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`.

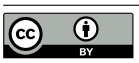
CCS Concepts: • **Software and its engineering → Compilers**; **Retargetable compilers**.

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

ACM Reference Format:
Devansh Jain, Akash Pardeshi, Marco Frigo, Kaustubh Khulbe, Krut Patel, Saatvik Lochan, Jai Arora, and Charith Mendis. 2026. Automatically Generating ML Compiler Backends from Tensor Accelerator ISA Descriptions. *Proc. ACM Program. Lang.* 10, OOPSLA2, Article 325 (October 2026), 32 pages. https://doi.org/10.1145/3839457

*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.



This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 2475-1421/2026/10-ART325
https://doi.org/10.1145/3839457

1 Introduction

In recent years, machine learning (ML) workloads have gained popularity, specifically deep learning models. ML practitioners use ML frameworks such as Tensorflow [1], PyTorch [4], and JAX [35] to express their computations, and ML compilers such as XLA [20], TorchInductor [4], and TVM [17] to lower these computations into highly performant executables targeting CPUs and GPUs.

Akin to general-purpose compilers, ML compilers also use a hierarchical compilation pipeline (Fig. 1). The frontend lowers programs written using ML frameworks to a tensor-operator-based intermediate representation (IR) known as tensor computation graphs. These graphs are then optimized by several graph-level optimization passes [76] such as algebraic simplification and layout selection. The scale of these optimized graphs [75] can range up to 1000s of nodes. This is followed by operator fusion and tiling passes, which emit 100s of *tiled kernels* (sub-graphs with typically 10–50 nodes). These tiled kernels are compiled to device-specific assembly code individually using *existing* compiler backends. For example, XLA [20], a popular ML compiler, uses the existing lowering pipeline in MLIR [59] and LLVM [58] to compile these tiled kernels in XLA-HLO IR [22], XLA's tensor IR, into assembly code for commodity hardware (CPUs and GPUs).

To get the best performance out of ML workloads, many domain-specific tensor accelerators (a.k.a. AI accelerators) have been designed. They have shown superior runtime performance, and the fast proliferation of such accelerator designs in both academia and industry is a testament to their importance and popularity. For example, consider commercial tensor accelerators such as Google TPUs [54, 55], AWS Trainium [80, 81] and Intel AMX [46], and a large body of academic designs such as Gemmini [36], MAERI [57], FEATHER [92], EVA [103]. These accelerators are diverse, with different software-managed memory hierarchies and instruction sets (compute capabilities).

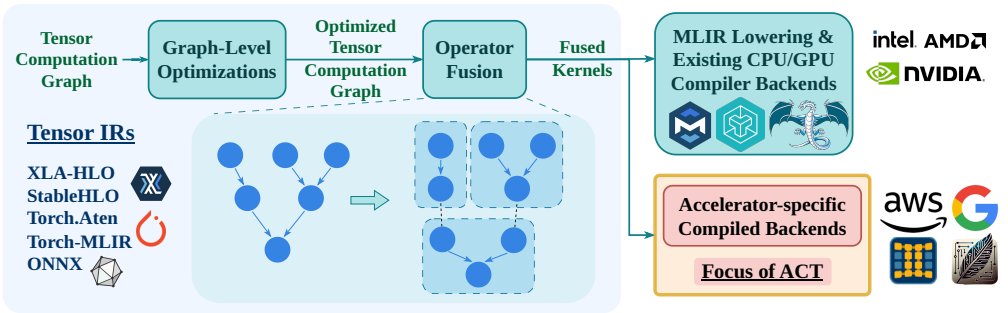


Fig. 1. Typical hierarchical compilation pipeline present in ML compilers. Our solution, ACT, focuses on compiler backend construction (shaded red) for tensor accelerators like AWS Trainium [81] and Gemmini [36]. We use XLA compiler [20] as our reference ML compiler, which uses XLA-HLO IR [22] and StableHLO IR [23].

1.1 Need for Generation of Accelerator-specific Compiler Backends

Widespread use of a hardware architecture requires compiler backends that generate device-specific code, as evident for CPUs and GPUs — companies like Google and Amazon [88] have invested heavily in compiler toolchains for their tensor accelerators [53, 54, 80, 81]. In contrast, a majority of tensor accelerators proposed in the literature do not have mature backends and are often evaluated only on synthetic workloads [43], which *inhibits their integration* with popular ML compilers.

Tensor accelerators without mature compiler backends instead rely on custom hand-written kernel libraries, which support only a limited set of kernels and miss out on fusion opportunities outside this set [25, 41, 69, 70]. For the Gemmini [36] accelerator, we observe that operator fusion can lead to over 1.7x better performance and a 50% reduction in data movement (§8.6). Therefore, compiler backends that support code generation for *arbitrary tiled kernels* are needed.

Manually constructing such backends for every accelerator is too tedious and too slow to adapt to the fast pace of accelerator research (§2.3). Therefore, we propose an *agile* methodology — *automated generation* of compiler backends for the multitude of tensor accelerator designs.

1.2 Challenges & Goals

Automatically generating compiler backends that support code generation for arbitrary tiled kernels for many diverse tensor accelerators comes with new challenges not present in commodity hardware (e.g., prior works like vectorizer generators for vector ISAs [18, 90, 93]).

Challenge 1: Code generation and optimizations for software-managed scratchpads. Most tensor accelerators have software-managed scratchpads with different memory hierarchies. The data access patterns of tensor accelerators vary significantly in shape and size with multi-dimensional addressing (e.g., AWS Trainium [83]) and multi-dimensional base elements (e.g., 16×64 matrices in Intel AMX [46]). An ideal compiler backend needs to *automatically* manage data allocation and reuse for these scratchpads. Works such as Exo [45] model scratchpads themselves, but leave the data allocation to end-users, whereas 3LA [43] does not model scratchpad reuse.

Challenge 2: Handling parameterized ISA instructions. Tensor accelerators have instructions with parameters that control execution counts. For example, the Google TPU [55] and AWS Trainium [83] instructions are parameterized by input shapes (e.g., B×128). Methodologies adopted by existing vectorizer generators [18, 90, 93] can only handle fixed-size vector instructions.

Challenge 3: Handling complex ISA descriptions. Tensor accelerator ISA descriptions often include complex data layout transformations [46, 81, 92], that require accelerator-specific legalization passes to support such instructions. Existing kernel libraries [45, 46] often rely on long and tedious pattern-matching rules manually written by experts for each new accelerator.

Moreover, the compiler backend generator should ideally achieve three goals.

Goal 1: Increased compilation coverage. Tensor accelerator designs often require expensive host fallback to execute certain operators (or sub-graphs) due to their limited compute capabilities. The compiler backend should have a large compilation coverage, i.e., correctly compile for all tiled kernels supported by the accelerator — in other words, a *sound and complete compiler backend.*

Goal 2: Full automation. To minimize the engineering effort, the compiler backend generator should *fully automate* the backend generation process with just the ISA descriptions as input.

Goal 3: Interoperability with existing ML compilers. The generated compiler backends should be *compatible with existing ML compilers* to facilitate seamless integration and deployment.

1.3 Our Solution

In this paper, we introduce **ACT (Accelerator Compiler Toolkit)**, the first compiler backend generator that *automatically* generates *sound and complete* compiler backends for tensor accelerators from just their ISA descriptions. ACT consumes ISA descriptions written in TAIDL [50], a popular ISA description language that supports diverse accelerator designs [36, 46, 55, 83, 92, 103]. ACT achieves automatic generation (goal 2) by *parameterizing* the compilation algorithm with TAIDL [50] constructs, which are concretized for each accelerator given their ISA descriptions. ACT-generated backends are compatible with XLA [20], a popular ML compiler (goal 3). These backends consume tiled kernels in XLA-HLO IR as input and produce accelerator-specific assembly code.

We introduce a novel intermediary, *partially instantiated instructions* (**pii**), to handle instructions parameterized by input matrix shapes (challenge 2), prevalent in tensor accelerator ISAs. We term such instruction parameters/attributes as *computational* ($\boldsymbol{\alpha}$), with the rest termed as *addressing* ($\boldsymbol{\beta}$).

ACT's *ISA-parameterized* compilation algorithm consists of two main phases—one for solving each attribute set. The first phase introduces *parameterized* instruction selection based on equality

saturation [89] and uses multiple types of rewrites (e.g., IR-to-IR, IR-to-ISA). ACT generates the IR-to-ISA rewrites from the TAIDL descriptions. These generated rewrites, combined with the XLA's IR-to-IR rewrites, perform IR legalization (challenge 3) and resolve the computational attributes (α). The second phase introduces *parameterized* memory allocation for multi-dimensional buffer accesses (challenge 1) based on constraint programming on the addressing attributes (β). We introduce intra- and inter-phase fallback paths to ensure completeness of the algorithm (goal 1).

In summary, the paper makes the following contributions.

- We introduce ACT, the first compiler backend generator that *automatically generates* compiler backends for tensor accelerators, given just their ISA descriptions in TAIDL. As a core component of ACT, we present a novel *ISA-parameterized* compilation algorithm. (§4–§6)
- We augment ACT-generated backends with cost models to generate performant code. (§7)
- We use ACT to generate compiler backends integrated with the XLA compiler for **six** popular accelerator platforms from industry (AWS Trainium, Intel AMX, Google TPUv1) and academia (Gemmini, FEATHER, EVA), demonstrating the generality of ACT. (§8.2)
- We show that the ACT-generated compiler backends match or outperform existing accelerator-specific backends and expert-written kernel libraries (up to **6.88x** speedup). We also improve compilation coverage for AWS NKI by **2.3x** over AWS's compiler `neuronx-cc`. (§8.3–§8.6)
- We show that compilation times of the ACT-generated compiler backends are less than a second for a wide range of kernel sizes, even for large kernels (**529 ms** for 390 nodes). (§8.7)

ACT is part of a larger open-source ecosystem built around our ISA description language TAIDL [50]. The ecosystem and its associated tutorials are accessible at https://github.com/act-compiler/act.

2 Background

We first briefly discuss XLA-HLO IR, tensor accelerators, and equality saturation workflow, providing the background needed for the problem formulation (§3) and our solution (§4–§7).

2.1 XLA-HLO IR: XLA Compiler's Tensor IR

XLA-HLO (High Level Operations) IR [22] is a tensor IR used in the XLA compiler [20], the default ML compiler for TensorFlow [1] and JAX [35]. Here, we describe the key concepts of XLA-HLO IR relevant to this paper. These concepts are also applicable to other tensor IRs such as Torch.Aten [4].

Tensors are n-dimensional objects (generalization of 0-D scalars, 1-D vectors, 2-D matrices) that are usually represented as multi-dimensional arrays. A tensor of *tensor-type* $\mathbb{E}[S]$ has a shape S, a tuple of integers denoting the dimension sizes, with elements of scalar basetype $\mathbb{E}$ such as uint8 (u8), int32 (i32), fp32 (f32), bfloat16 (bf16). A *tensor value* has a fixed tensor-type with fixed data for each of its elements. A *tensor variable* has a fixed tensor-type but can hold variable data at runtime.

Tensor Operators. The XLA compiler manipulates tensors as first-class objects using tensor operators, which are usually tensor-type agnostic and parameterized by various parameters, collectively called *operator attributes*. For example, the operator $\text{slice}(x, s, e, p)$ extracts a sub-tensor from an input tensor x of any tensor-type, from indices s to e with a stride of p. Here, s, e, p and the tensor-type of x are operator attributes of slice, with validity conditions such as $0 \leq s < e \leq \text{size}(x)$ and $p > 0$. We call tensor operators parameterized by these attributes *abstract tensor operators*. Table 1 briefly describes the XLA-HLO tensor operators used in this paper. Appendix D visualizes these, with their operational and denotational semantics documented in [22, 23] and [6], respectively.

During the compilation of a given model, all operator attributes are instantiated. We call these instantiated tensor operators *concrete tensor operators*, which take tensor variables as input and produce a tensor variable as output. Consider an abstract tensor operator f instantiated with attributes A, producing a concrete tensor operator f_A. It takes n tensor variables $X = (x_i|_{i=1}^n)$ as input and computes a tensor variable y as output, i.e., $y = f_A(x_1, \ldots x_n)$, or in short, $f_A(X)$.

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

Table 1. Brief descriptions of tensor operators based on XLA-HLO IR [22] discussed in the paper. $\text{type}(x)$ 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 visualizations.

Tensor Computation Graphs. XLA's internal IR, called XLA-HLO IR, represents computations as a *tensor computation graph* — a directed acyclic graph with tensor operators as nodes and operands as edges. The edges establish the data dependencies between different tensor operators. Unless specified otherwise, a tensor computation graph consists of concrete tensor operators.

IR Properties. XLA-HLO IR is a *data flow graph* IR, i.e., it does not have explicit imperative control flow. Control flow is represented using higher-order functional operators such as `while`. XLA-HLO IR only supports *static tensor shapes*, i.e., the shapes of all tensor variables are known at compile time. Shape dynamism is handled by StableHLO passes [21] prior to XLA-HLO IR.

2.2 Tensor Rewrites in XLA Compiler

The XLA compiler performs semantic-preserving transformations on tensor computation graphs using a set of algebraic rewrite rules, termed *tensor rewrites*. A tensor rewrite $P_l \longrightarrow P_r$ substitutes subgraph P_l with subgraph P_r, under certain preconditions on the operator attributes.

XLA uses 175+ tensor rewrites, with 115 of these verified to full generality by TensorRight [6]. In our work, we adopt this rewrite set, denoted as $\mathcal{R}$, as foundational axioms of the XLA-HLO IR. One such rewrite is $\text{slice}_{[s_1:e_1]}(\text{slice}_{[s_2:e_2]}(x)) \longrightarrow \text{slice}_{[s_1+s_2:e_1+s_2]}(x)$. Similar to prior works [51, 96, 100, 104], we define semantic equivalence modulo these tensor rewrites $\mathcal{R}$.

