ACT: Automatically Generating 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
KRUT PATEL*, *NVIDIA*, USA
KAUSTUBH KHULBE, *University of Illinois Urbana-Champaign*, USA
JAI ARORA, *University of Illinois Urbana-Champaign*, USA
CHARITH MENDIS, *University of Illinois Urbana-Champaign*, USA

Tensor compilers play a key role in enabling high-performance implementations of deep learning workloads. These compilers rely on existing CPU and GPU code generation backends to generate device-specific code. Recently, many tensor accelerators (neural processing units) have been proposed to further accelerate these workloads. Compared to commodity hardware, however, most of the proposed tensor accelerators do not have compiler backends with code generation support. Moreover, the accelerator designs are subject to fast iteration cycles, making it difficult to manually develop compiler backends similar to commodity hardware platforms. Therefore, to increase adoption and enable faster software development cycles for novel tensor accelerator designs, we need to make the compiler backend construction process more agile.

To address this gap, we introduce ACT, a compiler backend generator that automatically generates compiler backends for tensor accelerators, given just the instruction set architecture (ISA) descriptions. We first formally specify the compiler backend generation problem that introduces a novel specification for describing tensor accelerator ISAs. Next, we design ACT such that it supports user-programmable memories and complex parameterized instructions that are prevalent in tensor accelerators. ACT uses a novel parameterized equality saturation-based instruction selection phase and a constraint programming-based memory allocation phase. We prove that compiler backends generated by ACT are sound and complete. Finally, we generate compiler backends for three accelerator platforms from industry and academia, and show that they match or outperform code written using hand-optimized kernel libraries while maintaining low compilation overheads.

1 INTRODUCTION

Machine learning (ML) workloads, specifically deep learning (DL) workloads, have gained popularity in recent years. ML practitioners use tensor programming languages such as Tensorflow [2], PyTorch [6], and JAX [25] to express their computations, and tensor compilers such as XLA [18], TorchInductor [6], and TVM [15] to lower these computations into highly performant executables targeting a variety of CPUs and GPUs. These languages and compilers have been the backbone of enabling both end-user productivity and high-performance realizations of deep learning models.

Akin to general-purpose compilers, tensor compilers also use a hierarchical compilation pipeline (Fig. 1). They first lower programs written in tensor programming languages to tensor-operator-based intermediate representations (IR) known as tensor computation graphs. These graphs are then optimized using several graph-level optimizations [56]. The scale of these optimized graphs can range up to 1000s of nodes [55]. Finally, these optimized tensor computation graphs are partitioned into smaller sub-graphs (*tensor kernels*) with typically 10-50 nodes, which are compiled to machine code individually and often in parallel [55] using *existing* compiler backends. For example, the XLA compiler [18] progressively lowers tensor computation graphs with high-level operators (HLO) [53] to low-level LLVM [42] IR using the MLIR [43] framework. Finally, for CPUs and GPUs[1], it utilizes LLVM's existing compiler backends[46] to generate respective machine code (e.g., x86, NVPTX).

*Work done when at *University of Illinois Urbana-Champaign*
[1]Triton dialect uses LLVM NVPTX backend [70]

To get the best performance out of tensor computations, many domain-specific tensor accelerators (a.k.a. neural processing units) have been proposed. 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's tensor processing units (TPU) [38, 39], Amazon's Inferentia [60] and Trainium [61], Xilinx's Versal AI Core [4], and Intel AMX [32] and a large body of academic designs such as Gemmini [26], FEATHER [69], and Eyeriss [17]. These accelerators are diverse, with different programmable memory hierarchies and compute capabilities.

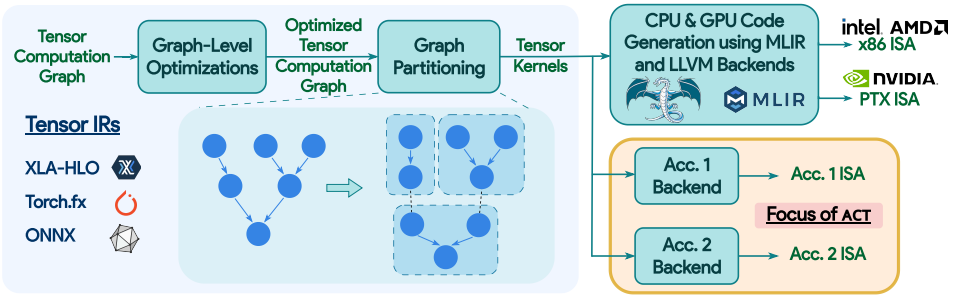


Fig. 1. Typical hierarchical compilation pipeline present in tensor compilers. Our solution, ACT, focuses on the accelerator-specific compiler backends (shaded orange), i.e., compiling tensor kernels to accelerator ISA.

1.1 Need for Generation of Accelerator-specific Compiler Backends

One of the key requirements for widespread use of a hardware architecture is the existence of compiler backends that generate device-specific code. For example, most CPU and GPU compilation pipelines for tensor programming languages rely on the mature compiler backends present in LLVM. Companies like Google and Amazon [65] have invested heavily in the compiler toolchains for their tensor accelerator offerings (Google TPUs and Amazon Inferentia) to enable widespread adoption. The (proprietary) compiler backend for Google TPUv3 supports 100s of fused tensor kernels as evident in TPUGraphs [55] Tile dataset. However, comparatively, a majority of tensor accelerators proposed in the literature do not have proper backends. In fact, according to the survey mentioned in [29], a majority of new accelerator papers only evaluate at the operator-level (e.g., convolution operator) or on synthetic kernels. One reason may be the *lack of specialized compiler backends* that enable the generation of device-specific code for a number of different tensor computation graphs.

Tensor accelerators with no compiler backend support often rely on custom hand-written kernel libraries. However, these kernel libraries support only a limited set of kernels and miss out on fusion opportunities outside this set. Therefore, compiler backends that support code generation for *arbitrary fused tensor kernels* are needed to promote the adoption of tensor accelerators.

Manually constructing compiler backends similar to CPUs and GPUs for each accelerator design can be too tedious and too slow to adapt to the variety and fast pace of research happening in the architecture community (§2.2). Therefore, the architecture and compiler community needs a *systemic* approach to *automate generation* of accelerator-specific compiler backends.

1.2 Challenges & Goals

There have been prior works that aim to automate compiler construction. Prior works such as Vegen [16], Isaria [67], and Diospyros [71] automatically generate vectorizers using instruction set architecture (ISA) semantics. 3LA [29] makes commendable progress in developing basic compiler support for prototype accelerators using formal software/hardware interface ILA [30]. However, the following challenges remain in automatically generating compiler backends for tensor accelerators with sufficient coverage over many diverse designs.

Challenge 1: Code generation and optimizations that involve user-programmable scratchpads. Most tensor accelerators have user-programmable scratchpads at different memory hierarchies that need to be explicitly controlled by the software. The data access patterns of tensor accelerators vary significantly in shape and size with multi-dimensional addressing (due to banked memory and/or multiple iso-compute units like in TPUv3 [38]) and multi-dimensional base elements (256-length vector in TPUv1 [39] unified buffer, 16×64 matrix in Intel AMX tile registers). An ideal compiler backend needs to *automatically* manage data allocation and reuse of these scratchpads during computation. Works such as Exo [31] have made commendable progress in modeling the scratchpads themselves, but leave the memory management to end-users. Similarly, 3LA's [29] flexible-matching-based technique for selecting accelerator offload does not model scratchpad reuse, incurring expensive reads/writes to host memory.

Challenge 2: Handling complex and parameterized ISA instructions. Tensor accelerators have complex instructions with parameters that control execution counts. For example, the matrix multiply instruction in Google TPUv1 [39] is parameterized by the shapes of the matrices involved. It takes a variable-sized B×256 (B is an instruction parameter) input, multiplies it by a 256×256 constant weight input, and produces a B×256 output. Methodologies adopted by vectorizer generators [16, 67, 71] can only handle fixed-size vector instructions, making them insufficient to handle tensor accelerator ISAs. A compiler backend generator for tensor accelerators should represent and handle these parameterized ISA instructions in a general manner applicable to multiple accelerators.

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

Goal 1: Increased compilation coverage. Tensor accelerator designs are often not Turing-complete and require host fallback for certain operators (or tensor kernels). Ideally, the compiler backend should have a large compilation coverage to minimize the host fallback, i.e., ideally compile successfully for all tensor kernels supported by the accelerator. In other words, the compiler backend generator should *generate sound and complete compiler backends*.

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.

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. We first provide a *novel formalization* of tensor accelerator ISA descriptions that capture the complexities mentioned in the challenges. The key insight behind this formalism is the use of the same operator language used in tensor IRs to partly represent accelerator ISA descriptions. This allows us to model scratchpad accesses as tensor slicing operations and provide rich parameterizations to ISA descriptions. We use these ISA descriptions to formally define the compiler backend generation problem and its soundness and completeness criteria in §3.

ACT achieves automatic generation (goal 2) by *parameterizing* compiler backend phases by the ISA descriptions, which are concretized for each accelerator given their ISA descriptions. ACT-generated backends consume tensor kernels (i.e., tensor computation sub-graphs) as input and produce accelerator-specific assembly code. ACT-generated backends consist of two main phases. The first phase introduces *parameterized* instruction selection based on equality saturation [66]. ACT generates and uses multiple types of rewrites (e.g., IR-to-IR, IR-to-ISA). Note that our representation of tensor ISA descriptions allows us to follow this axiomatic approach. We handle parameterized instructions with variable-size attributes (challenge 2) using a novel intermediary of *partially instantiated instructions* (pii). The second phase introduces a *parameterized* memory allocation pass based on constraint programming that supports multi-dimensional buffer accesses (challenge 1). Finally, we introduce both intra- and inter-phase fallback paths to make the generated compiler

complete. We prove that Act-generated compiler backends are *sound and complete* (goal 1) under the definitions in §3 and evaluate Act's compilation capabilities on different accelerator platforms.

In summary, the paper makes the following contributions.

- We provide a novel formalization of tensor accelerator ISA descriptions and formally define the compiler backend generation problem with soundness and completeness criteria. (§3)
- We introduce Act and its methodology that *automatically generates* compiler backends given just the ISA descriptions of tensor accelerators with completeness guarantee. (§4-§7)
- We design Act-based code generator using cost models to generate performant code. (§8)
- We study the role of key features in Act in modeling complex and parameterized ISA (Challenge 2) and increasing compilation coverage (Goal 1) by generating compiler backends for a commercial accelerator (Intel AMX), an academic accelerator (Gemmini), and an accelerator generated using an accelerator design language (ADL). (§9.2)
- We show that Act-generated compiler backends generate performant assembly code that matches or outperforms the state-of-the-art kernel libraries (up to 1.77x speedup). (§9.3) Further, we perform fuzz testing and stress testing to show that the compilation time is less than a second, even for extremely large kernels (311 ms for 390 nodes). (§9.4)

2 BACKGROUND

We first provide the necessary background about tensor compilers and tensor accelerators to facilitate a more formal problem formulation (§3) and our solution (§4-§6) in subsequent sections.

2.1 Tensors and computation representations in tensor compilers

Tensors are n-dimensional objects (generalization of 0-dimensional scalars, 1-dimensional vectors, and 2-dimensional matrices) that are usually represented as multi-dimensional arrays. A tensor of *tensor-type* $\mathbb{E}[S]$ has a shape S with elements of scalar basetype $\mathbb{E} \in \mathcal{E}$. Shape S is represented as a tuple of integers, each representing the dimension size. $\mathcal{E}$ is a set of basetypes such as uint8 (u8), int32 (i32), float32 (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. Tensor compilers such as XLA [18] and TorchInductor [6] manipulate tensors as first-class objects using tensor operators. These operators are usually tensor-type agnostic and are parameterized by various operator parameters. Together, we call these as *operator attributes*. For example, consider XLA's slice(x, s, e, p) operator that extracts a sub-tensor from the input tensor x of any tensor type, starting at indices s, ending at indices e, with a stride of p. The tensor type of x, s, e, p are operator attributes of slice. We call tensor operators parameterized by these attributes as *abstract tensor operators*. Table 1 briefly describes the tensor operators used in the paper based on the XLA compiler's High-Level Operator (HLO) IR. Visualizations of these operators are present in Appendix A, with detailed operational semantics documented in [53, 64].

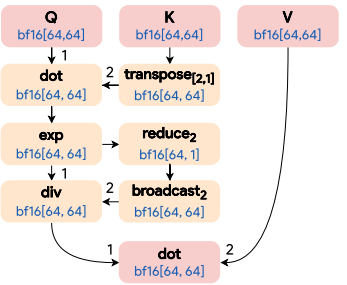


Fig. 2. Self-attention kernels in Transformer models like BERT [19] use QKV computation softmax$(Q \cdot K^T) \cdot V$ over 4-dimensional tensors. For illustration purposes, we assume a single-batch single-head QKV computation G_{QKV} over bf16[64, 64] tensors.

During the compilation of a given model, all operator attributes are instantiated. We call such instantiated tensor operators as *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	Concretizing 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 $dims$
$y = \text{dot}_{(c_1,c_2)}(x_1, x_2)$	$\text{type}(x_1), \text{type}(x_2), c_1, c_2$	Dot product of x_1, x_2 over contracting dimensions c_1, c_2
$y = \exp(x)$	$\text{type}(x)$	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 discussed in the paper (more details in Appendix A, [53, 64]). $\text{type}(x)$ 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. Tensor compilers internally represent computations as a sequence of tensor operators organized as a tensor computation graph. More formally, a *tensor computation graph* is a directed acyclic graph with tensor operators as nodes and operands as edges. The edges establish the data dependencies between different tensor operators. For example, consider a simplified tensor computation graph for attention computation, which is prevalent in many transformer [72] topologies, shown in Fig. 2. A tensor computation graph can be either concrete or abstract, depending on whether the operators within it are concrete or abstract. Unless specified otherwise, a tensor computation graph is considered concrete.

Tensor Rewrites. A tensor rewrite $P_l \rightarrow P_r$ is a semantic-preserving rule that substitutes a tensor computation subgraph P_l with another subgraph P_r, under certain preconditions on the operator attributes. Prior works [8, 36, 44, 74] have proposed rewrite rule verification systems.

Foundational Axioms ($\mathcal{R}$). XLA compiler uses 175+ rewrite rules, with 115 of these verified to full generality by TensorRight [8]. In our work, we adopt these rewrite rules, denoted as $\mathcal{R}$, as foundational axioms of the IR. One such rewrite rule is $\text{slice}_{[s_1:e_1]}(\text{slice}_{[s_2:e_2]}(x)) \rightarrow \text{slice}_{[s_1+s_2:e_1+s_2]}(x)$.

Semantic Equivalence ($\equiv_{\mathcal{R}}$). Tensor computation graphs G_1 and G_2 are said to be semantically equivalent, denoted as $G_1 \equiv_{\mathcal{R}} G_2$, if G_1 can be transformed into G_2 by applying rewrite rules in $\mathcal{R}$.

2.2 Tensor Accelerators

Tensor accelerators (a.k.a. neural processing units) are a class of hardware accelerators optimized for tensor computations using different microarchitectural innovations such as systolic array-based executions. Many such accelerators have been proposed with different capabilities and programmabilities. Some accelerators have programmable dataflows [26, 41, 57], while others are fixed [14, 17, 21, 39, 78]. They have diverse functional units such as matrix transpositions [38], matrix reshaping units, etc., with novel innovations such as intelligent on-chip routing [69].

Amongst this diversity, we notice two key features which we utilize in this work. First, from a compiler backend's point-of-view, for generating sound code, it is sufficient to focus just on the ISA's functionality without explicitly modeling micro-architectural implementations. Second, most of these accelerators support coarse-grained instructions that work with 1-D, 2-D, or 3-D tensors as input (e.g., matrix multiplication instructions). This enables us to model the ISA semantics at a higher level compared to more general-purpose ISAs such as CPUs.

2.3 Term Rewriting using Equality Saturation

The classical approach of term rewriting [9] is destructive, making optimizations sensitive to the order in which rewrites are applied. This is the well-known "phase-ordering" problem [73]. Equality saturation [66] uses *e-graphs* to overcome this phase-ordering problem by simultaneously applying rewrite rules. E-graphs can compactly represent equivalence relations over terms. An e-graph is a set of equivalence classes (*e-classes*), each of which is a set of equivalent nodes (*e-nodes*).

