SCIENCE CHINA
Information Sciences

• RESEARCH PAPER •

October 2024, Vol. 67, Iss. 10, 200403:1–200403:18
https://doi.org/10.1007/s11432-024-4071-6

Special Topic: AI Chips and Systems for Large Language Models

TSCompiler: efficient compilation framework for dynamic-shape models

Xiang LUO[1], Chen ZHANG[2*], Chenbo GENG[3], Yanzhi YI[4], Jiahui HU[4], Renwei ZHANG[4], Zhen ZHANG[5], Gianpietro CONSOLARO[5], Fan YANG[6], Tun LU[1], Ning GU[1] & Li SHANG[1*]

[1]*School of Computer Science, Fudan University, Shanghai* 200433*, China;*
[2]*School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University, Shanghai* 200240*, China;*
[3]*School of Computer Science and Technology, Harbin Institute of Technology, Harbin* 150001*, China;*
[4]*Huawei Technologies Co., Ltd., Beijing* 100095*, China;*
[5]*Huawei Paris Research Center, Paris* 92100*, France;*
[6]*School of Microelectronics, Fudan University, Shanghai* 201203*, China*

Received 29 February 2024/Revised 15 May 2024/Accepted 11 June 2024/Published online 13 September 2024

Abstract Today's deep learning models face an increasing demand to handle dynamic shape tensors and computation whose shape information remains unknown at compile time and varies in a nearly infinite range at runtime. This shape dynamism brings tremendous challenges for existing compilation pipelines designed for static models which optimize tensor programs relying on exact shape values. This paper presents TSCompiler, an end-to-end compilation framework for dynamic shape models. TSCompiler first proposes a symbolic shape propagation algorithm to recover symbolic shape information at compile time to enable subsequent optimizations. TSCompiler then partitions the shape-annotated computation graph into multiple subgraphs and fine-tunes the backbone operators from the subgraph within a hardware-aligned search space to find a collection of high-performance schedules. TSCompiler can propagate the explored backbone schedule to other fusion groups within the same subgraph to generate a set of parameterized tensor programs for fused cases based on dependence analysis. At runtime, TSCompiler utilizes an occupancy-targeted cost model to select from pre-compiled tensor programs for varied tensor shapes. Extensive evaluations show that TSCompiler can achieve state-of-the-art speedups for dynamic shape models. For example, we can improve kernel efficiency by up to 3.97× on NVIDIA RTX3090, and 10.30× on NVIDIA A100 and achieve up to five orders of magnitude speedups on end-to-end latency.

Keywords machine learning, tensor compilers, dynamic shape, operator fusion, code generation, auto-tuning

1 Introduction

With the widespread deployments across varied scenarios, deep learning models are now struggling with the increasing demand to handle dynamic shape tensors and computation. Large Language models (LLMs), for instance, are expected to seamlessly process input sequences with lengths that span from just a few to several million tokens [1]. This shape dynamism is entirely user-defined and unpredictable in advance which presents substantial challenges for deep learning systems and compilers [2, 3].

Traditional deep learning compilers, like TVM [2] or Ansor [4], have relied on static tensor shapes to optimize critical aspects of computation, such as workloads' tiling and compute scheduling. As a result, the loss of shape information at compile time together with the increasing variability in data size and tensor shapes at runtime has forced deep learning compilers to adopt a just-in-time (JIT) compilation approach. In JIT compilation, schedules of kernels are deferred from compile time until runtime, contingent upon the specific shape of the data. Unfortunately, due to the intricate nature of high-dimensional deep learning computations and hardware characteristics of accelerators, JIT compilation can

* Corresponding author (email: chenzhang.sjtu@sjtu.edu.cn, lishang@fudan.edu.cn)

introduce significant delays, often stretching from several minutes to even hours for individual kernels [5]. Such delays are entirely impractical in today's AI applications, which demand low latency and high throughput.

While several approaches [6, 7] have been developed to address the dynamism in deep neural networks (DNNs) models, they primarily focused on optimizing individual operators, often overlooking the broader impact on graph-level optimizations. For example, dynamic dimensions in tensors often preclude optimizations commonly applied in static graphs, such as kernel fusion and constant folding. Given that graph-level optimizations are critical for enhancing the performance of DNN programs, the existing methods for optimizing and compiling tensors with dynamic dimensions fall short.

Optimizing the dynamic-shape computation graph is challenging as it requires shape values at compile time to enable various compilation pipelines. To enable graph-level optimizations, such as operator fusion, compilers must determine whether the index space of two adjacent operators matches. Existing systems [8] either use attributes, like Any, to represent unknown shape dimensions, which only works for one-dimension dynamism, such as dynamic batching or they only focus on static parts of the graph. The latter approach thus fails to optimize the memory access holistically. Furthermore, due to the absence of exact shape information, fused subgraphs generated from graph-level optimizations cannot be effectively optimized with auto-schedulers [4], leading to the inefficiency of the resulting tensor programs.

In this paper, we introduce TSCompiler, a deep learning compiler that serves dynamic-shape models with compile-time and runtime co-scheduling. Our approach involves a two-phase process: offline compilation and runtime program loading. During the "offline" phase, to enable the graph-level optimizations, we first introduce symbolic shape variables to represent unknown dimensions at compile time which can be propagated across the whole graph with the dataflow analysis to recover shape information. The symbolic shape representation facilitates comprehensive operator fusion to partition the whole graph into fused subgraphs. To further optimize the schedule of symbolic-shape subgraphs, TSCompiler then constructs a hardware-aligned search space for the backbone operator within the subgraph constrained with hardware characteristics and explores the search space navigated by a Bayesian optimization to find a collection of high-performance schedules. TSCompiler propagates the backbone schedule to other fusion groups within the same subgraph following the inter-operator dependence to achieve the whole-program schedule and finally implements some hardware-specific optimizations to transform the program to parameterized tensor programs called macro-kernels.

In the "runtime" phase, when the tensor shape becomes available, TSCompiler further schedules the program by loading one of the pre-compiled macro-kernels. To identify the optimal adaptation for the given tensor shape, we introduce an occupancy-targeted cost model. This model efficiently associates the subgraph of a specific shape with various macro-kernel configurations and GPU hardware resources. Moreover, this program loading scheme is lightweight and bypasses the time-consuming just-in-time compilation.

We practice TSCompiler on three representative dynamic-shape DNN models and compare TSCompiler with state-of-the-art frameworks on two GPU platforms, NVIDIA RTX3090 and A100. Experiments show that TSCompiler achieves up to 3.97× speedup in terms of kernel efficiency and five orders of improvements on end-to-end latency. This paper makes the following contributions:

- We identify that the core challenge of dynamic shape compilation is the lack of shape information at compile time and significantly varying ranges of data sizes at runtime.
- We introduce a two-phase process that combines compile-time and runtime co-scheduling for dynamic-shape model execution optimization. At compile time, we implement shape propagation to recover essential shape information and utilize operator fusion to partition the computation graph into various fused subgraphs. Then we conduct a hardware-aligned schedule tuning to find high-performance schedules for single operators and propagate the schedule across the whole subgraph following dependence analysis. During runtime, we conduct a lightweight occupancy-targeted cost model to select programs from pre-compiled repositories to serve tensor computation.
- We implement TSCompiler, a dynamic-shape deep learning compiler based on the proposed ideas. Extensive experiments show that TSCompiler outperforms state-of-the-art DNN frameworks and compilers significantly in terms of both kernel efficiency and end-to-end latency.

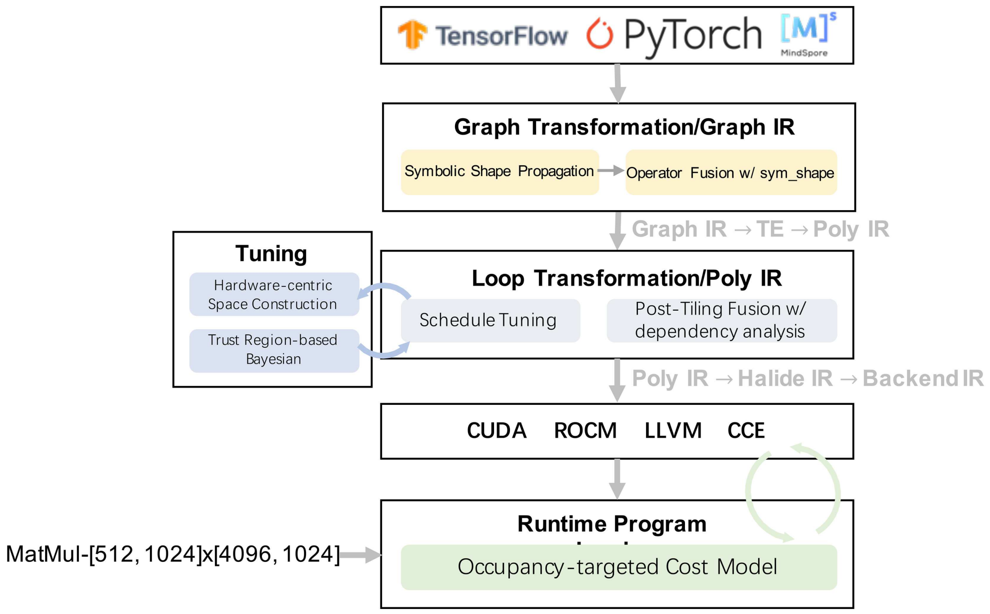
TensorFlow
PyTorch
MindSpore
Graph Transformation/Graph IR
Symbolic Shape Propagation
Operator Fusion w/ sym_shape
Graph IR → TE → Poly IR
Tuning
Hardware-centric Space Construction
Trust Region-based Bayesian
Loop Transformation/Poly IR
Schedule Tuning
Post-Tiling Fusion w/ dependency analysis
Poly IR → Halide IR → Backend IR
CUDA
ROCM
LLVM
CCE
Runtime Program
MatMul-[512, 1024]x[4096, 1024]
Occupancy-targeted Cost Model

Figure 1 Architecture of TSCompiler.

2 Background

Deep learning compilers play an important role in the automated generation of high-performance programs for the deployment of DNN models. Existing deep learning compilers like TVM [2] leverage both graph- and operator-level transformations to optimize the model execution. For the computation graph, compilers utilize a series of graph rewriting such as operator fusion and layout transformation to partition the whole graph into a set of fused subgraphs. These subgraphs are further lowered into a loop-level representation, with each operator represented as a primitive computation enclosed by nested multi-level loops. Compilers then apply fine-grained loop transformations like tiling, reordering and vectorization to optimize the performance of programs and finally generate native codes for different hardware backends.

