernel programming languages like Exo [7] model custom hardware instructions under similar assumptions, albeit defining semantics with scalar operators and Python-like syntax. Intuitively, a scalar pseudo-code description with FOR and IF statements, as used by the Intel Intrinsics Guide [8], can be represented with element-wise tensor operators — scalars being 0-D tensors — together with the while[1] and select[2] operators.
We now formalize this generalized representation using the notation defined above.

[1] https://openxla.org/xla/operation_semantics#while
[2] https://openxla.org/xla/operation_semantics#select

Def A.4: Abstract Instruction θ

An **abstract instruction** $\theta \in \Theta^H$, which reads n_θ tensors located in buffers $d_\theta^{(i)}|_{i=1}^{n_\theta}$ and modifies a tensor located in buffer $d_\theta^{(0)}$ (by convention, superscript (0) denotes the output and $(1), \ldots, (n_\theta)$ the inputs), is represented using an abstract tensor computation g_θ concretized by the set of computational attributes α, together with $(n_\theta + 1)$ *data slice address maps* $h_\theta^{(i)}|_{i=0}^{n_\theta}$ over the set of addressing attributes β, and a *validity constraint* e_θ over the attributes α and β.

Def A.5: Concrete Instruction $\theta_{\alpha,\beta}$

A **concrete instruction** $\theta_{\alpha,\beta}$ represents a valid modification if $e_\theta(\alpha, \beta)$ is true. It reads data slices $x_i = d_\theta^{(i)}[I_s^{(i)}{:}I_e^{(i)}]|_{i=1}^{n_\theta}$ and modifies data slice $y = d_\theta^{(0)}[I_s^{(0)}{:}I_e^{(0)}]$ with the corresponding concrete tensor computation $y = f(x_i|_{i=1}^{n_\theta})$, where $f = g_\theta(\alpha)$ and $\forall i \in [0, n_\theta]$ $(I_s^{(i)}, I_e^{(i)}) = h_\theta^{(i)}(\beta)$.

Def A.6: Execution of a Concrete Instruction $\mathcal{E}(\theta_{\alpha,\beta})$

Execution $\mathcal{E}(\theta_{\alpha,\beta})$ of a concrete instruction $\theta_{\alpha,\beta}$ is a concrete tensor computation over a tuple of tensors that updates the memory state $M = (d_0, d_1, \ldots)$ to $M' = (d_0', d_1', \ldots)$, where

$$d_i' = \begin{cases} \mathsf{upslice}_{[h_\theta^{(0)}(\beta)]}\left(d_\theta^{(0)},\ g_\theta(\alpha)\left(\mathsf{slice}_{[h_\theta^{(i)}(\beta)]}(d_\theta^{(i)})\Big|_{i=1}^{n_\theta}\right)\right) & \text{for } d_i = d_\theta^{(0)} \\ \mathsf{copy}(d_i) & \text{for } d_i \neq d_\theta^{(0)} \end{cases}$$

Here upslice, slice, and copy are the tensor operators of Table 2, visualized in Appendix D. Intuitively, execution slices each input operand out of its buffer at the addressed range, applies the concretized computation $g_\theta(\alpha)$ to produce the output sub-tensor, and upslices that back into the output buffer $d_\theta^{(0)}$ at its addressed range; every other buffer is copied unchanged.

Revisiting the concrete instruction $\mathsf{load\_rm}(\mathsf{n} = 4, \mathsf{addr_{in}} = 0, \mathsf{addr_{out}} = 2)$, which loads 512 bytes from $d_0[0{:}512]$ to $d_1[2{:}6]$: Fig. 2 visualizes its execution $\mathcal{E}$, where the orange nodes represent $g_{\mathsf{load\_rm}}(\mathsf{n} = 4)$ with input and output tensor-types $\mathsf{u8}[512]$ and $\mathsf{bf16}[4, 64]$ respectively. The data slice addresses for the input and output tensors, that is the operator attributes of slice and upslice, are $h^{(1)}_{\mathsf{load\_rm}}(\mathsf{addr_{in}}{=}0, \mathsf{addr_{out}}{=}2) = (0, 512)$ and $h^{(0)}_{\mathsf{load\_rm}}(\mathsf{addr_{in}}{=}0, \mathsf{addr_{out}}{=}2) = (2, 6)$ respectively.

A.4 Semantic Equivalence of a Tiled Kernel and Compiled Assembly Code

An assembly code ASM^H for a tensor accelerator H consists of a stream of concrete instructions, defined in the ISA description ISA^H, together with a constant tensor. This is analogous to x86 assembly code with its code and data sections.

Semantic equivalence of a compiled assembly code and a tiled kernel G is defined over the output produced by sequentially executing the concrete instructions, compared against the golden output $G(X)$ computed by G for given input tensors X. Since the multi-dimensional input and output tensors of G are stored in HBM, which is a 1-D byte-addressable memory, we first define a byte encoding for tensors.

*Def A.7 (**bflat**).* The *byte-flattening* operator bflat converts a tensor t of tensor-type $\mathbb{E}[S]$ into a 1-D tensor $\mathsf{bflat}(t)$ of type $\mathsf{u8}[\mathsf{mem}(t)]$, where $\mathsf{mem}(t) = \mathsf{mem}(\mathbb{E}) \times \prod S$ and $\mathsf{mem}(\mathbb{E})$ is the memory size in bytes of the scalar basetype $\mathbb{E}$.

bflat names what §5.1 of the main paper calls the 1-D byte representation of a tensor: for a bf16 tensor it is a bitcvt of each bf16 into two u8 values followed by a reshape.

Because equivalence is stated modulo rewrites, we also need the underlying relation on tensor computation graphs.

Def A.8 ($\equiv_{\mathcal{R}}$). Tensor computation graphs G_1 and G_2 are **semantically equivalent modulo** $\mathcal{R}$, written $G_1 \equiv_{\mathcal{R}} G_2$, if G_1 can be transformed into G_2 by applying rewrite rules in $\mathcal{R}$. We take $\mathcal{R}$ to be the XLA compiler's curated set of 175+ tensor rewrites, treated as the foundational axioms of the XLA-HLO IR. Following prior work [11, 16, 18, 19], we assume the axioms in $\mathcal{R}$ are closed under inversion and composition, so that $\equiv_{\mathcal{R}}$ is reflexive, symmetric and transitive.

The closure assumption in Def. A.8 is what licenses chaining equivalences, which the proofs of Appendix C rely on.

Intuitively, semantic equivalence is a sequence of two events: (1) initialization of HBM with the byte-flattened input tensors and the compile-time constant tensor of the assembly code, and (2) sequential execution of the concrete instructions of the assembly code according to the ISA description. The compiled assembly code is semantically equivalent to the input tiled kernel G if, for all input tensors X, HBM ends up holding the byte-flattened input tensors X followed by the output tensor $G(X)$. We formalize this as a transformation Λ_{LHS} of ASM_G^H and the expected HBM state as a transformation Λ_{RHS} of G.

Fix $\mathsf{mem}_{in} = \mathsf{mem}(X)$ and $\mathsf{mem}_{out} = \mathsf{mem}(X) + \mathsf{mem}(G(X))$. The initial HBM layout places the byte-flattened inputs at offset 0, reserves the region $[\mathsf{mem}_{in}, \mathsf{mem}_{out})$ for the output, and places the constant tensor at offset mem_{out}, so the constant never overlaps the region that equivalence is checked over.

Def A.9 (Λ_{LHS}). Given an assembly code $\mathsf{ASM}_G^H = ((\theta_i(\alpha_i, \beta_i)|_{i=1}^N), \mathsf{const})$, $\Lambda_{LHS}(\mathsf{ASM}_G^H)$ is a concrete tensor computation over a memory state $M = (d_0, d_1, \dots)$ and input tensors X. The memory state after initialization of HBM is

$$M_0 = \big(\mathsf{upslice}_{[\mathsf{mem}_{out}:\mathsf{mem}_{out}+\mathsf{mem}(\mathsf{const})]}(\mathsf{upslice}_{[0:\mathsf{mem}_{in}]}(d_0, \mathsf{bflat}(X)),\ \mathsf{bflat}(\mathsf{const})),\ d_1, \dots\big).$$

The final memory state is

$$M_N = \mathcal{E}(\theta_N(\alpha_N, \beta_N)) \circ \cdots \circ \mathcal{E}(\theta_1(\alpha_1, \beta_1))(M_0),$$

of which only the first mem_{out} bytes of HBM, written $M_N[0]$, are relevant for equivalence. Finally,

$$\Lambda_{LHS}(\mathsf{ASM}_G^H)(M, X) = \mathsf{slice}_{[0:\mathsf{mem}_{out}]}(M_N[0]).$$

Def A.10 (Λ_{RHS}). Given an input tiled kernel G,

$$\Lambda_{RHS}(G)(M, X) = \mathsf{concat}_1(\mathsf{bflat}(X), \mathsf{bflat}(G(X))).$$

Both Λ_{LHS} and Λ_{RHS} map a memory state and input tensors to a single $\mathsf{u8}[\mathsf{mem}_{out}]$ tensor, so they can be compared directly. We now define semantic equivalence using these bisimulation transformations.

Def A.11 ($\equiv_{\mathcal{R}}^H$). Given the foundational axioms $\mathcal{R}$ of the IR and the ISA description ISA^H, the semantic equivalence $\equiv_{\mathcal{R}}^H$ is defined as

$$\mathsf{ASM}_G^H \equiv_{\mathcal{R}}^H G \iff \forall M, \forall X,\ \Lambda_{LHS}(\mathsf{ASM}_G^H)(M, X) \equiv_{\mathcal{R}} \Lambda_{RHS}(G)(M, X),$$

where $\equiv_{\mathcal{R}}$ is the relation of Def. A.8. Λ_{LHS} and Λ_{RHS} are visualized in Fig. 3.

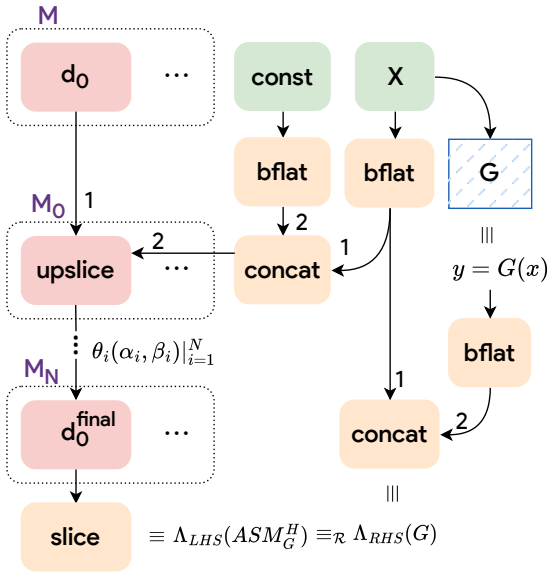


Fig. 3. Visualization of $\equiv_{\mathcal{R}}^H$. M and X are the inputs to the tensor computations. The leaf nodes represent the LHS and RHS of the semantic equivalence.

Since $\equiv_{\mathcal{R}}$ is an equivalence relation (Def. A.8) and Def. A.11 quantifies over all M and X, $\equiv_{\mathcal{R}}^H$ inherits reflexivity, symmetry and transitivity.

A.5 Compiler Backend Generator

We can now define the soundness and completeness of a compiler backend in terms of the formalism developed above. A compiler backend Compiler^H consumes a tiled kernel G and may (i) emit an assembly code, (ii) return FAIL, denoting compilation failure, or (iii) not terminate, denoted $\perp$.

*Def A.12 (**Soundness**).* A compiler backend Compiler^H is **sound** iff

$$\forall G,\ \text{Compiler}^H(G) \notin \{\text{FAIL}, \perp\} \implies \text{Compiler}^H(G) \equiv_{\mathcal{R}}^H G$$

*Def A.13 (**Completeness**).* A compiler backend Compiler^H is **complete** iff

$$\forall G,\ \exists \text{ASM}_G^H \equiv_{\mathcal{R}}^H G \implies \text{Compiler}^H(G) \notin \{\text{FAIL}, \perp\}$$

Discussion: semi-decidability versus totality. Soundness and completeness as defined above do not require the compiler backend to be a total function: it may fail to terminate on tiled kernels that have no semantically equivalent assembly code. This is a deliberate design choice, because requiring all three properties at once is impossible.

Theorem A.1. *There exists no compiler backend generator* CompGen *such that for every tensor accelerator ISA description* ISA^H *in TAIDL, the generated backend* $\text{CompGen}(\text{ISA}^H)$ *is sound, complete, and total.*

Proof. Assume for contradiction that such a generator CompGen exists. Then for every ISA description ISA^H, the backend $\text{Compiler}^H = \text{CompGen}(\text{ISA}^H)$ is sound (Def. A.12), complete (Def. A.13), and total, meaning it always terminates and so never returns $\perp$.

Let M be an arbitrary Turing machine and w an arbitrary input. Construct an ISA description $\mathsf{ISA}^{H(M,w)}$ with one abstract instruction θ parameterized by one computational attribute $\alpha = k \in \mathbb{N}$. Define each concrete instantiation $\theta(k)$ by bounded simulation:

(1) Simulate M on w for exactly k steps.
(2) If M halts within k steps, $\theta(k)$ computes the identity on a unit tensor.
(3) Otherwise, $\theta(k)$ computes the constant-zero function.

Each $\theta(k)$ has finite, computable semantics, and this encoding is expressible in TAIDL [10].

Let G be the tiled kernel computing the identity on a unit tensor.

If M halts on w, there exists k^* such that the one-instruction assembly $\mathsf{ASM} = (\theta(k^*), \emptyset)$ satisfies $\mathsf{ASM} \equiv^H_{\mathcal{R}} G$. By completeness, $\mathsf{Compiler}^H(G) \notin \{\mathsf{FAIL}, \bot\}$, so it compiles successfully.

If M does not halt on w, then for all k the instruction $\theta(k)$ computes the constant-zero function, which is not equivalent under $\mathcal{R}$ to the identity computed by G; hence no assembly built from these instructions is equivalent to G. By soundness, $\mathsf{Compiler}^H(G)$ cannot return a successful assembly, and by totality it cannot return $\bot$; therefore it returns FAIL.