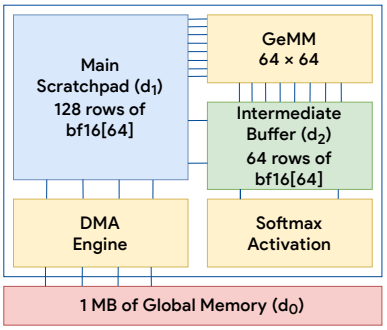
The equality saturation workflow consists of three stages: initialization, exploration, and extraction. First, the initial e-graph is created from the input term. Next, the exploration stage applies rewrite rules until saturation is reached (i.e., applying rewrites does not add any new information) or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

3 PROBLEM FORMULATION

In this section, we formalize the problem statement of a compiler backend generator that generates sound and complete compiler backends from just the accelerator ISA description.

First, we formalize the ISA description (§3.1-§3.3) of a tensor accelerator built on top of the existing formal models [8, 36, 44, 74] of tensors and tensor operators (§2.1). This formalism is driven by the key observation (as discussed in §2.2) that tensor accelerators support coarse-grained instructions that work on multi-dimensional data (commonly represented as tensors). Therefore, we use compositions of tensor operators (Table 1) to model the semantics of their instruction sets. To the best of our knowledge, this is the *first formulation* of tensor accelerator ISA descriptions.

Then, we formally define the equivalence condition (§3.4) between a tensor computation graph (IR) and an accelerator-specific assembly code. We use this equivalence to define the soundness and completeness of a given compiler backend, thereby completing the problem formulation (§3.5) of a compiler backend generator that addresses the challenges and goals mentioned in §1.2.



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

3.1 Tensor Accelerator ISA Description ISA^H

A tensor accelerator ISA description comprises (1) the user-programmable storage units (collectively termed *data model*) like scratchpads, and (2) a set of instructions that perform computations like data movement and compute instructions on the storage units. This is analogous to the x86 ISA description comprising a data model (registers, memory) and an instruction set (add, sub, mov, etc.). These storage units store the input, output, and intermediate variables. Similarly, the storage units in tensor accelerators store the intermediate tensor data and often have multi-dimensional shapes.

Running example. For illustration purposes, let's consider a hypothetical QKV accelerator H_{QKV} (Fig. 3 (a)) that supports the simple QKV kernel G_{QKV} (Fig. 2). The accelerator has two on-chip

storage units – a 16 KB scratchpad and an 8 KB intermediate buffer, and access to global memory via a DMA engine. It has two compute units – a general matrix multiply (gemm) unit and a softmax activation unit. It supports five data movement and two compute instructions listed in Fig. 3 (b). The gemm unit can be implemented as a 2-D systolic array with output or weight stationary dataflow like Gemmini [26], or more complex designs like MAERI [41] & FEATHER [69]. We use this hypothetical accelerator H_{QKV} as our running example for the rest of the paper.

3.2 Data Model D^H

The user-programmable storage units, like scratchpads on an accelerator, can be abstractly represented as *tensor buffers*. A tensor buffer is a tensor with a subset of dimensions representing *access dimensions* and the remaining dimensions representing the granularity of data access. Accelerator instructions modify sub-tensors of these tensor buffers, called *data slices*. We assume that an accelerator has a byte-addressable 1-D global memory (HBM) that is accessible by the host processor and is denoted d_0. Next, we formally define these terms using tensor notations discussed in §2.

DEF 3.1. *A **tensor buffer** d of type $(S_0, \mathbb{E}[S_1])$ is a multi-dimensional storage unit of shape S_0 with elements of uniform tensor-type $\mathbb{E}[S_1]$, which can be used to store tensor data. S_0 represents* access dimensions *and $\mathbb{E}[S_1]$ is the* granularity of data access. *In other words, d can be thought of as a tensor of type $\mathbb{E}[S_0, S_1]$ with first $\mathsf{dim}(S_0)$ dimensions as access dimensions.*

DEF 3.2. *A **data slice** $d[I_s{:}I_e]$ of tensor buffer d is a tensor of type $\mathbb{E}[(I_e - I_s), S_1]$ where $I_s, I_e \in \mathbb{N}_0^{|S_0|}$ and $I_s < I_e \leq S_0$. I_s and I_e are the start (inclusive) and end (exclusive) addresses of the data slice under zero-based multi-dimensional addressing.*

DEF 3.3. *A **memory state** $M = (d_0, d_1, \dots)$ is a tuple of tensors, one for each tensor buffer.*

Tensor buffer representations for user-programmable storage units of H_{QKV} are annotated in Fig. 3. The 16 KB scratchpad (d_1) consists of 128 rows of 64-length bf16 vectors. Instructions read and/or modify the scratchpad at a row granularity. Thus, it is represented as $d_1 = (128, \mathsf{bf16}[64])$.

3.3 Instruction Set Θ^H

The instruction set Θ^H is a set of *abstract instructions* that modify the tensor buffers in D^H and are parameterized by instruction attributes. This is analogous to x86 instructions, such as mov and add, which modify registers or memory and are parameterized by immediate values, register names, and memory addresses. A *concrete instruction* is an instantiation of an abstract instruction with specific values for the instruction attributes. The concrete instructions are *deterministic* modifications of the tensor buffers. Fig. 3 (b) briefly describes the instruction set $\Theta^{H_{QKV}}$ of H_{QKV}.

Let's consider the abstract instruction load_rm in $\Theta^{H_{QKV}}$ that is parameterized by three attributes: $\mathsf{n} \in [1, 128]$, $\mathsf{addr}_{\mathsf{in}} \in [0, 2^{20})$, and $\mathsf{addr}_{\mathsf{out}} \in [0, 128)$. It is a data movement instruction that loads n rows of data ($128 \times \mathsf{n}$ bytes) from the HBM (d_0) starting at address $\mathsf{addr}_{\mathsf{in}}$, to the main scratchpad (d_1) starting at row $\mathsf{addr}_{\mathsf{out}}$. The concrete instruction $\mathsf{load\_rm}(\mathsf{n} = 4, \mathsf{addr}_{\mathsf{in}} = 0, \mathsf{addr}_{\mathsf{out}} = 2)$ is an instantiation of load_rm and reads 512 bytes from data slice $d_0[0{:}512]$ of tensor-type $\mathsf{u8}[512]$, to data slice $d_1[2{:}6]$ of tensor-type $\mathsf{bf16}[4, 64]$. Each bf16 value (2 bytes) is a bitwise cast of 2 contiguous u8 values (1 byte each). This transformation can be represented as a concrete tensor computation from $\mathsf{u8}[512]$ to $\mathsf{bf16}[4, 64]$ using tensor operators described in Table 1.

More generally, instruction semantics of load_rm can be represented as an abstract tensor computation $y = \mathsf{bitcvt}(\mathsf{reshape}(x_1))$ with the input and output tensor as data slices of d_0 and d_1, respectively. The addresses of these data slices are a function of attributes $\mathsf{addr}_{\mathsf{in}}$ and $\mathsf{addr}_{\mathsf{out}}$, and the number of rows loaded is a function of the attribute n. We classify the attributes into two sets $\alpha = \{\mathsf{n}\}$ and $\beta = \{\mathsf{addr}_{\mathsf{in}}, \mathsf{addr}_{\mathsf{out}}\}$, termed *computational* and *addressing* attributes, respectively. Using these observations, we define a generalized representation for instruction set semantics.

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

Kernel programming languages like Exo [31] model custom hardware instructions under similar assumptions, albeit defining semantics using scalar operators and Python-like syntax. Intuitively, we can represent scalar pseudocode descriptions with FOR and IF statements (like Intel Intrinsics Guide [33]) using element-wise tensor operators (since scalars are 0-D tensors) with while and select operators. We provide more detailed reasoning on generality in Appendix B. Fig. 15 shows that we can compactly represent real-world complex instruction semantics like Intel AMX instruction tdpbusd [33].

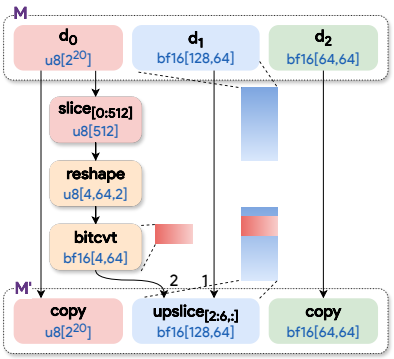


Fig. 4. Execution of concrete instruction load_rm(n = 4, addr$_{\text{in}}$ = 0, addr$_{\text{out}}$ = 2) as a concrete tensor computation graph.

Next, we formally define this generalized representation using notations defined previously.

Def 3.4. *An **abstract instruction** $\theta \in \Theta^H$ which reads n_θ tensors located at $d_\theta^{(i)}[I_s^{(i)}{:}I_e^{(i)}]|_{i=1}^{n_\theta}$ and modifies a tensor located at $d_\theta^{(0)}[I_s^{(0)}{:}I_e^{(0)}]$, is represented using an abstract tensor computation g_θ concretized by the set of computational attributes α, $(n_\theta + 1)$ data slice addresses $h_\theta^{(i)}|_{i=0}^{n_\theta}$ over the set of addressing attributes β and a validity constraint e_θ over the set of attributes α and β.*

Def 3.5. *A **concrete instruction** $\theta_{\alpha,\beta}$ represents a valid modification if $e_\theta(\alpha,\beta)$ is true. It reads data slices $x_i = d_\theta^{(i)}[I_s^{(i)}{:}I_e^{(i)}]|_{i=1}^{n_\theta}$ and modifies data slice $y = d_\theta^{(0)}[I_s^{(0)}{:}I_e^{(0)}]$ with the corresponding concrete tensor computation $y = f(x_i|_{i=1}^{n_\theta})$, where $f = g_\theta(\alpha)$ and $\forall i \in [0, n_\theta]$ $(I_s^{(i)}, I_e^{(i)}) = h_\theta^{(i)}(\beta)$.*

Def 3.6. ***Execution** $\mathcal{E}(\theta_{\alpha,\beta})$ of a concrete instruction $\theta_{\alpha,\beta}$ is a concrete tensor computation (over a tuple of tensors) that updates the memory state $M = (d_0, d_1, \dots)$ to $M' = (d'_0, d'_1, \dots)$ where*

$$d'_i = \begin{cases} \text{upslice}_{[h_\theta^{(0)}(\beta)]}\left(d_\theta^{(0)}, g_\theta(\alpha)\left(\text{slice}_{[h_\theta^{(i)}(\beta)]}(d_\theta^{(i)})\Big|_{i=1}^{n_\theta}\right)\right) & \text{for } d_i = d_\theta^{(0)} \text{ (d_i is the output buffer)} \\ \text{copy}(d_i) & \text{for } d_i \neq d_\theta^{(0)} \text{ (rest are not modified)} \end{cases}$$

Here, upslice, slice, & copy are tensor operators described in Table 1 and visualized in Appendix A.

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

3.4 Semantic Equivalence of Tensor Computation Graph and Compiled Assembly Code

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

Semantic equivalence of a compiled assembly code and a tensor computation graph G is defined over the output generated by the sequential execution of the concrete instructions and the golden output $G(X)$ computed by G for a given input tensors X. Since the multi-dimensional input and output tensors of G are stored on the HBM, which is a 1-D byte-addressable memory, we define a byte-encoding for tensors called byte-flatten or bflat, in short.

DEF 3.7 (**bflat**). *The byte-flattening operator* bflat *converts a tensor* t *of tensor-type* $\mathbb{E}[S]$ *to a 1-D tensor* $\text{bflat}(t)$*, which has the type* $u8[\text{mem}(t)]$*, where* $\text{mem}(t) = \text{mem}(\mathbb{E}) \times \prod S$ *and* $\text{mem}(\mathbb{E})$ *is the memory size (in bytes) of scalar basetype* $\mathbb{E}$.

Let's define the semantic equivalence intuitively as a sequence of events – (1) initialization of HBM with byte-flattened input tensors and the compile-time constant tensor in the assembly code, and (2) sequential execution of the concrete instructions in the assembly code based on the ISA description. The compiled assembly code is said to be semantically equivalent to the input graph G if, for all input tensors X, HBM holds the byte-flattened input tensors X and output tensor $G(X)$.

Next, we formalize this sequence of events as a transformation Λ_{LHS} of ASM_G^H and the expected state of HBM as a transformation Λ_{RHS} of G.

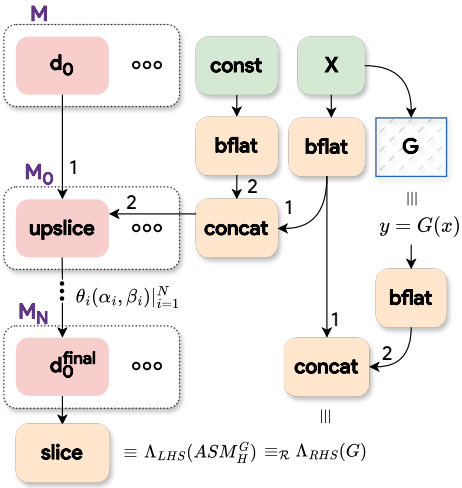


Fig. 5. Visualization of $\equiv_{\mathcal{R}}^H$. M and X are the inputs to the tensor computations. The leaf nodes represent the LHS and RHS of the semantic equivalence.

DEF 3.8 ($\boldsymbol{\Lambda_{LHS}}$). *Given an assembly code* $\text{ASM}_G^H = ((\theta_i(\alpha_i, \beta_i)|_{i=1}^N, const)$*,* $\Lambda_{LHS}(\text{ASM}_G^H)$ *is a concrete tensor computation over memory state* $M = (d_0, d_1, \dots)$ *and input tensors* X*. The memory state after initialization of HBM is* $M_0 = (\text{upslice}(d_0, \text{concat}_1(\text{bflat}(X), \text{bflat}(const))), d_1, \dots)$*. The final memory state is* $M_N = \mathcal{E}(\theta_N(\alpha_N, \beta_N)) \circ \cdots \circ \mathcal{E}(\theta_0(\alpha_0, \beta_0))(M_0)$ *with first* mem_{out} $(= mem(X) + mem(G(X)))$ *bytes of the final HBM* $(M_N[0])$ *relevant for semantic equivalence definition. Finally,* $\Lambda_{LHS}(\text{ASM}_G^H)(M, X) = \text{slice}_{[0:mem_{out}]}(M_N[0])$.

DEF 3.9 ($\boldsymbol{\Lambda_{RHS}}$). *Given an input graph* G*,* $\Lambda_{RHS}(G)(M, X) = \text{concat}_1(\text{bflat}(X), \text{bflat}(G(X)))$.

Finally, we define the semantic equivalence using these transformations.

DEF 3.10 ($\boldsymbol{\equiv_{\mathcal{R}}^H}$). *Given the foundational axioms* $\mathcal{R}$ *of the IR (see §2.1) and the ISA description* ISA^H*, the semantic equivalence* $\equiv_{\mathcal{R}}^H$ *is defined as:* $\text{ASM}_G^H \equiv_{\mathcal{R}}^H G \iff \Lambda_{LHS}(\text{ASM}_G^H) \equiv_{\mathcal{R}} \Lambda_{RHS}(G)$*, where* $\equiv_{\mathcal{R}}$ *is the semantic equivalence defined in §2.1.* Λ_{LHS} *and* Λ_{RHS} *are visualized in Fig. 5.*

3.5 Compiler Backend Generator

We now formally define the soundness and completeness of a compiler backend based on the formalisms introduced so far. A compiler backend Compiler^H consumes a tensor computation graph G and either emits an assembly code or returns FAIL, denoting a compilation error.

Intuitively, a compiler backend is said to be sound if, upon successful compilation, the assembly code is correct, i.e., semantically equivalent to the input graph. Note that a compiler backend that always fails to generate compiled code is trivially sound. A compiler backend is said to be complete if the compilation is successful for all input graphs that have an equivalent assembly code.

DEF 3.11 (**SOUNDNESS**). *A compiler backend* Compiler^H *is said to be sound iff*

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

DEF 3.12 (**COMPLETENESS**). *A compiler backend* Compiler^H *is said to be complete iff*

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

We solve for compiler backend generator CompGen such that $\forall \text{ISA}^H$, $\text{CompGen}(\text{ISA}^H)$ is *sound* and *complete*.

