



S-GPT: Shape-aware graph partitioning and tuning for sustainable inference of large-scale deep neural networks^{☆,☆☆}

Hongen Shao^a, Cen Chen^{a,b,*}, Xin Fang^a, Ziqian Zeng^c, Keqin Li^d

^a School of Future Technology, South China University of Technology, Guangzhou, 510641, China

^b Pazhou Laboratory, Guangzhou, 510330, China

^c Shien Ming Wu School of Intelligent Engineering, South China University of Technology, Guangzhou, 510641, China

^d Department of Computer Science, State University of New York, New Paltz, NY 12561, USA

ARTICLE INFO

Keywords:

Sustainable inference
Graph partitioning
Deep neural networks
Large language models

ABSTRACT

The rapid growth of large-scale deep neural networks has pushed inference workloads toward increasingly diverse shapes. In practical inference, these models are often invoked with tensors of varying shapes rather than a single fixed shape. Such shape variability reshapes operator loops and intermediate tensors, leading to shape-dependent operator costs, memory usage, fusion benefits, and backend tuning complexity. Consequently, graph partitions and kernel schedules optimized for one shape may become inefficient for another. Existing compilers either rely on fixed fusion patterns and static graph partitioning, or handle variable shapes mainly through symbolic representation and runtime specialization, leaving limited ability to jointly optimize graph partitioning and kernel tuning across representative shape distributions.

This work proposes S-GPT, a shape-aware compiler framework that co-optimizes graph-level partitioning and backend kernel tuning. At the graph level, we introduce a shape-aware operator weighting method that models both the expected tuning complexity and the cross-shape variance of each operator. Then, we further perform weight-guided subgraph partitioning, enabling different shape intervals to obtain customized partitions while preserving fusion legality and backend schedulability. Finally, we develop a shape-aware kernel tuning mechanism, where structurally similar subgraphs and neighboring shape intervals can reuse historical schedules, measurement records, and cost-model states. Experimental results demonstrate that, compared to the state-of-the-art compilation framework, S-GPT reduces compilation time by 1.9×, improves average speedups of 1.72× and 1.63× on the NVIDIA Tesla A40 and NVIDIA GeForce RTX3090, respectively.

1. Introduction

Recent advances in large-scale deep neural networks (DNNs) have accelerated their deployment in embedded and edge-oriented scenarios, including conversational agents [1–3], multimodal perception [4–6], and real-time decision making [7–9]. As model scale and architectural diversity continue to grow, especially in large language models (LLMs) with variable sequence lengths, inference workloads are increasingly executed under diverse representative shapes. Such dynamic workloads alter operator costs, memory pressure, fusion profitability, and backend tuning complexity, making graph partitions and kernel schedules optimized for one shape inefficient for another. Therefore,

sustainable inference requires compilers to maintain stable performance across shape variations while controlling subgraph variants and tuning overhead.

Existing compiler frameworks primarily depend on hand-optimized operator libraries, such as cuBLAS [10] and CUTLASS [11], which perform well on canonical operators like matrix multiplication, but provide limited coverage for the diverse and fast-evolving operator patterns of modern networks. Each architectural innovation necessitates custom kernels and scheduling rules, creating a widening gap between model evolution and compiler support. Automated compilers [12–17], including TVM [18], BladeDisc [19], and Apollo [17], aim to bridge this gap by systematically translating high-level computation graphs into efficient low-level kernels through automatic optimization and search.

[☆] This article is part of a Special issue entitled: ‘Sustainable Computing 3A’ published in Journal of Systems Architecture.

^{☆☆} This work was sponsored by the CAAI-MindSpore Open Fund, developed on OpenI Community.

* Corresponding author at: School of Future Technology, South China University of Technology, Guangzhou, 510641, China.

E-mail addresses: ftshaohongen@mail.scut.edu.cn (H. Shao), chencen@scut.edu.cn (C. Chen), 202510194375@mail.scut.edu.cn (X. Fang), zqzeng@scut.edu.cn (Z. Zeng), lik@newpaltz.edu (K. Li).

<https://doi.org/10.1016/j.sysarc.2026.103992>

Received 18 November 2025; Received in revised form 10 August 2026; Accepted 2 September 2026

Available online 10 September 2026

1383-7621/© 2026 Published by Elsevier B.V.

Most existing compilers adopt a two-layer hierarchical design. The graph layer performs structural transformations through operator fusion [17,20], layout rewriting [21], and graph partitioning [22,23], defining the granularity and boundaries of computation units. Fig. 1 shows the computation graph of LLMs. The tuning layer then explores scheduling design spaces for each subgraph, selecting tiling, parallelization, and memory reuse strategies to maximize hardware efficiency [12,18,24]. In principle, this hierarchical design enables performance portability of LLMs. While this separation enables portability and systematic optimization, the interdependence between layers creates a fundamental tension in which overly fine-grained partitions limit fusion opportunities and reuse, whereas overly coarse partitions enlarge the search space and impede convergence. This tension directly affects the compiler's execution efficiency.

Current compilers still rely on a pattern-based strategy [17,18,25] in the graph layer, capturing only trivial motifs such as *matmul+add* or simple element-wise chains. Such approaches fail to generalize to the diverse operator combinations in modern networks, leaving large optimization opportunities untapped. The tuning layer is further constrained by these partitions, often missing profitable fusions among compute-intensive operators, thereby limiting data locality.

To address this, recent work has explored adaptive clustering partitioning [22,26], moving beyond fixed fusion templates by grouping operators according to structural affinity. However, these methods are largely designed for static workloads and do not generalize well to the dynamic input shapes prevalent in large-scale DNN inference [21,27]. In production settings, tensor shapes often vary continuously, such as the sequence lengths in LLMs, which directly influence computational cost, memory footprint, and fusion profitability. Under such variability, static partitions may suffer substantial runtime degradation, thereby limiting scalability and sustainable execution.

Some research [19,28–30] such as Relax [30] and BladeDisc [19] extend IR representations or incorporate runtime-aware fusion, yet they generally retain pattern-driven graph partitioning. As a result, under dynamic workloads, LLMs struggle to maintain an effective balance between fusion depth and execution stability. This indicates a need for partitioning mechanisms that jointly reason about operator semantics, shape variability, and tuning cost to achieve sustainable, scalable, and efficient execution for large-scale DNNs. Enlightened by these factors, we identify several key challenges inadequately addressed by existing approaches: (i) static operator weights do not generalize across shape distributions, leading to unstable partitioning outcomes; (ii) explosive growth in the number of subgraphs impacts the framework's kernel tuning efficiency.

In this work, we propose S-GPT, a shape-aware graph partitioning and tuning framework for sustainable inference of large-scale DNNs under varying input shapes. We jointly consider operator semantics, shape information, and backend tuning cost. Instead of relying on static fusion patterns or single-shape operator weights, we model the influence of representative shape distributions on operator complexity and use this information to guide graph partitioning. We further organize input shapes into shape buckets, where each bucket can obtain a customized partitioning result under semantic and backend constraints. To avoid the rapidly increasing tuning cost caused by multiple shapes and subgraph variants, we introduce a shape-aware mechanism that reuses historical schedules, measurement records, and cost-model states among structurally similar subgraphs and neighboring shape intervals. In summary, our contributions are:

- We propose a shape-aware weighted approach for operators in large-scale DNNs, which models the expectation and variance of operator complexity across different shape distributions.
- We greedily divide the operators based on the weights of the shape-aware operators, and different fusion granularities can be used for different shape intervals.
- We design a shape-aware kernel tuning mechanism to reduce redundant kernel search.
- Experimental results demonstrate that S-GPT reduces compilation time by 1.9×, improves average speedups of 1.72× and 1.63× on

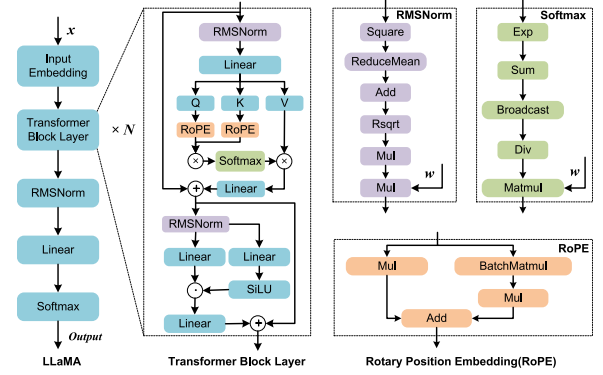


Fig. 1. An illustration of the model's computational graph, which shows the structure of the LLaMA, including composite operators such as Softmax, RMSNorm, and RoPE.

the NVIDIA Tesla A40 and NVIDIA GeForce RTX3090, respectively.

