Enabling Tensor Language Model to Assist in Generating High-Performance Tensor Programs for Deep Learning

Yi Zhai, *University of Science and Technology of China;* Sijia Yang, *Huawei Technologies Co., Ltd.;* Keyu Pan, *ByteDance Ltd.;* Renwei Zhang, *Huawei Technologies Co., Ltd.;* Shuo Liu, *University of Science and Technology of China;* Chao Liu and Zichun Ye, *Huawei Technologies Co., Ltd.;* Jianmin Ji, *University of Science and Technology of China;* Jie Zhao, *Hunan University;* Yu Zhang and Yanyong Zhang, *University of Science and Technology of China*

https://www.usenix.org/conference/osdi24/presentation/zhai

This paper is included in the Proceedings of the 18th USENIX Symposium on Operating Systems Design and Implementation.

July 10–12, 2024 • Santa Clara, CA, USA

978-1-939133-40-3

ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE

ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

ARTIFACT EVALUATED usenix ASSOCIATION REPRODUCED

Enabling Tensor Language Model to Assist in Generating High-Performance Tensor Programs for Deep Learning

Yi Zhai[1] Sijia Yang[2] Keyu Pan[3] Renwei Zhang[2]
Shuo Liu[1] Chao Liu[2] Zichun Ye[2]
Jianmin Ji[1] Jie Zhao[4] Yu Zhang[1*] Yanyong Zhang[1*]

[1]*University of Science and Technology of China*
[2]*Huawei Technologies Co., Ltd.* [3]*ByteDance Ltd.* [4]*Hunan University*

Abstract

Obtaining high-performance tensor programs with high efficiency continues to be a substantial challenge. Approaches that favor efficiency typically limit their exploration space through heuristic constraints, which often lack generalizability. Conversely, approaches targeting high performance tend to create an expansive exploration space but employ ineffective exploration strategies.

We propose a tensor program generation framework for deep learning applications. Its core idea involves maintaining an expansive space to ensure high performance while performing powerful exploration with the help of language models to generate tensor programs efficiently. We thus transform the tensor program exploration task into a language model generation task. To facilitate this, we explicitly design the language model-friendly tensor language that records decision information to represent tensor programs. During the compilation of target workloads, the tensor language model (TLM) combines knowledge from offline learning and previously made decisions to **probabilistically sample** the best decision in the current decision space. This approach allows more informed space exploration than **random sampling** commonly used in previously proposed approaches.

Experimental results indicate that TLM excels in delivering both efficiency and performance. Compared to fully tuned Ansor/MetaSchedule, TLM matches their performance with a compilation speedup of 61×. Furthermore, when evaluated against Roller, with the same compilation time, TLM improves the performance by 2.25×. Code available at https://github.com/zhaiyi000/tlm.

1 Introduction

As deep learning rapidly grows in both scale and complexity, the gap between the computational needs of deep learning workloads and the capabilities of existing computing platforms is widening. This gap underscores the imperative for low-latency execution of deep learning workloads. Solutions for low-latency execution primarily include kernel libraries provided by vendors (e.g., cuDNN [1], oneDNN [2]) and search-based tensor compilers (e.g., TVM [3], Halide [4], Tensor comprehensions [5], Flextensor [6], NeoCPU [7]). Existing deep learning frameworks (e.g., TensorFlow [8], PyTorch [9], and MXNet [10]) map the operators (e.g., convolution, matrix multiplication) in deep learning workloads to vendor-provided kernel libraries to optimize performance. Given the high development costs of vendor-provided kernel libraries, developers are increasingly turning to tensor compilers to auto-explore for tensor programs, i.e., optimized, low-level implementations of operators.

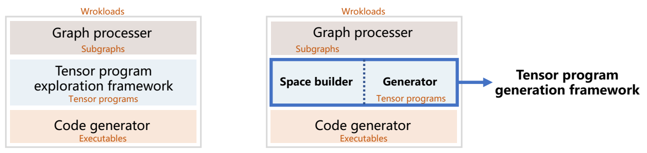

Figure 1: The left shows the architecture of common tensor compilers, while the right illustrates the structure incorporating our tensor program generation framework.

The left of Figure 1 shows that a tensor compiler can be divided into three parts: a graph processor (e.g., Relay [11], HLO), a tensor program exploration framework (e.g., AutoTVM [12], Ansor [13], MetaSchedule [14], AKG [15]), and a code generator (e.g., LLVM [16], NVCC [17]). The graph processor is responsible for converting various deep learning workloads of different formats (e.g., ONNX [18], TensorFlow PB) into a unified graph representation, performing graph optimizations, and splitting the workloads into subgraphs. Then, the tensor program exploration framework takes subgraphs as inputs, which, after scheduling, are lowered into tensor programs. Finally, a code generator is invoked to produce executables explicitly tailored for the target hardware.

The tensor program exploration framework, in lowering subgraphs to tensor programs, is tasked with making a series of critical decisions. These include determining the tiling

*Corresponding authors.

sizes of loop axes, setting unroll steps, choosing computation locations for operators, strategizing on parallelization and vectorization, and, particularly on GPUs, deciding thread bindings. The options for each decision create a decision space, and all decision combinations form the exploration space. The primary objective of this exploration framework is to find a decision combination that minimizes the execution latency of the resulting tensor program.

Previous work either introduces heuristic constraints to define or prune the exploration space, resulting in limited performance or constructs an expansive exploration space but employs less effective exploration strategies – for instance, the random sampling used by Ansor [13] and MetaSchedule [14]. For a detailed literature review, please refer to section §2.1.

To address this challenge, we propose an approach to maintain an expansive exploration space while conducting more powerful exploration – i.e., enabling language models to assist in generating high-performance tensor programs. Specifically, we present a *generation-based* tensor program exploration framework, i.e., a tensor program generation framework, for deep learning applications. The proposed framework consists of two components, a space builder and a generator, as shown in the right of Figure 1. The space builder is dedicated to building an expansive tensor program exploration space, ensuring high performance; meanwhile, the generator focuses on efficiently generating high-performance tensor programs, regardless of the exploration space's magnitude. They function independently without mutual constraints, each focusing solely on ensuring high performance and high efficiency.

Generating tensor program source code with language models naturally presents challenges. The length of tensor program source codes often exceeds ten thousand tokens, and these programs must adhere to strict syntactic rules. Crafting such lengthy and syntactically valid tensor program source codes is nearly unfeasible. Therefore, instead of aiming for direct end-to-end generation source code, we utilize tensor compilers for constructing tensor programs and leverage language models to assist in decision-making. To facilitate this, we explicitly design the language model-friendly tensor language to represent tensor programs. A tensor language sentence (abbreviated as tensor sentence) uniquely corresponds to a tensor program by recording the input subgraph, hardware specifications, and decision information of the tensor program. Compared to tensor program source code, a tensor sentence conveys the same semantics (i.e., both represent a tensor program) but in a far more concise manner, capped at no more than 1024 tokens. Building on the tensor language, we draw on the training methods of ChatGPT [19] to develop a language model, the tensor language model (TLM). We utilize millions of tensor sentences and a select few demonstration sentences (corresponding to high-performance tensor programs) to pre-train and fine-tune TLM in a supervised manner. After that, during the compilation of target workloads, TLM combines knowledge from offline learning and previous decisions to make probabilistic predictions for the current decision space, resulting in more effective exploration than random sampling.

用 TC 生成, 再用 LM 决策

It is noteworthy that, in contrast to methods like Ansor/MetaSchedule that depend exclusively on online data, TLM requires about 300K pieces of offline labeled data (i.e., tensor programs with measured execution latency) to select demonstration sentences, requiring tens of hours to collect. However, the labeled data for TLM is still significantly less than that for TenSet [20] or TLP [21], which amounts to 8.6 million and requires several weeks to collect.

In this paper, we also refer to our proposed tensor program generation framework as the TLM framework. The primary innovations of the TLM framework focus on the tensor language and tensor language model. The supported decision spaces are adapted from previous frameworks, which is largely an engineering effort. The space builder currently supports decision spaces adapted from several previous search frameworks, including Ansor, MetaSchedule, AKG, and AKG-MLIR.

We conducted extensive experiments to validate the high efficiency and performance of the TLM framework, examining scenarios with various exploration budget points. Under a limited budget, TLM's performance matches that of Ansor/MetaSchedule yet compiles 61× faster. While its compilation time aligns with Roller, its performance is 2.25× better. In ample exploration times, TLM's compilation duration is consistent with Ansor and MetaSchedule, delivering a performance boost of 1.08× and 1.04×, respectively.

In summary, this paper makes the following contributions:

- We design the language model-friendly tensor language to represent tensor programs, bridging the gap between tensor programs and language models.
- We develop a tensor language model that combines knowledge from offline learning and previously made decisions to probabilistically sample the best decision in the current decision space, enabling more effective space exploration.
- Experimental results show that TLM excels in delivering both high efficiency and performance.

2 Background

2.1 Tensor Program Exploration Framework

In the development of tensor program exploration frameworks, previous studies mainly employ search algorithms to locate optimal tensor programs automatically. As a result, the search-based tensor program exploration framework often gets dubbed as the tensor program search framework, with its exploration space used as the search space.

Earlier, the Halide auto-scheduler [4] aggressively prunes the search space by evaluating incomplete programs; AutoTVM and FlexTensor [6] employ predefined, manually written templates to define their search space. These methodologies introduce constraints that limit the search space, thereby missing out on many potent decision combinations and leading to suboptimal performance.