4 OVERVIEW OF COMPILER BACKEND GENERATOR 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. The key contribution of this work is a novel compilation algorithm parameterized by formal ISA constructs ($\mathrm{D}^H[d_i], \Theta^H[g_\theta, h_\theta, e_\theta]$) defined in §3. We outline the compiler backend generation process based on this parameterized algorithm in §4.1, and the high-level modular design of the algorithm in §4.2.

4.1 Overview of the Compiler Backend Generation Process

The design of ACT is guided by a theory-first philosophy, drawing inspiration from classic compiler component generators such as Flex [22] (for lexical analysis) and Yacc [37]/Bison [58] (for parsing). These tools consist of a core theoretical algorithm, such as NFA-to-DFA construction (Dragon Book [5] Chapter 3) in Flex or LALR(1) table generation (Dragon Book [5] Chapter 4) in Bison, that is generalized for a class of input grammars. This core algorithm is implemented as a generic skeleton and grammar-parameterized templates. For a given grammar, these templates are specialized to generate grammar-specific lexers and parsers. Table 2 summarizes this design methodology.

Similarly, ACT consists of a novel compilation algorithm parameterized by formal constructs defined in §3. We discuss this parameterized algorithm in §5-§7. This is implemented as a generic compiler backend with pure virtual functions and symbolic templates based on ISA constructs. At compiler backend generation time, a user specifies an accelerator ISA using a Python-based API that mirrors the formal ISA constructs. 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. 6. Note that a user only needs to provide an ISA description, rather than embedding ISA-specific logic in the compiler. This significantly reduces the engineering effort required to develop and maintain compiler backends for new accelerator ISAs.

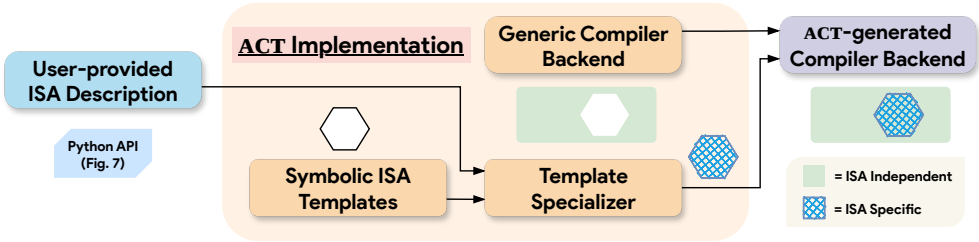


Fig. 6. Overview of ACT's compiler backend generation process. The orange region shows the implementation of the parameterized algorithm as a generic compiler backend (with missing holes) and symbolic ISA templates. At generation-time, these symbolic templates are specialized using the user-provided ISA description.

	Flex [22]	Bison [58]	ACT (Our work)
User-provided input	Regular expressions	Context-free grammar	ISA description
Generated output	Lexer/Scanner	LALR(1) Parser	Compiler backend
Core Parameterized Algorithm (Theory)	NFA-to-DFA construction [5]	LALR(1) table generation [5]	§5-§7, parameterized by ISA constructs (§3)
Generic Skeleton	Generic table-driven scanning loop	Generic shift-reduce parser loop	Generic backend with pure virtual functions
Specialized Templates	DFA Transition table with accept states	ACTION/GOTO tables with lookahead sets	Rewrites & overriden virtual functions
Specialization Approach	GNU M4 Macro processor	GNU M4 Macro processor	Python-based Regex processor

Table 2. Template-driven compiler component generators based on a parameterized algorithm.

```
1 qkv = Accelerator("QKV")
2
3 qkv.set_hbm("d0", size="1MB")                              # u8[2^20]
4 qkv.add_data_model("d1", S0=[128], S1=[64], E="bf16") # bf16[128,64]
5 qkv.add_data_model("d2", S0=[64], S1=[64], E="bf16")  # bf16[64,64]
6
7 instr = qkv.add_instr(name="load_rm", alpha=["n"], beta=["addr_in", "addr_out"])
8 instr.set_inputs([["d0", start=["addr_in"], len=["n * 128"]]])  # u8[n*128]
9 instr.set_output(["d1", start=["addr_out"], len=["n"]])         # bf16[n,64]
10 instr.set_constraints(["n <= 128"])
11 instr.set_abstract_computation("""
12 ENTRY load_rm {
13   x1 = u8[`n * 128`] parameter(0);
14   a = u8[`n`, 64, 2] reshape(x1);
15   ROOT y = bf16[`n`, 64] bitcast_convert(a);
16 }
17 """) # y = bitcvt(reshape(x1)) represented using XLA-HLO syntax
```

Fig. 7. Snippet of Python-based ISA description of H_{QKV} consumed by the compiler backend generator ACT.

Fig. 7 shows a snippet of the ISA description for H_{QKV} using ACT's Python API, defining the data model d_i (lines 3-5), and semantics $(g_\theta, h_\theta, e_\theta)$ of the abstract instruction load_rm (lines 7-17). ACT converts this information into rewrite rules and preconditions for use in equality saturation (more details in §5). It also generates implementations of abstract methods such as `get_h0()` and `get_e()` from symbolic templates, allowing constraint problem formulation (more details in §6) to operate purely through the formalisms defined in §3, without any hardcoded ISA assumptions. This separation ensures that compiler backends are derived systematically from just the ISA description.

4.2 Overview of the ISA-parameterized Compilation Algorithm

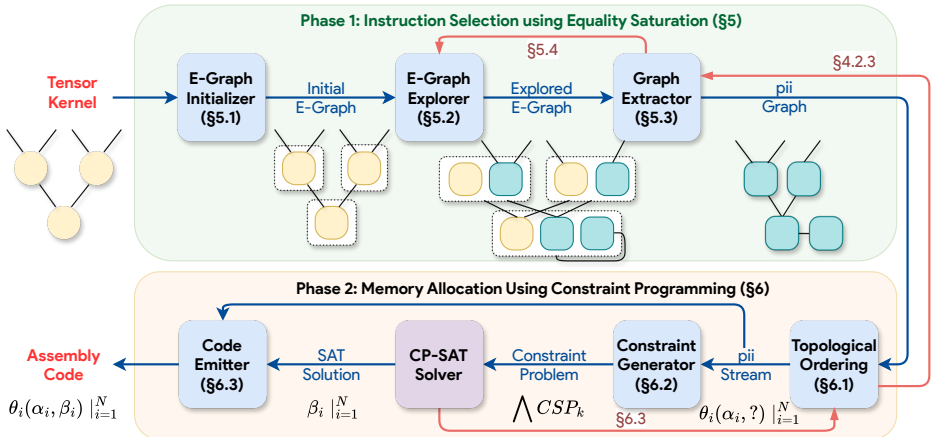


Fig. 8. Overview of ACT's parameterized compilation algorithm. Blue modules are generated by ACT from an accelerator ISA description (generation process in Fig. 6). The purple module is an external tool. The red edges represent the fallback mechanisms designed for completeness guarantee (§7) and enumeration pipeline (§8).

The ACT algorithm is designed as a modular pipeline, with each module focusing on a specific aspect of the compilation algorithm (see Fig. 8). The algorithms for each module are *parameterized* by the ISA description. In §5 and §6, we describe the algorithms for these modules using ISA formalisms defined in §3. In §7, we prove the soundness and completeness of the ACT algorithm. We use the step-wise lowering of G_{QKV} (Fig. 2) for ISA$^{H_{QKV}}$ (Fig. 3) as an illustrative example. In this paper, we focus on the novel features and relevant lemmas of ACT that address the challenges and goals mentioned in §1.2, leaving detailed discussions and proofs to Appendix C and D.

Next, we provide a high-level structure of the compilation algorithm visualized in Fig. 8.

The compiler backend Compiler^H compiles a tensor computation graph G into an accelerator-specific assembly code ASM_G^H. An accelerator-specific assembly code ASM_G^H consists of concrete accelerator instructions, i.e., abstract instructions in $\text{ISA}^{(H)}$ with initialized integer attributes α and β (as defined in §3.1). The compilation pipeline is divided into two phases - (1) initializing α, analogous to *instruction selection*, and (2) initializing β, analogous to *memory allocation*.

4.2.1 Instruction Selection. The first phase of Compiler^H consumes the input graph and generates a directed acyclic graph of *partially* instantiated accelerator instructions, also called a pii graph. A *partially* instantiated accelerator instruction is an abstract accelerator instruction with α initialized, i.e., of form $\theta_{\alpha,?}$. The first phase focuses on exploring semantically equivalent instruction selection opportunities. This is analogous to the Instruction Selection pass in compiler backends for commodity hardware. The phase is formulated as an equality saturation problem and utilizes an existing equality saturation engine (like egg [76]) to perform graph-level semantically-preserving substitutions. *Act automatically generates the three modules required for equality saturation – e-graph initializer, rewrite applier, and graph extractor – from the accelerator ISA description.*

4.2.2 Memory Allocation. The second phase of Compiler^H consumes the generated pii graph and generates the final assembly code. This phase focuses on initializing β, i.e., assigning addresses to the unmapped slices in the pii graph. This is analogous to the Register Allocation pass in compiler backends for commodity hardware. The phase adheres to three constraints – (1) maintain the instruction ordering, (2) assign non-overlapping address ranges to slices with overlapping live ranges, and (3) the mapped slices correspond to valid instruction attributes (α, β). These constraints are formulated as a constraint satisfaction problem over integers and then passed to an existing CP-SAT solver library (like Google OR-Tools [1]). The output of the constraint solver for a given ordering of pii graph is then used to emit the final assembly code. *Act automatically generates the three modules required for constraint satisfaction – topological numbering generator, integer constraint generator, and code emitter – from the accelerator ISA description.*

4.2.3 Inter-phase fallback mechanism. Unlike commodity hardware, accelerator designs have constraints not captured during instruction selection, like the constraints on the buffer mapping, lack of spill-reload instructions for some buffers, etc. Therefore, the second phase may fail to find a valid solution for a given pii graph. Compiler^H has a fallback mechanism to give control back to the first phase if the second phase fails. The first phase then extracts a different pii graph or performs another iteration of rewrites. This inter-phase fallback mechanism, along with intra-phase fallback mechanisms, discussed later in §5.4 and §6.4, play an important role in guaranteeing completeness. These fallback mechanisms are also used in the enumeration pipeline discussed in §8 (Fig. 14).

5 PHASE 1: TENSOR INSTRUCTION SELECTION USING EQUALITY SATURATION

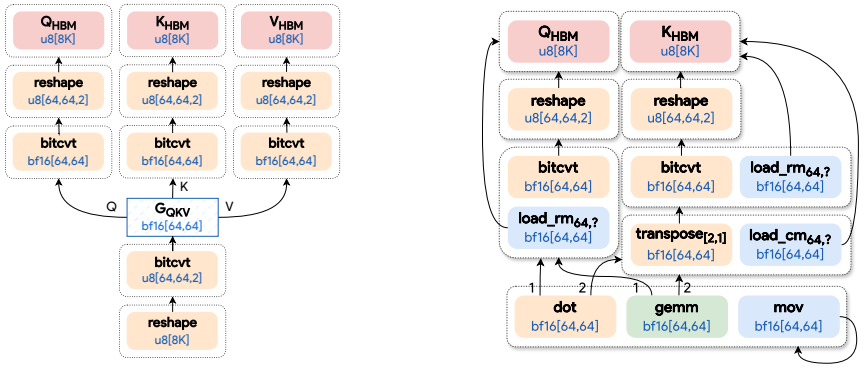
We formulate the tensor instruction selection problem as a search problem over the space of equivalent tensor computation graphs. We use e-graphs [66] to compactly represent this space of equivalent tensor computation graphs and enable efficient exploration for instruction selection.

Key novel features of the first phase are (1) leveraging the ISA formalized in terms of existing tensor operators to generate IR-to-ISA rewrite rules (§5.2.2), (2) modeling identity instructions (§5.2.3) to support tiling and padding, and (3) a novel graph extraction algorithm that detects compile-time constants (§5.3) and enumerates all equivalent pii graphs (§5.4).

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). An e-class is a set of e-nodes representing semantically equivalent computations rooted at the e-nodes. By design, each e-node is associated with a tensor buffer (if it is a pii $\theta_{\alpha,?}$) based on the output tensor buffer of the instruction ($d_\theta^{(0)}$ in Def. 3.4) or none

(if it is a concrete tensor operator). The e-graph is initialized with the input tensor computation graph G with byte-flattened input and output tensors. The leaf e-nodes are assigned to the tensor buffer d_0, which represents the HBM. Fig. 9 (a) shows the initial e-graph for G_{QKV} (Fig. 2) without expanding the sub-graph of G_{QKV}. Note that $\text{bflat}(t) = \text{reshape}(\text{bitcvt}(t))$ for $\mathbb{E} = \text{bfloat16}$.



(a) (b)

Fig. 9. (a) The initial e-graph for G_{QKV} [after Module 1]. G_{QKV} (Fig. 2) is not expanded here for compactness. Note that the edges are now flipped with the output tensor as the root of the e-graph. (b) Snippet of explored e-graph after applying rewrite rules on the initial e-graph [after Module 2]. Entire e-graph is in Appendix C.

5.2 Module 2: Rewrite Applier

The second module specifies the rewrite rules used in the exploration stage of equality saturation. The rewrite rules can be broadly categorized into: (1) subset of existing IR-to-IR rewrite rules tailored to the accelerator ISA description (§5.2.1), (2) new IR-to-ISA rewrite rules generated from the accelerator ISA description (§5.2.2), and (3) rewrite rules based on identity instructions, i.e., accelerator instructions that do not modify the memory state (§5.2.3).

5.2.1 IR-to-IR rewrite rules tailored to the accelerator ISA description. ACT uses the existing IR-to-IR rewrite rules (foundational axioms $\mathcal{R}$ of the IR discussed in §2.1) to explore the space of equivalent tensor computation graphs. It only considers rewrite rules over tensor operators appearing in the abstract tensor computations (g_θ). This filtering process is discussed in more detail in Appendix C.

5.2.2 IR-to-ISA rewrite rules. At generation-time, ACT generates new rewrite rules by reversing the instruction semantics. These IR-to-ISA rewrite rules create new pii nodes in the e-graph. For instruction θ, ACT generates the rewrite rule $g_\theta \rightarrow \theta$ that substitutes a concrete tensor computation $g_\theta(\alpha)$ with pii $\theta_{\alpha,?}$ under the precondition $\exists\beta$ such that $e_\theta(\alpha, \beta)$ is true. For example, rewrite rule $\text{bitcvt}(\text{reshape}(x)) \rightarrow \text{load\_rm}(x)$ is generated from lines 11-17 of Fig. 7. Fig. 9 (b) shows a snippet of the explored e-graph of Fig. 9 (a) with new pii nodes colored as blue (d_1) and green (d_2).

5.2.3 Identity instructions. We model identity instructions slice^H and concat^H that do not modify the memory state. These are semantically equivalent to the tensor operators slice and concat but are represented as pii nodes in the e-graph. These instructions are used to generate data slices of all sizes and are useful for tiling and padding tensor variables to match the instruction constraints.

5.3 Module 3: Graph Extractor

The third module performs the extraction stage of equality saturation. It traverses the e-graph explored by the second module and extracts directed acyclic graphs of piis (referred to as pii graphs).

First, a bottom-up traversal (from leaf e-classes) is performed to detect e-classes representing compile-time constants. The leaf e-nodes of flattened input tensor variables are marked as non-constant, and leaf e-nodes of tensor operators like *constant* and *eye* are marked as constant. An e-class is marked as constant if any one of the e-nodes has all its outgoing edges marked as constant.

Then, a top-down traversal is performed to extract the pii graphs. The traversal starts at the root e-class and terminates at the leaf e-classes. It is guided by the tensor buffers ($d_\theta^{(i)}|_{i=0}^{n_\theta}$) defined in instruction semantics (Def. 3.4). Given an e-class and a *def* node (θ_d), the traversal selects a *use* node (θ_u) such that its output tensor buffer ($d_{\theta_u}^{(0)}$) is same as the input tensor buffer ($d_{\theta_d}^{(i)}$) of the *def* node. For example, load_cm$_{64,?}$ is the *use* node selected for *def* node gemm in Fig. 9 (b) since $d_{\text{load\_cm}}^{(0)} = d_1 = d_{\text{gemm}}^{(2)}$. The traversal terminates upon reaching a flattened input tensor variable or a constant e-class. Detailed pseudocode for the above algorithm is present in Appendix C.