2. Background and motivation

2.1. Preliminaries of computation graph optimization

In deep neural network compilers [12–16], a model defined in a high-level framework is typically transformed into a directed acyclic computational graph $G = V, E$, where nodes represent operators and edges denote tensor data flow. As illustrated in Fig. 1, models such as LLaMA contain numerous composite operators (e.g., RMSNorm, Softmax, RoPE), which are often decomposed into meta-operators (e.g., Exp, ReduceSum, Div for Softmax).

Taking a computation graph as input, TVM [18] and Relay [31] lower high-level subgraphs into tensor programs through graph rewriting, operator fusion, layout transformation, and schedule optimization. Based on this design, compilers such as TASO [23], DNNFusion [25], AStitch [16], and Apollo [17] further explore graph substitutions, operator fusion, and partition-based optimization to reduce kernel launch overhead, eliminate redundant memory access, and improve data locality. Dynamic-shape compilers, including Nimble [29], DISC [32], BladeDisc [19], and Relax, extend this pipeline with symbolic shape representation, shape constraint propagation, and runtime specialization to support variable input dimensions. At the backend level, Anso [33], AKG [12], and Triton generate efficient kernels by exploring scheduling choices such as tiling, parallelization, memory reuse, and thread binding.

2.2. Motivation

Model inference in real-world scenarios exhibits significant shape variability, which directly affects the optimal partitioning of computational graphs. This phenomenon manifests at two levels. (i) When input shapes change, the computational intensity, memory footprint, and intermediate tensor size of each operator also change. For example, in LLM inference, different sequence lengths lead to different costs for MatMul, Softmax, and element-wise operators. (ii) The same fusion pattern may bring performance benefits under one shape but become inefficient under another shape. As shown in Fig. 2, the optimal partitioning results of the same computation graph are different under different input shapes. This indicates that the compiler can incorporate shape information into the partitioning process and identify shape-aware subgraphs.

In existing research on Anso, the cost model is trained using measurement records collected from tensor programs, and the learned

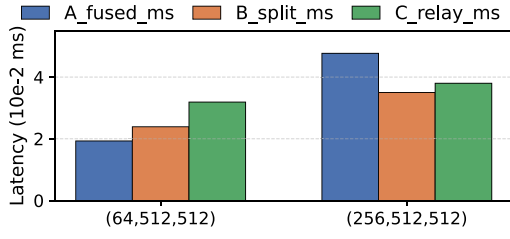


Fig. 2. Illustration of the computation graph latency under different shapes and different subgraph partitions.

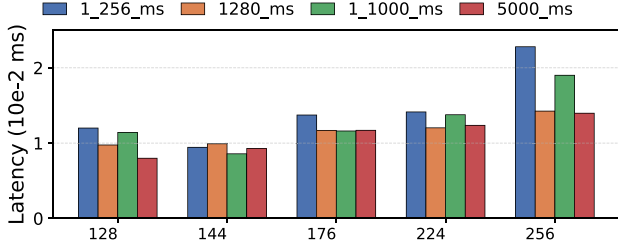


Fig. 3. Illustration of the computation graph latency under different shapes and different tuning methods (1_256_ms and 1_1000_ms are tuned individually for each shape; 1280_ms and 5000_ms are shared-shape tuning).

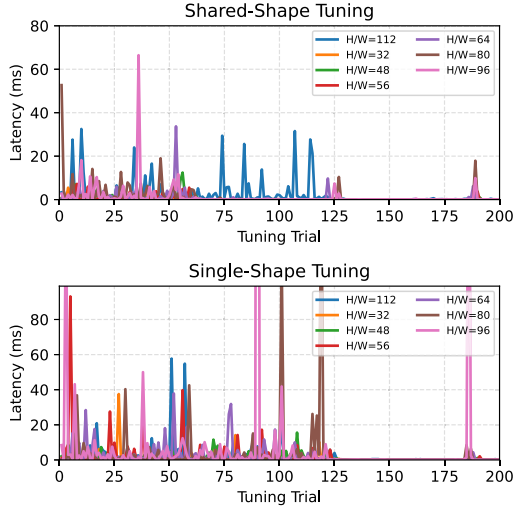


Fig. 4. Illustration of the delayed convergence state of the computation graph under two tuning modes (shared-shape tuning and single-shape tuning), as the number of tuning iterations increases.

model is used to guide subsequent schedule exploration. This means that tuning records from structurally similar subgraphs, or from the same subgraph under different shapes, can provide useful information for predicting better schedules. As shown in Fig. 3, compared with tuning each shape (1_256_ms and 1_1000_ms) independently, jointly tuning multiple shapes (1280_ms and 5000_ms) enables the cost model to learn more general performance patterns and reduces redundant exploration. Meanwhile, as shown in Fig. 4, shared-shape tuning accelerates convergence to a high-performance schedule. *This motivates a shape-aware tuning mechanism that shares tuning knowledge across subgraphs and shape intervals.*

2.3. Challenges

The above observations reveal that shape variability affects both the graph-level partitioning decision and the backend tuning process. To

fully exploit shape information for efficient model inference, two key challenges must be addressed.

Challenge 1: Lack of Shape Awareness. The optimal partitioning of a computational graph varies with input shapes because operator costs, memory footprints, and fusion benefits are all shape-dependent. However, existing graph partitioning methods [18,26] usually rely on fixed fusion patterns or static operator weights, which cannot accurately capture how fusion profitability varies across different shapes. A partition that performs well for one shape may become inefficient for another due to excessive intermediate memory usage, irregular indexing, or reduced data locality. Therefore, the first challenge is to construct shape-aware operator weights and partitioning criteria that reflect the performance behavior of the same computation graph across different input shapes. The partitioning mechanism should not only identify profitable fusion regions for a single shape, but also preserve stable performance across a range of representative shapes.

Challenge 2: Prohibitive Kernel Tuning Cost. Introducing multiple shapes into the optimization process inevitably increases the number of tunable subgraph variants. If each shape-specific subgraph is tuned independently, the measurement cost and search time will grow rapidly, making the overall compilation process inefficient. Meanwhile, structurally similar subgraphs and different shapes of the same subgraph often share common scheduling characteristics, such as similar tiling preferences, memory reuse patterns, and parallelization strategies. Although Ansor already uses a learning-based cost model to exploit measurement records during schedule search, it does not explicitly organize tuning knowledge according to shape intervals and subgraph similarity. Therefore, how to reuse tuning information across multiple shapes and similar subgraphs to accelerate cost-model convergence while avoiding negative transfer from dissimilar shapes. An effective mechanism should reduce redundant tuning trials and improve convergence efficiency without sacrificing the final runtime performance.

3. S-GPT design

S-GPT jointly optimizes graph partitioning and kernel tuning for efficient inference of large-scale DNNs under dynamic input shapes. This section elaborates on this.

3.1. Overview

The proposed framework enables sustainable inference of large-scale deep neural networks through shape-aware graph partitioning and kernel tuning under dynamic input shapes. S-GPT consists of these major stages: *shape-aware operator weighting*, *weight-guided subgraph partitioning*, and *shape-aware kernel tuning*. As illustrated in Fig. 5, these components jointly transform a computation graph into optimized kernels with controllable tuning complexity.

Shape-aware Operator Weighting (Section 3.2). S-GPT models each operator's cost under dynamic shapes by treating loop extents as random variables over representative shape distributions, computing both expectation and variance. This yields shape-aware weights that capture average tuning effort and robustness, enabling stable partitioning decisions across diverse runtime shapes.

Weight-Guided Subgraph Partitioning (Section 3.3). Using these weights, S-GPT groups representative shapes into buckets according to dominant dynamic dimensions and performs greedy partitioning within each bucket. Candidate merges are guided by shape-aware weights and locality benefits, while DAG, legality, and tuning-budget constraints ensure valid and tunable subgraphs.