2.3 Tensor Accelerators

Tensor accelerators (a.k.a. AI accelerators) are a class of hardware accelerators optimized for tensor computations using different microarchitectural innovations such as systolic array-based executions. Some have programmable dataflows [36, 57, 77], while others are fixed [16, 19, 29, 55, 102]. They have diverse functional units such as matrix transpositions [54], matrix reshaping units, etc., with novel innovations such as intelligent on-chip routing [92]. TAIDL [50] standardizes how ISA semantics are described for a variety of diverse tensor accelerator designs.

2.4 Term Rewriting using Equality Saturation

Term rewriting [7] is classically destructive, making optimizations sensitive to the order in which rewrites are applied — the well-known "phase-ordering" problem [95]. Equality saturation [89] overcomes it using *e-graphs*, which apply rewrites simultaneously and compactly represent equivalence relations over terms as equivalence classes (*e-classes*) of equivalent nodes (*e-nodes*).

The equality saturation workflow consists of three stages: initialization, exploration, and extraction. First, the initial e-graph is created from the input term. Next, the exploration stage applies rewrites until saturation is reached (i.e., applying rewrites does not add any new information) or a e-node limit is reached. Finally, the optimal term is extracted from the e-graph.

3 Problem Statement

In this section, we formulate the problem of a compiler backend generator that generates sound and complete accelerator-specific backends from just the accelerator ISA description. First (§3.1), we describe the input to these backends by illustrating the code generation pipeline of the XLA compiler [20]. Next (§3.2), we provide a high-level summary of TAIDL [50], a popular ISA description language for tensor accelerators that serves as the input to our generator. Finally (§3.3), we define soundness and completeness, and state the problem of a compiler backend generator.

3.1 Tiled Kernels: Input to Accelerator-specific Compiler Backends

XLA's code generation pipeline starts after the middle-end passes of operator fusion and tiling [75], which partition the optimized tensor computation graph in XLA-HLO IR (§2.1) into several sub-graphs and determine the tile factors for each of these sub-graphs. This pipeline is summarized in Fig. 2, and snippets from different stages of this pipeline are shown in Fig. 3.

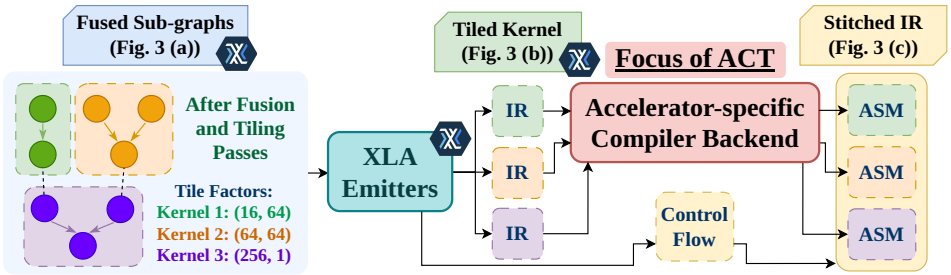


Fig. 2. Summary of code generation pipeline of XLA compiler [20] after operator fusion and tiling passes. Snippets from different stages of this pipeline are shown in Fig. 3. In this paper, we focus on accelerator-specific compiler backends (shaded red) and borrow XLA's existing implementation for the rest of the pipeline.

```
1 ...
2 %fused_subgraph.1 {
3   %Q = bf16[2048,1024]{:T(16,64)}
      parameter(0)
4   ...
5 }
6 ...
7 ENTRY %main {
8   ...
9   %qkv.out = bf16[2048,1024]{:T(16,64)}
      fusion(...), calls=%fused_subgraph.1
10  ...
11 }
```

(a) XLA-HLO IR after operator fusion and tiling passes. The tile factors are annotated by `{:T(...)}` in purple.

```
1 ENTRY %tiled_kernel.1 {
2   %Q = bf16[16,64] parameter(0)
3   ...
4 }
```

(b) Tiled kernel emitted for `%fused_subgraph.1`

```
1 ...
2 scf.for %i = 0 to 128 {
3   ... {
4     %x0 = llvm.inline_asm ...
5     ...
6   }
7 } ...
```

(c) Stitched IR with accelerator-specific assembly code inlined in the control flow for tiling.

Fig. 3. Snippet from different stages of XLA's code generation pipeline. The accelerator-specific compiler backend consumes a tiled kernel (b) and emits assembly code (lines 4–5 in (c)) inlined in the stitched IR.

After XLA middle-end passes, the XLA-HLO IR includes multiple sub-graphs (e.g., Fig. 3 (a) lines 2–5) with tile factors annotated in purple (e.g., `{:T(16,64)}`). First, XLA's `Emitter` modules [24] emit (i) control flow for tiling (e.g., `scf.for` loop in Fig. 3 (c) line 2) and (ii) single-tile versions of all sub-graphs (e.g., Fig. 3 (b)). We name these single-tile versions of sub-graphs as *tiled kernels*.

A **tiled kernel** is syntactically similar to the original sub-graph, except that the tensor shapes are based on the tile factors (e.g., `bf16[16,64]` in Fig. 3 (b) vs. `bf16[2048,1024]` in Fig. 3 (a)). It is a tensor computation graph in XLA-HLO IR (§2.1), which includes static tensor-types, and computes a single tile of the original sub-graph. In this paper, we will visualize tiled kernels as directed acyclic graphs with nodes as tensor operators (from Table 1), for example, in Fig. 4.

The **accelerator-specific compiler backend** compiles these tiled kernels individually into accelerator-specific assembly code. Finally, the generated assembly code for different kernels is inlined and stitched together with control flow for tiling into a single *stitched IR* (e.g., Fig. 3 (c)). We defer the specifics of execution of the stitched IR for various accelerators to evaluations (§8).

> The input to an accelerator-specific compiler backend is a tiled kernel in XLA-HLO IR and the output is a sequential accelerator-specific assembly code.

Running Example. Fig. 4 visualizes a tiled kernel G_{QKV} for QKV computation, which is prevalent in many transformer topologies [27, 94]. For illustration purposes, we assume a single-batch single-head QKV computation over bf16[64, 64] tiles. We use this as a running example for the rest of the paper.

Challenges

Key challenges in designing accelerator-specific compiler backends are the diversity of tensor accelerator designs and the complexity of the ISA descriptions. Different accelerators have different memory hierarchies and instruction granularities, which require different compilation strategies. Furthermore, ISA descriptions of tensor accelerators are often parameterized by input shapes [36, 55, 103] and include complex data layout transformations [46, 81, 92], which require accelerator-specific legalization passes to map tiled kernels to the accelerator ISA.

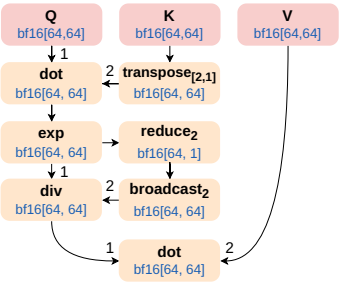


Fig. 4. Self-attention kernels in Transformer models like BERT [27] use QKV computation $\text{softmax}(Q \cdot K^T) \cdot V$ over 4-dimensional tensors. For illustration purposes, we assume a single-batch single-head QKV computation G_{QKV} over bf16[64, 64] tiles.

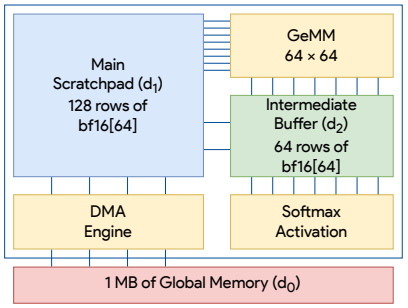
3.2 TAIDL: Tensor Accelerator ISA Description

We choose TAIDL [50], Tensor Accelerator ISA Definition Language, to describe accelerator ISAs since it supports several existing tensor accelerators [36, 46, 55, 83, 92, 103] and can precisely model the semantics of new tensor accelerator ISAs. Here, we provide a high-level summary of TAIDL [50] necessary for later sections. Detailed definitions and explanations are deferred to Appendix A.

TAIDL standardizes the way ISAs and their semantics are developed for tensor accelerators. It builds on top of the existing formal models [6, 51, 60, 97] of tensors and tensor operators. The key principle behind TAIDL is that tensor accelerators support coarse-grained instructions that work on multi-dimensional data (commonly represented as tensors). Therefore, TAIDL descriptions use compositions of tensor operators in XLA-HLO IR to model the instruction semantics.

An *ISA description* comprises (1) the software-managed storage units (collectively termed *data model*) like scratchpads, and (2) the semantics of instructions that perform computations like data movement and compute instructions on these storage units. This is analogous to the x86 ISA description comprising a data model (registers, memory) and an instruction set (add, mov, etc.).

Running Example. Consider a hypothetical accelerator H_{QKV} (Fig. 5 (a)) with two on-chip storage units—a 16 KB scratchpad and an 8 KB intermediate buffer—and access to global memory via a DMA engine. Its two compute units are a general matrix multiply (gemm) unit and a softmax activation unit. It supports five data movement and two compute instructions listed in Fig. 5 (b). We use this hypothetical accelerator H_{QKV} as our running example for the rest of the paper.



(a)

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

(b)

Fig. 5. Running example. (a) High-level accelerator design for a hypothetical accelerator H_{QKV}.
(b) Brief description of its instruction set. Tensor variables are colored based on their storage unit (d_0, d_1, d_2).

(i) TAIDL constructs: Data Model

The software-managed storage units, like scratchpads on an accelerator, are abstractly represented as *tensor buffers*. TAIDL represents a tensor buffer $\boldsymbol{d}$ as a tensor (of type $\mathbb{E}[S_0, S_1]$) with a subset of dimensions representing *access dimensions* ($\boldsymbol{S_0}$) and the remaining dimensions representing the granularity of data access ($\boldsymbol{S_1}$). TAIDL assumes that an accelerator has a byte-addressable 1-dimensional global memory (HBM) that is accessible by the host processor and is denoted d_0.

Running Example. Tensor buffer representations for software-managed storage units of H_{QKV} are represented in lines 2–4 of Fig. 6. The main scratchpad (d_1) consists of 128 rows (S_0=[128]) of 64-length bf16 vectors (S_1=[64]). Instructions read and/or modify the scratchpad at a row granularity.

(ii) TAIDL constructs: Instruction Set

The instruction set is a set of *abstract instructions* that modify the tensor buffers and are parameterized by instruction attributes. A *concrete instruction* is an instantiation of an abstract instruction with specific values for the attributes. For example, x86 instructions, such as mov and add, modify registers or memory and are parameterized by register names and memory addresses.

An *assembly code* for a tensor accelerator consists of a stream of concrete instructions and a constant tensor. This is analogous to x86 assembly code with code and data sections, respectively.

TAIDL models the semantics of abstract instructions of tensor accelerators as a composition of tensor operators in XLA-HLO IR [22]. Fig. 5 (b) provides a brief description of the instruction set of H_{QKV} using tensor operators from Table 1.

In this work, we extend TAIDL descriptions by classifying the instruction attributes into two sets: (i) *computational* ($\boldsymbol{\alpha}$) attributes, like input matrix shapes (challenge 2), that determine the core tensor computation; and rest of the attributes as (ii) *addressing* ($\boldsymbol{\beta}$) attributes, like input addresses. Therefore, an abstract instruction $\boldsymbol{\theta}$ which reads n_θ tensors located in buffers $d_\theta^{(i)}|_{i=1}^{n_\theta}$ and modifies a tensor located in buffer $d_\theta^{(0)}$, is represented using an *abstract tensor computation* $\boldsymbol{g_\theta}$ concretized by the set of computational attributes α, $(n_\theta + 1)$ *data slice address maps* $\boldsymbol{h_\theta^{(i)}}|_{i=0}^{n_\theta}$ over the set of addressing attributes β, and a *validity constraint* $\boldsymbol{e_\theta}$ over the set of attributes α and β.

Running Example. The abstract instruction load_rm in H_{QKV} is represented in lines 7–16 of Fig. 6. Lines 8–9 describe the tensor buffers $\boldsymbol{d_{\text{load\_rm}}}$ and data slice address maps $\boldsymbol{h_{\text{load\_rm}}}$ of the input and output tensors. It is a data movement instruction that loads n rows of data (n×128 bytes) from the HBM (d_0) starting at address addr_{in}, to the main scratchpad (d_1) starting at row addr_{out}. The concrete instruction load_rm(n=4, addr_{in}=0, addr_{out}=2) loads 512 bytes from $d_0[0{:}512]$ to $d_1[2{:}6]$. Line 10 describes the validity constraint $\boldsymbol{e_{\text{load\_rm}}} = (n \leq 128)$. Finally, lines 12–16 represent the abstract tensor computation $\boldsymbol{g_{\text{load\_rm}}}$ using tensor operators (Table 1) and attribute $\text{n} \in \boldsymbol{\alpha_{\text{load\_rm}}}$.

```
# Data Model
qkv.set_hbm("d0", size="1MB")                              # u8[2^20]
qkv.add_data_model("d1", S0=[128], S1=[64], E="bf16") # bf16[128,64]
qkv.add_data_model("d2", S0=[64], S1=[64], E="bf16")  # bf16[64,64]

# Instruction Semantics
instr = qkv.add_instr(name="load_rm", alpha=["n"], beta=["addr_in", "addr_out"])
instr.set_inputs([["d0", start=["addr_in"], len=["n * 128"]]])  # u8[n*128]
instr.set_output(["d1", start=["addr_out"], len=["n"]])         # bf16[n,64]
instr.set_constraints(["n <= 128"])
# y = bitcvt(reshape(x1)) represented using XLA-HLO syntax
instr.set_abstract_computation("""ENTRY load_rm {
  x1 = u8[`n * 128`] parameter(0);
  a = u8[`n`, 64, 2] reshape(x1);
  ROOT y = bf16[`n`, 64] bitcast_convert(a);
}""")
```

Fig. 6. Snippet of ISA description of H_{QKV} in TAIDL with instruction attributes annotated as $\boldsymbol{\alpha}$ and $\boldsymbol{\beta}$.

3.3 Problem Formulation: Compiler Backend Generator

Finally, we formulate the problem statement of a compiler backend generator that generates sound and complete compiler backends from just the accelerator ISA descriptions in TAIDL.

