Anchor Attention, Small Cache: Code Generation With Large Language Models

Xiangyu Zhang, Yu Zhou, *Senior Member, IEEE*, Guang Yang, Harald C. Gall, *Member, IEEE*, and Taolue Chen

***Abstract*—The development of large language models (LLMs) has revolutionized automated code generation. However, their high demand of computation resources has hindered a broader deployment and raised environmental concerns. A common strategy for diminishing computational demands is to cache Key-Value (KV) states from the attention mechanism which is adopted predominately by mainstream LLMs. It can mitigate the need of repeated attention computations, but brings significant memory overhead. Current practices in NLP often use sparse attention which may, unfortunately, lead to substantial inaccuracies, or hallucinations, in code generation tasks. In this paper, we analyze the attention weights distribution within code generation models via an empirical study, uncovering a sparsity pattern, i.e., the aggregation of information at specific anchor points. Based on this observation, we propose a novel approach, `AnchorCoder`, which features token-wise anchor attention designed to extract and compress the contextual information, and layer-wise anchor attention enabling cross-layer communication to mitigate the issue of excessive superposition caused by the compression. The extensive experiments across multiple benchmark datasets confirm the effectiveness of `AnchorCoder`, which can consistently achieve a significant (at least 70%) reduction in KV cache requirements, while preserving the majority of model's performance.**

***Index Terms*—Code generation, attention mechanism, transformers, large language models.**

I. Introduction

AUTOMATED generation of code that aligns with user intentions poses a significant and enduring challenge in software engineering [1], [2], [3]. In recent years, the tremendous progress in deep learning and NLP, especially the advent of Large Language Models (LLMs [4], [5]), has revolutionized the research of automated code generation [6], [7]. LLMs for code, e.g., CodeGen [8], CodeLlama [9] and CodeGeeX [10], have showcased impressive proficiency in writing code, boosting the productivity of developers across various programming environments remarkably [11].

Almost all mainstream LLMs (including those for code which are the main focus of the current paper) adopt the Transformer architecture [12], which, in a nutshell, comprise either an encoder or a decoder, or both, each stacked with multiple identical blocks. In general, the first block takes the tokenized sequence encoded by a word embedding layer, followed by a multi-head scaled-dot self-attention (MHA) layer with an attention mask corresponding to specific language modeling objectives and a feed-forward network (FFN) layer. The attention mechanism [13] underpinning the Transformer architecture is implemented in the MHA layer, which computes a weighted representation of each token in the input sequence based on its relevance to others. Slightly more technically, the word-embedded token sequence which normally concatenates long contexts and user prompts gives rise to three embedding matrices, i.e., the query Q, the key K and the value V, on which the attention (kernel) operations are performed.

$$P := Q \times K^{\mathrm{T}}, \tag{1}$$

$$A := \mathrm{softmax}[\frac{P}{\sqrt{d_k}} \odot M], \tag{2}$$

$$O := (A \times V) \times W_O, \tag{3}$$

Namely, assuming the token sequence length L, each entry of the (unnormalized) relevance matrix $P \in \mathbb{R}^{L \times L}$ measures the relevance of the corresponding pair of tokens. The normalized *attention weight* matrix $A \in \mathbb{R}^{L \times L}$ is computed as a scaling operation and an element-wise mask operation with $M \in \mathbb{R}^{L \times L}$, together with a row-wise softmax. Finally, the output hidden state matrix O is generated by a weighted sum of V with attention weights in each row of A, usually with an extra linear transformation W_O.

The attention mechanism is very costly, albeit effective. To reduce its computational demands, a common strategy is to use the Key-Value (KV) cache. In a nutshell, it is a list of tensors that stores the K, V embeddings for all previous tokens in the attention layer for each block (prefilling), utilized and updated during the autoregressive generation process of LLMs (decoding). A

Received 8 November 2024; revised 11 May 2025; accepted 12 May 2025. Date of publication 15 May 2025; date of current version 16 June 2025. This work was supported in part by the National Natural Science Foundation of China under Grant 62372232 and in part by the Collaborative Innovation Center of Novel Software Technology and Industrialization. The work of Taolue Chen was supported in part by an Overseas Grant from the State Key Laboratory of Novel Software Technology, Nanjing University under Grant KFKT2023A04. Recommended for acceptance by C. McMillan. *(Corresponding authors: Yu Zhou; Taolue Chen.)*

Xiangyu Zhang, Yu Zhou, and Guang Yang are with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing 211106, China (e-mail: zhangx1angyu@nuaa.edu.cn; zhouyu@nuaa.edu.cn; novelyg@outlook.com).

Harald C. Gall is with the University of Zurich, CH-8050 Zurich, Switzerland (e-mail: gall@ifi.uzh.ch).

Taolue Chen is with School of Computing and Mathematical Sciences, Birkbeck, University of London, WC1E 7HX London, U.K. (e-mail: t.chen@bbk.ac.uk).

Digital Object Identifier 10.1109/TSE.2025.3570680

deficiency of KV caching is that LLMs (with billions of parameters) may consume substantial additional memory during the decoding stage, as they need to cache extensive KV states [14], [15], [16]. For instance, CodeLlama-7B [9] (which requires 14 GB to store model parameters) needs an additional 16 GB for the KV cache, under a batch size of 32 and a sequence length of 1,024.[1] The memory demand poses challenges for deploying these models, especially in low resource environments.

Various methods have been proposed to reduce the size of KV cache. For instance, window attention [17], [18] and StreamingLLM [19] predict subsequent words by only caching the most recent KV states. H_2O [20] and FastGen [21] have explored to preserve subsets of the states crucial for prediction by employing specific patterns. In this paper, these methods are collectively referred to as *KV compression methods*. To a large extent, they leverage sparse, low-rank attention approximation [22], based on the belief that a subset of tokens contributes the most values when performing attention operations.

Although current KV compression methods turn out to be fairly effective in NLP tasks (related to dialogue and text completion), it is risky to apply them in code generation. Fundamentally, these methods typically encourage models to focus on local information. In code generation tasks, however, excessive reliance on local information may result in discrepancies between the generated code snippet and either user's intention (e.g., in the prompt) or the ongoing decoding process (e.g., from the context). This primarily stems from the inherent complexity of code, which naturally exhibits long-range dependencies. For instance, in repository-level code generation [23], [24], [25], the relevant context that needs to be considered during generation comes from not only the current, but also externals, files, e.g., imported packages, source code files in the same directory, configuration files and even API documentation. In many cases, these artifacts have their own dependencies. Capturing these long-range dependencies demands more than mere understanding of the local context.

We present an example to illustrate the limitations in current KV compression methods in the left part of Fig. 1, where the LLM is supposed to generate code for multiplication of three numbers. The content within the sliding window (which captures the local information) only includes '*w*'. As a result, the model erroneously interprets this as symbolic emoticons, which wholly deviates from user's request (of a mathematical function).

In this paper, we aim to explore KV compression techniques devised for code LLMs without the over-reliance on local information. To this end, we first carry out an empirical study on code LLMs, based on which we introduce `AnchorCoder`, a novel approach that leverages "anchors" to aggregate sufficient contextual information.

Empirical study. To verify code LLMs' potential for KV compression, we first identify the sparsity pattern of the attention weights matrix A in Eq. (2) within code LLMs. We use the Gini coefficient [26] and the sum of top-2 attention weights to measure the sparsity degree of the attention weights.

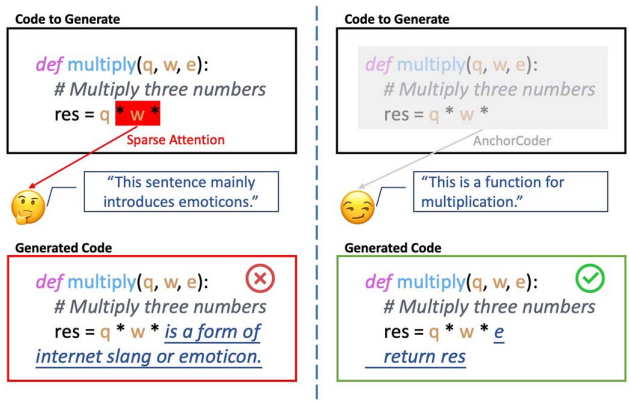

Fig. 1. `AnchorCoder`'s performance on context compression.

Our empirical study (cf. Section II-A) reveals that code LLMs exhibit high sparsity on attention weights. In the majority of layers, the Gini coefficient for attention weights exceeds 0.9. Furthermore, the sum of top-2 attention weights typically accounts for 80% of the total weights. This implies that, in these layers, the model concentrates on a subset of KV states to complete generation, while the vast majority of the KV cache is largely redundant.

Importantly, we discover that code LLMs exhibit a phenomenon of information aggregation around specific tokens. These tokens, referred to as *anchor points* [27], [28], [29], [30], are identified in the first layer which aggregate essential contextual information, particularly the semantics of each line of code. They enable the model to effectively summarize and distill the essential information for subsequent computation. As the computation progresses through later layers, a prominent concentration of attention weights on these anchor points is observed.

Furthermore, we evaluate the state-of-the-art KV compression methods to see whether they can capture the sufficient contextual information with different context lengths. To this end, we design a "needle in a haystack" experiment [31], [32] tailored for code generation. The experiments reveal that, while the existing methods can achieve a high accuracy for shorter code snippets, their performance diminishes significantly for longer ones, where the "needle" is deeply embedded. In such cases, the model frequently fails to detect the "needle", and consequently, may not adhere to the given instructions. As illustrated in Fig. 1, the word 'Multiply' is buried too deeply, leading the model to misinterpret the context due to the sparse attention mechanism. The details are given in Section II-B.

Our new approach. The failure of NLP methods necessitates a rethinking of KV compression methods for code generation. While natural language typically exhibits strong local dependencies where words primarily relate to their nearby context [33] enabling effective local attention strategies, code presents fundamentally different challenges. Long-distance dependencies are critical to the semantic correctness in programming

[1]We assess the storage overhead using fp16 precision. In the case of fp32, the KV cache demands 32 GB.

languages [23], as references to declarations, imports and function definitions may span hundreds or thousands of tokens, creating dependency structures that NLP-focused compression methods fail to preserve. Furthermore, conventional compression techniques substantially alter the inherent attention patterns of code LLMs as we shall see in Section II, making them difficult to generalize to programming contexts. The failure of accounting for code-specific patterns severely impacts model performance on programming tasks. Additionally, code comprehension differs fundamentally from NLP. While NLP methods leverage locality that aligns with human reading patterns, programmers typically rely on both the current line and its global dependencies. Moreover, each line of code typically represents an atomic semantic unit [34], reflecting how programmers naturally structure and interpret code. This human intuition aligns with our empirical analysis of code LLMs, which reveals that these models implicitly learn to treat line boundaries as meaningful structural cues for context compression. Such alignment between model behavior and code organization suggests that line-level boundaries provide natural and effective compression points, helping preserve the semantic integrity of code representations.