Shape-aware Kernel Tuning (Section 3.4). S-GPT associates each shape bucket with its partitioned subgraphs and reuses historical schedules, measurement records, and cost-model states from a global tuning database. This enables similar subgraphs and neighboring shape intervals to share tuning knowledge, reducing redundant kernel search.

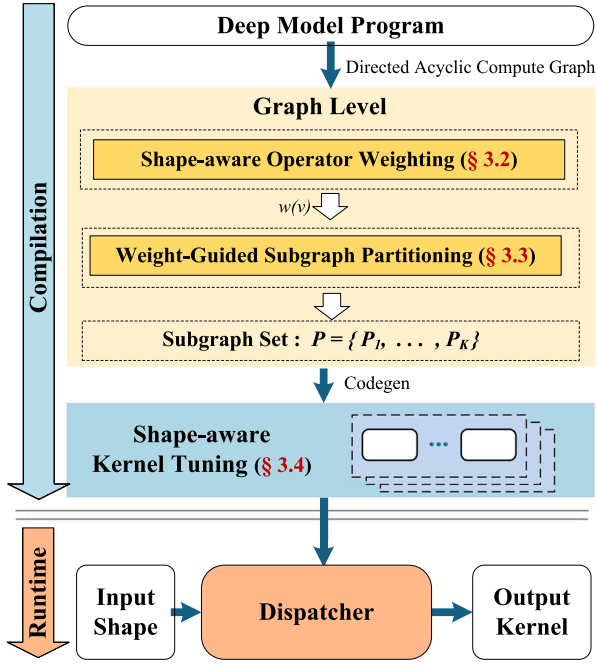


Fig. 5. Overview of S-GPT.

3.2. Shape-aware operator weighting

A central limitation of existing partitioning frameworks (e.g., AGO [26]) lies in their reliance on static operator weights to guide clustering decisions. These weights are typically computed from the tensor shape observed at compile time, assuming a static input shape. Concretely, the weight of an operator v with loop extents L_v is often defined as:

$$w_v = c \cdot \prod_{l \in L_v} \log(s_l) + b, \quad (1)$$

where s_l denotes the loop size along dimension l , and c, b are constants calibrated to approximate tuning complexity. While effective for single-shape workloads, this formulation lacks the expressiveness to characterize operator complexity over a set of input shapes.

To address this, we introduce shape-aware operator weighting, which extends the static formulation into a model that explicitly models shape distributions. Instead of assuming fixed loop extents, we treat them as random variables drawn from an input shape distribution D_s . We first define the expected weight of operator v as:

$$\bar{w}_v = c \cdot \prod_{l \in L_v} \mathbb{E}_{s \sim D_s} [\log(s_l)] + b, \quad (2)$$

which reflects the operator's average tuning complexity under the shape distribution. However, relying solely on expectation ignores the fact that dynamic shapes may induce large performance variance. To capture this, we augment the weight with a variance-sensitive term:

$$\tilde{w}_v = \bar{w}_v + \lambda \cdot \sqrt{\sum_{l \in L_v} \text{Var}_{s \sim D_s} [\log(s_l)]}, \quad (3)$$

where $\lambda \geq 0$ is a risk factor controlling sensitivity to uncertainty. The proposed weights characterize the tuning complexity of operators in two ways: the expectation term estimates their average complexity over representative shapes, while the variance term measures their sensitivity to shape variations. Therefore, operators with high average tuning costs or large cross-shape variability are given greater weights, preventing them from being over-merged into excessively large subgraphs and improving stability across multiple shape partitions.

Approximation via Representative Shapes. In practice, the full distribution D_s is rarely available. Rather than assuming a specific parametric distribution (e.g., uniform or Gaussian), we adopt an empirical distribution derived from observed inputs. Specifically, we approximate D_s using a representative shape set S_{rep} , collected from profiling logs or by clustering observed input shapes. This empirical approximation enables the weighting scheme to capture realistic workload characteristics without introducing bias from manually assumed distributions. Expectation and variance terms are then estimated via Monte Carlo sampling over S_{rep} :

$$\mathbb{E}_{s \sim D_s} [\log(s_l)] \approx \frac{1}{|S_{rep}|} \sum_{s \in S_{rep}} \log(s_l), \quad (4)$$

$$\text{Var}_{s \sim D_s} [\log(s_l)] \approx \frac{1}{|S_{rep}|} \sum_{s \in S_{rep}} (\log(s_l) - \mu_l)^2, \quad (5)$$

where μ_l denotes the sample mean. In this way, the operator weights reflect both the average computational cost and its variability under practical shape distributions. Substituting Eqs. (4) and (5) into Eq. (3) yields a practical estimate of $\tilde{w}_v$, which can be seamlessly plugged into the clustering algorithm without modifying its overall structure.

3.3. Weight-guided subgraph partitioning

With shape-aware operator weights available, the next challenge is to design a partitioning scheme that exploits these weights while also incorporating pattern-based fusion knowledge widely adopted in modern compilers.

Objectives. Formally, given a computation graph $G = (V, E)$, each node $v \in V$ is associated with a shape-aware operator weight $\tilde{w}_v$, and each edge $(u, v) \in E$ is labeled with a data transfer volume $D(u, v)$. The partitioning algorithm seeks a decomposition $P = \{P_1, \dots, P_K\}$ that satisfies the following objectives.

$$W(P_i) = \sum_{v \in P_i} \tilde{w}_v \leq W_{\max}, \quad \forall i, \quad (6)$$

where $W(\cdot)$ represents the aggregated weight of a subgraph, and $W_{\max}$ is the predefined upper bound on the subgraph weight. Low inter-subgraph communication is minimized to reduce tensor movement across partition boundaries.

$$\text{Cut}(P) = \sum_{\substack{(u,v) \in E \\ u \in P_i, v \in P_j, i \neq j}} D(u, v). \quad (7)$$

Together, these constraints ensure compiler-feasible partitioning without sacrificing fusion efficiency. Minimizing $\text{Cut}(P)$ improves data locality by co-locating producer-consumer operators linked by high-volume tensor edges, enabling local reuse of intermediate tensors rather than cross-partition materialization and transfer. This reduces off-chip traffic and launch overhead, while increasing opportunities for shared-memory reuse and schedule co-optimization.

Partitioning. As shown in Fig. 6, following the derivation of shape-aware operator weights, we categorize input shapes into dedicated buckets and partition the computation graph in a bucket-wise manner. We then perform greedy partitioning intra-bucket, thereby enabling customized partition topologies that align precisely with local shape.

(1) *Stage 1: Shape Bucket.* We first divide the representative shape set S_{rep} into several shape buckets. The bucket construction is based on the dominant dynamic dimensions that affect operator loop bounds. For LLMs, the dominant dimension is usually the sequence length. For CNNs, it is usually the spatial size H/W . The related $M/N/K$ dimensions determine matrix multiplication. We denote the resulting buckets as $B_k = \{s_i | \Delta W(s_i, s_j) < \eta, s_i, s_j \in B_k\}$. Shapes with small descriptor distance ΔW are placed into the same bucket. A bucket is accepted only when the shapes inside it have similar operator weights for high-cost operators. Otherwise, the bucket is further split. This

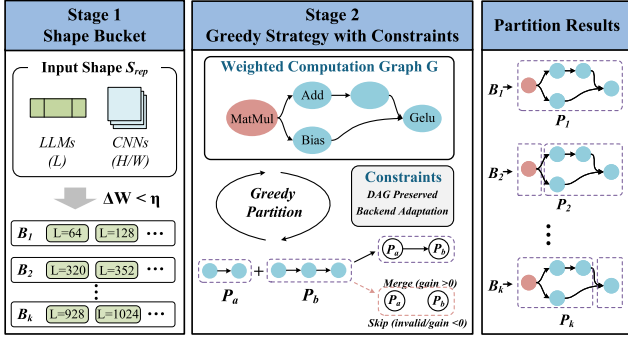
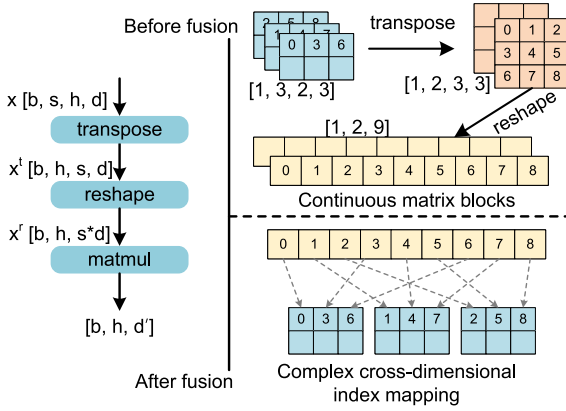


Fig. 6. Overview of weight-guided subgraph partitioning.