A compiler backend for a tensor accelerator consumes a tiled kernel (§3.1) as the input and either (i) successfully compiles to emit an *accelerator-specific assembly code* (§3.2), or (ii) does not compile, i.e., either a compilation failure or does not terminate. We define the semantic equivalence between the tiled kernel and the compiled assembly code using bisimulation transformations, modulo the set of known tensor rewrites of XLA-HLO IR (as discussed in §2.2).

Soundness: A compiler backend is said to be *sound* if, for all successfully compiled tiled kernels, the emitted assembly code is semantically equivalent to the input tiled kernel.

Completeness: A compiler backend is said to be *complete* if it successfully compiles for all input tiled kernels that have a semantically equivalent assembly code.

The detailed definitions and formalisms of these concepts are deferred to Appendix A.

> We design a compiler backend generator such that for any given ISA description in TAIDL, it generates an accelerator-specific compiler backend that is *sound* and *complete*.

4 Overview of Accelerator Compiler Toolkit (ACT)

We present a compiler backend generator — ACT, Accelerator Compiler Toolkit — that generates a sound and complete compiler backend from a given tensor accelerator ISA description in TAIDL. A key contribution of our work is a novel compilation algorithm parameterized by TAIDL constructs ($d_i, \theta, \alpha, \beta, g_\theta, h_\theta, e_\theta$, discussed in §3.2). In this section, we provide a high-level overview of the ISA-parameterized algorithm (in §4.1-§4.2) and outline the compiler backend generation process based on this algorithm (in §4.3). We elaborate the details in §5 and §6.

4.1 Overview of the ISA-parameterized Compilation Algorithm

ACT's parameterized compilation algorithm is designed as a modular pipeline, with each module focusing on a specific aspect of the compilation. The algorithms for each module are *parameterized* by the TAIDL constructs, which are instantiated given an ISA description.

Fig. 7 visualizes the high-level structure of ACT's ISA-parameterized compilation algorithm. The algorithm consumes a tiled kernel and emits an assembly code. The assembly code consists of concrete instructions $\theta_i(\alpha_i, \beta_i)|_{i=1}^{N}$, i.e., abstract instructions θ_i with initialized computational (α_i) and addressing (β_i) attributes. The algorithm consists of two phases: (1) initializing α_is, (i.e., *instruction selection*), and (2) initializing β_is, (i.e., *memory allocation*).

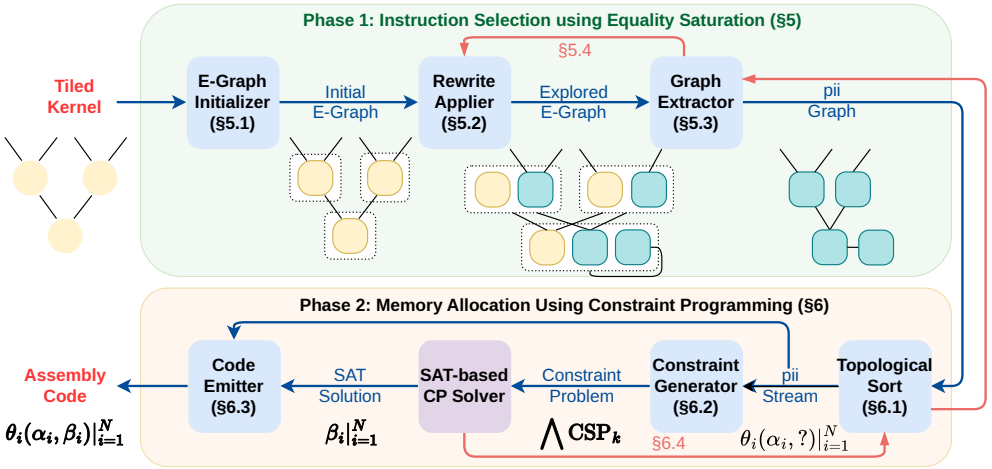


Fig. 7. Overview of ACT's parameterized compilation algorithm. Blue modules are parameterized by the TAIDL constructs (§3.2). The purple module is an external tool. The red edges represent the fallback mechanisms designed for completeness (proof in Appendix C) and the enumeration pipeline (discussed later in §7).

Phase 1: Tensor Instruction Selection

The first phase consumes the tiled kernel and emits a directed acyclic graph of *partially* instantiated instructions (pii), called a pii graph. A *partially* instantiated instruction is an abstract instruction θ with α initialized and β uninitialized, denoted as $\theta_{\alpha,?}$. This phase initializes α, i.e., explores semantically equivalent instruction selection opportunities for the tiled kernel. This is analogous to the instruction selection pass in CPU backends. It is formulated as an equality saturation problem and utilizes an existing equality saturation engine [99] to perform graph-level semantic-preserving substitutions. The three modules required for equality saturation—e-graph initializer, rewrite applier, and graph extractor—are parameterized by the TAIDL constructs.

Phase 2: Tensor Memory Allocation

The second phase consumes the pii graph and emits the final assembly code. This phase initializes β, i.e., assigns addresses to the unmapped slices. This is analogous to the register allocation pass in CPU backends. First, the pii graph is topologically sorted to respect instruction dependencies. Then, these slices are mapped to the address space of the accelerator's tensor buffers. This mapping must (1) maintain instruction dependencies, (2) assign non-overlapping address ranges to slices with overlapping live ranges, and (3) yield valid instruction attributes. These are formulated as a constraint satisfaction problem over integers and passed to an existing SAT-based CP solver [39]. Finally, the assembly code is emitted based on the solver's output. The three modules—topological sort, constraint generator, and code emitter—are also parameterized by the TAIDL constructs.

Inter-phase fallback mechanism

Unlike commodity hardware, accelerator designs have constraints not captured during instruction selection, such as non-trivial addressing constraints and lack of spill-reload instructions for some tensor buffers (e.g., AWS Trainium [81]). Therefore, the second phase may fail to find a valid solution for a pii graph. The compilation algorithm then falls back to the first phase, which extracts a different pii graph or performs another iteration of rewrites. This inter-phase mechanism, along with the intra-phase fallbacks, is essential for completeness.

Soundness. ACT's parameterized compilation algorithm is sound by construction (proof in Appendix C). Note that the soundness is defined modulo the foundational axioms of XLA-HLO IR (as discussed in §2.2). This axiomatic approach is similar to prior works [51, 96, 100, 104].

Completeness. ACT's parameterized compilation algorithm is also complete. Intuitively, this is shown by proving that for any input tiled kernel with an equivalent assembly code, there exists a sequence of rewrites and a valid ordering of the pii graph that leads to a successful compilation, and the fallback mechanisms ensure that all possible sequences of rewrites and orderings are explored and enumerable. The detailed proof of completeness is provided in Appendix C.

Termination. For kernels with no equivalent assembly code, the compilation algorithm may not terminate, since completeness and guaranteed termination cannot both hold (proven by reduction from the halting problem in Appendix A). In our context of ML compiler backends, completeness is crucial to ensure that the middle-end does not fall back to host (CPU) code generation, avoiding the associated performance degradation (§8.6). In practice, we bound non-termination with a timeout.

4.2 Novelty of the ISA-parameterized Compilation Algorithm

Code generation for tensor accelerators brings unique challenges (§1.2) that are not articulated nor solved by existing techniques. Off-the-shelf rewrite synthesizers, e-graph extractors, and memory/register allocators do not handle parameterized tensor instructions, multi-dimensional scratchpad allocation, or complex layout-transforming ISA semantics. Table 2 summarizes the novel features of ACT's parameterized compilation algorithm that address these challenges.

Limitations of existing techniques	Novelty in ACT
Vector-ISA instruction selectors [18, 90, 93] assume fixed-size instructions, unlike shape-parameterized tensor instructions.	*Reduction to a two-phase algorithm* (§4.1) via a new pii intermediary, splitting attributes into computational (α, Phase 1) and addressing (β, Phase 2).
Typical rewrite synthesizers [71, 72] generate rewrites *without preconditions*, and cannot model parameterized instructions with computational attributes.	*Preconditioned rewrite synthesis* (§5.2) that generates one IR-to-ISA rewrite per instruction, with *preconditions* on computational attributes for parameterized instructions.
Existing accelerator backends and libraries do not ensure *completeness* and may fail to compile kernels even when a valid assembly code exists, forcing host (CPU) fallback.	*No-ops modeled as identity pii nodes* (§5.2) for inner-loop tiling and padding, enabling provable discovery of *all* equivalent instruction sequences and ensuring completeness.
One-shot extractors [11, 98–100] extract a *single* graph, freely choosing at most one e-node per e-class, unable to handle fallback scenarios or e-node selection constraints.	*Enumerative e-graph extraction* (§5.3) of *all* equivalent pii graphs for fallback scenarios, restricting e-nodes by previously selected ones and allowing *multiple* e-nodes per e-class.
Register allocators (graph coloring [3], puzzles [79]) and scratchpad allocators [64, 66] model only interference over 1-D sites, without accelerator addressing constraints.	*CSP-based memory allocation* (§6.2) combining accelerator addressing constraints with live-range interference over multi-dimensional tensor buffers and related pruning strategies (§6.4).

Table 2. Novel features of ACT's parameterized compilation algorithm.

4.3 Overview of the Compiler Backend Generation Process

ACT's ISA-parameterized compilation algorithm is implemented as a generic compiler backend with pure virtual functions and symbolic templates based on TAIDL constructs. A user specifies an accelerator ISA in TAIDL (as shown in Fig. 6). ACT processes this ISA description to instantiate a specialized compiler backend by filling in the symbolic templates with ISA-specific details. This generation process is visualized in Fig. 8. This is similar to the design principle of classic compiler component generators such as Flex [30] (for lexical analysis) and Yacc [52] / Bison [78] (for parsing).

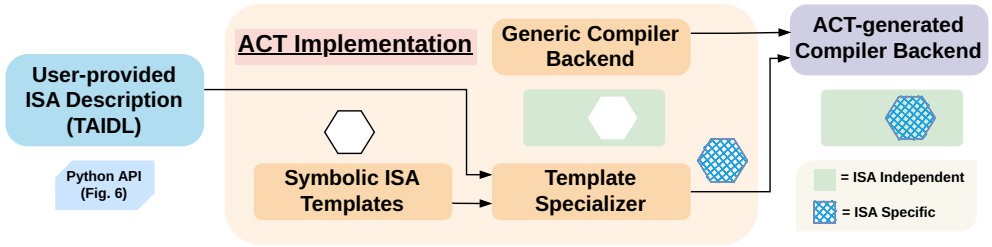


Fig. 8. Overview of ACT's compiler backend generation process. The orange region shows the implementation of the parameterized algorithm as a generic compiler backend (with missing holes) and symbolic ISA templates. At generation time, these symbolic templates are specialized using the user-provided ISA description.

The generation process runs once per accelerator and produces a standalone accelerator-specific compiler backend executable. At compile-time, this generated backend executable is called once per tiled kernel. Detailed correspondence with Flex and Bison is tabulated in Appendix B.

More concretely, ACT converts the TAIDL description into rewrite rules and preconditions for equality saturation (Phase 1). It also generates ISA-specific implementations of abstract methods such as `get_h0()` and `get_e()`, corresponding to the TAIDL constructs $h_\theta^{(0)}$ and e_θ. The constraint problem (Phase 2) is formulated abstractly on the TAIDL constructs, without any hardcoded ISA assumptions. This separation ensures that compiler backends are derived systematically from just the ISA description, without embedding ISA-specific logic in the compiler.

Engineering Effort per Accelerator. Existing TAIDL descriptions take around 10-12 lines per simple instruction (like Intel AMX [46] load/store, Intel AVX-512 [47] vector compute, Gemmini [36]) and 12-15 lines per complex one (like Intel AMX [46] matrix compute, FEATHER [92] layout reconfigurations), so generating a new accelerator-specific backend takes little effort. These backends are also compatible with XLA, a production ML compiler, amplifying this reduction.

5 Phase 1: Tensor Instruction Selection Using Equality Saturation

We formulate the tensor instruction selection problem as a search problem over the space of equivalent tensor computation graphs. We use e-graphs [89] to compactly represent this space of equivalent tensor computation graphs and enable efficient exploration for instruction selection.

In this section, we focus on the novel features of Phase 1 highlighted in Table 2 — preconditioned IR-to-ISA rewrites and identity instructions (§5.2), as well as an enumerative e-graph extraction algorithm (§5.3-§5.4). We use the step-wise lowering of the tiled kernel G_{QKV} (Fig. 4) for the accelerator H_{QKV} (Fig. 5) as an illustrative example.

5.1 Module 1: E-Graph Initializer

In our work, an e-graph is a directed graph where each e-node is a concrete tensor operator or a *partially instantiated instruction* (pii $\theta_{\alpha,?}$, defined in §4.1). An e-class is a set of e-nodes representing semantically equivalent computations rooted at the e-nodes. By design, an e-node is optionally associated with a tensor buffer. Specifically, a pii e-node $\theta_{\alpha,?}$ is associated with the output tensor buffer of its instruction ($d_\theta^{(0)}$ notation from §3.2). The e-graph is initialized with the input tiled kernel G with a 1-D byte representation of the input and output tensors. The 1-D byte representation for the bf16 tensors requires

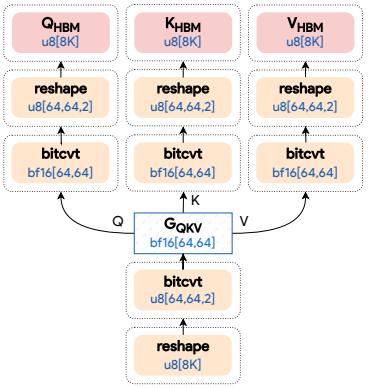


Fig. 9. The initial e-graph for G_{QKV} [after Module 1]. G_{QKV} (Fig. 4) is not expanded here for compactness. Note that the edges are now flipped with the output tensor as the root e-node.

bitcasting a bf16 to two u8s followed by a reshape operation. The leaf e-nodes are assigned to the tensor buffer d_0, i.e., allocated on HBM, while the root is the output tensor, where extraction begins (§5.3). Fig. 9 shows the initial e-graph for G_{QKV} (Fig. 4), without expanding the sub-graph of G_{QKV}.