Hence we can decide whether M halts on w by running $\mathsf{Compiler}^H(G)$ and checking whether the result is a successful compilation or FAIL. This decides the halting problem, a contradiction. Therefore no sound, complete, and total generator exists. □

In practice, semi-decidability is not a concern, since a timeout can always be imposed on the compiler backend and treated as a compilation failure. We therefore focus on soundness and completeness without requiring totality. §8.7 of the main paper reports that our generated compiler backends have reasonable compilation times and that we encountered no non-termination in practice.

B Detailed Discussion of the Solution

This appendix expands on the six modules of ACT's compilation algorithm in the places where the main paper had to be brief. It is organized to mirror §5 of the main paper and §6 of the main paper: one subsection per module, in the order the modules run. We begin with an orientation that is easy to lose sight of in the module-by-module presentation, namely what kind of artifact ACT actually is.

B.1 Analogy to Classical Compiler Component Generators

ACT is a *generator*, not a compiler, and the distinction is the source of most of its design. The closest familiar analogues are Flex [5] and Bison [15], which are also generators: each takes a declarative description of an artifact, instantiates a fixed parameterized algorithm against that description, and emits a working component. Table 1 lays the three side by side.

Reading the table row by row makes the correspondence concrete. The *core parameterized algorithm* is the piece of theory that is fixed once and for all and never regenerated: NFA-to-DFA construction for Flex, LALR(1) table generation for Bison, and ACT's ISA-parameterized compilation algorithm. The *generic skeleton* is the code that is identical across every generated instance — a table-driven scanning loop, a shift-reduce parser loop, and in our case a compiler backend whose accelerator-specific decisions are left as pure virtual functions. The *specialized templates* are what the generator actually writes: transition tables, ACTION/GOTO tables, and for ACT the rewrite rules and the concrete implementations that override those virtual functions. This is why ACT needs no accelerator-specific algorithmic work: as with Flex and Bison, all the accelerator-specific content lives in the specialized templates, and the algorithm above them never changes.

	Flex [5]	Bison [15]	ACT (Our work)
User-provided input	Regular expressions	Context-free grammar	ISA description
Generated output	Lexer/Scanner	LALR(1) Parser	Compiler backend
Core Parameterized Algorithm (Theory)	NFA-to-DFA construction [1]	LALR(1) table generation [1]	ISA-parameterized compilation algorithm
Generic Skeleton	Generic table-driven scanning loop	Generic shift-reduce parser loop	Generic backend with pure virtual functions
Specialized Templates	DFA transition table with accept states	ACTION/GOTO tables with lookahead sets	Rewrites & overridden virtual functions
Specialization Approach	GNU M4 macro processor	GNU M4 macro processor	Python-based regex processor

Table 1. Template-driven compiler component generators based on a parameterized algorithm. The core parameterized algorithm and the generic skeleton are fixed once, when the generator itself is built; the specialized templates are what the generator writes anew for each user-provided input, using the specialization approach of the last row.

B.2 Module 1: E-Graph Initializer

The main paper describes Module 1 as lowering the tiled kernel into an initial e-graph, without detailing how XLA-HLO's tiled memory formats are represented. We expand on that here, since tiled layouts are what make ACT-generated backends drop-in compatible with the XLA compiler.

The XLA compiler often stores tensors in a tiled memory format,[3] written $\mathbb{E}[S_t]\{\mathsf{T}(S_T)\}$, in which a tensor of type $\mathbb{E}[S_t]$ is broken into tiles of shape S_T that are each stored contiguously. Fig. 4 (a) shows the XLA-HLO IR for a tiled kernel commonly observed in oneDNN examples: a matrix multiplication whose input tensor B and output tensor both carry tiled layouts. ACT models such a layout as a composition of reshape and transpose operators together with the byte-flattening operator bflat (Def. A.7), so that no new operator or IR extension is required. Fig. 4 (b) shows this modeling for the layout i32[32,32]{T(16,16)}: the $[32, 32]$ tensor is split into $[2, 16, 2, 16]$, the tile axes are rearranged into contiguous order by a transpose, and bflat then yields the flat byte sequence actually held in memory. Because the layout is expressed in the same operator vocabulary as the rest of the graph, equality saturation can rewrite across it: ACT-AMX applied 22 IR-to-IR rewrites over 6 iterations to compile this kernel into the assembly of Fig. 4 (c).

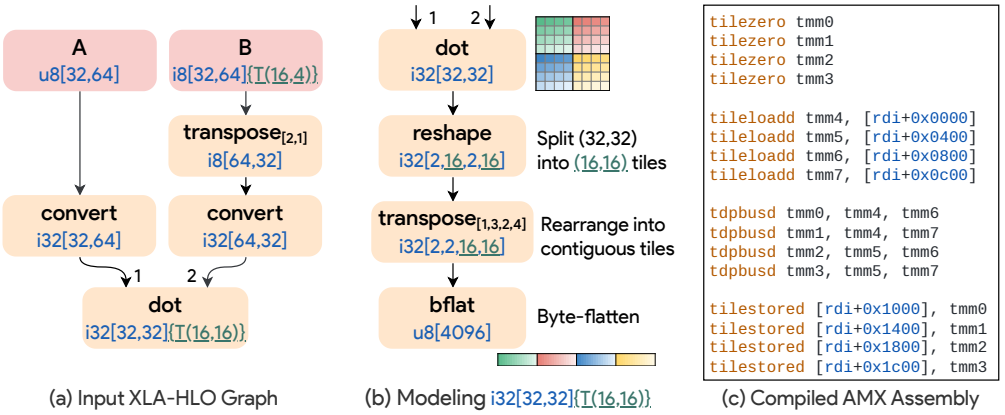


Fig. 4. Modeling a tiled memory layout in Module 1. (a) The input XLA-HLO IR, with tiled layouts on the input tensor B and on the output tensor. (b) The layout i32[32,32]{T(16,16)} as modeled by ACT, using reshape and transpose to gather each tile into contiguous order and bflat to flatten it to bytes. (c) The final assembly code generated by ACT-AMX.

B.3 Module 2: Rewrite Applier

Module 2 supplies the rewrites used during the exploration stage. Two implementation details are worth recording beyond the paper's description.

Accelerator-specific rewrite filtering. The XLA compiler's 175+ IR-to-IR rewrites are not specific to any accelerator, and applying all of them wastes exploration budget on operators the target cannot express. We therefore filter the rewrite set against the target ISA before saturation begins. A rewrite is retained only if it can eventually contribute to an IR-to-ISA rewrite: starting from the operators that appear in some abstract computation g_θ, we add any rewrite that produces such an operator, then add the operators that rewrite consumes, and iterate. For example, if operator C appears in some g_θ, then a rewrite producing C is added to the set; if that set now contains a rewrite $B \longrightarrow C$, operator B is added in turn, and the process repeats until a fixed point. Iterating rather than taking a single pass is what makes the filter handle transitive rewrite chains. The result is that the operator set potentially supported by an accelerator is pruned at backend-generation time rather than at compile time.

[3] https://openxla.org/xla/tiled_layout

Identity instructions. Beyond their role in completeness, the identity instructions slice^H and concat^H are what let ACT tile an inner loop and pad tensor variables without any dedicated machinery: both are expressible as no-ops on the memory state, so the e-graph can introduce them freely wherever a shape needs adjusting. Fig. 5 shows a small pii graph generated by ACT-Gemmini in which exactly this happens: two $[32, 16]$ operands are each split by a pair of slice^H nodes into $[16, 16]$ halves, fed through two matmul instructions, and the results reassembled by a concat^H. Appendix C proves that the availability of these two instructions is what makes Phase 2's reconstruction of a pii graph from an assembly code possible, and with it the completeness of the algorithm as a whole.

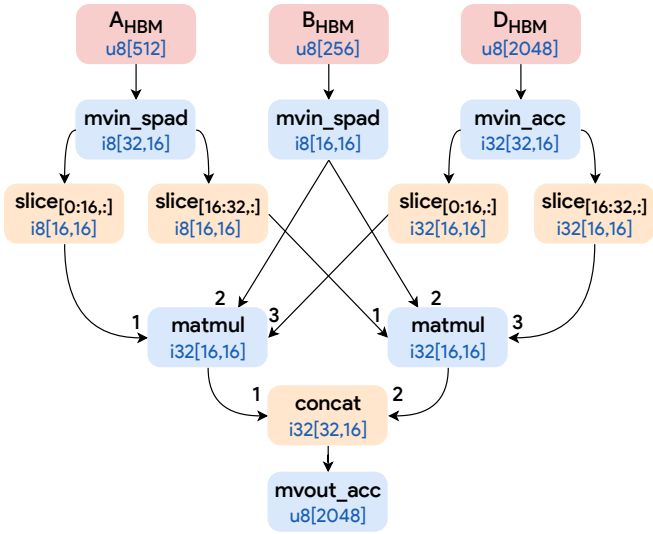


Fig. 5. A pii graph generated by ACT-Gemmini that uses the identity instructions slice^H and concat^H to split an operand across two matmul instructions and reassemble the results.

B.4 Module 3: Graph Extractor

Module 3 extracts pii graphs from the saturated e-graph by a top-down traversal. We expand here on its termination condition, which the paper's pseudocode states compactly.

The traversal terminates at an e-class when it is expecting a tensor in d_0 (HBM) and that e-class either is marked as constant, or contains an e-node representing a flattened input tensor variable. In the second case the flattened input tensor is added directly to the pii graph, since it is already available in HBM and needs no instruction to produce it. In the first case the tensor expression that computes the constant value is added to the pii graph instead, so that the constant is materialized by the accelerator rather than fetched from the host. Constant e-classes are detected by a bottom-up traversal, with the base case appearing in the extraction pseudocode of §5.3 of the main paper. The two cases are mutually exclusive and can never both apply to the same e-class, which is what makes the termination condition well defined.

B.5 Module 4: Topological Ordering Generator

Module 4 orders the pii nodes before memory allocation. The paper states that we extend Sethi-Ullman numbering to multiple tensor buffers; we give the extension in full here, together with a counterexample showing why the heuristic cannot be relied on alone.

Extending Sethi-Ullman numbering. Classic Sethi-Ullman numbering [1] labels each node of an expression tree with the minimum number of registers needed to evaluate it without spilling: a leaf needs one register, and a binary node with child labels ℓ_1, ℓ_2 needs $\max(\ell_1, \ell_2)$ registers if $\ell_1 \neq \ell_2$ and $\ell_1{+}1$ otherwise, evaluating the costlier child first. Two things differ in our setting: there are several tensor buffers rather than one register file, and the values occupying them have widely varying sizes. We therefore track occupancy in bytes, per buffer. For a tensor buffer d and a node y with operands x_i, we define

$$\mathsf{esun}_d(y) = \max\big(\mathsf{mem}_d(y),\ \min_i \tau_d(x_i)\big), \qquad \tau_d(x_i) = \mathsf{esun}_d(x_i) + \sum_{j \neq i} \mathsf{mem}_d(x_j),$$

where $\mathsf{mem}_d(x_j)$ is the memory size in bytes of the tensor variable represented by node x_j, and is 0 whenever that tensor does not reside on buffer d. For a leaf node y, an input or constant tensor, $\mathsf{esun}_d(y) = \mathsf{mem}_d(y)$. Intuitively $\tau_d(x_i)$ is the peak occupancy of buffer d when operand x_i is evaluated last, after every sibling x_j has already been materialized; the $\min_i$ picks the operand order that minimizes this peak. We compute esun_d for every tensor buffer in D^H except d_0 (HBM), and select operands in decreasing order of $\max_d\big(\tau_d(x_i)/\mathsf{mem}(d)\big)$, where $\mathsf{mem}(d)$ is the total capacity of buffer d. Normalizing by $\mathsf{mem}(d)$ is what makes pressure on buffers of very different sizes comparable.

Why the heuristic is not sufficient. We chose Sethi-Ullman numbering as the base algorithm because its objective matches ours: to improve the likelihood that the CSP of Module 5 is satisfiable we want the interference graph to be as small as possible, which is analogous to minimizing register pressure. But while Sethi-Ullman numbering is provably optimal for a single register file, it is not optimal for more than one, and Fig. 6 is a counterexample. The graph has two branches that meet at a common sink, and each branch produces values in *both* register files, drawn blue and red — that crossover is what defeats a depth-first order. Any depth-first topological ordering evaluates one subtree fully before starting the other, and is therefore forced to hold three live values in one file, yielding $(3, 2)$ or $(2, 3)$. The ordering A–I instead interleaves the two subtrees and keeps at most two live values in each file, using $(2, 2)$. No depth-first traversal produces this ordering.

This is a genuine limitation rather than an artifact of our extension, and it is precisely why Phase 2 does not rely on the heuristic ordering for completeness but falls back to an exhaustive search over topological orderings (§6.4 of the main paper). In practice the heuristic is nonetheless effective: it guides the search toward low-pressure schedules, and we rarely observe the fallback being needed.

B.6 Module 5: Constraint Satisfaction Problem Generator

Module 5 encodes memory allocation as a constraint satisfaction problem and discharges it with the Google OR-Tools CP-SAT solver. One practical consequence deserves recording. CP-SAT is multi-threaded, and its search is therefore non-deterministic: the same tiled kernel can compile to different — though equally valid — assembly across runs. Compiler engineers usually expect a backend to be deterministic, so ACT exposes a boolean flag that forces it. When the flag is set, we invoke CP-SAT with `NumSearchWorkers = 1`, which serializes the search and makes the solution reproducible, at some cost in solving time (see the OR-Tools discussion at https://groups.google.com/g/or-tools-discuss/c/ZGj5vhhZX48).

B.7 Module 6: Code Emitter

Module 6 is the simplest of the six and needs little expansion: it walks the pii nodes in the order fixed by Module 4 and emits each one with the addressing attributes solved by Module 5. The single point of interest is that identity instructions are *discarded* at this step rather than emitted.

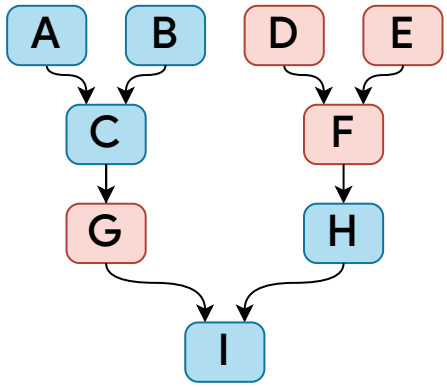


Fig. 6. A counterexample with two register files (blue, red) showing that depth-first topological orderings are not optimal for more than one register file. Every depth-first ordering uses $(3, 2)$ or $(2, 3)$ registers, whereas the interleaved ordering A–I uses only $(2, 2)$.