Fig. 7. Illustration of the inefficiency introduced by fusing *transpose*, *reshape*, and *matmul*.

prevents shapes with very different tuning behaviors from sharing the same partition.

(2) *Stage 2: Greedy Strategy with Constraints*. For each bucket B_i , we recompute the operator weights and perform greedy subgraph partitioning. Initially, each operator node $v \in V$ is treated as a singleton subgraph. The initial partition is $P = \{v_1, v_2, \dots, v_n\}$. We then build a candidate merge set M . A candidate (P_a, P_b) is generated only when two subgraphs are connected by producer-consumer edges in G . For each candidate, we compute a merge gain $gain_i(P_a, P_b)$ under bucket B_i . The gain considers two factors: the saved intermediate tensor transfer and the weight of the operators. Candidates with larger positive gains are merged first.

The greedy process is performed iteratively. At each step, we select the candidate with the largest $gain_i(P_a, P_b)$. If the gain is positive and all constraints are satisfied, P_a and P_b are merged into a new subgraph P_{ab} . Then, the candidate set M is updated by adding new merge candidates between P_{ab} and its neighboring subgraphs. If a candidate violates any constraint, it is discarded. The process stops when no legal candidate has positive gain. In this way, high-weight operators tend to become the core of subgraphs, while lightweight operators such as *Add*, *Bias*, and *GELU* are absorbed when they improve locality.

In addition, each candidate merge must pass several constraints. First, the merged subgraph must still be a directed acyclic graph. Second, the operator semantics must be compatible with the backend code generation. As shown in Fig. 7, the fusion of *Transpose*, *Reshape*, and *Matmul* is unsuitable if it causes the access pattern to become discontinuous. In addition, we also allow a subgraph to contain multiple *Matmul* operators. This relaxes the overly conservative rule that only one compute-intensive operator can appear in a fused subgraph.

After the greedy process finishes, each bucket B_i obtains its own partition set $P_i = \{P_{i,1}, P_{i,2}, \dots, P_{i,k}\}$. Different buckets may produce

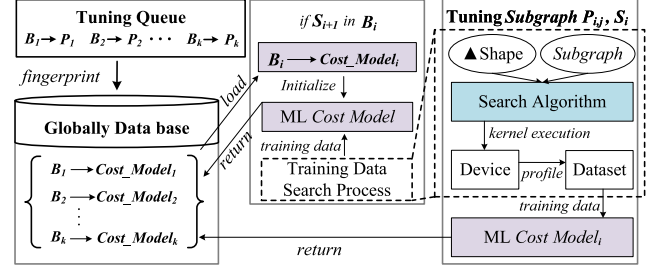


Fig. 8. Overview of shape-aware kernel tuning. Subgraph structural similarity is introduced, allowing similar subgraphs to share a cost model, thereby reducing the tuning time.

different partitions. Small-shape buckets may prefer more aggressive fusion to reduce kernel launch overhead, while large-shape buckets may keep smaller subgraphs to control tuning cost and resource pressure. Finally, we record the mapping $B_i \rightarrow P_i$, which is used by the following kernel tuning stage.

3.4. Shape-aware kernel tuning

Generating efficient kernels for dynamic workloads with varying shape characteristics requires not only adaptive cost models but also tight integration with upstream graph-level decisions. To this end, we explicitly establish a close interaction between subgraph partitioning and kernel tuning, bridging the gap between high-level structural optimization and low-level execution efficiency. Task-centric design is effective for fixed-shape workloads, but efficiency decreases when the same model is compiled for multiple input shapes. In such scenarios, many subgraphs generated from different shape intervals share similar operator structures and scheduling spaces, but they are still tuned independently. As a result, measurements collected for one subgraph or one shape are difficult to explicitly benefit other structurally similar subgraphs, leading to repeated search and excessive tuning overhead.

As shown in Fig. 8, we introduce a shape-aware kernel tuning mechanism. After subgraph clustering, each shape bucket B_i is associated with a subgraph set P_i , forming a tuning queue $\{B_1 \rightarrow P_1, B_2 \rightarrow P_2, \dots, B_k \rightarrow P_k\}$. For each tuning task, S-GPT extracts the subgraph fingerprint and queries the global database, where each bucket B_i maintains a historical $Cost_Model_i$. If a new shape S_{i+1} falls into an existing bucket, the corresponding cost model is loaded to initialize the ML cost model instead of tuning from scratch. The search algorithm then scores candidate schedules for the target subgraph $P_{i,j}$ and shape, executes promising candidates on the device, and uses the profiled latency to update $Cost_Model_i$. The updated model is returned to the database, enabling later subgraphs and shapes within the same bucket to reuse accumulated tuning knowledge. Compared with Ansor-style isolated tuning, this mechanism builds a reusable tuning loop across subgraphs and within the same bucket.

4. Implementation

Algorithm 1 presents the implementation pipeline of S-GPT. Given a computation graph and a representative shape set, S-GPT first groups input shapes into shape buckets according to their dominant dynamic dimensions. For each bucket, it recomputes the shape-aware operator weights and performs greedy partitioning from singleton operators. At each step, the algorithm selects the connected subgraph pair with the largest positive merge gain. The merge is accepted only when it preserves the DAG structure, satisfies backend legality constraints, and keeps the subgraph weight within the tuning budget W_{max} ; otherwise, the candidate is discarded. This process stops when no legal candidate can further improve the partition. Finally, S-GPT records the bucket-wise mapping $B_i \rightarrow P_i$ and invokes shape-aware kernel tuning, where

Algorithm 1 Implementation of S-GPT Subgraph Partition

Input: $G = (V, E)$, computation graph,
 S_{rep} , representative shape set,
Output: $\mathcal{M} = B_i \rightarrow P_i$

```

1: function S-GPT_COMPILE( $G, S_{\text{rep}}$ )
2:    $B \leftarrow \text{BuildShapeBuckets}(S_{\text{rep}})$ 
3:    $\mathcal{M} \leftarrow \emptyset$ 
4:   for each bucket  $B_i \in B$  do
5:      $\tilde{w}_i(v) \leftarrow \text{ComputeWeight}(v, B_i), \forall v \in V$ 
6:      $\mathcal{P}_i \leftarrow \{\{v\} \mid v \in V\}$ 
7:      $C_i \leftarrow \text{BuildCandidates}(G, \mathcal{P}_i)$ 
8:     while  $C_i \neq \emptyset$  do
9:        $g_i(P_a, P_b) \leftarrow \text{ComputeGain}(P_a, P_b, B_i),$ 
        $\forall (P_a, P_b) \in C_i$ 
10:       $(P_a, P_b) \leftarrow \arg \max_{(P_x, P_y) \in C_i} g_i(P_x, P_y)$ 
11:      if  $g_i(P_a, P_b) \leq 0$  then
12:        break
13:      end if
14:      if  $W(P_a \cup P_b) \leq W_{\max}$  then
15:         $P_{ab} \leftarrow \text{Merge}(P_a, P_b)$ 
16:         $\mathcal{P}_i \leftarrow (\mathcal{P}_i - \{P_a, P_b\}) \cup \{P_{ab}\}$ 
17:         $C_i \leftarrow \text{UpdateCandidates}(C_i, P_{ab}, G, \mathcal{P}_i)$ 
18:      else
19:         $C_i \leftarrow C_i - \{(P_a, P_b)\}$ 
20:      end if
21:    end while
22:     $\mathcal{M}[B_i] \leftarrow P_i$ 
23:  end for
24:  return  $\mathcal{M}$ 
25: end function

```

historical schedules, measurement records, and cost-model states in the tuning database are reused for structurally similar subgraphs and neighboring shape buckets.

5. Evaluation

Our objective is to demonstrate that our method effectively generates optimal executable kernels through adaptive graph partitioning and outperforms state-of-the-art baseline methods.