Due to the intricate interactions between the tensor computation and the hardware architecture, compilers often utilize machine learning (ML)-based auto-schedulers [3,4] to search for high-performance loop schedules. This process involves constructing a search space based on the shape values of fused subgraphs and then exploring the search space navigated by a cost model. Since most DNN models' computation graphs and shape specifications are available at compile time, compilers can apply auto-tuning offline to generate a specific high-performance operator schedule to serve following tensor computation.

However, as models are evolving to accommodate dynamic shape input tensors, especially for natural language processing (NLP) and multi-modality Transformer-based models [9–11], shape values keep unknown until runtime. While there have been efforts [6, 8, 12] to handle dynamic shape problems in deep learning compilers, many of them are limited to single operator granularity, which fails to incorporate essential graph-level optimizations (e.g., operator fusion) due to the loss of shape information, leading to greater pressures on memory access.

3 Overview

Figure 1 depicts the architecture of TSCompiler. TSCompiler takes as input a static model from the existing framework and transforms it into a unified graph-level intermediate representation (IR). The graph engine of TSCompiler then performs symbolic shape propagation to convert the model from a static representation into a symbolic one and partitions it to multiple fused subgraphs specified by parametric tensor expression [2].

The tensor expression is then lowered to a polyhedral IR, schedule tree [13], to facilitate automatic fusion and tiling optimizations. To moderate the scalability issue of the downstream polyhedral analysis [14], we implement operator inlining to reduce the number of program statements before lowering.

On top of the polyhedral schedule tree, we manipulate the program with the pattern-based schedule matrix [15] to find a tuning-friendly representation that separates backbone operators (e.g., Matmul, Conv, and Reduce) from others. We then employ a loop-based code abstraction to establish a comprehensive hardware-aligned schedule space for the backbone operator and navigate the design space exploration by Bayesian optimization [16]. The selected high-performance backbone schedules are propagated to other operators within the fused graph to get the whole-program schedule through the dependence analysis [17].

The optimized polyhedral IR is next lowered to Halide IR to apply hardware-specific transformations that are difficult to model by polyhedral compilation, including tensorization [18] and asynchronous double buffering. Finally the Halide IR is lowered to CUDA code to generate a pool of high-performance parameterized tensor programs for explored backbone schedules.

During runtime, we select the most efficient candidate to launch from the pre-compiled programs established in the compilation phase. Given the shape dimensions unknown until runtime, we introduce a lightweight cost model to devise computation schedules based on the available GPU resources. With the scheduling in place, our approach promptly initiates the corresponding pre-compiled programs, bypassing the need for time-consuming JIT compilation.

4 Graph transformation

Tensor shapes play an important role in various optimization passes including operator fusion and loop tiling [3, 4]. Since the tensor shape is unknown at compile time, existing deep learning compilers suffer a lot from dynamic-shape models due to the fact that previous compilation passes designed for static-shape models cannot be reused, leading to increasing deployment complexity. Fortunately, we observe that we can introduce symbolic shape values and infer dynamic tensor shapes at compile time according to operator semantics to propagate the symbolic representation across the computation graph. These symbolic shapes can then be used to develop subsequent graph- and operator-level optimizations for tensor programs.

We thus first resort to shape propagation to convert the tensor shapes from static form to symbolic form and then utilize the operator fusion to partition the whole computation graph to fused subgraphs to reduce the runtime overhead of kernel launch and off-chip memory access.

4.1 Symbolic shape propagation

In the context of dynamic-shape models, exact tensor shapes are only determined at runtime. We introduce symbolic shape variables shared across different tensors to construct symbolic shape representation, termed sym_shape. This representation is integrated into both graph-level and operator-level IR, enabling the transformation of the computation graph into a symbolic form through propagation. For example, we can represent ViT's input tensor's shape as [N, H, W, 3], where N, H, W are global shape variables and can be instantiated with different values at runtime.

Before shape propagation, we first implement a compound operator splitting pass to decompose the compound operators to primitive operators to significantly reduce the number of operator types in the subsequent compilation pipeline [19], thus reducing the engineering efforts for both graph- and operator-level optimization as well as enabling more optimization opportunities across the original operator boundary. We then develop a dataflow analysis framework to infer shape and size information throughout the whole computation graph.

Algorithm 1 shows our detailed worklist-based shape propagation algorithm. Its goal is to propagate symbolic representation along dataflow according to recognized operator types. TSCompiler initializes propagation by assigning all parameter tensors within the computation graph to static type and then transforms the shape values of the whole graph's input tensors to symbolic form for different model types (e.g., [B, L] for common natural language processing models and [N, 3, H, W] for computer vision models), as shown in lines 2–10 of Algorithm 1. TSCompiler then scans all the input tensors' consumer nodes to check whether they are ready to propagate shape representation from inputs to outputs and adds eligible nodes to the worklist for subsequent processing. All the input tensors of eligible nodes should be either symbolic form or static type, thus can be used for shape computation.

TSCompiler takes eligible nodes from the worklist to compute output tensor shapes according to operator types and input tensor shapes. Next, TSCompiler scans the current node's successors to try to broadcast shape values to sibling nodes and adds eligible sibling and successive nodes to the worklist.

Algorithm 1 Worklist-based shape propagation algorithm

```
Require: $g$ : Graph, $e$ : Config;
 1: Worklist = [];
 2: for tensor ∈ $g$.params() do
 3:     tensor.is_static = true;
 4: end for
 5: for tensor ∈ $g$.inputs() do
 6:     Init(tensor, $g$);
 7:     for node ∈ tensor.nodes() do
 8:         eligible = CheckNodeStatus(node);
 9:         if eligible ∧ node.is_processed ≠ true then
10:             Worklist = Worklist + node;
11:         end if
12:     end for
13: end for
14: while Worklist is not empty do
15:     node = get first node out of Worklist;
16:     node.is_processed = true;
17:     ComputeSymShape(node);
18:     for successor ∈ GetSuccessors(node) do
19:         for sibling ∈ GetPredecessors(successor) do
20:             InferSibSymShape(successor, node, sibling);
21:             eligible = CheckNodeStatus(sibling);
22:             if eligible ∧ sibling.is_processed ≠ true then
23:                 Worklist = Worklist + sibling;
24:             end if
25:         end for
26:         eligible = CheckNodeStatus(successor);
27:         if eligible ∧ successor.is_processed ≠ true then
28:             Worklist = Worklist + successor;
29:         end if
30:     end for
31: end while
32: Return $g$.
```

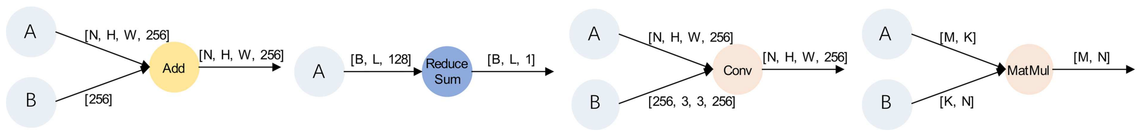

Figure 2 (Color online) Shape computation for common operator types.

TSCompiler repeats this process until running out of nodes within the worklist, as shown in lines 14–31 of Algorithm 1. Figure 2 takes some common operator types as examples to illustrate shape computation according to operator semantics. Element-wise operators' input tensor shapes should broadcast their values along each shape dimension to the output tensor. The operators' output tensor's reduction dimension should be reduced to 1 compared to input tensors. Convolution operators' output tensor shapes depend on input tensor shapes, filter maps, padding size, and stride. MatMul's input tensor shapes should agree with the reduction dimensions which can be used to infer sibling node's tensor shapes and check shape validity during propagation.

4.2 Operator fusion for dynamic operators

With the help of symbolic shape information across the computation graph, TSCompiler builds a prioritized fusion pipeline to partition the whole graph into fused subgraphs for subsequent operator-level scheduling. TSCompiler leverages a multi-pass scanning across the graph to enable a holistic fusion to reduce the runtime overhead of kernel launch and off-chip memory access. Meanwhile, the multi-pass manner is flexible to adjust the phase order of the fusion pipeline as well as add customized passes for graph rewriting or other fusion templates [20, 21].

Figure 3 describes the fusion pipeline of the TSCompiler. TSCompiler utilizes a backbone-based operator fusion that scans the whole graph multiple times to find operators with different priorities and then iteratively fuses adjacent producer or consumer nodes once they agree with the intermediate symbolic tensor shapes. The fusion stops when meeting another backbone-based fusion group or violating some predefined rules. TSCompiler incorporates the operator fusion along with symbolic shape propagation into a state-of-the-art operator fusion framework for static-shape models (Apollo [19]), enabling the

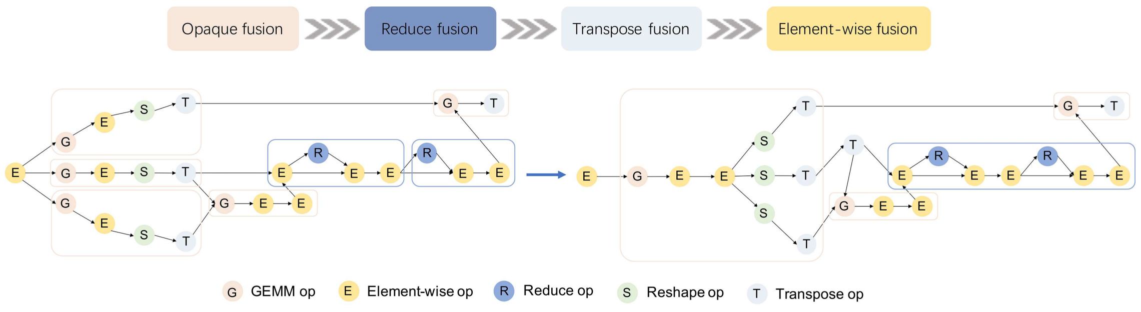

Figure 3 Backbone-based prioritized fusion pipeline.

generation of symbolic-shape subgraphs for subsequent operator-level optimizations.

Taking the example in Figure 3, TSCompiler starts with general matrix multiply (GEMM)-level fusion that fuses element-wise, reshapes and transposes operators to reduce the off-chip memory access, and then re-scans the graph with reduce-level fusion and other fusion with lower priorities. Finally, there are three cast (element-wise) operators that are not included in GEMM's fusion groups since we enable TensorCore code generation where FP16 data in GEMM's fusion group can bring more efficient memory promotion. Force to fuse GEMM's fusion group with cast operators may result in insufficient use of global memory bandwidth. Thus we leave these cast operators either alone or fused into Reduce's fusion group. After the common fusion passes, we enable further fusion by graph rewriting that rewrites three horizontal GEMM's fusion groups into a new fusion group to further squeeze thread-level parallelism [22]. Meanwhile, we also support stitch fusion as well to form multi-Reduce fusion group [21, 23].