This is sound because by the time Module 6 runs, whatever selection or reassembly an identity instruction expressed has already been folded into the data slice addresses that Module 5 solved for its neighbours: the value each consumer reads is fixed by its own addressing attributes, so dropping the intervening slice^{H} or concat^{H} changes nothing observable. It is also what keeps their use free: Modules 2 and 3 may introduce as many of them as the search finds convenient without any of them reaching the generated assembly.

C Theoretical Guarantees of ACT's ISA-parameterized Compilation Algorithm

The claims and proofs in this appendix rest on the formalization of Appendix A and are independent of any implementation detail. Their purpose is to establish a rigorous foundation for the soundness and completeness of ACT's core algorithm for automatically generating compiler backends from tensor accelerator ISA descriptions.

C.1 Notation and Standing Assumptions

The proofs are stated in the vocabulary of the compilation algorithm, which we restate here so this appendix can be read on its own.

Phase 1 is built on *equality saturation*. The tiled kernel is loaded into an *e-graph*: a data structure whose nodes (*e-nodes*) are operators and whose *e-classes* are sets of e-nodes known to compute the same value. Applying a rewrite does not replace anything — it adds the rewritten form to the same e-class, so one e-graph represents every graph reachable by the rewrites applied so far. ACT alternates an *exploration* stage, which applies rewrites, with an *extraction* stage, which reads candidate pii graphs back out. Three kinds of rewrite are used (§B.3): the XLA-HLO IR-to-IR axioms, the IR-to-ISA rewrites generated from the ISA description, and the identity-instruction rewrites.

Four further symbols recur throughout.

Phase_1, Phase_2. The two phases of the compilation algorithm. $\mathsf{Phase}_1(G)$ denotes the sequence of pii graphs enumerated by tensor instruction selection on the tiled kernel G, with $\mathsf{Phase}_1(G)[i]$ its i^{th} element. $\mathsf{Phase}_2(\mathsf{PG}^H)$ denotes the assembly code produced by tensor memory allocation on the pii graph PG^H, or FAIL if the constraint problem is unsatisfiable.

Γ. The pii node budget used by the extraction stage. ACT interleaves exploration and extraction, raising the node limit $N_{\max} = \Gamma(k)$ with the exploration iteration count k, where Γ is strictly monotonically increasing and unbounded (for example $\Gamma(k) = 2^k$).

CSP_4. The interference constraint of Module 5, $\mathsf{CSP}_4 = \bigwedge_{(i,j)\in\phi} \mathsf{disjoint}(i, j)$, where the interference graph ϕ pairs two nodes exactly when they write the *same* tensor buffer and their live ranges overlap. It forces such a pair to be given non-overlapping addresses within that buffer; nodes on different buffers are never constrained against each other.

We also use freely that $\equiv_{\mathcal{R}}$ is an equivalence relation, which Def. A.8 establishes from the closure of $\mathcal{R}$ under inversion and composition. This is what licenses chaining equivalences in §C.5. Lemma C.2 additionally requires $\equiv_{\mathcal{R}}$ to be a *congruence* — that replacing a sub-graph by an equivalent one leaves the enclosing graph equivalent — which is what makes reasoning e-class by e-class sound.

We provably guarantee that ACT generates *sound* and *complete* compiler backends:

Theorem C.1. *For every ISA description* ISA^H*, the backend ACT*(ISA^H) *is sound (Def. A.12) and complete (Def. A.13).*

We break the proof of Theorem C.1 into five lemmas, combined in §C.5.

C.2 The Intermediary pii Graph

The proofs are organized around an intermediary representation that sits between the tiled kernel and the assembly code: a graph whose nodes are instructions with their computational attributes fixed but their addressing attributes still unknown. Writing the unknown as ?, we call these *partially instantiated instruction* (pii) nodes. Splitting the problem at exactly this point is what lets Phase 1 reason about α alone and Phase 2 about β alone.

Def C.1. A **pii graph** PG^H is a directed acyclic graph of N pii nodes $\theta_i(\alpha_i, ?)|_{i=1}^N$, which can be transformed into a concrete tensor computation by the substitutions $\theta_i(\alpha_i, ?) \to g_{\theta_i}(\alpha_i)$. A pii node

is either an instruction $\theta_{\alpha,?}$ from the instruction set Θ^H, or one of the identity instructions $\text{slice}^H_{\alpha,?}$ and $\text{concat}^H_{\alpha,?}$.

*Def C.2 ($\equiv^H_{\mathcal{R}}$ **lifted to pii graphs**).* Semantic equivalence between a pii graph PG^H and a tiled kernel G is defined as

$$\text{PG}^H \equiv^H_{\mathcal{R}} G \iff \Lambda_{pii}(\text{PG}^H) \equiv_{\mathcal{R}} \Lambda_{RHS}(G),$$
$$\text{where} \quad \Lambda_{pii}(\text{PG}^H)(M, X) = \text{concat}_1(\text{bflat}(X), \text{PG}^H(X)).$$

Similarly, $\text{ASM}^H_G \equiv^H_{\mathcal{R}} \text{PG}^H \iff \Lambda_{LHS}(\text{ASM}^H_G) \equiv_{\mathcal{R}} \Lambda_{pii}(\text{PG}^H)$. Here $\text{PG}^H(X)$ denotes the byte-level output the pii graph produces in HBM, so that it has the same tensor-type as $\text{bflat}(G(X))$ in Def. A.10, and Λ_{LHS}, Λ_{RHS} are as defined in §A.4. The memory-state argument M is carried so that Λ_{pii}, Λ_{LHS} and Λ_{RHS} share one input signature. It is inert in Λ_{pii} and Λ_{RHS}, which read only X; Λ_{LHS} does read it, which is why Def. A.11 quantifies over all M.

Def. C.2 reuses the symbol $\equiv^H_{\mathcal{R}}$ of Def. A.11 on two further pairs of objects. All three readings unfold to $\equiv_{\mathcal{R}}$ on $\text{u8}[\text{mem}_{out}]$ tensors, so they compose, which is what §C.5 relies on.

C.3 Theoretical Guarantees of Phase 1

Phase 1 performs tensor instruction selection. Its two guarantees are that everything it enumerates is correct, and that it enumerates everything.

Lemma C.2. *Every enumerated pii graph is equivalent to the input tiled kernel G, i.e.,*

$$\forall \boldsymbol{i},\ \textbf{Phase}_1(G)[\boldsymbol{i}] \equiv^H_{\mathcal{R}} G.$$

Proof. Equality saturation only ever replaces a term by one the rewrite set declares equal, so it suffices that every rewrite used in Module 2 is semantics-preserving. There are three kinds. The IR-to-IR rewrites are a subset of the foundational axioms $\mathcal{R}$, hence valid by definition of $\equiv_{\mathcal{R}}$. The IR-to-ISA rewrites are derived mechanically from the ISA description: each rewrites a sub-graph matching the abstract tensor computation g_θ into the corresponding pii node $\theta_{\alpha,?}$, and is therefore valid by Def. A.4. The identity-instruction rewrites are valid because slice^H and concat^H are by construction equal to their tensor-operator counterparts slice and concat. Since each rewrite preserves $\equiv_{\mathcal{R}}$ and $\equiv_{\mathcal{R}}$ is transitive, every e-class of the saturated e-graph contains only terms equivalent to the initial one, and hence every extracted pii graph is equivalent to G. □

Lemma C.3. *All equivalent pii graphs are enumerated, i.e.,*

$$\forall \textbf{PG}^H \equiv^H_{\mathcal{R}} G,\ \exists \boldsymbol{i}\ \textit{s.t.}\ \textbf{Phase}_1(G)[\boldsymbol{i}] = \textbf{PG}^H.$$

Proof. Let $\text{PG}^H \equiv^H_{\mathcal{R}} G$ have N_0 nodes. By Def. C.2 this means $\Lambda_{pii}(\text{PG}^H) \equiv_{\mathcal{R}} \Lambda_{RHS}(G)$, so the two are joined by a finite sequence of rewrites; since Λ_{pii} and Λ_{RHS} differ from PG^H and G only by the common $\text{concat}_1(\text{bflat}(X), \cdot)$ wrapper, the same sequence joins PG^H to G. Every rewrite in it is drawn from the three kinds Module 2 supplies, and the exploration stage applies every applicable rewrite at each iteration, so there is a finite k_0 after which PG^H is represented in the e-graph. Because Γ is strictly monotonically increasing and unbounded, the set $\{k : N_0 \le \Gamma(k)\}$ is non-empty; let $k_1 = \min\{k : N_0 \le \Gamma(k)\}$ be its least element. At iteration $k' = \max(k_0, k_1)$ the graph is both present in the e-graph and within the extraction node budget, so it is among the graphs extracted at k'. □

C.4 Theoretical Guarantees of Phase 2

Phase 2 performs tensor memory allocation. It contributes one soundness lemma, one completeness lemma, and one structural lemma relating assembly codes back to pii graphs.

Lemma C.4. *Whenever Phase 2 succeeds, the assembly code it emits is equivalent to the input pii graph, i.e.,*

$$\textbf{Phase}_2(\textbf{PG}^H) \neq \textbf{FAIL} \implies \textbf{Phase}_2(\textbf{PG}^H) \equiv^H_{\mathcal{R}} \textbf{PG}^H.$$

Proof. Write $\text{ASM}^H_G = \text{Phase}_2(\text{PG}^H)$ and transform $\Lambda_{LHS}(\text{ASM}^H_G)$ into $\Lambda_{pii}(\text{PG}^H)$. By Def. A.6 each executed instruction contributes an upslice of its result into its output buffer, and each later instruction that consumes a value contributes a slice of some buffer. Working backwards from the last instruction, every slice-of-upslice pair that arises falls into exactly one of two cases.

The read is the consumer of that write. Module 5's def-use constraint assigns the consumer the same data slice addresses the producer was given, so the pair is $\text{slice}_{[s:e]}(\text{upslice}_{[s:e]}(M, X))$, which reduces to X: the consumer receives precisely the tensor the producer computed, and the intermediate buffer write disappears while the value it carried is retained.

The read belongs to some other value. Then the two data slices have overlapping live ranges, so the interference constraint CSP_4 has assigned them non-overlapping addresses within their common buffer, discharging the precondition of

$$\text{slice}_{[s_1:e_1]}(\text{upslice}_{[s_2:e_2]}(M, X)) \to \text{slice}_{[s_1:e_1]}(M) \quad \text{under} \quad \text{disjoint}([s_1{:}e_1], [s_2{:}e_2]),$$

which steps the read back past a write it cannot observe.

Alternating the two cases eliminates every intermediate buffer write while preserving each consumed value, leaving exactly the composition of g_θ sub-graphs that constitutes $\Lambda_{pii}(\text{PG}^H)$. Both rewrites preserve $\equiv_{\mathcal{R}}$, so the whole transformation does. The second has been verified in full generality using TensorRight [2], and Fig. 7 visualizes it. □

Lemma C.5. *Phase 2 does not fail if an equivalent assembly code exists, subject to the completeness of the CP-SAT solver, i.e.,*

$$\exists \textbf{ASM}^H_G \equiv^H_{\mathcal{R}} \textbf{PG}^H \implies \textbf{Phase}_2(\textbf{PG}^H) \neq \textbf{FAIL}.$$

Proof. Phase 2 exhaustively searches topological orderings, using interference-graph pruning (§6.4 of the main paper) which discards only orderings whose interference graph is a supergraph of one already shown unsatisfiable, and so preserves completeness.[4] The search therefore covers the ordering in which the instructions of ASM^H_G appear. For that ordering, the data slice addresses already present in ASM^H_G constitute a satisfying assignment to the constraint problem, so the CSP is satisfiable and Phase 2 does not fail. The qualification on CP-SAT is necessary because ACT delegates satisfiability to an external solver; the argument shows a model exists, not that the solver finds it. □

Lemma C.6. *Every assembly code has an equivalent pii graph, i.e.,*

$$\forall \textbf{ASM}^H_G,\ \exists \textbf{PG}^H,\ \textbf{ASM}^H_G \equiv^H_{\mathcal{R}} \textbf{PG}^H.$$

Proof. We construct the pii graph from $\Lambda_{LHS}(\text{ASM}^H_G)$. By Def. A.6 and Def. A.9, $\Lambda_{LHS}(\text{ASM}^H_G)$ consists of g_θ sub-graphs together with bflat, slice and upslice nodes. The bflat nodes coincide with those of Λ_{pii} and can be ignored. Each g_θ sub-graph is replaced by the corresponding pii node

[4] This is the direction implied by the monotonicity of CSP_4 under ϕ, namely $\phi_1 \subseteq \phi_2 \implies \neg\text{CSP}(\phi_1) \to \neg\text{CSP}(\phi_2)$; §6.4 of the main paper states the containment the other way round.

$\theta_{\alpha,?}$. Each slice node is replaced by a slice^H node, and each upslice node is decomposed into slice^H and concat^H nodes using

$$\text{upslice}_{[s:e]}(M, X) \to \text{concat}_1(\text{concat}_1(\text{slice}_{[0:s]}(M), X), \text{slice}_{[e:]}(M)).$$

The result satisfies Def. C.1, so it is a pii graph, and each step preserves $\equiv_{\mathcal{R}}$. Note where the identity instructions earn their place: without slice^H and concat^H as admissible pii nodes there would be no way to express the decomposition above, and this lemma — and with it completeness — would fail. The generalized rewrite has been verified using TensorRight [2], and Fig. 8 visualizes it. □

C.5 Proving Soundness and Completeness

We now assemble Theorem C.1 from the lemmas of §C.3 and §C.4.