5.1. Experimental setup

DNN Workloads. We evaluate the end-to-end inference latency of representative models across a broad spectrum of input shapes and hardware backends. Specifically, we selected four widely deployed architectures, including computer vision and transformer-based models as benchmarks, such as Bert [34], ViT [35], LLaMA [36], and ResNet [37]. These networks contain a large part of operators commonly seen in DNNs. Each exhibits distinct operator heterogeneity and sensitivity to input shape variations. For each model, we sample input shapes from a representative interval as shown in Table 1. In the current evaluation, the sampled dynamic dimensions primarily correspond to sequence length for transformer workloads and spatial resolution for conventional vision models.

Platform. We evaluate S-GPT on two hardware platforms. The first platform is equipped with two NVIDIA Tesla A40 GPUs with 40 GB of memory and two Intel(R) Xeon(R) Silver 4314 CPUs. The second platform features two NVIDIA GeForce RTX 3090 GPUs with 24 GB of memory and two Intel(R) Xeon(R) Silver 4210R CPUs. The software environment for our experimental evaluation includes CUDA 12.1, PyTorch 2.1.2, and TVM 0.14.

Table 1

Details for transformer and conventional class models.

Transformer				
Method	Heads	M/N	Hidden	# Cases
Bert [34]	12	[16, 512]	768	32
ViT [35]	12	[16, 256]	768	32
LLaMA [36]	32	[16, 4096]	128	32
Conventional				
ResNet [37]	–	[64, 224]	1024	32

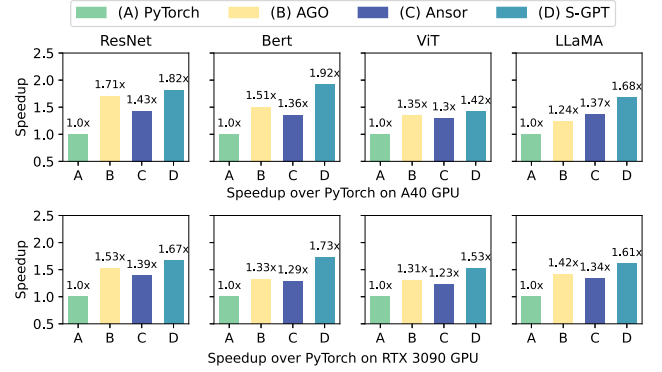


Fig. 9. Comparison of end-to-end inference performance between PyTorch, AGO, Ansor, and S-GPT.

Baselines. We adopt an evaluation strategy that compares S-GPT against both compiler-based approaches and manually optimized implementations. For static compilation baselines, we evaluate Ansor [33] (configured with 1000 tuning trials per workload) and AGO [26]. In addition, we include PyTorch [38] as the baseline deep learning framework.

Evaluation Index. We present the evaluation results from three aspects: (1) Inference time: the execution latency of the end-to-end model, which we take as the average of twenty runs. (2) Subgraph partitioning display: the subgraph structure after the model is converted into a computation graph and partitioned. (3) Tuning time: the time to generate the optimal executable binary file for the model during the offline compilation stage, in hours (h). For the variance-sensitive weighting in Eq. (4), we use a fixed $\lambda = 0.5$ in all experiments.

5.2. End-to-end performance

We first evaluate the end-to-end inference latency and compare S-GPT against PyTorch, Ansor, and AGO. Fig. 9 reports the latency distribution normalized to PyTorch baselines on A40 and RTX 3090. The experimental results are the average values obtained from the model's sampling.

On the A40, compared with PyTorch, S-GPT achieves 1.82x, 1.92x, 1.42x, and 1.68x speedups on ResNet, Bert, ViT, and LLaMA, respectively, outperforming all baselines. A similar pattern is observed on the RTX 3090. S-GPT achieves 1.67x, 1.73x, 1.53x, and 1.61x speedups on ResNet, Bert, ViT, and LLaMA, respectively. The close alignment of results across both GPUs indicates that S-GPT's optimizations generalize well to different architectures without relying on platform-specific heuristics.

5.3. Subgraph partitioning under different shapes

We next use the LLaMA model as a case study to compare the partitioning behavior of S-GPT and MindSpore [39] under different sampled shapes, thereby examining whether the performance gain of S-GPT remains stable across shape variations rather than being dominated by

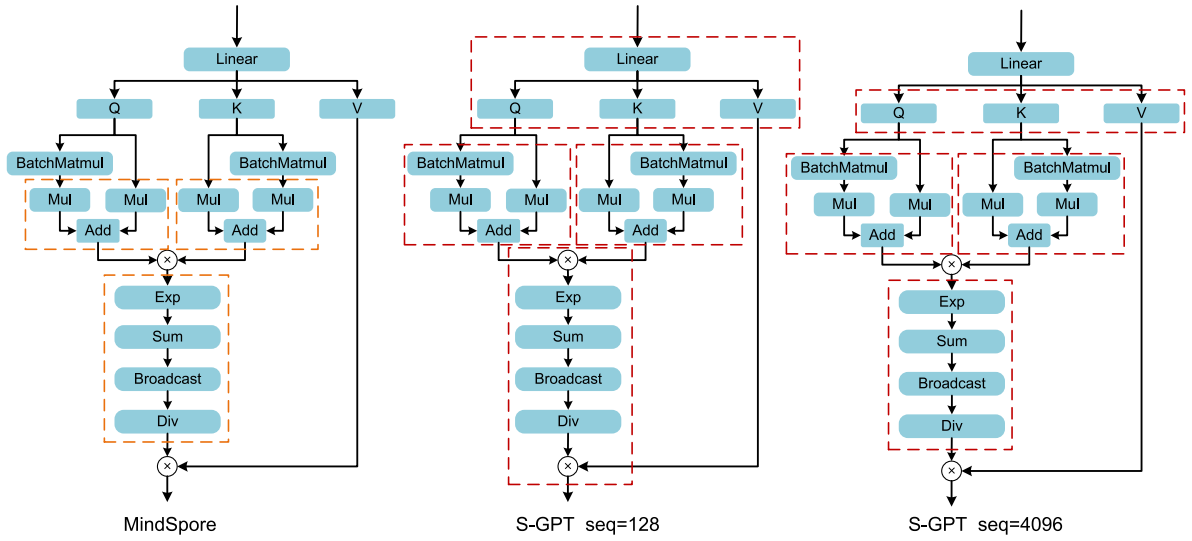


Fig. 10. Comparison of subgraph partitioning strategies for the LLaMA model across varying tensor shapes using MindSpore and the S-GPT framework.

Table 2

Performance comparison between MindSpore [39] and S-GPT on the A40 platform for the LLaMA model under various shapes (unit: 10^1 ms).

	MindSpore	S-GPT		MindSpore	S-GPT
S1	5.14	4.27	S5	11.74	9.78
S2	5.77	4.81	S6	22.35	19.11
S3	5.47	4.37	S7	52.47	42.10
S4	7.32	6.31	S8	133.79	105.99

a few favorable cases. The sampled configurations are drawn from the shape range in Table 1 and denoted as $S1, \dots, S8$.

As reported in Table 2, S-GPT delivers an average $1.2\times$ performance improvement over MindSpore on sampled shapes, and consistently outperforms MindSpore on all sampled shapes, from short to long sequence lengths. The latency reduction is not confined to a small subset of inputs. Instead, the advantage is preserved throughout the sampled shape spectrum, with the absolute benefit becoming more pronounced as the sequence length increases. This result indicates that the performance improvement of S-GPT is shape-consistent, rather than an artifact of averaging over heterogeneous dynamic-shape inputs.

Fig. 10 further illustrates the partitioning differences between the two methods at the computation graph. In our evaluated setup, MindSpore produces largely similar partition layouts across the sampled sequence lengths. This suggests that, although the framework supports dynamic-shape execution, its graph-level fusion behavior in this setting remains relatively rule-driven, with limited partition-boundary adaptation to shape variation. In contrast, S-GPT adjusts partition boundaries according to the underlying shape. For short sequences (e.g., $\text{seq} = 128$), it fuses multiple MatMul-centered operators into larger regions to reduce launch overhead. For long sequences (e.g., $\text{seq} = 4096$), it adopts more conservative partitions to avoid over-fusion and resource-induced degradation. This adaptive behavior explains why S-GPT maintains latency gains over MindSpore across all sampled shapes in Table 2.

