TensorIR: An Abstraction for Automatic Tensorized Program Optimization

Siyuan Feng*
Shanghai Jiao Tong University
Shanghai, China
hzfengsy@sjtu.edu.cn

Bohan Hou*
Carnegie Mellon University
Pittsburgh, USA
bohanhou@cs.cmu.edu

Hongyi Jin†
Carnegie Mellon University
Pittsburgh, USA
hongyij@cs.cmu.edu

Wuwei Lin
OctoML
Seattle, USA
wlin@octoml.ai

Junru Shao
OctoML
Seattle, USA
jshao@octoml.ai

Ruihang Lai†
Carnegie Mellon University
Pittsburgh, USA
ruihangl@cs.cmu.edu

Zihao Ye
University of Washington
Seattle, USA
zhye@cs.washington.edu

Lianmin Zheng
UC Berkeley
Berkeley, USA
lmzheng@berkeley.edu

Cody Hao Yu
Amazon Web Services
Seattle, USA
hyuz@amazon.com

Yong Yu
Shanghai Jiao Tong University
Shanghai, China
yyu@apex.sjtu.edu.cn

Tianqi Chen
Carnegie Mellon University, OctoML
Pittsburgh, USA
tqchen@cmu.edu
tqchen@octoml.ai

ABSTRACT

Deploying deep learning models on various devices has become an important topic. The wave of hardware specialization brings a diverse set of acceleration primitives for multi-dimensional tensor computations. These new acceleration primitives, along with the emerging machine learning models, bring tremendous engineering challenges. In this paper, we present TensorIR, a compiler abstraction for optimizing programs with these tensor computation primitives. TensorIR generalizes the loop nest representation used in existing machine learning compilers to bring tensor computation as the first-class citizen. Finally, we build an end-to-end framework on top of our abstraction to automatically optimize deep learning models for given tensor computation primitives. Experimental results show that TensorIR compilation automatically uses the tensor computation primitives for given hardware backends and delivers performance that is competitive to state-of-art hand-optimized systems across platforms.

1 INTRODUCTION

Deploying high-performance machine learning models has become an emerging challenge in various areas, including image recognition [18, 19, 39], natural language processing [13, 34, 44], and games [27, 29, 37]. The advances in machine learning bring demands to support a broad range of models. In the meantime, there are increasing demands to deploy smart applications to a broad spectrum of devices ranging from servers to embedded environments.

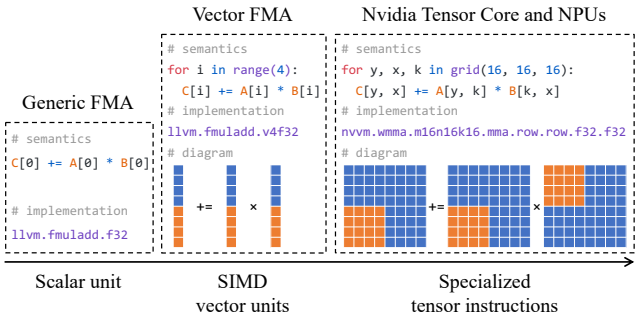

Figure 1: Trends of hardware specialization. The classical acceleration technique uses vector units to process multiple scalar computations simultaneously, which is still widely used on CPU platforms. However, to cater to increasingly heavier computation throughput requirements, modern accelerators usually contain specialized high-dimensional tensor computation instructions, creating the need for tensorized program optimization.

The wave of hardware specialization further complicates the problem (Figure 1). Driven by the goal of machine learning acceleration, modern hardware backends introduce specialized primitives to speed up tensor computations (e.g., Nvidia Tensor Core [31], Google TPU [22]). Domain experts also start to develop micro-kernel primitives, which carefully organize a series of highly optimized instructions to perform a sub-computation to speed up domain-specific

*Both authors contributed equally to the paper
†Part of this work was done at Shanghai Jiao Tong University

tensor operator libraries. These hardware instructions and micro-kernel primitives typically operate on multi-dimensional tensor regions and effectively perform tensor operations such as multi-dimensional loads, dot product, and matrix multiplication (Figure 1). We call these opaque tensor computation acceleration constructs *tensorized intrinsics* and transformation procedure to use these intrinsic *tensorization*. In order to get the best out of these hardware backends, modern machine learning systems need to optimize programs that contain hierarchical loop nests, multi-dimensional loads, and tensor intrinsics – we call this problem *tensorized program optimization*.

Most of the current tensorized programs are optimized by domain experts, who compose tensorized primitives together with multi-dimensional loops, threading patterns, and data caching to craft specialized kernel libraries such as Intel MKL-DNN [20], ARM Compute Library [3] and NVIDIA cuDNN [11]. These libraries are then used by machine learning frameworks such as TensorFlow [1], PyTorch [33] and MXNet [8]. However, huge engineering efforts are required to support the growing sets of models and backends, and it takes iteration cycles for these libraries to adapt to the rapidly changing and growing machine learning applications, which hinders the evolution of new machine learning models.

In this paper, we propose to address the tensorized program optimization problem using a automatic compilation approach. Most past works in machine learning compilation [9, 43] search over a program space of loop nest transformations and do not handle tensorized programs automatically. Bringing automatic program optimization to tensorized programs would unlock the benefits from domain-specific accelerations in modern hardware backends. We identify the following key challenges to achieving this goal:

Abstraction for Tensorized Programs. To build an automated compiler for tensorized programs, we need an abstraction that can pragmatically capture possible equivalent tensorized computations for a given machine learning operator. Notably, the abstraction needs to represent multi-dimensional memory accesses, threading hierarchies, and tensorized computation primitives from different hardware backends. The abstraction also needs to be expressive enough to represent most of the operators of interest in machine learning.

Large Design Space of Possible Tensorized Program Optimizations. Another challenge is to produce an optimized tensorized program for a given operator automatically. A compiler needs to make use of a rich set of techniques that domain experts might use, including making effective use of loop tiling, threading, and data layout transformations. Importantly, these transformations now need to be made in conjunction with tensorized computations, bringing additional complexities to analysis and automation. The combinations of these transformations form a large search space. We need an effective way to find an optimized tensorized program for a given search space.

To address these challenges, we introduce *TensorIR*, an abstraction for automatic tensor program optimization. To begin with, we introduce a new construct called *block* that allows us to divide and isolate tensorized computation region from the outer loop nests. The new abstraction allows us to effectively represent tensorized computations and combine them with loop nests, threading, and memory hierarchy. We also introduce program transformation primitives to express a rich space of potential optimizations. We build a novel automatic scheduling algorithm on top of the abstraction and transformation primitives. Additionally, TensorIR abstraction also allows us to represent and optimize programs that contain a mixture of irregular computations and tensor computations, expanding the possible support beyond a normal tensor expression [9]. This paper makes the following contributions:

- We propose a novel abstraction for tensorized programs that separates tensorized computation from the loop transformations. Meanwhile, the same abstraction allows us to uniformly represent tensor intrinsics and hardware constraints.
- We build transformation primitives to generate a rich search space of tensorized program optimization with correctness validation.
- We design and implement a new tensorization-aware automatic scheduler.

We integrate TensorIR with an end-to-end compilation framework and show that it outperforms existing machine learning compilation solutions by up to 7x and automatically brings competitive performance to heavily optimized platform-specific solutions.

2 OVERVIEW

This section describes the key insights of our approach and gives an overview of the paper. To motivate our approach, we start with an example flow of how a domain expert optimizes a tensorized program in Figure 2. Tensorized computation primitives usually correspond to a sub-problem of the original tensor operator computation. As a result, it is natural for domain experts to choose a *divide and conquer* approach – divide the original program into sub-problems of tensorized computation and loop nests that use the tensorized computation, then optimize them separately. The divide and conquer approach allows developers to focus on a sub-problem without worrying about the others. Additionally, it also enables us to target multiple tensorized computation implementations.

Most existing machine learning compilers take two kinds of approaches (Figure 3). Halide [35], TVM [9], Tiramisu [5], AKG [47], MLIR/Affine [26] and AMOS [49] take a bottom-up approach that models the search space using loop nests iterators around scalar operation bodies, and then optimizes the program by finding the best loop nest transformation (through search or polyhedral optimization). HTA [16], Fireiron [17] and Stripe [46] use a top-down approach that gradually decomposes the problem into sub-problems through nested polyhedral structures. Given the significance of the divide and conquer approach in manual tensorized program optimizations, it is natural to ask whether it is possible to bring the same insight to machine learning compiler design.