5 Operator scheduling optimization

In this section, we describe how TSCompiler generates high-performance tensor programs for dynamic-shape fused subgraphs. As tensor shapes remain unknown during compile time and vary across a large range at runtime, we design a hardware-aware code generation pipeline for fused subgraphs. TSCompiler builds the hardware-aligned search space for backbone operators based on hardware characteristics and explores the space guided by Bayesian optimization to generate a set of promising candidates. To facilitate the shape-generic tuning process, TSCompiler utilizes a pattern-based schedule matrix to guide the polyhedral scheduler to compute an initial schedule flexible for subsequent exploration of the backbone's transformation [24]. After the tuning process, TSCompiler propagates the backbone's schedule to other fusion groups to generate the whole program schedule according to dependence analysis. And finally TSCompiler applies some hardware-specific transformations to fully make use of new acceleration primitives.

5.1 Bayesian-based schedule tuner

Loop tiling plays an important role in squeezing the parallelism and locality of the program and even benefits from the acceleration primitives for multi-dimensional tensor computations. Due to the large range of possible runtime tensor shapes, TSCompiler proposes a hardware-aligned loop-based modeling to analyze the tensor programs of backbone operators and leverages a trust region-based Bayesian optimization to explore high-performance tile size configurations efficiently.

Loop-based modeling. As tensor shapes remain unknown during compile time, we introduce a versatile loop-based, bottom-up code analysis approach to extract essential kernel attributes from backbone operators. Additionally, we align loop schedules with various backend characteristics and incorporate hardware-dependent constraints such as shared memory capacity and memory hierarchies when constructing the search space. Given that deep learning operators primarily involve multi-dimensional tensor computations, this methodology offers universal applicability, extending to nearly all operators, including primitive operators and auto-generated or fused ones. Without loss of generality, we use a GPU-oriented GEMM schdule as an example to illustrate our idea for clarity (convolution can also be transformed to GEMM-based computations through implicit GEMM algorithm [25] by TSCompiler).

The computation of GEMM is basically a three-dimension nested loop. To efficiently leverage the GPU's computing power, the whole computation should be split into hierarchical tiles. As shown in

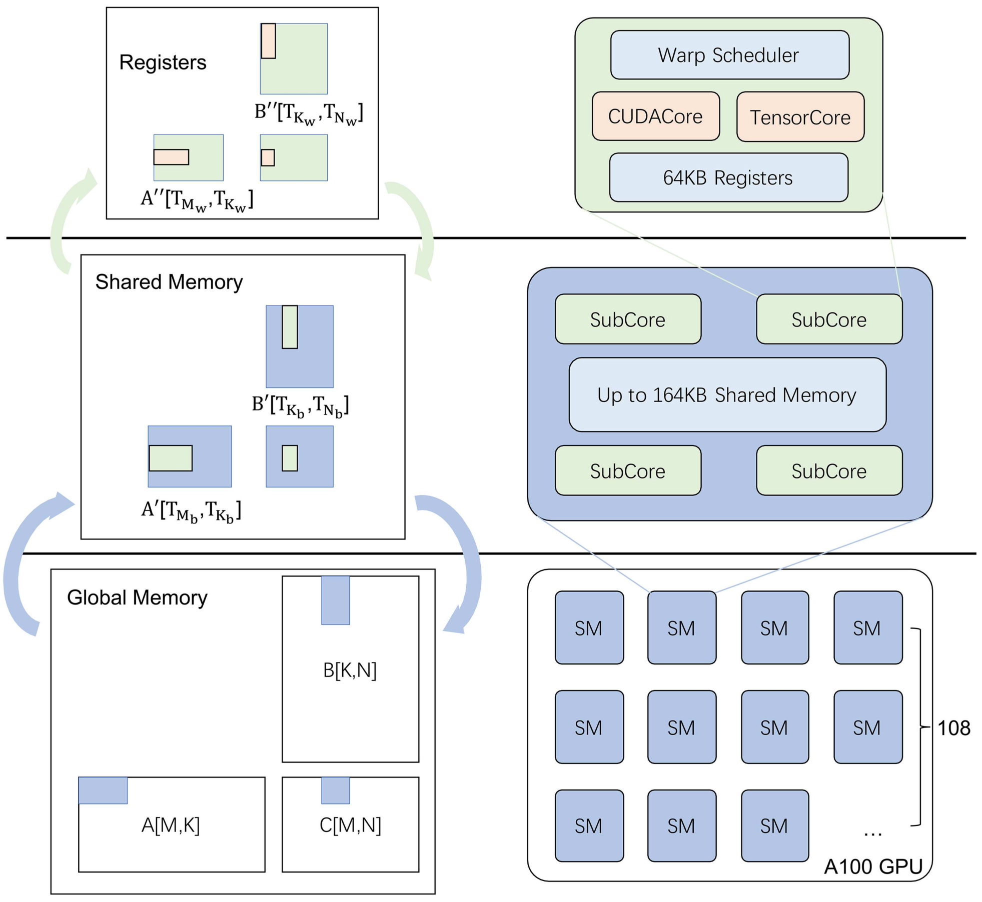

Figure 4 Tiled GEMM on A100 GPU.

Figure 4, the original three-dimension nested loops are first decomposed into multiple independent $T_{M_b} \times T_{N_b}$ matrix products with the outermost two loops bind to thread blocks. Each thread block iterates on the reduction dimension at the stride of T_{K_b}, loads $T_{M_b} \times T_{K_b}$ and $T_{K_b} \times T_{N_b}$ tiles from global memory to shared memory, performs $T_{M_b} \times T_{N_b} \times T_{K_b}$ matrix products, and accumulates the results. The $T_{M_b} \times T_{N_b} \times T_{K_b}$ thread block subprograms are further decomposed into smaller independent $T_{M_w} \times T_{N_w}$ matrix products and distributed across warps. Each warp iterates on T_{K_b} and loads $T_{M_w} \times T_{K_w}$ and $T_{K_w} \times T_{N_w}$ tiles from shared memory to registers to perform accumulated matrix products. The $T_{M_w} \times T_{N_w} \times T_{K_w}$ warp tiles are next mapped onto different hardware instructions to perform $T_{M_i} \times T_{N_i} \times T_{K_i}$ matrix products, e.g., $1 \times 1 \times 4$ for INT8 dp4a and $16 \times 8 \times 16$ for FP16 mma on A100. After finishing the accumulation, the final results are written back to the lower memory hierarchy. Following this hierarchical schedule, a legal design of GPU-oriented GEMM schedule can thus be defined by the tiling tuples $\langle T_{M_b}, T_{N_b}, T_{K_b} \rangle$, $\langle T_{M_w}, T_{N_w}, T_{K_w} \rangle$, and $\langle T_{M_i}, T_{N_i}, T_{K_i} \rangle$.

GPU architecture also exhibits a hierarchical organization. Taking the current mainstream A100 GPU as an example, it consists of 108 streaming multiprocessors (SM), with each SM partitioned into four sub-cores. Each sub-core includes one warp scheduler, enabling concurrent execution of 32 threads in an SIMT fashion, or a blocked matrix multiplication via dedicated TensorCore hardware. Those threadblocks and warps are executed and scheduled concurrently to maximize the underlying resource usage.

Leveraging the loop-based analysis scheme and considering GPU architecture characteristics, we introduce two categories of constraints on tile and computation schedules, as depicted in Figure 5.

- Hard constraints define a set of constraints that programs must obey or otherwise they may encounter failure or significant performance degradation. (1) Shared memory limit: As each thread block is assigned to an SM, its shared memory usage must not surpass the SM's capacity. (2) Register capacity: NVIDIA GPUs impose a limit on the number of registers available to each thread, typically around 255 registers per thread. Exceeding this threshold leads to register spilling and decreased performance. Despite not surpassing resource limits, the increase in resource utilization can also lead to potentially diminished

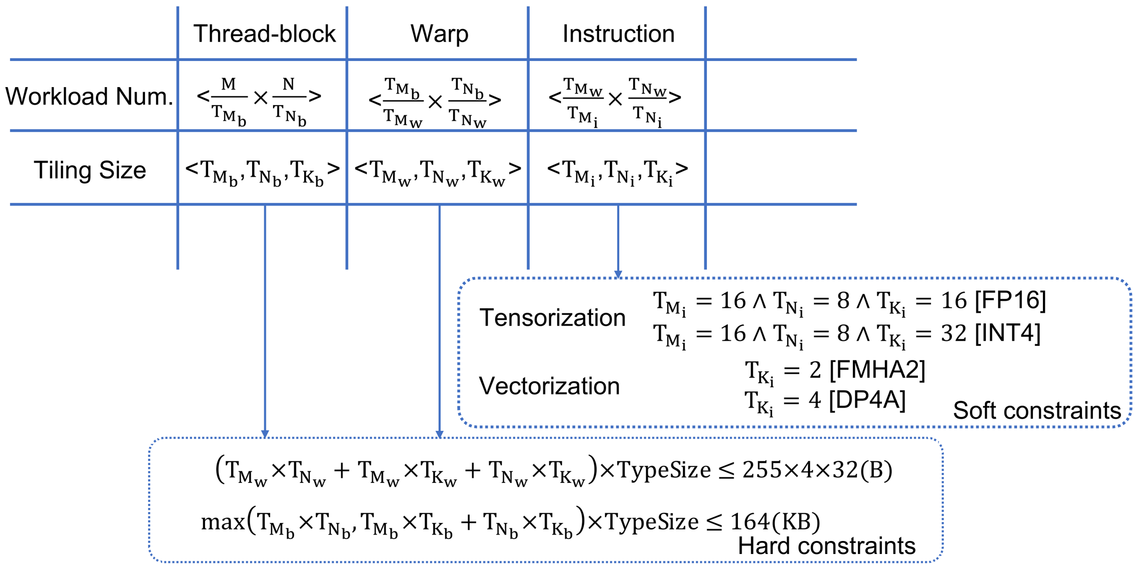
Thread-block
Warp
Instruction
Workload Num.
$<\frac{M}{T_{M_b}}\times\frac{N}{T_{N_b}}>$
$<\frac{T_{M_b}}{T_{M_w}}\times\frac{T_{N_b}}{T_{N_w}}>$
$<\frac{T_{M_w}}{T_{M_i}}\times\frac{T_{N_w}}{T_{N_i}}>$
Tiling Size
$<T_{M_b},T_{N_b},T_{K_b}>$
$<T_{M_w},T_{N_w},T_{K_w}>$
$<T_{M_i},T_{N_i},T_{K_i}>$
Tensorization
$T_{M_i}=16\wedge T_{N_i}=8\wedge T_{K_i}=16$ [FP16]
$T_{M_i}=16\wedge T_{N_i}=8\wedge T_{K_i}=32$ [INT4]
Vectorization
$T_{K_i}=2$ [FMHA2]
$T_{K_i}=4$ [DP4A]
Soft constraints
$(T_{M_w}\times T_{N_w}+T_{M_w}\times T_{K_w}+T_{N_w}\times T_{K_w})\times \text{TypeSize}\leq 255\times 4\times 32(\text{B})$
$\max(T_{M_b}\times T_{N_b}, T_{M_b}\times T_{K_b}+T_{N_b}\times T_{K_b})\times \text{TypeSize}\leq 164(\text{KB})$
Hard constraints

Figure 5 (Color online) Design space definition and constraints.

parallelism at runtime.