Unlike rule-based partitioners that isolate compute-intensive operators such as MatMul, S-GPT fuses surrounding reshape, broadcast, and element-wise operators into cohesive regions whenever legality and shape robustness permit. This converts global-memory communication into intra-subgraph reuse. The effect is particularly clear in LLaMA, where S-GPT avoids excessive fragmentation and produces more compact regions around attention and feed-forward blocks.

Table 3

Comparison of tuning time between Ansor and S-GPT across different models on the A40 platform.

Model	Nodes	Candidate Kernels	Tuning Time (Ansor/S-GPT)
ResNet	122	1031	1.18 h/0.88 h
Bert	681	7305	3.23 h/1.8 h
ViT	569	5921	2.62 h/1.32 h
LLaMA	966	8495	4.25 h/2.36 h

5.4. Tuning time

We further analyze the tuning overhead of S-GPT and summarize the results in Table 3, including the number of primitive graph nodes, generated candidate kernels, and the total end-to-end tuning time for all evaluated models. In principle, the number of candidate kernels grows quadratically with the number of graph nodes. However, our partitioning and constraint-guided fusion effectively prunes the search space, eliminating redundant or semantically invalid kernel candidates. Across all models, the actual number of candidates remains within a manageable range. This reduction directly translates into shorter tuning cycles.

Benefiting from cross-subgraph knowledge transfer, S-GPT reuses cost models between structurally similar subgraphs and shape intervals, avoiding repeated tuning for equivalent computation patterns. Consequently, the overall tuning time is reduced by $1.9\times$ on average compared with Ansor. Even for the LLaMA model with the most complex attention structure, S-GPT can be tuned in 2.36 h, fully demonstrating the scalability of its adaptive kernel tuning strategy.

5.5. Impact of semantic constraints

We now evaluate the role of semantic constraints in shaping both the efficiency and legality of subgraph fusion. Using the LLaMA model as the case study, we compare two variants of our framework: S-GPT (w/ constraints), which enforces the constraint-guided fusion rules, and S-GPT (w/o constraints), which relies purely on weight-based partitioning without semantic validation. The end-to-end tuning time and the generated subgraph structure were measured.

Table 4 reports the tuning time results. Without constraints, the partitioner occasionally merges operators that are semantically incompatible or shape-fragile, such as MatMul fused with Reduce along non-aligned axes, or Transpose fused into large MatMul kernels. While

Table 4

S-GPT's tuning time for ResNet, Bert, ViT, and LLaMA with and without constraints.

	ResNet	Bert	ViT	LLaMA
w/o constraints	1.2	2.31	1.69	3.15
w/ constraints	0.88	1.8	1.32	2.36

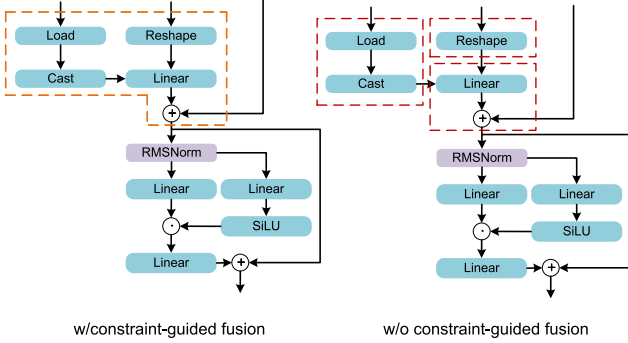


Fig. 11. Comparison of subgraph partitioning strategies for the LLaMA model across varying tensor shapes using MindSpore and S-GPT framework.

such fusions may appear beneficial, they often lead to poor performance or kernel scheduling crashes due to deteriorated memory access patterns. We observed that the constrained version of S-GPT eliminated some ill-conditioned fusions, and the average tuning time for all models was reduced by 1.26 $\times$ compared to the unconstrained version.

Fig. 11 also illustrates representative subgraph partitions. In the unconstrained case, we observe over-fused subgraphs in which heterogeneous operators (e.g., *reshape + matmul*) are forced into a single kernel, leading to intractable tuning. In contrast, S-GPT with constraints yields cleaner partitions, retains computationally intensive operators as anchor nodes, merges element-level chains, and only merges Reshape and Transpose operators if the data storage layout remains unchanged, as analyzed in **Fig. 7**. These partitions are not only semantically effective but also easy to tune, resulting in higher-quality downstream scheduling.

5.6. Subgraph weight distribution

Fig. 12 presents the subgraph weight distribution of the LLaMA model, where each bin represents a logarithmic weight interval and the vertical axis indicates the number of subgraphs within that range.

To further characterize the differences in partitioning behavior, we analyze the subgraph structures generated on the LLaMA model. MindSpore produces 1715 subgraphs, exhibiting a highly skewed weight distribution: over 820 subgraphs fall into the lowest weight bin (< 30), and the average subgraph weight is only 81, with a median of 37, indicating significant fragmentation. Moreover, compute-intensive operators such as MatMul often appear as isolated subgraphs, surrounded by multiple reshape/transpose-only regions. Such over-segmentation discards profitable fusion opportunities and forces the backend tuner to operate on small, locality-poor kernels.

In contrast, S-GPT generates 974 subgraphs, a 43% reduction with a markedly more balanced weight profile. Approximately 310 subgraphs lie in the low-weight bin (< 30), while a substantial portion occupies mid-range bins, yielding an average subgraph weight of 236 and a median of 87. This distribution reflects S-GPT's ability to consolidate semantically compatible operators into larger, stable regions without exceeding tuning budgets. Crucially, S-GPT avoids isolating MatMul kernels, reshapes, broadcasts, and elementwise chains surrounding attention blocks are often fused into cohesive units, preserving locality and enabling the tuner to explore richer scheduling spaces. These

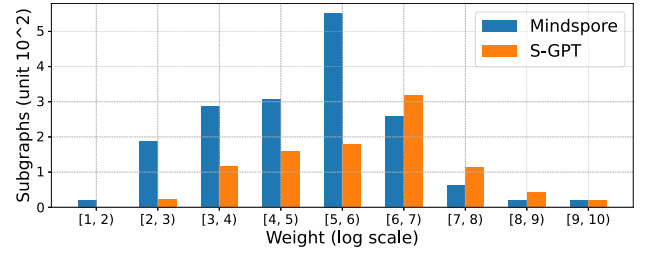


Fig. 12. Subgraph weight distribution for LLaMA. The x-axis is a logarithmic value with base 2.

structural statistics further support the shape-wise latency comparison in **Table 2**. In the evaluated LLaMA setting, MindSpore tends to produce a more fragmented partition structure, whereas S-GPT yields fewer and more balanced subgraphs. This locality benefit comes not from blindly enlarging subgraphs, but from selectively consolidating communication-critical operator neighborhoods while keeping the resulting regions tunable and shape-stable, thereby preserving more profitable fusion opportunities and explaining the consistent latency gains over MindSpore across different shapes.

6. Related work

Pattern-based Operator Fusion. Traditional compilers [15–17,23,25,31,40] primarily perform pattern-based operator fusion, focusing on predefined fusion templates among memory-intensive operators or between memory- and compute-intensive operators to optimize resource utilization. TASO [23] automatically performs graph transformations based on a given set of tensor operators. DNNFusion [25] expands the possibilities of fusion by establishing a classification system of independent operators and their combinations. Other methods [20,41,42] focus on the fusion of computationally intensive operators.

Dynamic-shape Compiler. Recently, some studies have begun to explore dynamic shape compilers [19,27–29,43,44], which aim to support dynamic shapes by symbolizing shape information during the compilation phase and instantiating these symbols in the online phase. Nimble [29] addresses the dynamic nature of models by introducing a dynamic type system. BladeDisc [19] proposes a dynamic shape fusion method based on shape information propagation. However, these dynamic compilers perform optimizations at the computation graph level, treating each layer as a fixed pattern. Several studies have explored dynamic-shape compilers [45–49] from the perspective of shape composition kernel generation, refining the search space for shape generality.