Fig. 10 (a) shows an extracted pii graph from the explored e-graph of G_{QKV}. In §9.2, we present a case study to show that detecting compile-time constant tensors increases compilation coverage.

5.4 Intra-phase fallback mechanism: Enumerating all equivalent pii graphs

The exploration and extraction stages of equality saturation may not terminate for all input graphs. The exploration stage iteratively applies the rewrite rules but may not reach saturation for all input graphs. The explored e-graph may have loops, and infinite equivalent pii graphs may exist. We use a bounded approach of graph extraction to enumerate equivalent pii graphs. The approach is based on increasing the maximum node limit N_{max} for the extraction stage with the iteration count k of the exploration stage, i.e., $N_{\text{max}} = \Gamma(k)$ 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 [11].

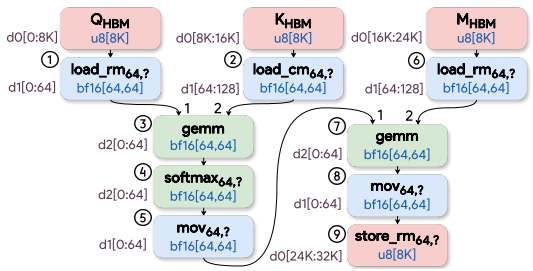
6 PHASE 2: MEMORY ALLOCATION USING INTEGER CONSTRAINT PROGRAMMING

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

Key novel features of the second phase are (1) a parameterized formulation of CSP that supports multi-dimensional buffers and access patterns (§6.2) and (2) a pruning technique that reduces the search space for correct assembly code while maintaining completeness (§5.4).

6.1 Module 4: Topological Ordering Generator

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 [62] generates optimal code for arithmetic binary trees with minimal registers. Module 4 extends this algorithm to multiple tensor buffers storing varying sizes of tensor variables. Details of this heuristic algorithm are present in Appendix C. Fig. 10 (a) shows the topological ordering of the extracted pii graph.



(a)

load_rm(n = 64, addr$_{\text{in}}$ = 0, addr$_{\text{out}}$ = 0)
load_cm(n = 64, addr$_{\text{in}}$ = 8192, addr$_{\text{out}}$ = 64)
gemm(addr$_1$ = 0, addr$_2$ = 64, addr$_{\text{out}}$ = 0)
softmax(n = 64, addr = 0)
copy(n = 64, addr$_{\text{in}}$ = 0, addr$_{\text{out}}$ = 0)
load_rm(n = 64, addr$_{\text{in}}$ = 16384, addr$_{\text{out}}$ = 64)
gemm(addr$_1$ = 0, addr$_2$ = 64, addr$_{\text{out}}$ = 0)
copy(n = 64, addr$_{\text{in}}$ = 0, addr$_{\text{out}}$ = 0)
store_rm(n = 64, addr$_{\text{in}}$ = 0, addr$_{\text{out}}$ = 24576)

(b)

Fig. 10. (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 $\text{ASM}_{G_{QKV}}^{H_{QKV}}$ [after Module 6].

6.2 Module 5: Constraint Satisfaction Problem Generator

By this stage, we have a pii graph and a topological ordering of piis $\theta^{(i)}_{\alpha^{(i)},?}|_{i=1}^{N}$. Next, Compiler^H has to instantiate the addressing attributes $\beta^{(i)}$ of the piis, i.e., allocate the intermediate and constant tensors on the tensor buffers. We formulate a constraint satisfaction problem for the interference graph and ISA-specific addressing constraints generalized for multi-dimensional tensor buffers.

Let's assume that the pii graph has N pii nodes (i.e., intermediate tensors and output tensor), N_V flattened input tensor variable nodes, and N_C constant nodes. The nodes are labeled as $V^{(i)}|_{i=1}^{N+N_V+N_C}$ with data slice addresses as $(I_s^{(i)}, I_e^{(i)})$. Note that the $V^{(i)}|_{i=N+1}^{N+N_V+N_C}$ are leaf nodes of the pii graph.

6.2.1 Constraint satisfaction problem formulation. The constraints over $\beta^{(i)}$ can be broken into four components: (1) pii node constraints based on $e_{\theta^{(i)}}$, (2) def-use edge constraints based on $h_{\theta^{(i)}}$, (3) input-output constraints based on input graph G, and (4) live range interference constraints.

CSP_1: *pii node constraints.* Every pii $\theta^{(i)}_{\alpha^{(i)},?}$ has a corresponding boolean expression $e_{\theta^{(i)}}$ and address concretizing function $h^{(0)}_{\theta^{(i)}}$. $\text{CSP}_1 = \wedge_{i=1}^{N}(e_{\theta^{(i)}}(\alpha^{(i)}, \beta^{(i)}) \wedge (I_s^{(i)}, I_e^{(i)}) = h^{(0)}_{\theta^{(i)}}(\beta^{(i)}))$

CSP_2: *def-use edge constraints.* For every def-use edge $(V^{(y)}, V^{(x_i)})$, the address concretizing function $h^{(i)}_{\theta^{(y)}}$ determine the data slice addresses of $V^{(x_i)}$, the i^{th} operand of the instruction $\theta^{(y)}$. $\text{CSP}_2 = \wedge_{(V^{(y)}, V^{(x_i)})}(I_s^{(x_i)}, I_e^{(x_i)}) = h^{(i)}_{\theta^{(y)}}(\beta^{(y)})$

CSP_3: *Input-Output constraints.* The addresses of the input and output tensor variables are predefined in §3.4. Let's say these addresses are $(I_s^{(G_{x_i})}, I_e^{(G_{x_i})})|_{i=1}^{N_V}$ and $(I_s^{(G_y)}, I_e^{(G_y)})$, respectively. $\text{CSP}_3 = (\wedge_{i=1}^{N_V}(I_s^{(N+i)}, I_e^{(N+i)}) = (I_s^{(G_{x_i})}, I_e^{(G_{x_i})})) \wedge (I_s^{(N)}, I_e^{(N)}) = (I_s^{(G_y)}, I_s^{(G_y)})$

CSP_4: *live range interference constraints.* Since the pii graph is a data dependency graph in itself, we can compute the live ranges without needing a fixed-point formulation. The live ranges are $[0, N]$ for input variable nodes, $[0, b-1]$ for constant leaf nodes, and $[a, b-1]$ for intermediate nodes, where a is the topological number of the node and b is the maximum of the topological numbers of the use nodes, i.e., incoming edges. We limit the live range to $b-1$ instead of b because the address constraints between def-use edges are already captured in the second component (CSP_2). The interference graph ϕ is generated with node pairs that are assigned the same buffer and have overlapping live ranges. $CSP_4 = \wedge_{(i,j)\in\phi}\,\text{disjoint}(i, j)$ where $\text{disjoint}(i, j)$ is the constraint expression for non-overlapping slices over $(I_s^{(i)}, I_e^{(i)})$ and $(I_s^{(j)}, I_e^{(j)})$.

6.2.2 Solving the constraint satisfaction problem. The four generated constraints are combined as $\text{CSP} = \text{CSP}_1 \wedge \text{CSP}_2 \wedge \text{CSP}_3 \wedge \text{CSP}_4$. This constraint problem is then passed to an existing CP-SAT solver library to find a satisfying solution over the addressing attributes $\beta^{(i)}$. Fig. 10 (a) shows the data slice addresses assigned to pii nodes as determined by solving the constraint problem.

6.2.3 Implementation of Module 5. Let's look at a concrete example of template-driven implementation (discussed in §4.1) of the *parameterized* algorithm of Module 5. It consists of a generic skeleton (Fig. 11) with symbolic templates (Fig. 12) specialized at generation-time (Fig. 13).

```
void CSP1(const std::vector<INSTR*>& instructions) {
  for (const auto& instr : instructions) {
    solver.AddConstraint(instr->get_e());
    auto h0 = instr->get_h0();
    solver.MakeEquality(instr->output->start_addr, h0.first);
    solver.MakeEquality(instr->output->end_addr, h0.second);
  }
}
```

Fig. 11. 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. 12. Symbolic Template for ISA construct $h_\theta^{(0)}$.

```
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. 13. Specialized Template for $h_{\text{load\_rm}}^{(0)}$ generated by processing line 9 of Fig. 7.

6.3 Module 6: Code Emitter

By this stage, we have ordered piis $\theta^{(i)}_{\alpha^{(i)},?}$ and computed their addressing attributes $\beta^{(i)}$, giving us a stream of concrete instructions $\theta^{(i)}(\alpha^{(i)}, \beta^{(i)})$. The sixth module emits the assembly code by discarding identity instructions (since they are no-ops) and concatenating all constant tensors in the pii graph into one. Fig. 10 (b) shows the compiled assembly code for the running example G_{QKV}.

6.4 Intra-phase fallback mechanism

6.4.1 Limitation of Sethi-Ullman-based Topological Ordering Heuristic. The depth-first topological ordering generated by the heuristic-based algorithm (§6.1) is not sufficient for the completeness of Phase 2. We present a counterexample in Appendix C. The tensor instructions in the instruction set Θ^H 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. Thus, we need to search through all topological orderings for completeness, with a complexity bound of $O(N!)$.

6.4.2 Pruned search space of topological orderings. In practice, we observe the number of topological orderings to be small. The pii graph in Fig. 10 has 9 pii nodes but only $12 (\ll 9!)$ topological orderings. These 12 topological orderings only result in 2 unique interference graphs. CSP_4 is monotonic w.r.t. the interference graph, i.e., $\phi_1 \subseteq \phi_2 \implies \neg\text{CSP}_4(\phi_1) \rightarrow \neg\text{CSP}_4(\phi_2)$. Therefore, we discard any topological ordering with a sub-interference graph of a previously failed topological ordering.

7 THEORETICAL GUARANTEES OF ACT-GENERATED COMPILER BACKENDS

We provably guarantee that Act generates *sound* and *complete* compiler backends, stated as theorem:

Theorem 1. $\forall \text{ISA}^H$, $\mathit{Act}(\text{ISA}^H)$ *is sound (Def. 3.11) and complete (Def. 3.12).*

We break the proof of Theorem 1 into multiple simpler lemmas stated and proven in §7.2 and §7.3.

7.1 Novel intermediary pii graph

First, we formally define the novel intermediary pii graph introduced in §4.2.

Def 7.1. *A **pii graph** PG^H is a directed acyclic graph of N pii nodes $\theta_i(\alpha_i, ?)|_{i=1}^N$, and it can be transformed into a concrete tensor computation using substitutions $\theta_i(\alpha_i, ?) \rightarrow g_{\theta_i}(\alpha_i)$. Note that pii nodes can be $\theta_{\alpha,?}$ from the instruction set Θ^H, or identity instructions $\text{slice}^H_{\alpha,?}$ & $\text{concat}^H_{\alpha,?}$ (§5.2.3).*

Def 7.2 ($\equiv^H_{\mathcal{R}}$). *Semantic equivalence between pii graph PG^H and input graph G is defined as: $\text{PG}^H \equiv^H_{\mathcal{R}} G \iff \Lambda_{pii}(\text{PG}^H) \equiv_{\mathcal{R}} \Lambda_{RHS}(G)$, where $\Lambda_{pii}(\text{PG}^H)(M, X) = \text{concat}(\text{bflat}(X), \text{PG}^H(X))$. Similarly, $\text{ASM}^H_G \equiv^H_{\mathcal{R}} \text{PG}^H \iff \Lambda_{LHS}(\text{ASM}^H_G) \equiv_{\mathcal{R}} \Lambda_{pii}(\text{PG}^H)$. Λ_{LHS} and Λ_{RHS} are defined in §3.4.*

7.2 Theoretical Guarantees of Phase 1

Next, we state the provable properties of the first phase, i.e., tensor instruction selection (§5).

LEMMA 1. *i^{th} pii graph enumerated is equivalent to input graph G, i.e.,* $\forall \boldsymbol{i}$, $\mathbf{Phase_1}(G)[\boldsymbol{i}] \equiv^H_{\mathcal{R}} G$.
This follows from the semantic validity of the rewrite rules used in Module 2. IR-to-IR rewrites are a subset of the foundational axioms $\mathcal{R}$. IR-to-ISA rewrites are derived from the ISA description (g_θ). Since the rewrites are semantic-preserving, extracted pii graphs are equivalent to initial e-graph.

LEMMA 2. *All equivalent pii graphs are enumerated, i.e.,* $\forall \mathbf{PG}^H \equiv^H_{\mathcal{R}} G$, $\exists \boldsymbol{i}$ *s.t.* $\mathbf{Phase_1}(G)[\boldsymbol{i}]{=}\mathbf{PG}^H$.
This follows from the bounded approach to graph extraction (§5.4) that interleaves exploration and extraction stages. If there exists an equivalent pii graph with N_0 nodes explored after k_0 iterations, it will be one of the graphs extracted at iteration $k' = \max(k_0, k_1)$, where $k_1 = \text{argmin}_k(N_0 \leq \Gamma(k))$.

7.3 Theoretical Guarantees of Phase 2

Next, we state the provable properties of the second phase, i.e., memory allocation (§6).

LEMMA 3. *The final assembly code is equivalent to the input pii graph, i.e.,* $\mathbf{Phase_2}(\mathbf{PG}^H) \equiv^H_{\mathcal{R}} \mathbf{PG}^H$.
This follows from the sound construction of the constraint problem in Module 5. Alternatively, this is proven by transforming $\Lambda_{LHS}(\text{ASM}^H_G)$ into $\Lambda_{pii}(\text{PG}^H)$ by iteratively applying the rewrite rule $\text{slice}_{[s_1:e_1]}(\text{upslice}_{[s_2:e_2]}(M, X)) \rightarrow \text{slice}_{[s_1:e_1]}(M)$ under the precondition $\text{disjoint}([s_1{:}e_1], [s_2{:}e_2])$. This rewrite rule has been verified in its full generality using TensorRight [8].

LEMMA 4. *Phase 2 does not fail if there is an equivalent assembly code (subject to the completeness of the CP-SAT solver, an external tool), i.e.,* $\exists \mathbf{ASM}^H_G \equiv^H_{\mathcal{R}} \mathbf{PG}^H \implies \mathbf{Phase_2}(\mathbf{PG}^H) \neq \mathbf{FAIL}$.
As discussed in §6.4, Phase 2 considers all possible instruction orderings, thereby covering the orderings of ASM^H_G. Since there exists a solution to the constraint problem for this ordering, we can guarantee that Phase 2 will not fail (subject to the limitations of the CP-SAT solver, if any).

LEMMA 5. *Every assembly code has an equivalent pii graph, i.e.,* $\forall \mathbf{ASM}^H_G$, $\exists \mathbf{PG}^H$, $\mathbf{ASM}^H_G \equiv^H_{\mathcal{R}} \mathbf{PG}^H$.
We prove this by constructing an equivalent pii graph from $\Lambda_{LHS}(\text{ASM}^H_G)$. As defined in Def. 3.6 and Def. 3.8, $\Lambda_{LHS}(\text{ASM}^H_G)$ consists of g_θ sub-graphs and bflat, slice, & upslice tensor operator nodes. The bflat nodes match those in Λ_{pii} and thus can be ignored. g_θ sub-graphs are replaced by the appropriate pii nodes of form $\theta_{\alpha,?}$. slice nodes are replaced with slice^H nodes, and upslice nodes are broken into slice^H & concat^H nodes using the rewrite rule $\text{upslice}_{[s:e]}(M, X) \rightarrow \text{concat}(\text{concat}(\text{slice}_{[0:s]}(M), X), \text{slice}_{[e:]}(M)))$. The generalized version of this rewrite rule has been verified using TensorRight [8]. Note that the constructed pii graph satisfies its Def. 7.1.

7.4 Proving Soundness and Completeness

Finally, we prove Theorem 1 using lemmas stated in §7.2 and §7.3.

Soundness of ACT-generated compiler backends follows from Lemma 1 and Lemma 3 which show that both phases of the compilation algorithm are sound, in short, $\text{ASM}^H_G \equiv^H_{\mathcal{R}} \text{PG}^H \equiv^H_{\mathcal{R}} G$.

Completeness of ACT-generated compiler backends follows from (1) Lemma 5 (every assembly code has an equivalent pii graph), (2) Lemma 2 (Phase 1 enumerates all equivalent pii graphs), and (3) Lemma 4 (Phase 2 doesn't fail if there is an equivalent assembly code). Combining these lemmas, we can conclude that if there exists an equivalent assembly code $\text{ASM}^H_G \equiv^H_{\mathcal{R}} G$, then there exists an equivalent pii graph (from Lemma 5) which is enumerated by Phase 1 (from Lemma 2) and also passes Phase 2 (from Lemma 4), i.e., graph G is compiled successfully ($\text{Compiler}^H(G) \neq \text{FAIL}$).

8 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.is_saturated: # §5.4
4     egraph = egraph.applier(k)   # Module2
5     piis = egraph.extract(Γ(k))  # Module3
6     for pii in piis:             # §4.2.3
7       schedules = topo(pii);     # Module4
8       for schd in schedules:     # §6.4
9         csp = csp_gen(schd);     # Module5
10        sol = solve(csp)         # CP-SAT
11        if sol.SAT:
12          asm = emit(pii, sol)   # Module6
13          yield asm                # Enumerate
```

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

Fig. 14. Pseudocode for code generator using cost models to generate performant code

The code generator (Fig. 14) 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. 14 (b) shows the pseudocode of the enumerator function based on the Act algorithm (described in Fig. 8) with yield statement[2] at line 11. 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. 14 (a) shows the code generator that iterates through the candidate assembly codes (line 4) enumerated by the Act-generated compiler backend (Fig. 14 (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. 14 (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 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 §9.3, we show that the code generators match the performance of state-of-the-art kernel libraries like the oneDNN library, with a maximum observed compilation time of 66 ms.