- Soft constraints pertain to hardware instructions designed to accelerate computations with specific functions, requiring predefined program patterns such as memory alignment or specific tiling sizes to achieve optimal throughput. (1) Vectorization: from Pascal generation, NVIDIA GPUs begin to support vectorized instructions, e.g., hfma2 for $1\times1\times2$ FP16 dot products and dp4a for $1\times1\times4$ INT8 dot products. (2) Tensorization: from Volta generation, NVIDIA GPUs begin to support tensorized instructions for GEMM computations with different data types and tensor shapes, e.g., mma.m16n8k16 for FP16 and mma.m16n8k32 for INT4.

Given a specific loop schedule and a tiling tuple $\langle T_{M_b}, T_{N_b}, T_{K_b}\rangle$, $\langle T_{M_w}, T_{N_w}, T_{K_w}\rangle$, $\langle T_{M_i}, T_{N_i}, T_{K_i}\rangle$, we are able to describe a tiled GEMM program optimized for GPU hardware. All compositions of tile sizes and variable length constitute a search space for the dynamic-shape kernel optimization, where the goal is to find a set of tiling tuples that align with hardware architecture and yield superior performance across variable shapes.

Bayesian-based design space exploration. With the loop-based abstraction provided above, we introduce a Bayesian optimization framework to explore the design space in pursuit of optimal kernel implementations. Given the variance across distinct hardware instructions, our strategy adopts a hierarchical selection process. After ensuring alignment with specific data types and computational semantics, our approach first favors tensorized instructions, then vectorized, and ultimately CUDA core instructions. Once the instruction mapping is determined, the focus of design space exploration shifts to thread-block tiling ($\langle T_{M_b}, T_{N_b}, T_{K_b}\rangle$) and warp-level tiling ($\langle T_{M_w}, T_{N_w}, T_{K_w}\rangle$).

Although the search space for a shape-specific workload exhibits non-convexity and non-smoothness [26], we discover and exploit the local smoothness of dynamic-shape workloads. Figure 6 shows the performance distribution of the dynamic-shape GEMM operators from the BERT-large model which evaluates $Y = XW^{\mathrm{T}}, X : [32\times\mathrm{Seq}, 1024], W : [3072, 1024]$. Here we sample 10 sequence length uniformly from the range of [1, 512] and utilize grid search to benchmark tensor programs scheduled with different tiling tuples and collect each explored kernel's variable length, tiling tuple, and computation throughput. It can be seen that there exist some tiling tuples exhibiting high performance across various shape configurations as marked with black boxes. This local smoothness is consistent with the loop-based modeling, since each tiling tuple specifies how a thread block within a kernel should be computed and determines whether the computation within a thread block saturates the computation units. Moreover, for a specific tiling tuple, varied shape configurations only lead to different numbers of thread blocks at runtime, and the total latency can thus be estimated by the number of waves (batch of concurrent thread blocks) multiplied with the latency of a single wave as described in Section 6.

The locally smooth nature of the performance distribution of dynamic-shape workloads motivates a trust region-based Bayesian optimization [16]. By maintaining multiple local Gaussian processing (GP)-

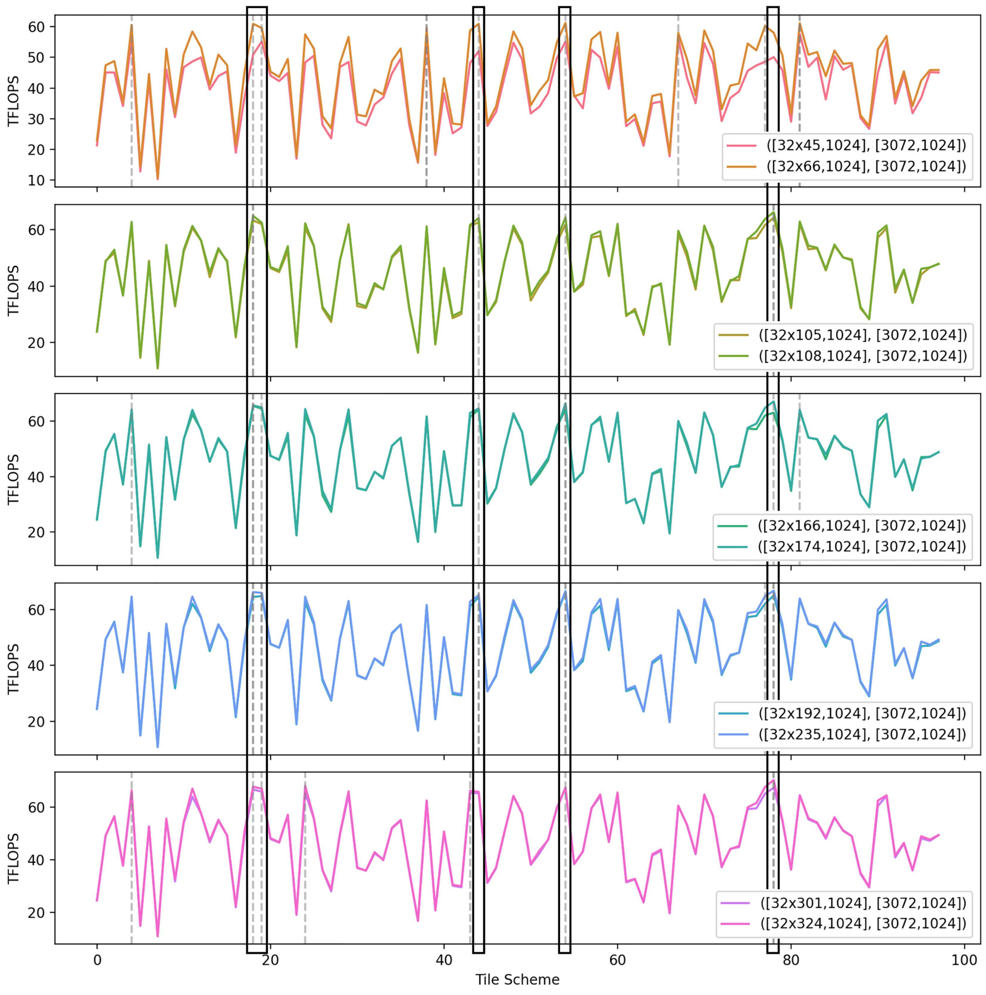

Figure 6 Local smoothness.

based surrogate models simultaneously, we iteratively allocate samples from sets of local trust regions to evaluate and update the corresponding local optimization runs. Here, the trust region is a space that is small enough for surrogate models to predict samples' performance accurately within the space. At each round, we will dynamically expand the size of the local space if we find samples improve upon current optima; otherwise, we will shrink the space after several consecutive failures. To start the Bayesian optimization, we first select some hardware-aligned candidates and set the surrounding hyperrectangle (base length L_{b}) as initial trust regions. In each iteration, we then sample a batch of programs from the union of all trust regions balanced by a multi-armed bandit that regards each trust region (TR) as a lever and update the corresponding local surrogate models based on on-device measurements.

Hardware-aligned exploration as initial populations. To initialize each local optimization run, we take inspiration from Roller [5] to search for hardware-aligned tiling tuples for backbone operators as beginning trials that saturate an SM considering compute and memory balance. We then instantiate the dynamic-shape workloads based on explored tiling schedules to generate large static-shape workloads that can saturate the whole GPU to touch the full potential of these explored schedules.

Efficient exploration across multiple local models balanced with multi-armed bandit strategy. In each iteration, we select a batch of candidates across all the local trust regions for on-device measurements where each surrogate model maintains a posterior function $f_i \sim \mathrm{GP}_i^{(t)}$ and $\mathrm{GP}_i^{(t)}$ is the GP posterior for TR_i as iteration t. We then select samples such that it minimizes the function value across all trust regions,

$$x^{(t)} \in \min_i \ \mathop{\arg\min}_{x\in \mathrm{TR}_i} \ f_i, \quad \text{where } f_i \sim \mathrm{GP}_i^{(t)},$$

and update corresponding local models based on on-device observations. We model the sampling allocation process as a multi-armed bandit to balance the exploitation and exploration across local models.

Post-iteration projection to ensure valid configurations and hardware-aware pruning. The sampled configuration s often falls outside the search space. We thus first project each s to the nearest configuration in the search space and then prune off configuration whose resource usages surpass GPU's resource capacity.

We replay the exploration process multiple times to gather a collection of high-performance tile size configurations for backbone operators and record the on-device execution cycles for a single wave as well as the resource usage such as registers and shared memory, which will be utilized in the runtime cost model. Each tile size configuration will be applied to the backbone's fusion group, including both Convolution and Add+Relu operators' corresponding statements to generate a single tiled schedule tree as shown in Figure 7(d).

5.2 Polyhedral scheduling

To facilitate the hardware-centric program tuning process, we resort to polyhedral analysis to compute an initial schedule flexible for backbone-oriented fine-tuning while guaranteeing the transformation validity. Before lowering to polyhedral IR [13], we utilize TVM's compute_inline primitive to compose multiple primitive operators into a compound one to reduce the number of statements when generating both dependence matrix and schedule matrix. On the one hand, the decrease of the number of statements moderates the complexity of polyhedral scheduling. On the other hand, it also helps to classify different subgraphs into some recognized fusion types for code generation and enables subsequent pattern-based polyhedral schedules for each type. For example, we often compose a chain of element-wise activation operations into a compound one; thus various composed activations like {Add, Relu} and {Add, Add, Relu} will introduce little scalability issues to polyhedral analysis.