Subsequently, Ansor [13] and MetaSchedule [14] utilize derivation rules to build an expansive search space, pushing performance to state-of-the-art levels. However, this came with a notably vexing issue — excessive compilation time. In our experience, using Ansor to bring a BERT-base workload to convergence takes 21.6 hours on the NVIDIA V100 GPU; on the Intel i7-10510U CPU, it requires 13.1 hours. The reason behind this is that Ansor/MetaSchedule utilizes a strategy of random sampling followed by evaluation using a learnable cost model. Until completing the random sampling of all decision spaces, Ansor/MetaSchedule lowers subgraphs to tensor programs and then extracts statistical features, including computation, memory access, and arithmetic strength for performance evaluation with the learnable cost model. However, this strategy has three main issues:

- Random sampling represents a form of inefficient exploration, with equal probabilities of sampling optimal or suboptimal decisions.
- The evaluation demonstrates hysteresis. Decisions made during the initial sampling might lead to poor performance but can only be ascertained when evaluating.
- The inadequacy of training data and the small parameter size of the cost model weaken the model's learning ability, limiting its effectiveness in guiding the search algorithm. Ansor/MetaSchedule relies exclusively on minimal online data to train cost models. Moreover, their models are primarily based on low-parameter machine learning or deep learning models (e.g., XGBoost [22], MLP, LSTM).

More recent studies, Roller [23], TenSet [20], and TLP [21], were proposed to address the issue of slow compilation. Roller speeds up the compilation by aligning tensor shapes with the properties of the hardware. However, in doing so, Roller experiences a declining performance (§6.4.2), still due to the generalizability of the heuristic constraint. TenSet and TLP collect offline datasets before compiling the target workloads to build a stronger cost model. While these approaches expedite compilation, they exhibit two key shortcomings. Firstly, they adopt the Ansor/MetaSchedule strategy of conducting random sampling followed by evaluating with a cost model. Moreover, they rely on 8.6 million pieces of labeled data.

An example. We illustrate an example to analyze the differences between heuristic compilers, exemplified by Roller, search-based frameworks like Ansor/MetaSchedule, and our generation-based TLM framework from a probabilistic perspective. Consider a decision space, D_i, with four valid decisions. The optimal decision is d_3, as depicted in Figure 2.

Heuristic compilers use heuristic constraints to eliminate d_1, d_2, and d_4. After this pruning, d_3 has a 100% chance of being sampled, reducing the search space and speeding up the

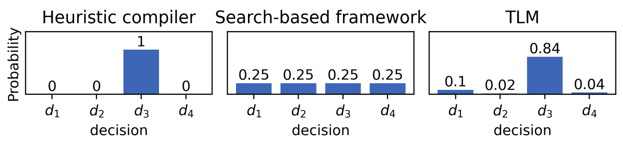

Figure 2: Example of probability distributions in a decision space for several different methods.

compilation. However, such heuristics aren't always accurate and can sometimes degrade performance. In contrast, search-based frameworks keep all decision options and sample one decision randomly. Here, d_3 has a 25% sampling probability. Until all decision spaces are sampled, a learnable cost model evaluates the results. After sufficient searching, search-based frameworks can always approximate the optimal solution, making them performance-oriented but inefficient methods. Unlike search-based frameworks, TLM combines the knowledge learned offline with the decisions already made to make probabilistic predictions about the best decision in the current decision space.

2.2 Language Model

Deep learning language models mainly fall into two categories: the Masked Language Model (MLM) and the Causal Language Model (CLM), with CLM also known as the Autoregressive Language Model. Notable examples of CLM include the GPT series, such as GPT-2 [24] and GPT-3 [25]. CLMs, recognized for their natural text generation method, often excel in tasks that need coherent text creation, like writing or chatbot conversations. In this paper, a language model refers specifically to a CLM.

Before training a language model, a vocabulary is created through tokenization. Tokenization typically refers to fragmenting an input sentence into its constituent tokens for subsequent language analysis or as input to a model, with all such tokens collectively forming a vocabulary. The steps ①② and ⑦⑧ in Figure 3 represent the processes of tokenizing a sentence into tokens and converting tokens back into a sentence, respectively. For simplicity in description, here we tokenize by words (in practice, the process is more complex). The only thing language models perform is to combine the learned knowledge with the given input to predict the probability distribution of the next token being a specific one from the vocabulary. The initial input is referred to as a prompt. During the training of a language model, natural language sentences are utilized as input. Through the backpropagation algorithm [26], the language model's parameters are iteratively updated to maximize the probability that the next token generated aligns with the corresponding token within the natural language.

Employing the language model for inference, take a real-life instance as an example: we ask a language model, "What

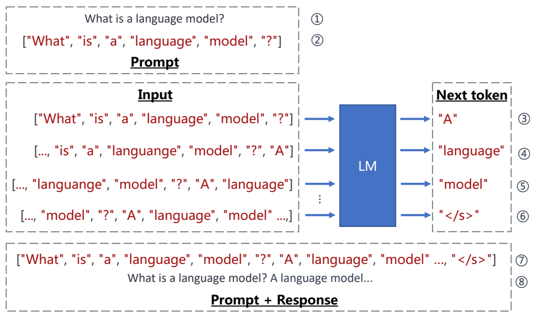
What is a language model?
①
["What", "is", "a", "language", "model", "?"]
②
Prompt
Input
["What", "is", "a", "language", "model", "?"]
[..., "is", "a", "languange", "model", "?", "A"]
[..., "languange", "model", "?", "A", "language"]
[..., "model", "?", "A", "language", "model" ...,]
LM
Next token
"A"
③
"language"
④
"model"
⑤
"</s>"
⑥
["What", "is", "a", "language", "model", "?", "A", "language", "model" ..., "</s>"]
⑦
What is a language model? A language model...
⑧
Prompt + Response

Figure 3: Employing a language model to generate a natural language sentence.

is a language model?" and it responds, "A language model is a type of artificial intelligence that is trained to understand, generate, and respond to human language...". The generation process is illustrated in Figure 3. In step ③, the language model uses the prompt as input to predict a probability for each token in the vocabulary. This probability signifies the likelihood of each token logically following the input. The next token is then sampled based on these probabilities. Step ④ involves using the prompt and the token generated in step ③ as input to sample the next token probabilistically, continuing this process until the "</s>" token is generated. The ending "</s>" signifies the end of a sentence.

2.3 ChatGPT/InstructGPT

ChatGPT [19] and InstructGPT [27] employ a similar training methodology that encompasses: pre-training, SFT, and RLHF (RM + RL). TLM partially adopts these training approaches. The training process for InstructGPT includes:
Pre-training GPT-3. InstructGPT is based on GPT-3, which boasts 175 billion parameters, leverages approximately 45TB of Internet text for training, and necessitates a significant amount of hardware resources throughout its training process, with the aim of enabling GPT-3 to learn the fundamental structures and semantics of language.
Supervised fine-tuning (SFT). Training a supervised policy by collecting demonstration data (around 14K entries) to fine-tune GPT-3 through supervised learning, with the demonstration data provided by humans representing the desired output behavior corresponding to a prompt.
Reward modeling (RM). Training a reward model by collecting comparison data (around 51K entries), wherein the comparison data, also provided by humans, offers a ranking of model outputs from best to worst.
Reinforcement learning (RL). Optimizing a policy against the reward model using the PPO [28] reinforcement learning algorithm, with rewards determined by the preceding reward model, this step will be iterated multiple times.

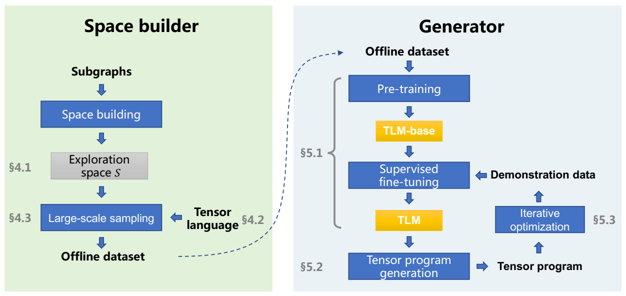

Figure 4: System overview of the TLM framework.

3 System Overview

The TLM framework is a tensor program generation framework designed to generate high-performance tensor programs efficiently. Maintaining a large exploration space can ensure that high-performance tensor programs are not eliminated by heuristic constraints. However, this requires stronger exploration capabilities. Utilizing deep learning models to glean knowledge from offline data for aiding online tensor program exploration presents a promising strategy. Given the current learning capabilities, language models are the most powerful method available. Hence, we propose to leverage the language model to assist in generating high-performance tensor programs, transforming the tensor program exploration task into a language model generation task.

Figure 4 shows the system overview and marks the corresponding subsections that detail each system component. Section 4 centers on collecting a large-scale offline dataset necessary for the pre-training of TLM. Section 4.1 details the exploration space for data sampling and its build process, providing a theoretical foundation for tensor language design through formalization of the exploration space. Section 4.2 discusses tensor language design, emphasizing its role in preserving tensor programs sampled from the exploration space in a format more amenable to language models. Following the discussion of the sampling space and preservation methods, Section 4.3 introduces large-scale sampling, where extensive random sampling achieves an unbiased estimation of the exploration space.

Section 5 focuses on the development of a tensor language model. Initially, Section 5.1 addresses the model architecture, training methods, and the training process of TLM, encompassing both pre-training and supervised fine-tuning. Upon completing pre-training, TLM should be able to generate any valid tensor program within the exploration space. Section 5.2 then details how the TLM framework generates tensor programs with decision-making support from TLM. For generating high-performance tensor programs, we employ demonstration data (corresponding to high-performance tensor programs) to fine-tune TLM through supervised learning, aligning the tensor sentences it generates with our anticipated

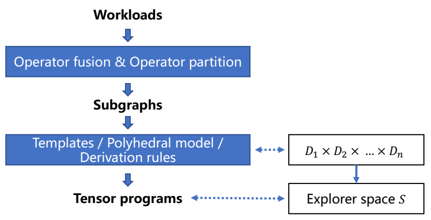
Workloads
Operator fusion & Operator partition
Subgraphs
Templates / Polyhedral model / Derivation rules
$D_1 \times D_2 \times \ldots \times D_n$
Tensor programs
Explorer space S

Figure 5: Two intermediate representations of common tensor compilers.

demonstration sentences. Section 5.3 discusses obtaining this demonstration data via iterative algorithms.