We present `AnchorCoder`, a novel approach designed to reduce storage demands of KV caches in code generation models while preserving essential contextual information. In a nutshell, `AnchorCoder` utilizes a mechanism that "communicates in superposition" [35], aggregating the context to a few planted anchors. Let us revisit the example in Fig. 1. Typical sparse attention mechanism tends to ignore context outside the sliding window (i.e., '*w*'). In contrast, `AnchorCoder`, as shown in the right part of Fig. 1, ensures effective code generation by compressing a sufficient context. The rationale lies in the compression phenomenon revealed in the empirical study, which can reduce the size of context inherently, but does not substantially degrade model's performance. More technically, `AnchorCoder` features multi-head positional encoding (cf. Section III-A) and layer-wise anchor attention (cf. Section III-B), which respectively address the loss of positional information due to compression and the degradation of information during transmission between layers.

To evaluate the performance of `AnchorCoder`, we conduct experiments on three benchmark datasets, i.e., HumanEval [1], HumanEvalPlus [36] and MBPP [37]. Experiments on the 7B model demonstrate that `AnchorCoder` maintains model performance at 102%, 110% and 97% on these three datasets, respectively. On the 34B model, it maintains performance at 101%, 93% and 101%, while achieving a KV cache budget of 30%, 30% and 28%, through efficient tuning. Furthermore, we design an experiment that trains `AnchorCoder` from scratch where the results show that with a KV cache budget of 30%, `AnchorCoder` can still achieve performance comparable to that of dense attention, thereby validating the effectiveness and generalizability of `AnchorCoder`.

Our contributions can be summarized as follows.

- We identify patterns of sparsity in the attention mechanisms of code LLMs and uncover the phenomenon of information aggregation on anchor points within them. Additionally, we reveal the limitations of current KV compression methods on code LLMs.
- We propose `AnchorCoder`, a novel sparse attention based approach, which compresses context through token-wise anchor attention and mitigates information degradation through layer-wise anchor attention. This approach can reduce the KV cache overhead while preserving sufficient contextual information.

To the best of our knowledge, this is the first systematic research on effective KV compression methods in LLMs for code generation, and software engineering in general.

Organization. The remainder of this paper is organized as follows. Section II presents an empirical study on code LLMs. Section III presents the proposed approach. Section IV gives the experimental design and Section V reports the results. Section VI discusses the limitation of the approach when applied to general LLMs, as well as potential threats to validity. Section VII reviews the related work. Section VIII concludes the paper.

The source code of `AnchorCoder` is available at https://github.com/NUAAZXY/Anchor_Coder and the models are available at https://huggingface.co/AnchorCoder.

II. Empirical Study

In general, models with sparse attention weights are relatively easier to be compressed, as only a limited number of KV states are needed. It is crucial to study the sparsity pattern of attention weights in code LLMs which is largely uncharted.

A. Sparsity Pattern of Code Generation Models

We carry out an empirical study with four code LLMs of varying scales, i.e., CodeGPT-0.1B [38], PolyCoder-0.4B [39], CodeGen-2B [8], CodeLlama-7B [9] on three typical benchmark datasets, i.e., HumanEval [1], HumanEvalPlus [36] and MBPP [37]. These datasets comprise programming challenges designed to assess functional correctness and user prompts alongside corresponding code.

Given a vector $\vec{w} = (w_1, \cdots, w_n)$ of extracted attention weights, where w_i denotes the attention weight on a specific token, we consider two metrics, i.e., Gini coefficient and the sum of top-2 weights, to measure the sparsity of attention weights distribution. The Gini coefficient is a standard metric for assessing sparsity [26], which is given by $G = \frac{\sum_{i=1}^{n}\sum_{j=1}^{n}|w_i - w_j|}{2n^2\bar{w}}$, where $\bar{w}$ is the mean of all attention weights. The sum of top-2 attention weights, is defined as the sum of the two highest attention scores within S. Formally, it is given by $T = w_{(1)} + w_{(2)}$ where $w_{(1)}$ and $w_{(2)}$ denote the highest and second highest weight in $\vec{w}$. The sum of top-2 attention weights provides an intuitive measure of concentration within attention, indicating the proportion of attention weights attributed to the key positions that the model focuses on the most.

Fig. 2 presents the Gini coefficient and the sum of top-2 attention weights for each layer of the four models, where higher values (approaching 1) indicate higher sparsity. Clearly, except for the first layer, the distributions of attention weights

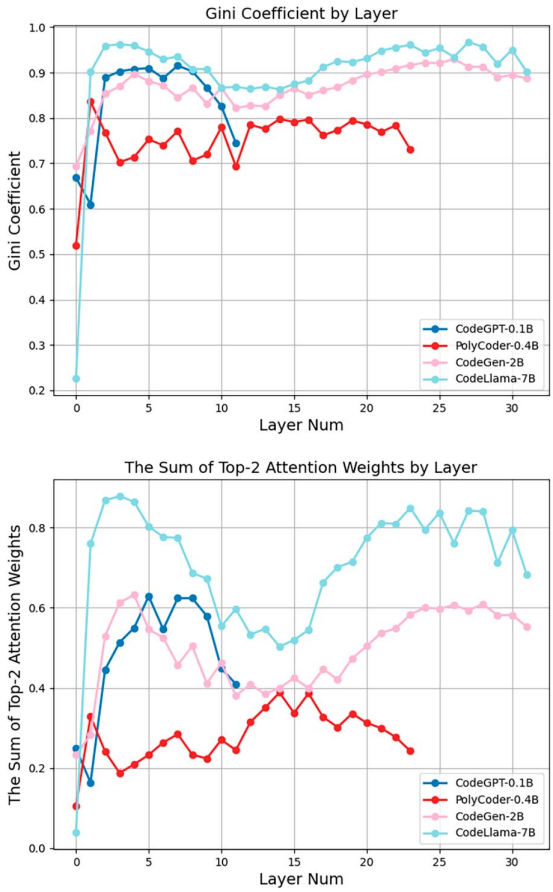

Fig. 2. Sparsity of attention weights in code LLMs.

tend to be sparse. This indicates that in the majority of layers within the model, attention is predominantly concentrated on a small subset of KV states, while the attention weights on other positions are close to zero[2]. Notably, larger models, especially those with increased dimensions in their hidden states, tend to achieve a higher degree of sparsity. For instance, CodeLlama-7B shows the highest Gini coefficient and sum of top-2 attention weights.

Importantly, the higher sparsity exhibited by the attention weights does *not* imply that the model can work based on only few tokens. Rather, it suggests that the positions with high attention weights aggregate the contextual information. The attention mechanism operates by computing attention weights through the dot product of previous query states and key states (Eq. 1), which is further used to generate the hidden state via a weighted sum of value states (Eq. 3). This indicates that these positions encode content from previous contexts rather than just a particular token. We also present an attention heatmap for CodeLlama-7B, depicted in Fig. 3 which visualizes the attention weights between tokens. (The darker colors represent higher attention weights.) In Layer 0, we notice that the attention weights are densely distributed, with the model allocating relatively even attention to each token. In contrast, at deeper layers (e.g., the

[2]In this paper, 'KV state' and 'position' refer to location in a particular layer, while 'token' refers to location across all layers.

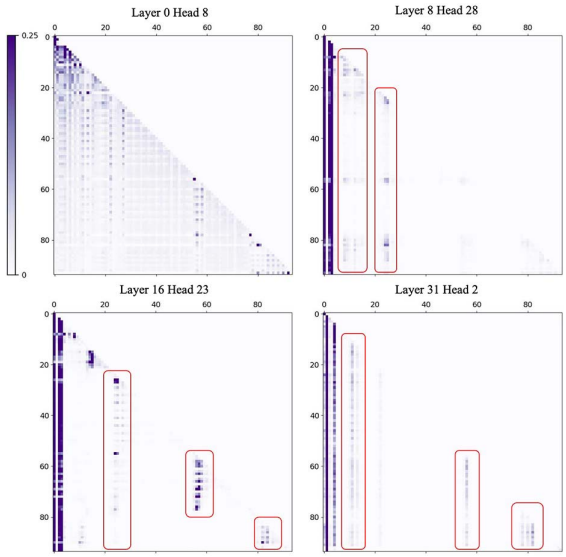

Fig. 3. Attention heatmap of CodeLlama-7B.

8th, 16th and 31st layer), the model tends to focus on fewer pieces of aggregated information, as highlighted by the red boxes. As mentioned in Section I, the specific tokens that the model concentrates on are referred to as anchor points, where the model aggregates and summarizes previous information in the initial layer via dense attention. As the processing proceeds to deeper layers, the model then uses these anchor points to predict the next token. In addition, the first few tokens in the code receive very high attention weights and often represent absolute positions, known as 'sink tokens', as introduced in [19].

To delve deeper into the patterns of attention weights in code LLMs, we examine the tokens (excluding 'sink tokens') that garnered the most attention across the three datasets (i.e., HumanEval, HumanEvalPlus and MBPP), as shown in Table I. (In this table, *num* represents the frequency of these tokens receiving the highest attention weights, while *ratio* indicates their proportion relative to the total number of tokens analyzed.) Surprisingly, the model does not predominantly focus on tokens that carry critical semantic content, such as Python keywords. Instead, it predominantly focuses on relatively semantic-free tokens such as linebreak tokens ('\n'). For instance, in CodeLlama, 78.2% of the attention is concentrated on '\n', while only 21.8% is distributed among other tokens. This further illustrates the phenomenon of information aggregation within the model, as these tokens would be inadequate for prediction. The possible explanation would be that the model compresses contextual information into these '\n', highlighting a unique aspect of compression mechanism in code LLMs.

In summary, by a thorough analysis of the attention mechanism for representative LLMs for code generation, we confirm the sparsity pattern of the model's attention and find that the linebreak token acts as an anchor point in the model, enabling the compression of each line of code. This discovery highlights the tremendous potential of using sparse attention for code generation by leveraging the compressed information.

TABLE I
DISTRIBUTION OF ATTENTION WEIGHTS ON TOKENS

Model	'\n'		others	
	num	ratio	num	ratio
CodeGPT-0.1B	1.5×10^5	24.8%	4.6×10^5	75.2%
PolyCoder-0.4B	3.5×10^5	32.4%	7.2×10^5	67.6%
CodeGen-2B	1.4×10^6	90.8%	1.4×10^5	9.2%
CodeLlama-7B	1.3×10^6	78.2%	3.5×10^5	21.8%

B. Evaluation of Existing KV Compression Methods

As the sparsity of attention weights within code LLMs is confirmed, a natural question is whether the state-of-the-art methods, (such as StreamingLLM [19] and H_2O [20]) in NLP can be applied to code LLMs directly. To answer this question, we design a "needle in a haystack" experiment [31], [32] tailored for code generation.