After the fused subgraph is lowered to polyhedral IR where each program is represented by integer sets and affine maps, we resort to polyhedral schedulers to compute a new schedule flexible for subsequent fine-tuning. Traditional polyhedral strategies that rely on solving integer linear programming (ILP) problems [24, 27–29] face many problems in dynamic-shape scenarios. On the one hand, these strategies are shape-sensitive which introduces lengthy scheduling times for large workloads. On the other hand, these strategies may generate varied schedules for different shape configurations which complicates the downstream compilation pipelines. TSCompiler utilizes pattern-based schedule matrices to compute initial schedules for dynamic-shape workloads [15]. Generating schedules by specifying schedule matrices can directly separate backbone operators from others within the same fused subgraph and avoid solving time-consuming ILP problems, handling the unexpected schedule trees outputted by heuristic-based cost models and even the inability to compute a valid schedule for the given program.

We showcase a common fused type obtained from graph optimization and function inlining in Figure 7(a), with the pseudo code shown in Figure 8(a). It performs a 2D convolution on padded feature maps, followed by two element-wise activations, BiasAdd and Relu. The statement S_4 refers to the compound Add+Relu operators generated by function inlining. The initial schedule tree depicted in Figure 7(b) is built according to the original program semantics.

To build a hardware-aligned search space, we need to expose the backbone operators alone to the schedule tuner as well as make the scheduled representation flexible for subsequent subtree rewriting to implement the post-tiling fusion [29]. Thus we utilize the pattern-based auto-generated schedule matrix [15] to specify the multidimensional schedules for each statement. Taking the fused subgraph as an example, we first assign different statements with different shift constants (0 for padding and 1 for others) for the first dimension of the schedule matrix to generate two fusion groups as shown in Figure 7(c). Meanwhile, we leave the exact shift constants (set to ? in the schedule matrix) for the 6th dimension of the element-wise operator's schedule to be determined by the scheduler while preserving each dependence. As shown in Figure 8(b), the statement S_4 is sinked to the innermost of the backbone's fusion group due to the calculated shift constant 255 in which element-wise operator can share backbone's tuned schedule as well. This schedule matrix is auto-generated based on the analysis of the mapping types of the operators within the fused subgraph and applicable to all subgraph instances of {Pad, Conv, Element-wise$^+$} no matter the number of element-wise operators as well as the feature map/filter size. We also explicitly mark the subtree of Pad and Add+Relu as prelogue and epilogue respectively after scheduling to help the schedule tuner in Subsection 5.1 skip them when constructing and exploring the search space. The matrix-based loop schedule guides the polyhedral scheduler to generate a predictable

```
A = placeholder((N, H, W, 256))
B = placeholder((256, 3, 3, 256))
C = placeholder((256))
rc = reduce_axis((0, 256))
rh = reduce_axis((0, 3))
rw = reduce_axis((0, 3))
A_pad = nn.pad(A, [0, 1, 1, 0], [0, 1, 1, 0])
D = compute((N, H, W, 256), lambda n, h, w, o :
          sum(A_pad[n, (h+rh), (w+rw), rc] * B[o, rh, rw, rc],
          axis=[rc, rh, rw]))
E = compute((N, H, W, 256), lambda n, h, w, o : D[n, h, w, o] + C[o])
F = compute((W, H, W, 256), lambda n, h, w, o : max(E[n, h, w, o], 0))
```

(a)

```
Domain
  Sequence
    Filter{S1(n,h,w,rc)}
      Band{S1->(n,h,w,rc)}
    Filter{S2(n,h,w,o);S3(n,h,w,o,rc,rh,rw)}
      Band{S2->(n,h,w,o);S3->(n,h,w,o)}
        Sequence
          Filter{S2(n,h,w,o)}
          Filter{S3(n,h,w,o,rc,rh,rw)}
            Band{S3->(rc,rh,rw)}
    Filter{S4(n,h,w,o)}
      Band{S4->(n,h,w,o)}
```

(b)

```
Domain
  Sequence
    Filter{S1(n,h,w,rc)}
      Mark{"prelogue"}
      Band{S1->(n,h,w,rc)}
    Filter{S2(n,h,w,o);S3(n,h,w,o,rc,rh,rw);S4(n,h,w,o)}
      Band{S2->(n,h,w,o);S3->(n,h,w,o);S4->(n,h,w,o)}
        Band{S2->(0,0,0);S3->(rc,rh,rw);S4->(255,2,2)}
          Sequence
            Filter{S2(n,h,w,o)}
            Filter{S3(n,h,w,o,rc,rh,rw)}
            Filter{S4(n,h,w,o)}
              Mark{"epilogue"}
```

(c)

```
Domain
  Sequence
    Filter{S1(n,h,w,rc)}
      Mark{"prelogue"}
      Band{S1->(n,h,w,rc)}
    Filter{S2(n,h,w,o);S3(n,h,w,o,rc,rh,rw);S4(n,h,w,o)}
      Mark{"block_mapping"}
      Band{S2->(n/32,h,w,o/128);S3->(n/32,h,w,o/128);S4->(n/32,h,w,o/128)}
        Mark{"C[shared->global]"}
        Band{S2->(0,0,0);S3->(rc/64,rh,rw);S4->(3,2,2)}
          Mark{"A/B[global->shared]"}
            Band{S2->(n,o,0);S3->(n,o,rc);S4->(n,o,63)}
              Sequence
                Filter{S2(n,h,w,o)}
                Filter{S3(n,h,w,o,rc,rh,rw)}
                Filter{S4(n,h,w,o)}
                  Mark{"epilogue"}
```

(d)

```
Domain
  Sequence
    Filter{S1(n,h,w,rc)}
      Mark{"prelogue"}
      Band{S1->(n,h,w,rc)}
    Filter{S2(n,h,w,o);S3(n,h,w,o,rc,rh,rw);S4(n,h,w,o)}
      Mark{"block_mapping"}
      Band{S2->(n/32,h,w,o/128);S3->(n/32,h,w,o/128);S4->(n/32,h,w,o/128)}
        Mark{"C[shared->global]"}
        Sequence
          Filter{S2(n,h,w,o);S3(n,h,w,o,rc,rh,rw)}
            Band{S2->(0,0,0);S3->(rc/64,rh,rw)}
              Mark{"A/B[global->shared]"}
                Band{S2->(n,o,0);S3->(n,o,rc)}
                  Sequence
                    Filter{S2(n,h,w,o)}
                    Filter{S3(n,h,w,o,rc,rh,rw)}
          Filter{S4(n,h,w,o)}
            Band{S4->(n,o)}
              Mark{"epilogue"}
```

(e)

Figure 7 (Color online) Running example and its schedule tree representation. (a) The tensor expression of a fused conv operator; (b) the initial schedule tree; (c) the schedule tree after polyhedral scheduling; (d) the schedule tree after fine-tuning tiled with a hardware-aligned configuration; (e) the schedule tree after rewriting.

```
for n in range(N), h in range(H+2), w in range(W+2), rc in range(256):
  A_pad[n][h][w][rc] = A[n][h-1][w-1][rc] if h>0 and h<H+1 and w>0 and w<W+1 else 0   # S1
for n in range(N), h in range(H), w in range(W), o in range(256):
  D[n][h][w][o] = 0                                                                  # S2
  for rc in range(256), rh in range(3), rw in range(3):
    D[n][h][w][o] += A_pad[n][h+rh][w+rw][rc] * B[o][rh][rw][rc]                     # S3
for n in range(N), h in range(H), w in range(W), o in range(256):
  E[n][h][w][o] = max(D[n][h][w][o] + C[o], 0)                                       # S4
```

(a)

```
for n in range(N), h in range(H+2), w in range(W+2), rc in range(256):
  A_pad[n][h][w][rc] = A[n][h-1][w-1][rc] if h>0 and h<H+1 and w>0 and w<W+1 else 0   # S1
for n in range(N), h in range(H), w in range(W), o in range(256):
  for rc in range(256), rh in range(3), rw in range(3):
    if rc == 0 and rh == 0 and rw == 0:
      D[n][h][w][o] = 0                                                              # S2
    D[n][h][w][o] += A_pad[n][h+rh][w+rw][rc] * B[o][rh][rw][rc]                     # S3
    if rc == 255 and rh == 2 and rw == 2:
      E[n][h][w][o] = max(D[n][h][w][o] + C[o], 0)                                   # S4
```

(b)

```
for n in range(N), h in range(H+2), w in range(W+2), rc in range(256):
  A_pad[n][h][w][rc] = A[n][h-1][w-1][rc] if h>0 and h<H+1 and w>0 and w<W+1 else 0   # S1
for n0 in range(N/32), h in range(H), w in range(W), o0 in range(2):
  for rc0 in range(4), rh in range(3), rw in range(3), n1 in range(32):
    for o1 in range(128), rc1 in range(64):
      if rc0 == 0 and rc1 == 0 and rh == 0 and rw == 0:
        D[n][h][w][o] = 0                                                            # S2
      D[n][h][w][o] += A_pad[n][h+rh][w+rw][rc] * B[o][rh][rw][rc]                   # S3
  for n1 in range(32), o1 in range(128):
    E[n][h][w][o] = max(D[n][h][w][o] + C[o], 0)                                     # S4
```

(c)

Figure 8 (Color online) Pseudo codes of the running example. (a) Initial program; (b) schedule before fine-tuning; (c) schedule after tiling and rewriting.

schedule tree for subsequent passes even when compiling convolution with irregular filter size such as 1 × 7 from Inception [30], which relieves the engineering efforts for tuning intricate isl scheduling options [31].

$$\Theta^{\mathrm{S}_1} = \begin{pmatrix} 0&0&0&0&0 \\ 1&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&1&0 \end{pmatrix} \cdot \begin{pmatrix} n \\ h \\ w \\ rc \\ 1 \end{pmatrix} = \begin{pmatrix} 0 \\ n \\ h \\ w \\ rc \end{pmatrix}, \quad \Theta^{\mathrm{S}_2} = \begin{pmatrix} 0&0&0&0&1 \\ 1&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&1&0 \end{pmatrix} \cdot \begin{pmatrix} n \\ h \\ w \\ o \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ n \\ h \\ w \\ o \end{pmatrix},$$

$$\Theta^{\mathrm{S}_3} = \begin{pmatrix} 0&0&0&0&0&0&0&1 \\ 1&0&0&0&0&0&0&0 \\ 0&1&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0 \\ 0&0&0&0&1&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&1&0 \end{pmatrix} \cdot \begin{pmatrix} n \\ h \\ w \\ o \\ rc \\ rh \\ rw \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ n \\ h \\ w \\ o \\ rc \\ rh \\ rw \end{pmatrix}, \quad \Theta^{\mathrm{S}_4} = \begin{pmatrix} 0&0&0&0&1 \\ 1&0&0&0&0 \\ 0&1&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&1&0 \\ 0&0&0&0&? \end{pmatrix} \cdot \begin{pmatrix} n \\ h \\ w \\ o \\ 1 \end{pmatrix} = \begin{pmatrix} 1 \\ n \\ h \\ w \\ o \\ ? \end{pmatrix}.$$

5.3 Loop fusion

After the hardware-centric fine-tuning, TSCompiler propagates the explored loop schedule from backbone operators to other unscheduled ones in the fused subgraph according to the comprehensive instance-level dependence analysis modeled by polyhedral compilation. We refine the fusion process in accordance with

IR rewriting and transformation (Polyhedral IR to Halide IR) to optimize the memory access and code generation holistically.

