FTP: A Fine-grained Token Pruner for Large Language Models via Token Routing

Anonymous ACL submission

Abstract

The substantial computational overhead of large language models (LLMs) often presents a major challenge for their deployment in industrial applications. Many works leverage traditional compression approaches to boost model inference, however, these methods typically incur additional training costs to restore performance by updating the LLM's weights. Alternatively, pruning often results in significant performance degradation compared to the original model when aiming for a specific level of acceleration. To address these issues, we propose a fine-grained token pruning approach for the LLMs, which presents a learnable router to adaptively identify the less important tokens and skip them across model blocks to reduce computational cost during inference. To construct the router efficiently, we present a search-based sparsity scheduler for pruning sparsity allocation, a trainable router combined with our proposed four low-dimensional factors as input and three proposed losses. Furthermore, we introduce a one-pass learnable router designed for batch inference and enhanced acceleration. We have conducted extensive experiments across different benchmarks on different LLMs to demonstrate the superiority of our method. Our approach achieves state-of-the-art (SOTA) pruning results, surpassing other existing pruning methods. For instance, our method outperforms BlockPruner and ShortGPT by approximately 10 points on both LLaMA2-7B and Qwen1.5-7B in accuracy retention at comparable token sparsity levels.

1 Introduction

Large language models (LLMs) (Zhao et al., 2023; Minaee et al., 2024) have recently attracted significant attention for their strong performance in various natural language processing (NLP) tasks, largely driven by the success of the ChatGPT series. Currently, the design of LLMs is typically following the scaling law, which increases their size and complexity to boost performance. However, this results in high memory usage and computational demands, thereby restricting its deployment in industrial applications.

Existing model compression techniques, such as knowledge distillation (Huang et al., 2022; Gu et al., 2024), model pruning (Gao et al., 2020; Li et al., 2023a; Wang et al., 2024), quantization (Dettmers et al., 2024; Yao et al., 2022), and conditional computation (Schuster et al., 2022; Liu et al., 2023; Akhauri et al., 2024), have proven effective in improving the efficiency of LLMs. Model pruning is a popular technique in industrial applications to boost model inference, which usually identifies and removes the less important weights to reduce the computation overhead. Adopting traditional model pruning to accelerate, often requires training LLM itself (e.g., APT (Zhao et al., 2024) and LLM-Pruner (Ma et al., 2023)) to restore accuracy. Retraining LLM is costly and would cause deployment instability. Recently, an interesting research named ShortGPT (Men et al., 2024) reveals the significant redundancy when forwarding features between blocks. They completely remove some blocks based on a simple metric without LLM training, encouragingly, yet the LLM still maintains the better performance than previous model pruning approaches despite this drastic modification. Additionally, further researchs (Zhong et al., 2024; Liu et al., 2024) corroborates this finding, demonstrating that pruning LLMs by reducing depth holds great potential. However, while these block-wise depth pruning approaches underscore the impressive redundancy in LLMs, it remains unconvincing to eliminate all operations within blocks through such radical pruning. According to their reports, pruned models exhibit noticeable deterioration on specific benchmarks, indicating that valuable knowledge may still reside in the discarded layer operations.

To address the above issues, we present a **F**ine-

grained **T**oken **P**runing framework (FTP) via token routing, which can adaptively prune tokens within each block of LLMs based on the varying inputs during inference. FTP integrates a sparsity scheduler to assign a sparsity ratio to each block, ensuring the overall pruning rate target is met. Additionally, it employs a learnable router to prune unimportant tokens in the sequence, leveraging four key factors as input. We design three simple but effective steps to obtain the token router. First, we introduce a sparsity scheduler and a static router (one with fixed routing rule) to get an initial sparsity allocation on blocks. Second, we train a learnable router based on initial sparsity allocation. Finally, we refine the final sparsity allocation based on the frozen learnable router via sparsity scheduler. Additionally, we implement three crucial losses in router training (i.e. guide, sparsity constraint, and distillation losses). These innovative implementations contribute to a robust and effective token LLM pruning method. Furthermore, we introduce a one-pass learnable router optimized for batch inference, enabling enhanced efficiency and acceleration.

To verify the effectiveness of our method, we conduct extensive experiments on various LLMs including Qwen and LLaMA series models with our token pruning method. Our method significantly surpasses the other SOTA pruning methods by a large margin without retraining the LLMs, which fully demonstrates the superiority of our method. Our contributions can be mainly summarized as follows:

- We propose a token pruning framework (**FTP**) consisting of three steps: 1) initial sparsity searching via sparsity scheduler, 2) A designed learnable router training with three losses, 3) final sparsity searching on the learnable router.
- We introduce a one-pass learnable router for an enhanced acceleration and batch inference.
- Extensive experiments have been conducted on various LLMs with our proposed method, which indicates that our method surpasses other SOTA pruning methods for LLMs by a large margin.

2 Related Work

LLM Pruning. Pruning in LLMs aims to identify and remove redundant weights or tokens from models. As for weight-level pruning, SparseGPT (Frantar and Alistarh, 2023) addresses the layer-wise reconstruction problem for pruning by computing Hessian inverses. Wanda (Sun et al., 2023) introduces a pruning criterion that involves multiplying weight magnitudes by input feature norms. Moreover, FLAP (An et al., 2024), LLM-Pruner (Ma et al., 2023), Sheared-LLaMA (Xia et al., 2023) and BlockPruner (Zhong et al., 2024) eliminate coupled structures in the aspect of network width while retaining the number of layers, while FoldGPT (Liu et al., 2024) and ShortGPT (Men et al., 2024) exploit model depth redundancy to obtain lightweight models. As for token-level pruning, selective-context (Li et al., 2023b) merges tokens into units, and then applies prompt pruning based on the self-information indicator. STDC (Yin et al., 2023) prunes the prompts based on the parse tree, which iteratively removes the phrase nodes that cause the smallest performance drop after pruning it. LLMLingua (Jiang et al., 2023a) and LongLLMLingua (Jiang et al., 2023b) perform demonstration-level pruning followed by token-level pruning based on perplexity. PCRL (Jung and Kim, 2024) introduces a token-level pruning scheme based on reinforcement learning. However, most existing pruning approaches permanently remove weights or tokens, which may significantly degrade accuracy for more challenging tasks.

Conditional Computing. Removes weights or tokens from LLMs would result in a significant drop in accuracy, especially for more challenging tasks. A wide variety of recent work has developed to dynamically activate weights or tokens instead of removing them, also named conditional computing. DejaVu (Liu et al., 2023) dynamically activates neurons and attention heads of each LLM's layer by building predictors to estimate sparsity patterns. ShadowLLM (Akhauri et al., 2024) dynamical activates weights based on the context (input) itself by training a predictor to predict the sparsity pattern dependent on the input tokens. However, the sparse activation of weights still hurts the generability of models. Many works (Elbayad et al., 2020; Liu et al., 2021; Schuster et al., 2022) utilize early exiting to learn to decide when to end computation on a given token, allowing the token to skip any remaining transformer layers. MoD (Raposo et al., 2024) dynamically selects tokens via a trainable router for each block which takes hidden states as input and manually specifies the sparsity ratios for every block, and requires training from scratch. In contrast, our work proposes a global token router

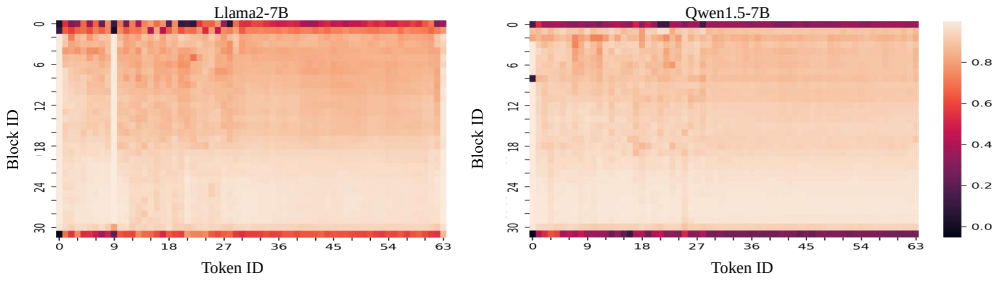

Figure 1: Token similarity across transformer blocks.

that takes designed input instead of hidden states, combined with a sparsity scheduler using a static router for pruning sparsity allocation for all blocks. It is trained to evaluate token importance to control tokens' skipping or computation for each block.