We construct a list `l=[`$x_1, x_2, \cdots$`,'needle',`$\cdots, x_n$`]` of length n, where x_i's are randomly generated numbers, and instruct LLMs to complete the code `assert 'needle' in l ==` and `assert 'needle' not in l ==` with the prompt "# Determine if there is 'needle' in this list, and complete the code by filling in True or False." We vary n from 64 to 512, with the 'needle' uniformly distributed across positions within the list.

The purpose of this experiment is to observe whether a model using the sparse attention mechanism can locate the "needle", reflecting their capability in retrieving contextual information. As it is a binary classification task, the prediction accuracy should ideally be significantly greater than 0.5; otherwise it suggests that the model fails to follow instructions to complete the code. Our designed experiment provides a direct assessment of the model's information context retrieval capability and its adherence to prompts [40]. Compressing the KV cache could potentially weaken these abilities, which are crucial for generating correct code.

The results, illustrated in the Fig. 4, show a comparison between sparse and dense attention methods across varying context lengths and depths of "needle" placement. StreamingLLM and H_2O display high accuracies with shorter list lengths, indicating that limited attention extraction can handle context information and follow prompts in short texts. However, as the list length increases (with deeper "needle" positions), the performance of these models experiences a significant degradation with accuracy approaching zero, primarily due to the constraints imposed by the window size which prevents the models from querying the input prompt effectively. We hence can conclude that the method of extracting parts of the context using sparse attention is insufficient for code generation models.

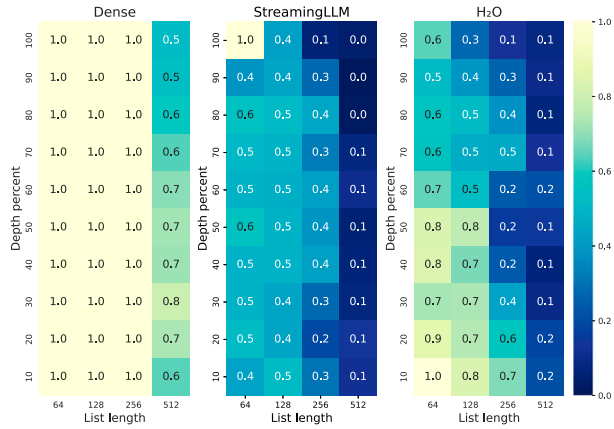

Fig. 4. Results of the "needle-in-a-haystack" experiment.

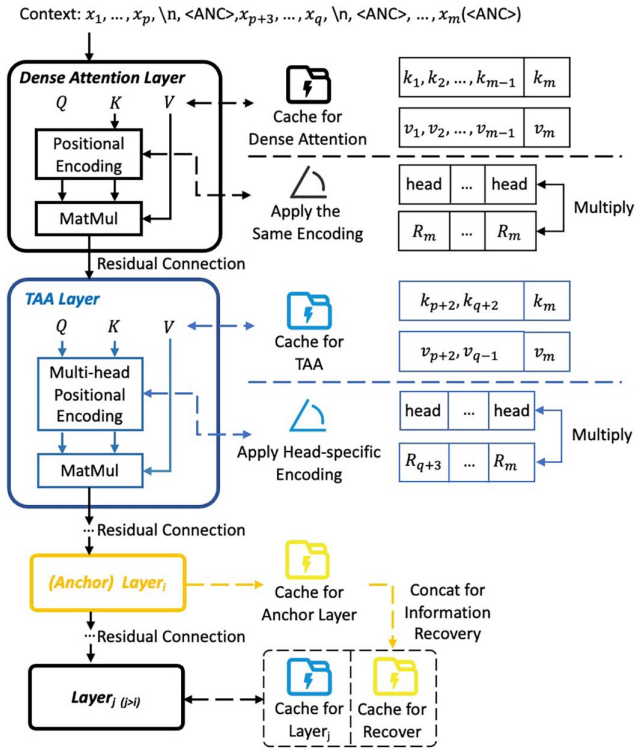

Fig. 5. Workflow of `AnchorCoder`. We illustrate the attention cache process of `AnchorCoder` (omitting MLP, Softmax and Normalization components of the Transformer model). `AnchorCoder` comprises three main components: The upper part represents Dense Attention Layers (implemented in layers 0, 16 in `AnchorCoder`-7B with 15% KV budgets), while the remaining layers utilize Token-wise Anchor Attention for code line cache compression, as shown in the middle part. Considering the limited bandwidth of residual stream, valuable compressed information from shallow layers may struggle to propagate to deeper layers of the model. Therefore, we set an Anchor Layer (layer 16) that bypasses upper-layer information directly to deeper layers (layer 24–31), as illustrated in the bottom part.

III. THE ANCHORCODER METHOD

In this section, we propose `AnchorCoder`, a novel KV compression method. The workflow of `AnchorCoder` is illustrated in Fig. 5. Considering that, as shown in Fig. 3, attention patterns in certain model layers are dense, we preserve selected dense layers to maintain model performance. Based on the observation of anchor phenomena in code LLMs, we implement cache compression for code lines via Token-wise Anchor Attention. Finally, since information compression inevitably leads to context loss which becomes more pronounced during the model's residual connection process, we set an Anchor Layer that bypasses upper-layer information directly to deeper layers. We first introduce the token-wise anchor attention mechanism

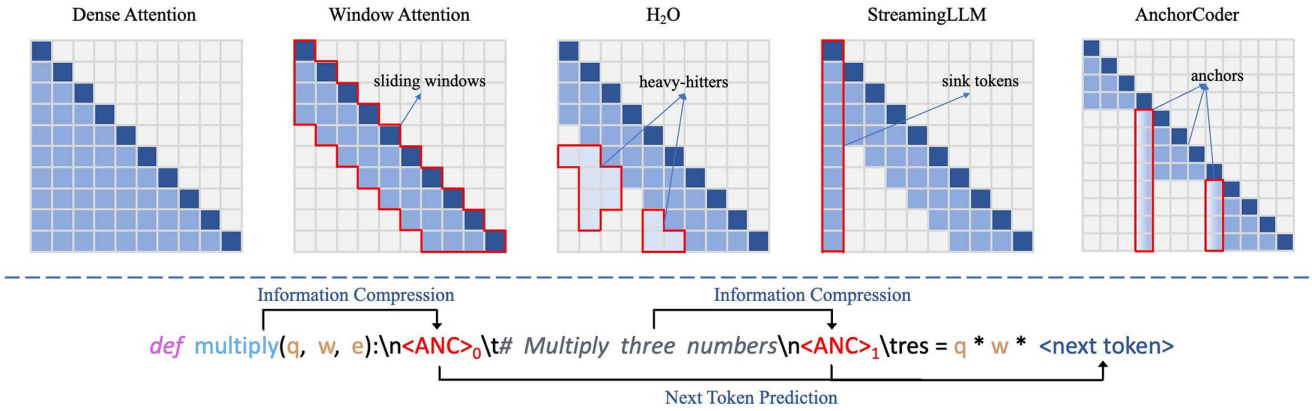

Fig. 6. Upper part provides the attention heatmap deploying different KV cache compression methods. The lower part provides the illustration of `AnchorCoder`.

to reduce the required size of the KV cache (Section III-A), followed by the layer-wise anchor attention to mitigate the bandwidth problem (Section III-B).

A. Token-wise Anchor Attention

The empirical study in Section II reveals that the distribution of attention weights in code generation models is highly sparse and tends to aggregate information on certain anchor points. Although most of these anchor points are linebreak tokens (e.g., making up 78.2% in CodeLlama-7B), relying solely on them may lead to contextual information loss (21.8% is discarded). To address this issues, we introduce Token-wise Anchor Attention (TAA), a method that plants artificial anchors for each line of code and trains them as aggregator of contextual information. An illustration of `AnchorCoder` is given in Fig. 6 (lower part). Compared to Window Attention and its variants, e.g., H_2O (which includes 'heavy-hitters') and StreamingLLM (which adds 'sink tokens'), `AnchorCoder` plants the anchor $<\text{ANC}>_0$ to compress the method name and parameters, and $<\text{ANC}>_1$ to compress the comments on function's purpose as shown in Fig. 6. By these predefined anchors, the model can perform attention operations merely on these positions, reducing the required number of KV states.

Formally, assume a code snippet (from a training dataset) takes the following form,

$$D = x_1, \ldots, x_p, \backslash n, x_{p+2}, \ldots, x_q, \backslash n, \ldots, x_n$$

which comprises multiple lines of code separated by linebreak tokens '\n'. (Here, the total length of the code snippet is n and each x_i is a token.)

We append the special token <ANC>[3] after each \n, obtaining $D_{\text{anchored}} =$

$$x_1, \ldots, x_p, \backslash n, <\text{ANC}>_0, x_{p+3}, \ldots, x_q, \backslash n, <\text{ANC}>_1, \ldots, x_n$$

To restrict model's concentration on anchors, we compute an attention mask (as per M in Eq. (2)) in Algorithm 1. This

[3]Note that <ANC> is only used in the computational and will not appear in the generated code.

Algorithm 1: Attention Mask Algorithm.

Input:
Input Length n and Anchor Indices $\mathcal{I}$;
Output:
Attention Mask M;

1 $M \leftarrow 0_{n \times n}$; `// Initialize` M `as an` $n \times n$ `zero matrix`
2 $neg\_inf = -1e9$; `// A Small Number Ensuring the Result After Applying Softmax is Close to Zero.`
3 **for** $i = 0$ **to** $n - 1$ **do**
4 $\quad M[i, i+1 : n] = neg\_inf$; `// Add Autoregressive Mask by Set Future Tokens Inviable`
5 $\quad$ **for** $j = 0$ **to** $|\mathcal{I}| - 2$ **do**
6 $\quad\quad$ **if** $\mathcal{I}[j+1] < i$ **then**
7 $\quad\quad\quad Start \leftarrow \mathcal{I}[j]$;
8 $\quad\quad\quad End \leftarrow \mathcal{I}[j+1]$;
9 $\quad\quad\quad M[i, Start : End] = neg\_inf$; `// Add Anchor Mask by Set Tokens between Anchors Inviable`
10 $\quad\quad$ **else**
11 $\quad\quad\quad$ **break**
12 **return** M

attention mask M comprises two elements, i.e., autoregressive mask (Line 3-4) and anchor mask (Line 5-9). The autoregressive mask is designed to prevent the model from attending to future tokens, which is achieved by setting the weights of these positions to a very small value, i.e., neg_inf, making their attention weights zero after applying softmax. The anchor mask is designed to mask tokens between anchors. Specifically, given a set $\mathcal{I}$ of anchor indices (e.g., in $D_{anchored}$ the anchor index for $<\text{ANC}>_0$ is $p+2$), we mask all tokens located between two consecutive anchor indices to ensure that attention is concentrated solely on these anchors.