We give a positive answer in this paper. Specifically, we introduce a new abstraction called *block* into the loop nests. A block contains the right amount of signature information to *isolate* the inner problem space and outer problem. With block, we can continue to perform transformations on both outer and inner problem independently, using the block signature as the interface. Similar to the manual divide and conquer approach, a common use case of a block is to represent a tensorized computation primitive in a hardware backend, but we can also use the block to isolate bigger

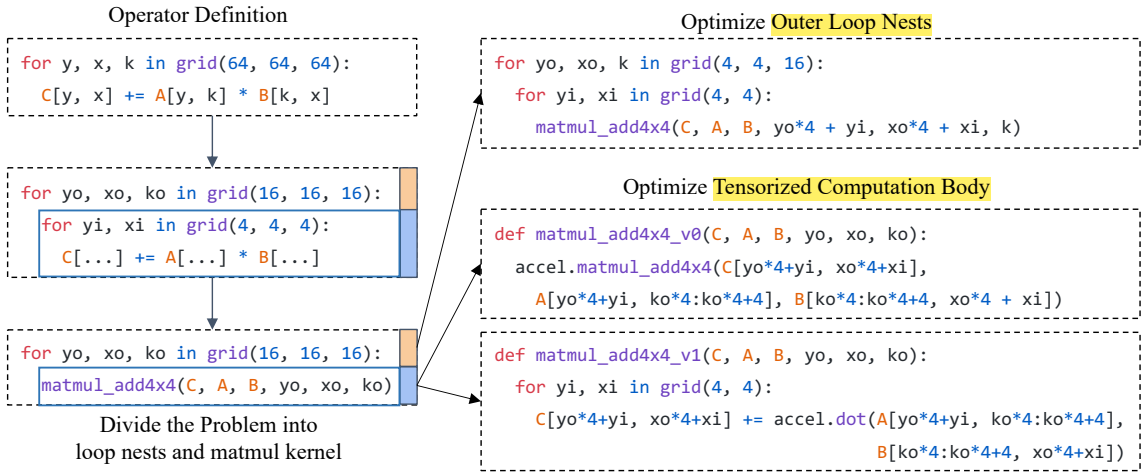

Figure 2: An expert developer can choose to divide the problem into 4x4 matmul and loops that uses the 4x4 matmul, then optimize them separately. This way we can effectively make use of specialized tensor instructions in the target hardware.

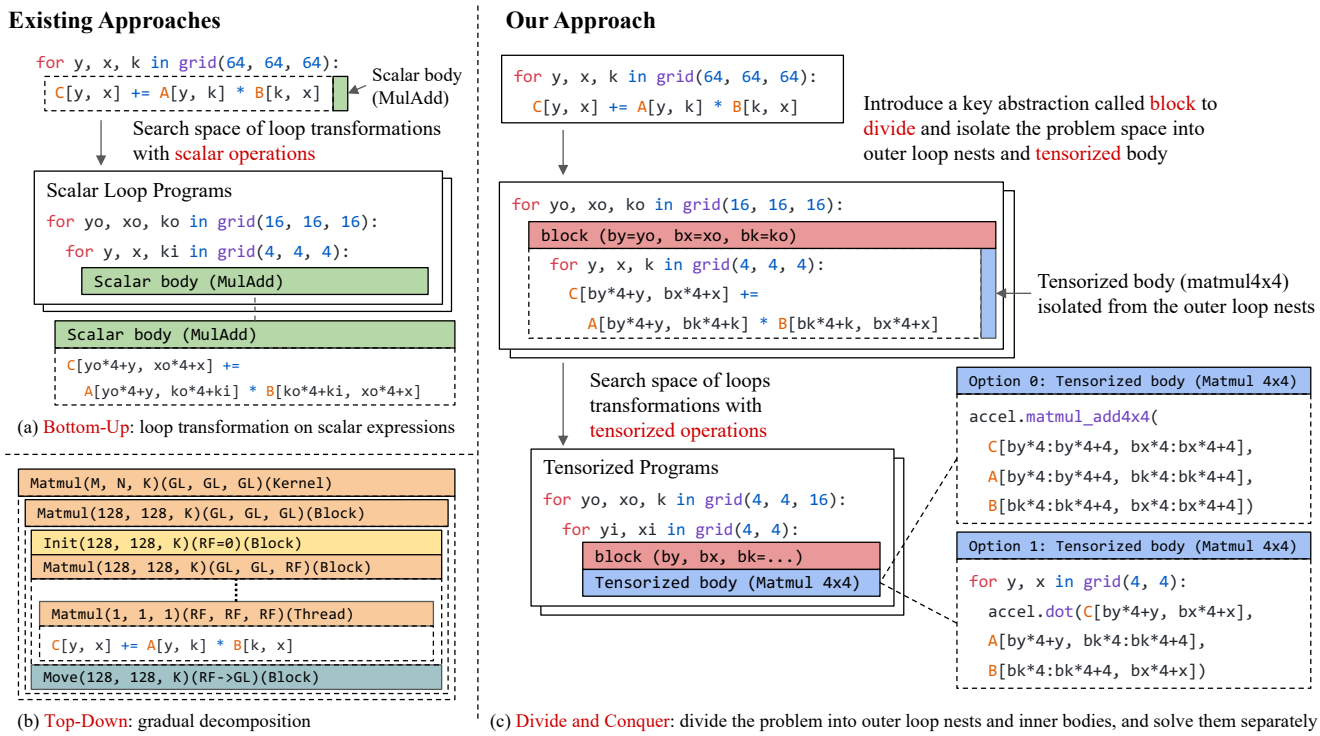

Figure 3: Overview of our approach. We use a key abstraction named block to divide and isolate the tensorized computations, and enables further loop transformations with tensorized operations.

sub-problems of interest when divide and conquer makes sense. Importantly, tensor computation is the first-class citizen in TensorIR. Loop nests with blocks can be viewed as a generalized abstraction of iteration space. We present the detailed design of the TensorIR abstraction in section 3.

To automate the tensorized program optimization, we construct a search space of possible ways to divide the problem guided by the hardware tensor computation primitives, then further search over

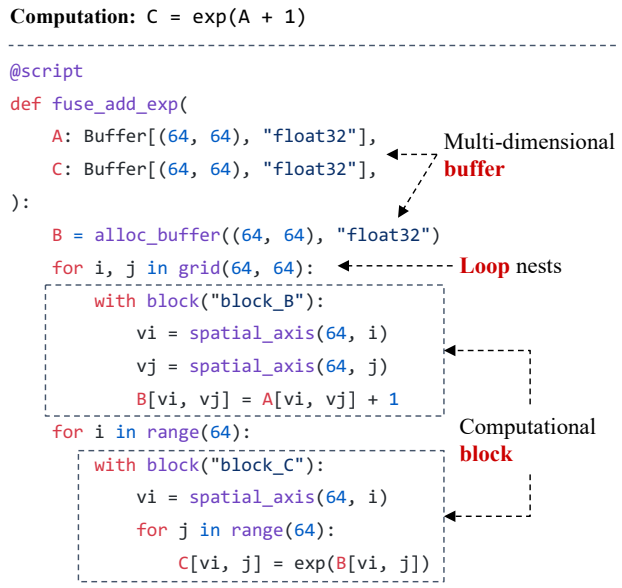

Figure 4: An example TensorIR program with three major elements - multi-dimensional buffers, loop nests and computational block. Details of block is omitted for simplification.

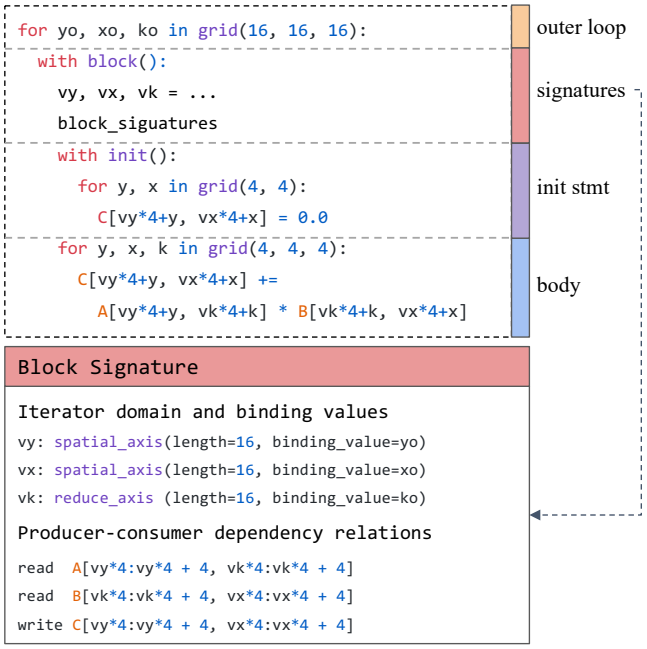

Figure 5: Blocks contain complete signature for dependency analysis and we make it an isolation level between body computation and outer loop nesting.

possible ways to solve sub-problems using program transformations. We present the automatic scheduling algorithm for tensorized programs in section 4.