5.2 Module 2: Rewrite Applier

The second module specifies the rewrites used in the exploration stage of equality saturation. These include three types of rewrites: (1) existing set of IR-to-IR tensor rewrites from the XLA compiler, (2) new IR-to-ISA rewrites generated from the accelerator ISA description, and (3) rewrites based on *no-ops*, termed as *identity instructions*, i.e., instructions that do not modify the memory state.

IR-to-IR rewrites from XLA compiler. ACT-generated backends use the curated set of existing IR-to-IR rewrites from the XLA compiler (discussed in §2.2) to explore the space of equivalent tensor computation graphs. Note that these 175+ rewrites are maintained by the XLA compiler developers and are not specific to any accelerator. Furthermore, e-graphs can simultaneously explore different sequences of rewrites, avoiding the need to implement accelerator-specific IR legalization passes.

IR-to-ISA rewrites generated from ISA description. At generation time, ACT generates new rewrites with preconditions using a pre-order traversal from the ROOT of the abstract tensor computation g_θ (line 15 of Fig. 6). These IR-to-ISA rewrites create new pii nodes in the e-graph. Combined with IR-to-IR rewrites, they replace manually-written accelerator-specific pattern-matching rules.

For instruction θ, ACT generates the rewrite $g_\theta \rightarrow \theta$ that substitutes a concrete tensor computation $g_\theta(\alpha)$ with pii node $\theta_{\alpha,?}$ under the precondition $\exists \beta, e_\theta(\alpha, \beta) = True$. For example, rewrite bitcvt(reshape(x))→load_rm(x) is generated from Fig. 6. Fig. 10 shows a snippet of the explored e-graph of Fig. 9 with new pii nodes colored as blue (d_1) and green (d_2). Fig. 11 shows the precondition generated for load_rm, which determines the computational attribute n at compile-time and checks the validity constraint $e_{\text{load\_rm}}$. Most existing rewrite synthesizers [71, 72] generate rewrites without such preconditions, requiring multiple rewrites to cover a parameterized instruction, whereas ACT generates only *one* IR-to-ISA rewrite per instruction.

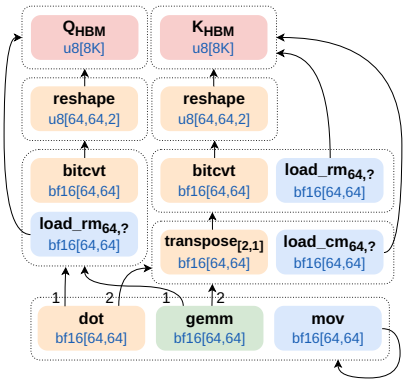


Fig. 10. Snippet of explored e-graph [after Module 2] after applying IR-to-ISA rewrites on the initial e-graph (Fig. 9). Dotted boxes are e-classes, solid boxes are e-nodes, and edge labels 1/2 are operand indices. The new pii nodes result from the ACT-generated IR-to-ISA rewrites for load_rm, load_cm, gemm, and mov.

Identity instructions (no-ops). We model *no-ops* as identity instructions slice^H and concat^H, i.e., instructions that do not modify the tensor buffers. These are pii nodes analogous to tensor operators slice and concat but with the constraint that the inputs and the output are on the same buffer. slice^H applies only under a precondition (the slice dimensionality must match a tensor buffer, §3.2), and concat^H constrains its two operands to be allocated contiguously in the same tensor buffer so the next instruction reads them as one operand. These identity instructions ensure exploration of all equivalent pii graphs (proof in Appendix C).

```
1 pub fn precond_load_rm(..) -> bool {
2   // y: bf16[`@alpha.n`, 64]
3   if y.shape.len() != 2
4      || y.shape[1] != 64
5      || y.dtype != Dtype::BF16 {
6     return false;
7   }
8   let alpha_n = y.shape[0];
9   if alpha_n > 128 {
10    return false;
11  }
12  ... // Repeat for a, x1
13 }
```

Fig. 11. Snippet of the rewrite precondition (Rust) generated from the semantics of load_rm in Fig. 6. Line 8 determines the computational attribute n.

5.3 Module 3: Graph Extractor

The third module performs the extraction stage of the equality saturation workflow. It extracts a directed acyclic graph of pii nodes (i.e., a pii graph) from the explored e-graph (after Module 2).

First, a bottom-up traversal (from leaf e-classes) is performed to detect e-classes representing compile-time constant tensors. The e-classes with a leaf e-node, such as tensor operators *eye*, *iota*, are constants. An e-class is a constant if any one of its e-nodes has all its operands as constants. Such e-classes become base cases of the extraction below (line 6 of Fig. 12).

Then, a top-down traversal (from root e-class) is performed to extract the pii graphs. It is guided by the input and output tensor buffers ($d_\theta^{(i)}|_{i=0}^{n_\theta}$ notation from §3.2) associated with each pii e-node. Fig. 12 presents the pseudocode — extract(EC, d, N) returns *all* pii graphs computing e-class EC into tensor buffer d within N nodes, invoked at the root with $d = d_0$ (line 24). Only the e-nodes whose output tensor buffer $d_\theta^{(0)}$ matches the requested d are admissible (line 12), and each of them in turn requests $d_\theta^{(i)}$ for its i-th operand (lines 15–16). For example, load_cm$_{64,?}$ is admissible for the buffer that gemm requests for its second operand in Fig. 10, since $d^{(0)}_{\text{load\_cm}} = d_1 = d^{(2)}_{\text{gemm}}$. The graphs of an admissible e-node are the Cartesian product of its operands' graph sets (line 19), and those of an e-class are the union over its admissible e-nodes. The traversal terminates upon reaching an input variable leaf e-node or a constant e-class, or hitting a pii node limit $N_{\max}$ (line 3).

```
1  def extract(ec, buf, limit):
2    # node budget N_max exhausted
3    if limit == 0: return {}
4
5    # base cases: input or constant
6    if ec.is_input or ec.is_const:
7      return {leaf(ec)}
8
9    graphs = {}
10   for e in ec.enodes:
11     # buffer-guided selection
12     if e.out_buf != buf: continue
13     children = []
14     for i, c in enumerate(e.args):
15       cbuf = e.in_buf[i]
16       sub = extract(c, cbuf, limit-1)
17       children.append(sub)
18     # enumerate all combinations
19     for combo in product(*children):
20       graphs.add(pii(e, combo))
21   return graphs
22
23 # Module 3 entry point
24 extract(egraph.root, HBM, N_max)
```

Fig. 12. Top-down traversal to extract all pii graphs following buffer constraints within the $N_{\max}$ node limit.

Classical one-shot extractors [11, 98–100] freely select at most one e-node per e-class without any e-node constraints. Our extraction differs on both counts — selection is constrained by the buffer requested by the parent e-node, and an e-class expanded under several requested buffers contributes *multiple* of its e-nodes to one pii graph. Enumerating *all* of them is necessary rather than merely thorough, since Phase 2 can fail for a given pii graph (§4.1) and its cost is known only after code generation (§7). Without our novel extraction algorithm, most kernels (> 95%) fail to compile for popular tensor accelerator ISAs such as AWS NKI [83] and Gemmini [36].

Fig. 13 (a) shows the first extracted pii graph from the explored e-graph of G_{QKV} (Fig. 10).

5.4 Intra-phase Fallback Mechanism: Enumerating All Equivalent pii Graphs

The exploration and extraction stages of equality saturation may not terminate for all tiled kernels. The exploration stage iteratively applies the rewrites but may not reach saturation for all tiled kernels. The explored e-graph may have loops, and infinite equivalent pii graphs may exist.

To ensure that all equivalent pii graphs are enumerable without getting stuck within an infinite loop of either of the two stages, we use a bounded approach of exploration and extraction. We iteratively increase the pii node limit $N_{\max}$ for the extraction stage with the iteration count k of the exploration stage, i.e., $N_{\max} = \Gamma(k)$ where Γ is a strictly monotonically increasing function of k (e.g., $\Gamma(k) = 2^k$). For every value of k, we extract a finite number of pii graphs, and thus, this set is enumerable. This is analogous to enumerating $(a, b) \in \mathbb{N}^2$ using the Cantor pairing function [13].

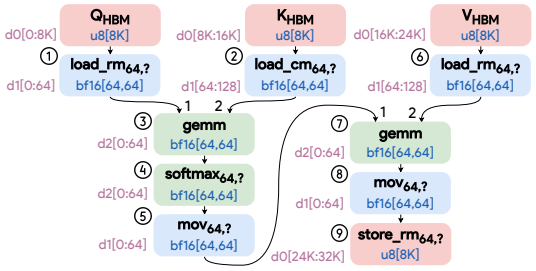
6 Phase 2: Tensor Memory Allocation using Integer Constraint Programming

The second phase of the compilation pipeline is responsible for compiling the generated pii graph down to accelerator-specific assembly code. This phase involves scheduling the pii graph and allocating the intermediate tensors on the tensor buffers. We formulate the memory allocation as an integer constraint satisfaction problem (CSP) and solve it using a SAT-based CP solver.

In this section, we focus on the novel features of Phase 2 highlighted in Table 2 — the parameterized multi-dimensional CSP formulation (§6.2) and interference-based pruning (§6.4).

6.1 Module 4: Topological Sort

Instruction scheduling needs to adhere to *def-use* edges of the pii graph and perform a topological ordering of the nodes. Our priority is to minimize the live ranges of the intermediate tensors and reduce the possibility of memory allocation failing. Sethi-Ullman algorithm [86] generates optimal code for arithmetic binary trees with minimal registers. We extend the Sethi-Ullman numbering to multiple tensor buffers storing varying sizes of tensor variables, and order the operands of each pii node by decreasing tensor buffer requirement. We defer the extended numbering to Appendix B. Fig. 13 (a) shows the topological ordering of the extracted pii graph annotated with circled numbers.



(a)

load_rm(n = 64, $addr_{in}$ = 0, $addr_{out}$ = 0)
load_cm(n = 64, $addr_{in}$ = 8192, $addr_{out}$ = 64)
gemm($addr_1$ = 0, $addr_2$ = 64, $addr_{out}$ = 0)
softmax(n = 64, addr = 0)
mov(n = 64, $addr_{in}$ = 0, $addr_{out}$ = 0)
load_rm(n = 64, $addr_{in}$ = 16384, $addr_{out}$ = 64)
gemm($addr_1$ = 0, $addr_2$ = 64, $addr_{out}$ = 0)
mov(n = 64, $addr_{in}$ = 0, $addr_{out}$ = 0)
store_rm(n = 64, $addr_{in}$ = 0, $addr_{out}$ = 24576)

(b)

Fig. 13. (a) Extracted pii graph annotated with topological ordering (circled) [after Module 4] and assigned tensor data slices (purple) [after Module 5]. (b) The compiled assembly code for G_{QKV} [after Module 6]. Input tensors Q, K, V occupy relative HBM addresses 0, 8K, 16K (as determined by the XLA middle-end).

6.2 Module 5: Constraint Satisfaction Problem Generator

By this stage, we have a pii graph and a topological ordering of pii nodes $\theta^{(i)}_{\alpha^{(i)},?}|_{i=1}^{N}$. Next step is to instantiate the addressing attributes $\beta^{(i)}$ of the pii nodes, i.e., allocate the intermediate and constant tensors on the tensor buffers. We formulate a constraint satisfaction problem (CSP) by computing the interference graph and incorporating addressing constraints of the tensor buffers.

The CSP is parameterized by TAIDL constructs (from §3.2) and consists of four components: (1) instruction validity constraints (e_θ), (2) def-use edge constraints over the data slice addresses from the address map h_θ, (3) HBM addressing constraints for the input and output tensors in the input tiled kernel G, whose addresses the XLA middle-end pre-allocates, and (4) live range interference constraints. The final CSP formulation is summarized in Fig. 14.

The final CSP is passed to an existing SAT-based CP solver library [39] to find a satisfying solution over the addressing attributes $\beta^{(i)}$. Fig. 13 (a) shows the data slice addresses assigned to pii nodes as determined by solving the constraint problem.

Implementation of Module 5. Let's look at a concrete example of template-driven implementation (discussed in §4.3) of the *parameterized* algorithm of Module 5. It consists of a generic skeleton (Fig. 15) with symbolic templates (Fig. 16) specialized at generation-time (Fig. 17). At compile time, the CSP builder calls these specialized functions at each pii node.

$$\text{CSP}_1 = \bigwedge_{i=1}^{N} \left(e_{\theta^{(i)}}(\alpha^{(i)}, \beta^{(i)}) \wedge (I_s^{(i)}, I_e^{(i)}) = h^{(0)}_{\theta^{(i)}}(\beta^{(i)}) \right)$$

$$\text{CSP}_2 = \bigwedge_{(V^{(y)}, V^{(x_i)})} (I_s^{(x_i)}, I_e^{(x_i)}) = h^{(i)}_{\theta^{(y)}}(\beta^{(y)})$$

$$\text{CSP}_3 = \left(\bigwedge_{i=1}^{N_V} (I_s^{(N+i)}, I_e^{(N+i)}) = (I_s^{(G_{x_i})}, I_e^{(G_{x_i})}) \right) \wedge (I_s^{(N)}, I_e^{(N)}) = (I_s^{(G_y)}, I_e^{(G_y)})$$

$$\text{live\_range}(i) = \begin{cases} [0, N], & V^{(i)} \text{ is an input variable node,} \\ [0, b_i - 1], & V^{(i)} \text{ is a constant leaf node,} \\ [a_i, b_i - 1], & V^{(i)} \text{ is an intermediate node} \end{cases}$$

$$\phi = \{(i, j) \mid d^{(0)}_{\theta^{(i)}} = d^{(0)}_{\theta^{(j)}} \ \wedge \ \text{live\_range}(i) \cap \text{live\_range}(j) \neq \emptyset\}$$

$$\text{CSP}_4 = \bigwedge_{(i,j) \in \phi} \text{disjoint}(i, j)$$

$$\text{CSP} = \text{CSP}_1 \wedge \text{CSP}_2 \wedge \text{CSP}_3 \wedge \text{CSP}_4$$

Fig. 14. Summary of the CSP formulation used in Module 5. Notations used: N is the number of pii nodes, N_V is the number of input variables in the tiled kernel G, $V^{(i)}$ denotes node i, $e_{\theta^{(i)}}$ is the instruction validity constraint, $h_{\theta^{(i)}}$ denotes the data slice address map, $(I_s^{(i)}, I_e^{(i)})$ are data slice addresses, a_i is the topological number of node i, b_i is the maximum topological number among uses of node i, $d^{(0)}_{\theta^{(i)}}$ is the output tensor buffer of node i, and disjoint(i, j) enforces non-overlapping slice addresses for nodes i and j.

```
void CSP1(const std::vector<INSTR*>& instructions) {
  for (const auto& instr : instructions) {
    solver.AddConstraint(instr->get_e());
    solver.MakeEquality(instr->output->start_addr, instr->get_h0().first);
    solver.MakeEquality(instr->output->end_addr, instr->get_h0().second);
  }
}
```

Fig. 15. Snippet of Generic implementation of Module 5 using abstract C++ class INSTR for ISA construct θ with pure virtual functions get_h0() and get_e() representing ISA constructs $h^{(0)}_{\theta}$ and e_{θ} respectively,

```
class INSTR_<<str instr.name>> : protected INSTR { ... }
std::pair<IntExpr*, IntExpr*> INSTR_<<str instr.name>>::get_h0() override {
  return {<<ortools::IntExpr* instr.output.start>>, MAKE_SUM(<<ortools::IntExpr*
    instr.output.start>>, <<int64 instr.output.len>>)};
}
```

Fig. 16. Symbolic Template for ISA construct $h^{(0)}_{\theta}$.

```
class INSTR_load_rm : protected INSTR { ... }
std::pair<IntExpr*, IntExpr*> INSTR_load_rm::get_h0() override {
  return {addr_out, MAKE_SUM(addr_out, n)};
}
```

Fig. 17. Specialized Template for $h^{(0)}_{\text{load\_rm}}$ generated by processing line 9 of Fig. 6.