Importantly, `AnchorCoder` preserves the original attention sparsity patterns of code LLMs, ensuring that their core functionalities remain unaffected. A comparison of the heatmaps from AnchorCoder in Fig. 6 (upper part) with those from dense attention mechanisms in Fig. 3 reveals a strong similarity which highlights AnchorCoder's capability to effectively leverage existing attention patterns. The similarity in attention patterns enables the model to be tuned efficiently without requiring

extensive computational resources. Indeed, we employ Low-Rank Adaptation (LoRA) [41], which achieves training efficacy by tuning a small set of parameters. LoRA is particularly suitable for our method because it allows for precise modifications to the attention mechanism while preserving the underlying sparsity patterns that naturally occur in the model.

Our context compression approach minimizes information loss through "communication in superposition", effectively preserving the semantics of the context. However, in its plain form, it primarily considers the positional information of anchors when calculating attention weights, disregarding the relative positions of overlapping contexts within them. In code LLMs, positional information is crucial [42], as even slight deviations can significantly alter the semantics.

To address this issue, inspired by the concept of Multi-Head Attention (MHA), we propose Multi-Head Positional Encoding (MHPE), which incorporates independent positional encodings into the various attention heads of anchors, thereby mitigating the loss of positional information.

In MHA, model's query, key, and value vectors are divided into multiple subspaces. Each attention head processes information from a distinct subspace, focusing on a specific subset of tokens and capturing their unique features and relationships within that subspace. This division allows the model to attend to different tokens across various attention heads, thereby enhancing its ability to process complex contextual information.

When handling tokens compressed within anchors, an optimal strategy is to assign individual positional information to each token. To achieve this, we leverage the characteristic of MHA for its ability to extract features from different subspaces. We treat these compressed tokens as distinct features stored in the anchors, distributed across their respective subspaces. By computing feature correlations within these subspaces, we can effectively perform attention calculations on these compressed tokens.

To implement MHPE, we utilize Rotary Position Embedding (RoPE) as the positional embedding method. RoPE rotates the input vectors to supervise dependencies among tokens at varying positions within the sequence. For anchors in D_{anchored}, we use specialized attention heads to process their subspaces independently, and incorporate positional information of the tokens compressed within these anchors. In detail, MHPE can be defined as

$$\tilde{k} = \text{concat}(R_{s_0} k_0, \cdots, R_{s_{head}} k_{head}),$$

where for each $s_i \in S$, R_{Θ, s_i} is a block diagonal matrix with blocks of the form

$$(R_{s_i})_t = \begin{pmatrix} \cos s_i\theta_t & -\sin s_i\theta_t \\ \sin s_i\theta_t & \cos s_i\theta_t \end{pmatrix}, \quad \theta_t = \theta^{-2t/d}$$

for $t = 1, \cdots, \frac{d}{2}$. Note that $\{k_0, \cdots, k_{head}\}$ represents a set of key states for attention heads with d dimensions, and S is a set of position indices between anchors.

B. Layer-Wise Anchor Attention

In LLMs, residual stream plays a crucial role in passing information across layers. However, in `AnchorCoder`, the residual stream may be affected by its bottleneck, leading to information degradation. The residual stream adds each layer's output back to its input before passing it to the next layer, ensuring a consistent flow of information. Intuitively, this reinforces the information from the previous layers, effectively allowing the model to track changes to the input as it propagates through the layers [35], [43]. Formally

$$\hat{r}_l = r_{l-1} + \text{attn}(\text{LN}(r_{l-1}))$$
$$r_l = \hat{r}_l + \text{mlp}(\text{LN}(\hat{r}_l)),$$

where

$$\text{attn}(x) = W_O \times V \times \text{softmax}\left(\frac{Q \times K^T}{\sqrt{d_k}}\right) \tag{4}$$

Here, r_l represents the state of the residual stream after writing information at the l-th layer, $\text{LN}(\cdot)$ denotes layer normalization, $\text{mlp}(\cdot)$ denotes the multi-layer perceptron in transformer, $\hat{r}_l$ represents the intermediate representation in residual computation, and $W_Q, W_K, W_V \in \mathbb{R}^{d \times d}$ are matrices that transform the inputs to query, key and value vectors, respectively, with d being the dimension of the model. Since each transformer layer writes information from the current layer to the residual stream, it often faces so called activation bottlenecks [35], where the dimension required to write complete information from each layer is much higher than that of the residual stream itself. As a result, it is virtually impossible to retain the complete information written by each layer, and the model must find ways to manage the limited capacity of the residual stream. Previous research has shown that attention heads writing in the opposite direction to the residual stream may effectively eliminate certain features and alleviate the problem of activation bottlenecks [44].

To determine whether a similar phenomenon happens in code LLMs, we analyze the eigenvalues of the matrix $W_{OV} = W_O \times W_V$. As per Eq. 4, W_{OV} writes "linearly" to the residual stream and does not mix information between tokens, thereby directly influences the state updates within this stream. In contrast, $W_{QK} = W_Q \times W_K$ mixes information between tokens and is gated by the (nonlinear) softmax [45]. The presence of negative eigenvalues in W_{OV} of attention heads would indicate that the model is removing information from the residual stream, revise-which may degrade the performance of the neural network. This is because negative eigenvalues suggest that certain dimensions in the feature space are being shrunk or collapsed, rather than being preserved or enhanced as intended. In the context of attention mechanisms, which are designed to focus on the most relevant parts of the input data, this could mean that the model is inadvertently ignoring or downplaying important features.

Our analysis (Fig. 7) reveals that, in some attention heads within the model, a majority of the eigenvalues of W_{OV} are negative. This suggests that these attention heads are writing in the opposite direction into the residual stream, deleting information and alleviating the activation bottleneck issue [35].

Previous studies observed that models can make accurate predictions at shallow layers, which may become incorrect in deeper layers [46], [47]. This suggests that crucial evidence present in the shallow layers is being removed or discarded

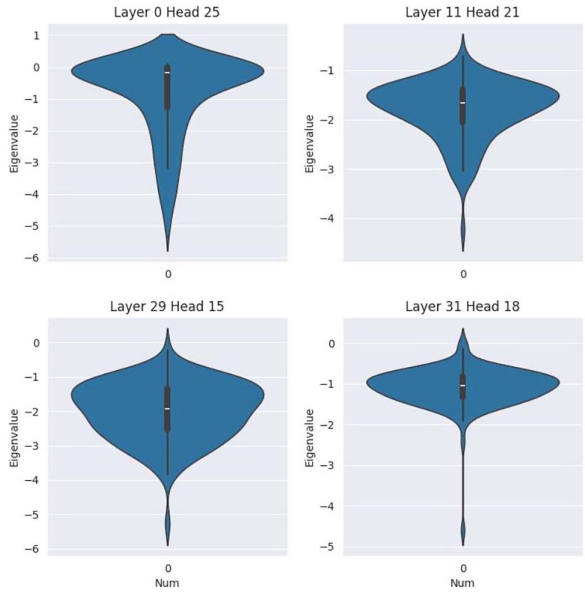

Fig. 7. Eigenvalue distribution of W_{OV}.

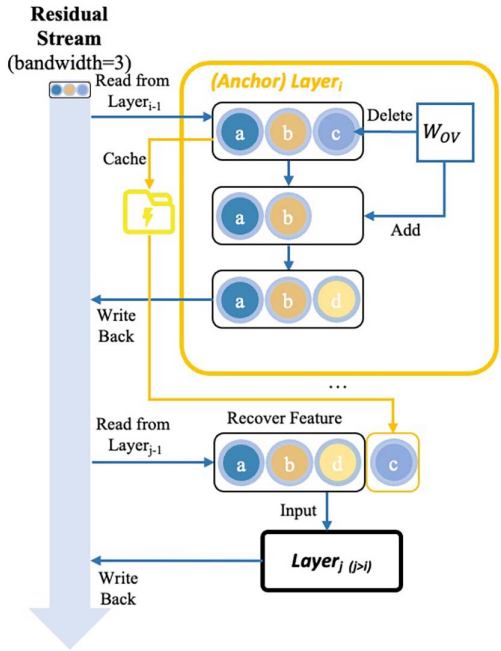

Fig. 8. Workflow of layer-wise anchor attention.

as information flows through the model. Logically, certain attention heads are responsible for deleting the evidence via the residual stream, leading to information degradation. In AnchorCoder, as the anchors aggregate contextual information resulting in a superposition, the deletion behavior of the attention heads may be amplified, thereby increasing the risk of losing crucial information in generation.

To mitigate this issue, we introduce a simple yet efficient method, namely, Layer-wise Anchor Attention (LAA), illustrated in Fig. 8. LAA alleviates the impact of information loss by setting a bypass for the residual stream. Specifically, we set up an specific anchor layer, within the shallow layers of the model. The KV states of this anchor layer are then utilized as additional targets for attention calculations in deeper layers, which can be formalized as

$$\text{LAA}(Q,K,V) = \text{softmax}\left(\frac{Q \times \tilde{K^T}}{\sqrt{d_k}}\right)\tilde{V},$$
$$\tilde{K} = \text{concat}(K,K'), \tilde{V} = \text{concat}(V,V'),$$

where Q, K, V are query, key and value states of the current layer, K', V' denote key and value states of the anchor layer, d_k represents the dimension of the model.

We provide an illustrative example in Fig. 8. Given a residual stream with a bandwidth of 3 features, when the residual stream bandwidth becomes saturated, W_{OV} will delete certain features and add new ones, resulting in information loss. LAA effectively recovers crucial information that might otherwise be lost as data flows through the model layers by incorporating shallow-layer KV states into deeper attention calculations without introducing additional parameters.

This design enhances the completeness of the residual stream, significantly boosting model's capacity to recognize and process shallow features. Notably, since LAA reuses the KV cache in anchor layer to recover information, it does not incur additional memory overhead. Moreover, although LAA requires extra computation, it does not significantly increase the inference latency, which will be discussed in Section V-C.

IV. Experiment Setup

We carry out extensive experiments for AnchorCoder to address the following three research questions:

RQ1: How accurate is the code generated by AnchorCoder?

RQ2: What are the KV cache requirements of AnchorCoder?

RQ3: How do MHPE and LAA contribute to model's performance?