Our divide and conquer covers the search space of previous compiler approaches (bottom-up, top-down) and generalizes the typical optimization techniques in HPC and ML engineering to an abstraction that allows automatic tensorization. We automate decisions common in library and compilation pipeline, enabling us to automatically generate competitive solutions with vendor-specific libraries. We present the experiment results in section 5.

3 TENSORIR ABSTRACTION

This section introduces the TensorIR abstraction. Figure 4 gives an example of TensorIR program. We introduce a Python-AST (abstract syntax tree) dialect of TensorIR to let developers directly construct and transform programs in Python. A TensorIR program contains three main elements: multi-dimensional buffers, loop nests (with possible thread bindings in GPU settings), and blocks. A block can contain one or more nested loop nests with sub-blocks or sequence of imperative statements that correspond to the content of the computation. This representation allows us to divide computations into the corresponding sub(block)-regions and do effective program transformations using dependency information stored in the block signature. We discuss the design details in §3.1.

3.1 Block

A block in TensorIR represents a tensorized computation on a sub-region of the multi-dimensional buffers. Figure 5 shows an example block for the matrix multiplication (matmul) computation. The body of a block is parameterized by a set of block iterator variables v_y, v_x, v_k which represents an abstract tensorized computation. Instantiated with different value combinations of these block iterator variables, the block maps to different concrete running block instances. These iterator variables can be bound to expressions that contain the outer loop iterators, which implies the execution order of block instances.

Rationale. The main design rationale of a block is to isolate tensorized computation – we want to be able to transform loop nests outside the block without looking into its body. However, unlike scalar computation, we may not be able to extract the dependency information needed for transformation from an opaque tensor computation body. As a result, we introduce a block signature that contains sufficient dependency information for transformations. We discuss these transformations in §3.2. Additionally, the signature can be used to independently verify the correctness of the iterator bindings during transformations (more details in §3.3).

Block Iterator Domain. While it is possible to instantiate a block's body computation by binding the block iterators to any loop nests, most instantiations do not correspond to the same computation. To ensure the consistency of computation among transformations, we store the iterator domain information and the constraints of iterators in the block signature. For the particular example in Figure 5, we know that vx, vy and vk must bind to iterators in domain $[0, 16)$. Additionally, because vk is a reduction axis, we know that we cannot bind it to a parallel loop unless the reduction is atomic. The domain constraints still leave massive room for outer loop transformations, as there are multiple ways to construct loops that

satisfy the constraint. Our domain signature can be viewed as a specific way to represent the integer domain sets and relations of the iterators. We choose the particular representation due to its implementation efficiency and simplicity in reasoning, but would also point out that the same design philosophy applies to other formal domain representations of integer sets and relations [42].

Access Region and Dependency. To provide sufficient dependency information, a block signature contains the access regions and read/write dependencies that a block has with respect to the multiple dimensional buffers. In Figure 5, the block writes the region $C[vy*4 : vy*4+4, vx*4 : vx*4+4]$ by reading $A[vy*4 : vy*4+4, vk*4 : vk*4+4], B[vk*4 : vk*4+4, vx*4 : vx*4+4]$. The dependency information is used during transformations. We only mark each block's dependency with respect to the multi-dimensional buffers instead of other statements (blocks). This indirection enables a broader range of transformations, such as data layout transformation and re-computation which are essential in tensorized program optimization.

Reduction Block and Initialization. A reduction computation usually contains an initialization step and an update step. We can naturally map the reduction computation into two blocks. But, on the other hand, it is usually helpful to jointly make the scheduling decisions (such as tiling and computation location) of the two steps. We introduce an optional initialization statement for blocks that perform reduction. An initialization statement is executed during the first iteration of a reduction. This reduction block representation is mainly useful during transformations. We provide transformation primitives to transform between the two-block-based representation and the init-block-based representation so we can pick the best representation for low-level code generation.

3.2 Scheduling Transformations

For a given input program, we need to generate a rich search space of programs with equivalent semantics. We introduce primitives to transform a TensorIR program to equivalent optimized programs. Following the existing convention of tensor program optimizations [5, 9, 35], we call this procedure scheduling.

A block is schedulable if it only contains loop nests with sub-blocks as its leaves. We can transform the loop nests and sub-block computation locations within a schedulable block by analyzing the sub-block signatures and their dependency information. Notably, a schedulable block can contain non-schedulable sub-blocks (e.g., opaque Tensor Core computation). An opaque block can also contain a schedulable sub-block. Based on the block isolation, we can still effectively explore the search space of the schedulable part independently while keeping the same opaque block. We describe the schedule primitives in the rest part of this subsection.

Loop Transformations. Loop transformations such as loop tiling (split, reorder) and compute location mutation are important ways to optimize programs for better memory locality. We also provide these loop transformation primitives (see examples in Figure 6). Unlike existing tensor compilers that directly extract the dependency of each leaf scalar computation statement, we calculate the dependencies by only inspecting the block signature. Besides loop transformations, we also support primitives that bind loops to GPU threads

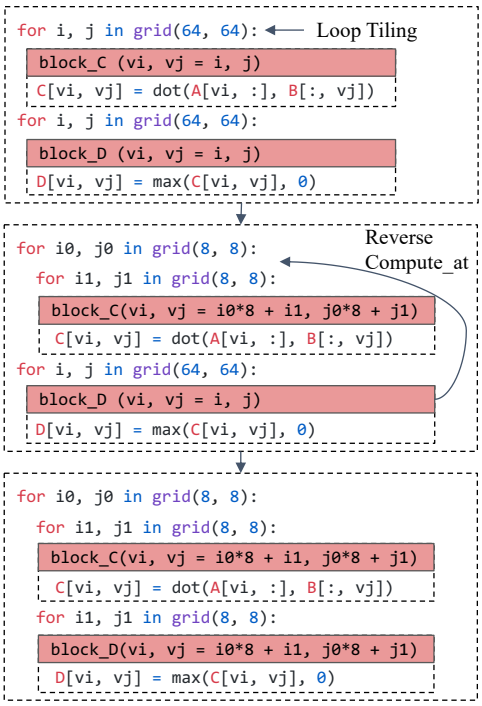

Figure 6: Loop transformations mutate outside loop nests but change nothing inside the block.

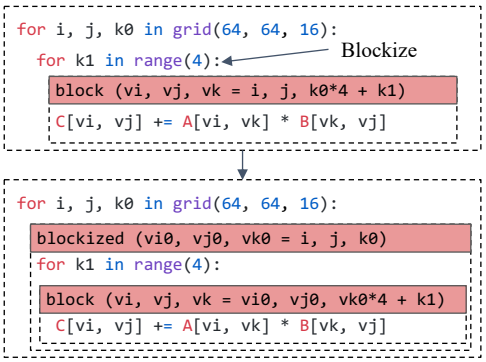

Figure 7: Blockization creates a new block to isolate inside computation and outside loop nesting.

and provide annotation hints such as vectorization and unrolling. Note that the block isolation does not prevent many important collaborative optimization across blocks (e.g. inlining, cooperative fetching). Our loop transformations cover the loop transformations provided by previous works which allows TensorIR to reproduce their search space as mentioned in section 2.

Blockization. The loop transformation primitives preserve the overall hierarchy of blocks. As we alluded in section 2, sometimes dividing the problem by isolating a sub-region computation into a new sub-block is helpful. We call this transformation *blockization* Figure 7. A blockized program is no longer scalar-based as the