Choice of Cost Model. Performance statistics can be collected using different fidelity of cost models, like cycle-accurate timing simulators, analytical cost models, and learned cost models. Note that the enumeration pipeline and the Act algorithm are *agnostic to the choice of cost model.*

9 EVALUATION

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

- **Case Studies (§9.2):** We studied the role of key features in Act in modeling complex and parameterized ISA (Challenge 2) and increasing compilation coverage (Goal 1).
- **Performance Comparison (§9.3):** We evaluated the performance of the compiled code for kernels supported by custom kernel libraries and compositions of such kernels.
- **Compilation Time Analysis (§9.4):** We performed breakdown analyses of the compilation time for varying tensor kernel sizes from synthetic and real-world workloads.

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

9.1 Evaluation Setup

We used ACT to generate compiler backends for various tensor accelerator ISAs. For evaluation, we selected three tensor accelerator designs – a commercial accelerator, Intel AMX [32], an academic accelerator design, Gemmini [26], and an accelerator H_{QKV} generated using an accelerator design language (ADL), Allo [12]. We selected these designs based on the availability of well-defined ISA semantics and open-source simulation platforms for testing compiled assembly code. Table 3 summarizes the three accelerators and their key features.

	H_{QKV} (Fig. 3 (a))	Gemmini [26]	Intel AMX [32]
Hardware Design	Using Allo [12]	Open-source	Proprietary
ISA	Fig. 3 (b)	[27]	[33]
Data Model	Scratchpads	Scratchpads	Register Files
#Instructions Modeled	7	11	4 AMX + 24 AVX512
Testing Platform	Xilinx FPGA	RTL simulator	Sapphire Rapids CPU
Parameterized ISA (with computational attributes α)	Yes	Yes	No
Reconfigurable Dataflow	No	Yes	No
Complex Layout Transformations	No	No	Yes

Table 3. Summary of the three tensor accelerators used in the evaluation.

(i) Allo-generated design for H_{QKV}

We generated a hardware design in Verilog based on the hypothetical QKV accelerator described in Fig. §3. The GeMM and softmax activation units are designed using Allo [12], an accelerator design language (ADL). We tested the compiled code using FPGA emulation of the Allo-generated design.

(ii) Gemmini

Gemmini [26] is a parameterized tensor accelerator with a software-controlled systolic array (similar to many TPU-like architectures). The systolic array is used to perform matrix multiplication of 16×16 matrices and is reconfigurable to support different dataflows (output- or weight-stationary).

ISA. The data model includes two scratchpads - spad and acc to store i8 and i32 matrices, respectively. spad consists of $16K$ rows of i8[16] elements, and acc consists of $1K$ rows of i32[16] elements. The ISA includes data movement instructions mvin and mvout for loading and storing data from/to the scratchpads, and compute instructions for matrix multiplication with and without bias addition. We modeled these instructions based on the Gemmini GitHub repository [27].

ACT-Gemmini. The ACT-generated compiler backend for the Gemmini ISA, with a simple analytical cost model based on the sum of per-instruction costs, is used to generate the final assembly code (see §8). We refer to this as ACT-Gemmini and used it to compile XLA-HLO kernels to Gemmini instructions. We tested the compiled code using Gemmini's Verilator-based RTL simulator [27, 63].

(iii) Intel AMX

Intel® Advanced Matrix Extensions (Intel® AMX) is a built-in accelerator with 2-dimensional registers (tiles) and a Tile Matrix Multiplication (TMUL) engine. Intel AMX is supported by Intel Sapphire Rapids processors, which also support AVX512 instructions.

ISA. The data model includes two register files - 8 tmm registers and 32 zmm registers to store i8[16,64] matrices and 512-bit vectors, respectively. We model the 4 AMX and 24 AVX512 instructions based on the Intel ISA documentation [33].

ACT-AMX. The ACT-generated compiler backend for the Intel AMX+AVX512 ISA, with a simple analytical cost model based on the sum of per-instruction costs, is used to generate the final assembly code (see §8). We refer to this as ACT-AMX and used it to compile XLA-HLO kernels to AMX & AVX512 instructions. We tested the compiled code using a Intel Xeon Gold 5415+ CPU.

9.2 Case Studies

We studied the role of key features in Act in modeling complex and parameterized ISA (Challenge 2) and increasing compilation coverage (Goal 1).

Case Study 1: Parameterized ISA
Act models parameterized accelerator instructions like load_rm, which takes the number of rows n as a computational attribute. The running example (Fig. 9-10) shows the step-wise compilation of a simple QKV computation by the Act-generated compiler backend. The simulation result for the compiled assembly code in Fig. 10 (b) matched the golden output for G_{QKV} (Fig. 2).

Case Study 2: Complex Layout Transformations
In §3.3, we formalized a generalized representation for instruction semantics of tensor accelerator ISAs using tensor operators. This helps us model instructions with complex layout transformations like Intel AMX tdpbusd [33] (visualized in Fig. 15) using a series of reshape and transpose operators (purple region).

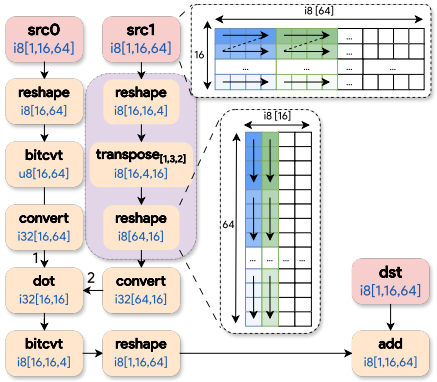
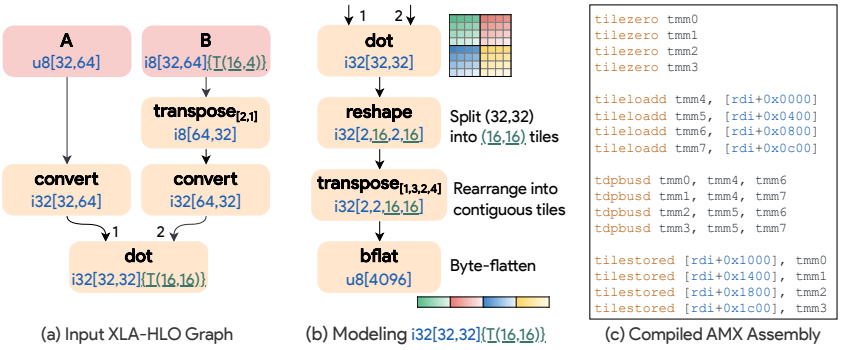


Fig. 15. Highlighted region shows the layout transformation applied to tile register src1 in AMX instruction tdpbusd(dst, src0, src1) [32].

Case Study 3: XLA-HLO Tiled Memory Layout
XLA compiler often stores tensors in tiled memory format $\mathbb{E}[S_t]\{\mathsf{T}(S_T)\}$, where a tensor of type $\mathbb{E}[S_t]$ is broken down into tiles of shape S_T that are each stored in a contiguous fashion. Fig. 16 (a) shows the XLA-HLO IR for a tensor kernel commonly observed in oneDNN examples (later referred to as K1 in Table 4). The kernel consists of a matrix multiplication with the input tensor B and the output tensor stored with tiled memory layouts. Act models tiled memory layouts in the e-graph initializer (§5.1) using a series of reshape and transpose operators alongside the bflat operator (Def. 3.7). For example, tiled memory layout i32[32,32]{T(16,16)} is modeled in Fig. 16 (b). Act-AMX applied 100+ IR-to-IR rewrites to compile this kernel into Fig. 16 (c). This feature makes Act-generated compiler backends compatible with the XLA compiler [18].



(a) Input XLA-HLO Graph (b) Modeling i32[32,32]{T(16,16)} (c) Compiled AMX Assembly

Fig. 16. (a) XLA-HLO IR with tiled memory layouts for the input tensor B and the output tensor. (b) Tiled memory layout i32[32,32]{T(16,16)} as modeled by Act. (c) The final assembly code generated by Act-AMX.

Case Study 4: Increased compilation coverage with compile-time constant tensors
Compile-time constant tensors are detected during the extraction stage of equality saturation using a bottom-up traversal (from leaf e-nodes) (see §5.3). This allows Act-generated compiler backends to enumerate assembly code requiring additional constant memory initialization. While these assembly codes consume extra stack memory, they are often significantly faster than executing the kernel

on the host processor. Fig. 17 shows pii graphs with constant memory (purple region) generated by ACT-Gemmini. This feature of ACT plays an important role in guaranteeing completeness and improves the compilation coverage of the compiler backend by minimizing host fallback.

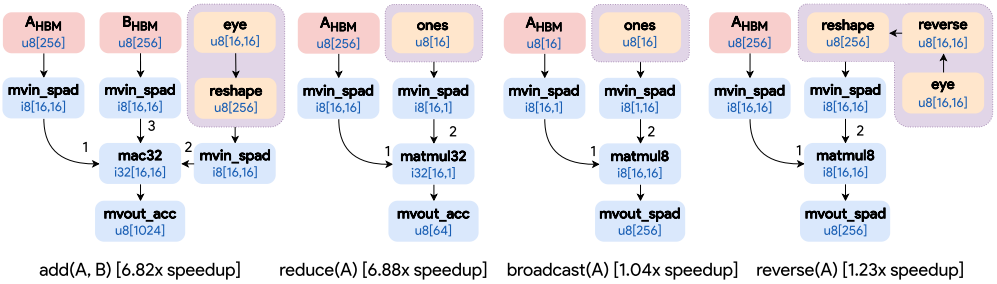


Fig. 17. pii graphs with constant memory (purple region) generated by ACT-Gemmini. The orange nodes represent the value of the constant memory. The speedup is w.r.t. the Gemmini host processor (Rocket chip). Tensor operator nodes eye and ones represent an identity matrix and a vector with all 1, respectively.

9.3 Performance Comparison

We evaluated the performance of the compiled code by ACT-AMX and ACT-Gemmini against custom hand-optimized kernel libraries – oneDNN library, Gemmini SW library, and Exo library.

Kernel Libraries

oneDNN Library. The oneDNN library consists of a set of common DNN kernels and is hand-written and heavily optimized for Intel ISA, including AMX and AVX512. XLA compiler generates fused subgraphs for these kernels (like Fig. 16 (b)), which are then compiled to assembly code using the oneDNN library (xla_cpu_use_mkl_dnn flag). We selected five such kernels commonly observed in oneDNN examples [35]. These kernels are based on simple matrix computations with complex memory layouts of the input matrices. Table 4 summarizes the statistics for the selected kernels labeled as K1-K5. Fig. 16 shows the input kernel and the final assembly code for K1. The rest are presented in Appendix E.

	#Nodes	#AMX	#AVX512
K1	6	16	0
K2	6	9	0
K3	7	0	100
K4	7	0	107
K5	23	16	37

Table 4. Statistics of selected oneDNN kernels and their optimized assembly code in oneDNN library

Gemmini SW Library. The Gemmini architecture ships with a custom kernel library for common DNN kernels that have been handwritten and optimized by Gemmini's designers. The Gemmini SW library supports Multiply-Accumulate (MAC: $A \times B + C$), Matrix Multiply (GEMM: $A \times B$), and Element-wise Addition (ADD: $A + B$), where A, B and D are i8 matrices.

Exo Library. Exo [31] is a kernel programming DSL based on an orthogonal paradigm of exo-compilation, i.e., the user writes both the input program and its schedule. This includes specifying tile sizes (line 106 of [3]), loop transformations (line 157 of [3]), instruction selection (line 149 of [3]), and buffer selection (line 207 of [3]). 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.

Benchmarks

The benchmarks are categorized into two classes – (i) kernels supported by the custom kernel libraries (labeled as "supported") and (ii) compositions of such kernels (labeled as "composite").

(i) "supported" kernels. We evaluated ACT-AMX on (1) the five selected oneDNN kernels (Table 4) against expert-written assembly code in the oneDNN library. We evaluated ACT-Gemmini on (2)

[3]https://github.com/exo-lang/exo/blob/main/src/exo/platforms/gemmini.py

the six shapes of GEMM (labels in Fig. 18) against hand-tuned Exo schedules, and (3) the three kernels MAC, GEMM, ADD on a single tile (tensor-type i8[16,16]) against Gemmini SW Library.

(ii) "composite" kernels. We evaluated Act-Gemmini on nine compositions of GEMM (×) and ADD (+) on a single tile (tensor-type i8[16,16]). These compositions are a fusion of 2-3 Gemmini SW library kernels. The baseline is the minimum cycle count of all equivalent "supported" kernel orderings (for example, $A \times (B \times C)$ and $(A \times B) \times C$ for $A \times B \times C$). Further, we analyzed the data movement between the host memory and the Gemmini accelerator scratchpads.

Evaluation methodology

The oneDNN library and Gemmini SW library use a fixed tile size across their respective benchmarks. For fair comparison, Act-AMX and Act-Gemmini use the same tiling factors as the baselines.

On the other hand, hand-tuned Exo schedules include middle-end and backend optimizations. Act focuses exclusively on the backend passes, delegating all middle-end optimizations to XLA's autotuner [56]. This separation allows Act to leverage XLA's search over tile sizes and loop transformations without burdening the user to manually set these. For fair comparison, we execute Act-Gemmini across a spectrum of tile sizes and loop fission factors to search for the best candidate. This search is fully automated, since Act-Gemmini can compile kernels with different tile sizes.

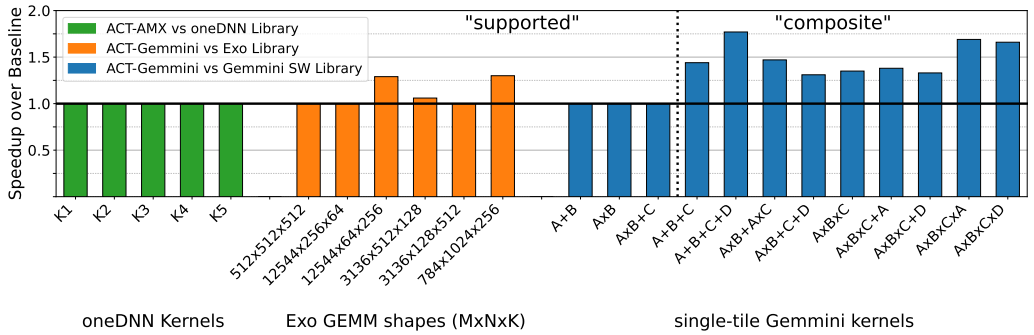


Fig. 18. Speedup of compiled code by Act-AMX and Act-Gemmini over kernel libraries (Higher is better).