6.3 Module 6: Code Emitter

The final module sequentially emits the sorted pii nodes with their solved addressing attributes and discarding identity instructions (no-ops). Fig. 13 (b) shows the emitted assembly code for G_{QKV}.

6.4 Intra-phase Fallback Mechanism

The depth-first topological ordering generated by the heuristic-based algorithm (§6.1) is not sufficient for the completeness of Phase 2. We prove this using a counterexample in Appendix B. The tensor accelerator instructions and the identity instructions (slice^H & concat^H) often have stricter constraints (e_θ) on the data slice addresses than what classical register allocation algorithms support. Theoretically, we need to search through all topological orderings for completeness.

Pruning technique. We observe that CSP_4 is monotonic w.r.t. the interference graph, i.e., $\phi_1 \subseteq \phi_2 \implies \neg\text{CSP}_4(\phi_1) \rightarrow \neg\text{CSP}_4(\phi_2) \implies \neg\text{CSP}(\phi_1) \rightarrow \neg\text{CSP}(\phi_2)$. Therefore, we discard any topological ordering with a sub-interference graph of a previously failed topological ordering. This reduces the exponential search space of topological orderings to a small number of unique interference graphs, which is tractable to search through if the heuristic-based ordering fails. Exhaustively searching these topological orderings guarantees the completeness of Phase 2.

7 Code Generation with Cost Models

In the previous sections, we have discussed the core algorithm of ACT with fallback mechanisms that guarantee completeness, i.e., find an equivalent assembly code if one exists. We observe that the first equivalent assembly code may not be performant enough. Therefore, we designed a code generator with a simple enumeration pipeline using cost models to generate performant code.

```
1  def codegen(G, n=2):
2    # (asm, cost, enumeration time)
3    final = (None, inf, 0)
4    for asm in act_enum(G):
5      # Compare candidate asm
6      time = now()
7      cost = cost_model(asm)
8      if cost < final.cost:
9        # Select candidate asm
10       final = (asm, cost, time)
11     elif time > n * final.time:
12       # Termination heuristic
13       return final
```

(a) Candidate selection using performance cost model

```
1  def act_enum(G):
2    egraph = init(G);              # Module1
3    while not egraph.saturated:    # §5.4
4      egraph = egraph.applier(k)   # Module2
5      piis = egraph.extract(Γ(k)) # Module3
6      for pii in piis:             # §4.1
7        schedules = topo(pii);     # Module4
8        for schd in schedules:     # §6.4
9          csp = csp_gen(schd);     # Module5
10         sol = solve(csp)         # CSP Solver
11         if sol.SAT:
12           asm = emit(pii, sol)   # Module6
13           yield asm              # Enumerate
```

(b) Candidate enumeration function using ACT-generated compiler backend

Fig. 18. Pseudocode for code generator using cost models to generate performant code

The code generator (Fig. 18) uses ACT-generated compiler backend as an enumerator to generate multiple equivalent assembly candidates. A simple candidate selection loop iterates through these candidates and selects the final assembly code with the least cost using a performance cost model.

Candidate enumeration. Fig. 18 (b) shows the pseudocode of the enumerator function based on the ACT algorithm (described in Fig. 7) with yield statement[1] at line 13. The fallback mechanisms (lines 3, 6, 8) give control back to the previous module. This allows the compiler backend to continue after a successful compilation and repeatedly *yield* equivalent assembly code.

Candidate selection. Fig. 18 (a) shows the code generator that iterates through the candidate assembly codes (line 4) enumerated by the ACT-generated compiler backend (Fig. 18 (b)). Performance statistics of these candidates are collected (line 7), and the candidate with the best performance (least execution cost) is selected (line 8) as the final assembly code.

Termination heuristic. We design a termination heuristic that stops the enumeration if the performance of the enumerated candidates doesn't improve for a considerable time (Fig. 18 (a) line 11). Such a heuristic is required since there are, theoretically, infinite equivalent assembly code candidates with redundant load-store pairs. However, redundant data movement makes such

[1] https://docs.python.org/3/reference/expressions.html#yield-expressions

candidates longer than ideal and less performant. The first phase prioritizes shorter assembly codes (§5.4), and thus, such candidates are likely to be enumerated after their ideal counterparts. Therefore, we expect the performance to saturate after the ideal candidates are enumerated. In §8.5, we show that the code generators match the performance of state-of-the-art kernel libraries like the oneDNN library [49], with the observed compilation time less than 100 ms.

Choice of Cost Model. Performance statistics can be collected using different fidelity cost models, including cycle-accurate simulators [87], analytical models [73], and learned cost models [65]. Note that the enumeration pipeline and the ACT algorithm are *agnostic to the choice of cost model.*

8 Evaluation

We performed three sets of evaluations to demonstrate the effectiveness of ACT.

- **Supported Accelerators (§8.2):** We generated seven compiler backends for diverse tensor accelerator designs from industry and academia, demonstrating generality of our approach.
- **Comparative Evaluations (§8.3–§8.6):** We evaluated (a) the compilation coverage of ACT-generated compiler backends and (b) performance of their compiled code, against existing accelerator-specific backends and expert-written kernel libraries for tensor accelerators.
- **Compilation Time Analysis (§8.7):** We performed breakdown analyses of the compilation time for varying tiled kernel sizes from synthetic and real-world workloads.

8.1 Implementation

We used egg [99] for equality saturation (§5) and Google OR-Tools [39] for solving CSPs (§6). ACT is implemented as a generic compiler backend (1.9 KLOC of C++ & 6.7 KLOC of Rust), symbolic ISA templates (64 LOC of C++ & 68 LOC of Rust), and a template specializer (1.7 KLOC of Python). The ISA descriptions are parsed by the ANTLR-based TAIDL front-end [50], which we reuse.

8.2 Supported Accelerators

TAIDL [50] provides ISA descriptions for diverse accelerator designs from industry [46, 55, 81] and academia [36, 55, 92]. We extend these TAIDL descriptions by annotating the instruction attributes as computational (α) or addressing (β) — a one-line change to the 10–15 lines of the original TAIDL descriptions for each instruction (e.g., line 7 of Fig. 6). Including the running example H_{QKV} implemented using Allo [14], we have a total of **seven** accelerators tabulated in Table 3.

We use ACT to generate compiler backends for these seven accelerators, showing *generality* of our approach. Since ACT's compilation algorithm is parameterized by TAIDL constructs, it handles diverse design features of these accelerators with no changes to the core compilation algorithm.

Tensor Accelerator	Existing Backend	Existing Library	ACT Backend
AWS Trainium [81]	✓	✓	✓
Intel AMX [46]	✓	✓	✓
Google TPUv1 [55]	✗	✗	✓
Gemmini [36]	✗	✓	✓
FEATHER [92]	✗	✗	✓
EVA [103]	✗	✗	✓
H_{QKV}	✗	✗	✓

Table 3. The seven tensor accelerators supported by ACT. ✗ refers to the absence of publicly available accelerator-specific compiler backends or expert-written kernel libraries.

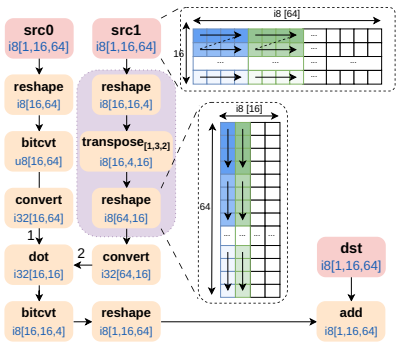


Fig. 19. Highlighted region shows the layout transformation applied to tile register src1 in Intel AMX instruction tdpbusd.

We list the notable design features below.

(1) **AWS Trainium** [81] includes two on-chip scratchpads with 2-dimensional memory [85], organized in 128 partitions. AWS provides a virtual ISA, AWS NKI [83], to program Trainium.
(2) **Intel AMX** [46] includes 2-dimensional registers (tiles) and supports instructions with complex layout transformations like tdpbusd (visualized in Fig. 19), which are semantically modeled in TAIDL as a series of reshape and transpose operators (purple region).
(3) **Google TPUv1** [55] supports instructions with variable-sized inputs (B×256).
(4) **Gemmini** [36] supports instructions to reconfigure dataflow (output vs weight-stationary).
(5) **FEATHER** [92] supports instructions to reconfigure input data layouts.
(6) **EVA** [103] is a CGRA-based accelerator with evolvable ISA and 3-dimensional addressing.
(7) H_{QKV} is implemented using an accelerator design language (ADL), Allo [14].

Execution of Compiled Assembly Code

Revisiting §3.1, we discussed that ACT-generated compiler backends emit code in the form of a sequence of TAIDL instructions, which are then stitched together with control flow for tiling into a single *stitched* IR. This stitched IR includes MLIR dialects [59] such as `scf` and `arith`.

Depending on the host-accelerator interface, the stitched IR is compiled and deployed differently.

(i) For built-in accelerators such as Intel AMX [46] and Gemmini [36], the stitched IR is compiled into an executable using the host CPU's code generation (e.g., x86 backend). In this case, the `scf.for` construct is compiled to the host's control flow instructions (e.g., branch and jump instructions).

(ii) For dedicated accelerators such as AWS Trainium [81] and FEATHER [92], the stitched IR is translated to the accelerator's kernel programming interface (e.g., AWS Neuron Kernel Interface (NKI) [83] for AWS Trainium [81]). In this case, the `scf.for` construct is directly mapped to the kernel programming interface's control flow constructs (e.g., `for` loop construct in AWS NKI [84]).

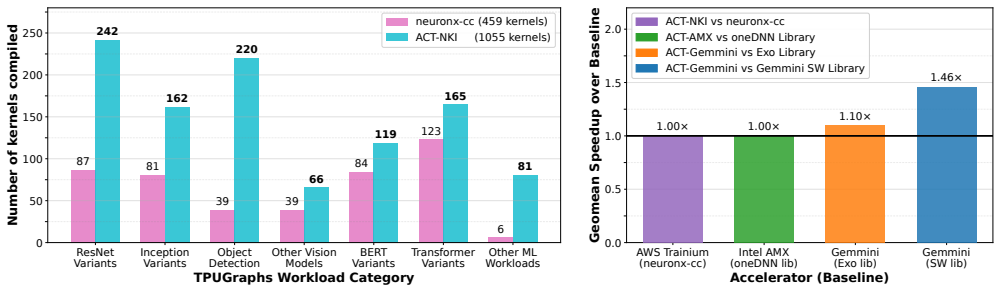
This mapping is automatically inferred from the TAIDL description of the accelerator's loop construct. The stitching implementation is otherwise generic across accelerators.

Note that regardless of the host-accelerator interface, ACT-generated compiler backends focus on tiled kernels in XLA-HLO IR and emit assembly code as per the ISA description in TAIDL.

8.3 Comparative Evaluations

Out of the seven tensor accelerators supported by ACT, only three have a publicly available accelerator-specific compiler backend or expert-written kernel library for comparative evaluations.

- **AWS Trainium (§8.4):** ACT-generated backend (ACT-NKI) vs. `neuronx-cc` [82] (backend)
- **Intel AMX (§8.5):** ACT-generated backend (ACT-AMX) vs. oneDNN library [49]
- **Gemmini (§8.6):** ACT-generated backend (ACT-Gemmini) vs. Exo [45] and SW library [37]



(a) Compilation Coverage Analysis for AWS NKI **(b) Performance Evaluation of Compiled Code**

Fig. 20. Summary of comparative evaluations. (a) Number of TPUGraphs kernels correctly compiled by ACT-NKI and neuronx-cc, grouped by workload category. (b) Geomean speedup of the compiled code by ACT-generated compiler backends against their corresponding baselines.

Here, we present the notable results with detailed analyses for each accelerator in §8.4–§8.6.

(a) Compilation Coverage

Fig. 20 (a) shows the number of tiled kernels in TPUGraphs [75] Tile dataset that can be compiled to AWS NKI by ACT-NKI and AWS's compiler `neuronx-cc`. More details are in §8.4.

$$\text{Compilation Coverage} = \frac{\text{Number of kernels that (i) are successfully compiled to AWS NKI and (ii) the compiled NKI code is functionally correct (randomized testing)}}{\text{Total number of tiled kernels in the benchmark suite (= 1055)}}$$

TPUGraphs kernels are only compatible with AWS Trainium since Google TPUv3 (the original target for TPUGraphs) and AWS Trainium have similar compute capabilities and the same tiling factor (128). Such an extensive dataset of tiled kernels doesn't exist for Intel AMX and Gemmini. Thus, the compilation coverage study is limited to AWS Trainium.