Base Model. We use CodeLlama-7B and CodeLlama-34B [9] (built on top of LLaMA 2 [48]) as the base models for AnchorCoder, which we will refer to as AnchorCoder$_T$-7B and AnchorCoder$_T$-34B, respectively. To mitigate the impact of utilizing additional data, we trained a Llama-like model from scratch for both the baseline and AnchorCoder. This model features 100 million parameters and a hidden size of 512, encompassing 16 transformer blocks and 8 attention heads. This variant of AnchorCoder is referred to as AnchorCoder$_P$. Since AnchorCoder$_P$ is initialized randomly, its attention remains dense, allowing us to assess the generalizability of AnchorCoder when applied to models with dense attention.

Datasets. We incrementally tune AnchorCoder$_T$ on the CodeSearchNet and CodeHarmony datasets, respectively, and evaluate the experimental results on the HumanEval [1],

HumanEvalPlus [36] and MBPP [37] datasets. For `AnchorCoder`$_P$, we train from scratch using CodeSearchNet with a total of 500M tokens.

- **CodeSearchNet:** The CodeSearchNet corpus [49] is a dataset comprising 2 million (comment, code) pairs from open-source libraries hosted on GitHub. It contains code and documentation for several programming languages. We utilize the Python code subset for incremental pre-training.
- **CodeHarmony:** CodeHarmony[4] is curated from existing open-source datasets, and employs LLMs for automated test case generation.
- **HumanEval:** The HumanEval dataset [1], released by OpenAI, includes 164 programming problems with function signatures, docstrings, bodies, and several unit tests.
- **HumanEvalPlus:** The HumanEvalPlus dataset [36], building upon the foundation established by HumanEval, has expanded its test cases by 80 times, thereby enhancing its capability to assess the correctness of code.
- **MBPP:** The MBPP benchmark [37] consists of approximately 1,000 crowd-sourced Python programming problems, designed to be solvable by entry-level programmers. Each problem includes a task description, code solution and three automated test cases.

Baseline. To compare with our approach, we select *Window Attention* [18], *StreamingLLM* [19] and *H_2O* [20] which are training-free methods based on sparse attention. In general, these methods employ the concept of sliding window to reduce the number of tokens that the model attends to, and incorporate additional global information, such as attention sinks for StreamingLLM and heavy hitters for H_2O. In addition to these approaches, we also consider training-based *AutoCompressors* [50] as a baseline, which incorporates additional summary tokens for context compression and utilizes them as soft prompts for subsequent generation.

Metrics. To evaluate the performance of `AnchorCoder`$_P$, we employ pass@k and KV Cache Budget as the metrics for evaluating model's performance. The pass@k metric reports the percentage of problems solved within the k generated codes. In this study, we utilize the pass@1 metric with greedy decoding for both `AnchorCoder` and baselines, which provides the most direct reflection of the model's generative capability and is not subject to the influence of randomness. Considering the different strategies employed by various methods, we are unable to set the same budget for each model. Therefore, we have set a similar KV cache budget for each model, allowing for a 1.5% deviation.

Due to the limited computational resource, we only use 500M tokens to train `AnchorCoder`$_P$, which is challenging for generating correct code in the HumanEval and MBPP datasets. Therefore, we use the perplexity and accuracy metric to evaluate the language modeling and next token prediction capability of different methods respectively.

[4]https://huggingface.co/datasets/Flab-Pruner/CodeHarmony

Implementation Detail. For `AnchorCoder`$_T$ and its corresponding baseline, we fine-tuned CodeLlama on the CodeSearchNet and CodeHarmony datasets, which together contain 150M tokens. For `AnchorCoder`$_T$-7B, we used Low-Rank Adaptation (LoRA) [41] to fine-tune the W_Q, W_K, W_O, and W_V matrices of the model with a rank of 16 and a learning rate of 5e-5. The training of `AnchorCoder`$_T$-7B was conducted on a single Nvidia RTX 4090. We controlled the sparsity of the attention weights by creating three variants, each utilizing TAA in a different number of layers. Specifically, we configured the models to use TAA in 24, 28 and 30 (out of 32) layers, with the remaining layers using dense attention to aggregate context. The anchor layers for the three variants are positioned at the 8th, 8th and 16th layer, respectively. For `AnchorCoder`$_T$-34B, to avoid the significant latency introduced by offloading during distributed training, we used QLoRA [51] with rank 32 and fine-tuned the model using 4 Nvidia RTX A6000 GPUs. The models used TAA in 36, 42 and 45 (out of 48) layers, while the rest of the settings remained unchanged.

For `AnchorCoder`$_P$ and its corresponding baseline, we employ the 500M tokens from CodeSearchNet dataset for full-parameter training with a learning rate of 5e-4. This training was accomplished using 5 Nvidia RTX 4090, with 12 out of 16 layers utilizing TAA, while the remaining layers employed dense attention.

All experimental results are produced using greedy decoding to eliminate randomness.

V. Experiment Results

A. RQ1: Accuracy of the Generated Code

To address **RQ1**, we evaluate `AnchorCoder` on the HumanEval, HumanEvalPlus and MBPP datasets, respectively. To ensure fairness, we set the KV budget in the baseline to the same level. Notably, due to the varying compression strategies employed by each method, the length of the generated code differs. Consequently, it is challenging to set the sparsity level exactly the same. We control the sparsity difference between methods to be within 1.5% for models of both scales.

The experimental results, shown in Table II, indicate that under the same KV budget and model scale, `AnchorCoder` surpasses the baselines and achieves performance comparable to dense attention with KV cache budget at 30%. This suggests that `AnchorCoder` maintains model performance even at a compression rate of approximately 30%.

Local attention-based methods such as Window Attention and StreamingLLM tend to generate code with hallucinations and inconsistent with the prompt, due to their inherent limitation in concentrating only on local information and failing to capture the user's intent specified in the prompt. Additionally, Window Attention cannot correctly generate tokens beyond the window size because it overlooks the tokens at the beginning of the text, which are crucial for representing the absolute position in the text [19].

H_2O achieves attention compression by discarding tokens with low attention weights in previous computations, assuming these tokens are insignificant for predicting subsequent tokens.

TABLE II
PERFORMANCE COMPARISON OF ANCHORCODER$_T$ IN RQ1

Model Size	Method	HumanEval		HumanEvalPlus		MBPP	
		KV Budget % (↓)	Pass@1 % (↑)	KV Budget % (↓)	Pass@1 % (↑)	KV Budget % (↓)	Pass@1 % (↑)
7B	Dense	100	31.10	100	23.17	100	39.40
	Window Attention	30	0.00	30	0.00	28	0.00
		20	0.00	20	0.00	18	0.00
		15	0.00	15	0.00	12	0.00
	StreamingLLM	30	6.10	30	4.27	28	3.40
		20	1.83	20	1.22	18	1.60
		15	1.22	15	0.00	12	1.80
	H_2O	30	17.68	30	14.02	28	17.60
		20	14.63	20	10.98	18	15.40
		15	14.02	15	10.36	12	14.00
	AutoCompressors	30	24.39	30	21.34	28	23.00
		20	23.17	20	20.12	18	20.20
		15	17.07	15	13.41	12	17.00
	AnchorCoder$_T$	30	**31.71**	30	**25.61**	28	**38.20**
		20	**29.88**	20	**25.00**	18	**38.00**
		15	**27.44**	15	**24.3**	12	**36.40**
34B	Dense	100	40.24	100	35.37	100	56.80
	Window Attention	30	0.00	30	0.00	28	0.00
		20	0.00	20	0.00	18	0.00
		15	0.00	15	0.00	12	0.00
	StreamingLLM	30	7.32	30	6.10	28	8.60
		20	3.66	20	2.44	18	6.40
		15	1.22	15	0.00	12	3.20
	H_2O	30	21.34	30	17.07	28	28.40
		20	18.90	20	15.86	18	23.20
		15	17.68	15	15.24	12	22.80
	AutoCompressors	30	35.98	30	29.88	28	49.80
		20	31.10	20	28.05	18	44.20
		15	28.66	15	23.78	12	38.40
	AnchorCoder$_T$	30	**40.95**	30	**32.93**	28	**57.40**
		20	**35.98**	20	**29.27**	18	**55.60**
		15	**35.37**	15	**27.44**	12	**53.20**

However, in code generation tasks the spatial positions of related code fragments can be far apart, and H_2O might discard tokens that are insignificant in the earlier context but critical in the subsequent context. Moreover, AutoCompressors uses multiple summary tokens as soft prompts to represent more contextual information with a few key-value caches. However, it assigns absolute positional information to these tokens, potentially affecting model's generalization ability. Furthermore, the summarization method based on soft prompts can disrupt the positional information of the context. In contrast, `AnchorCoder` compresses context into the anchor points, thereby maintaining the model's generation capability with less KV states. In Section V-D, we conduct further case studies where an analysis of the limitations of the window attention-based approach is presented.

Note that the training-based methods, such as AutoCompressors and `AnchorCoder`, we only update the self-attention-related parameters using LoRA. However, one may argue incorporating additional data could introduce an unfair comparison with other baseline methods. To ensure fairness and further validate the effectiveness of `AnchorCoder`, we pre-trained `AnchorCoder`$_P$ and other baselines from scratch on the CodeSearchNet dataset and evaluated their performance on language modeling tasks.

The experimental results are shown in Table III. In this experiment, the KV cache budget is set to 30%. Notably, under this setting, `AnchorCoder` can achieve performance comparable to Dense Attention. Regarding the training-free, sliding window-based method, it performed well across the evaluation metrics in this experiment, largely due to its design, which prioritizes reducing the KV cache and perplexity in long-text generation, rather than generating contextually coherent code. AutoCompressors demonstrated superior performance in this experiment compared to directly tuning CodeLlama, mainly due to the use of full-parameter fine-tuning in this setting, which enhanced the method's overall effectiveness. This result also supports the efficient fine-tuning capability of `AnchorCoder`. In addition, this experiment can also demonstrate the generalizability of `AnchorCoder`, where `AnchorCoder` can be applied to models with lower sparsity of attention, as the `AnchorCoder`$_P$ is initialized randomly, which prompts the model to allocate attention uniformly across all tokens.