3 Method

In this section, we first provide a detailed analysis of token redundancy of LLMs, and then introduce the details of **F**ine-grained **T**oken **P**runing framework (FTP) via token routing.

3.1 Preliminary

3.1.1 Token Redundancy

LLMs are mostly built upon the transformer architecture. Here we define L is the sequence length, and d is the dimension of transformer. Previous works have shown that transformers exhibit certain semantic capabilities in earlier blocks (Hasan et al., 2021), and there is substantial block-wise redundancy throughout the model (Men et al., 2024). Furthermore, MoD (Raposo et al., 2024) demonstrates that selectively dropping tokens across blocks can still maintain performance comparable to a fully dense transformer. In this paper, we uncover significant token-wise redundancy across blocks during the inference phase.

We first randomly select 50 sequences from Alpaca dataset (Taori et al., 2023) following ShortGPT (Men et al., 2024), each consisting of 64 tokens, and calculate the similarity between the input hidden state and output hidden state from each token of all blocks on both the LLaMA2-7B-base and Qwen1.5-7B-base models. We illustrate the heatmap from above collected data in Figure 1. Higher similarity indicates that a block has less influence on the token, while greater changes in hidden states suggest lower token redundancy. Our analysis reveals the following key insights:

First, we observe substantial token redundancy across both models. Specifically, 89.94% and 93.16% of tokens in LLaMA2-7B and Qwen1.5-7B, respectively, exhibit a similarity score higher than 0.8, suggesting minimal changes and a high potential for pruning. Conversely, only 10.06% and 6.84% of tokens have similarity scores below 0.8, indicating meaningful transformations. Second, token redundancy varies across the blocks of the transformer. Tokens in the initial and final blocks show more significant changes, while tokens in the middle blocks exhibit greater redundancy. Specifically, in the first and last three blocks, 49.74% and 35.42% of tokens have similarity scores below 0.8, while in the middle blocks, 99.10% and 99.76% of tokens have similarity scores above 0.8 in LLaMA2-7B and Qwen1.5-7B, respectively.

3.1.2 Simultaneously Allocating Sparsity and Pruning Tokens is Non-Trivial

As shown in Figure 1, we observe that the redundancy levels of different blocks are inconsistent, and the redundancy patterns of tokens within the block are not fixed. Therefore, we need to design a scheme that can simultaneously determine the sparsity ratios for each block and the pruning patterns for each block's tokens. However, optimizing both the sparsity allocation and token pruning patterns across blocks increases the complexity of the optimization. Previous methods typically relied on empirical values to manually specify sparsity rates for each block, which can result in suboptimal performance.

3.2 Fine-Grained Token Pruner (FTP)

To address the challenge of simultaneously allocating sparsity and optimizing token pruning within blocks, we divide the problem into several steps. As shown in Figure 2, the pruning pipeline consists of three steps: 1) initial sparsity searching via sparsity scheduler, 2) learnable router training with three losses, 3) final sparsity searching for the learnable router. Note that these steps can be repeated more times to further enhance performance; however, in our approach, we only repeat them once for simplicity of training.

3.2.1 Token Routing Paradigm

For each block, the input tokens are assessed by a token router, which gives the decision of skip

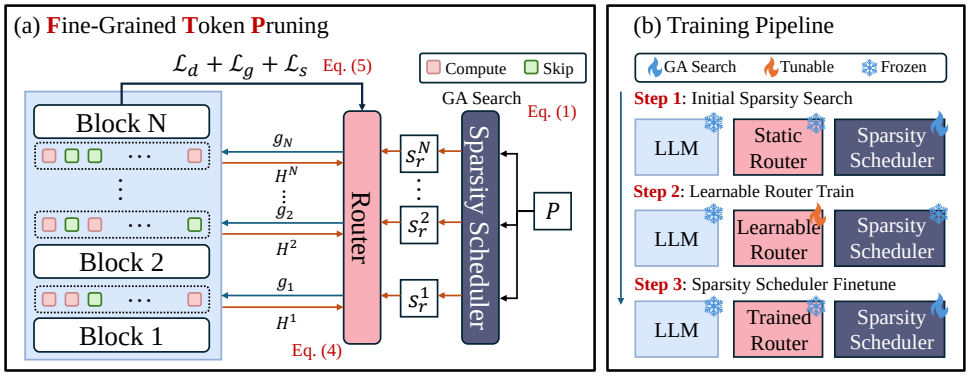

Figure 2: Overview of our method. (a) Our Fine-Grained Token Pruning uses designed input H to guide gate prediction, skipping computation instead of discarding tokens. A GA-based scheduler optimizes sparsity per block, and the router is trained with three proposed losses. (b) We decouple sparsity scheduling and router training into three steps, simplifying the optimization.

and computation. A specific proportion of the most important tokens is then selected to undergo the block computation based on sparsity requirements. The unselected tokens, meanwhile, skip the block's computation and remain unchanged until next block.

3.2.2 Sparsity Scheduler

We implement a sparsity scheduler that aims to allocate the sparsity ratio for each block while maintaining overall model sparsity and maximizing the evaluation accuracy. Here we utilize the Genetic Algorithm (Harada and Alba, 2020) as the searching implementation. We formulate this search objective as Equation 1.

$$S^* = \arg\max_S \text{Accuracy}(\theta(\mathcal{R}(S), X), Y) \quad \text{s.t.} \quad \sum s_i = P \tag{1}$$

where the $\theta(\mathcal{R}(S), X)$ indicates that the LLM θ works with a router $\mathcal{R}$ assigned a sparsity ratio configuration S and is fed with input X for prediction.

Static router is one that uses a fixed rule to select tokens with no learnable modules. Inspired by the work (Xiao et al., 2023) that indicates that the initial token plays a key role in the window attention of LLMs for long-text inference. Therefore, we construct a static router in following rule: Ranking the importance of tokens x at the i-th block as $\{x_0^i, x_{L-1}^i, x_{L-2}^i, ..., x_1^i\}$, and the top-$k$ unimportant tokens would be skipped to meet the sparsity requirement and passed directly to the next block. We have found that the first and the last few tokens in a sequence are typically important for model performance, this means that the first token and the last few tokens will be prioritized for computation in static router. This static router provides a good preliminary pruning configuration, which serves as the starting point. This design is simple yet effective, surpassing existing state-of-the-art (SOTA) methods, as shown in Table 1 (FTP-static). More results of the static router can be found in Appendix A.4.1. Note that, FTP-static in Section 4 is employed with the sparsity searched in step 1.

3.2.3 Learnable Dynamic Router

After obtaining the sparsity allocation, we use this allocation on the other lightweight learnable router (a two-layer MLP) with dynamic decisions (i.e., select tokens at runtime) to train. Recent router-based methods (Raposo et al., 2024) leverage hidden states to predict pruning configurations. However, we argue that this approach is not suitable for LLM pruning, as hidden states are high-dimensional abstract features that require heavy network fitting, leading to increased training and inference costs, and potentially degrading generalization. Therefore, we leverage four factors as router input, these factors are weakly correlated with token hidden states but are related to token redundancy. Here we formulate the input of dynamic router, we define sequence of length L for the i-th block,

$$H^i = \{\boldsymbol{h}_j^i \mid j \in \mathbb{N}, 1 \le j \le L\} = \{(p^j, s_a^j, r_a^j, s_r^i) \mid j \in \mathbb{N}, 1 \le j \le L\}, \tag{2}$$

where $\boldsymbol{h}_j^i$ is the hybrid input vector of the j-th token in the sequence, which is a 4-dimensional vector that includes the token position p^j, absolute attention scores s_a^j, relative attention score rank r_a^j, and sparsity requirements s_r^i of the i-th block.