Results

Fig. 18 shows the speedup of compiled code by Act-AMX and Act-Gemmini over the custom kernel libraries. Fig. 19 analyzes the data movement for "composite" kernels. Final assembly codes by Act-AMX and Act-Gemmini *match* the performance of oneDNN and Gemmini SW library, respectively, for "supported" kernels (i.e., speedup of **1x**). Final assembly codes by Act-Gemmini with XLA autotuning *match or outperform* Exo library with a speedup of up to **1.30x** (geomean of 1.1x). We observe that Act-Gemmini *outperforms* the baseline for "composite" kernels with a speedup of up to **1.77x** (geomean of 1.48x) and reduces the data movement by up to **55.5%** (average of 40%).

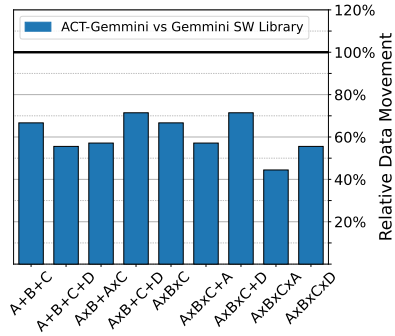


Fig. 19. Relative data movement between host memory and Gemmini scratchpads for "composite" kernels (Lower is better).

Observations

(i) "supported" kernels. Act-generated compiler backends enumerate the hand-optimized assembly code from custom kernel libraries, by virtue of the completeness guarantee. As the kernel libraries are heavily optimized for the "supported" kernels, the final assembly codes selected by the enumeration pipeline (§8) were the same for all kernels (except three shapes of Exo kernels).

For example, the ACT-AMX compiles the oneDNN kernel K1 into Fig. 16 (c), which is identical to the corresponding oneDNN assembly [34]. For the three GEMM shapes where ACT-Gemmini outperforms Exo library, we observed that ACT-Gemmini used a better loop fission factor.

(ii) "composite" kernels. Gemmini SW library misses out on reuse opportunities for "composite" kernels, incurring extra data movement. On the other hand, ACT-Gemmini minimizes data movement by storing intermediates on the accelerator scratchpads. Fig. 20 shows the breakdown for Gemmini SW library and ACT-Gemmini compiled code for kernel $A \times B \times C$. ACT-Gemmini eliminates the redundant load-store instructions (highlighted in red) across library calls. As a result, ACT-Gemmini has a speedup of **1.35x** with **33.3%** reduction in data movement for a single tile. This increases with the scratchpad utilization and reaches a speedup of **1.48x** with **50%** reduction in data movement at maximum utilization (with A of tensor type i8[5540,16]). Similar observations were made for other "composite" kernels. We also observed that speedup increases as the kernel size (# nodes) increases. 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.

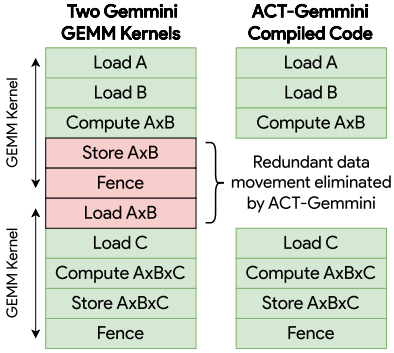


Fig. 20. Breakdown of Gemmini SW library kernels and ACT-Gemmini compiled code for $A \times B \times C$. ACT-Gemmini eliminates redundant load-store instructions.

ACT-generated compiler backends generate performant assembly code that matches or outperforms the state-of-the-art kernel libraries.

9.4 Compilation Time Analysis

In §9.3, we discussed tensor kernels supported by custom kernel libraries and their compositions. ACT-generated compiler backends took up to 66 ms (average 45 ms) to generate the performant assembly code for all these benchmarks. However, these tensor kernels consist of up to 10 nodes (except K5). Typical tensor kernels after graph partitioning contain 10-50 nodes, as evident in the TPUGraphs [55] Tile dataset. Therefore, to analyze the compilation time for larger kernel sizes, we performed two sets of experiments - (i) fuzz testing using synthetic tensor kernels (size 7-89 nodes) randomly generated using NNSmith [45], (ii) stress testing using the three largest tensor kernels present in the TPUGraphs [55] Tile dataset (150-390 nodes).

Evaluation methodology

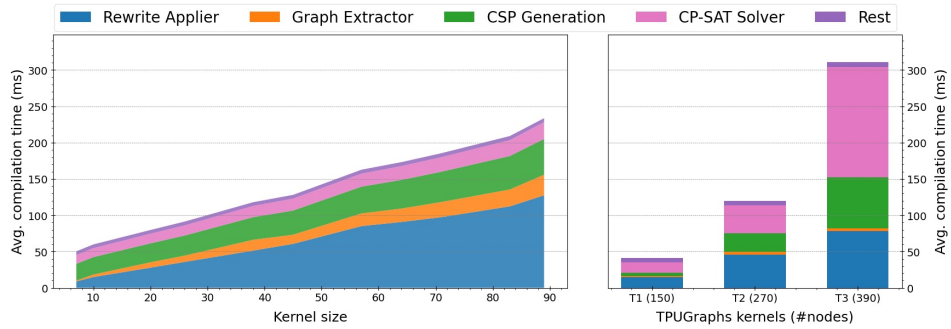
The compilation time reported is final.time in Fig. 14 (a), recorded using a 20-core Intel i7-13700H CPU, averaged over 20 trials. The enumeration pipelines (§8) for ACT-AMX and ACT-Gemmini were set to an overall timeout after 5 seconds, which was never reached in any of the trials.

(i) Fuzz testing ACT-Gemmini using NNSmith [45]

NNSmith [45] is a random DNN generator and fuzzing infrastructure, designed for automatically validating deep-learning (or tensor) compilers. We used NNSmith [45] to generate 100 synthetic tensor kernels consisting of 10 XLA-HLO operators (Appendix D). The kernel size varied uniformly from 7 to 89 nodes. The Gemmini kernels used in §9.3 are all smaller than these synthetic kernels.

(ii) Stress testing ACT-AMX using TPUGraphs [55] Tile dataset

For stress testing, we used the three largest tensor kernels in TPUGraphs [55] Tile dataset - T1, T2, T3 with 150, 270, 390 nodes, respectively. T1 & T3 are from an RL workload, Brax ES [24], and T2 is from a Transformer workload, Trax LSH [28]. These kernels are computations over u64 tensors.



(a) Fuzz testing ACT-Gemmini (b) Stress testing ACT-AMX

Fig. 21. Breakdown of compilation time. (a) ACT-Gemmini on synthetic kernels generated using NNSmith [45] and (b) ACT-AMX on the three largest kernels in TPUGraphs [55] Tile dataset.

Discussion

Fig. 21 shows the breakdown of compilation time for fuzz testing and stress testing. We observed small compilation times for a wide range of kernel sizes (up to 233 ms) and even at extremely large kernel sizes (up to 311 ms). 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. 22 shows a simple snippet of an e-graph with cycles representing infinite equivalent assembly code.
- Bounded approach of graph extraction (see §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 (see §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 extremely large pii graphs (151 ms for 390 nodes).

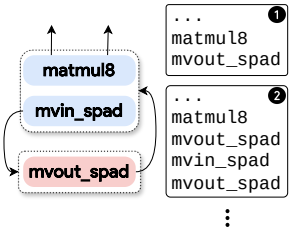


Fig. 22. Snippet of e-graph (left) with cycles between mvin and mvout e-classes. This represents infinite equivalent candidates (right), and the candidate with no redundant data movement is enumerated first.

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

10 RELATED WORKS

Vectorizer generators and synthesis-based compilation. Vectorizer generators such as Vegen [16], Diospyros [71], and Isaria [67] target CPUs like x86 and DSPs like Hexagon HVX with fixed-length vector instructions. Synthesis-based instruction selectors such as Hydride [40] and Rake [3] 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. Further, prior works cannot directly handle the complex parameterized instructions present in tensor accelerator ISAs.

Automated compiler backend generation. Prior works [10, 20] have explored generation of instruction selection rules via custom DSLs and CEGIS from CPU ISA semantics. They focus on many-to-one matching to find algebraic equivalences of arithmetic instructions. 3LA [29] 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 to optimize scratchpad reuses. In contrast, ACT focuses on automatically generating a complete compiler backend just from the accelerator ISA description.

Kernel programming languages like Exo [31], Pallas [54], Triton [68], and NKI [51] 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 [77], SPORES [75], and Cranelift [23] 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 generated IR-to-ISA rewrites to concurrently explore instruction selection opportunities. MISAAL [52] uses equality saturation to lower Halide IR to AutoLLVM IR (vector ISAs) using synthesized rewrite rules. However, it only synthesizes rewrite rules without preconditions, which are insufficient to model parameterized instructions of tensor accelerator ISAs (§5.2.2) during e-graph exploration. Furthermore, all these works use a local cost function to select the "best" graph from the explored e-graph. In contrast, ACT selects instruction nodes based on matching tensor buffers and enumerates all equivalent graphs for completeness.

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 [50]. But unlike our problem, these algorithms minimize the critical path length rather than the interference graphs. The Sethi–Ullman algorithm [62] generates optimal code for arithmetic expressions using minimal registers. Prior works have generalized this to 1-load binary DAGs [13] and non-binary trees [7]. However, any depth-first traversal of the expression tree is not always optimal for more than one register file (counterexample in Appendix C). We extend the Sethi-Ullman algorithm to multiple multi-dimensional tensor buffers with a fallback strategy (§6.4) for completeness.

Static memory and register allocation is the closest problem to the memory allocation discussed in §6.2. A widely adopted solution for register allocation is graph coloring [5] over the interference graph. However, this assumes allocation sites (registers) to be of the same size. An alternative solution proposed to handle registers of different sizes is to model the register file as a puzzle board [59]. Static memory allocation for tensor buffers is often related to the rectangle packing problem, which can be modeled as a constraint satisfaction problem (CSP). Pruning techniques proposed in meta-CSP [49], TelaMalloc [47], and MiniMalloc [48] improve scalability. However, these solutions only model interference graphs and cannot trivially support addressing constraints of the tensor instructions in the accelerator ISA and the identity instructions (slice^H & concat^H). In ACT, we formulate a constraint satisfaction problem for the interference graph and ISA-specific addressing constraints generalized to multi-dimensional tensor buffers.

11 CONCLUSION

In this paper, we propose the first accelerator compiler backend generator, ACT - Accelerator Compiler Toolkit, that automatically generates a sound and complete compiler backend from a tensor accelerator ISA description. In doing so, we provide the first formalization of the tensor accelerator backend generation problem and propose a novel approach to solve it in a parameterized manner. We demonstrate the effectiveness of ACT by generating backends for three different tensor accelerators and show that these backends generate performant code with 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.

ACKNOWLEDGMENTS

We thank Stefanos Baziotis and Ahan Gupta for their constructive feedback. We would also like to thank Zhihao Wang and Saatvik Lochan for testing and debugging our implementation. 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] 2021. Google OR Tools: CP-SAT Solver. https://developers.google.com/optimization/cp/cp_solver.

[2] 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.

[3] 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. https://doi.org/10.1145/3503222.3507714

[4] Sagheer Ahmad, Sridhar Subramanian, Vamsi Boppana, Shankar Lakka, Fu-Hing Ho, Tomai Knopp, Juanjo Noguera, Gaurav Singh, and Ralph Wittig. 2019. Xilinx First 7nm Device: Versal AI Core (VC1902). In *2019 IEEE Hot Chips 31 Symposium (HCS)*. 1–28. https://doi.org/10.1109/HOTCHIPS.2019.8875639

[5] 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.

[6] 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. https://doi.org/10.1145/3620665.3640366

[7] 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. https://doi.org/10.1002/spe.4380170607

[8] 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. https://doi.org/10.1145/3704865

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

[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. https://doi.org/10.1145/3168821

[11] 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. https://doi.org/10.1016/S0304-3975(00)00109-2

[12] 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. https://doi.org/10.1145/3656401

[13] Stephen Chen. 1975. On the Sethi-Ullman algorithm. *International Journal of Computer Mathematics* 5 (1975), 37–55. https://api.semanticscholar.org/CorpusID:120540045

[14] 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. https://doi.org/10.1145/2541940.

2541967

[15] 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.

[16] 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. https://doi.org/10.1145/3445814.3446692

[17] 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. https://doi.org/10.1109/ISCA.2016.40

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

[19] Jacob Devlin. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

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

[21] 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. https://doi.org/10.1145/2749469.2750389

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

[23] 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.

[24] C. Daniel Freeman, Erik Frey, Anton Raichuk, Sertan Girgin, Igor Mordatch, and Olivier Bachem. 2021. Brax - A Differentiable Physics Engine for Large Scale Rigid Body Simulation. *CoRR* abs/2106.13281 (2021). arXiv:2106.13281 https://arxiv.org/abs/2106.13281

[25] Roy Frostig, Matthew James Johnson, and Chris Leary. 2018. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning* 4, 9 (2018).

[26] 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)*.

[27] 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.

[28] Google. 2020. Trax Tutorials. https://trax-ml.readthedocs.io/en/latest/.

[29] 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. https://doi.org/10.1145/3639051

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

[31] 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. https://doi.org/10.1145/3519939.3523446

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

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

[34] Intel. 2025. oneDNN assembly code for AMX GEMM (K1). https://github.com/uxlfoundation/oneDNN/blob/v3.8.1/src/cpu/x64/gemm/amx/jit_avx512_core_amx_gemm_kern.cpp#L336.

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

[36] 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. https://doi.org/10.1145/3341301.3359630

[37] Stephen Johnson. 2001. Yacc: Yet Another Compiler-Compiler. *Unix Programmer's Manual* 2 (11 2001).

[38] 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. https://doi.org/10.1145/3360307

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

[40] 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. https://doi.org/10.1145/3620665.3640385

[41] 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. https://doi.org/10.1145/3296957.3173176

[42] 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.

[43] 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. https://doi.org/10.1109/CGO51591.2021.9370308

[44] Amanda Liu, Gilbert Louis Bernstein, Adam Chlipala, and Jonathan Ragan-Kelley. 2022. Verified tensor-program optimization via high-level scheduling rewrites. *Proc. ACM Program. Lang.* 6, POPL, Article 55 (Jan. 2022), 28 pages. https://doi.org/10.1145/3498717

[45] 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. https://doi.org/10.1145/3575693.3575707

[46] LLVM Target [n. d.]. LLVM Targets - GitHub. https://github.com/llvm/llvm-project/tree/main/llvm/lib/Target.

[47] 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. https://doi.org/10.1145/3567955.3567961

[48] 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. https://doi.org/10.1145/3623278.3624752

[49] 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.

[50] Steven S. Muchnick and Phillip B. Gibbons. 2004. Efficient instruction scheduling for a pipelined architecture. *SIGPLAN Not.* 39, 4 (April 2004), 167–174. https://doi.org/10.1145/989393.989413
[51] nki [n. d.]. Neuron Kernel Interface (NKI) - Beta. https://awsdocs-neuron.readthedocs-hosted.com/en/latest/general/nki/index.html.
[52] 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. https://doi.org/10.1145/3729301
[53] Operation semantics | OpenXLA Project 2025. https://openxla.org/xla/operation_semantics.
[54] pallas [n. d.]. Pallas: a JAX kernel language. https://jax.readthedocs.io/en/latest/pallas/index.html.
[55] 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. https://proceedings.neurips.cc/paper_files/paper/2023/file/ded1a89e2b3b925444ada973af66336e-Paper-Datasets_and_Benchmarks.pdf
[56] 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. https://doi.org/10.1109/PACT52795.2021.00008
[57] 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. https://doi.org/10.1109/ISSCC42614.2022.9731612
[58] GNU Project. 2021. Bison 3.8.1. https://www.gnu.org/software/bison/manual/bison.html.
[59] Fernando Magno Quintão Pereira and Jens Palsberg. 2008. Register allocation by puzzle solving. *SIGPLAN Not.* 43, 6 (June 2008), 216–226. https://doi.org/10.1145/1379022.1375609
[60] Amazon Web Services. 2024. AWS Inferentia. https://aws.amazon.com/ai/machine-learning/inferentia/.
[61] Amazon Web Services. 2024. AWS Trainium. https://aws.amazon.com/ai/machine-learning/trainium/.
[62] Ravi Sethi and J. D. Ullman. 1970. The Generation of Optimal Code for Arithmetic Expressions. *J. ACM* 17, 4 (Oct. 1970), 715–728. https://doi.org/10.1145/321607.321620
[63] Wilson Snyder, Paul Wasson, and Duane et al Galbi. [n. d.]. *Verilator*. https://github.com/verilator/verilator
[64] StableHLO GitHub repository - StableHLO Specification 2025. https://github.com/openxla/stablehlo/blob/main/docs/spec.md/.
[65] 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.
[66] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality saturation: a new approach to optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915
[67] 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. https://doi.org/10.1145/3617232.3624873
[68] 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. https://doi.org/10.1145/3315508.3329973
[69] 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. https://doi.org/10.1109/ISCA59077.2024.00024
[70] triton [n. d.]. Towards Agile Development of Efficient Deep Learning Operators. https://www.jokeren.tech/slides/triton_next.pdf.
[71] 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. https://doi.org/10.1145/3372799.3394358
[72] 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

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

[74] 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

[75] 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. https://doi.org/10.14778/3407790.3407799

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

[77] 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

[78] Gaofeng Zhou, Jianyang Zhou, and Haijun Lin. 2018. Research on NVIDIA Deep Learning Accelerator. , 192-195 pages. https://doi.org/10.1109/ICASID.2018.8693202

A VISUALIZATION OF TENSOR OPERATORS

Tensor Operator	Concretizing 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 $dims$
$y = \text{dot}_{(c_1,c_2)}(x_1, x_2)$	$\text{type}(x_1), \text{type}(x_2), c_1, c_2$	Dot product of x_1, x_2 over contracting dimensions c_1, c_2
$y = \exp(x)$	$\text{type}(x)$	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 5. Brief descriptions of tensor operators discussed in the paper (detailed operation semantics in [53, 64]). $\text{type}(x)$ 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.