Proof of Theorem C.1, soundness. Suppose $\text{Compiler}^H(G) \notin \{\text{FAIL}, \perp\}$, so the backend emitted an assembly code $\text{ASM}_G^H = \text{Phase}_2(\text{PG}^H)$ for some enumerated $\text{PG}^H = \text{Phase}_1(G)[i]$. Lemmas C.4 and C.2 give the two halves of the chain:

$$\Lambda_{LHS}(\text{ASM}_G^H) \equiv_{\mathcal{R}} \Lambda_{pii}(\text{PG}^H) \qquad \text{and} \qquad \Lambda_{pii}(\text{PG}^H) \equiv_{\mathcal{R}} \Lambda_{RHS}(G).$$

By transitivity of $\equiv_{\mathcal{R}}$ (Def. A.8) it follows that $\Lambda_{LHS}(\text{ASM}_G^H) \equiv_{\mathcal{R}} \Lambda_{RHS}(G)$, which is exactly $\text{Compiler}^H(G) \equiv_{\mathcal{R}}^H G$. Hence Compiler^H is sound in the sense of Def. A.12. □

Proof of Theorem C.1, completeness. Suppose there exists $\text{ASM}_G^H \equiv_{\mathcal{R}}^H G$. Def. A.13 requires $\text{Compiler}^H(G) \notin \{\text{FAIL}, \perp\}$, so we must exclude both outcomes.

Not FAIL. By Lemma C.6 there is a pii graph PG^H with $\text{ASM}_G^H \equiv_{\mathcal{R}}^H \text{PG}^H$. Unfolding both hypotheses, $\Lambda_{LHS}(\text{ASM}_G^H) \equiv_{\mathcal{R}} \Lambda_{pii}(\text{PG}^H)$ and $\Lambda_{LHS}(\text{ASM}_G^H) \equiv_{\mathcal{R}} \Lambda_{RHS}(G)$; by symmetry and transitivity of $\equiv_{\mathcal{R}}$ we get $\Lambda_{pii}(\text{PG}^H) \equiv_{\mathcal{R}} \Lambda_{RHS}(G)$, that is $\text{PG}^H \equiv_{\mathcal{R}}^H G$. By Lemma C.3, Phase 1 enumerates this PG^H at some finite index. By Lemma C.5, Phase 2 does not fail on it. So the backend emits an assembly code rather than returning FAIL.

Not $\perp$. Let k' be the finite iteration at which Lemma C.3 places PG^H among the extracted graphs. Each exploration iteration applies finitely many rewrites to a finite e-graph and each extraction is bounded by the node budget $\Gamma(k)$, so iterations 1 through k' complete in finite time. For each of the finitely many pii graphs enumerated up to that point, Phase 2 searches a finite set of topological orderings and solves a finite constraint problem. The backend therefore reaches PG^H and succeeds on it after finitely many steps, and so terminates.

Hence $\text{Compiler}^H(G) \notin \{\text{FAIL}, \perp\}$ and Compiler^H is complete in the sense of Def. A.13. □

The termination argument above applies only under the hypothesis of Def. A.13, namely that an equivalent assembly code exists. When none exists, the algorithm may enumerate indefinitely and return $\perp$ — and by Theorem A.1 no generator can do better while remaining sound and complete.

C.6 Visualization of the Relevant Rewrite Rules

The two rewrite rules that carry the Phase 2 proofs are visualized below on concrete tensors, using the colour convention of Appendix D.

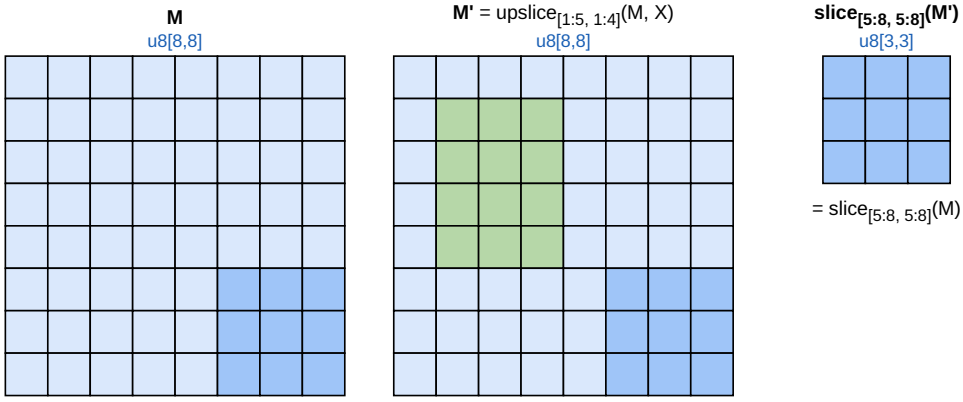


Fig. 7. Visualization of the rewrite rule $\text{slice}_{[s_1:e_1]}(\text{upslice}_{[s_2:e_2]}(M, X)) \equiv_{\mathcal{R}} \text{slice}_{[s_1:e_1]}(M)$, valid under the precondition $\text{disjoint}([s_1{:}e_1], [s_2{:}e_2])$. Because the written region (green) and the read region are disjoint, the read returns the same bytes whether or not the write happened, so the upslice can be discarded.

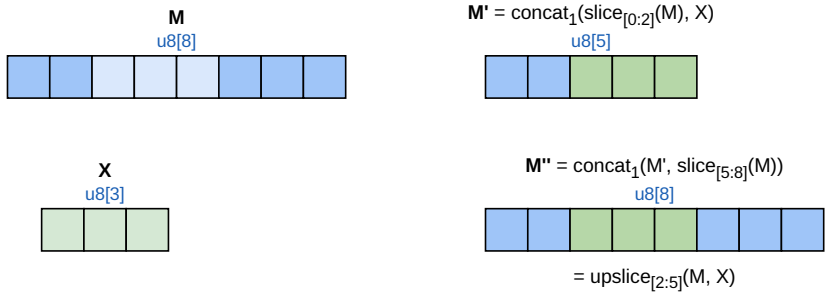


Fig. 8. Visualization of the rewrite rule $\text{upslice}_{[s:e]}(M, X) \equiv_{\mathcal{R}} \text{concat}_1(\text{concat}_1(\text{slice}_{[0:s]}(M), X), \text{slice}_{[e:]}(M))$. Writing X into the middle of M is the same as concatenating the untouched prefix of M, then X, then the untouched suffix — which is what lets an upslice be re-expressed using only the identity instructions slice^H and concat^H.

D Visualization of Tensor Operators

The tensor operators of Table 2 are the vocabulary in which ACT states everything else: tiled kernels are graphs over these operators, the abstract computation g_θ of every instruction is a graph over these operators, and the foundational axioms $\mathcal{R}$ are rewrites between such graphs. Their operational semantics are specified normatively by XLA-HLO [3, 4], and denotational semantics for a subset are given by TensorRight [2]. This appendix restates the operator list for convenience and then visualizes each operator on a small concrete example, since the shape-manipulating operators in particular are far easier to read as pictures than as index arithmetic.

The visualizations share one colour convention throughout:

Blue marks the first operand,

Green marks the second operand, and

Peach marks a tensor whose element values are *computed* rather than moved.

Colouring the operands separately means that when an operator merely rearranges elements, the reader can trace each output cell back to the operand it came from. A darker shade marks a region that is only *part* of a larger tensor — the window a slice reads, or the range an upslice writes — so that partial access is distinguishable at a glance from whole-tensor access. Cells are annotated with element values only where the values themselves carry information; for the purely structural operators the grid geometry is the whole story.

Tensor Operator	Operator Attributes	Description
$y = \mathsf{slice}_{[s:e]}(x)$	$\mathrm{type}(x), s, e$	Read sub-tensor $x[s{:}e]$
$y = \mathsf{upslice}_{[s:e]}(x_1, x_2)$	$\mathrm{type}(x_1), s, e$	Update sub-tensor $x_1[s{:}e] = x_2$
$y = \mathsf{concat}_{dim}(x_1, x_2)$	$\mathrm{type}(x_1), \mathrm{type}(x_2), dim$	Concatenate x_1, x_2 across dimension dim
$y = \mathsf{reshape}(x)$	$\mathrm{type}(x), \mathrm{type}(y)$	Reshape from $\mathrm{type}(x)$ to $\mathrm{type}(y)$
$y = \mathsf{bitcvt}(x)$	$\mathrm{type}(x), \mathrm{type}(y)$	Bitcast conversion between basetypes
$y = \mathsf{broadcast}_{dim}(x)$	$\mathrm{type}(x), dim$	Broadcast over dimension dim
$y = \mathsf{reduce}_{dim}(x)$	$\mathrm{type}(x), dim$	Reduce-sum over dimension dim
$y = \mathsf{transpose}_{[dims]}(x)$	$\mathrm{type}(x), dims$	Permute the dimension order to $dims$
$y = \mathsf{dot}_{(c_1,c_2)}(x_1, x_2)$	$\mathrm{type}(x_1), \mathrm{type}(x_2), c_1, c_2$	Dot product of x_1, x_2 over contracting dimensions c_1, c_2
$y = \exp(x)$	$\mathrm{type}(x)$	Element-wise exponential
$y = \mathrm{div}(x_1, x_2)$	$\mathrm{type}(x_1), \mathrm{type}(x_2)$	Element-wise divide
$y = \mathrm{copy}(x)$	$\mathrm{type}(x)$	Identical copy

Table 2. Brief descriptions of the tensor operators discussed in the paper (detailed operational semantics in [3, 4] and denotational semantics for a subset in [2]). $\mathrm{type}(x)$ refers to the tensor-type of tensor variable x. dim represents the dimension number (starting from 1). We use NumPy-like slice notation $x[s_1{:}e_1, s_2{:}e_2, \dots]$ and ignore type-based attributes in the visualizations.

D.1 Addressing Operators

slice and upslice are the two operators that name a sub-region of a tensor by address, and they are the pair on which ACT's memory allocation rests: a data slice written by one instruction and read by another is expressed as an upslice followed by a slice, and Appendix C turns exactly that composition into the rewrite rule that discharges Phase 2's correctness. Note that upslice returns a tensor of the shape of its *first* operand, not of the sub-tensor being written.

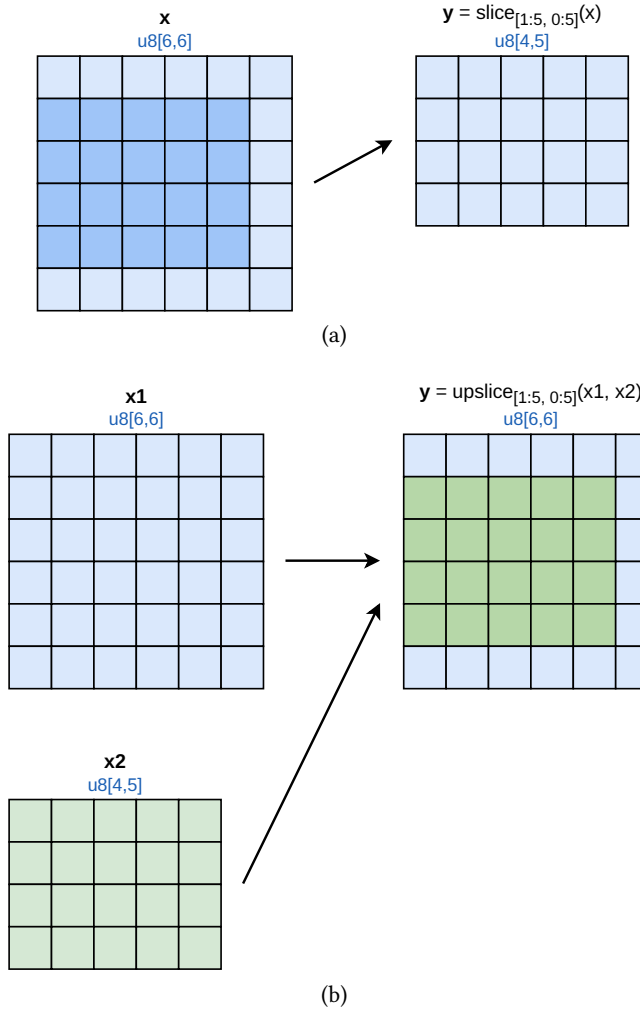


Fig. 9. The addressing operators. (a) slice extracts the 4×5 sub-tensor of x spanning rows 1–4 and columns 0–4; the darker region marks the elements read. (b) upslice returns a copy of x_1 with the 4×5 tensor x_2 written into the range $[1{:}5, 0{:}5]$. The output therefore has the shape of x_1, and the green region records which of its elements came from x_2.

D.2 Structural Operators

Every output element of these operators is a copy of some input element: they rearrange, replicate or reinterpret the shape of a tensor without doing arithmetic. This is exactly the class whose outputs stay blue or green in the visualizations, since each one can be traced back to the operand it came from. concat and reshape are of particular interest to ACT: reshape together with transpose is how ACT models tiled memory layouts (Appendix B), and concat is one of the two identity instructions whose availability makes Phase 1 complete.

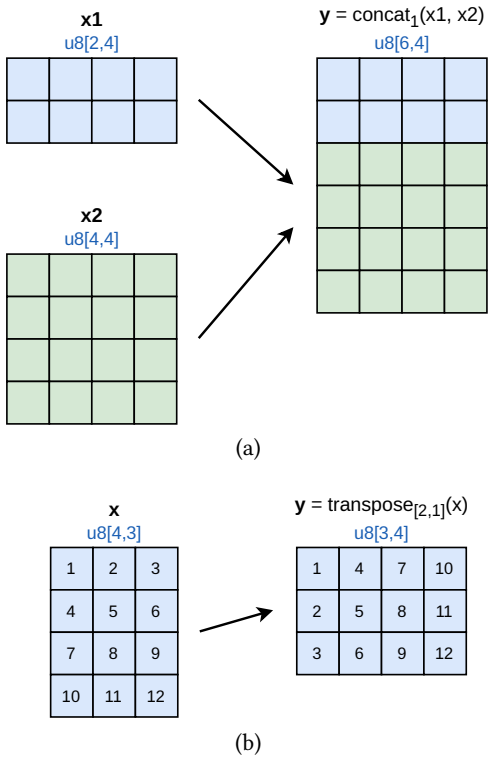


Fig. 10. Concatenation and transposition. (a) concat composes a $[6, 4]$ tensor from x_1 of shape $[2, 4]$ and x_2 of shape $[4, 4]$ by stacking them across dimension 1; the output colours record which operand contributed each row. (b) transpose permutes the dimension order, taking x from $[4, 3]$ to $[3, 4]$. The element values are shown so that the permutation can be read off directly.

D.3 Computational Operators

The remaining operators produce element values that are not present in any operand, and so are drawn in peach. bitcvt belongs here even though it invents no information: reinterpreting the bytes of a u32 as four u8 values yields elements that did not exist before, and its output cannot be traced back cell-for-cell to its input.