4 Space Builder

The function of a space builder is to create a large exploration space to ensure high performance. This exploration space consists of all possible tensor programs. Regarding the idea that a large exploration space can guarantee high performance, two aspects need to be discussed. First, a larger space implies that, in comparison to a smaller one, it is constructed with fewer constraints, thus enabling the generation of more tensor programs. To some extent, it can be said that the large space includes the smaller space. Second, ensuring high performance means maintaining the potential for high-performance tensor programs to exist within the exploration space. The larger the space, the less likely it is to be pruned by constraints, but this also demands greater exploration capabilities.

4.1 Exploration Space

In this section, we introduce two primary topics. The first is about building the exploration space. The second involves formalizing the tensor program generation process, providing a theoretical basis for designing the tensor language.

Tensor compilers customarily extract two intermediate representations (IRs) for optimization purposes, i.e., the graph layer and the tensor layer, which are specifically tailored for hardware-independent and hardware-dependent enhancements, respectively. As depicted in Figure 5, the graph layer ingests a workload, optimizing with several passes and employing algorithms for operator fusion and partitioning. Then, it produces subgraphs, each consisting of one or more operators. A subgraph describes the expected computational results, whereas the tensor program exploration framework transforms this subgraph into a tensor program, providing a detailed computational implementation. The collective of all possible tensor programs forms the exploration space S of the tensor layer. Previous exploration frameworks utilize expert knowledge, including templates (AutoTVM), the polyhedral model (AKG), and derivation rules (Ansor, MetaSchedule), to map subgraphs into tensor programs. For areas where expert knowledge falls short, these frameworks use tunable parameters (AutoTVM, AKG), annotations (Ansor), or random variables (MetaSchedule) to create a decision space D_i and then employ search algorithms, such as simulated annealing [29] and genetic algorithm [30], to locate the optimal solution.

The decision spaces chiefly involve determining tiling sizes for loop axes, setting unroll steps, selecting computation locations for operators, strategizing parallelization and vectorization, and on GPUs, specifically determining thread bindings.

From the analysis above, tensor programs result from optimizing input subgraphs through a series of decisions. We formalize the tensor layer's exploration space as follows:

$$S = \left\{ s^{(n)} \;\middle|\; \begin{array}{l} s^{(i)} = \text{apply}\left(s^{(i-1)}, d_i\right), \\ \forall d_i \in D_i,\ 1 \le i \le n \end{array} \right\}$$

where $s^{(0)}$ denotes the initial program of the input subgraph, and d_i represents a random sample from the set D_i. Thus, the size of the exploration space aligns with the number of decision combinations. We have:

$$|S| = |D_1| \times |D_2| \times \ldots \times |D_n|.$$

This work discusses only the exploration space of the tensor layer. On a CPU, the exploration space size corresponding to a subgraph is roughly 10^6; on a GPU, it approximates 10^9.

We reuse the decision space from previous work (e.g., Ansor, MetaSchedule), as we think that this space is already sufficiently large to yield high-performance tensor programs. Expanding the exploration space of tensor layers to larger is a considerable challenge. In future endeavors, we advocate exploring larger spaces mainly by integrating the decision space of the graph layer. Certainly, the TLM framework supports exploration spaces of varying sizes since they all integrate with TLM in the same manner.

4.2 Tensor Language

In this section, we focus on how to store tensor programs sampled from the exploration space.

Recall that our objective is to teach language models of these tensor programs, enabling them to aid in generating high-performance tensor programs during target workload compilation. Tensor program source codes often exceed ten thousand tokens, posing a challenge for language models to generate such lengthy, coherent, and valid source codes. Therefore, we do not pursue the end-to-end generation of tensor program source codes. Instead, we utilize language models to assist in decision-making. To facilitate this, we explicitly design the language model-friendly tensor language that records decision information to represent tensor programs.

As understood from Section 4.1, a tensor program $s^{(n)}$ pertains exclusively to the initial program $s^{(0)}$ of the input subgraph and decisions $d_1, d_2, \ldots, d_n$. Each decision d_i is randomly sampled from decision space D_i, the design of which

Algorithm 1: Sampling tensor sentences from decision spaces.

```
Func GenerateSampleData(subgraph, hardware):
    tokens = []
    ExtractTokensFromSubgraph(subgraph, tokens)
    ExtractTokensFromHardware(hardware, tokens)
    decision_spaces = DetermineDecisionSpaces(subgraph,
      hardware)
    foreach space in decision_spaces do
        switch space.type do
            case "tile_size" do
                HandleTileSizeSpace(space, tokens)
            case "unroll" do
                HandleParallelSpace(space, tokens)
            // Additional space types
            case ... do
                ...
    return tokens
Func HandleTileSizeSpace(space, tokens):
    tokens.append("split")
    tokens.extend(Serialize(space.operator))
    tokens.extend(Serialize(space.axis))
    tiles = RandomSample(space)
    tokens.extend(Serialize(tiles))
    // Other properties
```

hinges on the hardware platform. Consequently, to ensure that a tensor language sentence uniquely corresponds to a tensor program, a tensor sentence must encapsulate the input subgraph, hardware specifications, and decisions. We utilize Algorithm 1 to sample data from the exploration spaces. Each sentence extracts information from the input subgraph, encompassing the type and shape information of each operator within the subgraph. Furthermore, it retrieves hardware details, including the number of processing cores and the supported vector instructions. Successively extracting information from each decision space, Algorithm 1 demonstrates how to extract tokens from the tile size decision space, conserving details such as the corresponding operator, axis, decision space type, sampled decisions from the space, and other pivotal information. A similar method is applied to other decision spaces as well.

Tensor language is a form of natural language, not a programming language, and thus does not strictly follow the Backus-Naur Form [31]. Its primary intent is to represent a tensor program using a single sentence, emphasizing its role in recording rather than programming. This recording process offers significant **flexibility**, with only one constraint being the **consistency** between tensor sentences collected offline and generated online. Such consistency is essential for deep learning models to ensure that training and testing data are independently and identically distributed.

Given the flexibility of tensor language, there are no strict guidelines on the exact details that need to be recorded about input subgraphs, hardware specifications, and decision infor-

p0 p1 T_matmul_NT p2 T_add 00a059b856ac30ac172b6252254479a6 1024 1024 512 1024 1024 512 1024 512 ***llvm -keys=cpu -mcpu=core-avx2 -model=i7 4 64 64 0 0 0 0 0 2*** $\boxed{\text{SP 2 0 1024 32 1 4 1}}$ SP 2 4 512 8 1 4 1 SP 2 8 1024 1024 1 RE 2 0 4 1 5 8 2 6 9 3 7 FSP 4 0 0 2 FSP 4 3 1 2 RE 4 0 3 1 4 2 5 CA 2 4 3 PPT SPC 2 0 1024 32 1 4 1 SPC 2 4 512 8 1 4 1 SPC 2 8 1024 1024 1 CLS FU 4 0 1 2 3 AN 4 0 3 PRS 2 PR 2 0 auto_unroll_max_step$0 VECS

fused nn dense add fast tanh float32 4 512 float32 512 512 float32 1 512 float32 4 512 ***llvm -keys=cpu -mcpu=core-avx2 -model=i7 -num-cores=4*** GetBlock T_matmul_NT main b0 GetBlock T_add main b1 GetBlock T_minimum main b2 GetBlock T_maximum main b3 GetBlock root main b4 ComputeInline b3 ComputeInline b2 ComputeInline b1 Annotate b0 \"SSRSRS\" meta_schedule.tiling_structure GetLoops b0 l5 l6 l7 SamplePerfectTile l5 4 64 v8 v9 v10 v11 Split l5 v8 v9 v10 v11 1 l12 l13 l14 l15 SamplePerfectTile l6 4 64 v16 v17 v18 v19 Split l6 v16 v17 v18 v19 1 l20 l21 l22 l23 SamplePerfectTile l7 2 64 v24 v25 Split l7 v24 v25 1 l26 l27 Reorder l12 l20 l13 l21 l26 l14 l22 l27 l15 l23 GetConsumers b0 b28 ReverseComputeAt b28 l20 1 -1 Annotate b4 1 meta_schedule.parallel Annotate b4 64 meta_schedule.vectorize SampleCategorical 0 16 64 512 0.25 0.25 0.25 0.25 v29 Annotate b4 v29 meta_schedule.unroll_explicit EnterPostproc 10 1 1 1 1 12 1 1 1 1 14 1 1 21 1 PPT 10 1 1 4 1 12 8 4 2 8 14 512 1 21 0 Annotate b4 2 meta_schedule.parallel

Figure 6: Tensor sentence samples tailored for Ansor (top) and MetaSchedule (bottom), encompassing **input subgraph**, ***hardware specifications***, and decision information. For example, at the top, "SP 2 0 1024 32 1 4 1" represents "split operator_index axis_index axis_extent tile_size_0 tile_size_1 tile_size_2 save_manner".

mation. Under the premise of maintaining consistency, these details can be dynamically adjusted in conjunction with specific engineering projects. Figure 6 showcases tensor sentence samples tailored for Ansor (top) and MetaSchedule (bottom).