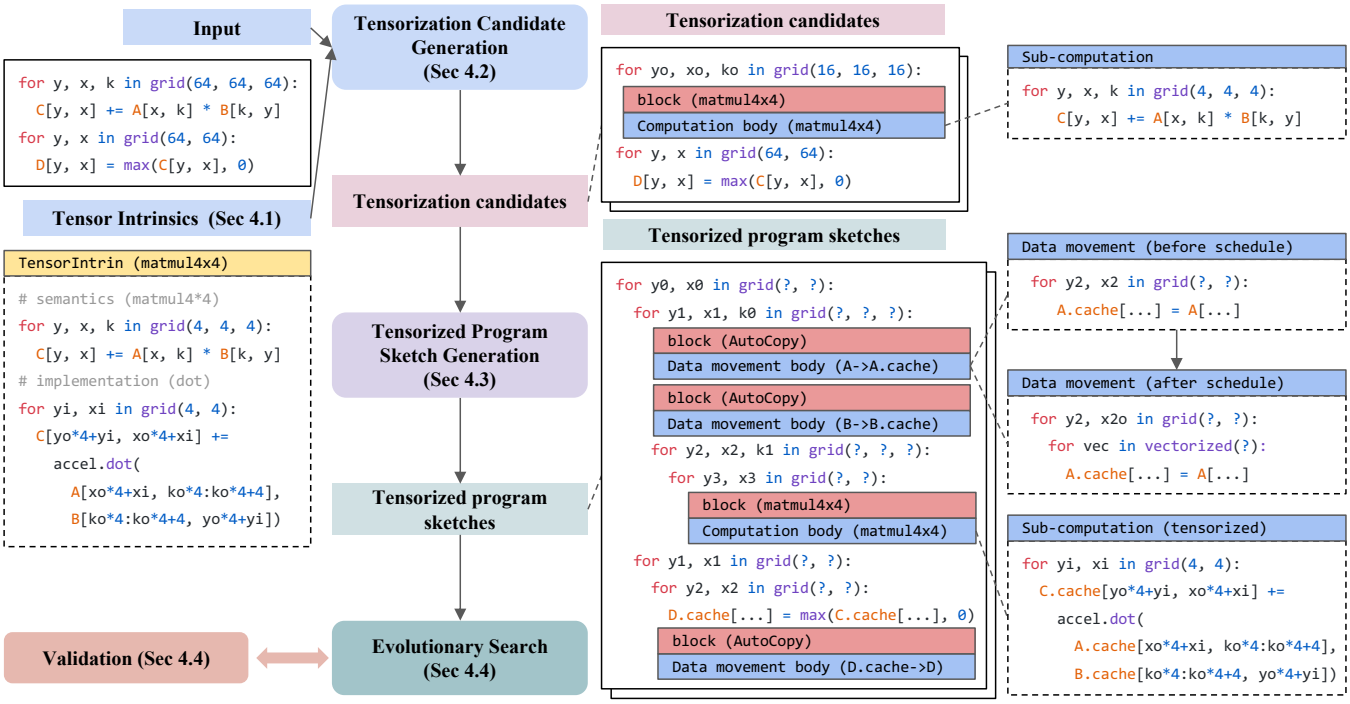

Figure 8: Automatic optimization for tensorized program with hardware intrinsics. We take 64x64x64 matrix multiplication followed by a RELU operator as the input workload and 4x4x4 matmul as the synthetic tensor intrinsic which is implemented by a dot product instruction. The tensorization candidate generation step tiles the 64x64x64 GEMM into 4x4x4 sub-tiles and isolate the sub-computation. Then the tensorized program sketch generation step schedules the computation and insert the resulting data movement (AutoCopy) blocks which are scheduled independently. Finally, we use evolutionary search to fill the random decisions in sketches with a validation mechanism to filter out incorrect programs.

new sub-block corresponds to a tensorized computation. We can use blockization to isolate possible candidates for tensorization. Besides blockization, we also introduce primitives that can change the block hierarchies. For example, we provide caching primitives that introduce sub-blocks to cache input data into shared memory. We also provide back and forth transformations between a single reduction block and the corresponding init- and update blocks.

Separation of Scheduling and TensorIR. Many previous tensor compilers [9, 35] rely on a declarative scheduling language to construct a schedule tree. Adding new scheduling primitives to these compilers requires changes to both the schedule tree data structure and the corresponding lowering rule in these compilers. We take a different approach and implement each schedule primitive as a standalone transformation from one TensorIR program to another. This design is easier to extend, as different developers can develop new primitives concurrently based on a stable TensorIR abstraction. Additionally, developers can print out the program at any transformation stage for debugging and mix automatic rewriting with schedule transformations.

3.3 Validation

The blocks and their buffer read/write relations capture a complete picture of the original computation and are used to validate the correctness of loop nests and threading assignments.

Loop Nest Validation. Loop nest validation checks whether the iterator binding provided by the loop nests matches the constraints of the iterator domain, including the domain size and iterator independence information. For example, if two data-parallel block iterators are bound as $v_1 = i; v_2 = i * 2$, then the corresponding program is invalid because v_1 and v_2 are not independent. But $v_1 = i/4; v_2 = i\%4$ can be a legal binding. We build pattern-matchers to find a quasi-affine mapping from the loop iterators to the block iterator variables and use the pattern to validate the independence and domain of the bindings. Besides the iterator domain validation, it is also important to check the producer-consumer relations to make sure producer blocks that write to buffer regions always cover the read region of downstream consumers.

Threading Validation. When building a program for GPUs and other accelerators with threading support, we also need to do additional validations with respect to the threading and memory hierarchies. We do three kinds of validations:

- **Thread binding**: Ensure different iterators bound to the same thread are consistent and meet the launching constraints of the backend.
- **Cooperative memory access**: For blocks that produce buffers stored in shared memory collaboratively across threads, we need to ensure the block covers downstream requirements from all the threads in the same group. Meanwhile, upstream blocks that provide inputs for this block need to cover the read requirement of this block from all the threads in this group.
- **Execution scope**: Validate that tensor intrinsic runs at the correct execution scope (e.g., TensorCore needs to run at the warp-level).

Correctness of Schedule Primitives. We add checks to each schedule primitive to ensure the correctness of the transformation. When a schedule primitive only changes the loop nests, we can also use the validation procedure to ensure correctness. Because the block iteration domains and dependencies stay the same in these cases. We find primitive-specific necessary conditions for schedule primitives that change the blocks (e.g., blockization).

Note that loop nest validation and threading validation are used as checks to filter out invalid TensorIR programs and schedule primitive checks are used to ensure the equivalence of TensorIR programs before and after transformations. Users will get warning or error information when they are incorrectly manually crafting, importing and scheduling TensorIR programs. When users use the compiler to generate programs automatically which will be discussed in section 4, validation can help filter out false positive cases during the exploration in the search space. Hence, both user programs and compiled programs will benefit from the validation.

3.4 Programming Effort

As shown in Figure 4, we provided a Python-AST dialect of TensorIR to allow developers directly construct, dump, inspect, modify, and transform TensorIR programs in Python. The program effort will be high if users need to specify all the computation and optimizations manually. Our framework allows users to import models from TensorFlow/PyTorch and automatically generates TensorIR programs from the high-level operators. Additionally, the system automatically provides the optimizations, such as tiling and caching, for a given hardware platform through the auto-scheduling (Section section 4). We still allow users to write TensorIR in Python dialect when they want customized operators. In these cases, the system provides optimization transformations automatically. As a result, the programming effort is usually minimized.

4 AUTO-SCHEDULING TENSORIZED PROGRAMS

In the last section, we introduced TensorIR abstraction and a set of transformation primitives. In order to fully make use of the set of improvements, we need an automatic solution to optimize over a set of transformations and map computations to the native tensor intrinsics. In this section, we describe a tensorization-aware automatic scheduler to solve this problem.

Figure 8 shows an overview of our approach. Our system takes a workload description from users and tensor intrinsic descriptions about the hardware platform as inputs. The auto-scheduler first generates candidates for tensorization by inspecting the computation pattern. It then generates program sketch candidates that use the tensorized computations and then decide the data movements according to the compute patterns. For a given search space induced by the tensorized program sketches, we perform evolutionary search guided by a learning-based cost model. The entire process centers itself around the tensorization and leverages the new block abstraction to isolate tensorized computations. We discuss the details of each step in the subsequent subsections.

4.1 Abstraction for Tensor Intrinsics

To make use of a tensor intrinsic in our optimization, we need a way to provide its semantics and backend implementation to the system. We leverage the *same* TensorIR abstraction to describe the tensor intrinsics of a given hardware backend. For each tensorized instruction, we introduce a TensorIntrin construct composed of two blocks. One block describes the computation *semantics*, and the other provides the low-level *implementation* of the tensorized computation.

In Figure 8's example, we use a normal loop nest with scalar body $C[i, j] \mathrel{+}= A[i, k] * B[k, j]$ to represent the computation semantics and implement the intrinsic using inner dot product instruction *accel.dot*. We also include the data type, storage scope, memory layout, and contiguity constraints through the multi-dimensional buffer specification in a TensorIntrin. Those constraints are used during the validation step.

Notably, tensor intrinsics are usually applied together with special memory scopes, data layouts, and corresponding load/store instructions in common platforms. For instance, on NVIDIA GPUs, if we decide to use *nvcuda::wmma::mma_sync* API to perform dense computation, then we need to apply *nvcuda::wmma::load_matrix_sync* and *nvcuda::wmma::store_matrix_sync* to prepare input operands and retrieve output results respectively. On ARM CPUs, microkernels like *a64_gemm_u8_8×12* require operands to be stored in interleaved layout. Developers can inform the system about these constraints by specifying special storage scopes for each input and output operands of the tensor computation.