TABLE III
PERFORMANCE COMPARISON OF ANCHORCODER$_P$ IN RQ1

Method	Perplexity (↓)	Accuracy % (↑)
Dense	4.32	70.46
Window Attention	5.28	67.12
StreamingLLM	5.30	67.02
H_2O	5.08	67.69
AutoCompressors	4.53	69.23
AnchorCoder$_P$	**4.38**	**70.35**

TABLE IV
KV CACHE OVERHEAD DURING DECODING PHASE ON HUMANEVAL IN RQ2

Method	Pass@1	KV cache (GB)	length	ratio
Dense-7B	31.10	5.02	55.74	9.01×10^{-2}
AnchorCoder$_T$-7B	31.71	1.52 (↓70%)	56.12	2.71×10^{-2}
	29.88	1.06 (↓79%)	56.01	1.90×10^{-2}
	27.44	0.77 (↓85%)	54.34	1.42×10^{-2}
Dense-34B	40.24	16.43	60.12	2.73×10^{-1}
AnchorCoder$_T$-34B	40.95	5.27 (↓70%)	60.89	8.65×10^{-2}
	35.98	3.34 (↓80%)	60.37	5.53×10^{-2}
	35.37	2.70 (↓84%)	66.79	4.04×10^{-2}

TABLE V
ABLATION RESULT IN RQ3

Model	Pass@1	Runtime		
		Prefilling	Decoding	Throughput
Dense	31.10	3.62×10^{-2}	560.39	29.24
AnchorCoder$_T$-7B	31.71	4.54×10^{-2}	420.35	38.98
-w/o LAA	29.27	4.15×10^{-2}	415.17	39.46
-w/o MHPE	29.88	3.86×10^{-2}	417.28	39.26
-TAA only	28.04	3.71×10^{-2}	411.49	39.82

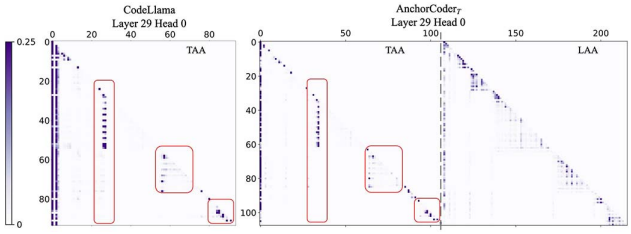

Fig. 9. Attention headmap of CodeLlama and AnchorCoder$_T$-7B.

B. RQ2: Memory Consumption of AnchorCoder

To address **RQ2**, we present a comparative analysis of the total KV cache consumption (GB) required during the model decoding phase of AnchorCoder$_T$-7B versus that of dense attention. Unlike the Sparsity metric, which focuses on the percentage of tokens involvement in computing the attention weights, the evaluation of KV cache overhead necessitates a consideration of the caching strategies employed during the prefilling phase. Specifically, dense attention requires the storage of the KV cache for all tokens in the prompt, as these cached representations will subsequently serve as context in the decoding phase. In contrast, while AnchorCoder also processes all tokens within the prompt for information aggregation during prefilling, it only needs to cache the KV states for the anchor points. The sparsity metric primarily assesses the number of tokens involved in the computation.

As shown in Table IV, AnchorCoder$_T$-7B achieves comparable performance to Dense Attention with an overall KV cache requirement of only 1.52 GB. Given the variance in text lengths generated by different methods, we computed the ratio (GB/token) of cache and length to determine the average cache required per token. AnchorCoder$_T$-7B maintains performance levels of 102%, 96% and 88% when reducing cache overhead by 70%, 79% and 85%, respectively. Additionally, AnchorCoder$_T$-34B similarly maintains performance of 102%, 89% and 88% when reducing cache overhead by 70%, 80% and 84%, respectively.

Specifically, for each layer, the use of TAA reduces KV cache overhead by 89% compared to layers utilizing dense attention. Based on the experimental results, we recommend targeting an overall cache compression rate of approximately 30%, which corresponds to three-quarters of model layers employing TAA. This configuration minimizes cache overhead while preserving model performance as much as possible.

C. RQ3: Ablation Study

For **RQ3**, we first design ablation experiments that primarily focus on two metrics: Pass@1 and runtime, shown in Table V. We then examine the distribution of attention within the model when utilizing LAA to underscore its significance.

Ablation experiments on Pass@1. The objective of studying the Pass@1 metric is to investigate the impact and contribution of each component within AnchorCoder evaluating on HumanEval, as shown in Table V, the 2nd column.

It can be observed that the removal of LAA leads to a degradation in model performance. This is because the model has limited dimensions, and compression inevitably leads to information loss. LAA serves as an alternative pathway for linking information across layers, significantly reducing this loss of information. In Fig. 9, we present the attention heatmaps for CodeLlama and AnchorCoder$_T$, demonstrating that the layers utilizing LAA do not alter the inherent distribution pattern of attention, which ensures the efficiency of tuning. Additionally, a substantial portion of attention is allocated to the KV states in anchor layer, which is crucial for sustaining the overall performance of the model. MHPE, tailored for anchor points, enhances the model's positional awareness within the overlapping information by adding relative positional information across different attention heads, thereby further augmenting the effectiveness of Anchor Attention.

Ablation experiments on runtime metrics. We aim to investigate the temporal cost of the model in generating extended text under different settings. To this end, we configure the generation length to 16,384, which is the training length used by CodeLlama, and present the experimental results in Table V, the 3rd-5th column. In this table, we dissect Runtime into three distinct metrics: Prefilling (s), which represents the time the model takes to process the prompt; Decoding (s), which denotes the time taken for autoregressive generation; Throughput (token/s), indicating the number of tokens generated per second during the decoding phase.

Our analysis reveals that AnchorCoder can reduce the inference latency of CodeLlama by 25%, primarily attributed to a bandwidth bottleneck during decoding caused by frequent cache accesses. By reducing the size of the KV cache, we observe a substantial decrease in decoding latency. The incorporation of LAA, serving as a communication channel between

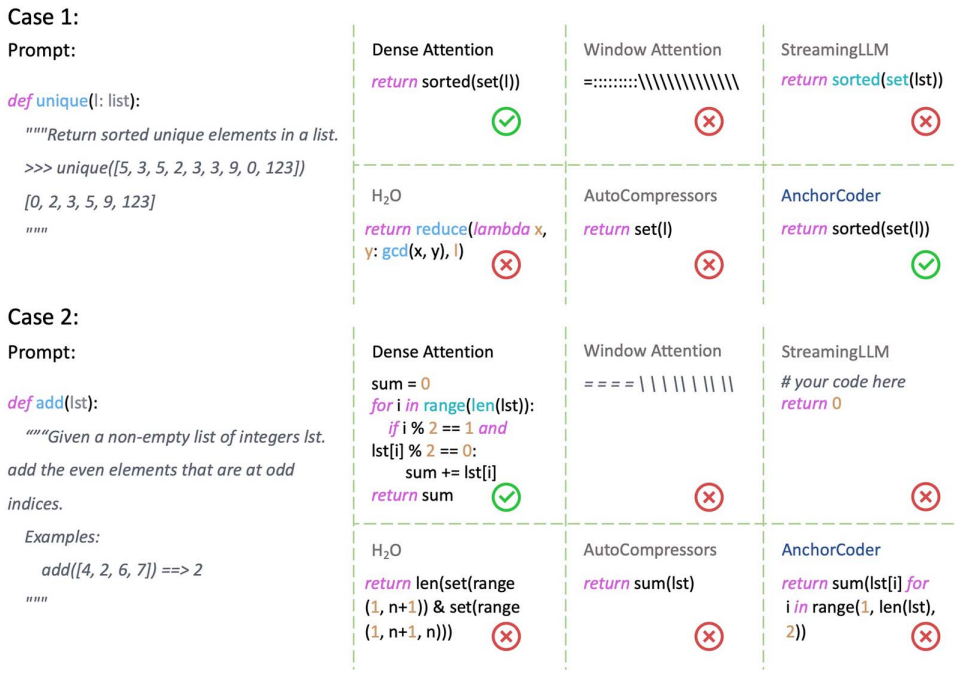

Fig. 10. Case study of code generated by AnchorCoder-7B and baseline methods.

layers, increases the computational demands of the model. However, our ablation studies demonstrate that during the decoding phase, latency does not increase significantly, as the presence of a bandwidth bottleneck necessitates the model expending more time accessing the KV cache than calculating attention [52], [53], [54]. Consequently, additional computational demands do not substantially contribute to delay. Nevertheless, during the prefilling stage, LAA introduces an additional delay of 3.91×10^{-3}s (9%) due to computational bottlenecks at this stage, which is deemed acceptable. Furthermore, our results indicate that MHPE increases inference latency, as it necessitates the computation of distinct positional encodings for each attention head.

In summary, our ablation experiments confirm the effectiveness and efficiency of MHPE and LAA. MHPE enhances the model's positional awareness by establishing independent position encodings for each attention head, improving model performance without added parameters and only adding minimal computational overhead related to position encoding. Similarly, LAA recovers information through cross-layer cache, adding no additional parameters and only incorporating extra key states into attention calculations, thereby improving model performance without significantly increasing inference latency.

D. Case Study

In Fig. 10, we present two cases collected from the HumanEval dataset, demonstrating examples where AnchorCoder generate correct and incorrect code. We also provide the results from baseline methods to discuss the effectiveness of AnchorCoder.

In the first case, Window Attention generates messy code due to its failure to account for 'sink tokens'. StreamingLLM uses an incorrect variable as it cannot access the entire context. H_2O produces hallucinated code as it lacks critical context, leading to inaccurate generation. AutoCompressors generates a result that diverges from the input prompt. In contrast, only AnchorCoder and Dense Attention generate the correct solution.

In the second case, the baseline methods produce code that significantly deviates from the intended prompt. Although AnchorCoder correctly considers the requirement for odd indices in its solution, it overlooks the condition regarding even elements, resulting in an incorrect output. We attribute this failure primarily to information loss during the compression process.

This comparison underscores the strengths and limitations of AnchorCoder, revealing areas for potential improvement, particularly in handling more complex conditions.

VI. Discussion

A. Limitations

While AnchorCoder demonstrates promising results for code LLMs by leveraging code-specific patterns, several limitations may arise when applied to general LLMs. This section analyzes why our approach is particularly effective for software engineering applications (such as code generation) but may have challenges in broader domains.

(1) AnchorCoder is designed based on the distinctive attention patterns observed in our empirical study (Section II) of code LLMs. These patterns, particularly the high attention weights assigned to structural tokens such as linebreaks, represent code-specific phenomena that may not manifest in general LLMs trained on natural language. The absence of these

consistent anchoring points would invalidate our basic assumption requiring effective information aggregation.

(2) code exhibits more regular syntactic structure and formatting, compared to natural language which varies dramatically in sentences and paragraphs. As demonstrated in the case studies (cf. Fig. 10), applying compression to irregularly structured text may lead to information loss. In contrast, code lines allow `AnchorCoder` to compress contextual information with minimal semantic degradation, which is difficult to achieve in natural language.

(3) the relationship between syntax and semantics differs fundamentally between code and natural language. Programming languages, as an artificial language, follow strict syntactic rules with predictable information flow: function definitions precede implementations, variable declarations precede usage, imports precede references, etc. These consistent patterns, combined with the critical importance of preserving long-range dependencies, make our anchoring strategy particularly effective for code generation. Natural language, with its more fluid structure and distinct dependency characteristics [33], may not benefit substantially from the anchoring mechanism.