4.3 Large-scale Sampling Tensor Sentences

This section introduces employing the sampling algorithm to create a large-scale offline dataset.

Large-scale sampling serves two main purposes. Firstly, it is to create an unbiased estimation of the exploration space S through widespread random sampling, which allows TLM to learn the basic structures and semantics of tensor language, enabling TLM to generate any tensor sentence within space S. Secondly, it is to build as large a vocabulary (§2.2) as possible, where all decision options like "i.0=16" and "i.0=32" are tokenized into discrete tokens. If a decision, such as "i.0=17", is not in the vocabulary, it will never be generated.

The workload dataset configured for TLM takes cues from TenSet. The workloads are derived from PyTorch's Vision Model Zoo and Huggingface's Transformer Model Zoo, encompassing tasks emblematic of both computer vision (CV) and natural language process (NLP). We adjust the input shape to generate a variety of subgraphs. Note that we focus on small batch sizes in this dataset because tensor compilers are mainly used for optimizing trained models for inference. Altogether, the dataset consists of 138 workloads, with 12 held out for testing; from the remaining approximately 3K subgraphs are extracted.

We collect 2 million tensor sentences for 3K subgraphs to pre-train TLM. It is worth noting that the 2 million data entries are distinct from the 8.6 million used in TenSet/TLP. The data here is unlabelled, i.e., it does not require measuring execution latency; measurement is typically the most time-consuming part. On a 96-core server, it takes about 2 hours to collect 2 million CPU data entries, and around 10 hours for the GPU. This longer duration for GPU data is due to additional checks, such as ensuring thread binding meets hardware constraints.

5 Tensor Language Model

5.1 Model Details

Model architecture. Taking into account both learning capacity and resource overhead, TLM adopts the architecture of GPT-2 Small, which encompasses approximately 100 million parameters. TLM is composed of 12 Transformer layers, each featuring 12 attention heads and 768 hidden units.

Training methodology. The TLM training is partially inspired by the training procedure utilized by ChatGPT, as introduced in §2.3. To put it succinctly, we modify the formula from $1 \times \text{pre-training} + 1 \times \text{SFT} + n \times (\text{RM} + \text{RL})$ to $1 \times \text{pre-training} + n \times (\text{measurement} + \text{SFT})$.

We substitute reward model (RM) with measurement. RM allocates a reward value to the model output and evaluates its quality. Here, tensor language has a natural advantage. A tensor sentence can be converted into a tensor program (§5.2), allowing its execution latency to be directly measured on hardware, with a lower latency suggesting a better tensor sentence. While ChatGPT only performs supervised fine-tuning (SFT) once, we conduct it multiple times. The input for SFT is demonstration data, provided by humans, symbolizing the desired output behavior corresponding to a prompt. In the NLP field, it's hard to say which demonstration data is the "best" for a prompt, but this is achievable in tensor language. That is, the sentence with the lowest execution latency is the "best" for its prompt. Performing measurement and SFT multiple times is consistently seeking the best tensor sentences (§5.3). We abstained from using reinforcement learning (RL) mainly because it requires adjusting various hyperparameters and presents a training challenge to convergence, and we find the performance was sufficiently good after performing SFT multiple times.

Training details. The model resulting from pre-training is termed TLM-base, and TLM is derived by performing SFT on TLM-base. We pre-train TLM-base using the offline dataset gathered in §4.3 in the same manner as pre-training other language models. It is noteworthy that TLM-base converges within just 2 epochs. This quicker convergence is due to the more pronounced regularity in tensor language compared to the broad diversity in natural languages. Among the 2 million data entries, the input subgraph types, hardware types, and decision types are all limited. Pre-training TLM-base for 2 epochs takes about 10 hours using 4 NVIDIA V100s.

Following its pre-training, TLM-base possesses the ability to generate valid tensor sentences. Furthermore, for the input subgraphs, we expect that TLM-base can aid in generating high-performance tensor programs. To this end, we employ demonstration data to fine-tune TLM-base through supervised learning. Demonstration data refers to the tensor sentences corresponding to a small subset of tensor programs with the lowest execution latency for a given input subgraph. **The purpose of SFT of a language model with demonstration data is to achieve that the model's responses to prompts align with the human intentions reflected in the demonstration data.** Similarly, we apply demonstration data to TLM, aiming to empower it to generate high-performance outputs, in response to prompts. Performing SFT on TLM-base requires a small batch (about 3K, selecting the best one for each subgraph) of demonstration data. We employ iterative optimization (§5.3) to continuously optimize these demonstration entries. With 3K demonstration data, using 4 NVIDIA V100s to perform SFT on TLM-base once takes approximately 10 minutes.

Algorithm 2: Generating tensor programs aided by TLM in decision-making.

```
Func GenerateTensorProgram(subgraph, hardware):
    tokens = []
    ExtractTokensFromSubgraph(subgraph, tokens)
    ExtractTokensFromHardware(hardware, tokens)
    program = GetInitProgram(subgraph, hardware)
    decision_spaces = DetermineDecisionSpaces(subgraph,
      hardware)
    foreach space in decision_spaces do
        switch space.type do
            case "tile_size" do
                ApplyTileSize(space, tokens, program)
            case "unroll" do
                ApplyParallel(space, tokens, program)
            // Additional space types
            case ... do
                ...
    return program
Func ApplyTileSize(space, tokens, program):
    tokens.append("split")
    tokens.extend(Serialize(space.operator))
    tokens.extend(Serialize(space.axis))
    response_tokens = TLM(tokens)
    tiles = ConvertTokensToTiles(response_tokens)
    if not CheckValidTiles(space, tiles) then
        raise Exception("Invalid Tensor Program")
    tokens.extend(Serialize(tiles))
    program.apply(space.operator, space.axis, tiles)
    // Other properties
```

5.2 Tensor Program Generation

The TLM framework utilizes TLM for decision-making during the tensor program generation process, as detailed in Algorithm 2. The steps in Algorithm 2 align with those in Algorithm 1, adhering to the consistency required for generating tensor sentences. After TLM generates a decision, the framework checks its validity within the decision space. If the decision is invalid, the framework discards the tensor program and initiates regeneration. Since the decision is obtained through sampling, the next generation might sample a different decision, preventing continuous failure in regeneration. Fortunately, following pre-training and fine-tuning,

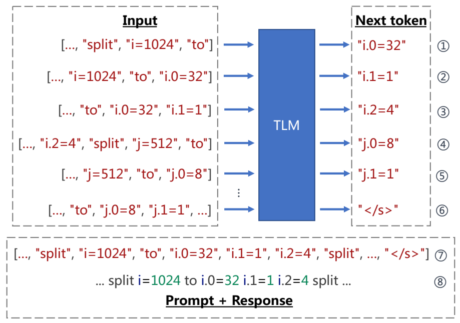
Input
[..., "split", "i=1024", "to"]
[..., "i=1024", "to", "i.0=32"]
[..., "to", "i.0=32", "i.1=1"]
[..., "i.2=4", "split", "j=512", "to"]
[..., "j=512", "to", "j.0=8"]
[..., "to", "j.0=8", "j.1=1", ...]
TLM
Next token
"i.0=32" ①
"i.1=1" ②
"i.2=4" ③
"j.0=8" ④
"j.1=1" ⑤
"</s>" ⑥
[..., "split", "i=1024", "to", "i.0=32", "i.1=1", "i.2=4", "split", ..., "</s>"] ⑦
... split i=1024 to i.0=32 i.1=1 i.2=4 split ... ⑧
Prompt + Response

Figure 7: Generating a tensor sentence for a matrix multiplication operator with dimensions $m = 1024$, $n = 512$, and $k = 1024$, with the tiling size component depicted therein.

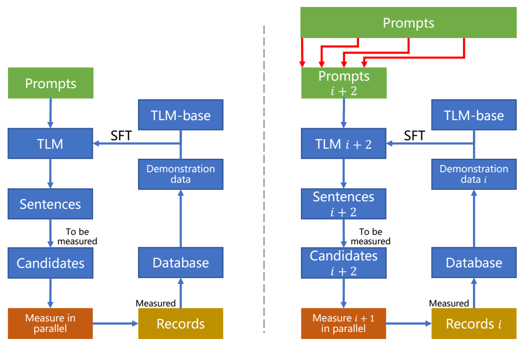

Figure 8: Flowchart of the iterative optimization process.

we observe that invalid decisions are exceedingly rare. On average, generating a valid tensor program necessitates no more than 1.1 calls to Algorithm 2.

Figure 7 illustrates the process of *response_tokens* = `TLM`(*tokens*) in Algorithm 2 for a matrix multiplication operator with dimensions $m = 1024$, $n = 512$, and $k = 1024$, with the tiling size component depicted therein. This tensor sentence matches the one at the top of Figure 6 and is presented here (with the operator's index omitted) in a more human-readable format. In Step ①, known information—including input subgraph, hardware specifications, and "split i=1024 to" (corresponds to $m = 1024$) — serves as the input prompt to TLM. TLM then predicts the probability distribution of the next token based on this prompt, subsequently choosing a token, assumably "i.0=32", via probabilistic sampling from the distribution. In Step ②, the prompt and the predicted next token from Step ① are combined to formulate a new input for TLM to forecast the next token. Steps ③ replicates Step ②. After completing the three steps, TLM finalizes tile sizes for the i-axis and returns to Algorithm 2. When necessary to generate tile sizes for the j-axis, TLM will be invoked again. In Step ④, the input from Step ③, its predicted next token, and "split j=512 to" (corresponding to $n = 512$) are merged into a new input, which is then input into TLM to predict the next token. TLM can but does not employ the input graph and hardware specifications as a prompt to generate all decision information in one go. Instead, the framework repeatedly invokes TLM within Algorithm 2, generating only a subset of decisions each time. This granular approach efficiently filters out invalid data, enhancing TLM availability and stability.