Fusion through IR rewriting. While the initial schedule is flexible for fine-tuning, it introduces unnecessary condition check for the sinked statement like statement S_4 in Figure 8(b). We introduce an IR rewriting pass after loop tiling to pull the sinked statements out of the backbone's fusion group (still under the common outermost band node, which will be mapped to blockIdx in GPU) as shown in Figure 8(c) corresponding to the schedule tree of Figure 7(e). Separating the element-wise activation from the backbone operator's fusion group enables different compilation pipelines for different fusion groups. For example, the backbone's computation will be later mapped onto the TensorCore primitives through the tensorization pass and its data will be promoted between shared memory and registers back and forth in accordance with fixed mma data layout. After materializing the immediate tensor in shared memory, the element-wise activation again promotes it back to registers to compute the final result with CUDA cores and directly writes them back to global memory in a vectorized manner. Separating fusion groups through rewriting thus presents a convenient schedule tree for subsequent hardware-specific optimizations.

Fusion through IR transformation. Besides enabling different compilation pipelines for different compute units. TSCompiler also fuses statements to the inserted memory promotion statements to propagate the backbone schedule to the unscheduled statements during IR transformation. When transforming the program representation from Polyhedral IR to Halide IR, isl code generator [13] will generate memory promotion statements for the mark node inserted by automatic storage management pass [29]. Taking Mark{A/B[global $\rightarrow$ shared]} in Figure 7(e) as an example, it indicates the code generator to insert statements that promote input tensors A and B from global memory to shared memory whose offset and tensor shapes can be calculated based on the outer band nodes. As for the input tensor A, which statement S_3 reads from and statement S_1 writes to, there exists a relationship between the memory promotion statement of input tensor A and statement S_1, through which we can apply the schedule on memory promotion statements to statement S_1 to figure out what each thread block should do for padding operation. When generating the memory promotion statements in Halide IR, we can just keep the left hand side of the originally generated memory promotion statement as the destination tensor and combine it with the right hand side of the scheduled statement S_1 as source tensor to construct a new combined statement. This dependence-driven schedule propagation removes the off-chip memory access for fused subgraphs and also enables the backbone-oriented schedule fine-tuning which exactly simplifies the code generation pipelines for fused kernels.

5.4 Hardware-specific transformation

Due to the intricacy of modeling hardware-specific transformations by polyhedral IR, TSCompiler chooses to lower the program to Halide IR to apply these transformations through IR rewriting in a pattern match way. We primarily support two hardware intrinsics for NVIDIA GPU backends: tensorization and asynchronous double buffering.

TSCompiler explicitly restricts the innermost tile size configurations aligned with hardware instructions such as $16 \times 8 \times 16$ for FP16 mma instructions on Ampere TensorCore GPUs during the fine-tuning stage. Thus TSCompiler can directly replace the innermost three-dimensional accumulated dot product statements with an invocation of TensorCore's programming interface such as wmma or mma APIs [32]. Meanwhile, TSCompiler supports to offload the memory promotion operation to a customized ldmatrix-based implementation for Ampere TensorCore GPUs to move swizzled data between shared memory and registers to avoid bank conflicts, which can be resolved when inserting the exact promotion statements between shared memory and registers during IR transformation.

TSCompiler also extends the traditional double buffering optimizations to support asynchronous copy, a newly-introduced data movement instruction. By leveraging memcpy_async API, TSCompiler's generated tensor programs can directly copy data from global memory into shared memory without the involvement of registers, improving both the register pressure and occupancy. Besides mapping the memory promotion statements to exposed API, TSCompiler inserts necessary synchronization instructions such as producer_commit and consumer_wait to the double buffering iteration loop to enable asynchronous data movement pipeline.

6 Runtime program loader

After the compilation phase, we have assembled a repository of pre-compiled tensor programs, aka macro-kernels, along with an extensive database detailing their associated resource usage and on-device measurements. Given the unpredictability of tensor shapes of dynamic-shape operators until runtime, we introduce a streamlined wave-based cost model. This model facilitates the dispatch of dynamic-shape workloads to macro-kernels based on the tensor shapes revealed during runtime. Consequently, we can promptly launch the selected macro-kernel, bypassing the time-consuming JIT compilation process.

As described in Subsection 5.1, GPUs provide a hierarchical parallelism structure that allows kernels to launch multiple thread blocks distributed across all available SMs. However, due to the limited resource capacity of SMs, including shared memory, registers, and warp number constraints, only a limited number of thread blocks can be scheduled for concurrent execution at the same time. The remaining thread blocks thus have to wait until on-device thread blocks finish their jobs and release the occupied resources. This scheduling mechanism motivates the basis for our wave-based cost model,

$$s^* = \mathop{\arg\min}_{s \in S_N} T_s \times \left\lceil \frac{N_{\text{total blocks}}}{N_{\text{blocks per wave}}} \right\rceil,$$

$$\text{where } N_{\text{blocks per wave}} = \min\left(\frac{N_{\text{Shared per SM}}}{N_{\text{Shared per block}}}, \frac{N_{\text{Reg per SM}}}{N_{\text{Reg per block}}}\right) \times N_{\text{SM}}.$$

Given a dynamic-shape workload during runtime, we query each macro-kernel denoted as S_N to identify the one with the shortest execution time. This execution time can be modeled by the processing latency T_s for a single batch/wave of thread blocks profiled offline multiplied with the number of batches calculated with the constraints of hardware capacity. After choosing the macro-kernels, we calculate the launch configurations, e.g., thread blocks, based on the macro-kernel's thread block tiling and runtime tensor shapes. Thanks to GPU's support for symbolic launch configurations, our macro-kernels can calculate processing thread blocks based on tensor shapes directly without JIT compilations, thus significantly accelerating the end-to-end latency.

7 Evaluation

7.1 Experimental setup

We evaluate TSCompiler on two servers equipped with NVIDIA GPUs. The first one has Intel(R) Xeon(R) Gold 6336Y CPUs and 8 NVIDIA GeForce RTX3090 GPUs running on Ubuntu 20.04 with CUDA 11.8. The second GPU server is equipped with Intel(R) Xeon(R) Gold 6248R CPUs and 1 NVIDIA A100-PCIE (40G) GPU running on Ubuntu 20.04 with CUDA 11.8.

DNN workloads. We evaluate TSCompiler on three representative dynamic-shape DNN models with different model types. Table 1 [9–11] characterizes them with a comparison of their model types, tasks, and shape dynamism. The range of potential sequence length is set to [1, 512] for BERT and ALBERT, two widely deployed natural language processing models, while the range of candidate image size is set to [32 $\times$ 32, 384 $\times$ 384] for ViT. We opt for a random sampling strategy that selects 25 different shape configurations for each workload.

Baselines. We compare TSCompiler against several static-/dynamic-shape tensor compilers and DNN frameworks, including BladeDISC (v0.3) [12], Ansor (v0.10) [4], TensorIR [18], and PyTorch (v1.12). For the BERT-large model, we also compare TSCompiler against DietCode [6] which implements auto-schedulers for BERT.

PyTorch leverages TorchScript to construct the computation graph by tracing the runtime dataflow and supports naive operator fusion with the help of nvFuser. Ansor and TensorIR both construct shape-dependent search space and fine-tune the space navigated by XGBoost to find the high-performance schedule in a JIT way. While Ansor mainly targets CUDA cores, TensorIR also supports domain-specific compute units like TensorCores by introducing expert knowledge when constructing the search space. In the evaluations, we set the available tuning trials for the whole graph to 500 times the number of subgraphs for both Ansor and TensorIR. BladeDISC is a dynamic shape compiler built on top of MLIR [33, 34]. It enables operator fusion for dynamic models in accordance with the shape constraints derived from operator semantics. However, BladeDISC only supports memory-intensive operator fusion

Table 1 Benchmark specifications[a)]

Model	Type	Task	Shape
BERT [9]	Transformer	NLP	(B, L)
ALBERT [11]	Transformer	NLP	(B, L)
ViT [10]	CNN+Transformer	Image classification	(N, 3, H, W)

a) Shape: symbolic input tensor shapes.

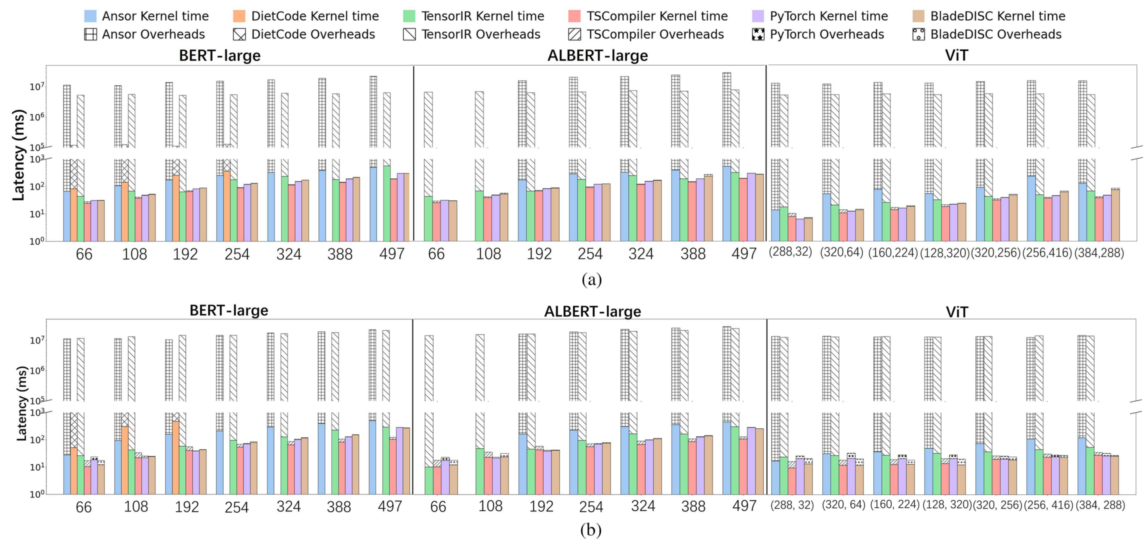

Figure 9 Speedups of kernel efficiency and end-to-end latency. (a) RTX3090; (b) A100.

like Elementwise and Reduce. DietCode extends the search space construction to enable shape-generic fine-tuning based on Ansor. However, DietCode only supports auto-tuning for single operators from BERT.