These distinctions highlight why `AnchorCoder` represents a bespoken, domain-specific solution for software engineering. By exploiting the unique characteristics of code–its structured format, consistent syntactic patterns and specific attention distribution–our approach achieves efficient compression while maintaining the semantic integrity essential for accurate code generation.

B. Threats to Validity

Internal Validity. A potential threat to internal validity is the randomness inherent in LLM decoding. To mitigate the impact of uncertainty on experimental results, we employ greedy decoding to ensure the certainty of the generated code by the model, as opposed to the nucleus sampling method commonly used by most LLMs, which introduces randomness.

External Validity. A potential threat to external validity in our study is the generalizability of our findings, which pertains to both language and model types. Our experiments were conducted primarily on Python datasets, which are most commonly required for code generation tasks. Limited by our devices, we conducted experiments only on CodeLlama with a 7B parameters. Notably, we have observed that larger models exhibited sparser attention. Therefore, we believe that larger-scale models have greater potential for compression, which also applies to the method we proposed.

Construct validity. Evaluation metrics pose a potential threat to construct validity for code generation tasks. The metric we selected, pass@k, is the most common and practical in everyday development, compared to the match-based BLEU and CodeBLEU metrics. We also compared sparsity and runtime metrics, which are crucial for assessing the efficiency of code generation. In addition to this, it is important to acknowledge that the accuracy of the pass@k metric in assessing the correctness of generated code is highly dependent on both the quantity and quality of the test cases used. To ensure the comprehensiveness and fairness of testing, we have not only utilized the HumanEval dataset but also employed HumanEvalPlus, which includes over 80 times more test cases compared to HumanEval. This significant increase in the number of test cases allows for a more thorough evaluation of the code's performance in edge cases. This approach ensures that the assessment of generated code is robust and reflects real-world operational challenges more accurately.

VII. Related Work

Code Generation. LLMs have recently demonstrated remarkable capabilities across various applications, particularly in programming-related tasks [1], [2]. An early standout is Codex [1], which leverages a vast GPT model fine-tuned on GitHub code, fueling the development of Copilot for real-time coding assistance. Codex has ignited considerable interest in both academia and industry, catalyzing the creation of numerous models. For example, DeepMind's AlphaCode [2] is engineered to address coding challenges in competitive programming environments. Similarly, Meta introduced models such as InCoder [3] and CodeLlama [9], while Salesforce developed CodeRL [55] and CodeGen [8]. The BigCode project unveiled StarCoder [56]. In addition, numerous open-source large-scale models has further enhanced the capabilities in code generation [57], [58], [59], [60]. This surge in model development underscores the significant enhancements in the quality and practicality of automated code generation, marking a substantial leap forward in the methodologies and efficiency with which coding tasks are addressed and accomplished [61], [62].

Beyond model performance, research has also focused on generation efficiency. SEC [47] proposes skipping certain layers during inference to obtain predictions, reducing unnecessary computation. CodeFast [63] aims to identify and halt generation when redundant tokens are encountered, thereby improving inference efficiency while maintaining model performance. [61] explores code-aware quantization strategies that could preserve accuracy while preventing robustness degradation. [64] utilizes active learning to train a model with a reduced dataset while maintaining the desired performance. CodeMentor [65] proposes a framework for few-shot learning that fine-tunes LLMs for code review tasks using organizational data. These approaches enhance model efficiency and reduce computational overhead without sacrificing output quality.

Context Compression. The concept of context compression is closely related to earlier efforts to archive past representations, enhancing memory and facilitating long-range sequence modeling in Transformers. Specifically, the Compressive Transformer [66] utilizes a learned convolutional operator to condense Transformer activations into a more compact memory representation. Gisting [67] involves training a language model to condense prompts into concise sets of "gist" tokens, which can be cached and reused to enhance computational efficiency. AutoCompressors [50] compresses long contexts into compact summary vectors, which are then accessible to the model as

soft prompts. [68] proposes a plug-and-play approach that can incrementally compress the intermediate activations of a specified span of tokens into more compact forms, thereby reducing both memory and computational costs in processing subsequent contexts. In contrast to the aforementioned methods, our proposed `AnchorCoder` is designed for code generation which compresses in a more natural manner without altering model's inference.

Sparse Attention. The sparse attention mechanism is a variation of the traditional attention mechanisms used in neural networks, specifically designed to handle large sequences more efficiently by reducing the computational time and memory usage. Window attention [17], [69] leverages the local features of text for prediction, enabling the model to cache only a minimal amount of KV state for long text predictions. StreamingLLM [19] observes the phenomenon of attention sink within large language models and, building upon window attention, introduces sink tokens to enhance long text generation capabilities. H_2O [20] achieves sparse attention by discarding values with low attention scores from the context during decoding, thus maintaining partial model performance with reduced cache requirements. FastGen [21] employs a combination of four strategies to effectively restore attention scores and significantly compress the KV cache. Based on these insights, `AnchorCoder` utilizing sparse attention in general, reduces the cache overhead via compression rather than extraction, further sustaining model performance.

VIII. Conclusion

We have conducted empirical research to explore the sparsity patterns of attention in code generation models. We designed a "needle in a haystack" experiment to demonstrate the ineffectiveness of current sparse attention methods in code generation. Based on these findings, we have proposed `AnchorCoder`, a novel approach which features token-wise and layer-wise anchor attention. designed to extract and compress the contextual information, and mitigate the issues of excessive superposition caused by the compression, respectively. Comprehensive experiments have demonstrated that `AnchorCoder` significantly reduces the KV cache overhead while maintaining model performance.

In the future, we plan to extend the application of Anchor Attention to a broader spectrum of attention mechanisms, including but not limited to multi-query attention (MQA) [70], multi-head latent attention (MLA) [71], and grouped-query attention (GQA) [72]. More experiments with larger code LLMs for the repository-level code generation are also planned.

References

[1] M. Chen et al., "Evaluating Large Language Models Trained on Code," 2021, *arXiv:2107.03374*.

[2] Y. Li et al., "Competition-level code generation with alphacode," *Science*, vol. 378, no. 6624, pp. 1092–1097, 2022.

[3] D. Fried et al., "Incoder: A generative model for code infilling and synthesis," 2022, *arXiv:2204.05999*.

[4] H. Huang et al., "ChatGPT for shaping the future of dentistry: the potential of multi-modal large language model," *Int. J. Oral Sci.*, vol. 15, no. 1, p. 29, 2023.

[5] Y. Yao, J. Duan, K. Xu, Y. Cai, Z. Sun, and Y. Zhang, "A survey on large language model (LLM) security and privacy: The good, the bad, and the ugly," *High-Confidence Comput.*, vol. 4, 2024, Art. no. 100211.

[6] OpenAI, "GPT-4 technical report," 2023, *arXiv:2303.08774*.

[7] X. Hou et al., "Large language models for software engineering: A systematic literature review," 2023, *arXiv:2308.10620*.

[8] E. Nijkamp et al., "Codegen: An open large language model for code with multi-turn program synthesis," 2022, *arXiv:2203.13474*.

[9] B. Roziere et al., "Code Llama: Open foundation models for code," 2023, *arXiv:2308.12950*.

[10] Q. Zheng et al., "Codegeex: A pre-trained model for code generation with multilingual benchmarking on humaneval-x," in *Proc. 29th ACM SIGKDD Conf. Knowl. Discovery Data Mining*, 2023, pp. 5673–5684.

[11] F. F. Xu, B. Vasilescu, and G. Neubig, "In-ide code generation from natural language: Promise and challenges," *ACM Trans. Softw. Eng. Methodol.*, vol. 31, no. 2, pp. 1–47, 2022.

[12] A. Vaswani et al., "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 5998–6008.

[13] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," 2014, *arXiv:1409.0473*.

[14] K. Hong et al., "Flashdecoding++: Faster large language model inference on GPUs," 2023, *arXiv:2311.01282*.

[15] Y. Yue, Z. Yuan, H. Duanmu, S. Zhou, J. Wu, and L. Nie, "WKVQUANT: Quantizing weight and key/value cache for large language models gains more," 2024, *arXiv:2402.12065*.

[16] Y. Wang and Z. Xiao, "Loma: Lossless compressed memory attention," 2024, *arXiv:2401.09486*.

[17] S. Chen, S. Wong, L. Chen, and Y. Tian, "Extending context window of large language models via positional interpolation," 2023, *arXiv:2306.15595*.

[18] I. Beltagy, M. E. Peters, and A. Cohan, "LongFormer: The long-document transformer," 2020, *arXiv:2004.05150*.

[19] G. Xiao, Y. Tian, B. Chen, S. Han, and M. Lewis, "Efficient streaming language models with attention sinks," 2023, *arXiv:2309.17453*.

[20] Z. Zhang et al., "H2o: Heavy-hitter oracle for efficient generative inference of large language models," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 37, 2023, pp. 34661–34710.

[21] S. Ge, Y. Zhang, L. Liu, M. Zhang, J. Han, and J. Gao, "Model tells you what to discard: Adaptive KV Cache Compression for LLMs," 2023, *arXiv:2310.01801*.

[22] B. Chen, T. Dao, E. Winsor, Z. Song, A. Rudra, and C. Ré, "Scatterbrain: Unifying sparse and low-rank attention," *Advances in Neural Inf. Process. Syst.*, vol. 34, pp. 17413–17426, 2021.

[23] C. Wang et al., "Teaching code LLMs to use autocompletion tools in repository-level code generation," 2024, *arXiv:2401.06391*.

[24] D. Shrivastava, H. Larochelle, and D. Tarlow, "Repository-level prompt generation for large language models of code," in *Proc. Int. Conf. Mach. Learn.*, PMLR, 2023, pp. 31693–31715.

[25] F. Zhang et al., "Repocoder: Repository-level code completion through iterative retrieval and generation," 2023, *arXiv:2303.12570*.

[26] N. Hurley and S. Rickard, "Comparing measures of sparsity," *IEEE Trans. Inf. Theory*, vol. 55, no. 10, pp. 4723–4741, 2009.

[27] L. Wang et al., "Label words are anchors: An information flow perspective for understanding in-context learning," 2023, *arXiv:2305.14160*.

[28] Q. Huang et al., "Opera: Alleviating hallucination in multi-modal large language models via over-trust penalty and retrospection-allocation," in *Proc. IEEE/CVF Conf. Comput. Vis. Pattern Recognit.*, 2024, pp. 13418–13427.

[29] Z. Zhang et al., "Anchor function: A type of benchmark functions for studying language models," 2024, *arXiv:2401.08309*.