Previous work (Xiao et al., 2023) has revealed that tokens at different positions in a sequence are of varying importance. Thus, we adopt the position p^j as factor. Additionally, attention scores s_a^j represent the degree of association between a token and other tokens, making it a crucial pruning factor.

If a token is highly associated with others, it can be replaced, indicating its redundancy. During the training stage, we maintain an attention score table to record the latest attention scores of all tokens and update the scores of the computed tokens in sequences. Moreover, we introduce relative attention score rank r_a^j to measure the relative importance of tokens. Sparsity requirements s_r^i enables our dynamic router to allocate pruning configurations from a global perspective.

Dynamic router produces a 2-dimensional score $\boldsymbol{o}_j^i$ would be normalized by a softmax operation and represents the probability of computing the j-th input token of the i-th block in the forward process. This score is processed by the $\arg\max$ operation and discretize it into a gate $\boldsymbol{g} \in \{0, 1\}$. This gate is used to control whether to skip ($\boldsymbol{g} = 0$) or compute ($\boldsymbol{g} = 1$) the token in the block. However, the $\arg\max$ operation is non-differentiable. Therefore, we utilize the Straight-Through (ST) Estimator (Jang et al., 2016) during the training phase to approximate the real gradient $\nabla_\theta \boldsymbol{g}$ with the gradient of the soft prediction $\nabla_\theta \boldsymbol{s}^i$. During training or inference, the proposed inputs of all tokens from a block are fed into the router to obtain the predicted importance scores for all the tokens. Note that all blocks share the same router, enhancing the router's generalization ability. Dynamic router training in step 2 is named "**FTP**" in Section 4.

3.2.4 Router Training

We design three crucial losses to enhance the learnable router's ability to learn token selection: guide, sparsity constraint, and knowledge distillation losses. Specifically, to accelerate the training process of the learnable router, we introduce the guide loss as a warm-up constraint at the beginning. The guide loss (binary cross-entropy (BCE)) leverages the predictions from static router to guide the learnable router in producing reasonable predictions during the early stages of training. The sparsity constraint loss is employed to align the predicted sparsity with the required sparsity of the blocks. The predicted sparsity ratio for each block is obtained via the summation of skipping tokens based on the gate $\boldsymbol{g}$. The constraint loss imposes a penalty on the router only if the predicted sparsity ratio is less than the assigned sparsity ratio as follows:

$$\mathcal{L}_s = \sum_i^N (\max(s_r^i - \frac{1}{L}\sum_j^L (1 - \boldsymbol{g}_j^i), 0)) \quad (3)$$

where N is the number of the LLM's blocks, $\boldsymbol{g}^i$ is the predicted discrete state of the token sequence in the i-th block of the LLM and s_r^i is the required sparsity ratio of that block.

Moreover, the knowledge distillation loss is utilized to improve the accuracy of the pruned model by aligning the predictions between the original and pruned models using mean squared error (MSE) loss. We apply the distillation loss to the hidden state of the last block output. These losses are combined to optimize the learnable router with different weights, resulting in the final loss as follows:

$$\mathcal{L}(\mathrm{X}, \mathrm{Y}; \theta, \mathcal{R}) = \lambda_d \mathcal{L}_d + \lambda_s \mathcal{L}_s + \lambda_g \mathcal{L}_g \quad (4)$$

where λ_d, λ_s and λ_g are the loss weights of distillation loss $\mathcal{L}_d$, sparsity constraint loss $\mathcal{L}_s$ and guide loss $\mathcal{L}_g$, respectively. θ and $\mathcal{R}$ denote the parameters of the LLM and learnable router, respectively. The loss weight L_g is initialized to 1 and gradually decays to 0 when it progresses to halfway through the total iterations.

3.2.5 Learnable One-Pass Router

The dynamic router selects tokens at runtime, which somewhat impacts the actual speedup. Additionally, it does not support batch inference due to varying token pruning patterns across different inputs. To address these issues, we propose a learnable one-pass strategy that enables batch inference and significantly boosts actual speedup with minimal runtime overhead, albeit with a slight trade-off in precision. Maintaining the attention scores table in the dynamic router may require complex hardware modifications. Additionally, it poses a real-time update bottleneck. Therefore, we discard all attention score-related inputs to alleviate the router's computational burden. Instead, we introduce a learnable position embedding PE to explore suitable token selections. We reconstruct the token factors like:

$$\begin{aligned} H^i = \{\boldsymbol{h}_j^i \mid j \in \mathbb{N}, 1 \le j \le L\} = \\ \{(p^j, PE, s_r^i) \mid j \in \mathbb{N}, 1 \le j \le L\}, \end{aligned} \quad (5)$$

Position embedding requires adjustment only once during the training phase in step 2 and does not need to be modified during runtime. Specifically, router decisions are made only once per batch job. Compared to dynamic routers, this approach is more straightforward and supports batch inference.

Existing works (e.g., BlockPruner (Zhong et al., 2024)) have indicated that there are differences in importance between attention modules (MHA) and

Model	Method	Ratio (%)	ARC-c	ARC-e	HellaSwag	MMLU	WinoGrande	Avg. Acc.	Avg. Percentage (%)
LLaMA2-7B	Dense	0	46.16	74.54	75.99	45.39	69.06	62.25	100
	LaCo	21.02	35.84	55.39	54.08	-	60.46	51.44	77.67
	RM	21.02	22.53	34.43	29.22	-	49.25	33.86	51.19
	LLMPruner	27.0	-	-	60.21	23.33	-	41.77	65.32
	SliceGPT	21.45	37.12	63.64	56.04	-	59.91	54.18	81.57
	ShortGPT	27.0	32.68	48.61	56.15	44.51	64.33	49.25	80.22
	BlockPruner	20.99	35.92	61.20	66.04	-	64.09	56.81	84.91
	FTP-static	22.0	44.88	72.31	72.66	45.83	**69.53**	61.04	98.30
	FTP-onepass	22.0	45.90	73.15	74.21	46.41	69.85	**61.90**	**99.73**
	FTP	22.0	**45.31**	**73.06**	**74.46**	**46.15**	69.22	61.64	99.21
	FTP-onepass	30.0	43.09	70.37	67.24	45.76	68.35	58.96	95.21
	FTP	30.0	43.65	72.31	67.37	46.07	68.97	59.67	96.32
LLaMA2-13B	Dense	0	49.23	77.36	79.36	54.94	72.14	65.81	100
	LaCo	24.37	34.56	54.34	60.44	-	59.27	52.15	74.69
	RM	24.37	41.98	66.12	66.80	-	66.61	60.38	86.81
	SliceGPT	21.52	42.41	68.52	60.71	-	65.59	59.31	85.53
	ShortGPT	24.60	42.92	63.55	69.27	53.83	69.85	59.88	90.28
	BlockPruner	24.31	40.53	63.55	71.93	-	70.40	61.60	88.18
	FTP-static	25.0	47.95	74.58	76.65	54.51	71.19	64.98	97.66
	FTP-onepass	25.0	48.38	74.75	75.99	54.47	71.67	65.53	98.63
	FTP	25.0	**48.98**	**75.55**	**77.49**	**54.56**	**72.22**	**65.76**	**98.84**
	FTP-onepass	30.0	48.55	75.93	75.29	54.28	72.06	65.22	98.04
	FTP	30.0	48.38	74.75	75.99	54.47	71.67	65.05	97.83
Qwen1.5-7B	Dense	0	42.66	62.16	76.92	60.52	66.46	61.74	100
	LaCo	20.97	32.85	46.89	56.35	-	58.64	48.68	78.48
	RM	20.97	28.58	54.17	42.00	-	49.88	43.66	70.95
	ShortGPT	21.88	33.79	48.44	63.09	49.54	60.93	51.16	82.54
	BlockPruner	21.83	33.02	53.49	57.29	-	56.99	50.20	80.92
	FTP-static	22.0	43.52	62.71	71.89	60.26	65.19	60.71	98.80
	FTP-onepass	22.0	43.65	62.13	73.53	60.12	66.29	61.14	99.39
	FTP	22.0	**43.69**	**62.81**	**74.02**	**60.86**	**67.32**	**61.74**	**100.03**
	FTP-onepass	30.0	40.15	58.86	67.95	60.23	65.26	58.16	95.23
	FTP	30.0	40.96	59.60	68.47	60.77	65.67	59.09	96.03

Table 1: **Downstream tasks performance.** FTP variants surpass all the competitors under comparable sparsity constraints. MMLU uses a 5-shot evaluation, and other tasks are all 0-shot.