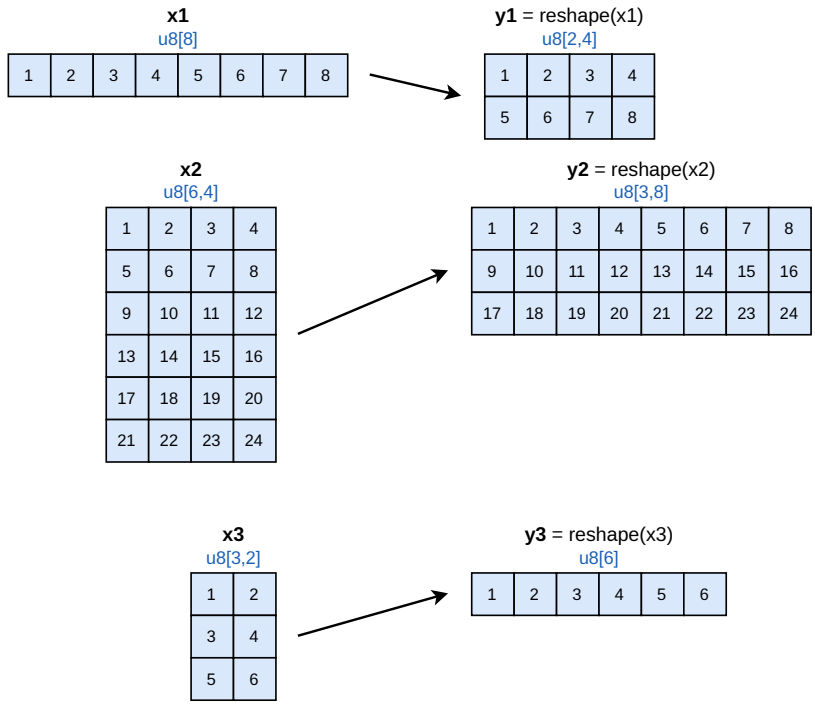


Fig. 11. reshape preserves the row-major element order and changes only the tensor-type, as shown by the element values carrying across each of the three examples: $[8] \rightarrow [2,4]$, $[6,4] \rightarrow [3,8]$, and $[3,2] \rightarrow [6]$. Because the element order is fixed, a reshape is fully determined by its input and output types, which is what allows ACT to treat it as a layout annotation rather than as data movement.

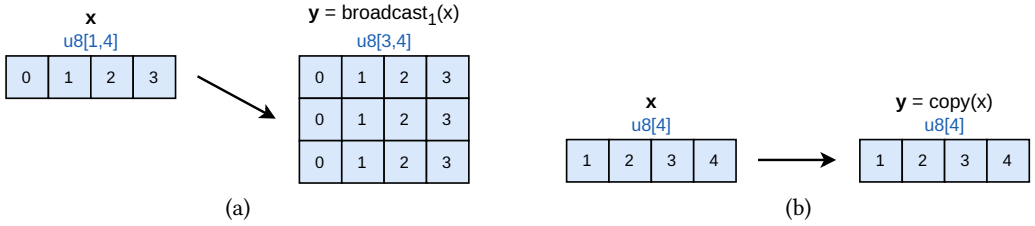


Fig. 12. Replication and copying. (a) broadcast replicates x across dimension 1, turning $[1,4]$ into $[3,4]$; every output row is a copy of the single input row. (b) copy copies x into y unchanged. ACT keeps copy as an explicit operator even though it changes nothing, because a copy between two tensor buffers is real data movement that the cost model has to account for.

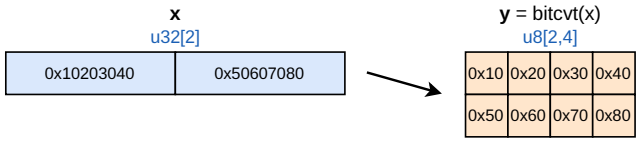


Fig. 13. bitcvt performs an element-wise bitcast between basetypes without changing the underlying bytes. Converting from u32 to u8 therefore turns each input element into four output elements, and the cell widths in the figure are drawn in proportion to the basetype width to make that four-to-one correspondence visible. bitcvt is how ACT reconciles the byte-addressable view of global memory with the typed view used by tensor operators.

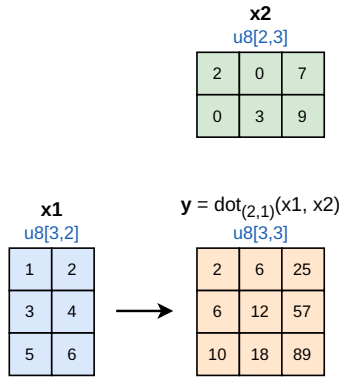


Fig. 14. $y = \mathrm{dot}_{(2,1)}(x_1, x_2)$ matrix-multiplies x_1 of shape $[3, 2]$ with x_2 of shape $[2, 3]$, contracting dimension 2 of x_1 against dimension 1 of x_2 to give y of shape $[3, 3]$. The operands are placed so that each output element sits at the intersection of the row of x_1 and the column of x_2 that produce it.

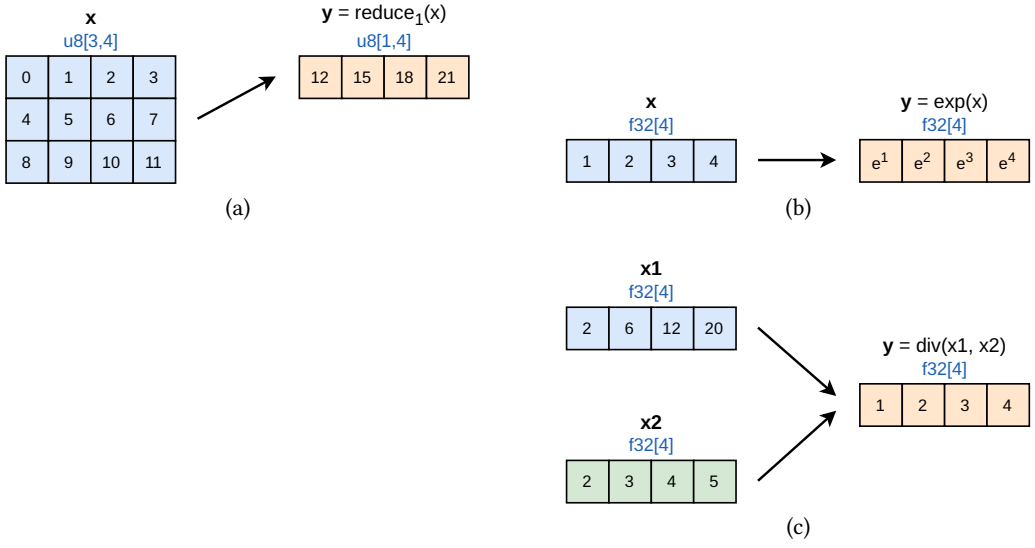


Fig. 15. Reduction and element-wise arithmetic. (a) reduce sum-reduces x across dimension 1, turning $[3, 4]$ into $[1, 4]$; each output element is the sum of the corresponding input column. (b) exp is the element-wise exponential. (c) div divides x_1 by x_2 element-wise. reduce is the counterpart of the broadcast of Fig. 12: both act along one dimension, but only reduce combines values.

E NNSmith Kernel Generation Setup

The compilation-time analysis of §8.7 of the main paper needs tiled kernels that are representative in *size* rather than in workload. Real Gemmini kernels do not serve: the ones used for the performance comparison were capped at about ten nodes because the baselines were hand-written kernel libraries, which exist only for small, common shapes. Tiled kernels in practice run to 10–50 nodes, as seen in the TPUGraphs dataset [14]. We therefore generated 100 synthetic kernels with NNSmith [13], a random DNN generator, and used them both for that compilation-time analysis and for fuzz testing ACT-Gemmini. This appendix records the generation setup so the kernel population can be reproduced exactly.

E.1 Operator Set

The generated kernels exercise the 10 XLA-HLO operators listed in Table 3, but only six of them — dot, add, sub, neg, min and max — are requested directly by the include-list of Fig. 16. The other four arrive indirectly. reverse is not in NNSmith's TensorFlow operator set, so we request `core.Abs`, which lowers to it; `core.Abs` is therefore the one include-list entry with no counterpart in Table 3. broadcast and reduce are introduced by XLA's own lowering and tiling passes rather than being generated, which is the same way they arise in real kernels. clamp likewise appears only after lowering, when XLA fuses an adjacent max and min pair into a single operator. We consider this indirection a feature rather than a limitation of the setup: it means the kernel population contains operators in the same combinations that XLA itself produces, rather than only those a generator was told to emit.

Tensor Operator	Description
$y = \mathsf{dot}(x_1, x_2)$	Matrix multiplication of x_1 and x_2
$y = \mathsf{broadcast}_{dim}(x)$	Broadcast over dimension dim
$y = \mathsf{reduce}_{dim}(x)$	Reduce-sum over dimension dim
$y = \mathsf{reverse}_{dim}(x)$	Reverse elements over dimension dim
$y = \mathsf{add}(x_1, x_2)$	Element-wise addition of x_1 and x_2
$y = \mathsf{sub}(x_1, x_2)$	Element-wise subtraction of x_1 and x_2
$y = \mathsf{neg}(x)$	Element-wise negation of x
$y = \mathsf{min}(x, c)$	Element-wise minimum of x and c
$y = \mathsf{max}(x, c)$	Element-wise maximum of x and c
$y = \mathsf{clamp}(low, x, high)$	Element-wise clamp of x with low and $high$

Table 3. The 10 XLA-HLO operators present in the tiled kernels generated for fuzz testing using NNSmith [13]. Here c, low and $high$ denote constant operands.

E.2 Generation Parameters and Resulting Kernel Sizes

Each invocation fixes a seed, a node budget, an operator include-list, a model type and a backend; Fig. 16 shows one representative command line.

The node budget needs a word of explanation, because the size of the kernel that finally reaches ACT is neither the budget nor the number of operators NNSmith emits. `mgen.max_nodes` is an upper bound on the operators generated, and the resulting graph has multiple outputs, whereas ACT consumes single-output tiled kernels. We therefore combine all outputs with a tree of add operators, so the node count of the final XLA-HLO graph is

$$(\#\text{operators}) + (\#\text{input parameters}) + (\#\text{output nodes} - 1).$$

The input parameters and the combining add nodes are what push the final graph above the operator count: with `mgen.max_nodes` set to 50, kernels reach up to 89 nodes. Because the budget is only an upper bound, the number of operators NNSmith actually emits — and hence the number of outputs that have to be combined — varies from seed to seed. Varying `mgen.seed` across the 100 invocations therefore yields final sizes spanning 7 to 89 nodes, which is the population reported in §8.7 of the main paper. That range brackets the 10–50 nodes typical of the TPUGraphs kernels, so the compilation-time measurements cover both smaller and substantially larger inputs than are seen in practice.

```
nnsmith.model_gen model.type=tensorflow backend.type=xla \
    mgen.seed=1002 \
    mgen.max_nodes=50 \
    mgen.include="[tensorflow.TFMatMul, core.Abs, core.Add, core.Max, core.Min,
    core.Neg, core.Sub]" \
    model.path="nnsmith_output/" \
    debug.viz=true

```

Fig. 16. A representative NNSmith invocation used to generate one of the 100 synthetic kernels. Only `mgen.seed` varies across the population.

F Loop Fission Factors for the Exo Kernels

§8.6 of the main paper compares ACT-Gemmini against Exo's expert-written kernel library on six GEMM shapes and reports one speedup per shape. That single number is the outcome of a search: for each shape, ACT-Gemmini enumerates candidate loop fission factors and its cost model selects one. This appendix gives the full candidate tables behind those six numbers, so that the reported speedup can be seen in context rather than taken on trust.

F.1 Reading the Tables

We evaluated every candidate on Gemmini's Verilator-based RTL simulator and recorded its cycle count. Each row is one combination of loop fission factors, together with the accumulator and scratchpad rows it occupies, its measured cycle count, and its speedup over the original hand-optimized Exo kernel.

The five loop variables split into three levels. The outer loops *io* and *jo* control outer tiling. The inner loops *i* and *j* are the fine-grained tile sizes that load and store into Gemmini's scratchpad. The innermost loop *ko* iterates over the partial sums held in the accumulator: within one *ko* iteration the kernel loads a single 16×64 tile of A and four 16×64 tiles of B, multiplies them as 16×16 matrix multiplications, accumulates the partial sums, and so computes one 16×64 tile of the output C. Candidates whose factors exceed the accumulator or scratchpad row limits are infeasible and are not listed.

In every table the original Exo factors are **bolded** and the best factors, meaning those with the lowest cycle count, are **highlighted in orange**. Where the two coincide, ACT-Gemmini rediscovered Exo's hand-tuned factors and the speedup is 1.00x. Where the highlighted row has a lower cycle count than the bolded row, ACT-Gemmini found a better loop fission factor than the expert did. This happened for three of the six kernels, which is the result summarized in §8.6 of the main paper.