[30] J. Pang, F. Ye, D. F. Wong, X. He, W. Chen, and L. Wang, "Anchor-based large language models," 2024, *arXiv:2402.07616*.

[31] Y. Kuratov et al., "Babilong: Testing the limits of LLMs with long context reasoning-in-a-haystack," 2024, *arXiv:2406.10149*.

[32] S. Chaudhury, S. Dan, P. Das, G. Kollias, and E. Nelson, "Needle in the haystack for memory based large language models," 2024, *arXiv:2407.01437*.

[33] H. Liu, C. Xu, and J. Liang, "Dependency distance: A new perspective on syntactic patterns in natural languages," *Phys. Life Rev.*, vol. 21, pp. 171–193, Jul. 2017.

[34] T. Lu et al., "From token to line: Enhancing code generation with a long-term perspective," 2025, *arXiv:2504.07433*.

[35] N. Elhage et al., "A mathematical framework for transformer circuits," *Transformer Circuits Thread*, vol. 1, no. 1, p. 12, 2021.

[36] J. Liu, C. S. Xia, Y. Wang, and L. Zhang, "Is your code generated by ChatGPT really correct? Rigorous evaluation of large language models for code generation," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 37, 2023, pp. 21558–21572.
[37] J. Austin et al., "Program synthesis with large language models," 2021, *arXiv:2108.07732*.
[38] S. Lu et al., "Codexglue: A machine learning benchmark dataset for code understanding and generation," 2021, *arXiv:2102.04664*.
[39] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, "A systematic evaluation of large language models of code," in *Proc. 6th ACM SIGPLAN Int. Symp. Mach. Program.*, 2022, pp. 1–10.
[40] F. Liu et al., "Exploring and evaluating hallucinations in LLM-powered code generation," 2024, *arXiv:2404.00971*.
[41] E. J. Hu et al., "Lora: Low-rank adaptation of large language models," 2021, *arXiv:2106.09685*.
[42] H. Peng, G. Li, Y. Zhao, and Z. Jin, "Rethinking positional encoding in tree transformer for code representation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2022, pp. 3204–3214.
[43] A. Kawasaki, A. Davis, and H. Abbas, "Defending large language models against attacks with residual stream activation analysis," 2024, *arXiv:2406.03230*.
[44] J. Dao, Y.-T. Lao, C. Rager, and J. Janiak, "An adversarial example for direct logit attribution: Memory management in Gelu-4l," 2023, *arXiv:2310.07325*.
[45] B. Millidge and S. Black, "The singular value decompositions of transformer weight matrices are highly interpretable," in *AI Alignment Forum*, 2022, p. 17.
[46] Y.-S. Chuang, Y. Xie, H. Luo, Y. Kim, J. Glass, and P. He, "Dola: Decoding by contrasting layers improves factuality in large language models," 2023, *arXiv:2309.03883*.
[47] Z. Sun, X. Du, F. Song, S. Wang, and L. Li, "When neural code completion models size up the situation: Attaining cheaper and faster completion through dynamic model inference," in *Proc. IEEE/ACM 46th Int. Conf. Softw. Eng.*, 2024, pp. 1–12.
[48] H. Touvron et al., "Llama 2: Open foundation and fine-tuned chat models," 2023, *arXiv:2307.09288*.
[49] S. Liu, X. Xie, J. Siow, L. Ma, G. Meng, and Y. Liu, "Graphsearchnet: Enhancing GNNs via capturing global dependencies for semantic code search," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 2839–2855, Apr. 2023.
[50] A. Chevalier, A. Wettig, A. Ajith, and D. Chen, "Adapting language models to compress contexts," 2023, *arXiv:2305.14788*.
[51] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLORA: Efficient finetuning of quantized LLMs," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 36, 2023, pp. 10088–10115.
[52] NVIDIA, "GPU performance background user's guide," 2022. [Online]. Available: https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html
[53] A. Ivanov, N. Dryden, T. Ben-Nun, S. Li, and T. Hoefler, "Data movement is all you need: A case study on optimizing transformers," *Proc. Mach. Learn. Syst.*, vol. 3, 2021, pp. 711–732.
[54] T. Dao, D. Fu, S. Ermon, A. Rudra, and C. Ré, "Flashattention: Fast and memory-efficient exact attention with IO-awareness," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 16344–16359.
[55] H. Le, Y. Wang, A. D. Gotmare, S. Savarese, and S. C. H. Hoi, "CODERL: Mastering code generation through pretrained models and deep reinforcement learning," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 35, 2022, pp. 21314–21328.
[56] R. Li et al., "StarCODER: May the source be with you!," 2023, *arXiv:2305.06161*.
[57] D. Guo et al., "Deepseek-Coder: When the large language model meets programming–the rise of code intelligence," 2024, *arXiv:2401.14196*.
[58] Q. Zhu et al., "Deepseek-Coder-v2: Breaking the barrier of closed-source models in code intelligence," 2024, *arXiv:2406.11931*.
[59] L. B. Allal et al., "SantaCoder: Don't Reach for the Stars!," 2023, *arXiv:2301.03988*.
[60] Y. Wang, H. Le, A. D. Gotmare, N. D. Q. Bui, J. Li, and S. C. H. Hoi, "Codet5+: Open code large language models for code understanding and generation," in *Proc. Conf. Empirical Methods Natural Lang. Process.*, 2023.
[61] X. Wei et al., "Towards greener yet powerful code generation via quantization: An empirical study," in *Proc. 31st ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2023, pp. 224–236.
[62] Z. Li et al., "Train big, then compress: Rethinking model size for efficient training and inference of transformers," in *Proc. Int. Conf. Mach. Learn.,* PMLR, 2020, pp. 5958–5968.
[63] L. Guo et al., "When to stop? Towards efficient code generation in LLMs with excess token prevention," in *Proc. 33rd ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2024, pp. 1073–1085.
[64] Q. Hu et al., "Active code learning: Benchmarking sample-efficient training of code models," *IEEE Trans. Softw. Eng.*, vol. 50, no. 5, pp. 1080–1095, May 2024.
[65] M. Nashaat and J. Miller, "Towards efficient fine-tuning of language models with organizational data for automated software review," *IEEE Trans. Softw. Eng.*, vol. 50, no. 9, pp. 2240–2253, Sep. 2024.
[66] J. W. Rae, A. Potapenko, S. M. Jayakumar, and T. P. Lillicrap, "Compressive transformers for long-range sequence modelling," 2019, *arXiv:1911.05507*.
[67] J. Mu, X. Li, and N. Goodman, "Learning to compress prompts with gist tokens," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 37, 2023, pp. 19327–19352.
[68] S. Ren, Q. Jia, and K. Q. Zhu, "Context compression for auto-regressive transformers with sentinel tokens," 2023, *arXiv:2310.08152*.
[69] B. Peng, J. Quesnelle, H. Fan, and E. Shippole, "Yarn: Efficient context window extension of large language models," 2023, *arXiv:2309.00071*.
[70] N. Shazeer, "Fast transformer decoding: One write-head is all you need," 2019, *arXiv:1911.02150*.
[71] X. Bi et al., "Deepseek LLM: Scaling open-source language models with Longtermism," 2024, *arXiv:2401.02954*.
[72] J. Ainslie, J. Lee-Thorp, M. de Jong, Y. Zemlyanskiy, F. Lebrón, and S. Sanghai, "GQA: Training generalized multi-query transformer models from multi-head checkpoints," 2023, *arXiv:2305.13245*.

Xiangyu Zhang received the M.D. degree from Nanjing University of Aeronautics and Astronautics in 2024. He is currently working toward the Ph.D. degree with the College of Computer Science and Technology of Nanjing University of Aeronautics and Astronautics. His research interests include code generation and model interpretability.

Yu Zhou (Senior Member, IEEE) received the B.Sc. and Ph.D. degrees in computer science from Nanjing University China, in 2004 and 2009, respectively. He is a Full Professor with the College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics (NUAA). Before joining NUAA in 2011, he conducted Postdoctoral Research on software engineering with the Politechnico di Milano, Italy. From 2015 to 2016, he visited the SEAL Lab with the University of Zurich, Switzerland, where he is also an Adjunct Researcher. His current research interests mainly generative models for software engineering, software evolution analysis, mining software repositories, and reliability analysis. He has been supported by several national research programs in China. For more information, see https://csyuzhou.github.io/.

Guang Yang received the M.D. degree in computer technology from Nantong University, Nantong, in 2022. He is currently working toward the Ph.D. degree with Nanjing University of Aeronautics and Astronautics, Nanjing. His research interest is AI4SE and he has authored or coauthored more than 20 papers in refereed journals or conferences, such as ACM *Transactions on Software Engineering and Methodology* (TOSEM), *Empirical Software Engineering*, *Journal of Systems and Software*, International Conference on Software Maintenance and Evolution (ICSME), and International Conference on Software Analysis, Evolution and Reengineering (SANER). For more information, see https://ntdxyg.github.io/.

Harald C. Gall (Member, IEEE) is the Dean of the Faculty of Business, Economics, and Informatics, University of Zurich. He is a Professor of software engineering with the Department of Informatics. He held visiting positions with Microsoft Research in Redmond, USA, and University of Washington, Seattle, USA. His research interests are software evolution, software architecture, software quality, and cloud-based software engineering. Since 1997, he has worked on devising ways in which mining repositories can help to better understand and improve software development.

Taolue Chen received the bachelor's and master's degrees from Nanjing University, China, and the Ph.D. degree from Vrije Universiteit Amsterdam, The Netherlands. He was a Junior Researcher (OiO) with the Centrum Wiskunde & Informatica (CWI). Currently, he is a Senior Lecturer with the School of Computing and Mathematical Sciences, Birkbeck, University of London. He had been a Postdoctoral Researcher with the University of Oxford. His research areas include software engineering, programming language, and verification. His present research focus is neuro-symbolic software engineering. He has published about 150 papers in journals and conferences such as POPL, LICS, CAV, OOPSLA, ICSE, ESEC/FSE, ASE, ISSTA, ETAPS (TACAS, FoSSaCS, ESOP, FASE), NeurIPS, ICLR, IJCAI, AAAI, EMNLP, and IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, ACM *Transactions on Software Engineering and Methodology*, Empirical Software Engineering, ACM *Transactions on Computational Logic*, *Information and Computation,* and *Logical Methods in Computer Science*. He won the Best Paper Award of SETTA'20, the 1st Prize in the CCF Software Prototype Competition 2022, the QF_Strings (Single Query Track) at the International Satisfiability Modulo Theories Competition 2023, and ACM SIGSOFT Distinguished Paper Award in 2024. He has served editorial board or program committee for various international journals and conferences. For more information, see https://chentaolue.github.io/.