4.2 Tensorization Candidate Generation

Given a pair of backend target and an input program, we first match the program body to possible TensorIntrin to generate tensorization candidates. The match is performed in a gradual way. We first match the expression pattern $C[.] \mathrel{+}= A[.] \times B[.]$ without considering the indices. We then refine the matches by proposing possible mappings between the indices. Figure 9 gives an example that walks through the matching process. In this example, we take a matrix multiplication intrinsic as the backend description. The computation of this tensor intrinsic can be described by the following formula

$$C[x, y] \mathrel{+}= A[x, k] \times B[k, y] \tag{1}$$

It is easy for us to match the tensor intrinsic to workloads like batch matrix multiplication, which can be described by

$$C[b, i, j] \mathrel{+}= A[b, i, r] \times B[b, r, j],$$

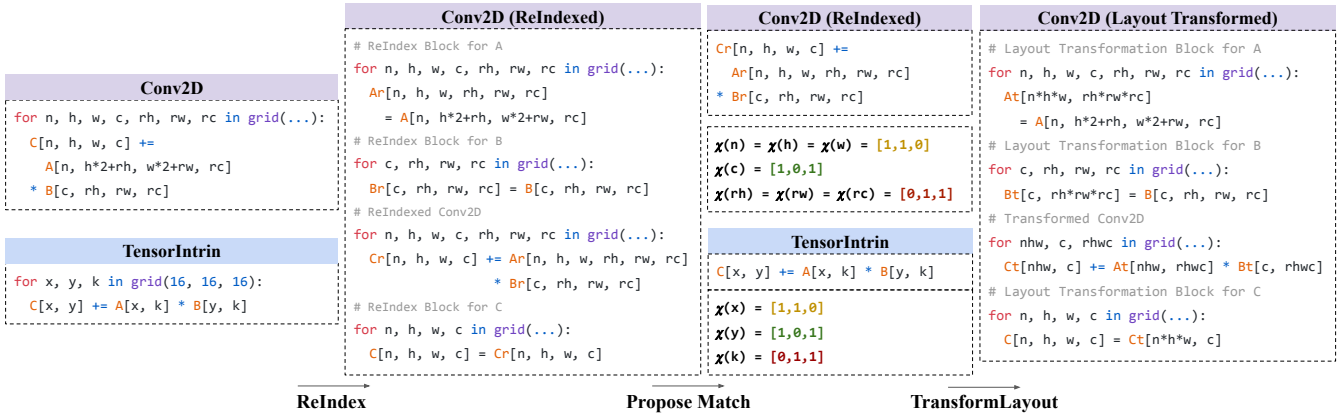

Figure 9: Example flow of tensorization candidate generation. We take standard NHWC 2D convolution as the input workload and 16x16x16 matrix multiplication as the hardware backend intrinsic. First, the system converts the buffer access expressions to intermediate iterators. Based on the buffer access patterns, we calculate characteristic functions for each iterator and build a mapping between iterators that share the same characteristic vector. The mapping further guides the transformation of block instance space and *ReIndex* buffers. Note that although the *ReIndex* stages of B and C are redundant, they will be inlined into consumers during the sketch generation phase and as a result do not affect the performance.

by mapping x, y, k to i, j, r. But for workloads like 2-dimensional Convolution (Conv2D) with more complex index expression patterns

$$C[n,h,w,co] \mathrel{+}= A[n, h*s_h + r_h * d_h, w*s_w + r_w * d_w, r_c] \times B[r_c, r_h, r_w, co],$$

the mapping between x, y, k and $n, h, w, c, r_h, r_w, r_c$ is not straightforward.

On these more general cases, we rewrite the computation expression into an equivalent form:

$$C[n,h,w,co] \mathrel{+}= A_r[n,h,w,r_h,r_w,r_c] \times B[r_c,r_h,r_w,co],$$
$$A_r[n,h,w,r_h,r_w,r_c] = A[n, h*s_h + r_h * d_h, w*s_w + r_w * d_w, r_c].$$

We call this transformation *ReIndex* which uses intermediate iterators that appear in the buffer access indices to rewrite the buffer access expressions. To match the new computation to the tensor intrinsic, we check the buffer access where each iterator appears. For example, we notice that n, h, w and x appear in indices of $A(Ar), C$, co and y appear in indices of B, C, and r_h, r_w, r_c and k appear in indices of A, C. We can then match the iterators in the computation to the iterators in the tensor intrinsic by inspecting their appearance patterns. Specifically, we map $\text{fuse}(n,h,y)$ to x, co to y, and $\text{fuse}(r_h, r_w, r_c)$ to k. Here fuse() is to fuse multiple iterators together and can be recursively defined by

$$\text{fuse}(i_1) = i_1$$
$$\text{fuse}(i_1, i_2, \ldots, i_r) = \text{fuse}(i_1, i_2, \ldots, i_{r-1}) * \text{extent}(i_r) + i_r,$$

where extent() is the extent of iterator i_r. We can then transform the computation to

$$C_t[\text{fuse}(n,h,w), co] \mathrel{+}= A_t[\text{fuse}(n,h,w), \text{fuse}(r_h, r_w, r_c)] \times B_t[\text{fuse}(r_h, r_w, r_c), co],$$

where

$$C_t[\text{fuse}(n,h,w), co] = C[n,h,w,co],$$
$$A_t[\text{fuse}(n,h,w), \text{fuse}(r_h,r_w,r_c)] = A_r[n,h,w,r_h,r_w,r_c]$$
$$B_t[\text{fuse}(r_h,r_w,r_c), co] = B[r_h,r_w,r_c,co].$$

We use this mapping to reshape the block instance space and the outer loops and transform the layout of *ReIndex* buffers. We insert layout rewrite blocks to rewrite A, B, C to A_t, B_t, C_t respectively and use A_t, B_t, C_t to rewrite the computation body. After these steps, the computation body is compatible with the tensor intrinsic.

Loop Reorganization and Early Blockize. Besides the computation body, we also need to ensure that the tensor computation region matches the description provided by the TensorIntrin. The shape of the reorganized block instance space might not be divisible by the sub-problem size of the tensor intrinsic. For each computation body from the last step, we do necessary padding on the computation block and input/output operands to the closest divisible shape. We then perform tiling to create inner loops to match the loop nest of the tensor intrinsic and further blockize the inner loop to isolate the corresponding tensor computations. Notably, the candidates generated in this step do not always lead to successful tensorizations. This is because other constraints, such as memory layout and threading depend on later transformations. These constraints are embedded in the tensorization candidates and checked during validation.

Formal Description of the Process. So far we gave a high-level overview of the tensorization candidate generation process. In the the remainder part of this subsection, we provide a formal description of the process. Suppose the intrinsic scalar expression can be formalized as

$$O[\mathbf{v}_0] = f(O[\mathbf{v}_0], I_1[\mathbf{v}_1], I_2[\mathbf{v}_2], \ldots, I_k[\mathbf{v}_k]). \quad (2)$$

where O is the output operand, $I_{[1:k]}$ are input operands, $\mathbf{v}$ is the set of iterators that parameterized this computation, $\mathbf{v}_{[0:k]}$ are all lists of iterators that belong to $\mathbf{v}$, and f is the expression pattern detected. Note that it accommodates common dot product and matrix multiplication intrinsics. Furthermore, suppose that the workload scalar expression can be formalized as

$$\tilde{O}[g_0(\tilde{\mathbf{v}}_0)] = f(\tilde{O}[g_0(\tilde{\mathbf{v}}_0)], \tilde{I}_1[g_1(\tilde{\mathbf{v}}_1)], \ldots, \tilde{I}_k[g_k(\tilde{\mathbf{v}}_k)]),$$

where $\tilde{O}$, $\tilde{I}_{[1:k]}$, $\tilde{\mathbf{v}}_{[0:k]}$ corresponds to their counterparts and f is exactly the same. $g_{[0:k]}$ are mappings that map lists of iterators to the actual buffer access position. In our Conv2D case, for instance, we have $g_A(n, h, w, r_h, r_w, r_c) = (n, h{*}s_h{+}r_h{*}d_h, w{*}s_w{+}r_w{*}d_w, r_c)$.

To reduce the problem to a simpler canonical form, we apply the *ReIndex* schedule transformation, which creates an intermediate cache buffer for an operand but with the layout changed according to the iterators. Formally, if we run *ReIndex* $\tilde{I}_1[g_1(\tilde{\mathbf{v}}_1)]$, we create the following rewrite block before the computation

$$\hat{I}_1[\tilde{\mathbf{v}}_1] = \tilde{I}_1[g_1(\tilde{\mathbf{v}}_1)].$$

Then if we apply *ReIndex* to all the operands, the workload scalar expression is reduced to

$$\hat{O}[\tilde{\mathbf{v}}_0] = f(\hat{O}[\tilde{\mathbf{v}}_0], \hat{I}_1[\tilde{\mathbf{v}}_1], \ldots, \hat{I}_k[\tilde{\mathbf{v}}_k]), \tag{3}$$

where buffer access indices in both 2 and 3 directly correspond to iterators.