F.2 Candidate Tables

The six tables follow in the order 512×512×512, 784×1024×256, 12544×256×64, 12544×64×256, 3136×512×128, 3136×128×512. Note that this is *not* the left-to-right order of the bars in Fig. 21 of the main paper, which places 784×1024×256 last rather than second; match tables to bars by shape rather than by position.

io	i	jo	j	ko	Accumulator Rows	Scratchpad Rows	Cycles	Speedup
2	**16**	**2**	**4**	**8**	**256**	**16384**	**624170**	**1.00**
2	16	4	2	8	128	12288	624228	1.00
4	8	2	4	8	256	12288	659384	0.95
2	16	8	1	8	64	10240	650725	0.96
8	4	2	4	8	256	10240	855191	0.73
16	2	2	4	8	256	9216	1039928	0.60
32	1	2	4	8	256	8704	1351586	0.46
4	8	4	2	8	128	8192	659357	0.95
4	8	8	1	8	64	6144	687626	0.91
8	4	4	2	8	128	6144	965931	0.65
16	2	4	2	8	128	5120	1223264	0.51
32	1	4	2	8	128	4608	1549222	0.40
8	4	8	1	8	64	4096	1240902	0.50
16	2	8	1	8	64	3072	1385353	0.45
32	1	8	1	8	64	2560	1663782	0.38

Table 4. Loop fission candidates for matmul $512 \times 512 \times 512$. ACT-Gemmini's best factor coincides with Exo's hand-tuned factor, giving a 1.00x speedup.

io	i	jo	j	ko	Accumulator Rows	Scratchpad Rows	Cycles	Speedup
1	**49**	**8**	**2**	**4**	**128**	**14592**	**968206**	**1.30**
1	49	16	1	4	64	13568	1126936	1.12
7	**7**	**2**	**8**	**4**	**512**	**9984**	**1258985**	**1.00**
7	7	4	4	4	256	5888	1457055	0.86
7	7	8	2	4	128	3840	1757805	0.72
7	7	16	1	4	64	2816	1857018	0.68
49	1	2	8	4	512	8448	2105258	0.60
49	1	4	4	4	256	4352	2428470	0.52
49	1	8	2	4	128	2304	2442627	0.52
49	1	16	1	4	64	1280	2634956	0.48

Table 5. Loop fission candidates for matmul $784 \times 1024 \times 256$. ACT-Gemmini's best factor outperforms Exo's hand-tuned factor, giving a 1.30x speedup.

io	i	jo	j	ko	Accumulator Rows	Scratchpad Rows	Cycles	Speedup
4	**196**	**1**	**4**	**1**	**256**	**13568**	**1499481**	**1.00**
4	196	2	2	1	128	13056	1513787	0.99
4	196	4	1	1	64	12800	2027815	0.74
7	112	1	4	1	256	8192	1569955	0.96
7	112	2	2	1	128	7680	1602036	0.94
7	112	4	1	1	64	7424	2059381	0.73
8	98	1	4	1	256	7296	1579550	0.95
8	98	2	2	1	128	6784	1604309	0.93
8	98	4	1	1	64	6528	1842807	0.81
14	56	1	4	1	256	4608	1702098	0.88
16	49	1	4	1	256	4160	1781546	0.84
14	56	2	2	1	128	4096	1540089	0.97
14	56	4	1	1	64	3840	2003559	0.75
16	49	2	2	1	128	3648	1696348	0.88
16	49	4	1	1	64	3392	2103647	0.71
28	28	1	4	1	256	2816	1737137	0.86
28	28	2	2	1	128	2304	1720557	0.87
28	28	4	1	1	64	2048	2119776	0.71
49	16	1	4	1	256	2048	1738821	0.86
56	14	1	4	1	256	1920	1733756	0.86
49	16	2	2	1	128	1536	1742679	0.86
98	8	1	4	1	256	1536	1738183	0.86
112	7	1	4	1	256	1472	1738658	0.86
56	14	2	2	1	128	1408	1743286	0.86
196	4	1	4	1	256	1280	1716865	0.87
49	16	4	1	1	64	1280	2183789	0.69
392	2	1	4	1	256	1152	1720090	0.87
56	14	4	1	1	64	1152	2184509	0.69
784	1	1	4	1	256	1088	1793534	0.84
98	8	2	2	1	128	1024	1775427	0.84
112	7	2	2	1	128	960	1745616	0.86
196	4	2	2	1	128	768	1752644	0.86
98	8	4	1	1	64	768	2186091	0.69
112	7	4	1	1	64	704	2105043	0.71
392	2	2	2	1	128	640	1802506	0.83
784	1	2	2	1	128	576	1883025	0.80
196	4	4	1	1	64	512	2175743	0.69
392	2	4	1	1	64	384	2114435	0.71
784	1	4	1	1	64	320	2146024	0.70

Table 6. Loop fission candidates for matmul $12544 \times 256 \times 64$. ACT-Gemmini's best factor coincides with Exo's hand-tuned factor, giving a 1.00x speedup.

io	i	jo	j	ko	Accumulator Rows	Scratchpad Rows	Cycles	Speedup
49	**16**	**1**	**1**	**4**	**1024**	**5120**	**1462706**	**1.29**
56	14	1	1	4	896	4608	1800241	1.05
98	**8**	**1**	**1**	**4**	**512**	**3072**	**1884887**	**1.00**
112	7	1	1	4	448	2816	2114836	0.89
196	4	1	1	4	256	2048	2110773	0.89
392	2	1	1	4	128	1536	1883721	1.00
784	1	1	1	4	64	1280	2171151	0.87

Table 7. Loop fission candidates for matmul $12544 \times 64 \times 256$. ACT-Gemmini's best factor outperforms Exo's hand-tuned factor, giving a 1.29x speedup.

io	i	jo	j	ko	Accumulator Rows	Scratchpad Rows	Cycles	Speedup
2	98	2	4	2	256	14592	1113009	1.06
2	98	4	2	2	128	13568	1205199	0.98
2	98	8	1	2	64	13056	1433162	0.82
4	**49**	**1**	**8**	**2**	**512**	**10368**	**1179333**	**1.00**
4	49	2	4	2	256	8320	1239771	0.95
7	28	1	8	2	512	7680	1156398	1.02
4	49	4	2	2	128	7296	1224648	0.96
4	49	8	1	2	64	6784	1328323	0.89
14	14	1	8	2	512	5888	1426823	0.83
7	28	2	4	2	256	5632	1173115	1.01
28	7	1	8	2	512	4992	1562962	0.75
49	4	1	8	2	512	4608	1641041	0.72
7	28	4	2	2	128	4608	1172983	1.01
98	2	1	8	2	512	4352	1667040	0.71
196	1	1	8	2	512	4224	1691607	0.70
7	28	8	1	2	64	4096	1324928	0.89
14	14	2	4	2	256	3840	1393443	0.85
28	7	2	4	2	256	2944	1520771	0.78
14	14	4	2	2	128	2816	1393532	0.85
49	4	2	4	2	256	2560	1611814	0.73
14	14	8	1	2	64	2304	1589328	0.74
98	2	2	4	2	256	2304	1656962	0.71
196	1	2	4	2	256	2176	1692347	0.70
28	7	4	2	2	128	1920	1520284	0.78
49	4	4	2	2	128	1536	1615518	0.73
28	7	8	1	2	64	1408	1689620	0.70
98	2	4	2	2	128	1280	1656610	0.71
196	1	4	2	2	128	1152	1691181	0.70
49	4	8	1	2	64	1024	1782025	0.66
98	2	8	1	2	64	768	1817532	0.65
196	1	8	1	2	64	640	1836277	0.64

Table 8. Loop fission candidates for matmul $3136 \times 512 \times 128$. ACT-Gemmini's best factor outperforms Exo's hand-tuned factor, giving a 1.06x speedup.

io	i	jo	j	ko	Accumulator Rows	Scratchpad Rows	Cycles	Speedup
14	14	1	2	8	128	11264	1145575	1.00
14	14	2	1	8	128	11264	1205532	0.95
28	7	1	2	8	128	7680	1150215	1.00
28	7	2	1	8	128	7680	1215381	0.94
49	4	1	2	8	128	6144	1408841	0.81
49	4	2	1	8	128	6144	1419196	0.81
98	2	1	2	8	128	5120	1599727	0.72
98	2	2	1	8	128	5120	1639071	0.70
196	1	1	2	8	128	4608	1699842	0.67
196	1	2	1	8	128	4608	1739270	0.66

Table 9. Loop fission candidates for matmul $3136 \times 128 \times 512$. ACT-Gemmini's best factor coincides with Exo's hand-tuned factor, giving a 1.00x speedup.

G Detailed Analysis of ACT-Gemmini versus the Gemmini SW Library

§8.6 of the main paper evaluates ACT-Gemmini against the Gemmini software library over three groups of kernels: *"supported"*, where the library has a hand-written implementation; *"composite"*, which compose several library calls; and *"not supported"*, whose operators have no Gemmini implementation at all. The paper reports the headline numbers and defers the per-kernel discussion here. This appendix takes the last two groups in turn, explains where the speedups come from, and then reports two results that did not fit in the paper: an experiment with Gemmini's CISC-type instructions, and the convolution kernels.

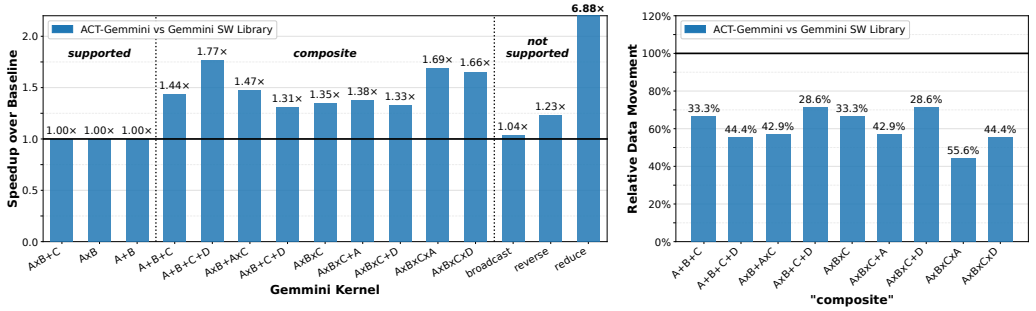


Fig. 17. ACT-Gemmini against the Gemmini SW library. Left: speedup of the compiled code across all three kernel groups, reproduced from Fig. 21 of the main paper so that the two panels can be read side by side. Right: data movement of ACT-Gemmini relative to the library for the *"composite"* kernels, where lower is better; the number printed on each bar is the *reduction*, that is 100% minus the plotted height. Note that the *"not supported"* bars on the left are measured against host execution, since those operators have no Gemmini implementation in the library.

G.1 *"composite"* Kernels

A *"composite"* kernel is a composition of several GEMM (×) and ADD (+) operations that the library can only execute as a sequence of separate calls. Each library call must leave its result in HBM, because the library's calling convention cannot assume anything about scratchpad residency across calls. The next call then loads that same tensor straight back in. ACT-Gemmini has no such constraint: it compiles the whole kernel at once, so an intermediate that is produced and immediately consumed never leaves the accelerator.

Fig. 18 makes the difference concrete for $A \times B \times C$. The library issues two GEMM kernels, and the store of $A{\times}B$, the fence, and the reload of $A{\times}B$ between them (shaded red) are pure overhead. ACT-Gemmini keeps the intermediate in the scratchpad and emits neither. For $A \times B \times C$ the library moves six tiles where ACT-Gemmini moves four, a **33.3%** reduction, and the measured speedup is **1.35x**. The saving grows with scratchpad utilization, because a larger resident intermediate displaces proportionally more traffic, and it grows with the length of the chain, because each additional library call contributes another store-fence-load round trip that ACT-Gemmini does not emit.

The same pattern holds across the other *"composite"* kernels, since each additional library call in the baseline adds another store-fence-load round trip that ACT-Gemmini removes. The gain is not monotone in kernel size, however: how much is saved also depends on how much of the intermediate stays resident, so two of the three-call kernels in Fig. 17 score slightly below the best two-call one. For chains of GEMM, the speedup goes from 1x for $A \times B$, to 1.35x for $A \times B \times C$,

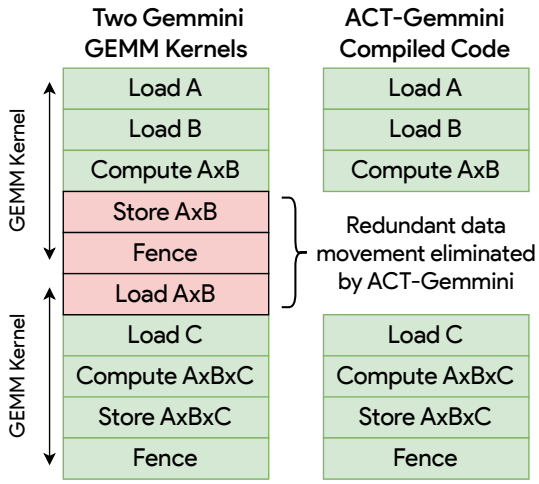


Fig. 18. Breakdown of the Gemmini SW library kernels and the ACT-Gemmini compiled code for $A \times B \times C$. ACT-Gemmini eliminates the redundant load-store instructions between library calls (shaded red).

to 1.66x for $A \times B \times C \times D$; for chains of ADD it goes from 1x, to 1.44x, to 1.77x. Taken over the group, Fig. 17 shows ACT-Gemmini outperforming the library by up to **1.77x** (geomean 1.48x) while reducing data movement by up to **55.6%** (average 39.3%).

G.2 *"not supported"* Kernels

The Gemmini SW library has no implementation of broadcast, reverse or reduce, so a conventional toolchain has no choice but to run them on the host Rocket CPU. ACT-Gemmini instead compiles all three onto the systolic array, and it does so without any operator-specific support.

The mechanism is compile-time constant detection (§5.3 of the main paper). Each of these three operators can be written as a matrix multiplication against a fixed matrix: broadcast against a $[1, 16]$ all-ones vector, reduce against a $[16, 1]$ all-ones vector, and reverse against a reversed identity matrix. ACT-Gemmini detects that those operands are compile-time constants, materializes them in constant memory, and emits an ordinary matmul. Fig. 19 shows the resulting pii graphs, with the constant memory region shaded purple and the constant-valued nodes eye (the identity matrix) and ones (the all-ones vector) in orange. Nothing in this is Gemmini-specific: the same constant detection would apply to any accelerator whose ISA description exposes a matrix-multiply instruction.

Because the baseline here is host execution rather than an accelerator kernel, the comparison is *cross-architecture*, and it should be read as the practical cost of host fallback rather than as an accelerator-to-accelerator speedup.

The outcome varies sharply across the three operators, and it is worth being precise about why. reduce gains the most, at **6.88x** (§8.6 of the main paper): it is the only one of the three that does real arithmetic proportional to its input, so moving it off the Rocket CPU removes a genuine compute bottleneck. reverse gains **1.23x** and broadcast only **1.04x**. Both are pure data movement, so on the host they are already memory-bound; expressing them as a systolic-array matmul against a constant removes the host round trip but replaces it with an accelerator that is no faster at moving the same bytes. The value of compiling them is therefore not raw speed but *coverage* — the kernel no longer has to leave the accelerator at all, which is what makes the surrounding *"composite"*

fusions of §G.1 possible in the first place. For reference on the same axis, `add(A,B)` reaches 6.82x over host execution, though it is a *"supported"* kernel that the library does implement.