7.2 End-to-end model execution

Overall results. We first compare the kernel efficiency and end-to-end time (which consists of kernel execution time and other preprocessing overheads, e.g., kernel launch and dispatching) for dynamic model executions on NVIDIA RTX3090 GPUs with 25 randomly sampled shape values. Figure 9 shows the partial results under batch size 32. Note that Ansor fails to produce legitimate programs for some shape configurations of the ALBERT-large model. DietCode also fails to compile and execute some cases of BERT-large model due to out-of-memory. For both vision and language models, TSCompiler is able to improve their kernel efficiency by 1.14×–3.97× as concluded in Table 2 and end-to-end latency by 0.98×–656458× on average as shown in Table 3, which means TSCompiler significantly accelerates the dynamic model executions, especially compared to traditional static compilers. These improvements mainly come from TSCompiler's two-stage scheduling which generates a set of high-performance tensor programs by hardware-aligned schedule fine-tuning and efficient code generation pipelines and then dispatches the runtime requests of specific shape values to pre-compiled executables at runtime, bypassing JIT compilation. TSCompiler's post-tuning fusion also enables code generation for fused subgraphs to reduce the runtime overhead of off-chip memory access and kernel launch while its hardware-specific program transformations combined with hardware-aware search space construction successfully map the tensor computation to hardware primitives like TensorCores to fully squeeze the computing power of modern GPUs.

There exist some hardware-aligned shape configurations like 192 that static compilers like TensorIR deliver higher kernel efficiency. However it often takes minutes if not hours to fine-tune the compilation process for static compilers, which is often longer than the kernel execution times by several orders of magnitude as shown in Figure 9. TSCompiler, however, completely avoids the expensive JIT compilation while still matching the performance of these thoroughly tuned schedules (up to 95%). What's more, for other irregular shape values, thanks to the hardware-aligned search space construction, TSCompiler can still successfully map those compute-intensive operators to hardware primitives, which transforms

Table 2 Kernel efficiency

Model	Backend	Ansor	DietCode	TensorIR	PyTorch	BladeDISC
BERT	RTX3090	2.86×	3.69×	1.74×	1.32×	1.37×
ALBERT		2.77×	–	1.67×	1.27×	1.36×
ViT		3.97×	–	1.74×	1.14×	1.42×
BERT	A100	4.14×	10.30×	2.18×	1.56×	1.59×
ALBERT		4.13×	–	1.85×	1.48×	1.45×
ViT		3.39×	–	2.14×	1.43×	1.02×

Table 3 End-to-end latency

Model	Backend	Ansor	DietCode	TensorIR	PyTorch	BladeDISC
BERT	RTX3090	264935×	3327×	113275×	1.23×	1.29×
ALBERT		216043×	–	136762×	1.20×	1.35×
ViT		656458×	–	268644×	0.98×	1.31×
BERT	A100	282784×	1365×	298072×	1.24×	1.26×
ALBERT		250580×	–	338028×	1.18×	1.20×
ViT		598889×	–	594166×	1.31×	0.99×

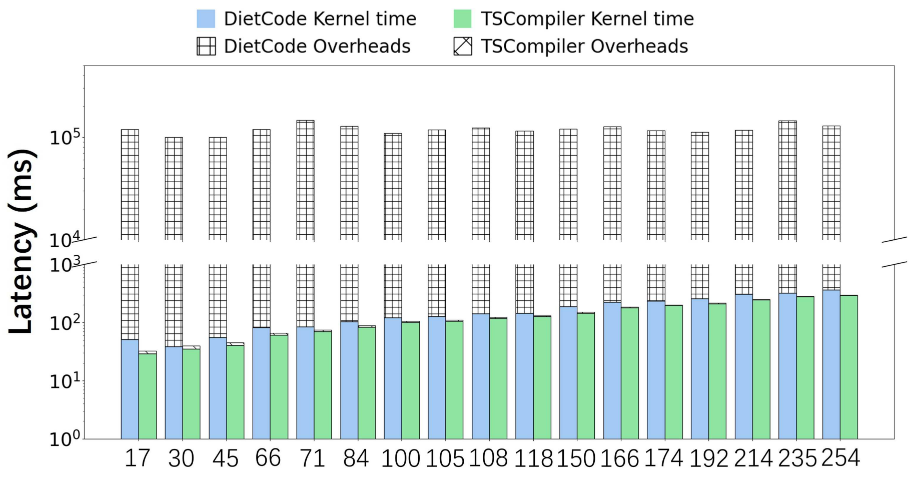

Figure 10 Comparing with DietCode under FP16 w/o TensorCore.

the boundary checker to a padding operator that can be fused to memory promotion statements by the post-tiling-fusion algorithm. The overall pipelines enable TSCompiler to deliver a consistent performance across various shape values and also facilitate the runtime cost model-based kernel dispatching. As for DietCode, although it is specially designed for the BERT model, its generated tensor programs still deliver inferior performance as they can only leverage CUDA cores, leaving high-performance TensorCores idle, as discussed in Subsection 7.4.

In terms of end-to-end latency, as shown in Table 3, both PyTorch and BladeDISC exhibit lower runtime scheduling overheads compared to TSCompiler. An in-depth bottleneck analysis shows that the improvement mainly comes from the asynchronous scheduling for different fused subgraphs. TSCompiler calculates the best schedule for different subgraphs one by one at runtime, while both BladeDISC and PyTorch implement a more powerful runtime that can schedule the computation at CPU- and GPU-side asynchronously. Thus the kernel dispatching process at the CPU-side can be completely parallel with the GPU kernel computation. TSCompiler will consider to extend to co-optimization of both CPU- and GPU-side as future work.

Note that DietCode does not support TensorCore. For a fair comparison, we conduct an additional set of experiments in which we disable TSCompiler's TensorCore feature and evaluate the BERT model in FP16 mode by comparing it with DietCode in Figure 10. Despite the absence of TensorCore support in TSCompiler for these tests, it still achieves a speedup of 1.24× in terms of kernel efficiency and 1264× in terms of end-to-end time, which is primarily attributed to the comprehensive operator fusion enabled by TSCompiler.

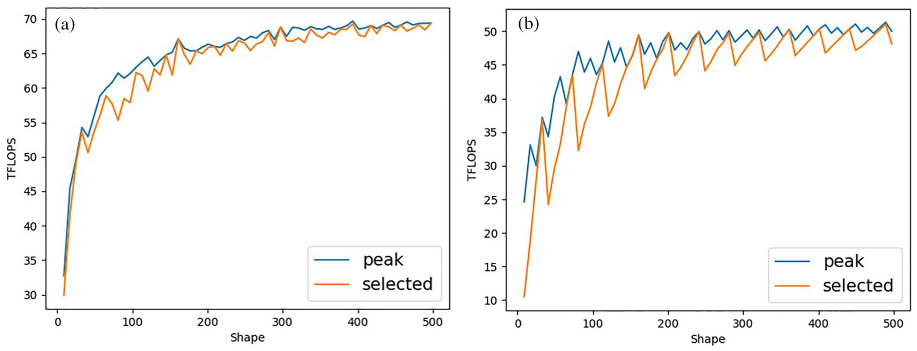

Figure 11 Performance comparison of selected and peak-performance programs. (a) Dense-(32×L, 1024)-(1024, 3072); (b) dense-(32×L, 4096)-(4096, 1024).

We also conduct evaluations on A100-PCIE (40G), a high-end server-class GPU, which features a greater number of SM and larger on-chip buffer sizes. TSCompiler demonstrates substantial performance improvements over existing systems, achieving 1.02×–10.30× better kernel efficiency as detailed in Table 2 and a 0.99×–598889× reduction in end-to-end latency as shown in Table 3. Notably, most of the improvements in kernel efficiency on the A100 surpass those observed on the RTX3090. This enhancement is primarily attributed to the A100's higher computation throughput, which significantly accelerates compute-intensive parts and underscores the importance of optimizing memory access. By employing graph-level optimizations such as symbolic-shape propagation and operator fusion, TSCompiler effectively minimizes the number of executed subgraphs, thereby dramatically reducing memory transactions during runtime.

7.3 Cost model validation

Runtime scheduling determines the final delivered performance for dynamic compilers. In this subsection, we evaluate the prediction quality of TSCompiler's occupancy-targeted cost model using Dense layers extracted from the BERT-large model and randomly pick 62 shape configurations (sequence length) from 1 to 512 as the test set. Figure 11 compares the performance of kernels selected by our cost model with the measured peak performance among pre-compiled tensor programs with an exhaustive search. The occupancy-targeted cost model achieves on average 97.9% and 92.6% of the performance of the best kernel found with exhaustive search, respectively. When the shape values are small, our model produces a relatively inferior throughput due to the insufficient runtime parallelism. Since our occupancy-oriented cost model assumes that the runtime thread blocks can completely saturate the available SMs on GPU, producing waves of thread blocks scheduled onto SMs to accomplish their workloads iteratively. And these waves can be regarded as non-interleaved and even independent with each other. However, as there are too few runtime thread blocks to saturate all the SMs on GPU, especially for large reduction-axis scenarios, the interference between waves or even within waves may dominate the overall runtime overheads, leading to the failure of our cost model to find the exact high-performance programs. There are some recent studies to partition the workloads along the reduction-axis to produce more parallelism, which can both improve the end-to-end latency and the prediction quality of occupancy-targeted cost model when the given parallelism is not enough [35] which TSCompiler can consider to integrate with to further improve its overall performance.