To match 2 and 3, we define the characteristic vector $\chi(v) \in \{0, 1\}^{k+1}$ of an iterator $v \in \mathbf{v}$ by inspecting whether each of $\mathbf{v}_0, \mathbf{v}_1, \mathbf{v}_2, \ldots \mathbf{v}_k$ contains v. Formally,

$$\chi(v)_i = [v \in \mathbf{v}_i] \quad i \in [0, k],$$

where [] is the Iverson bracket that returns 1 if the corresponding condition is true or 0 otherwise (Figure 9). We can successfully propose the mapping as long as $\forall v \in \mathbf{v}, \exists \tilde{v} \in \tilde{\mathbf{v}}, \chi(v) = \tilde{\chi}(\tilde{v})$. In the current implementation, we can further safely assume that iterators in $\mathbf{v}$ all have different characteristic vectors. Then for all $v \in \mathbf{v}$, we fuse all such $\tilde{v}$ where $\chi(v) = \tilde{\chi}(\tilde{v})$ and map the fused iterator to v.

Notably, the iterator order inside each of $\mathbf{v}_{[0:k]}$ or $\tilde{\mathbf{v}}_{[0:k]}$ does not affect the value of characteristic function χ or $\tilde{\chi}$. But when we fuse all $\tilde{v}$ where $\chi(v) = \tilde{\chi}(\tilde{v})$, the order of fusion affects how the operands are reorganized in memory to be compatible with the tensor intrinsic. Our implementation now uses a default order for all the workloads and can generalize to different fusion orders in the future.

4.3 Tensorized Program Sketch Generation

For a given set of tensorization candidates, we need to construct a large program search space that contains the tensorization. We generalize existing hierarchical search space generation [48] to tensor computations. We construct the search space by generating program sketches that contain the tensorized computation, then enumerate over choices induced by the generated sketches. As shown in the right part in Figure 8, a program sketch fixes parts of program structures while leaving space for remaining choices of parameters such as loop tiling size and computation caching decisions. We generate sketches by applying pre-defined sketch generation rules iteratively. Importantly, we need to build sketch generation rules that work on tensorized computations by looking at the block signatures and make use of the access region information during our analysis.

Data Movement as First-Class Citizen. Existing auto-schedulers for tensor programs focus their designs on the schedule of computations and treat data movement between different memory scopes with secondary priority. However, since tensor intrinsics vastly improve the throughput of computations, data movements become the bottleneck of tensor programs. Moreover, data movement decisions usually depend on computation schedule decisions like tilings, thread bindings, execution scopes, and producer-consumer data flow granularity. We take these insights and bring data movements as first-class citizens in our automatic scheduler and decouple them from computation schedules. Specifically, we insert *AutoCopy* blocks into the places where the sketch generation rules decide to perform data movements (Figure 8). The copy block hides the memory schedule details and only exposes the necessary buffer access information at the block signature level. The isolated copy blocks allow the sketch generation to independently make computation schedule decisions without considering how to do data movements. The body of the *AutoCopy* block describes the details of the data movement task, including buffer position mapping, threading, and storage scope requirements. A data movement scheduler takes this information as input and performs memory-related schedule transformations, such as inserting intermediate cache stages, utilizing data movement tensor intrinsics, vectorization, cooperative fetching, or stride padding to avoid bank conflicts.

4.4 Evolutionary Search

After the tensorized program sketch generation phase, we can get billions of possible induced programs. We use evolutionary search to explore the space and find an optimized tensorized program. Our search starts from random initializations of choices for given program sketches. We then perform mutations on the current set of programs. We then select promising programs from the mutated candidates and benchmark them on our hardware backend of interest. We collect data from the evaluation phase to update the learned cost model.

Cost Model for Tensorized Computation. We build a boosting tree ensemble [7] based cost models that use features extracted from the program. The feature vector contains information related to memory access patterns, reuse, and loop annotations. Importantly, we extract features from both block signatures in an isolated way as well as the body of the block (e.g., to mark the use of Tensor Core). Our cost model can be viewed as a generalization of previous approaches to tensorized programs. We believe an effective cost model for tensorized programs is a promising area for future research.

Validation. Randomly mutating programs during the search can generate invalid programs due to the unmet constraints of tensor intrinsic or invalid loop nest candidates. The possibility of false positives necessitates a validation step during the search. We apply techniques in subsection 3.3 to validate a program candidate in the evolutionary search to identify and reject invalid programs. The

validation step reduces the burden on evolutionary search algorithms and allows us to generate a small number of false positives during the search.

5 EVALUATION

We implement TensorIR on top of Apache TVM [9]. Notably, the insights described in the paper can benefit other machine learning compilation frameworks as well. This section provides evaluations to answer the following questions:

- Can TensorIR optimize common set of machine learning operators (§5.1)?
- Can TensorIR bring performance boost to end-to-end network execution (§5.2)?
- Can TensorIR support tensor intrinsics on different hardware platforms (§5.3)?

To evaluate TensorIR along those axes, we compare our solution to existing machine learning compilation solutions on GPU and CPU platforms. We will discuss the specific setups in the corresponding subsections.

Additionally, we are interested in the following question throughout all evaluations: How does TensorIR compare to vendor-specific libraries and frameworks that rely on these libraries? Importantly, most of these libraries are heavily optimized by a dedicated team of engineers. In all of these settings, TensorIR performs end-to-end automatic optimization without the need to call into external libraries.

5.1 Single Operator Evaluation

This section evaluates TensorIR on operators in deep learning models. We pick a common collection of workloads, including: 1D convolution (C1D), 2D convolution (C2D), 3D convolution (C3D), depth-wise convolution (DEP), dilated convolution (DIL), general matrix multiply (GMM), group convolution (GRP), and transposed 2D convolution (T2D). The evaluations are done on an NVIDIA RTX 3080 platform with Tensor Cores. We pick this platform as it has a wide spectrum of machine learning compiler solutions and libraries that we can use as comparison reference points. We use float16 as the input and accumulator data type for all operators. We include TVM (commit: 27b0aad5, with auto-scheduler [48]) and AMOS (commit: 6aee6fe2) as two machine learning compiler baselines. Finally, we compare against two vendor-specific solutions: CUTLASS (version 2.9) and TensorRT (PyTorch-TensorRT container Release 22.06).

Comparisons to Machine Learning Compilers. Figure 10 shows the comparisons to AMOS and TVM. TVM [9] works well on less compute-intensive workloads (*e.g.*DEP), but has poor performance on heavy ones (*e.g.*C2D, C3D, GMM) due to the limited Tensor Core support. AMOS [49] can use Tensor Core for every workload but not doing as well as TensorIR. Overall, TensorIR brings up to 7.5× number of improvement over existing machine learning compilation solutions. These improvements come from better abstraction and automatic scheduling that leverages tensor compute intrinsics and corresponding data movements.

Comparisons to Platform Specific Libraries. Figure 11 shows the comparisons of TensorIR to two platform specific solutions: CUTLASS [23] and TensorRT [32]. TensorIR outperforms the baselines

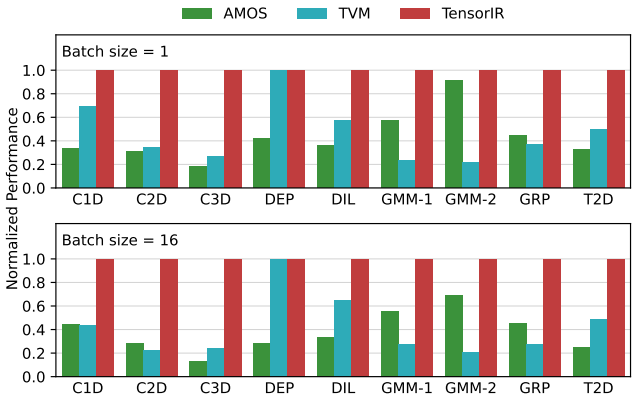

Figure 10: Single operator comparison to existing machine learning compilers on Nvidia GPU. TensorIR brings up to 7.5x speed across workloads.

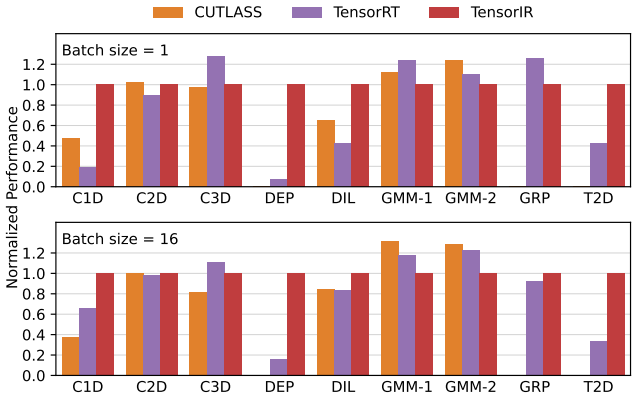

Figure 11: Single operator comparison to platform-specific libraries. We did not show the numbers of CUTLASS on DEP, GRP, and T2D as the library does not support them. TensorIR outperforms the baselines on C1D, C2D, DEP, T2D, and DIL by up to 13.9x and gets to more than 75% throughput on C3D, GMM, and GRP.

on C1D, C2D, DEP, T2D, and DIL by up to 13.9×. These results shows the advantage of automatic optimizations provided by TensorIR. Notably, TensorIR gets to more than 75% on C3D, GRP an GMM. These results show that even on workloads that are intensively optimized by dedicated engineering teams, TensorIR can still get close to or match existing vendor-specific solutions. We expect the remaining gap continues to close as we bring additional insights from these libraries to TensorIR. In all cases, the baseline solutions are optimized by dedicated engineering teams, while TensorIR enables automated compilation for a given tensor intrinsic declaration.