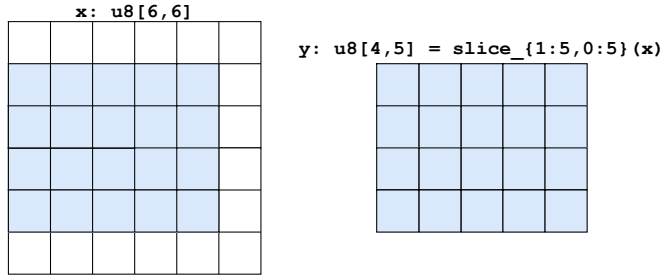


Fig. 23. Visualization of the operation $y = \text{slice}_{[1:5,0:5]}(x)$. slice extracts a 4×5 sub tensor from x.

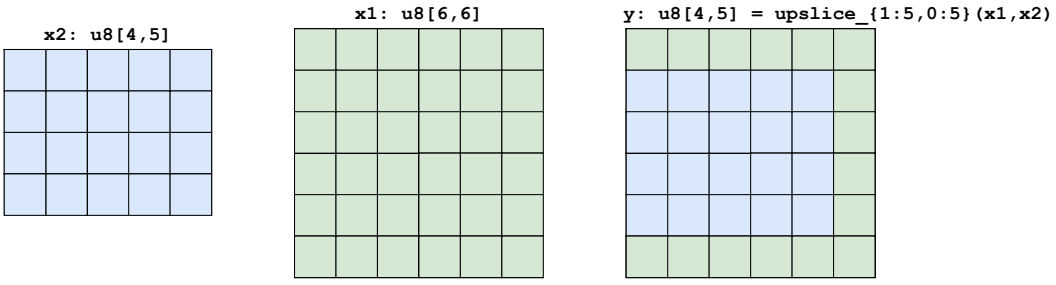


Fig. 24. Visualization of the operation $y = \text{upslice}_{[1:5,0:5]}(x_1, x_2)$. upslice inserts a 4×5 subtensor x_2 and puts it in x_1 from ranges $[1:5, 0:5]$.

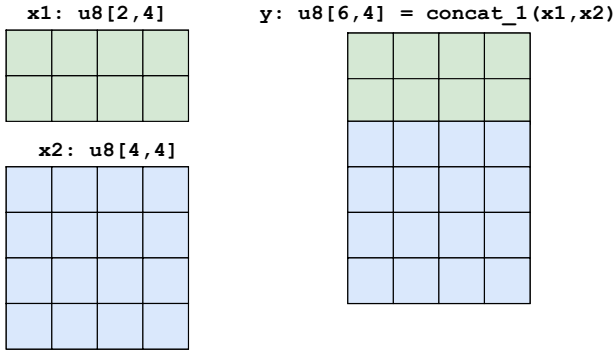


Fig. 25. Visualization of the operation $y = \text{concat}_1(x_1, x_2)$. Concatenate composes a new tensor of dimension $[6, 4]$ from two sub tensors x_1 and x_2 of dimensions $[2, 4]$ and $[4, 4]$, respectively, by stacking across `dim=1`.

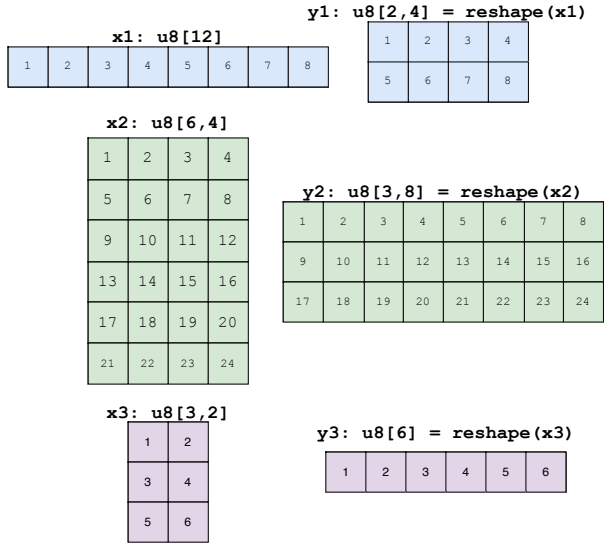


Fig. 26. Visualization of the operation $y = \text{reshape}(x)$. The blue tensor is reshaped from $[12, 1]$ to $[2, 4]$. The green tensor is reshaped from $[6, 4]$ to $[3, 8]$. The purple tensor is reshaped from $[3, 2]$ to $[6, 1]$.

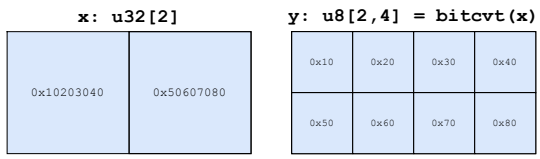


Fig. 27. Visualization of the operation $y = \text{bitcvt}(x)$. Element-wise bitcast conversion from u32 to u8.

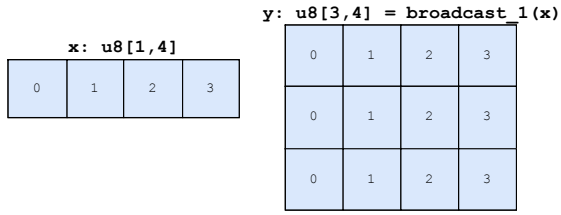


Fig. 28. Visualization of the operation $y = \text{broadcast}_1(x)$. Broadcasts x across dimension 1.

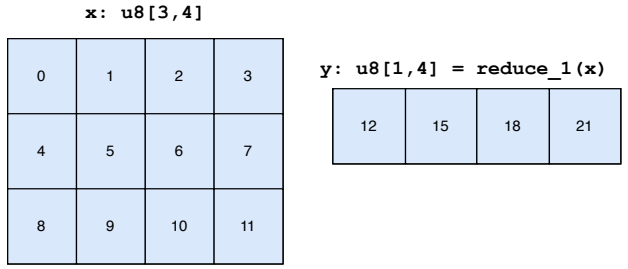


Fig. 29. Visualization of the operation $y = \text{reduce}_1(x)$. Sum reduces x across dimension 1.

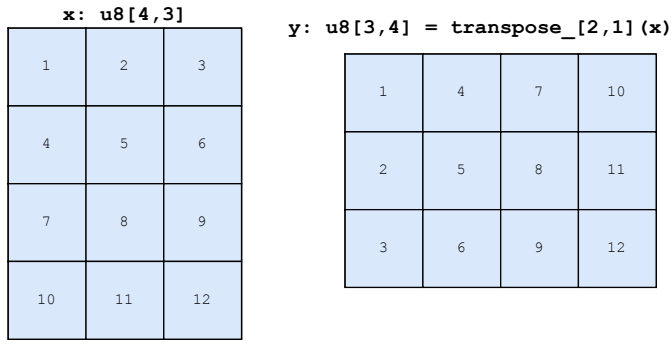


Fig. 30. Visualization of the operation $y = \text{transpose}_{[2,1]}(x)$. Transposes x from $[4, 3]$ to $[3, 4]$.

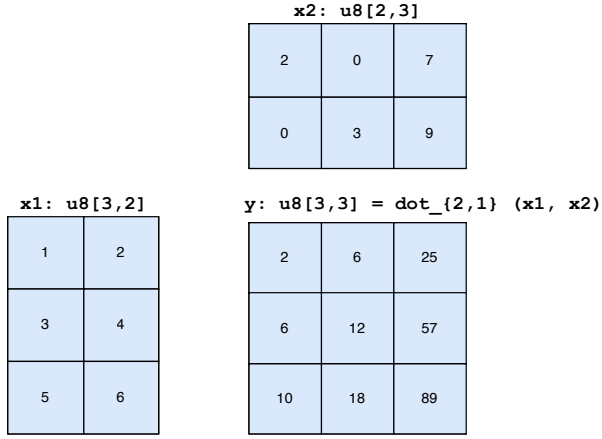


Fig. 31. Visualization of the operation $y = \text{dot}_{\{2,1\}}(x)$. Matrix multiplies x_1 of dimension $[3, 2]$ and x_2 of dimension $[2, 3]$ to get output tensor y of dimension $[3, 3]$.

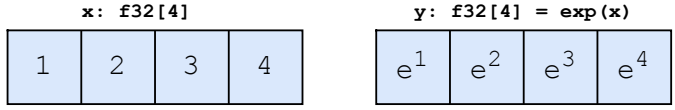


Fig. 32. Visualization of the operation $y = \exp(x)$. Element-wise exponential of the vector x.

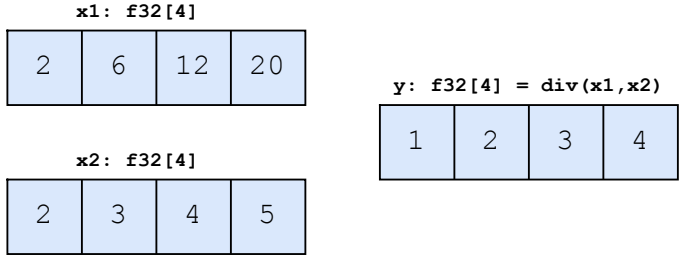


Fig. 33. Visualization of the operation $y = \text{div}(x1, x2)$. Element-wise division of x_1 by x_2.

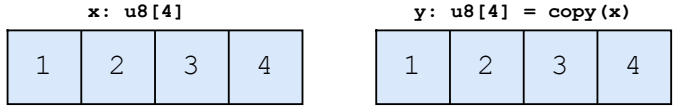


Fig. 34. Visualization of the operation $y = \text{copy}(x)$. Copies tensor x into y.

B GENERALIZED REPRESENTATION OF TENSOR ACCELERATOR ISA DESCRIPTION

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

Next, we formally define this generalized representation using notations defined previously.

DEF B.1. *An **abstract instruction** $\theta \in \Theta^H$ which reads n_θ tensors located at $d_\theta^{(i)}[I_s^{(i)}{:}I_e^{(i)}]|_{i=1}^{n_\theta}$ and modifies a tensor located at $d_\theta^{(0)}[I_s^{(0)}{:}I_e^{(0)}]$, is represented using an abstract tensor computation g_θ concretized by the set of computational attributes α, $(n_\theta+1)$ data slice addresses $h_\theta^{(i)}|_{i=0}^{n_\theta}$ over the set of addressing attributes β and a validity constraint e_θ over the set of attributes α and β.*

DEF B.2. *A **concrete instruction** $\theta_{\alpha,\beta}$ represents a valid modification if $e_\theta(\alpha,\beta)$ is true. It reads data slices $x_i = d_\theta^{(i)}[I_s^{(i)}{:}I_e^{(i)}]|_{i=1}^{n_\theta}$ and modifies data slice $y = d_\theta^{(0)}[I_s^{(0)}{:}I_e^{(0)}]$ with the corresponding concrete tensor computation $y = f(x_i|_{i=1}^{n_\theta})$, where $f = g_\theta(\alpha)$ and $\forall i \in [0, n_\theta]$ $(I_s^{(i)}, I_e^{(i)}) = h_\theta^{(i)}(\beta)$.*

DEF B.3. ***Execution** $\mathcal{E}(\theta_{\alpha,\beta})$ of a concrete instruction $\theta_{\alpha,\beta}$ is a concrete tensor computation (over a tuple of tensors) that updates the memory state $M = (d_0, d_1, \dots)$ to $M' = (d_0', d_1', \dots)$ where*

$$d_i' = \begin{cases} \mathsf{upslice}_{[h_\theta^{(0)}(\beta)]}\left(d_\theta^{(0)}, g_\theta(\alpha)\left(\mathsf{slice}_{[h_\theta^{(i)}(\beta)]}(d_\theta^{(i)})\Big|_{i=1}^{n_\theta}\right)\right) & \text{for } d_i = d_\theta^{(0)} \text{ (d_i is the output buffer)} \\ \mathsf{copy}(d_i) & \text{for } d_i \neq d_\theta^{(0)} \text{ (rest are not modified)} \end{cases}$$

Here, upslice, slice, & copy are tensor operators described in Table 5 and visualized in Appendix A.

Kernel programming languages like Exo [31] model custom hardware instructions under similar assumptions, albeit defining semantics using scalar operators and Python-like syntax.
Intuitively, we can represent scalar-based pseudocode descriptions with FOR and IF statements (like Intel Intrinsics Guide [33]) using element-wise tensor operators (since scalars are 0-D tensors) with while and select operators, respectively.

We can prove the generality of the above representation in the following steps:

- All *relevant* concrete instructions can be represented as a concrete tensor computation graph with the input and output being read and written to data slices. (Hint: it is a deterministic modification)
- There are finite concrete instructions for a given accelerator.
- There are finite abstract instructions for a given accelerator.
- There are finite concrete instructions for a given abstract instruction.
- An abstract instruction θ can be represented by a finite set of abstract instructions θ's such that for every abstract instruction θ', the concrete tensor computations are isomorphic, i.e., can be represented by an abstract tensor computation.
- Since there are finite abstract instructions for a given accelerator, every ISA can be represented using finite abstract instructions of form θ'.
- Worst case, every concrete instruction is an abstract instruction in itself.
- Such a θ can always be represented as the generalized definition.
- A simple solution is that since we have finite concrete instructions, α can enumerate the unique concrete tensor computations present, which makes g_θ the enumeration function and β can be the set of addresses (size $2 \cdot (n_\theta + 1)$, which makes $h_\theta^{(i)}$ as identity mappings. We define $e_\theta(\alpha,\beta)$ as true iff there is a concrete instruction with concrete tensor computation $g_\theta(\alpha)$ and data slices β. This strategy shows that every instruction can always be split into integer attributes.

C MORE DETAILS ON CORE ACT ALGORITHM (SECTION 5 & 6)

C.1 Module 1: E-Graph Initializer

In the paper, we have skipped the details of XLA-HLO IR and how tiled memory formats are represented. We have a simple example in §9.2 (Case Study 3) that discusses this.

An important feature of bflat is that it is invertible since reshape and bitcvt are both invertible operators. The input and output nodes in the e-graph that represent byte-flattened tensors are connected to the input graph G with bflat^{-1} and bflat respectively, as shown in Fig. 9.

C.2 Module 2: Rewrite Applier

In §5.2.1, we mentioned the filtering from the set of IR-to-IR rewrite rules (foundational axioms $\mathcal{R}$). Rewrite rules are selected based on whether they are going to lead to any IR-to-ISA rewrite rule or not. We do this iteratively to handle transitive rewrite rules. For example, if op C appears in any g_θ, and then any rewrite rule generating C is added to the set. If there is a rewrite rule B -> C, then op B is also added to the set, and this process continues iteratively. This helps to prune the operator set potentially supported by an accelerator during compiler backend generation time itself.