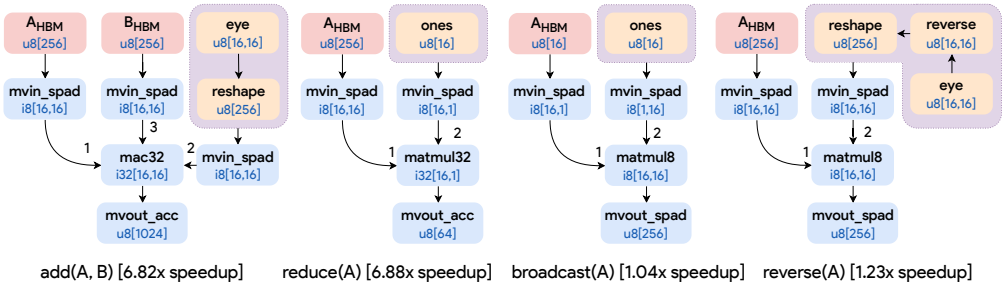


Fig. 19. pii graphs with constant memory (purple region) generated by ACT-Gemmini. The orange nodes hold the values materialized in constant memory: eye is an identity matrix and ones is a vector of all ones. Speedups are with respect to the Gemmini host processor (Rocket chip).

G.3 CISC-type Instructions

The TAIDL description used to generate ACT-Gemmini covers Gemmini's 11 RISC-type instructions. Gemmini [6] also provides experimental CISC-type instructions, which perform a matrix multiplication over a large tile in a single instruction and stream data directly to and from HBM. These are implemented with a hardware FSM that double-buffers data movement against computation. That gives them an advantage a RISC-type sequence cannot match: a compiler emitting separate load, compute and store instructions is limited by the instruction scheduler's reordering and pipelining window, whereas the FSM overlaps the two by construction. We therefore expected CISC-type instructions to win even where a RISC-type schedule moves less data.

To test this, we added the CISC-type instructions to the TAIDL description and generated a new backend, ACT-Gemmini+. The outcome was as predicted. ACT-Gemmini+ still enumerated the RISC-only assembly among its candidates — the RISC-type instructions remained in the description, so those candidates were never lost — but its cost model ranked a CISC-type candidate above them, and that is what it emitted.

The result itself is unsurprising; what matters is what it took to obtain. ACT required *no* algorithmic changes to exploit the new instruction class — only the TAIDL description was extended, and the cost-model-driven enumeration of §7 of the main paper preferred the faster code automatically. A hand-written backend would have needed a new scheduling strategy to use an FSM-driven instruction well. This is evidence that ACT generalizes across RISC- and CISC-style instructions of the same accelerator.

G.4 Convolution Kernels

Convolutions on Gemmini are lowered onto the same systolic array as matrix multiplications and share its compute unit, so they exercise the same instruction selection and memory allocation decisions. We observed correspondingly similar results for convolution kernels, both against the Gemmini SW library and against Exo's expert-written kernel library. Since they add no separate insight, §8.6 of the main paper reports no convolution kernels.

H The oneDNN Kernels Compiled by ACT-AMX

§8.5 of the main paper evaluates ACT-AMX against oneDNN, a library hand-written and heavily optimized for Intel ISA including AMX and AVX512. The comparison is unusual among our evaluations in that the target is not a speedup but a *match*: oneDNN's assembly is already expert-tuned, and the question is whether an automatically generated backend can reach it. This appendix presents the five kernels behind that claim so the match can be inspected rather than taken on trust.

For each kernel we give three artifacts: its tensor computation graph, the XLA-HLO IR that ACT-AMX takes as input, and the assembly ACT-AMX emits. The emitted assembly is the same hand-optimized code the oneDNN library produces — ACT-AMX enumerates it as one candidate among many and its cost model selects it (§8.5 of the main paper) — which is why the two match in performance. For the fp32 and mixed kernels (K3–K5) the assembly listings show a representative excerpt, with elided lines marked . . . .

H.1 Kernel Selection and Statistics

The XLA compiler emits tiled kernels which are then compiled to assembly using the oneDNN library, via the `xla_cpu_use_mkl_dnn` flag. We selected five kernels commonly observed in the oneDNN examples [9], listed in Table 10. They are deliberately simple as computations — all five are matrix multiplications with element-wise post-processing — but carry complex memory layouts on their inputs, which is what makes them a demanding test of Module 1's layout modeling (Appendix B).

Table 10 separates the size of the input from the size of the output. #Nodes counts every instruction of the kernel's tensor computation graph, parameters and constants included, except the ROOT tuple that packages K5's two outputs, which is bookkeeping rather than computation. #AMX and #AVX512 count the AMX tile instructions and AVX512 instructions in the compiled assembly. The split across the five kernels is clean and follows their element types: the two int8 kernels (K1, K2) compile purely to AMX tile instructions, the two fp32 kernels (K3, K4) purely to AVX512, and the mixed kernel (K5) to a combination of both. This much is forced rather than chosen — the TAIDL description we used covers AMX instructions only for int8 and AVX512 instructions only for fp32, so no other assignment is type-correct. What the cost model decides is the code *within* each class: which tiling, which register blocking, which instruction order.

Kernel	oneDNN example	#Nodes	#AMX	#AVX512
K1	`cnn_inf_amx`	6	16	0
K2	`rnn_inf_amx`	6	9	0
K3	`mem_format_avx`	7	0	100
K4	`sgemm_avx`	7	0	107
K5	`cnn_inf_mix`	23	16	37

Table 10. The five selected oneDNN kernels and their optimized assembly code in the oneDNN library. #Nodes is the number of operators in the tensor computation graph; #AMX and #AVX512 count instructions of each class in the compiled assembly.

H.2 K1: `cnn_inf_amx`

K1 is the int8 convolution-inference kernel and the simplest of the five: a single matrix multiplication of a u8[32,64] activation against an i8[32,64] weight tensor carried in the AMX-native tiled layout

{T(16,4)}. That layout is the VNNI packing the tdpbusd instruction requires, with four elements of the reduction dimension interleaved; §8.2 of the main paper shows the corresponding transformation on tile register src1. The compiled code is 16 AMX instructions with no AVX512 at all: four tilezero, four tileloadd, four tdpbusd, and four tilestored, tiling the 32×32 output into four 16×16 tile registers.

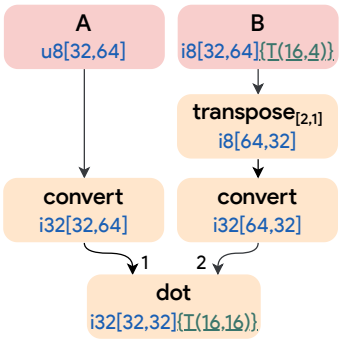


Fig. 20. Tensor computation graph for oneDNN kernel K1.

```
ENTRY k1 {
  A = u8[32,64] parameter(0)
  B = i8[32,64]{T(16,4)} parameter(1)
  convert.0 = i32[32,64] convert(A)
  transpose.1 = i8[64,32] transpose(B), dimensions=[2,1]
  convert.2 = i32[64,32] convert(transpose.1)
  ROOT dot.3 = i32[32,32]{T(16,16)} dot(convert.0, convert.2)
}
```

Fig. 21. XLA-HLO code for kernel K1.

```
tilezero(dst = tmm0)
tilezero(dst = tmm1)
tilezero(dst = tmm2)
tilezero(dst = tmm3)
tileloadd(dst = tmm4, mem_addr = 0x0000)
tileloadd(dst = tmm5, mem_addr = 0x0400)
tileloadd(dst = tmm6, mem_addr = 0x0800)
tileloadd(dst = tmm7, mem_addr = 0x0c00)
tdpbusd(dst = tmm0, src0 = tmm4, src1 = tmm6)
tdpbusd(dst = tmm1, src0 = tmm4, src1 = tmm7)
tdpbusd(dst = tmm2, src0 = tmm5, src1 = tmm6)
tdpbusd(dst = tmm3, src0 = tmm5, src1 = tmm7)
tilestored(src = tmm0, mem_addr = 0x1000)
tilestored(src = tmm1, mem_addr = 0x1400)
tilestored(src = tmm2, mem_addr = 0x1800)
tilestored(src = tmm3, mem_addr = 0x1c00)
```

Fig. 22. Compiled assembly code for kernel K1.

H.3 K2: `rnn_inf_amx`

K2 is the int8 recurrent-inference kernel, structurally identical to K1 but with a half-height activation, u8[16,64] instead of u8[32,64]. The consequence is visible in the assembly: because the output is 16×32 rather than 32×32, it occupies two tile registers instead of four, and the instruction count falls from 16 to 9. Nothing else changes, which is the point — the same generated backend handles both without any shape-specific path.

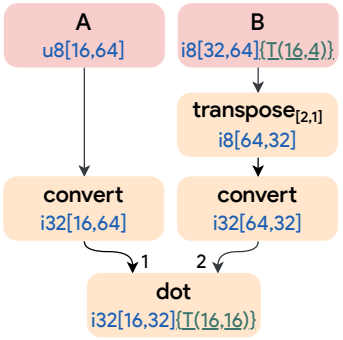


Fig. 23. Tensor computation graph for oneDNN kernel K2.

```
ENTRY k2 {
  A = u8[16,64] parameter(0)
  B = i8[32,64]{T(16,4)} parameter(1)
  convert.0 = i32[16,64] convert(A)
  transpose.1 = i8[64,32] transpose(B), dimensions=[2,1]
  convert.2 = i32[64,32] convert(transpose.1)
  ROOT dot.3 = i32[16,32]{T(16,16)} dot(convert.0, convert.2)
}
```

Fig. 24. XLA-HLO code for kernel K2.

```
tilezero(dst = tmm0)
tilezero(dst = tmm1)
tileloadd(dst = tmm2, mem_addr = 0x0000)
tileloadd(dst = tmm3, mem_addr = 0x0400)
tileloadd(dst = tmm4, mem_addr = 0x0800)
tdpbusd(dst = tmm0, src0 = tmm2, src1 = tmm3)
tdpbusd(dst = tmm1, src0 = tmm2, src1 = tmm4)
tilestored(src = tmm0, mem_addr = 0x0c00)
tilestored(src = tmm1, mem_addr = 0x1000)
```

Fig. 25. Compiled assembly code for kernel K2.

H.4 K3: `mem_format_avx`

K3 is an fp32 kernel exercising memory-format propagation. It is an outer product: a f32[32] vector reshaped to a column, multiplied by a single sliced column of f32[14,4] transposed to a row, with the result transposed into the tiled layout {1,0:T(14,16)}. Since AMX has no fp32 tile instruction, ACT-AMX compiles it entirely to AVX512: 28 accumulator registers are zeroed, the two input rows

are loaded once, and 14 iterations each broadcast one scalar and issue two fused multiply-adds, followed by 28 stores. The excerpt below shows the first two iterations and the last.

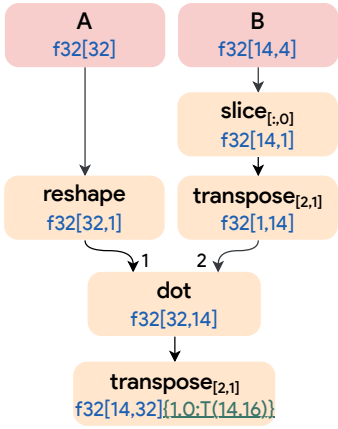


Fig. 26. Tensor computation graph for oneDNN kernel K3.

```
ENTRY k3 {
  A = f32[32] parameter(0)
  B = f32[14,4] parameter(1)
  reshape.0 = f32[32,1] reshape(A)
  slice.1 = f32[14,1] slice(B), dimensions=[:,0]
  transpose.2 = f32[1,14] transpose(slice.1), dimensions=[2,1]
  dot.3 = f32[32,14] dot(reshape.0, transpose.2)
  ROOT transpose.4 = f32[14,32]{1,0:T(14,16)} transpose(dot.3), dimensions=[2,1]
}
```

Fig. 27. XLA-HLO code for kernel K3.

```
vpxord(dst = zmm31, src0 = zmm31, src1 = zmm31)
...
vpxord(dst = zmm4, src0 = zmm4, src1 = zmm4)
vmovupsz(dst = zmm2, mem_addr=0x000)
vmovupsz(dst = zmm3, mem_addr=0x040)
vbroadcastssl(dst = zmm0, mem_addr=0x080)
vfmadd231ps(dst = zmm31, src0 = zmm3, src1 = zmm0)
vfmadd231ps(dst = zmm30, src0 = zmm2, src1 = zmm0)
vbroadcastssl(dst = zmm0, mem_addr=0x090)
vfmadd231ps(dst = zmm29, src0 = zmm3, src1 = zmm0)
vfmadd231ps(dst = zmm28, src0 = zmm2, src1 = zmm0)
...
vbroadcastssl(dst = zmm0, mem_addr=0x150)
vfmadd231ps(dst = zmm5, src0 = zmm3, src1 = zmm0)
vfmadd231ps(dst = zmm4, src0 = zmm2, src1 = zmm0)
vmovupsz(src = zmm31, mem_addr=0x160)
vmovupsz(src = zmm30, mem_addr=0x1a0)
...
vmovupsz(src = zmm4, mem_addr=0x820)
```

Fig. 28. Compiled assembly code for kernel K3. Iteration n accumulates into the register pair $(\text{zmm}_{31-2(n-1)}, \text{zmm}_{30-2(n-1)})$, so the 14 iterations consume the 28 accumulators zeroed at the top.

H.5 K4: sgemm_avx

K4 is the fp32 general matrix multiplication. Its graph has the same topology as K3's, but the two dimensions move in opposite directions: the vector grows from f32[32] to f32[48] while the sliced row shrinks from 14 elements to 8. That is what changes the register blocking. K3 holds 28 accumulators and runs 14 iterations of two fused multiply-adds; K4 holds 24 and runs 8 iterations of three, against three loaded input registers rather than two, giving 107 AVX512 instructions. The vaddpsz block near the end accumulates a further operand read from memory before the results are stored.