5.3 Iterative Optimization

Fine-tuning TLM-base using demonstration sentences is crucial for its operation. This section focuses on the methods for acquiring a batch of demonstration data.

Throughout acquiring demonstration data, measurement is the most time-consuming step and lies on the critical path, acting as the bottleneck of the process. To address this, we develop a pipeline system that executes the iterative optimization algorithm for collecting demonstration data. This system maximizes the utilization of the measurement hardware by employing two separate processes: one for executing SFT and another for measurement, with the former's execution time controlled to be shorter than the latter, ensuring continuous measurement. When operating on GPUs, these processes operate on separate GPU cards.

Figure 8 shows a flowchart of the iterative optimization on the left and a flowchart incorporating a pipeline on the right. We split (the text color matching that in Figure 8) all subgraphs (with each subgraph corresponding to one prompt) in the workload dataset into k_b (e.g., 4) batches and then cyclically select each batch. Upon the completion of the measurement of batch i (referred to as records i), the optimal batch of data is extracted from the database, noted as demonstration data i, and employed to fine-tune TLM-base, yielding TLM $i+2$. Subsequently, TLM $i+2$ is used to produce sentences $i+2$ (one prompt produces k_p (e.g., 16 or 32) sentences). TLM 0 and TLM 1 are replaced by TLM-base.

There are several details worth noting:

- When splitting subgraphs, we **sort them by subgraph type** (e.g., fused_nn_dense_add, fused_nn_conv2d_add_nn_relu, and fused_nn_adaptive_avg_pool2d), selecting one every k_b, based on the rationale that a superior decision often also has a certain optimizing effect on subgraphs of the same type.
- Only when TLM can generate superior tensor sentences can this iterative optimization algorithm operate normally. **So why can TLM generate more optimal data?** From a genetic perspective, when generating a sentence for a subgraph, TLM has already learned the demonstration data corresponding to the current subgraph, as well as demonstration data from other subgraphs, thereby inheriting the advantages of both itself and other subgraphs. When producing the next token for a prompt, probability sampling is used, which, owing to its stochastic nature, provides additional exploration and thus may introduce mutations. The combination of inheritance and mutation may potentially

generate higher quality data, aligning with the design philosophy of genetic algorithms.

- Demonstration refers to low execution latency. The total execution latency of all subgraphs can be expressed as $latency_{\text{total}} = \sum_{s_i \in \text{subgraphs}} \min(\text{all record latencies of } s_i)$. The latency of the demonstration data will not rise since it always selects the best from all measurements, so it decreases monotonically. Furthermore, the best latency of the demonstration data cannot be lower than its physical limit, so it is bounded. **Mathematically speaking, a monotonic and bounded limit results in convergence.** Section 6.2 of the evaluation discusses how much data needs to be measured for convergence.
- TLM is designed to assist in making decisions within a decision space, trained by a gradient descent strategy. TLM i may perform worse than TLM $i-1$ because gradient descent does not guarantee that TLM can always be trained to its optimal state. However, since TLM i is always trained from TLM-base, it remains unaffected by TLM $i-1$ and does not influence TLM $i+1$. Hence, **TLM i does not likely exhibit cumulative errors**, and the final TLM is only related to TLM-base and the final demonstration data.
- The demonstration data in Figure 8 is obtained through iterative optimization from scratch. Similarly, **there are also other methods to obtain a batch of demonstration data**. For instance, it can be obtained using other search algorithms; if there is a batch of demonstration data on Hardware A, it can be transferred to Hardware B through transfer learning; the data can be directly written using expert knowledge. The good news is that this demonstration data can still continue to use iterative optimization algorithms until convergence.
- **The iterative optimization algorithm can also be viewed as a tuning system, particularly suitable for situations where multiple workloads need to be tuned at once.** For instance, in dynamic shape scenarios, the target workload with different shapes need to be tuned simultaneously.

6 Evaluation

6.1 Experimental Settings

TLM supports several decision spaces, including those adapted from Ansor (V0.12), MetaSchedule (V0.12), AKG (V2.1), and AKG-MLIR (V0.1). In subsequent sections, these TLMs are referred to as TLM-Ansor, TLM-Meta, TLM-AKG, and TLM-AKG-MLIR. In the evaluation, we focus only on TLM-Ansor and TLM-Meta, which together are implemented in Python and C++ with about 10K lines of code.

The dataset we configured for TLM consists of 138 workloads. We hold out a test set that consists of 12 workloads, as shown in Table 1.

For the CPU experiments, we use a notebook equipped with a 4-core Intel(R) Core(TM) i7-10510U CPU supporting the AVX2 instruction set, 16GB memory, and running on Ubuntu 20.04. For the GPU experiments, we utilize a server outfitted with a 48-core Intel(R) Xeon(R) Gold 6226 CPU, 376GB memory, and four 32GB NVIDIA Tesla V100 GPUs. It runs on Ubuntu 20.04 with CUDA 11.6 and cuDNN 8.4.0.

Table 1: Workloads in the TLM test set.

Model	Input shape	Model	Input shape
ResNet-50 [32]	[1, 3, 224, 224]	DenseNet-121 [33]	[8, 3, 256, 256]
MobileNet-V2 [34]	[1, 3, 224, 224]	BERT-large	[4, 256]
ResNeXt-50 [35]	[1, 3, 224, 224]	Wide-ResNet-50 [36]	[8, 3, 256, 256]
BERT-base [37]	[1, 128]	ResNet3D-18 [38]	[4, 3, 144, 144, 16]
BERT-tiny	[1, 128]	DCGAN [39]	[8, 3, 64, 64]
GPT-2	[1, 128]	LLAMA [40]	[4, 256]

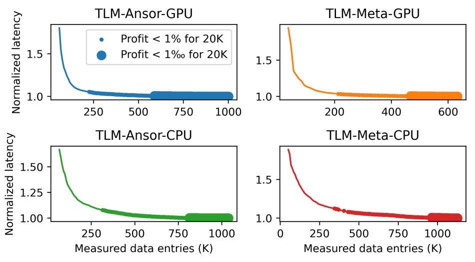

Figure 9: Demonstration data convergence curves of TLM-Ansor and TLM-Meta on the GPU and the CPU.

6.2 Convergence Behavior of Demonstration Data

This section discusses the convergence behavior of obtaining demonstration data through the iterative algorithm, which is the slowest phase of the entire pipeline. Figure 9 displays four curves corresponding to the convergence of TLM-Ansor and TLM-Meta on both the GPU and the CPU, respectively labeled as TLM-Ansor-GPU, TLM-Meta-GPU, TLM-Ansor-CPU, and TLM-Meta-CPU. These four scenarios utilize 2169, 3120, 2169, and 2657 subgraphs (extracted from 126 workloads). The horizontal axis denotes measured data entries (across all subgraphs), while the vertical axis indicates normalized total latency. We define "convergence" as the point at which 20K measurements (across all subgraphs) result in a performance improvement of less than one percent, denoted in the graph as "Profit < 1% for 20K." We also mark the instance where 20K measurements (across all subgraphs) yield a performance gain of less than one per thousand, corresponding to "Profit < 1‰ for 20K".

To reach the convergence state, an average of 104, 69, 145, and 130 measurements per subgraph are required, corresponding to overall totals of 225K, 213K, 314K, and 344K measurements, respectively, for all subgraphs. In other words, inducing convergence across all 126 workloads utilizing TLM calls for approximately 200K tensor program measurements on the GPU and about 300K on the CPU. TLM involves a data volume an order of magnitude smaller compared to TenSet and TLP, which utilize approximately 8.6 million measurements. Furthermore, for a performance gain of less than 1‰, individ-

Table 2: The overall speedup for the 23 TLM-Ansor subgraphs. The higher the overall speedup, the better. In the table, "Times" represents the measurement times for each subgraph.

		Ansor			TLM-Ansor
	Times	64	1K	10K	10K
TLM-Ansor	1	1.26	0.98	0.92	0.85
	10	1.40	1.08	1.03	0.95
	16	1.43	1.10	1.04	0.96
	32	1.45	1.12	1.06	0.98
	64	1.45	1.12	1.06	0.98
	1K	1.46	1.13	1.07	0.99
	10K	1.48	1.14	1.08	1.00
Ansor	10K	1.37	1.06	1.00	0.92

Table 3: The overall speedup for the 40 TLM-Meta subgraphs.

		MetaSchedule			TLM-Meta
	Times	64	1K	10K	1K
TLM-Meta	1	1.00	0.69	0.68	0.67
	10	1.41	0.96	0.95	0.94
	16	1.45	1.00	0.99	0.97
	32	1.46	1.01	1.00	0.97
	64	1.49	1.02	1.01	0.99
	1K	1.50	1.02	1.01	1.00
MetaSchedule	10K	1.48	1.01	1.00	0.99

ual subgraphs require 272, 150, 375, and 363 measurements each, which also translates to a cumulative total of 588K, 467K, 813K, and 963K measurements for all subgraphs. The TLM used in the experiments of the subsequent sections has been fine-tuned using about 300K labelled data.

6.3 Subgraph Benchmark

After SFT, we conduct subgraph experiments on the NVIDIA V100. The TLM test set comprises 12 workloads, yielding 232 and 364 subgraphs for TLM-Ansor and TLM-Meta, respectively. These subgraphs fall into 23 and 40 categories, and we select one representative subgraph from each category for the experiments. Two comparisons are established: TLM-Ansor vs. Ansor and TLM-Meta vs. MetaSchedule, with measurement times set at 10K, 10K, 1K, and 10K for each subgraph, respectively. The latency of a subgraph is defined as the lowest value among its measurements. The latency comparison curves are presented in Figures 14 and 15 in the Appendix B, while here, we focus on critical data highlights in Tables 2 and 3.