FFN within a block. Therefore we apply twin one-pass learnable routers for ATTN and FFN modules respectively. Note that twin routers share the same sparsity but do not share position embeddings. One-pass router training in step 2 is referred to as "**FTP-onepass**" in Section 4.

4 Experiments

4.1 Experimental Settings

Models and Baselines. We apply FTP to LLaMA2-7B, LLaMA2-13B (Touvron et al., 2023), LLaMA3-8B (Dubey et al., 2024), and Qwen1.5-7B (Bai et al., 2023), with initialization by non-instruct-tuning pretrained weights. To assess the effectiveness of our approach, we benchmark it against state-of-the-art structured pruning techniques, including LLMPruner (Ma et al., 2023), SliceGPT (Ashkboos et al., 2024), LaCo (Yang et al., 2024), ShortGPT (Men et al., 2024), Relative Magnitude (RM) (Samragh et al., 2023), and BlockPruner (Zhong et al., 2024). LLMPruner and SliceGPT primarily target pruning through reductions in embedding dimensions, whereas LaCo, ShortGPT, RM, and BlockPruner focus on depth pruning strategies.

Datasets and Implement Details. Following previous works, we use Alpaca (Taori et al., 2023) as training set, and evaluate on these well-known benchmarks: HellaSwag (Zellers et al., 2019), MMLU (Hendrycks et al., 2020), ARC-easy, ARC-challenge (Clark et al., 2018), WinoGrande (Sakaguchi et al., 2021); specifically, we utilize the WinoGrande to search the sparsity ratios in step 1. We report the accuracies together with average accuracy retention percentages on these benchmarks. In step 2, we train the learnable routers 10,000 iterations on 7/8B, and 50,000 on 13B models, both with batch size of 1. We utilize the AdamW optimizer with learning rate of 1e-4. All experiments are conducted on a single AMD MI250 GPU with 64GB of memory, taking approximately 1 hour for the router training phase, and 2 hours for sparsity searching. We provide more training and inference details in Appendix A.1. All blocks share the same router in FTP. We provide more ablations on Appendix A.3 compared with non-shared and recurrent strategies.

4.2 Main Results

Compare with SOTA Methods. As shown in Table 1, FTP and its variants demonstrate superior performance across public benchmarks, covering various tasks such as reasoning, language understanding, knowledge retention, and examination capacity. Our FTP method consistently outperforms other SOTA pruning methods, such as BlockPruner and ShortGPT, across models like LLaMA2-7B, LLaMA2-13B, and Qwen1.5-7B. For example, at a 22% sparsity ratio, FTP achieves 99.21% on

Model	Method	Ratio (%)	ARC-c	ARC-e	HellaSwag	MMLU	WinoGrande	Avg. Acc.	Avg. Percentage (%)
LLaMA2-7B	Dense	0	46.16	74.54	75.99	45.39	69.06	62.25	100
	FTP-onepass	30	43.09	70.37	67.24	45.76	68.35	58.96	95.21
	FTP	30	**43.65**	**72.31**	**67.37**	46.07	**68.97**	**59.67**	**96.32**
	FTP-onepass	40	41.89	67.89	62.53	46.17	64.72	56.64	91.91
	FTP	40	40.02	70.01	62.67	**46.56**	66.03	57.06	92.26
LLaMA2-13B	Dense	0	49.23	77.36	79.36	54.94	72.14	65.81	100
	FTP-onepass	30	48.55	75.93	75.29	54.28	72.06	**65.22**	**98.04**
	FTP	30	**48.38**	**74.75**	**75.99**	**54.47**	**71.67**	65.05	97.83
	FTP-onepass	40	45.99	72.26	64.02	54.61	69.69	61.31	92.70
	FTP	40	45.22	70.88	66.50	54.57	70.40	61.51	92.84
LLaMA3-8B	Dense	0	53.33	77.69	79.19	65.28	72.85	69.67	100
	FTP-onepass	30	48.15	73.31	62.21	62.88	68.81	63.07	91.10
	FTP	30	**48.63**	**73.36**	**62.41**	**64.29**	**69.69**	**63.68**	**91.71**
	FTP-onepass	40	43.21	67.05	54.33	62.99	68.77	59.27	85.37
	FTP	40	43.00	67.17	54.89	63.72	69.30	59.62	85.83
Qwen1.5-7B	Dense	0	42.66	62.16	76.92	60.52	66.46	61.74	100
	FTP-onepass	30	40.15	58.86	67.95	60.23	65.26	58.16	95.23
	FTP	30	**40.96**	**59.60**	**68.47**	**60.77**	**65.67**	**59.09**	**96.03**
	FTP-onepass	40	36.09	52.88	62.61	60.03	58.87	54.10	87.77
	FTP	40	36.15	53.03	62.59	60.83	59.04	54.32	88.15

Table 2: **Various sparsity ratios.** FTP still maintains relatively roubst performance at higher sparsity ratio (40%), and is even better than BlockPruner, ShortGPT and other methods on LLaMA2-7B with a sparsity ratio of 22%.

Method	ARC-c	MMLU	Avg. Percentage
Uniform	26.02	40.50	72.80
BI score based	34.81	45.29	87.60
Sparsity scheduler w.o step 3	40.96	45.67	94.68
Sparsity scheduler	**43.65**	**46.07**	**98.03**

Table 3: Sparsity scheduler ablations on LLaMA2-7B with 30% sparsity.

LLaMA2-7B, compared to BlockPruner's 84.91%. FTP surpasses other methods even at a higher sparsity ratio (30%). On larger model (e.g., LLaMA2-13B), FTP achieves an average accuracy of 97.83% at 30% sparsity, significantly outperforming BlockPruner (88.18%) and ShortGPT (90.28%). FTP-onepass does not depend on varying inter-layer features such as attention scores, thus requiring only onepass token selections for each layer before inference starts. In comparison with FTP, FTP-onepass enables a more significant speedup, with a marginally diminished performance in some models (e.g., Llama2-13B and Qwen1.5-7B), yet it remains superior to other methods. Moreover, FTP-static (without learnable modules) performs well, owing to the effectiveness of our sparsity scheduler. These results highlight the remarkable ability of our proposal, effectively reducing inference costs while preserving performance across diverse tasks.

Higher Sparsity on Different Models. In Table 2, we examine the impact of higher sparsity. At a 40% sparsity ratio, FTP maintains an impressive performance range of 85% to 93% across various models and benchmarks. Specifically, on LLaMA2-7B, FTP achieves 92.26% at 40% sparsity, significantly outperforming BlockPruner (84.91%) at 22% sparsity and ShortGPT (80.22%) at 27% sparsity. This indicates that FTP not only manages higher sparsity more effectively but also surpasses other methods even under more conservative pruning settings. Even when reducing the number of tokens by 40%, FTP's performance still remains strong compared to other methods. Among of most cases, FTP performs better than onepass variant, but in Llama2-13B with 30% sparsity, FTP-onepass has a slight advantage. Both FTP and FTP-onepass demonstrate the consistent high performance across different model sizes (i.e., LLaMA2-7B and 13B). It indicates that our proposal is highly scalable and reliable for deployment in larger models where computational efficiency is critical.