ACT-NKI achieves **2.3x** better compilation coverage for AWS NKI than `neuronx-cc`.

(b) Performance of Compiled Code

Fig. 20 (b) summarizes the geomean speedup of the assembly code compiled by ACT-generated compiler backends against their respective baselines [37, 45, 49, 82]. More details are in §8.4–§8.6.

ACT-generated compiler backends generate performant assembly code that matches or outperforms commercial compiler backends and expert-written kernel libraries.

8.4 Comparative Evaluations: AWS Trainium [81]

We evaluate ACT-NKI against AWS's production compiler `neuronx-cc`.

Benchmarks

The TPUGraphs [75] Tile dataset is a benchmark suite of tiled kernels generated by Google's TPUv3 compiler from real-world ML workloads.

Evaluation Methodology

We use AWS's compiler `neuronx-cc` [82] with `-print-nki` flag to emit AWS NKI code for the benchmark kernels. We evaluate correctness and performance on an AWS EC2 instance of `trn1`.

Results

Fig. 20 (a) shows that ACT-NKI achieves **2.3x** better compilation coverage than `neuronx-cc`. Since ACT-NKI has the completeness property, it correctly compiles all 1055 benchmark kernels.

For kernels compiled by both, we observe identical performance, i.e., speedup of **1x**. Note that achieving identical performance is a positive result in itself since `neuronx-cc` is a production compiler, whereas ACT-NKI is automatically generated by ACT from just the TAIDL description.

Observations & Analysis

The three main reasons for the difference in compilation coverage are:

(i) ACT-NKI transforms operators not directly supported in AWS NKI like `shift-left` using IR-to-IR rewrites ($\mathcal{R}$ in §2.1) during equality-saturation-based instruction selection (§5.2).
(ii) `neuronx-cc` doesn't support certain single-node tiled kernels, leading to empty NKI code.
(iii) ACT-NKI detects and allocates compile-time constant tensors (§5.3), unlike `neuronx-cc`, which fails to initialize constants, leading to unsound assembly code (fails correctness tests).

We reported these results to the AWS Trainium team, who have acknowledged these issues with `neuronx-cc`. Furthermore, AWS has adopted our tooling, showing the practical impact of our work.

8.5 Comparative Evaluations: Intel AMX [46]

We evaluate ACT-AMX against the Intel oneDNN library, the state-of-the-art kernel library used by production tensor compilers like XLA [20] and TorchInductor [4] to compile to Intel AMX.

Benchmarks

The oneDNN library [49] consists of common DNN kernels heavily optimized for Intel x86 ISA, including AMX. We selected five tiled kernels commonly observed in oneDNN examples [48], also used in AMX-related evaluations in [50]. These kernels are broadly categorized into three types: (i) only int8 matrix operations (`cnn_inf_amx` and `rnn_inf_amx`), (ii) only fp32 matrix operations (`mem_format_avx` and `sgemm_avx`), and (iii) a mixture of int8 and fp32 matrix operations (`cnn_inf_mix`). Detailed statistics and visualization for these kernels are present in Appendix H.

ISA Description

The TAIDL description we used to generate ACT-AMX included 4 AMX-INT8 instructions and 24 AVX512-FP32 instructions. Thus, it supports compilation to both AMX and AVX512.

Evaluation Methodology

The enumeration pipeline (§7) of ACT-AMX uses a simple analytical cost model based on individual instruction costs. For fair comparison, ACT-AMX uses the same fixed tile size as the oneDNN library. We evaluate correctness and performance on an Intel Xeon Gold 5415+ CPU.

Results

Final assembly codes by ACT-AMX *match* the performance of oneDNN library, i.e., speedup of **1x**. The int8 computations are mapped to AMX instructions by both ACT-AMX and oneDNN.

Observations & Analysis

The hand-optimized oneDNN assembly code is one of the candidates enumerated by ACT-AMX, by virtue of ACT's completeness guarantee. This candidate has the best performance, as per the cost model, and thus selected as the final assembly code by ACT-AMX's enumeration pipeline.

Similar to AWS NKI results, achieving identical performance is a positive result since oneDNN is an expert-written kernel library with hand-written pattern-matching rules that perform complicated IR legalization due to the complex layout transformations in the instruction semantics (see Fig. 19), whereas ACT-AMX is automatically generated by ACT from just the TAIDL description.

ACT-AMX performs the same IR legalization using a sequence of IR-to-IR rewrites: 22 rewrites over 6 iterations in Phase 1 for `cnn_inf_amx`, and 14 rewrites over 4 iterations for `rnn_inf_amx`. This shows the importance of our design choice to integrate IR-to-IR rewrites with the instruction selection phase of ACT (§5.2), with no hand-written accelerator-specific legalization rules.

8.6 Comparative Evaluations: Gemmini [36]

We evaluate ACT-Gemmini against the Exo library [45] and Gemmini SW library [37]

Baselines

Exo Library. Exo [45] is a kernel programming DSL based on an orthogonal paradigm of exocompilation, i.e., the user writes both the input program and its schedule. This includes specifying tile sizes (line 106 of [2]), loop transformations (line 158 of [2]), instruction selection (line 149 of [2]), and buffer selection (line 207 of [2]). Exo developers have manually written hand-tuned schedules for six shapes of GEMM targeting Gemmini. This is the state-of-the-art kernel library for these shapes.

Gemmini SW Library. The Gemmini architecture ships with a custom kernel library [37] hand-optimized by Gemmini's designers. The Gemmini SW library supports three common DNN kernels—MAC: $A \times B + C$, GEMM: $A \times B$, and ADD: $A + B$, where A, B, and D are i8 matrices.

Benchmarks

Exo Library. We selected the six shapes of GEMM (Fig. 21, left) with hand-tuned Exo schedules.

Gemmini SW Library. We selected 15 tiled kernels (Fig. 21, right) categorized into three classes—(i) *"supported"*: three kernels directly supported by the Gemmini SW library (MAC, GEMM, ADD), (ii) *"composite"*: nine compositions of GEMM (×) and ADD (+) (fusion of upto 3 library calls), and (iii) *"not supported"*: three tensor operators not supported by the library (broadcast, reverse, reduce).

[2] https://github.com/exo-lang/exo/blob/v1.0.0/src/exo/platforms/gemmini.py

ISA Description

The TAIDL description we used to generate ACT-Gemmini included 11 RISC-type instructions for Gemmini. Gemmini also includes experimental CISC-type instructions that use an on-chip FSM. Here, we focus on the RISC-type instructions, with the rest discussed in Appendix G.

Evaluation Methodology

The enumeration pipeline (§7) of ACT-Gemmini uses a simple analytical cost model based on individual instruction costs. The reported performance is measured using RTL simulations. Since ACT focuses on backend passes, we delegate middle-end optimizations to XLA's autotuner [76]. Against the Exo library, we let ACT-Gemmini search across tiling and loop fission factors. Against the Gemmini SW library, we use a fixed tiling factor, same as the SW library.

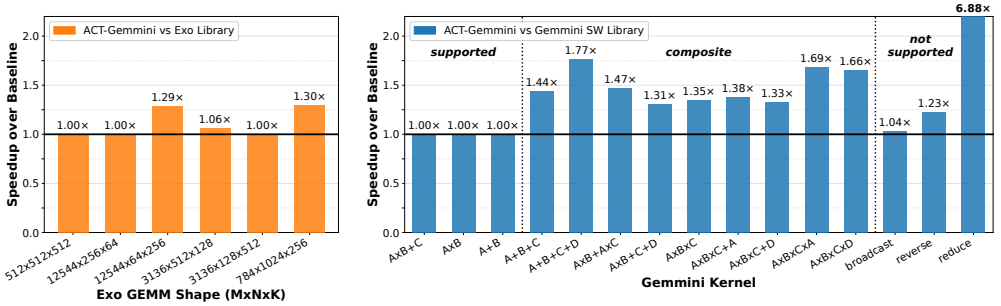


Fig. 21. Speedup of compiled code by ACT-Gemmini over Exo library (left) and Gemmini SW library (right). Note that the "not supported" evaluation (right) is a *cross-architecture* comparison: these operators have no Gemmini implementation in the SW library, so the baseline numbers were measured against the host CPU.

Results

Fig. 21 shows the speedup of compiled code by ACT-Gemmini over the expert-written kernel libraries. ACT-Gemmini with XLA autotuning *matches or outperforms* the hand-tuned Exo library with a speedup of up to **1.30x**. ACT-Gemmini achieves speedups up to **1x**, **1.77x**, and **6.88x** over the Gemmini SW library for "supported", "composite", and "not supported" kernels, respectively. For "not supported" kernels, the SW library has no Gemmini implementation, so this speedup is measured against host (Gemmini's Rocket CPU) execution — the practical cost of host fallback.

Observations & Analysis

Here, we summarize the key observations with individual kernels discussed in Appendix F & G.

Exo Library. ACT-Gemmini enumerates candidates with different tuning factors, including Exo's hand-tuned factors. Similar to §8.5, the hand-tuned baseline code is one of the enumerated candidates, by virtue of ACT's completeness guarantee. For three shapes, this candidate has the best performance, as per the cost model, and thus selected as the final assembly code by ACT-Gemmini. For other three shapes, candidates with better loop fission factors were selected by ACT-Gemmini.

Gemmini SW Library ("supported"). Similarly, we observed that the final assembly codes compiled by ACT-Gemmini were the same as the Gemmini SW Library for the three "supported" kernels.

Gemmini SW Library ("composite"). ACT-Gemmini eliminates the redundant load-store instructions between library calls for "composite" kernels by storing the intermediates in the accelerator scratchpads. This increases data reuse and reduces data movement, thus better performance. We observed that speedup increases with the kernel size (# nodes). For GEMM (×), speedup goes from 1x ($A \times B$) to 1.35x ($A \times B \times C$) to 1.66x ($A \times B \times C \times D$). Likewise for ADD (+), 1x to 1.44x to 1.77x. We limit our evaluations to a maximum of 3 library calls to avoid exaggerated performance results.

Gemmini SW Library ("not supported"). ACT-Gemmini is able to compile previously unsupported tensor operators like `reduce` to Gemmini using IR-to-IR rewrites (§2.1) and compile-time constant detection (§5.3). This leads to large speedups (up to 6.88x) over host execution fallback.

8.7 Compilation Time Analysis

Next, we perform breakdown analyses of the compilation time of ACT-generated compiler backends for varying tiled kernel sizes from synthetic and real-world workloads.

Tiled kernels typically contain 10–50 nodes, as seen in TPUGraphs dataset [75]. However, the Gemmini kernels used in §8.6 were limited to at most 10 nodes since the baselines were kernel libraries. Therefore, to analyze the compilation time of ACT-Gemmini on representative tiled kernels, we generated 100 synthetic kernels (size 7–89 nodes) using NNSmith [61], a random DNN generator (more details in Appendix E). We also analyze the compilation time of ACT-NKI for the three largest TPUGraphs kernels, labeled as T1 (150 nodes), T2 (270 nodes), and T3 (390 nodes).

Evaluation methodology

The compilation time reported is final.time in Fig. 18 (a) and was measured on a 20-core Intel i7-13700H CPU, averaged over 20 trials. This is the wall clock time of the *entire* compilation, including all pii candidates, fallback re-attempts (§4.1), and the cost model loop (§7). ACT-Gemmini and ACT-NKI were set to an overall timeout after 5 seconds, which was never reached (max ~500 ms).

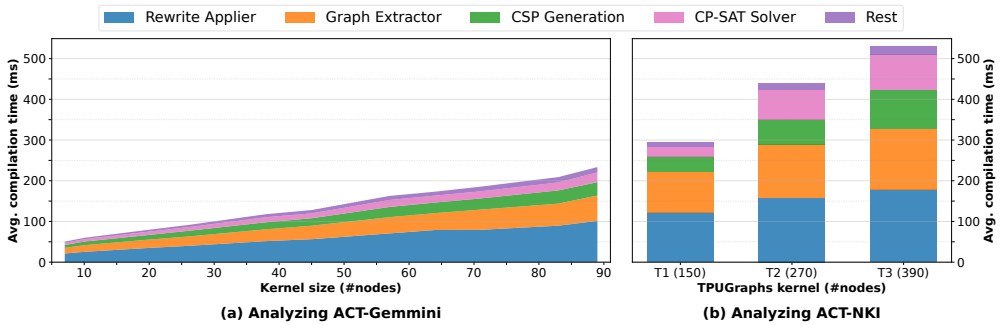


Fig. 22. Breakdown of compilation time. (a) ACT-Gemmini on synthetic kernels generated using NNSmith [61] and (b) ACT-NKI on the three largest kernels in TPUGraphs [75] Tile dataset.

Results & Observations

Fig. 22 shows the breakdown of compilation time of ACT-Gemmini and ACT-NKI. We observed small compilation times (< 233 ms) for a wide range of kernel sizes and even for large kernel sizes (529 ms for 390 nodes). The main reasons for the small compilation time are:

- Compact representation of infinite equivalent assembly code by modeling instruction selection as an equality saturation problem. Fig. 23 shows a simple snippet of an e-graph with cycles representing infinite equivalent assembly code.
- Bounded approach of graph extraction (in §5.4) prioritizes shorter assembly code, thereby enumerating candidates with no redundant data movement first. This leads to faster saturation of performance cost and shorter compilation times.
- Instruction scheduling heuristic (in §6.1) aims to minimize the interference graph. This results in fewer constraints in the generated CSP and, thereby, results in short solver time, even for large pii graphs (86 ms for 390 nodes).

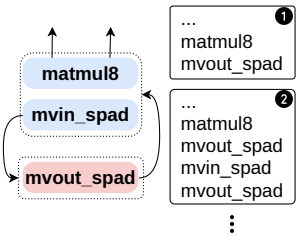


Fig. 23. Snippet of e-graph (left) with cycles between mvin and mvout e-classes. This represents infinite equivalent candidates (right), and the candidate with no redundant data movement is enumerated first.