Backend Tuning. Recent research has proposed several compilers [12,18,24,33,50] to automatically generate dedicated kernels for various hardware for single and fused operators. Specifically, TVM [18] uses a learning-based approach to predict scheduling costs and discover efficient scheduling schemes within a predefined scheduling space. Ansor [33] automatically generates scheduling templates and fine-tunes the best tiling information to generate high-performance kernels. AKG [12] utilizes auto-tuners or polyhedral models to generate optimal executable kernels for fixed shapes. Triton [50] provides an automatic kernel optimization and tuning framework that enables developers to generate high-performance kernels with minimal manual effort.

7. Conclusion

In this paper, we introduce S-GPT, a shape-aware graph partitioning and tuning framework for sustainable inference of large-scale DNNs under dynamic shape. By jointly modeling shape variability and tuning

tractability, S-GPT produces robust, tunable subgraphs and significantly reduces tuning search overhead. Experiments across LLM workloads show consistent end-to-end speedups and shorter tuning times than state-of-the-art compilers. Although S-GPT improves graph partitioning and kernel tuning across representative shape distributions, it still has several limitations. Maintaining multiple shape buckets introduces more subgraphs and compilation overhead, particularly when the shape space is highly fragmented. Future work will investigate adaptive bucket merging and splitting, as well as incremental tuning mechanisms, to improve its robustness under constantly changing shape distributions.

CRedit authorship contribution statement

Hongen Shao: Writing – original draft, Methodology, Conceptualization. **Cen Chen:** Writing – review & editing, Validation, Supervision, Funding acquisition. **Xin Fang:** Visualization, Validation, Software, Data curation. **Ziqian Zeng:** Writing – review & editing, Validation, Investigation. **Keqin Li:** Writing – review & editing, Validation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

- [1] H. Chen, H. Chen, M. Yan, W. Xu, G. Xing, W. Shen, X. Quan, C. Li, J. Zhang, F. Huang, SocialBench: Sociality evaluation of role-playing conversational agents, in: Findings of the Association for Computational Linguistics: ACL 2024, 2024, pp. 2108–2126.
- [2] Q. Tu, S. Fan, Z. Tian, T. Shen, S. Shang, X. Gao, R. Yan, CharacterEval: A chinese benchmark for role-playing conversational agent evaluation, in: Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2024, pp. 11836–11850.
- [3] S. Yuan, L.T. Yang, D. Liu, X. Wang, J. Yang, THPFF: A tensor-based high-precision feature fusion model for multi-source data in smart healthcare systems, Inf. Fusion 124 (2025) 103324.
- [4] P. Xu, S. Xiong, J. Zhang, Y. Chen, B. Zhou, C.C. Loy, D. Clifton, K.M. Lee, L. Van Gool, R. He, et al., Mars2 2025 challenge on multimodal reasoning: Datasets, methods, results, discussion, and outlook, in: Proceedings of the IEEE/CVF International Conference on Computer Vision, 2025, pp. 6517–6546.
- [5] J. He, J. Qiu, A. Zeng, Z. Yang, J. Zhai, J. Tang, Fastmoe: A fast mixture-of-expert training system, 2021, arXiv preprint arXiv:2103.13262.
- [6] Z. Zeng, J. Yu, Q. Pang, Z. Wang, H. Zhuang, F. Yu, H. Shao, X. Zou, Resdecode: accelerating large language models inference via residual decoding heads, Big Data Min. Anal. 8 (4) (2025) 779–793.
- [7] L. Zhao, M. Nybacka, M. Aramrattana, M. Rothhämel, A. Habibovic, L. Drügge, F. Jiang, Remote driving of road vehicles: A survey of driving feedback, latency, support control, and real applications, IEEE Trans. Intell. Veh. (2024).
- [8] Y. Ma, Y. Cao, J. Sun, M. Pavone, C. Xiao, Dolphins: Multimodal language model for driving, in: European Conference on Computer Vision, Springer, 2024, pp. 403–420.
- [9] X. Wang, D. Yuan, L. He, Z. Qiu, H. Huang, J. Pan, Y. Jia, A multiscale collaborative learning method on long-term time series for cyber-physical-social intelligence, IEEE Trans. Comput. Soc. Syst. (2026).
- [10] NVIDIA, cuBLAS: Basic linear algebra on NVIDIA GPUs, 2025, <https://developer.nvidia.com/cublas>.
- [11] NVIDIA, Cutlass: Cuda templates for linear algebra subroutines, 2025, <https://github.com/NVIDIA/cutlass>.
- [12] J. Zhao, B. Li, W. Nie, Z. Geng, R. Zhang, X. Gao, B. Cheng, C. Wu, Y. Cheng, Z. Li, 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, pp. 1233–1248.
- [13] X. Qin, Y. Li, F. Lin, W. Li, NLTSP: A cost model for tensor program tuning using nested loop trees, J. Syst. Archit. 158 (2025) 103305.
- [14] S. Zheng, S. Chen, S. Gao, L. Jia, G. Sun, R. Wang, Y. Liang, Tileflow: A framework for modeling fusion dataflow via tree-based analysis, in: Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, 2023, pp. 1271–1288.
- [15] R. Chen, Z. Ding, S. Zheng, C. Zhang, J. Leng, X. Liu, Y. Liang, Magis: Memory optimization via coordinated graph transformation and scheduling for dnn, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 3, 2024, pp. 607–621.
- [16] Z. Zheng, X. Yang, P. Zhao, G. Long, K. Zhu, F. Zhu, W. Zhao, X. Liu, J. Yang, J. Zhai, et al., Astitch: enabling a new multi-dimensional optimization space for memory-intensive ML training and inference on modern SIMD architectures, in: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, 2022, pp. 359–373.
- [17] J. Zhao, X. Gao, R. Xia, Z. Zhang, D. Chen, L. Chen, R. Zhang, Z. Geng, B. Cheng, X. Jin, Apollo: Automatic partition-based operator fusion through layer by layer optimization, Proc. Mach. Learn. Syst. 4 (2022) 1–19.
- [18] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, et al., {TVM}: An automated {End-to-End} optimizing compiler for deep learning, in: 13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 18, 2018, pp. 578–594.
- [19] Z. Zheng, Z. Pan, D. Wang, K. Zhu, W. Zhao, T. Guo, X. Qiu, M. Sun, J. Bai, F. Zhang, et al., Bladedisc: Optimizing dynamic shape machine learning workloads via compiler approach, Proc. ACM Manag. Data 1 (3) (2023) 1–29.
- [20] L. Xu, Z. Mo, Q. Wang, J. Jiang, N. Jing, Enabling multiple tensor-wise operator fusion for transformer models on spatial accelerators, in: Proceedings of the 61st ACM/IEEE Design Automation Conference, 2024, pp. 1–6.
- [21] S. Chen, S. Wei, C. Liu, W. Yang, Dycl: Dynamic neural network compilation via program rewriting and graph optimization, in: Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis, 2023, pp. 614–626.
- [22] M. Hu, A. Venkatram, S. Biswas, B. Marimuthu, B. Hou, G. Oliaro, H. Wang, L. Zheng, X. Miao, J. Zhai, et al., Optimal kernel orchestration for tensor programs with korch, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 3, 2024, pp. 755–769.
- [23] Z. Jia, O. Padon, J. Thomas, T. Warszawski, M. Zaharia, A. Aiken, TASO: optimizing deep learning computation with automatic generation of graph substitutions, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019, pp. 47–62.
- [24] L. Qiao, J. Shi, X. Hao, X. Fang, S. Zhang, M. Zhao, Z. Zhu, J. Chen, H. An, X. Tang, et al., Pruner: A draft-then-verify exploration mechanism to accelerate tensor program tuning, in: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, 2025, pp. 949–965.
- [25] W. Niu, J. Guan, Y. Wang, G. Agrawal, B. Ren, 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, pp. 883–898.
- [26] Z. Xu, H. Peng, W. Wang, AGO: Boosting mobile AI inference performance by removing constraints on graph optimization, in: IEEE INFOCOM 2023-IEEE Conference on Computer Communications, IEEE, 2023, pp. 1–10.
- [27] X. Luo, C. Zhang, C. Geng, Y. Yi, J. Hu, R. Zhang, Z. Zhang, G. Conso-laro, F. Yang, T. Lu, et al., TSCompiler: efficient compilation framework for dynamic-shape models, Sci. China Inf. Sci. 67 (10) (2024) 200403.
- [28] W. Niu, G. Agrawal, B. Ren, SoD2: Statically optimizing dynamic deep neural network execution, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 1, 2024, pp. 386–400.
- [29] H. Shen, J. Roesch, Z. Chen, W. Chen, Y. Wu, M. Li, V. Sharma, Z. Tatlock, Y. Wang, Nimble: Efficiently compiling dynamic neural networks for model inference, Proc. Mach. Learn. Syst. 3 (2021) 208–222.
- [30] R. Lai, J. Shao, S. Feng, S. Lyubomirsky, B. Hou, W. Lin, Z. Ye, H. Jin, Y. Jin, J. Liu, et al., Relax: composable abstractions for end-to-end dynamic machine learning, in: Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, 2025, pp. 998–1013.
- [31] J. Roesch, S. Lyubomirsky, M. Kirisame, L. Weber, J. Pollock, L. Vega, Z. Jiang, T. Chen, T. Moreau, Z. Tatlock, Relay: A high-level compiler for deep learning, 2019, arXiv preprint arXiv:1904.08368.
- [32] K. Zhu, W. Zhao, Z. Zheng, T. Guo, P. Zhao, J. Bai, J. Yang, X. Liu, L. Diao, W. Lin, DISC: A dynamic shape compiler for machine learning workloads, in: Proceedings of the 1st Workshop on Machine Learning and Systems, 2021, pp. 89–95.
- [33] L. Zheng, C. Jia, M. Sun, Z. Wu, C.H. Yu, A. Haj-Ali, Y. Wang, J. Yang, D. Zhuo, K. Sen, et al., Ansor: Generating {High-Performance} tensor programs for deep learning, in: 14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 20, 2020, pp. 863–879.
- [34] J.D.M.-W.C. Kenton, L.K. Toutanova, Bert: Pre-training of deep bidirectional transformers for language understanding, in: Proceedings of NaACL-HLT, vol. 1, Minneapolis, Minnesota, 2019, p. 2.
- [35] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, et al., An image is worth 16x16 words: Transformers for image recognition at scale, in: International Conference on Learning Representations, 2020.