4.3 Ablation Study

Impact of Sparsity Scheduler. In Section 3.1.1, we highlight the varying sensitivity of blocks at different depths to token pruning and introduce a GA-based sparsity scheduler to determine the optimal sparsity ratios for all blocks in the LLM, while meeting the overall pruning requirement. Table 3 illustrates the ablations of allocating the sparsity to each layer. The uniform variant is a strategy to allocate the average sparsity to each layer, and we can observe that it is the worst. Then, we compare the strategy via the BI score (Men et al., 2024), which shows a performance gap of about 10% when compared to our baseline. The 4th row shows the effectiveness of the post-tuning (i.e., step 3 searching) via the learnable router.

Impact of Designed Input. The core idea of our method is to rank tokens based on their predicted importance and skip the less significant ones within a block. The input design for the dynamic router plays a crucial role in determining the outcome. We compare various inputs in Table 4 to illustrate the effectiveness of our designed input. Previous work (Raposo et al., 2024) uses hidden states from each block as the sole feature for the router's decision-making. Thus, we directly compare the

hidden states as input with our designed input. As shown in Table 4, our designed input significantly outperforms both the hidden states and combinations that include hidden states. Additionally, we conduct ablations to assess the individual elements of the designed input, confirming the importance of all components.

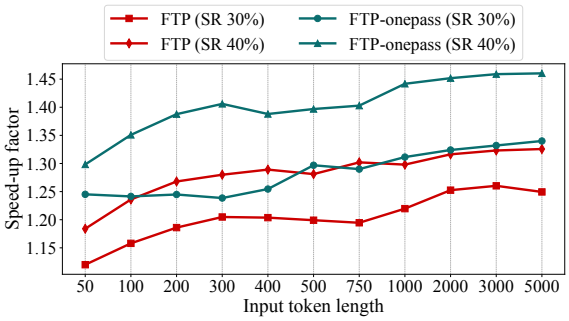

Figure 3: Inference speedup of FTP and FTP-onepass in LLaMA2-7B on different settings including different sparsity ratios and input token lengths.

4.4 More Analysis

4.4.1 Inference Speedup

The forward computation of transformers constitutes a large portion of the inference time, whereas the computational cost of our global router—comprised of just a two-layer MLP—is only a small fraction of the overall inference time. We report the average relative speedup based on throughput (tokens processed per second), and further show the details of different input token lengths via the Alpaca dataset as input prompts on LLaMA2-7B, as illustrated in Figure 3. FTP-onepass is a variant aimed at higher speedup with a slight performance loss. We can observe that FTP-onepass accelerates better than FTP. However, both FTP and FTP-onepass achieve higher than 1.2× actual speedup when the token length is larger than 300. Compared to LLMPruner (1.14×) (Ma et al., 2023) and ShortGPT (1.24×) (Men et al., 2024), our approach (FTP at 1.23×, FTP-onepass at 1.28×) outperforms in terms of average speedup. The computational advantage of our approach grows as the sequence length increases. When the router selects tokens to skip within a block, the length of the sequence involved in attention is reduced, thereby decreasing computational complexity at a quadratic rate. Also, the feed-forward network (FFN) costs are eliminated at the same time. With the increasing importance of ultra-long context, our approach gains a significant advantage as sequence lengths grow.

Method	ARC-c	MMLU	Avg. Percentage
Dense	46.16	45.39	100
Hidden states	33.87	44.78	86.02
DI w. hidden states	34.57	44.99	87.01
DI w.o. position	30.72	44.59	82.39
DI w.o. attntion score	41.21	45.43	94.68
DI w.o. attntion rank	42.13	45.15	95.37
DI w.o. sparsity	38.51	45.79	92.15
DI	**43.65**	**46.07**	**98.03**

Table 4: Dynamic router input ablations on LLaMA2-7B with 30% sparsity. DI indicates our designed input.

4.4.2 Compatible with KV Cache

Key and value (KV) cache stores previous key-value pairs for each token across blocks, enabling faster computation during inference, improving efficiency. The primary computational cost shifts to focus mainly on the last token in the sequence, which includes FFN operations and attention computations with other tokens in the sequence. FTP does not impose a sparsity constraint on the last token in the depth dimension and prioritizes the forward computation of the last token. Thus, the KV cache reduces the acceleration benefits gained from token-wise pruning on the entire token sequence. Here, we design a strategy to overcome this challenge. If the router's predicted score for the last token exceeds a threshold (set to 0.5 in the experiment), it performs computation within the block; otherwise, it is skipped. We provide more details and reports in Appendix A.8. The pruning results show virtually no performance loss.

5 Conclusion

In this paper, we present a fine-grained token pruning framework for LLMs, which outperforms other SOTA LLM pruning methods with low training costs for learnable routers. Our proposed token-wise pruning framework is structured around three key steps: first, we conduct an initial sparsity search utilizing a static router to determine the appropriate sparsity allocation. Next, we train a dynamic router informed by our four proposed factors and three distinct loss functions. Finally, we fine-tune the sparsity scheduler using the trained router. Furthermore, we introduce a one-pass learnable router designed for batch inference and enhanced acceleration. Comprehensive experiments underscore the importance of each component in improving the overall effectiveness of our approach. The results reveal that our method significantly outperforms other SOTA methods, further demonstrating its superiority.

6 Limitations

While FTP demonstrates strong performance on widely-used models, its applicability to extremely large models, such as those exceeding 100B parameters, remains unexplored. Additionally, FTP's effectiveness in long-context tasks, such as retrieval-augmented generation (RAG) or multi-document QA, has yet to be fully evaluated. The robustness of pruned models against adversarial inputs or fairness-sensitive tasks also remains unexamined, with the potential for token-wise pruning to inadvertently amplify biases or reduce resilience. These aspects emphasize the need for further research into scalability, long-context reliability, and ethical implications of token-wise pruning.

References

Yash Akhauri, Ahmed F AbouElhamayed, Jordan Dotzel, Zhiru Zhang, Alexander M Rush, Safeen Huda, and Mohamed S Abdelfattah. 2024. Shadowllm: Predictor-based contextual sparsity for large language models. *arXiv preprint arXiv:2406.16635*.

Yongqi An, Xu Zhao, Tao Yu, Ming Tang, and Jinqiao Wang. 2024. Fluctuation-based adaptive structured pruning for large language models. In *Proc. AAAI*, pages 10865–10873.

Saleh Ashkboos, Maximilian L Croci, Marcelo Gennari do Nascimento, Torsten Hoefler, and James Hensman. 2024. Slicegpt: Compress large language models by deleting rows and columns. *arXiv preprint arXiv:2401.15024*.

Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenbin Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, Keming Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xingxuan Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. 2023. Qwen technical report. *arXiv preprint arXiv:2309.16609*.

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training text encoders as discriminators rather than generators. In *ICLR*.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. 2018. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2024. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. 2024. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*.

Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. 2020. Depth-adaptive transformer. In *ICLR 2020-Eighth International Conference on Learning Representations*, pages 1–14.

Elias Frantar and Dan Alistarh. 2023. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *Proc. ICML*, pages 10323–10337. PMLR.

Shangqian Gao, Feihu Huang, Jian Pei, and Heng Huang. 2020. Discrete model compression with resource constraint for deep neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 1899–1908.

Yuxian Gu, Li Dong, Furu Wei, and Minlie Huang. 2024. Minillm: Knowledge distillation of large language models. In *The Twelfth International Conference on Learning Representations*.

Tomohiro Harada and Enrique Alba. 2020. Parallel genetic algorithms: a useful survey. *ACM Computing Surveys (CSUR)*, 53(4):1–39.

Tahmid Hasan, Abhik Bhattacharjee, Md Saiful Islam, Kazi Mubasshir, Yuan-Fang Li, Yong-Bin Kang, M Sohel Rahman, and Rifat Shahriyar. 2021. Xl-sum: Large-scale multilingual abstractive summarization for 44 languages. In *Findings of the Association for Computational Linguistics: ACL-IJCNLP 2021*, pages 4693–4703.

Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. 2020. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*.