ACT-generated compiler backends maintain low compilation overhead for a wide range of tiled kernel sizes.

9 Related Works

ISA description languages. VADL [33] and ILA [44] describe ISAs at the *scalar* level, using constructs such as `forall`, `fold`, and explicit array indexing, and focus primarily on CPU ISAs. TAIDL [50] instead describes instruction semantics at the *tensor* level (§3.2), and its tensor operators and types can themselves be parameterized by instruction attributes. Together, these features enable TAIDL to model the complex, parameterized instruction sets of tensor accelerators. We adopt TAIDL since it already supports several existing accelerator platforms [14, 36, 46, 55, 81, 92, 103].

Vectorizer generators and synthesis-based compilation. Vectorizer generators such as Vegen [18], Diospyros [93], and Isaria [90] target CPUs like x86 and DSPs like Hexagon HVX with fixed-length vector instructions. Synthesis-based instruction selectors such as Hydride [56] and Rake [2] also target fixed-width instructions, which are amenable to SMT-based reasoning. Compared to these techniques that only focus on instruction selection, ACT generates the entire compiler backend, including memory allocation for tensor accelerators. Furthermore, prior works cannot directly handle the complex parameterized instructions present in tensor accelerator ISAs.

Automated compiler backend generation. Prior work on automated CPU backend generation includes declarative machine-description approaches [28], CEGIS-based instruction-selector synthesis [10], processor-description-based generation [32], and LLM-based approaches [101]. OpenVADL [34, 74] generates TableGen-style LLVM backends from VADL [33], assuming little semantic gap between VADL and LLVM IR and thus requiring no complex legalization. Another related work in this area is a backend generator for the xADL processor-description language [8, 9], which targets only scalar register-based CPU ISAs and leaves vector ISAs to future work. Tensor accelerators instead have parameterized instructions, software-managed scratchpads, and layout-transforming semantics (e.g., Intel AMX [46], Fig. 19) that require accelerator-specific legalization. ACT therefore interleaves legalization with instruction selection in an e-graph rather than using a pattern-based selector, which also avoids phase ordering (§2.4).

Compiler support for novel accelerators. 3LA [43] is a recent work that makes commendable progress in developing basic compiler support by enabling simulation testing for end-to-end ML workloads on novel hardware. However, it requires the user to provide tensor IR-to-accelerator IR mapping for instruction selection and an accelerator IR-to-assembly code generation driver, and does not model scratchpad reuses. In contrast, ACT focuses on automatically generating a compiler backend just from the accelerator ISA description.

MLIR-based compiler stacks. MLIR [59] provides a multi-level compiler infrastructure with extensible dialects, and many modern ML compilers are built around this ecosystem, including the XLA compiler [20] that uses the StableHLO dialect [23]. MLIR ecosystem includes various lowering passes at a machine-independent IR level such as memref bufferization (note that this is different from the on-chip memory allocation discussed in §6), as well as hand-written target-specific lowering passes [38, 42] for code generation. ACT is complementary to these efforts — it focuses on automatically generating the accelerator-specific compiler backend itself (instruction selection, scheduling, and on-chip tensor memory allocation), and these generated backends can be integrated as a target-specific lowering pass in a larger MLIR-based pipeline.

Kernel programming languages like Exo [45], Pallas [40], Triton [91], and NKI [83] externalize target-specific code generation to user code, simplifying accelerator programming. These languages offer high-level constructs to program hardware, rather than directly coding in assembly, increasing user productivity. Exo, in particular, offers an intuitive scheduling DSL for exploring optimizations. However, these languages do not automatically generate compiler backends; instead, they rely on users to manually write instructions and schedules, and require a device-specific code generator.

Equality Saturation in Compilers. Prior works like Tensat [100], SPORES [98], Cranelift [31] use equality saturation to optimize machine learning computation graphs, linear algebra, and Rust functions, respectively. These works focus only on middle-end optimizations using IR-to-IR rewrites, whereas ACT uses both IR-to-IR rewrites and auto-generated IR-to-ISA rewrites to concurrently explore instruction selection opportunities. MISAAL [71] uses equality saturation to lower Halide IR using synthesized rewrite rules, and LIAR [26] likewise applies synthesized rewrites to recognize latent idioms in a functional array language. However, their rewrites carry no preconditions, and thus can not model parameterized instructions prevalent in tensor accelerator ISAs [83] (§5.2).

E-graph extraction algorithms. Prior equality-saturation workflows typically perform one-shot extraction using either greedy heuristics (the default strategy in egg [99]) or ILP-based optimal extraction [98, 100], with SmoothE [11] recently relaxing the same one-shot problem into a differentiable one. These methods are typically tied to a local objective integrated into the extraction algorithm. In our setting, this is insufficient because a candidate pii graph selected in Phase 1 can still fail in Phase 2 (as discussed in §4.1). Therefore, ACT uses an enumerative extraction algorithm (§5.3) with inter-phase fallback ensuring all equivalent pii graphs are enumerated and remains independent of any built-in cost objectives, allowing integration of external black-box cost models.

Instruction scheduling is the closest problem to the topological ordering discussed in §6.1. A common technique for scheduling instructions for pipelined processors is list scheduling [68]. But unlike our problem, these algorithms minimize the critical path length rather than the interference graphs. The Sethi–Ullman algorithm [86] generates optimal code for arithmetic expressions using minimal registers. Prior works have generalized this to 1-load binary DAGs [15] and non-binary trees [5]. However, any depth-first traversal of the expression tree is not always optimal for more than one register file (counterexample in Appendix B). We extend the Sethi-Ullman algorithm to multiple multi-dimensional tensor buffers with a fallback pruning strategy (§6.4).

On-chip scratchpad and register allocation are the closest problems to the tensor memory allocation discussed in §6.2. A widely adopted solution for register allocation is graph coloring [3] over the interference graph, while the puzzle board model [79] handles registers of different sizes. Unison [62, 63] formulates register allocation and instruction scheduling as a single combinatorial problem, and bit-aware register binding [12] admits a polynomial-time exact solution when re-allocation is permitted. None of these, however, extend to our setting, since we pack multi-dimensional tensor buffers into multi-dimensional scratchpads (§6.2) rather than one-dimensional registers, and not all buffers are spillable [83]. On-chip scratchpad memory allocation is often related to the rectangle packing problem, which can be modeled as a constraint satisfaction problem (CSP), with improved scalability by pruning techniques proposed in meta-CSP [67], TelaMalloc [64], and MiniMalloc [66]. However, these solutions only model interference graphs and, thus, can not represent complex addressing constraints prevalent in tensor accelerator ISAs [83] and the identity instructions (discussed in §5.2), both of which ACT encodes in its generated CSP.

10 Conclusion

In this paper, we propose the first accelerator compiler backend generator, ACT, that automatically generates a sound and complete compiler backend from a tensor accelerator ISA description. We provide a novel ISA-parameterized compilation algorithm and demonstrate the generality of our approach by generating backends for seven tensor accelerators. These backends generate performant code with large compilation coverage and small compilation times.

ACT fills an important gap in software development for accelerator platforms left under-explored due to the lack of compiler backends targeting them. It provides an agile and evolvable methodology to quickly build compiler backends that bridge the hardware and software communities.

11 Data Availability Statement

Our artifact is a repository of code (10+ KLOC) written in C++, Rust, and Python that implements ACT compatible with XLA-HLO IR. The artifact also includes benchmarks and docker environments used for performance evaluation, fuzz testing, stress testing, and scripts to reproduce the plots. Zenodo archive of the artifact is available at https://doi.org/10.5281/zenodo.21926082.

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 and its associated tutorials are accessible at https://github.com/act-compiler/act; we welcome contributions and feedback.

12 Acknowledgments

We thank the anonymous reviewers for their constructive feedback, as well as Stefanos Baziotis and Damitha Lenadora for their valuable comments on the paper. Special thanks to Zhihao Wang, Advait Tahilyani, Alan Xie, Pedro Couto, Tanmay Garudadri, Nitin Krishna, Mihir Tandon, and Aarav Khanna for their contributions to the ecosystem and its tooling. We are grateful to Niansong Zhang, Jianming Tong, and Minsik Kim for their guidance on using their respective accelerator designs. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by NSF under grant CCF-2338739.

References

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

[2] Maaz Bin Safeer Ahmad, Alexander J Root, Andrew Adams, Shoaib Kamil, and Alvin Cheung. 2022. Vector Instruction Selection for Digital Signal Processors Using Program Synthesis. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '22)*. Association for Computing Machinery, New York, NY, USA, 1004–1016. doi:10.1145/3503222.3507714

[3] 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

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 929–947. doi:10.1145/3620665.3640366

[5] A. W. Appel and K. J. Supowit. 1987. Generalization of the Sethi-Ullman algorithm for register allocation. *Softw. Pract. Exper.* 17, 6 (June 1987), 417–421. doi:10.1002/spe.4380170607

[6] 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

[7] Franz Baader and Tobias Nipkow. 1998. *Term Rewriting and All That.* Cambridge University Press. doi:10.1017/CBO9781139172752

[8] Florian Brandner, Dietmar Ebner, and Andreas Krall. 2007. Compiler Generation from Structural Architecture Descriptions. In *Proceedings of the 2007 International Conference on Compilers, Architecture, and Synthesis for Embedded*

Systems (CASES '07). Association for Computing Machinery, 13–22. doi:10.1145/1289881.1289886

[9] Florian Brandner, Viktor Pavlu, and Andreas Krall. 2013. Automatic Generation of Compiler Backends. *Software: Practice and Experience* 43, 2 (2013), 207–240. doi:10.1002/spe.2106

[10] Sebastian Buchwald, Andreas Fried, and Sebastian Hack. 2018. Synthesizing an instruction selection rule library from semantic specifications. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization* (Vienna, Austria) *(CGO '18)*. Association for Computing Machinery, New York, NY, USA, 300–313. doi:10.1145/3168821

[11] Yaohui Cai, Kaixin Yang, Chenhui Deng, Cunxi Yu, and Zhiru Zhang. 2025. SmoothE: Differentiable E-Graph Extraction. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '25)*. Association for Computing Machinery, 1020–1034. doi:10.1145/3669940.3707262

[12] Michael Canesche, Ricardo Ferreira, José Augusto Nacif, and Fernando Magno Quintão Pereira. 2022. A Polynomial Time Exact Solution to the Bit-Aware Register Binding Problem. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction (CC 2022)*. Association for Computing Machinery, 29–40. doi:10.1145/3497776.3517773

[13] Patrick Cegielski and Denis Richard. 2001. Decidability of the theory of the natural integers with the cantor pairing function and the successor. *Theor. Comput. Sci.* 257, 1–2 (April 2001), 51–77. doi:10.1016/S0304-3975(00)00109-2

[14] Hongzheng Chen, Niansong Zhang, Shaojie Xiang, Zhichen Zeng, Mengjia Dai, and Zhiru Zhang. 2024. Allo: A Programming Model for Composable Accelerator Design. *Proc. ACM Program. Lang.* 8, PLDI, Article 171 (June 2024), 28 pages. doi:10.1145/3656401

[15] Stephen Chen. 1975. On the Sethi-Ullman algorithm. *International Journal of Computer Mathematics* 5 (1975), 37–55. doi:10.1080/00207167508803101

[16] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems* (Salt Lake City, Utah, USA) *(ASPLOS '14)*. Association for Computing Machinery, New York, NY, USA, 269–284. doi:10.1145/2541940.2541967

[17] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. 2018. TVM: An automated End-to-End optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 578–594.

[18] Yishen Chen, Charith Mendis, Michael Carbin, and Saman Amarasinghe. 2021. VeGen: a vectorizer generator for SIMD and beyond. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Virtual, USA) *(ASPLOS '21)*. Association for Computing Machinery, New York, NY, USA, 902–914. doi:10.1145/3445814.3446692

[19] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 367–379. doi:10.1109/ISCA.2016.40

[20] OpenXLA Contributors. 2024. XLA Compiler. https://openxla.org/xla

[21] OpenXLA Contributors. 2025. Dynamism in StableHLO. https://openxla.org/stablehlo/dynamism.

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

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

[24] OpenXLA Contributors. 2026. XLA:GPU Emitters. https://openxla.org/xla/emitters.

[25] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. FLASHATTENTION: fast and memory-efficient exact attention with IO-awareness. In *Proceedings of the 36th International Conference on Neural Information Processing Systems* (New Orleans, LA, USA) *(NIPS '22)*. Curran Associates Inc., Red Hook, NY, USA, Article 1189, 16 pages. doi:10.52202/068431-1189

[26] Jonathan Van der Cruysse and Christophe Dubach. 2024. Latent Idiom Recognition for a Minimalist Functional Array Language Using Equality Saturation. In *Proceedings of the 2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2024)*. IEEE, 270–282. doi:10.1109/CGO57630.2024.10444879

[27] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018). doi:10.48550/arXiv.1810.04805

[28] João Dias and Norman Ramsey. 2010. Automatically generating instruction selectors using declarative machine descriptions. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 403–416. doi:10.1145/1706299.1706346

[29] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 92–104. doi:10.1145/2749469.2750389

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

[31] Chris Fallin. 2023. ægraphs: Acyclic E-graphs for Efficient Optimization in a Production Compiler. https://pldi23.sigplan.org/details/egraphs-2023-papers/2/-graphs-Acyclic-E-graphs-for-Efficient-Optimization-in-a-Production-Compiler Invited Talk, EGRAPHS 2023 - E-Graph Research, Applications, Practices, and Human-factors Symposium, PLDI 2023.