5.2 End-to-end Model Evaluation

In this section, we evaluate the impacts that TensorIR can bring to end-to-end model execution. We evaluate our solutions on four widely-used models [13, 14, 18, 19] on on NVIDIA RTX 3080. We include TVM and AMOS as machine learning compiler baselines. Additionally, we also include PyTorch (version 1.13.0.dev20220612) as a framework reference point. Finally, we include TensorRT, which is a vendor-specific solution that is heavily optimized by engineering teams at Nvidia.

The results are shown in Figure 12. TensorIR outperforms PyTorch, TVM, and AMOS by 1.2 – 8.8×. Additionally, TensorIR brings 30% better performance on MobileNet V2 comparing to TensorRT, and achieves the 88% – 100% throughput on ResNet50 and BERT_large. Additionally, TensorIR can automatically support emerging models such as Vision Transformer, which TensorRT does not yet support. These results show that our abstraction and the automatic scheduler can bring close or even better performance than the best effort libraries on common machine learning models. Additionally, the automated solution enables us to bring faster support for emerging models.

Tuning time is an important factor of practical usability for search based automatic deep learning compilers. We compare the end-to-end tuning time of TVM and TensorIR which is shown in Table 1. Our framework tunes up to 2x faster compared with TVM, and this improvement comes from two aspects. Firstly, hardware profiling contributes the most to the tuning time, and auto tensorization of TensorIR generates faster programs compared with TVM due to the utilization of Tensor Core and hence the profiling time is less accordingly. Secondly, our divide and conquer approach divides and isolates the problem space into outer loop nests and inner body. The inner body is tensorized with hardware intrinsics and we search over the loop transformations of outer loops, which results in a smaller search space. The search time can be tolerated when we deploy these models to many devices for months in production. TensorIR can eliminate search time further by caching historical cost models and search records. So no search is needed to build a model for an operator already tuned.

5.3 ARM CPU Evaluation

The last two subsections evaluate TensorIR on an Nvidia GPU. In this section, we study how easy it is to generalize TensorIR to different platforms. We evaluate results on an ARM platform by providing the description with 8-bit integer dot(*sdot*). This instruction is different from the Tensor Core used in the last two subsections. Importantly, we use the same TensorIR framework by providing the new description of the tensor intrinsic to the system. The evaluations are done on an AWS Graviton2 CPU.

Single Operator Results. We evaluate the results on two commonly used operators: C2D and GMM. We include TVM as a machine learning compiler baseline and ARMComputeLib [3] as a platform-specific library baseline. The results are shown in Figure 13. TensorIR achieves up to 12.5× speed up compared with TVM thanks to the ability to leverage native hardware acceleration. In the meantime, TensorIR reaches 85% – 105% throughput of ARMComputeLib [3], showing our ability to get to the same level of performance as vendor-specific solutions on this platform.

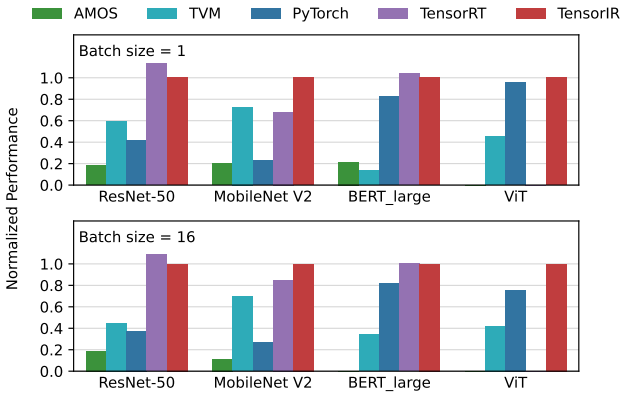

Figure 12: End-to-End model evaluations on NVIDIA GPU. TensorIR significantly outperforms existing machine learning compilation solutions and achieves similar or better throughputs on popular networks compared with the inference libraries on GPUs. TensorIR get better performance on ViT, an emerging model that TensorRT does not yet support.

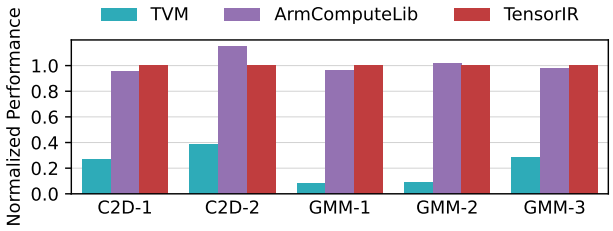

Figure 13: Single operator evaluations on ARM CPU. TensorIR get up to 12.6x faster than TVM due to the use of native tensor instrinsic acceleration. It also gets to the same level of performance as heavily optimized platform specific library (ArmComputeLib).

Tuning time		
Model	TVM (min)	TensorIR (min)
ResNet-50	308	156
MobileNet-v2	292	261
BERT	410	189
ViT	247	145

Table 1: Tuning time comparison of end-to-end models on NVIDIA GPU. TensorIR tunes up to 2x faster.

End-to-End Results. Finally, we evaluate the end-to-end neural network executions on this platform. Our baselines include PyTorch and TVM. We achieve up to 2.3× on this platform in Figure 14. Notably, PyTorch contains a specialized quantized model support with QNNPACK [28] backend. However, QNNPACK has not yet added *sdot* support. This result alludes to the maintenance cost for these frameworks to keep up with hardware changes. TensorIR can

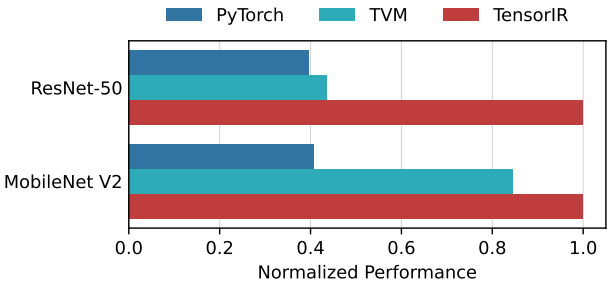

Figure 14: End-to-end evaluation results on ARM CPU. TensorIR outperform with 1.2x–2.5x than PyTorch and TVM.

help to reduce the maintenance burden through automation and still bring competitive performance to hand-optimized systems.

6 RELATED WORKS

Deep learning frameworks [1, 8, 33] optimize deep neural networks by invoking vendor optimized libraries (*e.g.*, cuDNN [11], MKL-DNN [20], TensorRT [32], ArmComputeLibrary [3]). Libraries have engineering development costs and are specific to a particular hardware. TensorIR complements library developments to enable better coverage and reduce development costs by automatically providing comparable performance with vendor libraries. These frameworks can leverage TensorIR to generate optimized tensorized programs for various hardware backends.

Compute-intensive linear algebra operators such as matrix multiplication and dot products has been a long-standing optimization target in HPC community (e.g., CUTLASS [23]) due to their importance in scientific computation. The divide-and-conquer is a typical optimization technique in HPC and ML engineering. TensorIR takes these ideas and generalizes them to an abstraction that allows automatic tensorization.

Machine learning and tensor compilers introduce different abstractions for tensor programs. Halide [35] and TVM [9] use a scheduling language that can describe loop optimization primitives of loop nests with a scalar body. Tensor Comprehensions [43], Tiramisu [5] and MLIR/Affine [26] use polyhedral model [42] to analyze loop nest dependencies. These works optimize loop nests with scalar computation in a bottom-up way. Fireiron [17] and Stripe [46] use nested polyhedral structures to model tensor programs in a top-down fashion. TensorIR combines insights from both approaches and generalizes the representation to tensorized programs. IREE [40] is a compiler chain for end-to-end compilation flow which utilizes platform-specific optimization pipelines. TensorIR focuses on automating the tensorization process to generate optimized code for multiple platforms without human intervention. TACO [12, 24, 38] is a compiler for sparse tensor algebra. Cortex [15] generalized tensor compilation to recursive computations. Our work is orthogonal to these efforts. We believe the TensorIR abstraction can be combined with insights from these works in the future to enable an even broader range of computations.