Yukun Huang, Yanda Chen, Zhou Yu, and Kathleen McKeown. 2022. In-context learning distillation: Transferring few-shot learning ability of pre-trained language models. *arXiv preprint arXiv:2212.10670*.

Eric Jang, Shixiang Gu, and Ben Poole. 2016. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*.

Huiqiang Jiang, Qianhui Wu, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023a. Llmlingua: Compressing prompts for accelerated inference of large language models. *arXiv preprint arXiv:2310.05736*.

Huiqiang Jiang, Qianhui Wu, Xufang Luo, Dongsheng Li, Chin-Yew Lin, Yuqing Yang, and Lili Qiu. 2023b. Longllmlingua: Accelerating and enhancing llms in long context scenarios via prompt compression. *arXiv preprint arXiv:2310.06839*.

Hoyoun Jung and Kyung-Joong Kim. 2024. Discrete prompt compression with reinforcement learning. *IEEE Access*.

Yixiao Li, Yifan Yu, Qingru Zhang, Chen Liang, Pengcheng He, Weizhu Chen, and Tuo Zhao. 2023a. Losparse: Structured compression of large language models based on low-rank and sparse approximation. In *International Conference on Machine Learning*, pages 20336–20350. PMLR.

Yucheng Li, Bo Dong, Chenghua Lin, and Frank Guerin. 2023b. Compressing context to enhance inference efficiency of large language models. *arXiv preprint arXiv:2310.06201*.

Songwei Liu, Chao Zeng, Lianqiang Li, Chenqian Yan, Lean Fu, Xing Mei, and Fangmin Chen. 2024. Foldgpt: Simple and effective large language model compression scheme. *arXiv preprint arXiv:2407.00928*.

Zhuang Liu, Zhiqiu Xu, Hung-Ju Wang, Trevor Darrell, and Evan Shelhamer. 2021. Anytime dense prediction with confidence adaptivity. *arXiv preprint arXiv:2104.00749*.

Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. 2023. Deja vu: Contextual sparsity for efficient llms at inference time. In *Proc. ICML*, pages 22137–22176. PMLR.

Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2023. Llm-pruner: On the structural pruning of large language models. *Advances in neural information processing systems*, 36:21702–21720.

Xin Men, Mingyu Xu, Qingyu Zhang, Bingning Wang, Hongyu Lin, Yaojie Lu, Xianpei Han, and Weipeng Chen. 2024. Shortgpt: Layers in large language models are more redundant than you expect. *arXiv preprint arXiv:2403.03853*.

Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. 2024. Large language models: A survey. *arXiv preprint arXiv:2402.06196*.

David Raposo, Sam Ritter, Blake Richards, Timothy Lillicrap, Peter Conway Humphreys, and Adam Santoro. 2024. Mixture-of-depths: Dynamically allocating compute in transformer-based language models. *arXiv preprint arXiv:2404.02258*.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. 2021. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106.

Mohammad Samragh, Mehrdad Farajtabar, Sachin Mehta, Raviteja Vemulapalli, Fartash Faghri, Devang Naik, Oncel Tuzel, and Mohammad Rastegari. 2023. Weight subcloning: direct initialization of transformers using larger pretrained ones. *arXiv preprint arXiv:2312.09299*.

Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Tran, Yi Tay, and Donald Metzler. 2022. Confident adaptive language modeling. *Advances in Neural Information Processing Systems*, 35:17456–17472.

Mingjie Sun, Zhuang Liu, Anna Bair, and J Zico Kolter. 2023. A simple and effective pruning approach for large language models. *arXiv preprint arXiv:2306.11695*.

Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B Hashimoto. 2023. Stanford alpaca: An instruction-following llama model.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Wenxiao Wang, Wei Chen, Yicong Luo, Yongliu Long, Zhengkai Lin, Liye Zhang, Binbin Lin, Deng Cai, and Xiaofei He. 2024. Model compression and efficient inference for large language models: A survey. *arXiv preprint arXiv:2402.09748*.

Mengzhou Xia, Tianyu Gao, Zhiyuan Zeng, and Danqi Chen. 2023. Sheared llama: Accelerating language model pre-training via structured pruning. *arXiv preprint arXiv:2310.06694*.

Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2023. Efficient streaming language models with attention sinks. *arXiv preprint arXiv:2309.17453*.

Yifei Yang, Zouying Cao, and Hai Zhao. 2024. Laco: Large language model pruning via layer collapse. *arXiv preprint arXiv:2402.11187*.

Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in Neural Information Processing Systems*, 35:27168–27183.

Fan Yin, Jesse Vig, Philippe Laban, Shafiq Joty, Caiming Xiong, and Chien-Sheng Jason Wu. 2023. Did you read the instructions? rethinking the effectiveness of task definitions in instruction learning. *arXiv preprint arXiv:2306.01150*.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. 2019. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*.

Bowen Zhao, Hannaneh Hajishirzi, and Qingqing Cao. 2024. Apt: Adaptive pruning and tuning pretrained language models for efficient training and inference. *arXiv preprint arXiv:2401.12200*.

Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. 2023. A survey of large language models. *arXiv preprint arXiv:2303.18223*.

Longguang Zhong, Fanqi Wan, Ruijun Chen, Xiaojun Quan, and Liangzhi Li. 2024. Blockpruner: Fine-grained pruning for large language models. *arXiv preprint arXiv:2406.10594*.

A Appendix

A.1 Implementation Details

Learnable token router consists of a two-layer MLP, with a hidden size of 64 and an output size of 2. In all experiments, the hyperparameters of the loss function (i.e., λ_d, λ_s, and λ_g) are all initially set to 1. During the sparsity optimization (i.e., in step1 and step3), we use a population of 50 sparsity configurations, with 10 generations and a mutation probability of 0.2. The sparsity optimization process takes approximately 2 hours. All experiments are employed on ROCm 6.1, Torch 2.3, and Torchtune 2.0. We utilize lm-eval to evaluate the benchmarks. For consistency and fairness, float32 precision is uniformly used during both training and testing. All benchmarks are evaluated using the "acc norm" score by default, and the average percentage reflects the average score across all benchmarks (i.e., pruned/dense model performance).

A.2 More Introductions of Router Workflow

Each transformer block consists of two main components: multi-head attention (MHA) and feed-forward network (FFN) as depicted in Figure 4. The attention mechanism is applied multiple times in parallel among the token sequence, allowing the model to focus on different parts of the sequence at different positions. The attention computation is calculated as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d}}\right)V \quad (6)$$

where $Q, K, V \in \mathbb{R}^{L\times d}$ are the query, key, and value matrices of the input sequence, L is the sequence length, and d is the dimension of the key. The result of the attention mechanism for each token is a weighted sum of all the tokens in the sequence. After applying attention, the model passes the output through a feed-forward network that consists of two fully connected layers with a non-linear layer. The forward computation of the i-th block in a transformer can be expressed as follows:

$$\begin{aligned} X_i' &= \text{MHA}(\text{LN}(X_i)) + X_i \\ X_{i+1} &= \text{FFN}(\text{LN}(X_i')) + X_i' \end{aligned} \quad (7)$$

where $X_i \in \mathbb{R}^{L\times d}$ is the input of the i-th block, LN is layer normalization applied to the inputs, MHA is the multi-head attention mechanism, and FFN is the feed-forward network. The computation complexity of a transformer block is mostly dominated