We define Framework$_k$ as follows:

$$\text{Framework}_k = \sum_{s_i \in \text{subgraphs}} \min \left(\begin{array}{c} \text{Framework's } k \\ \text{record latencies of } s_i \end{array} \right),$$

where Framework can be TLM-Ansor, TLM-Meta, Ansor, or MetaSchedule, and k represents the measurement times. Each speedup in Table 2 represents the overall speedup for the 23 subgraphs of Framework1$_{k_1}$ in the first two columns compard to Framework2$_{k_2}$ in the first two rows. Table 3 is similar to Table 2.

In Tables 2 and 3, TLM-Ansor$_{1K}$ achieves 99% (the text color matching that in the tables) of the performance of TLM-Ansor$_{10K}$, while TLM-Meta$_{64}$ attains 99% of the performance of TLM-Meta$_{1K}$. Furthermore, MetaSchedule$_{10K}$ shows a speedup of 1.01× compared to MetaSchedule$_{1K}$. We observe that a tenfold increase in the number of measurements results in a performance gain of no more than one percent. Achieving further acceleration in the **current** exploration space is challenging, and a better approach to gain additional speedup is to explore a larger space.

The primary goal of the TLM framework is to generate high-performance tensor programs efficiently. Notably, even with 10 measurements, TLM can achieve 103% and 95% of the performance of Ansor and MetaSchedule after 10K measurements, respectively. Additionally, TLM-Ansor$_{32}$ achieves a 1.06× speedup over Ansor$_{10K}$, while TLM-Meta$_{32}$ reaches a 1.00× speedup compared to MetaSchedule$_{10K}$. It is evident that TLM's performance, with 32 measurements, has already exceeded that of both Ansor and MetaSchedule.

TLM-Ansor$_{1K}$ achieves a 1.13× speedup over Ansor$_{1K}$, TLM-Ansor$_{10K}$ reaches a 1.08× speedup compared to Ansor$_{10K}$, and TLM-Meta$_{1K}$ attains a 1.02× speedup over MetaSchedule$_{1K}$. These ratios demonstrate that TLM consistently achieves acceleration relative to Ansor and MetaSchedule with equal measurement times. To attain a higher acceleration ratio, building a larger exploration space might be necessary.

6.4 End-to-End Workload Benchmark

6.4.1 Comparison with Ansor/MetaSchedule

For both GPU and CPU, we set up four experiments: 1) Tuning with Ansor (V0.12) and conducting 20K measurements; 2) Tuning with MetaSchedule (V0.12) and conducting 20K measurements; 3) Generating tensor programs using TLM-Ansor, carrying out 20K measurements, and reporting end-to-end performance for $1 \times g$, $10 \times g$, and $32 \times g$ measurements, where g indicates the number of subgraphs partitioned from the test workload[1]; 4) Generating tensor programs using TLM-Meta, with the same settings as in 3.

GPU results. Initially focusing on the last four columns of each workload in Figure 10, TLM-Ansor-20K shows a speedup of 0.99-1.38× compared to Ansor-20K across 12 test workloads, with the average speedup being 1.08. Similarly, TLM-Meta-20K achieves a speedup of 0.98-1.14× relative to MetaSchedule-20K, with the average speedup being 1.04. These results align with those from the subgraph benchmark, indicating that TLM offers acceleration over Ansor/MetaSchedule.

The primary goal of the TLM framework is to efficiently generate high-performance tensor programs. We now turn our attention to the first six columns in Figure 10. For Ansor-20K,

[1]For instance, TLM-Ansor can partition 9 subgraphs from BERT-base; thus, $1 \times g$ represents 9 measurements, $10 \times g$ represents 90, and $32 \times g$ represents 288.

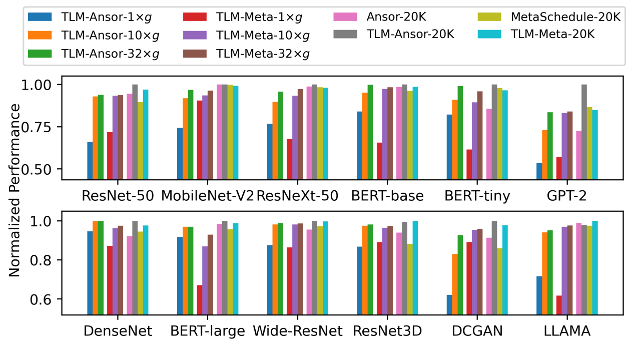

Figure 10: Workload inference performance comparison with Ansor/MetaSchedule on V100.

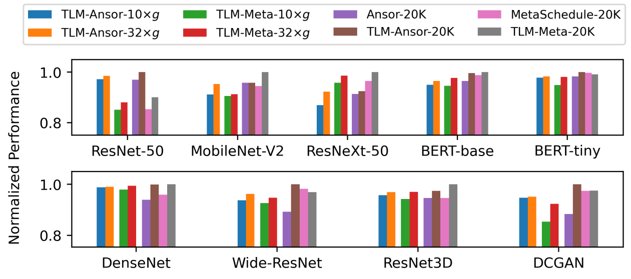

Figure 11: Workload inference performance comparison with Ansor/MetaSchedule on the Intel CPU.

TLM-Ansor-1×g achieves a 0.68-1.03 speedup, with an average speedup of 0.83; TLM-Ansor-10×g offers a 0.91-1.08 speedup, averaging at 0.99; and TLM-Ansor-32×g speeds up by 0.96-1.16, with an average speedup of 1.03. Regarding MetaSchedule-20K, TLM-Meta-1×g can accelerate by 0.63-1.04, with an average speedup of 0.80; TLM-Meta-10×g achieves a 0.91-1.11 speedup, averaging at 1.00; and TLM-Meta-32×g offers a 0.96-1.12 speedup, with an average speedup of 1.02.

To summarize, TLM can reach 80% of Ansor/MetaSchedule's performance by conducting only 1×g measurements, as opposed to the 20K measurements required by Ansor/MetaSchedule. With 10×g measurements, TLM's performance aligns with that of Ansor/MetaSchedule. Notably, the purpose of measurement is to identify the tensor program with the lowest execution latency. The fact that only one measurement is needed implies no necessity for measurement, indicating that TLM can reach 83% of Ansor/MetaSchedule's performance without any measurement. This indicates the feasibility of applying the TLM strategy within deep learning training frameworks (e.g., PyTorch, TensorFlow, MindSpore [41]).

CPU results. The CPU results (excluding BERT-large, GPT-2, and LLAMA, which will trigger the OOM error), depicted in Figure 11, align with the GPU outcomes. Here,

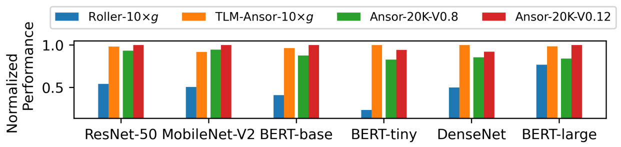

Figure 12: Workload inference performance comparison with Roller on V100.

we simply present the average speedup. Relative to Ansor-20K, TLM-Ansor-10×g achieves a 1.01× speedup, while TLM-Ansor-32×g accomplishes a 1.03× speedup. Compared to MetaSchedule-20K, TLM-Meta-10×g attains a 0.97× speedup, and TLM-Meta-32×g achieves a 1.00× speedup.

Compilation time. This paper uses the measurement times to calculate the speedup of compilation time, the justification for which is discussed in the Appendix A. Simply put, the time allocated to measurement predominates the entire compilation time and remains unaffected by the system's load, establishing it as a stable metric for the issue.

In the TLM-Ansor test set, the 12 workloads comprise 5-72 subgraphs, averaging 20.9 subgraphs. As a result, TLM-Ansor-10×g, compared to Ansor-20K, can deliver the same performance level while accelerating compilation by 95×. Similarly, TLM-Meta-10×g can speed up compilation by 61× relative to MetaSchedule-20K.

Summary. In subgraph and end-to-end benchmarks, we primarily analyze from a statistical perspective rather than investigating why certain subgraphs or workloads surpass the baseline. This is due to the inherent randomness of the probabilistic/random sampling, which results in a lack of clear patterns in speedup. What becomes apparent, however, is that across nearly all subgraphs and workloads, TLM matches the baseline results with significantly fewer measurements. This indicates a general improvement in exploration capabilities.

6.4.2 Comparison with Roller on V100

Roller is implemented on top of TVM (V0.8) and Rammer [42]. We utilize the Docker image provided by Roller for experiments, which runs on Ubuntu 16.04 with CUDA 10.2 and cuDNN 7.6.5; it lacks maintenance to utilize the latest CUDA. To observe the impact of software versions on performance, we present the performance of Ansor, integrated within TVM (V0.8), as a bridge for comparing TLM and Roller; the performance of Ansor is measured in the same execution environment as Roller. Compiling workloads with Roller requires carefully configured, workload-specific script files; we offer script files for six workloads.

We set up four experiments: 1) Compiling with Roller and performing 10×g measurements; 2) Adopting TLM-Ansor-10×g from §6.4.1; 3) Tuning with Ansor (V0.8) and executing 20K measurements; 4) Adopting Ansor-20K from §6.4.1, designated as Ansor-20K-V0.12.