Automation is an essential topic in machine learning compilation and tensor program optimization. AutoTVM [10] introduced a learning-based approach to optimize tensor programs via a learned cost model and template-guided search. Triton [41] introduces a tile-based template representation for effective program optimization. FlexTensor [50] automatically generates the template. Halide builds an automatic scheduler using Monte-Carlo tree search [2]. Ansor [48] improves automatic scheduling using a hierarchical search space. Our automatic scheduling algorithm takes lessons from these approaches and generalizes them to tensorized computation best for domain-specific hardware acceleration.

Auto-vectorization [25, 36] is a long-standing topic in compiler research. Tensorization can be viewed as a generalization of the vectorization problem to enable tensor intrinsic in modern accelerators [4, 21, 30, 31]. There are some existing works[6, 45, 47, 49] on this topic. AKG [47] uses the polyhedral method to explore tensorized search space, UNIT [45] introduces a generic flow for tensorization, while AMOS [49] enables automatic mapping to tensorized intrinsic through tensor expression. Our method generalizes these previous approaches by proposing a novel abstraction for tensorization computation and jointly performing tensorization along with other optimizations. TensorIR serves as a foundation to further develop tensorization-aware automatic scheduling methods.

7 CONCLUSION

We propose TensorIR, an abstraction for automatic tensorized program optimization. We design a key abstraction called block that can isolate tensorized computations and provide effective transformation primitives for program optimization. We build an automatic scheduling algorithm that performs tensorization jointly with other optimizations and generates performant programs. We hope this work will encourage additional studies of tensorized program optimization and provide new opportunities for hardware and software specialization.

ACKNOWLEDGMENTS

This work is supported in part by a gift from Oppo. We would like to thank Masahiro Masuda from OctoML and the TVM community for helpful discussion and support of the work. We would like to thank anonymous reviewers and our shepherd Christian DeLozier for their helpful feedback and discussions.

REFERENCES

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 265–283.

[2] Andrew Adams, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, Kayvon Fatahalian, Frédo Durand, et al. 2019. Learning to optimize halide with tree search and random programs. *ACM Transactions on Graphics (TOG)* 38, 4 (2019), 1–12.

[3] ARM. 2017. ARM Compute Library. https://github.com/ARM-software/ComputeLibrary/

[4] ARM. 2017. Exploring the Arm dot product instructions. https://community.arm.com/developer/tools-software/tools/b/tools-software-ides-blog/posts/exploring-the-arm-dot-product-instructions

[5] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. 2019. Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code. In *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization* (Washington, DC, USA) *(CGO 2019)*. IEEE Press, 193–205.

[6] Somashekaracharya G Bhaskaracharya, Julien Demouth, and Vinod Grover. 2020. Automatic kernel generation for volta tensor cores. *arXiv preprint arXiv:2006.12645* (2020).

[7] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (San Francisco, California, USA) *(KDD '16)*. ACM, New York, NY, USA, 785–794.

[8] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[9] 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.

[10] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. Learning to optimize tensor programs. In *Advances in Neural Information Processing Systems*. 3389–3400.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759* (2014).

[12] Stephen Chou, Fredrik Kjolstad, and Saman Amarasinghe. 2020. Automatic Generation of Efficient Sparse Tensor Format Conversion Routines. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation* (London, UK) *(PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 823–838. https://doi.org/10.1145/3385412.3385963

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

[14] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[15] Pratik Fegade, Tianqi Chen, Phil Gibbons, and Todd Mowry. 2021. Cortex: A Compiler for Recursive Deep Learning Models. *MLSys* (2021).

[16] Basilio B. Fraguela, Jia Guo, Ganesh Bikshandi, María J. Garzarán, Gheorghe Almási, José Moreira, and David Padua. 2004. The Hierarchically Tiled Arrays Programming Approach. In *Proceedings of the 7th Workshop on Workshop on Languages, Compilers, and Run-Time Support for Scalable Systems* (Houston, Texas, USA) *(LCR '04)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/1066650.1066657

[17] Bastian Hagedorn, Archibald Samuel Elliott, Henrik Barthels, Rastislav Bodik, and Vinod Grover. 2020. Fireiron: A Data-Movement-Aware Scheduling Language for GPUs. In *Proceedings of the ACM International Conference on Parallel Architectures and Compilation Techniques*. 71–82.

[18] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[19] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[20] Intel. 2017. Intel® Math Kernel Library for Deep Learning Networks. https://software.intel.com/en-us/articles/intel-mkl-dnn-part-1-library-overview-and-installation

[21] Intel. 2019. Introduction to Intel® Deep Learning Boost on Second Generation Intel® Xeon® Scalable Processors. https://software.intel.com/content/www/us/en/develop/articles/introduction-to-intel-deep-learning-boost-on-second-generation-intel-xeon-scalable.html

[22] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 1–12.

[23] Andrew Kerr, Haicheng Wu, Manish Gupta, Dustyn Blasig, Pradeep Ramini, Duane Merrill, Aniket Shivam, Piotr Majcher, Paul Springer, Markus Hohnerbach, Jin Wang, and Matt Nicely. 2022. *CUTLASS*. https://github.com/NVIDIA/cutlass

[24] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. 2017. The Tensor Algebra Compiler. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 77 (Oct. 2017), 29 pages. https://doi.org/10.1145/3133901

[25] Martin Kong, Richard Veras, Kevin Stock, Franz Franchetti, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. 2013. When polyhedral transformations meet SIMD code generation. In *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*. 127–138.

[26] 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)*. IEEE, 2–14.

[27] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[28] Hao Lu Marat Dukhan, Yiming Wu. 2018. QNNPACK: Open source library for optimized mobile deep learning. https://engineering.fb.com/2018/10/29/ml-applications/qnnpack/

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[30] Thierry Moreau, Tianqi Chen, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. VTA: an open hardware-software stack for deep learning. *arXiv preprint arXiv:1807.04188* (2018).

[31] Nvidia. 2017. NVIDIA Tensor Cores. https://www.nvidia.com/en-us/data-center/tensorcore/

[32] Nvidia. 2017. NVIDIA TensorRT: Programmable Inference Accelerator. https://developer.nvidia.com/tensorrt

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*. 8026–8037.

[34] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[35] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. 2013. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. *Acm Sigplan Notices* 48, 6 (2013), 519–530.

[36] Ira Rosen, Dorit Nuzman, and Ayal Zaks. 2007. Loop-aware SLP in GCC. In *GCC Developers Summit*. Citeseer.

[37] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347* (2017).

[38] Ryan Senanayake, Changwan Hong, Ziheng Wang, Amalee Wilson, Stephen Chou, Shoaib Kamil, Saman Amarasinghe, and Fredrik Kjolstad. 2020. A Sparse Iteration Space Transformation Framework for Sparse Tensor Algebra. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 158 (Nov. 2020), 30 pages. https://doi.org/10.1145/3428226

[39] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[40] The IREE Authors. 2019. *IREE*. https://github.com/iree-org/iree

[41] 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

[42] Nicolas Vasilache, Cédric Bastoul, and Albert Cohen. 2006. Polyhedral code generation in the real world. In *International Conference on Compiler Construction*. Springer, 185–201.

[43] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary DeVito, William S Moses, Sven Verdoolaege, Andrew Adams, and Albert Cohen. 2018. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions. *arXiv preprint arXiv:1802.04730* (2018).

[44] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *arXiv preprint arXiv:1706.03762* (2017).

[45] Jian Weng, Animesh Jain, Jie Wang, Leyuan Wang, Yida Wang, and Tony Nowatzki. 2021. UNIT: Unifying tensorized instruction compilation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 77–89.

[46] Tim Zerrell and Jeremy Bruestle. 2019. Stripe: Tensor compilation via the nested polyhedral model. *arXiv preprint arXiv:1903.06498* (2019).

[47] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng Li, et al. 2021. AKG: automatic kernel generation for neural processing units using polyhedral transformations. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1233–1248.

[48] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. 2020. Ansor: generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*. 863–879.

[49] Size Zheng, Renze Chen, Anjiang Wei, Yicheng Jin, Qin Han, Liqiang Lu, Bingyang Wu, Xiuhong Li, Shengen Yan, and Yun Liang. 2022. AMOS: enabling automatic mapping for tensor computations on spatial accelerators with hardware abstraction. In *ISCA '22: The 49th Annual International Symposium on Computer Architecture, New York, New York, USA, June 18 - 22, 2022*, Valentina Salapura, Mohamed Zahran, Fred Chong, and Lingjia Tang (Eds.). ACM, 874–887.

https://doi.org/10.1145/3470496.3527440
[50] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. 2020. FlexTensor: An Automatic Schedule Exploration and Optimization Framework for Tensor Computation on Heterogeneous System. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 859–873.