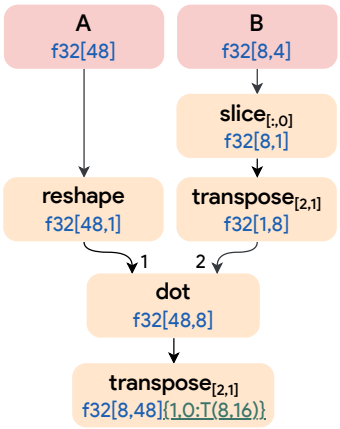


Fig. 29. Tensor computation graph for oneDNN kernel K4.

```
ENTRY k4 {
  A = f32[48] parameter(0)
  B = f32[8,4] parameter(1)
  reshape.0 = f32[48,1] reshape(A)
  slice.1 = f32[8,1] slice(B), dimensions=[:,0]
  transpose.2 = f32[1,8] transpose(slice.1), dimensions=[2,1]
  dot.3 = f32[48,8] dot(reshape.0, transpose.2)
  ROOT transpose.4 = f32[8,48]{1,0:T(8,16)} transpose(dot.3), dimensions=[2,1]
}
```

Fig. 30. XLA-HLO code for kernel K4.

```
vpxord(dst = zmm8, src0 = zmm8, src1 = zmm8)
...
vpxord(dst = zmm31, src0 = zmm31, src1 = zmm31)
vmovupsz(dst = zmm0, mem_addr=0x000)
vmovupsz(dst = zmm1, mem_addr=0x040)
vmovupsz(dst = zmm2, mem_addr=0x080)
vbroadcastssl(dst = zmm6, mem_addr=0x0c0)
vfmadd231ps(dst = zmm8,  src0 = zmm6, src1 = zmm0)
vfmadd231ps(dst = zmm16, src0 = zmm6, src1 = zmm1)
vfmadd231ps(dst = zmm24, src0 = zmm6, src1 = zmm2)
vbroadcastssl(dst = zmm7, mem_addr=0x0d0)
vfmadd231ps(dst = zmm9,  src0 = zmm7, src1 = zmm0)
vfmadd231ps(dst = zmm17, src0 = zmm7, src1 = zmm1)
vfmadd231ps(dst = zmm25, src0 = zmm7, src1 = zmm2)
...
vbroadcastssl(dst = zmm6, mem_addr=0x120)
vfmadd231ps(dst = zmm14, src0 = zmm6, src1 = zmm0)
vfmadd231ps(dst = zmm22, src0 = zmm6, src1 = zmm1)
vfmadd231ps(dst = zmm30, src0 = zmm6, src1 = zmm2)
vbroadcastssl(dst = zmm7, mem_addr=0x130)
vfmadd231ps(dst = zmm15, src0 = zmm7, src1 = zmm0)
vfmadd231ps(dst = zmm23, src0 = zmm7, src1 = zmm1)
vfmadd231ps(dst = zmm31, src0 = zmm7, src1 = zmm2)
vaddpsz(dst = zmm8, src0 = zmm8, mem_addr=0x140)
...
vaddpsz(dst = zmm31, src0 = zmm31, mem_addr=0x700)
vmovupsz(src = zmm8, mem_addr=0x740)
...
vmovupsz(src = zmm31, mem_addr=0xd00)
```

Fig. 31. Compiled assembly code for kernel K4. The accumulators zmm8–zmm31 are consumed in ascending order by both the vaddpsz block and the stores that follow it.

H.6 K5: `cnn_inf_mix`

K5 is the mixed-precision convolution-inference kernel and the most interesting of the five, because it is the only one whose graph spans both element types and therefore both instruction classes. Its graph has two independent halves. The first is exactly K1's int8 matrix multiplication, which compiles to the same 16 AMX instructions. The second is an fp32 post-processing chain — scale, bias, ReLU, scale again, clamp to $[0, 255]$, and convert down to u8 — which compiles to 37 AVX512 instructions. Neither half is annotated as AMX or AVX512 anywhere in the input; ACT-AMX enumerates candidates for the whole graph and emits the mixture without any per-kernel direction. This is the concrete sense in which ACT does not need per-kernel tuning: the same generated backend produces pure-AMX code for K1, pure-AVX512 code for K3, and the correct mixture here.

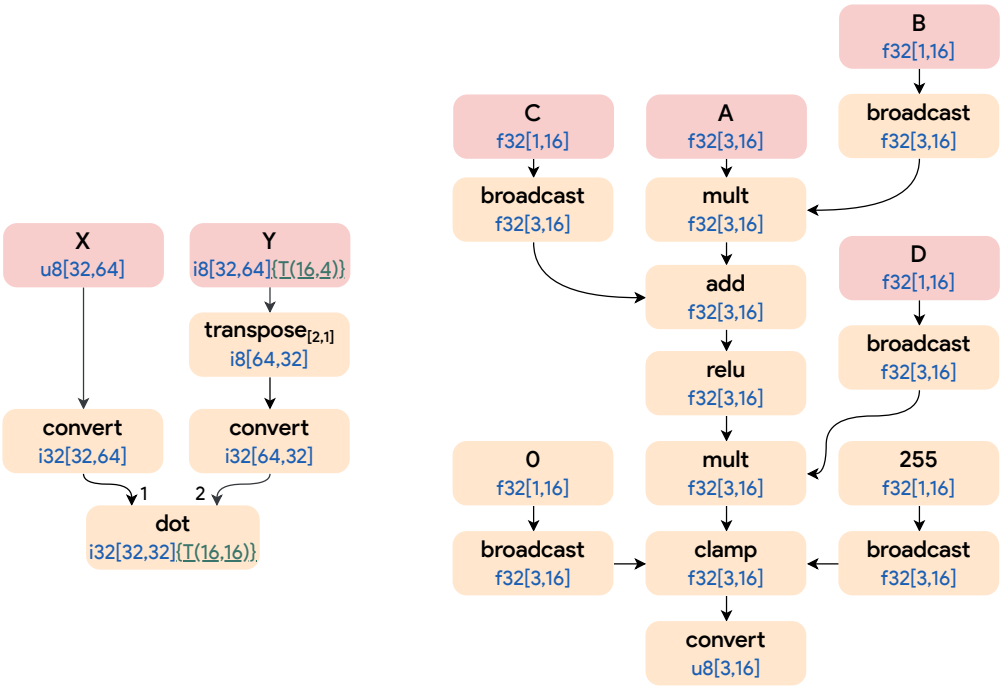


Fig. 32. Tensor computation graph for oneDNN kernel K5. The left half is K1's int8 matrix multiplication; the right half is the fp32 post-processing chain.

```
ENTRY k5 {
  X = u8[32,64] parameter(0)
  Y = i8[32,64]{T(16,4)} parameter(1)
  convert.0 = i32[32,64] convert(X)
  transpose.1 = i8[64,32] transpose(Y), dimensions=[2,1]
  convert.2 = i32[64,32] convert(transpose.1)
  dot.3 = i32[32,32]{T(16,16)} dot(convert.0, convert.2)
  A = f32[3,16] parameter(2)
  B = f32[1,16] parameter(3)
  C = f32[1,16] parameter(4)
  D = f32[1,16] parameter(5)
  constant.0 = f32[1,16] constant(0)
  constant.255 = f32[1,16] constant(255)
  broadcast.4 = f32[3,16] broadcast(B), dimensions=[1]
  multiply.5 = f32[3,16] multiply(A, broadcast.4)
  broadcast.6 = f32[3,16] broadcast(C), dimensions=[1]
  add.7 = f32[3,16] add(multiply.5, broadcast.6)
  broadcast.11 = f32[3,16] broadcast(constant.0), dimensions=[1]
  broadcast.12 = f32[3,16] broadcast(constant.255), dimensions=[1]
  relu.8 = f32[3,16] maximum(broadcast.11, add.7)
  broadcast.9 = f32[3,16] broadcast(D), dimensions=[1]
  multiply.10 = f32[3,16] multiply(relu.8, broadcast.9)
  clamp.13 = f32[3,16] clamp(broadcast.11, multiply.10, broadcast.12)
  convert.14 = u8[3,16] convert(clamp.13)
  ROOT tuple.15 = tuple(dot.3, convert.14)
}
```

Fig. 33. XLA-HLO code for kernel K5. The ROOT tuple packages the two outputs and is not counted as an operator in Table 10.

```
tilezero(dst = tmm0)
tilezero(dst = tmm1)
tilezero(dst = tmm2)
tilezero(dst = tmm3)
tileloadd(dst = tmm4, mem_addr = 0x0000)
tileloadd(dst = tmm5, mem_addr = 0x0400)
tileloadd(dst = tmm6, mem_addr = 0x0800)
tileloadd(dst = tmm7, mem_addr = 0x0c00)
tdpbusd(dst = tmm0, src0 = tmm4, src1 = tmm6)
tdpbusd(dst = tmm1, src0 = tmm4, src1 = tmm7)
tdpbusd(dst = tmm2, src0 = tmm5, src1 = tmm6)
tdpbusd(dst = tmm3, src0 = tmm5, src1 = tmm7)
tilestored(src = tmm0, mem_addr = 0x1000)
tilestored(src = tmm1, mem_addr = 0x1400)
tilestored(src = tmm2, mem_addr = 0x1800)
tilestored(src = tmm3, mem_addr = 0x1c00)
vmovupsz(dst = zmm10, mem_addr = 0x000)
vmovupsz(dst = zmm15, mem_addr = 0x040)
mov(dst = ebx, imm = 0x437f0000)
vmovq(dst = xmm9, src = rbx)
vbroadcastss(dst = zmm9, src = xmm9)
vmovupsz(dst = zmm31, mem_addr = 0x080)
vcvtdq2ps(dst = zmm31, src = zmm31)
vmovupsz(dst = zmm30, mem_addr = 0x0c0)
vcvtdq2ps(dst = zmm30, src = zmm30)
vmovupsz(dst = zmm29, mem_addr = 0x100)
vcvtdq2ps(dst = zmm29, src = zmm29)
vmulps(dst = zmm31, src0 = zmm31, src1 = zmm15)
...
vaddps(dst = zmm31, src0 = zmm31, src1 = zmm10)
...
vmaxps(dst = zmm31, src0 = zmm31, src1 = zmm8)
...
vbroadcastssl(dst = zmm0, mem_addr = 0x144)
vmulps(dst = zmm31, src0 = zmm31, src1 = zmm0)
...
vmaxps(dst = zmm31, src0 = zmm31, src1 = zmm8)
vminps(dst = zmm31, src0 = zmm31, src1 = zmm9)
vcvtps2dq(dst = zmm31, src = zmm31)
vpmovusdbx(src = zmm31, mem_addr = 0x144)
...
```

Fig. 34. Compiled assembly code for kernel K5. The AMX half is emitted first and is identical to Fig. 22; the AVX512 half follows. The immediate `0x437f0000` is the fp32 encoding of 255.0, broadcast into zmm9 as the clamp upper bound.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA. https://www.pearson.com/en-us/subject-catalog/p/compilers-principles-techniques-and-tools/P200000003472

[2] Jai Arora, Sirui Lu, Devansh Jain, Tianfan Xu, Farzin Houshmand, Phitchaya Mangpo Phothilimthana, Mohsen Lesani, Praveen Narayanan, Karthik Srinivasa Murthy, Rastislav Bodik, Amit Sabne, and Charith Mendis. 2025. TensorRight: Automated Verification of Tensor Graph Rewrites. *Proc. ACM Program. Lang.* 9, POPL, Article 29 (Jan. 2025), 32 pages. doi:10.1145/3704865

[3] OpenXLA Contributors. 2026. Operation Semantics. https://openxla.org/xla/operation_semantics.

[4] OpenXLA Contributors. 2026. StableHLO Specification. https://github.com/openxla/stablehlo/blob/main/docs/spec.md/.

[5] Will Estes. 2016. Lexical Analysis With Flex, for Flex 2.6.2. https://westes.github.io/flex/manual/index.html.

[6] Hasan Genc, Seah Kim, Alon Amid, Ameer Haj-Ali, Vighnesh Iyer, Pranav Prakash, Jerry Zhao, Daniel Grubb, Harrison Liew, Howard Mao, Albert Ou, Colin Schmidt, Samuel Steffl, John Wright, Ion Stoica, Jonathan Ragan-Kelley, Krste Asanovic, Borivoje Nikolic, and Yakun Sophia Shao. 2021. Gemmini: Enabling Systematic Deep-Learning Architecture Evaluation via Full-Stack Integration. In *Proceedings of the 58th Annual Design Automation Conference (DAC)*. doi:10.1109/DAC18074.2021.9586216

[7] 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* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 703–718. doi:10.1145/3519939.3523446

[8] Intel. 2024. Intel® Intrinsics Guide. https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

[9] Intel. 2025. oneDNN Examples. https://github.com/uxlfoundation/oneDNN/tree/v3.8.1/examples.

[10] 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 58th IEEE/ACM International Symposium on Microarchitecture (MICRO '25)*. Association for Computing Machinery, New York, NY, USA, 1316–1333. doi:10.1145/3725843.3756075

[11] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. 2019. TASO: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles* (Huntsville, Ontario, Canada) *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 47–62. doi:10.1145/3341301.3359630

[12] 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. doi:10.1145/3498717

[13] Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. 2023. NNSmith: Generating Diverse and Valid Test Cases for Deep Learning Compilers. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 530–543. doi:10.1145/3575693.3575707

[14] Mangpo Phothilimthana, Sami Abu-El-Haija, Kaidi Cao, Bahare Fatemi, Michael Burrows, Charith Mendis, and Bryan Perozzi. 2023. TpuGraphs: A Performance Prediction Dataset on Large Tensor Computational Graphs. In *Advances in Neural Information Processing Systems*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine (Eds.), Vol. 36. Curran Associates, Inc., 70355–70375. doi:10.52202/075280-3083

[15] GNU Project. 2021. Bison 3.8.1. https://www.gnu.org/software/bison/manual/bison.html.

[16] Arya Vohra, Leo Seojun Lee, Jakub Bachurski, Oleksandr Zinenko, Phitchaya Mangpo Phothilimthana, Albert Cohen, and William S. Moses. 2025. Mind the Abstraction Gap: Bringing Equality Saturation to Real-World ML Compilers. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 284 (Oct. 2025), 28 pages. doi:10.1145/3763062

[17] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. 2021. PET: Optimizing Tensor Programs with Partially Equivalent Transformations and Automated Corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 37–54. https://www.usenix.org/conference/osdi21/presentation/wang

[18] 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*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268. https://proceedings.mlsys.org/paper_files/paper/2021/file/cc427d934a7f6c0663e5923f49eba531-Paper.pdf

[19] Kahfi S. Zulkifli, Wenbo Qian, Shaowei Zhu, Yuan Zhou, Zhen Zhang, and Chang Lou. 2025. Verifying Semantic Equivalence of Large Models with Equality Saturation. In *Proceedings of the 5th Workshop on Machine Learning and Systems* (World Trade Center, Rotterdam, Netherlands) *(EuroMLSys '25)*. Association for Computing Machinery, New York, NY, USA, 82–89. doi:10.1145/3721146.3721943