- [36] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al., Llama 2: Open foundation and fine-tuned chat models, 2023, arXiv preprint [arXiv:2307.09288](https://arxiv.org/abs/2307.09288).
- [37] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 770–778.
- [38] J. Ansel, E. Yang, H. He, N. Gimselshein, A. Jain, M. Voznesensky, B. Bao, P. Bell, D. Berard, E. Burovski, et al., Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, 2024, pp. 929–947.
- [39] HUAWEI, Mindspore, 2025, <https://gitee.com/mindspore/mindspore>.
- [40] Z. Zheng, P. Zhao, G. Long, F. Zhu, K. Zhu, W. Zhao, L. Diao, J. Yang, W. Lin, Fusionstitching: boosting memory intensive computations for deep learning workloads, 2020, arXiv preprint [arXiv:2009.10924](https://arxiv.org/abs/2009.10924).
- [41] F. Wang, M. Shen, TileMap: Mapping multi-head attention on spatial accelerators with tile-based analysis, in: 2024 IEEE 42nd International Conference on Computer Design, ICCD, IEEE, 2024, pp. 653–660.
- [42] S.-C. Kao, S. Subramanian, G. Agrawal, A. Yazdanbakhsh, T. Krishna, Flat: An optimized dataflow for mitigating attention bottlenecks, in: Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, 2023, pp. 295–310.
- [43] R. Lai, J. Shao, S. Feng, S.S. Lyubomirsky, B. Hou, W. Lin, Z. Ye, H. Jin, Y. Jin, J. Liu, et al., Relax: Composable abstractions for end-to-end dynamic machine learning, 2023, arXiv preprint [arXiv:2311.02103](https://arxiv.org/abs/2311.02103).
- [44] P. Pegade, T. Chen, P. Gibbons, T. Mowry, ACROBAT: Optimizing auto-batching of dynamic deep learning at compile time, *Proc. Mach. Learn. Syst.* 6 (2024) 14–30.
- [45] P. Mu, Y. Liu, R. Wang, G. Liu, Z. Sun, H. Yang, Z. Luan, D. Qian, Haotuner: A hardware adaptive operator auto-tuner for dynamic shape tensor compilers, *IEEE Trans. Comput.* 72 (11) (2023) 3178–3190.
- [46] B. Zheng, Z. Jiang, C.H. Yu, H. Shen, J. Fromm, Y. Liu, Y. Wang, L. Ceze, T. Chen, G. Pekhimenko, DietCode: Automatic optimization for dynamic tensor programs, *Proc. Mach. Learn. Syst.* 4 (2022) 848–863.
- [47] F. Yu, G. Li, J. Zhao, H. Cui, X. Feng, J. Xue, Optimizing dynamic-shape neural networks on accelerators via on-the-fly micro-kernel polymerization, in: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, vol. 2, 2024, pp. 797–812.
- [48] P. Mu, L. Wei, Y. Liu, R. Wang, FTuner: A fast dynamic shape tensors program auto-tuner for deep learning compilers, 2024, arXiv preprint [arXiv:2407.21418](https://arxiv.org/abs/2407.21418).
- [49] Y. Zhou, H. Zhu, Q. Qiu, W. Cui, Z. Liu, C. Guo, S. Feng, J. Meng, H. Lan, J. Leng, et al., Vortex: Efficient sample-free dynamic tensor program optimization via hardware-aware strategy space hierarchization, 2024, arXiv preprint [arXiv:2409.01075](https://arxiv.org/abs/2409.01075).
- [50] P. Tillett, H.-T. Kung, D. Cox, 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, 2019, pp. 10–19.



Hongen Shao received his bachelor's and master's degrees in software engineering from Central South University of Forestry and Technology, China, in 2020 and 2023. He is currently a PhD student in the School of Future Technology, South China University of Technology, Guangzhou, China. His main research fields include artificial intelligence algorithms and computer architecture design, which also includes compilation optimization between deep learning training inference frameworks and hardware.



Cen Chen (Senior Member, IEEE) received his PhD degree in Computer Science, Hunan University, China. Currently, he is a professor in the School of Future Technology at the South China University of Technology in China. His research interests include parallel and distributed computing, machine learning, and deep learning. He has published several research articles in international conferences and journals, such as HPCA, DAC, AAAI, IEEE-TC, IEEE-TPDS, IEEE-TNNLS, IEEE-TCYB, and many more. His research interests include parallel and distributed computing, machine learning, and deep learning.



Xin Fang received his bachelor's degree in software engineering from Jiangxi Agricultural University, China, in 2019 and his master's degree in computer technology from Fuzhou University, China, in 2025. He is currently a PhD student in the School of Future Technology, South China University of Technology, Guangzhou, China. His main research fields include artificial intelligence algorithms and computer architecture design, which also includes convolution algorithms and compilation optimization, between deep learning training inference framework and hardware.



Ziqian Zeng received the PhD degree in computer science and engineering from The Hong Kong University of Science and Technology in 2021. She is currently an associate professor at Shien Ming Wu School of Intelligent Engineering, South China University of Technology, China. Her main research interest is efficient inference, privacy protection, and safety of Large Language Models.



Keqin Li received a B.S. degree in computer science from Tsinghua University in 1985 and a Ph.D. degree in computer science from the University of Houston in 1990. He is a SUNY Distinguished Professor with the State University of New York and a National Distinguished Professor with Hunan University (China). He has authored or co-authored more than 1080 journal articles, book chapters, and refereed conference papers. He received the Distinguished Alumnus Award from the Computer Science Department at the University of Houston in 2018. He received the IEEE TCCLD Research Impact Award from the IEEE CS Technical Committee on Cloud Computing in 2022 and the IEEE TCSVC Research Innovation Award from the IEEE CS Technical Community on Services Computing in 2023. He won the IEEE Region 1 Technological Innovation Award (Academic) in 2023. He is a Member of the SUNY Distinguished Academy. He is an AAAS Fellow, an IEEE Fellow, an AAIA Fellow, an ACIS Fellow, and an AIIA Fellow. He is a Member of the European Academy of Sciences and Arts. He is a Member of Academia Europaea (Academician of the Academy of Europe).