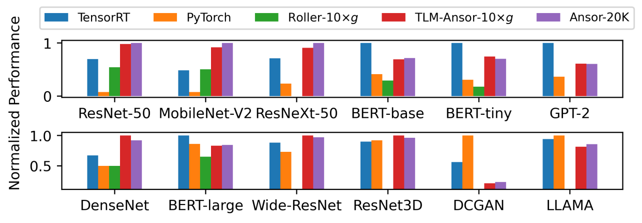

Figure 13: Workload inference performance comparison with TensorRT/PyTorch on V100.

Figure 12 illustrates that in the same execution environment, Ansor-20K-V0.8 outperforms Roller-10×g. Additionally, TLM-Ansor-10×g and Ansor-20K-V0.12 have comparable performance levels, slightly better than Ansor-20K-V0.8. Therefore, it is evident that TLM-10×g surpasses Roller-10×g when the influence of software versions is excluded. In a direct comparison, TLM-Ansor-10×g achieves a speedup of 1.28-4.23× compared to Roller-10×g, with the average speedup being 2.25. The reason is that, in pursuit of high efficiency, Roller significantly prunes the exploration space by aligning tensor shapes with the key features of the hardware, as discussed earlier.

6.4.3 Comparison with TensorRT/PyTorch on V100

In this section, we conduct a performance comparison against TensorRT [43] (V8.6) and PyTorch (V1.13.1) on V100. Both TensorRT and PyTorch are backed by static kernel libraries.

We set up five experiments: 1, 2) Performing inference using TensorRT and PyTorch; 3) Adopting Roller-10×g from §6.4.2; 4, 5) Adopting TLM-Ansor-10×g and Ansor-20K from §6.4.1.

Figure 13 illustrates that relative to TensorRT, TLM-Anosr-10×g achieves a 0.38-1.89× speedup, averaging at 1.04; compared to PyTorch, TLM-Anosr-10×g sees a 0.21-12.92× increase in performance, with an average speedup of 3.42. TensorRT outperforms TLM-Anosr-10×g in BERT-tiny, BERT-base, BERT-large, GPT-2, LLAMA, and DCGAN workloads, while PyTorch excels over TLM-Anosr-10×g in BERT-large, LLAMA, and DCGAN. The primary components of BERT and LLAMA are batch matmul operators, GPT-2 mainly involves matmul operators, and DCGAN primarily uses transposed 2D convolution operators. The speedup achieved by TensorRT/PyTorch is attributed to the deep optimization of batch matmul, matmul, and transposed 2D convolution operators in recent kernel libraries, as well as the utilization of hardware computing units. Overall, tensor compilers excel in supporting a wide range of operators, while static kernel libraries are more adept at deeply optimizing commonly used operators.

7 Related Work

Halide [44] introduces the concept of separating compute and schedule, employing a domain specific language (DSL) to define computations and scheduling primitives to abstract hardware characteristics, enhanced by an auto scheduler [4,45,46] for optimal primitive combination. TVM [3], inheriting the philosophy of Halide, utilizes scheduling primitives for operator implementation. It currently boasts three generations of tensor program search frameworks: The first generation maps subgraphs to tensor programs using templates and optimizes them through AutoTVM [12]. The second generation, Ansor [13], addresses the limitations of template-based exploration spaces by constructing tensor programs with derivation rules and searching for efficient programs using the genetic algorithm. The third generation, which includes TensorIR [47] and MetaSchedule [14], tackles the challenges of supporting TensorCore, introducing a block abstraction that isolates tensorized computations for mapping to tensor computing units. TenSet [20] and TLP [21] propose using offline datasets to address the issue of extended search times brought by Ansor/MetaSchedule. Roller [23] introduces a tile abstraction that encapsulates tensor shapes, aligning them with the key features of the underlying accelerator to limit shape choices. FlexTensor [6] is a schedule exploration and optimization framework proposing automatically general templates to map the tensor algorithms onto low-level implementations for different hardware platforms.

Tiramisu [48], AKG [15], and Tensor Comprehensions [5] apply polyhedral-based techniques, formulating the optimization of programs as an Integer Linear Programming (ILP) problem. Triton [49] introduces a tile-based template representation where programmers can specify block sizes and manage their scheduling for effective program optimization. CUTLASS [50] is a collection of template abstractions for implementing high-performance matrix-matrix multiplication and related computations within CUDA at all levels and scales. MLIR [51] builds reusable and extensible compiler infrastructure to address software fragmentation and improve compilation for heterogeneous hardware.

8 Discussion

Limitation. 1) At its core, TLM is a deep learning model that leverages offline data to guide exploration. The data distributions of the target and training scenarios need to be aligned. To achieve optimal performance in scenarios with significant discrepancy, it might be necessary to incorporate additional training data. 2) TLM's design philosophy is trading compile time for pre-compile time. The overhead of training TLM, which could span tens of hours, should not be overlooked. If the intent is solely to compile one or just a handful of models, TLM might not be the most economical choice. Instead, TLM is better suited for compiling a vast array of models.

Future Work. 1) With 100M parameters, TLM's time and hardware overhead for training and inference should be noticed. It may be worthwhile to explore substantial reductions in parameter size while ensuring TLM's performance remains robust. 2) In contrast, with just 100M parameters, there is an opportunity to substantially expand TLM's parameter size and the training data. These improvements could lead to remarkable generalization capabilities. Examples include compiling directly from TLM-generated results without measurement or achieving strong generalization across various hardware platforms. 3) Delve into a more expansive exploration space. Equipped with TLM's potent exploration capabilities, it's feasible to navigate vast territories without being constrained by exploration costs.

9 Conclusion

We introduce the TLM framework, a novel tensor program generation framework that consists of two decoupled components: a space builder and a generator. The TLM framework is pioneering in its integration of language models into the tensor program domain. Owing to TLM's dual strengths of adeptly learning from offline data and effectively capturing context, it demonstrates formidable generative capabilities. Experimental results show that TLM consistently delivers both high efficiency and high performance.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (No. 62332016, No. 62272434, and No. U20A20226).

References

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

[2] Intel® oneAPI Deep Neural Network Library. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onednn.htm.

[3] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis 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)*, pages 578–594, 2018.

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

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

[6] Size Zheng, Yun Liang, Shuo Wang, Renze Chen, and Kaiwen Sheng. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 859–873, 2020.

[7] Yizhi Liu, Yao Wang, Ruofei Yu, Mu Li, Vin Sharma, and Yida Wang. Optimizing {CNN} model inference on {CPUs}. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 1025–1040, 2019.

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

[9] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

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

[11] Jared Roesch, Steven Lyubomirsky, Logan Weber, Josh Pollock, Marisa Kirisame, Tianqi Chen, and Zachary Tatlock. Relay: A new ir for machine learning frameworks. In *Proceedings of the 2nd ACM SIGPLAN international workshop on machine learning and programming languages*, pages 58–68, 2018.

[12] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Learning to optimize tensor programs. *Advances in Neural Information Processing Systems*, 31, 2018.

[13] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating {High-Performance} tensor programs for deep learning. In *14th USENIX symposium on operating systems design and implementation (OSDI 20)*, pages 863–879, 2020.

[14] Junru Shao, Xiyou Zhou, Siyuan Feng, Bohan Hou, Ruihang Lai, Hongyi Jin, Wuwei Lin, Masahiro Masuda, Cody Hao Yu, and Tianqi Chen. Tensor program optimization with probabilistic programs. *Advances in Neural Information Processing Systems*, 35:35783–35796, 2022.

[15] Jie Zhao, Bojie Li, Wang Nie, Zhen Geng, Renwei Zhang, Xiong Gao, Bin Cheng, Chen Wu, Yun Cheng, Zheng 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*, pages 1233–1248, 2021.

[16] The LLVM Compiler Infrastructure. URL: https://llvm.org/.

[17] NVIDIA CUDA Compiler Driver NVCC. URL: https://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/index.html.

[18] Open standard for machine learning interoperability. URL: https://onnx.ai.

[19] ChatGPT. URL: https://openai.com/blog/chatgpt.

[20] Lianmin Zheng, Ruochen Liu, Junru Shao, Tianqi Chen, Joseph E Gonzalez, Ion Stoica, and Ameer Haj Ali. Tenset: A large-scale program performance dataset for learned tensor compilers. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1)*, 2021.

[21] Yi Zhai, Yu Zhang, Shuo Liu, Xiaomeng Chu, Jie Peng, Jianmin Ji, and Yanyong Zhang. Tlp: A deep learning-based cost model for tensor program tuning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 833–845, 2023.

[22] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[23] Hongyu Zhu, Ruofan Wu, Yijia Diao, Shanbin Ke, Haoyu Li, Chen Zhang, Jilong Xue, Lingxiao Ma, Yuqing Xia, Wei Cui, et al. {ROLLER}: Fast and efficient tensor compilation for deep learning. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 233–248, 2022.

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

[25] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[26] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[27] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

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

[29] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[30] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. MIT press, 1992.

[31] John W Backus, Friedrich L Bauer, Julien Green, Charles Katz, John McCarthy, Alan J Perlis, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Joseph Henry Wegstein, et al. Report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, 1960.

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

[33] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[34] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[35] Saining Xie, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. Aggregated residual transformations for deep neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1492–1500, 2017.

[36] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. *arXiv preprint arXiv:1605.07146*, 2016.

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

[38] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. Learning spatio-temporal features with 3d residual networks for action recognition. In *Proceedings of the IEEE international conference on computer vision workshops*, pages 3154–3160, 2017.

[39] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[40] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[41] MindSpore. URL: https://www.mindspore.cn/.

[42] Lingxiao Ma, Zhiqiang Xie, Zhi Yang, Jilong Xue, Youshan Miao, Wei Cui, Wenxiang Hu, Fan Yang, Lintao Zhang, and Lidong Zhou. Rammer: Enabling holistic deep learning compiler optimizations with {rTasks}. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 881–897, 2020.