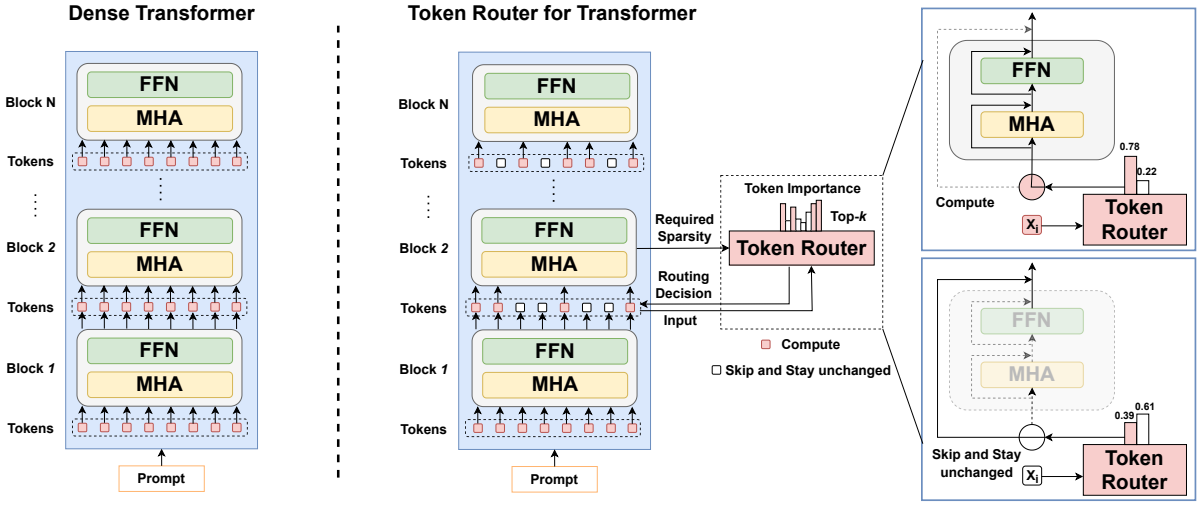

Figure 4: Overview of LLM structure and router workflow. (Left) Dense Transformer where all tokens are processed in every block. (Middle) Token router for transformer, which dynamically selects tokens to compute or skip based on their importance and block-wise sparsity at each block. (Right) A detailed view of how the token router uses token importance features to make binary decisions (compute or skip) for each token within a block.

Method	ARC-c	MMLU	Avg. Percentage
Recurrent	40.23	45.53	93.73
Non-shared	38.21	43.69	89.52
Global shared	**43.65**	**46.07**	**98.03**

Table 5: Different global routing designs on LLaMA2-7B with 30% sparsity.

by two components: the MHA and the FFN. The MHA has a complexity of $O(L^2 \cdot d)$, where L is the sequence length and d is the hidden dimension, due to the pairwise attention computation across tokens. The FFN, which processes each token independently, has a complexity of $O(L \cdot d^2)$. Therefore, the overall complexity of a transformer block is $O(L^2 \cdot d + L \cdot d^2)$, where the quadratic dependency on L makes attention particularly expensive for long sequences (Clark et al., 2020). Transformers capture contextual information and predict the next token by leveraging the effectiveness of the attention mechanism. However, the computational cost of large language models (LLMs) is extremely high. The attention mechanism in a transformer block has a complexity of $O(L^2 \cdot d + L \cdot d^2)$, meaning the FLOPs of an LLM grow exponentially with the number of tokens. Token routing, as shown in 4, a mechanism that selectively permits only a subset of tokens to engage in the computation of each transformer block, offers an effective strategy to reduce the sequence length processed by each block, thereby significantly lowering the overall computational cost.

A.3 Global Routing Implementations.

In FTP and its variants, all blocks share a same router. Here we provide a further study on different implementations in Table 5. First is a recurrent approach, we use a LSTM model, treating each block as a timestep and predicting decisions based on the designed input, along with the previous block's decision and token importance. Second is a non-shared approach, we assign routers for each block and following the same FTP steps. We can observer that the shared approach achieves the best performance. Global sharing approach learns a more comprehensive view of block interdependences than others.

A.4 More Analysis on FTP-static

FTP-static is a well-performed variant, and benefited from the step 1 sparsity searching. Here we provide more ablations on it.

A.4.1 Compared to the Random Selection.

After obtaining the sparsity configuration under the overall sparsity ratio of 30%, we compare the performance between random token selection and FTP-static, as shown in Table 6. In the random selection, we randomly choose the same number of tokens for skipping, while keeping within its sparsity ratio limits. The random selection is cross-validated 5 times, with results averaged across trials. Notably, FTP-static outperforms random selection, highlighting the critical importance of to-

Method	Priority Retained Token ID	ARC-c	MMLU	Avg. Percentage
Dense	-	46.16	45.39	100
Random selection	-	28.92	34.16	68.96
Random selection	1st	32.17	34.84	73.22
FTP-static*	2nd	31.91	34.54	72.61
FTP-static w. perturbation	1st	40.19	43.95	91.95
FTP-static	1st	43.26	46.09	97.63

Table 6: Comparisons of different static routers with 30% sparsity.

Model	Results (Block ID: Sparsity ratio)
LLama-2-7B (Initial)	16: 0.2596, 17: 0.3987, 18: 0.4808, 19: 0.5481, 20: 0.5451, 21: 0.6642, 22: 0.682, 23: 0.7337, 24: 0.7589, 25: 0.7973, 26: 0.7766, 27: 0.7996, 28: 0.7862, 29: 0.7729, 30: 0.5962
LLama-2-7B (Finetuned)	13: 0.1708, 14: 0.1904, 15: 0.1912, 16: 0.1839, 17: 0.3372, 18: 0.4277, 19: 0.5019, 20: 0.4986, 21: 0.6299, 22: 0.6494, 23: 0.7065, 24: 0.7342, 25: 0.7766, 26: 0.7538, 27: 0.7791, 28: 0.7644, 29: 0.7497, 30: 0.5548
LLama2-13B (Initial)	11: 0.0693, 12: 0.1014, 13: 0.1182, 14: 0.1477, 15: 0.1595, 16: 0.1164, 17: 0.1401, 18: 0.1283 , 19: 0.2169, 20: 0.2363, 21: 0.3318, 22: 0.3019, 23: 0.4579, 24: 0.7259, 25: 0.659, 26: 0.5836, 27: 0.5282, 28: 0.5267, 29: 0.6702, 30: 0.5661, 31: 0.658, 32: 0.6851, 33: 0.6721, 34: 0.6825, 35: 0.6053, 36: 0.8427, 37: 0.6111, 38: 0.5934
LLama2-13B (Finetuned)	12: 0.0171, 13: 0.0148, 14: 0.0476, 15: 0.0795, 16: 0.1144, 17: 0.1467, 18: 0.2193, 19: 0.4022, 20: 0.4383, 21: 0.4738, 22: 0.5108, 23: 0.6531, 24: 0.5355, 25: 0.597, 26: 0.5807, 27: 0.599, 28: 0.627, 29: 0.6175, 30: 0.6068, 31: 0.6059, 32: 0.6012, 33: 0.6019, 34: 0.613, 35: 0.6007, 36: 0.6127, 37: 0.6111, 38: 0.4786

Table 7: Block-wise sparsity ratios obtained by sparsity scheduler for overall 30% sparsity.

ken position in selection. Despite this, random selection achieves nearly 70% performance, due to the underlying block configuration derived from the search process. This underscores the significance of the sparsity scheduler in maintaining performance.

A.4.2 Priority Token Retained.

We conduct a further investigation into the token importance. A comparison between the random token selection approaches in rows 3 and 4 reveals that performance improves when the first token is retained. This is further supported by the results in row 5 of Table 6, where the firstly retaining of the second token leads to a performance drop in the FTP-static. These findings highlight the critical importance of the first token selection. Furthermore, we observe a significant drop in performance when introducing random perturbations into the final static decision process. Specifically, we randomly select 10% of tokens from the sequence (take 5% from skip tokens, and 5% from updated tokens) and swap their decision flags to maintain the block sparsity ratio. This highlights the sensitivity of token selection within the model. Nevertheless, our dynamic FTP outperforms the static version, demonstrating the robustness and efficacy of the dynamic routing mechanism.