In §5.2.2, we mentioned that rewrite rule $\text{bitcvt}(\text{reshape}(x)) \rightarrow \text{load\_rm}(x)$ is generated from lines 11-17 of Fig. 7. Snippet of its precondition is present in Fig. 35.

```
pub fn precond_load_rm(...) -> bool {
    ...
    // y: bf16[`@alpha.n`, 64]
    if y.shape.len() != 2 || y.shape[1] != 64 || y.dtype != Dtype::BF16 {
        return false;
    }
    let alpha_n = y.shape[0];
    if alpha_n <= 128 { // check based on instr.set_constraints(...)
        return false;
    }
    ... // Repeat for a, x1
}
```

Fig. 35. Snippet of rewrite precondition (Rust) generated from semantics of load_rm in Fig.7 of the paper

In §5.2.3, we discuss identity instructions that are useful for tiling the inner loop and padding tensor variables. Fig. 36 shows a simple pii graph generated by ACT-Gemmini. These identity instructions play a crucial role in guaranteeing completeness.

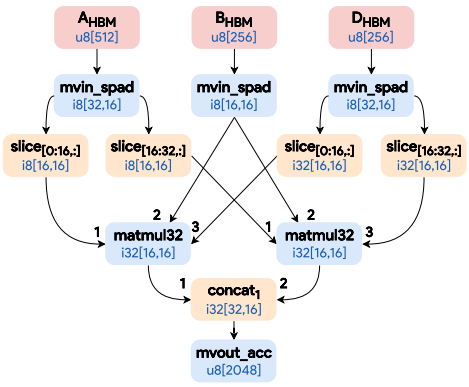


Fig. 36. pii graph using identity instruction slice & concat

C.3 Module 3: Graph Extractor

A high-level Python-like pseudocode for the extraction algorithm is in Fig. 37.

```
selected = set()
def extract(eclass, output_buffer):
    for enode in eclass:
        if enode.type != "pii":
            continue
        if enode.output != output_buffer:
            continue
        # Iterate through all the operand e-classes of the e-node
        for idx, operand in enumerate(enode.operands):
            if not extract(operand, enode.input_buffer[idx]):
                return False
        # If we reach here, we have successfully selected this enode
        selected.add(enode)
        return True
    return False

extract(ROOT_eclass, "d0")
```

Fig. 37. A simplified Python-like pseudocode for the extraction algorithm discussed in §5.3

To expand on the termination condition, if you reach an e-class expecting a tensor in d_0 (HBM) and the e-class is marked as constant or contains an e-node representing a flattened input tensor variable, we terminate. If it were a flattened input tensor, we add that to the pii graph. If the e-class was marked as constant, then the tensor expression that computes this value is added to the pii graph (Fig.17 of the paper). These two cases are exclusive and can never happen in the same e-class.

C.4 Module 4: Topological Ordering Generator

We extend the Sethi-Ullman numbering to multiple tensor buffers storing varying sizes of tensor variables. For a tensor buffer d, we define $\text{esun}_d(y) = \max(\text{mem}_d(y), \min_i \tau_d(x_i))$ where $x_i's$ are the operands of node y, $\tau_d(x_i) = \text{esun}_d(x_i) + \sum_{i \neq j} \text{mem}_d(x_j)$ and $\text{mem}_d(x_j)$ corresponds to the memory size (in bytes) of the tensor variable represented by the node x_j. We compute esun_d for all tensor buffers in D^H except d_0 (HBM) and select the operands x_i in decreasing order of $\max_d(\tau_d(x_i)/\text{mem}(d))$, which represents the tensor buffer requirement.

A key reason for choosing Sethi-Ullman numbering as the base algorithm is the matching objective to our work. To improve the likelihood of CSP being SAT, we want the interference graph to be as small as possible. This is analogous to minimizing register pressure, which is the objective of Sethi-Ullman numbering. While Sethi-Ullman numbering was provably optimal for register files, we have a simple counterexample (Fig. 38) to show it is not always optimal for more than one register file. In fact, any depth-first topological ordering will not minimize the number of registers required for this counterexample. While this makes it difficult for us to prove an optimal schedule but the algorithm guides towards a schedule with low register pressure. In practice, we observe that this heuristic yields good results as well.

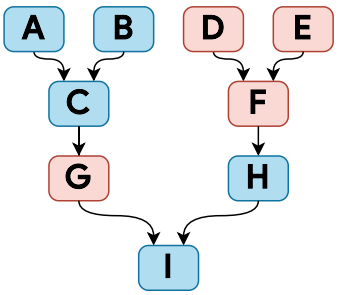


Fig. 38. A simple counter-example with two register files (blue, red) where all depth-first topological ordering will use (3, 2) or (2, 3) registers. Topological ordering A-I only uses (2, 2) registers.

C.5 Module 5: Constraint Satisfaction Problem Generator

Google OR-Tools CP-SAT solver is a multi-threaded program, and thus, the solution is non-deterministic. However, a compiler engineer might expect determinism from the ACT-generated compiler backend. So, we allow the compiler engineers to force determinism using a simple boolean flag. If the flag is set, determinism is maintained by invoking the CP-SAT solver with `NumSearchWorkers = 1` as discussed at https://groups.google.com/g/or-tools-discuss/c/ZGj5vhhZX48

C.6 Module 6: Code Emitter

A short discussion on pre-allocated constant tensors. As defined in CSP_4, all constant tensors have an overlapping live range. Therefore, all pre-allocated constant tensors are assigned non-overlapping address ranges. Therefore, the final constant tensor is just the concatenation of these constant tensors.

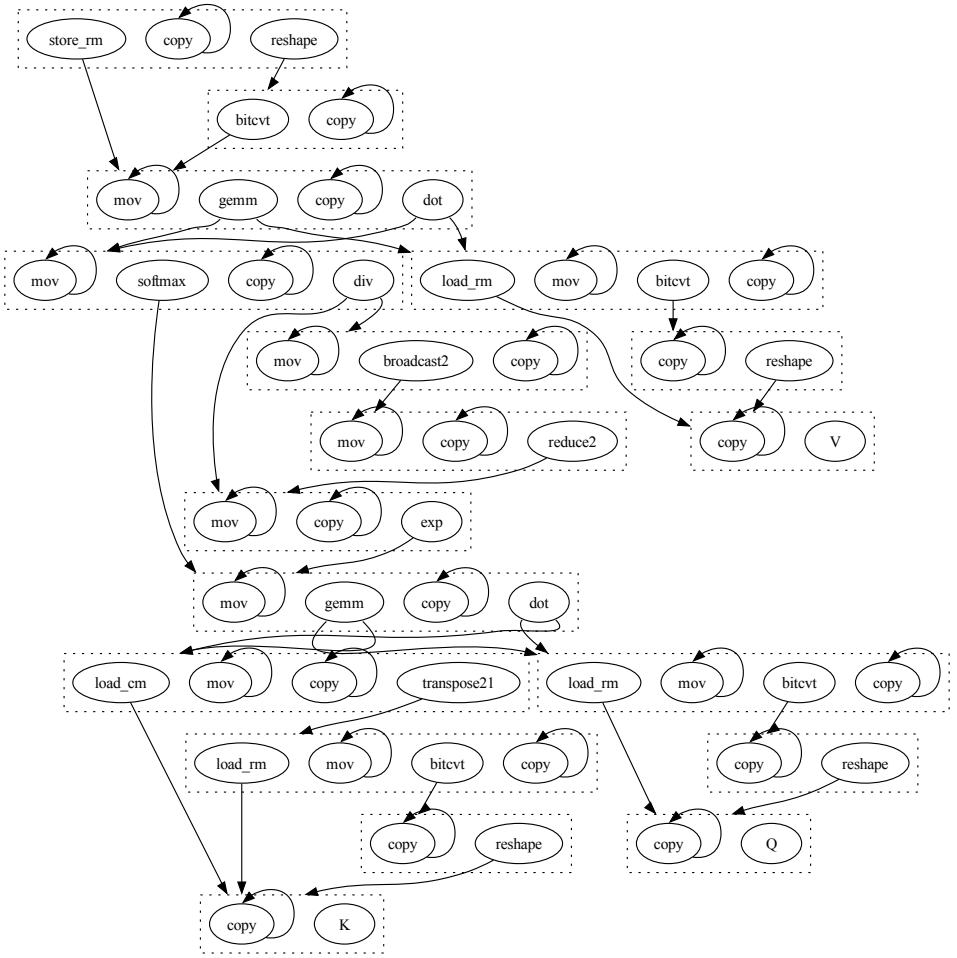


Fig. 39. Saturated e-graph for running example G_{QKV} (snippet in Fig. 9).

D FUZZ TESTING SETUP

We use NNSmith [45] to generate computation graphs for ACT-Gemmini fuzz testing. Our setup supports 10 XLA-HLO operators listed in Table 6. To generate the graphs, we specify the seed, max_nodes, include, type, backend, and dtype_choices. nnsmith generates graphs with max_nodes amount of operators. The computation graphs have multiple outputs, and in order to have a singular output, we combine all the outputs with the add operator. Therefore, the total nodes in the final XLA-HLO graph is (#operators) + (#input parameters) + (#output nodes - 1). Even with a max_nodes of 50, it eventually leads to a kernel size of 89.

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

Table 6. 10 XLA-HLO operators present in the tensor kernels generated for fuzz testing using NNSmith [45].

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

E STATISTICS OF ONEDNN KERNELS

The oneDNN library consists of a set of common DNN kernels and is hand-written and heavily optimized for Intel ISA, including AMX and AVX512. XLA compiler generates fused subgraphs for these kernels, which are then compiled to assembly code using the oneDNN library (xla_cpu_use_mkl_dnn flag). We selected five such kernels commonly observed in oneDNN examples [35]. These kernels are based on simple matrix computations with complex memory layouts of the input matrices. Table 40 summarizes the statistics for the selected kernels labeled as K1-K5.

	#Nodes	#AMX	#AVX512
K1	6	16	0
K2	6	9	0
K3	7	0	100
K4	7	0	107
K5	23	16	37

Fig. 40. Statistics of selected oneDNN kernels and their optimized assembly code in oneDNN library

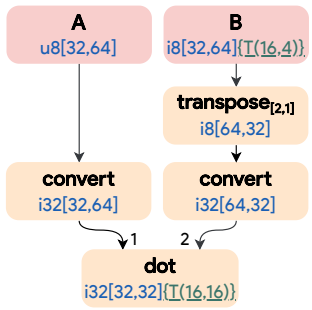


Fig. 41. Tensor computation graph for oneDNN kernel K1

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

Fig. 42. XLA-HLO code for kernel K1.

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

Fig. 43. Compiled assembly code for kernel K1.

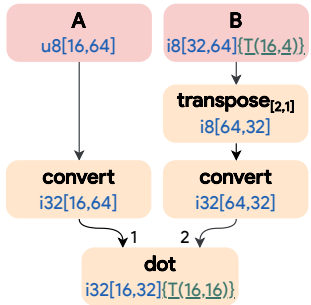


Fig. 44. Tensor computation graph for oneDNN kernel K2

```
1 ENTRY k2 {
2   A = u8[16,64] parameter(0)
3   B = i8[32,64]{T(16,64)} parameter(1)
4   convert.0 = i32[16,64] convert(A)
5   transpose.1 = i8[64,32] transpose(B), dimensions=[2,1]
6   convert.2 = i32[64,32] convert(transpose.1)
7   ROOT dot.3 = i32[16,32]{T(16,16)} dot(convert.0, convert.2)
8 }
```

Fig. 45. XLA-HLO code for kernel K2.

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

Fig. 46. Compiled assembly code for kernel K2.

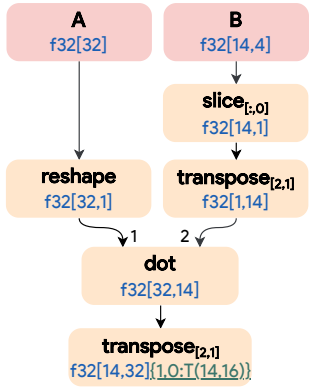


Fig. 47. Tensor computation graph for oneDNN kernel K3

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

Fig. 48. XLA-HLO code for kernel K3.

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

Fig. 49. Compiled assembly code for kernel K3.

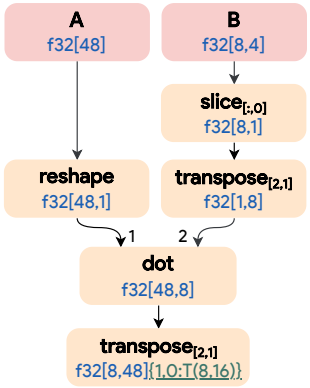


Fig. 50. Tensor computation graph for oneDNN kernel K4

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

Fig. 51. XLA-HLO code for kernel K4.

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

Fig. 52. Compiled assembly code for kernel K4.

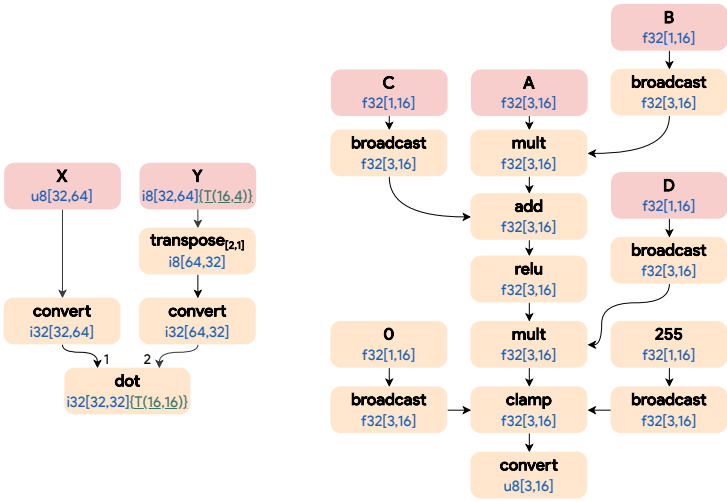


Fig. 53. Tensor computation graph for oneDNN kernel K5

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

Fig. 54. XLA-HLO code for kernel K5.

```
tilezero(dst = tmm0)
tilezero(dst = tmm1)
tilezero(dst = tmm2)
tilezero(dst = tmm3)
tileloadd(dst = tmm4, mem_addr = 0x0000)
tileloadd(dst = tmm5, mem_addr = 0x0400)
tileloadd(dst = tmm6, mem_addr = 0x0800)
tileloadd(dst = tmm7, mem_addr = 0x0c00)
tdpbusd(dst = tmm0, src0 = tmm4, src1 = tmm6)
tdpbusd(dst = tmm1, src0 = tmm4, src1 = tmm7)
tdpbusd(dst = tmm2, src0 = tmm5, src1 = tmm6)
tdpbusd(dst = tmm3, src0 = tmm5, src1 = tmm7)
tilestored(src = tmm0, mem_addr = 0x1000)
tilestored(src = tmm1, mem_addr = 0x1400)
tilestored(src = tmm2, mem_addr = 0x1800)
tilestored(src = tmm3, mem_addr = 0x1c00)
vmovupsz(dst = zmm10, mem_addr = 0x000)
vmovupsz(dst = zmm15, mem_addr = 0x040)
mov(dst = ebx, imm = 0x437f000)
vmovq(dst = xmm9, src = rbx)
vbroadcastss(dst = zmm9, src = xmm9)
vmovupsz(dst = zmm31, mem_addr = 0x080)
vcvtdq2ps(dst = zmm31, src = zmm31)
vmovupsz(dst = zmm31, mem_addr = 0x0c0)
vcvtdq2ps(dst = zmm30, src = zmm30)
vmovupsz(dst = zmm31, mem_addr = 0x100)
vcvtdq2ps(dst = zmm29, src = zmm29)
vmulps(dst = zmm31, src0 = zmm31, src1 = zmm15)
...
vaddps(dst = zmm31, src0 = zmm31, src1 = zmm10)
...
vmaxps(dst = zmm31, src0 = zmm31, src1 = zmm8)
...
vbroadcastssl(dst = zmm0, mem_addr = 0x144)
vmulps(dst = zmm31, src0 = zmm31, src1 = zmm0)
...
vmaxps(dst = zmm31, src0 = zmm31, src1 = zmm8)
vminps(dst = zmm31, src0 = zmm31, src1 = zmm9)
vcvtps2dq(dst = zmm31, src = zmm31)
vpmovusdbx(src = zmm31, mem_addr = 0x144)
...
```

Fig. 55. Compiled assembly code for kernel K5.