[43] NVIDIA TensorRT. URL: https://developer.nvidia.com/tensorrt.

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

[45] Tzu-Mao Li, Michaël Gharbi, Andrew Adams, Frédo Durand, and Jonathan Ragan-Kelley. Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics (ToG)*, 37(4):1–13, 2018.

[46] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically scheduling halide image processing pipelines. *ACM Transactions on Graphics (TOG)*, 35(4):1–11, 2016.

[47] Siyuan Feng, Bohan Hou, Hongyi Jin, Wuwei Lin, Junru Shao, Ruihang Lai, Zihao Ye, Lianmin Zheng, Cody Hao Yu, Yong Yu, 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, Volume 2*, pages 804–817, 2023.

[48] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. Tiramisu: A polyhedral compiler for expressing fast and portable code. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 193–205. IEEE, 2019.

[49] Philippe Tillet, Hsiang-Tsung Kung, and David 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*, pages 10–19, 2019.

[50] CUTLASS. URL: https://github.com/NVIDIA/cutlass.

[51] Multi-Level Intermediate Representation Overview. URL: https://mlir.llvm.org.

A Compilation Speedup Metrics

In this section, we discuss the rationality of using measurement times to calculate the speedup of compilation time.

$$\begin{aligned}
T_{total} &= T_{exploration} + T_{post\ exploration\ compilation} \\
&= \sum_k \left(c \times T_{sample} + T_{measurement}\right) \\
&\quad + T_{post\ exploration\ compilation} \\
&= \sum_k \left(c \times T_{sample} + (T_{compilation} + T_{execution})\right) \\
&\quad + T_{post\ exploration\ compilation}
\end{aligned}$$

The total time for compiling a workload (T_{total}), as indicated in the formula above, consists of two main components: the time spent exploration to identify high-performance tensor programs ($T_{exploration}$), and the time for the final compilation of the workload ($T_{post\ exploration\ compilation}$). The exploration process involves sampling c tensor programs ($c \times T_{sample}$) and then measuring the execution latency of one of these programs ($T_{measurement}$). In Ansor/MetaSchedule, the purpose of sampling multiple tensor programs is to use a cost model to select the most promising tensor program; if a program sampled by TLM is found to be invalid, it will be resampled. The sampling coefficients c_{Ansor} and $c_{MetaSchedule}$ are approximately 128, while c_{TLM} does not exceed 1.1. Measuring a tensor program includes both its compilation ($T_{compilation}$) and execution ($T_{exection}$).

$$\begin{aligned}
T_{total} &= T_{exploration} + T_{post\ exploration\ compilation} \\
&\approx T_{exploration} \quad \text{if } k \geq k_0 \\
&= \sum_k \left(c \times T_{sample} + T_{measurement}\right) \\
&= \sum_k T_{measurement} \quad \text{if } T_{measurement} \text{ hides } c \times T_{sample} \\
&= \sum_k \left(T_{compilation} + T_{execution}\right)
\end{aligned}$$

When measurement times k exceeds a certain threshold k_0 (e.g., 100), $T_{exploration}$ becomes the dominant factor in T_{total}, making $T_{post\ exploration\ compilation}$ negligible. Sampling and measurement processes can be optimized using a pipeline approach for parallel execution. Overall, measurement is a critical part of the entire compilation process, and reducing measurement times requires methodological innovation rather than just engineering efforts.

TLP uses the speedup on T_{total} to calculate the acceleration of compilation time, while Roller uses $\sum_k T_{compilation}$ to calculate the acceleration of compilation time. Although they indeed reflect the speedup of compilation time to some extent, these metrics are unstable due to the influence of system load. In this paper, we use measurement times k to calculate the speedup of compilation time.

B Subgraph Performance

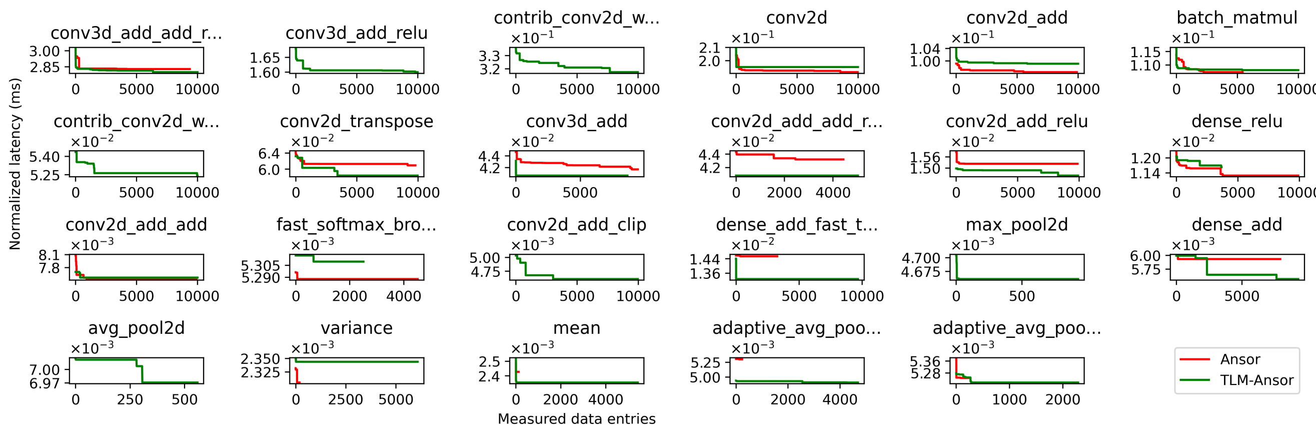

Figure 14: Subgraph latency comparison curves for TLM-Ansor vs. Ansor. To enhance the comparison of subtle details, the curve only includes latencies that do not exceed 1.1 times the lowest latency of either TLM-Ansor or Ansor. In the figure, if one is not visible throughout, it indicates that its lowest latency exceeds that of the other by 10%.

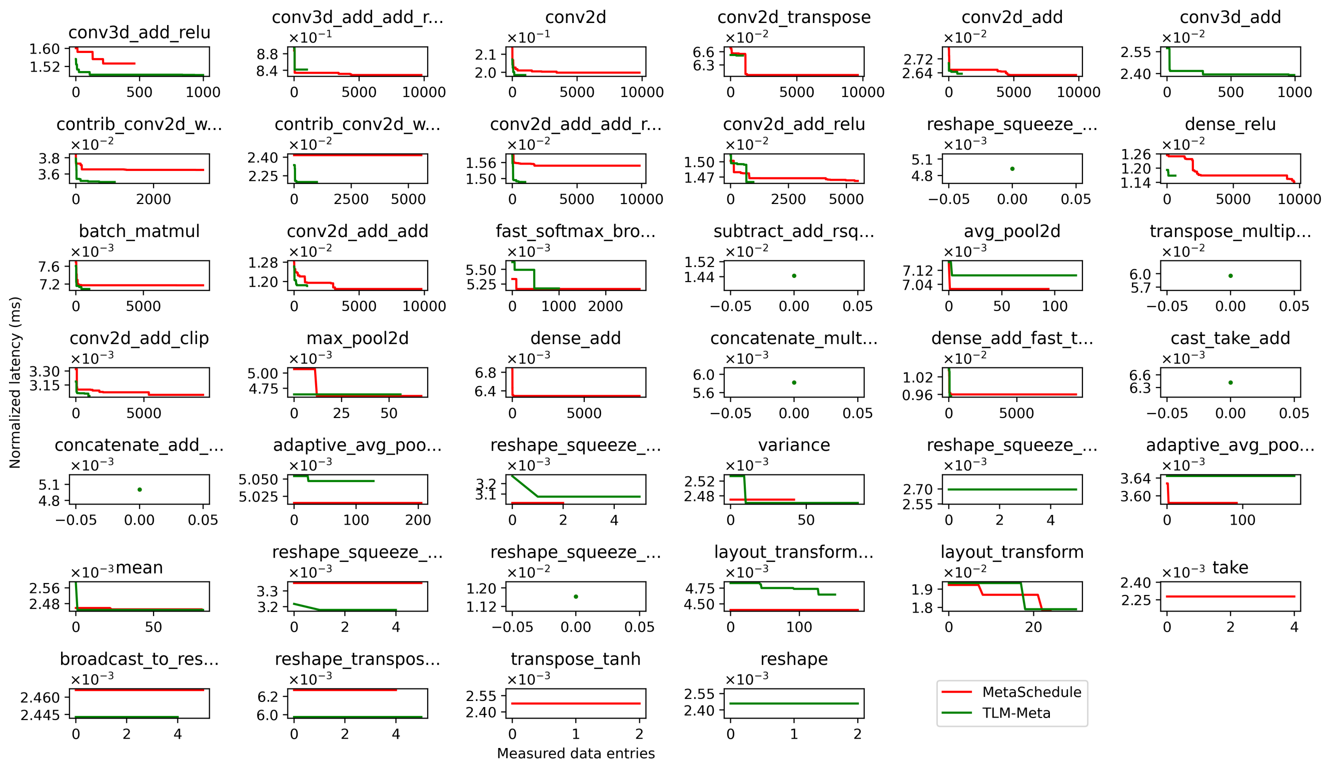

Figure 15: Subgraph latency comparison curves for TLM-Meta vs. MetaSchedule. To enhance the comparison of subtle details, the curve only includes latencies that do not exceed 1.1 times the lowest latency of either TLM-Meta or MetaSchedule. In the figure, if one is not visible throughout, it indicates that its lowest latency exceeds that of the other by 10%.