A.5 Attention Score

Define the $Q \in \mathbb{R}^{L\times d\times N}$ and $K \in \mathbb{R}^{L\times d\times N}$, where the L is the sequence lengths of the query and key in attention. The N is the head number of the multi-head attention. The attention score $\mathrm{A}_s \in \mathbb{R}^{L}$ can be formulated as following:

$$\begin{aligned} \mathrm{A} &= \frac{QK^T}{\sqrt{d}} \\ \mathrm{A}_s &= \frac{1}{L}\sum_{j=1}^{L} \mathrm{A}_{i,j}, i = 1, 2, \ldots, L \end{aligned} \tag{8}$$

After obtaining the $A \in \mathbb{R}^{L\times L\times N}$, we execute a mean operation in head dimension N, then we obtain the A_s by a mean operation in the dimension of the key length. The attention score can reflect the relationships among the tokens, which is an important factor as input for the learnable router.

A.6 Pseudo Code of GA-based Sparsity Scheduler

We introduce the details of the GA-based sparsity scheduler via pseudo-code in Algorithm 1. The GA-based approach aims to find an optimal block-wise sparsity configuration, S^*, for an LLM $\mathcal{M}$, that satisfies a target overall sparsity ratio P_{overall}, while maximizing model performance. The process begins by generating an initial population $\mathcal{P}$ of candidate configurations, where each configuration S_i is sampled from the search space $\mathcal{S}_{\text{space}}$,

Method	Ratio (%)	ARC-c	MMLU	Avg. Percentage	PPL
Dense	0	46.16	45.39	100	5.47
ShortGPT	21.02	36.09	44.51	88.04	18.45
BlockPruner	21.99	37.29	-	80.78	11.51
FTP	22.0	45.31	46.15	99.90	11.14
FTP (threshold)	21.92	45.52	46.35	100.35	11.12
FTP (strict constraint)	22.10	45.30	46.12	99.86	11.14

Table 8: Performance comparisons of different methods on LLaMA2-7B.

ensuring $\sum s_i = P_{\text{overall}}$. Each configuration is assessed by applying it to the LLM and measuring the model's accuracy on the evaluation dataset, $\mathcal{D}_{\text{eval}}$. Following these evaluations, the configurations are ranked by accuracy, with the highest-performing ones selected for reproduction.

In each iteration, parents are selected to produce offspring through crossover and mutation. Mutation is applied with a probability of p_{mutate} to introduce diversity while preserving the overall sparsity constraint. The offspring are evaluated and replaced with the worst-performing configurations in the population. This process is repeated for $T_{\text{max\_iter}}$ iterations, with the population progressively evolving towards an optimal solution. The final configuration, S^* , which achieves the highest accuracy, is returned as the optimal sparsity configuration, effectively balancing model performance and computational efficiency.

A.7 Sparsity Ratio Results

As shown in Table 7, we report the block-wise sparsity ratio details obtained from the scheduler. Note that, the block ID is started from 0. We join the (block number - 2) blocks into the scheduler, e.g., 32 blocks in Llama2-7B and 30 blocks involve optimization. Note that, the sparsity ratio of blocks not mentioned in this table are default 0.

A.8 The Results of Supporting KV Cache

As depicted in Section 4.4.2, we introduce a specific threshold to constrain the sparsity ratio for the last token in the depth dimension of LLMs. Apart from the last token, the router's decisions for the other tokens continue to follow the original approach, selecting the required ratio of remaining tokens to be skipped based on the predicted score within each block. Thus, our method, incorporating KV cache modifications, enforces two sparsity constraints: token sparsity across the sequence and last token sparsity in the depth dimension. However, the threshold strategy can not strictly constrain the sparsity of the last token in different input sequences.

Furthermore, we introduce a strict sparsity constraint strategy, combined with the threshold strategy during autoregressive decoding, to consistently ensure that the sparsity target for the last token is achieved. This method monitors the sparsity of the last token across the depth dimension and halts the processing of additional blocks once the target sparsity is reached. If the cumulative sparsity reaches the target before finishing all block computations, subsequent blocks for the last token are required to undergo forward computation. Meantime, it also monitors the number of the remaining blocks waiting for computation together with the current sparsity of the last token to ensure the final sparsity can meet the target sparsity. If the combination of the ratio of remaining blocks and the current sparsity is close to the target, the subsequent blocks should be skipped to guarantee that the final sparsity meets the intended goal. As shown in Table 8, the pruning results, along with the supporting KV cache modifications, demonstrate virtually no performance loss compared to the original results on the ARC-c and MMLU benchmarks. Moreover, the perplexity (PPL) results further demonstrate the robustness of our method in text generation, with a PPL of 11.12 using a threshold strategy to support KV cache, which surpasses the other SOTA methods, indicating that text generation performance remains stable. Additionally, applying the strict sparsity constraint ensures that the overall sparsity target can be met, with a PPL of 11.14 and minimal accuracy impact, confirming that our method is effectively compatible with the KV cache.

A.9 Ethics Statement

This research focuses on improving the efficiency of large language models (LLMs) through fine-grained token-wise pruning, with the goal of reducing computational costs during inference while maintaining model performance. Our work does not involve human subjects or the collection of sensitive data, and thus, does not raise concerns related to privacy, security, or legal compliance. In terms of dataset usage, we primarily evalu-

Algorithm 1 GA-Based Sparsity Scheduler

Input:

$\mathcal{M}$: Pretrained LLM
P_{overall}: Target sparsity ratio
$\mathcal{D}_{\text{eval}}$: Evaluation dataset
$\mathcal{S}_{\text{space}}$: Search space of block-wise sparsity
$\mathcal{T}_{\text{max\_iter}}$: Max iterations for GA

Output:

S^*: Optimal block-wise sparsity ratio configuration

Initialize population $\mathcal{P}$ of block-wise sparsity configurations $\{S_i\}$ from $\mathcal{S}_{\text{space}}$, where $\sum s_i = P_{\text{overall}}$.
Evaluate each S_i in $\mathcal{P}$ by applying it to $\mathcal{M}$ on $\mathcal{D}_{\text{eval}}$ and record Accuracy$(\mathcal{M}_{S_i}, \mathcal{D}_{\text{eval}})$.
Sort $\mathcal{P}$ by accuracy and select top configurations.
Set $t = 0$.
while $t < \mathcal{T}_{\text{max\_iter}}$ **do**
  Select parents from $\mathcal{P}$ based on performance.
  Crossover selected parents to generate new configurations.
  Mutate offspring configurations with probability p_{mutate}, ensuring $\sum s_i = P_{\text{overall}}$.
  Evaluate offspring by computing Accuracy$(\mathcal{M}_{S_{\text{offspring}}}, \mathcal{D}_{\text{eval}})$.
  Replace worst-performing configurations with the best offspring.
  Sort updated $\mathcal{P}$ by accuracy.
  $t \leftarrow t + 1$
end while
return S^* with the highest accuracy from the final population.

ate our approach using publicly available benchmark datasets, such as WinoGrande, ARC-c, and MMLU, which are widely used in the field. We ensure compliance with the licensing and usage terms of these datasets. No personally identifiable information or sensitive data is included in our experiments. We are mindful of the potential societal impact of our research, especially concerning the deployment of LLMs in real-world applications. While the techniques proposed in this work can lead to more efficient LLM deployments, which may lower computational resource requirements and costs, we recognize that LLMs, in general, can perpetuate biases present in their training data. Our research focuses on improving efficiency and does not directly address fairness or bias in language models. However, we acknowledge the importance of addressing these issues in future work. Additionally, all authors have no conflicts of interest influencing the research presented in this paper.

A.10 Reproducibility Statement

Comprehensive descriptions of the datasets used in our experiments are provided, please refer to the Section 4.1. We report the software version and hardware environments, and related hyperparameters in training and validation, please refer to the Section A.1 and 4.1. In Section 3.1, we introduce the LLM architecture and discuss the token redundancy in Section 3.1.1. The implementation details of the sparsity scheduler are provided in Section 3.2.2, followed by a description of the static router in Section 3.2.2 and the dynamic router in Section 3.2.3. Finally, the loss formulations are presented in Section 3.2.4. We report the ablation study results in Section 4.3. We believe these efforts will facilitate the replication and verification of our findings by other researchers. The research is conducted with full adherence to research integrity standards, and all relevant documentation, code, and experimental results will be made available after obtaining a public license.