[32] Stefan Farfeleder, Andreas Krall, Edwin Steiner, and Florian Brandner. 2006. Effective Compiler Generation by Architecture Description. In *Proceedings of the 2006 ACM SIGPLAN/SIGBED Conference on Language, Compilers, and Tool Support for Embedded Systems* (Ottawa, Ontario, Canada) *(LCTES '06)*. Association for Computing Machinery, New York, NY, USA, 145–152. doi:10.1145/1134650.1134671

[33] Florian Freitag, Linus Halder, Simon Himmelbauer, Christoph Hochrainer, Benedikt Huber, Benjamin Kasper, Niklas Mischkulnig, Michael Nestler, Philipp Paulweber, Kevin Per, Matthias Raschhofer, Alexander Ripar, Tobias Schwarzinger, Johannes Zottele, and Andreas Krall. 2024. The Vienna Architecture Description Language. arXiv:2402.09087 [cs.PL] https://arxiv.org/abs/2402.09087

[34] Florian Freitag, Linus Halder, Benedikt Huber, Benjamin Kasper, Michael Nestler, Kevin Per, Matthias Raschhofer, Alexander Ripar, Johannes Zottele, and Andreas Krall. 2025. OpenVADL: An Open Source Implementation of the Vienna Architecture Description Language. In *Architecture of Computing Systems (ARCS 2025) (Lecture Notes in Computer Science)*. Springer, 156–171. doi:10.1007/978-3-032-03281-2_11

[35] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018). https://mlsys.org/Conferences/doc/2018/146.pdf

[36] 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

[37] 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. 2024. ucb-bar/gemmini: Berkeley's Spatial Array Generator. https://github.com/ucb-bar/gemmini.

[38] Renato Golin, Lorenzo Chelini, Adam Siemieniuk, Kavitha Madhu, Niranjan Hasabnis, Hans Pabst, Evangelos Georganas, and Alexander Heinecke. 2024. Towards a high-performance AI compiler with upstream MLIR. arXiv:2404.15204 [cs.PL] https://arxiv.org/abs/2404.15204

[39] Google. 2021. Google OR Tools: CP-SAT Solver. https://developers.google.com/optimization/cp/cp_solver.

[40] Google. 2024. Pallas: a JAX kernel language. https://jax.readthedocs.io/en/latest/pallas/index.html.

[41] Muyan Hu, Ashwin Venkatram, Shreyashri Biswas, Balamurugan Marimuthu, Bohan Hou, Gabriele Oliaro, Haojie Wang, Liyan Zheng, Xupeng Miao, Jidong Zhai, and Zhihao Jia. 2024. Optimal Kernel Orchestration for Tensor Programs with Korch. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 755–769. doi:10.1145/3620666.3651383

[42] Pengchao Hu, Man Lu, Lei Wang, and Guoyue Jiang. 2023. TPU-MLIR: A Compiler For TPU Using MLIR. arXiv:2210.15016 [cs.PL] https://arxiv.org/abs/2210.15016

[43] Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Gus Henry Smith, Thierry Tambe, Akash Gaonkar, Vishal Canumalla, Andrew Cheung, Gu-Yeon Wei, Aarti Gupta, Zachary Tatlock, and Sharad Malik. 2024. Application-level Validation of Accelerator Designs Using a Formal Software/Hardware Interface. *ACM Trans. Des. Autom. Electron. Syst.* 29, 2, Article 35 (Feb. 2024), 25 pages. doi:10.1145/3639051

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

[45] 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

[46] Intel. 2024. Intel® Advanced Matrix Extensions. https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/advanced-matrix-extensions/overview.html.

[47] Intel. 2025. Intel® Advanced Vector Extensions 512 (Intel® AVX-512) Overview. https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html.

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

[49] Intel. 2025. oneDNN Library. https://github.com/uxlfoundation/oneDNN/tree/v3.8.1.

[50] 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

[51] 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

[52] Stephen Johnson. 2001. Yacc: Yet Another Compiler-Compiler. *Unix Programmer's Manual* 2 (11 2001). https://www.cs.utexas.edu/users/novak/yaccpaper.htm

[53] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, Clifford Young, Xiang Zhou, Zongwei Zhou, and David A Patterson. 2023. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. In *Proceedings of the 50th Annual International Symposium on Computer Architecture* (Orlando, FL, USA) *(ISCA '23)*. Association for Computing Machinery, New York, NY, USA, Article 82, 14 pages. doi:10.1145/3579371.3589350

[54] Norman P. Jouppi, Doe Hyun Yoon, George Kurian, Sheng Li, Nishant Patil, James Laudon, Cliff Young, and David Patterson. 2020. A domain-specific supercomputer for training deep neural networks. *Commun. ACM* 63, 7 (June 2020), 67–78. doi:10.1145/3360307

[55] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3079856.3080246

[56] Akash Kothari, Abdul Rafae Noor, Muchen Xu, Hassam Uddin, Dhruv Baronia, Stefanos Baziotis, Vikram Adve, Charith Mendis, and Sudipta Sengupta. 2024. Hydride: A Retargetable and Extensible Synthesis-based Compiler for Modern Hardware Architectures. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 514–529. doi:10.1145/3620665.3640385

[57] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. *SIGPLAN Not.* 53, 2 (March 2018), 461–475. doi:10.1145/3296957.3173176

[58] Chris Lattner and Vikram Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-Directed and Runtime Optimization* (Palo Alto, California) *(CGO '04)*. IEEE Computer Society, USA, 75. doi:10.1109/CGO.2004.1281665

[59] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14. doi:10.1109/CGO51591.2021.9370308

[60] 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

[61] 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

[62] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2016. Register Allocation and Instruction Scheduling in Unison. In *Proceedings of the 25th International Conference on Compiler Construction (CC 2016)*. Association for Computing Machinery, 263–264. doi:10.1145/2892208.2892237

[63] Roberto Castañeda Lozano, Mats Carlsson, Gabriel Hjort Blindell, and Christian Schulte. 2019. Combinatorial Register Allocation and Instruction Scheduling. *ACM Transactions on Programming Languages and Systems* 41, 3, Article 17 (2019), 53 pages. doi:10.1145/3332373

[64] Martin Maas, Ulysse Beaugnon, Arun Chauhan, and Berkin Ilbeyi. 2022. TelaMalloc: Efficient On-Chip Memory Allocation for Production Machine Learning Accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 123–137. doi:10.1145/3567955.3567961

[65] Charith Mendis, Alex Renda, Dr.Saman Amarasinghe, and Michael Carbin. 2019. Ithemal: Accurate, Portable and Fast Basic Block Throughput Estimation using Deep Neural Networks. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, 4505–4515. https://proceedings.mlr.press/v97/mendis19a.html

[66] Michael D. Moffitt. 2024. MiniMalloc: A Lightweight Memory Allocator for Hardware-Accelerated Machine Learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Vancouver, BC, Canada) *(ASPLOS '23)*. Association for Computing Machinery, New York, NY, USA, 238–252. doi:10.1145/3623278.3624752

[67] Michael D. Moffitt and Martha E. Pollack. 2006. Optimal rectangle packing: a meta-CSP approach. In *Proceedings of the Sixteenth International Conference on International Conference on Automated Planning and Scheduling* (Cumbria, UK) *(ICAPS'06)*. AAAI Press, 93–102. https://cdn.aaai.org/ICAPS/2006/ICAPS06-010.pdf

[68] Steven S. Muchnick and Phillip B. Gibbons. 2004. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Not.* 39, 4 (April 2004), 167–174. doi:10.1145/989393.989413

[69] Wei Niu, Jiexiong Guan, Yanzhi Wang, Gagan Agrawal, and Bin Ren. 2021. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 883–898. doi:10.1145/3453483.3454083

[70] Wei Niu, Md Musfiqur Rahman Sanim, Zhihao Shu, Jiexiong Guan, Xipeng Shen, Miao Yin, Gagan Agrawal, and Bin Ren. 2024. SmartMem: Layout Transformation Elimination and Adaptation for Efficient DNN Execution on Mobile. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 916–931. doi:10.1145/3620666.3651384

[71] Abdul Rafae Noor, Dhruv Baronia, Akash Kothari, Muchen Xu, Charith Mendis, and Vikram S. Adve. 2025. MISAAL: Synthesis-Based Automatic Generation of Efficient and Retargetable Semantics-Driven Optimizations. *Proc. ACM Program. Lang.* 9, PLDI, Article 198 (June 2025), 24 pages. doi:10.1145/3729301

[72] Anjali Pal, Brett Saiki, Ryan Tjoa, Cynthia Richey, Amy Zhu, Oliver Flatt, Max Willsey, Zachary Tatlock, and Chandrakana Nandi. 2023. Equality Saturation Theory Exploration à la Carte. *Proceedings of the ACM on Programming Languages* 7, OOPSLA2, Article 258 (2023), 29 pages. doi:10.1145/3622834

[73] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W. Keckler, and Joel Emer. 2019. Timeloop: A Systematic Approach to DNN Accelerator Evaluation. In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 304–315. doi:10.1109/ISPASS.2019.00042

[74] Kevin Per. 2025. *LLVM Compiler Generator in OpenVADL*. Master's thesis. Technische Universität Wien.

[75] 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

[76] Phitchaya Mangpo Phothilimthana, Amit Sabne, Nikhil Sarda, Karthik Srinivasa Murthy, Yanqi Zhou, Christof Angermueller, Mike Burrows, Sudip Roy, Ketan Mandke, Rezsa Farahani, Yu Emma Wang, Berkin Ilbeyi, Blake Hechtman, Bjarke Roune, Shen Wang, Yuanzhong Xu, and Samuel J. Kaufman. 2021. A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers. In *2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 1–16. doi:10.1109/PACT52795.2021.00008

[77] Raghu Prabhakar, Sumti Jairath, and Jinuk Luke Shin. 2022. SambaNova SN10 RDU: A 7nm Dataflow Architecture to Accelerate Software 2.0. In *2022 IEEE International Solid-State Circuits Conference (ISSCC)*, Vol. 65. 350–352. doi:10.1109/ISSCC42614.2022.9731612

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

[79] Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register allocation by puzzle solving. *SIGPLAN Not.* 43, 6 (June 2008), 216–226. doi:10.1145/1379022.1375609

[80] Amazon Web Services. 2024. AWS Inferentia. https://aws.amazon.com/ai/machine-learning/inferentia/.

[81] Amazon Web Services. 2024. AWS Trainium. https://aws.amazon.com/ai/machine-learning/trainium/.

[82] Amazon Web Services. 2026. Neuron Compiler CLI Reference Guide (neuronx-cc). https://awsdocs-neuron.readthedocs-hosted.com/en/v2.27.1/compiler/neuronx-cc/api-reference-guide/neuron-compiler-cli-reference-guide.html.

[83] Amazon Web Services. 2026. Neuron Kernel Interface (NKI) Documentation. https://awsdocs-neuron.readthedocs-hosted.com/en/v2.27.1/nki/index.html.

[84] Amazon Web Services. 2026. NKI Language Guide. https://awsdocs-neuron.readthedocs-hosted.com/en/v2.27.1/nki/get-started/nki-language-guide.html#control-flow.

[85] Amazon Web Services. 2026. The Trainium Memory Hierarchy. https://awsdocs-neuron.readthedocs-hosted.com/en/v2.27.1/nki/get-started/about/memory-hierarchy-overview.html.

[86] Ravi Sethi and J. D. Ullman. 1970. The Generation of Optimal Code for Arithmetic Expressions. *J. ACM* 17, 4 (Oct. 1970), 715–728. doi:10.1145/321607.321620

[87] Wilson Snyder, Paul Wasson, and Duane et al Galbi. 2026. *Verilator*. https://github.com/verilator/verilator

[88] Amazon Staff. 2024. Amazon invests $110 million to support AI research at universities using Trainium chips. https://www.aboutamazon.com/news/aws/amazon-trainium-investment-university-ai-research.

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

[90] Samuel Thomas and James Bornholt. 2024. Automatic Generation of Vectorizing Compilers for Customizable Digital Signal Processors. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 19–34. doi:10.1145/3617232.3624873

[91] Philippe Tillet, H. T. Kung, and David Cox. 2019. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages* (Phoenix, AZ, USA) *(MAPL 2019)*. Association for Computing Machinery, New York, NY, USA, 10–19. doi:10.1145/3315508.3329973

[92] Jianming Tong, Anirudh Itagi, Prasanth Chatarasi, and Tushar Krishna. 2024. FEATHER: A Reconfigurable Accelerator with Data Reordering Support for Low-Cost On-Chip Dataflow Switching. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. 198–214. doi:10.1109/ISCA59077.2024.00024

[93] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. 2020. A Synthesis-Aided Compiler for DSP Architectures (WiP Paper). In *The 21st ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems* (London, United Kingdom) *(LCTES '20)*. Association for Computing Machinery, New York, NY, USA, 131–135. doi:10.1145/3372799.3394358

[94] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[95] Steven R. Vegdahl. 1982. Phase coupling and constant generation in an optimizing microcode compiler. *SIGMICRO Newsl.* 13, 4 (Oct. 1982), 125–133. doi:10.1145/1014194.800942

[96] 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

[97] 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

[98] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. 2020. SPORES: sum-product optimization via relational equality saturation for large scale linear algebra. *Proc. VLDB Endow.* 13, 12 (July 2020), 1919–1932. doi:10.14778/3407790.3407799

[99] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. egg: Fast and extensible equality saturation. *Proc. ACM Program. Lang.* 5, POPL, Article 23 (Jan. 2021), 29 pages. doi:10.1145/3434304

[100] 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

[101] Ming Zhong, Fang Lv, Lulin Wang, Lei Qiu, Yingying Wang, Ying Liu, Huimin Cui, Xiaobing Feng, and Jingling Xue. 2025. VEGA: Automatically Generating Compiler Backends using a Pre-trained Transformer Model. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 90–106. doi:10.1145/3696443.3708931

[102] Gaofeng Zhou, Jianyang Zhou, and Haijun Lin. 2018. Research on NVIDIA Deep Learning Accelerator. In *2018 12th IEEE International Conference on Anti-counterfeiting, Security, and Identification (ASID)*. 192–195. doi:10.1109/ICASID.2018.8693202

[103] Junkang Zhu, Minsik Kim, Cheng-Hsun Lu, Wei Tang, Tianyu Wei, and Zhengya Zhang. 2025. EVA: A 16mm2 1.54TFLOPS Tiled-Based Accelerator for Evolvable Edge Computing. In *2025 Symposium on VLSI Technology and Circuits (VLSI Technology and Circuits)*. 1–3. doi:10.23919/VLSITechnologyandCir65189.2025.11075176

[104] 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

Received 2026-03-17; accepted 2026-08-06