7.4 Search time

While the fine-tuning overheads no longer accumulate to the end-to-end latency for TSCompiler, it still leads to a tedious deployment experience for dynamic models. This subsection thus compares the fine-tuning efficiency of two dynamic compilers, DietCode and TSCompiler, as shown in Table 4. TSCompiler can achieve 1.07×, 1.11×, and 1.16× in terms of search efficiency for subgraphs from BERT-large, respectively, compared to DietCode. This tuning efficiency mainly comes from the hardware-aligned search space construction combined with hardware-specific transformations. While DietCode also constructs a shape-generic search space to find a set of high-performance tensor programs called microkernels, it does not expose the hardware architecture to the construction of the search space. Thus they can only generate tensor programs that fully squeeze the power of CUDA cores, leaving TensorCores idle. However, as the

Table 4 Search time comparison of subgraphs

	DietCode (s)	TSCompiler (s)	Time-saving
Dense	12025	11227	1.07×
BatchMatMul^{NN}	8066	7251	1.11×
BatchMatMul^{NT}	9234	7953	1.16×

accelerators are becoming increasingly powerful, e.g., TensorCores deliver about 10× peak throughput compared to CUDA cores, combined with the fact that models are also becoming increasingly compute-intensive, how to fully leverage the computing power of hardware primitives should be more important to deep learning compilers [36]. Moreover, in the fine-tuning process, it needs hundreds of on-device measurements to help to model the structure of the search space to navigate subsequent explorations. Thus the accumulation of TSCompiler's explored schedule's performance improvements leads to final time-saving even for the fine-tuning process.

8 Related work

Support for dynamic-shape compilation has become an area of active research in recent years. Nimble [8], based on TVM, introduces Any to represent dynamic-shape dimensions and interpret the exact shape values at runtime with the help of a virtual machine. This implementation, however, only works for one-dimension dynamism, mainly targeting dynamic-batching cases. TSCompiler provides symbolic shape propagation to recover shape information at compile time with the help of shape variables and can support multi-dimension dynamism cases. DISC [12] extends MLIR-HLO by propagating the shape information with shape constraints; e.g., input and output tensors have the same size (Transpose, Reshape). However, DISC only supports memory-intensive operator fusions and offloads compute-intensive cases to vendor libraries. TSCompiler supports both memory/compute-intensive cases and optimizes the memory access holistically. Axon [37] proposes a shape language to calculate the symbolic shape with constraint solver, while TSCompiler utilizes symbolic shape propagation to propagate symbolic shape in a dataflow-like way and incorporates many operator-level optimization passes to generate high-performance tensor programs. DietCode [6] builds an auto-scheduler for dynamic-shape workloads based on Ansor and utilizes a cost model to predict runtime performance, however it can only support single-operator scenarios while TSCompiler propagates the backbone operator's tuned schedule to other fusion groups in fused operators with polyhedral analysis. Relax [38] proposes an end-to-end compilation framework for dynamic-shape workloads that integrates first-class dynamic shape annotations to track symbolic shape computation and cross-level representation to enable graph-level, operator-level optimization as well as library calls in a single abstraction. However, Relax still relies on manually composing program transformations to optimize the model execution which requires a deep understanding of both algorithmic and architectural backgrounds, while TSCompiler leverages hardware-centric schedule fine-tuning to search for high-performance schedules automatically.

9 Conclusion

In this paper, we design and implement a two-stage dynamic shape compilation framework named TSCompiler to tackle the challenge of the lack of shape information at compile time and nearly infinite potential shape values at runtime. During the offline phase, we introduce shape propagation to recover essential shape information for subsequent optimizations and utilize operator fusion to partition the whole shape-annotated graph into fused subgraphs. Then we conduct a hardware-aligned fine-tuning for the backbone operator within the subgraph to find a collection of high-performance schedules and propagate the backbone schedule to other operators in accordance with dependence analysis. Finally, we allocate dynamic-shape workloads into pre-compiled parameterized programs guided by our occupancy-targeted cost model at runtime. Our results outperform the baselines in terms of both kernel efficiency and end-to-end latency and prove the effectiveness of TSCompiler.

Acknowledgements This work was supported in part by National Natural Science Foundation of China (Grant Nos. 62090025, 92373207) and National Key Research and Development Program of China (Grant Nos. 2023YFB4405101, 2022YFB4400400, 2023YFB4405103, 2023YFB4405104).

References

1 Bai Y, Jones A, Ndousse K, et al. Training a helpful and harmless assistant with reinforcement learning from human feedback. 2022. ArXiv:2204.05862

2 Chen T, Moreau T, Jiang Z, et al. TVM: an automated end-to-end optimizing compiler for deep learning. In: Proceedings of the 13th USENIX conference on Operating Systems Design and Implementation, 2018

3 Chen T, Zheng L, Yan E, et al. Learning to optimize tensor programs. In: Proceedings of the 32nd International Conference on Neural Information Processing Systems, 2018

4 Zheng L, Jia C, Sun M, et al. Ansor: generating high-performance tensor programs for deep learning. In: Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation, 2020

5 Zhu H, Wu R, Diao Y, et al. ROLLER: fast and efficient tensor compilation for deep learning. In: Proceedings of 16th USENIX Symposium on Operating Systems Design and Implementation, 2022

6 Zheng B, Jiang Z, Yu C, et al. DietCode: automatic optimization for dynamic tensor programs. In: Proceedings of Machine Learning and Systems, 2022

7 Mu P, Liu Y, Wang R, et al. HAOTuner: a hardware adaptive operator auto-tuner for dynamic shape tensor compilers. IEEE Trans Comput, 2023, 72: 3178–3190

8 Shen H, Roesch J, Chen Z, et al. Nimble: efficiently compiling dynamic neural networks for model inference. In: Proceedings of Machine Learning and Systems 3, 2021

9 Devlin J, Chang M, Lee K, et al. BERT: pre-training of deep bidirectional transformers for language understanding. In: Proceedings of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2019

10 Dosovitskiy A, Beyer L, Kolesnikov A, et al. An image is worth 16×16 words: transformers for image recognition at scale. In: Proceedings of International Conference on Learning Representations, 2021

11 Lan Z, Chen M, Goodman S, et al. ALBERT: a lite BERT for self-supervised learning of language representations. 2019. ArXiv:1909.11942

12 Zhu K, Zhao W, Zheng Z, et al. DISC: a dynamic shape compiler for machine learning workloads. In: Proceedings of the 1st Workshop on Machine Learning and Systems, 2021

13 Grosser T, Verdoolaege S, Cohen A. Polyhedral AST generation is more than scanning polyhedra. ACM Trans Program Lang Syst, 2015, 37: 1–50

14 Baghdadi R, Cohen A. Scalable polyhedral compilation, syntax vs. semantics: 1–0 in the first round. In: Proceedings of the 12th International Workshop on Polyhedral Compilation Techniques (associated with HIPEAC 2020), 2020

15 Bastoul C, Zhang Z, Razanajato H, et al. Optimizing GPU deep learning operators with polyhedral scheduling constraint injection. In: Proceedings of /ACM International Symposium on Code Generation and Optimization (CGO), 2022

16 Eriksson D, Pearce M, Gardner J, et al. Scalable global optimization via local Bayesian optimization. In: Proceedings of Advances in Neural Information Processing Systems, 2019

17 Zhao J, Di P. Optimizing the memory hierarchy by compositing automatic transformations on computations and data. In: Proceedings of the 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020

18 Feng S, Hou B, Jin H, et al. TensorIR: an abstraction for automatic tensorized program optimization. In: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2023

19 Zhao J, Gao X, Xia R, et al. Apollo: automatic partition-based operator fusion through layer by layer optimization. In: Proceedings of Machine Learning and Systems 4 (MLSys 2022), 2022

20 Niu W, Guan J, Wang Y, et al. DNNFusion: accelerating deep neural networks execution with advanced operator fusion. In: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation, 2021

21 Zheng Z, Yang X, Zhao P, et al. AStitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMT architectures. In: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022

22 Li A, Zheng B, Pekhimenko G, et al. Automatic horizontal fusion for GPU kernels. In: Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization, 2022

23 Zhao J, Bastoul C, Yi Y, et al. Parallelizing neural network models effectively on GPU by implementing reductions atomically. In: Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, 2022

24 Vasilache N, Zinenko O, Theodoridis T, et al. Tensor comprehensions: framework-agnostic high-performance machine learning abstractions. 2018. ArXiv:1802.04730

25 Chetlur S, Woolley C, Vandermersch S. cuDNN: efficient primitives for deep learning. 2014. ArXiv:1410.0759

26 Hegde K, Tsai P, Huang S, et al. Mind mappings: enabling efficient algorithm-accelerator mapping space search. In: Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2021

27 Verdoolaege S, Janssens G. Scheduling for PPCG. CW Report, Department of Computer Science, 2017

28 Feautrier P. Some efficient solutions to the affine scheduling problem. Part II. multidimensional time. In: Proceedings of International Journal of Parallel Programming, 1992

29 Zhao J, Li B, Nie W, et al. 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, 2021

30 Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions. In: Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2015

31 Verdoolaege S. isl: an integer set library for the polyhedral model. In: Proceedings of International Congress on Mathematical Software, 2010

32 Sun W, Li A, Geng T, et al. Dissecting tensor cores via microbenchmarks: latency, throughput and numeric behaviors. IEEE Trans Parallel Distrib Syst, 2023, 34: 246–261

33 Vasilache N, Zinenko O, Bik A, et al. Composable and modular code generation in MLIR: a structured and retargetable approach to tensor compiler construction. 2022. ArXiv:2202.03293

34 Lattner C, Amini M, Bondhugula U, et al. MLIR: scaling compiler infrastructure for domain specific computation. In: Proceedings of the IEEE/ACM International Symposium on Code Generation and Optimization, 2021

35 Osama M, Merrill D, Cecka C, et al. Stream-K: work-centric parallel decomposition for dense matrix-matrix multiplication on the GPU. In: Proceedings of the 28th ACM SIGPLAN Annual Symposium on Principles and Practice of Parallel Programming, 2023

36 Jouppi N, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit. In: Proceedings of the 44th Annual International Symposium on Computer Architecture, 2017

37 Collins A, Grover V. Axon: a language for dynamic shapes in deep learning graphs. 2022. ArXiv:2210.02374

38 Lai R L, Shao J R, Feng S Y, et al. Relax: composable abstractions for end-to-end dynamic machine learning. 2023. ArXiv:2311.02103