TSINGHUA SCIENCE AND TECHNOLOGY
ISSN 1007-0214 03/38 pp1365–1380
DOI: 10.26599/TST.2025.9010166
Volume 31, Number 3, June 2026

Review

Efficient Inference for Edge Large Language Models: A Survey

Guanyu Cai, Ruiming Tian, Lang Yang, Yunzhe Jia, Lingkun Li, and Jiliang Wang*

Abstract: Large language models (LLMs) have demonstrated remarkable capabilities in natural language processing. Their massive computational and memory requirements often necessitate cloud-based deployment, introducing challenges related to cost, latency, privacy, and network reliability. Deploying on-device LLMs alleviates these challenges, but is hindered by the severe resource constraints of edge hardware. This survey reviews efficient inference techniques for edge LLMs, with a focus on two key strategies of speculative decoding and model offloading. We categorize strategies into single-device and multi-device types, systematically analyzing the principles, recent advancements, implementations, and support within edge frameworks. Finally, we highlight the open challenges and future research directions that will advance the field of edge LLM inference.

Key words: large language models (LLMs); edge computing; on-device; speculative decoding; model offloading

1 Introduction

Large language models (LLMs) have revolutionized natural language processing (NLP), showing remarkable capabilities in language understanding, generation, and reasoning. LLMs have attracted significant interest from both academia and industry due to their superb performance. Several open-source foundation LLMs have been released, including Meta's LLaMA series[1–3], DeepSeek AI's DeepSeek series[4, 5], and Google's Gemma series[6–8]. Although scaling up model size and training data enhances performance, most personal computers lack the necessary memory capacity and processing power required to run LLMs locally. Consequently, many companies have developed cloud-based tools like ChatGPT[9], Copilot[10], and Gemini[11], which are built on foundation models and delivered as services.

Cloud-based LLMs pose challenges in scenarios that require low service cost, low latency, strong privacy, or offline operation. Thus, there is a growing demand to deploy LLMs on resource-constrained edge platforms, such as smartphones, Internet of Things (IoT) devices, embedded systems, and vehicles. Deploying LLMs locally offers several key advantages. Edge LLMs provide a stable service with no internet connectivity. Moreover, users can build context or environment-specific applications without privacy concerns[12]. For online LLMs service providers like OpenAI, part of the computational load can be offloaded to edge devices to improve throughput.

However, deploying on-device LLMs is challenging due to the severe resource constraints of edge environments: limited computational power, memory capacity, and memory bandwidth. These constraints directly conflict with LLMs' massive parameter requirements and computationally intensive inference processes. For example, the LLaMA family ranges from 1 B to 405 B parameters, but a typical mobile phone can only run a 7 B model at approximately 5

• Guanyu Cai, Lang Yang, Yunzhe Jia, and Jiliang Wang are with School of Software, Tsinghua University, Beijing 100190, China. E-mail: caigy25@mails.tsinghua.edu.cn; yanglang21@mails.tsinghua.edu.cn; jiayz22@mails.tsinghua.edu.cn; jiliangwang@tsinghua.edu.cn.
• Ruiming Tian and Lingkun Li are with School of Software Engineering, Beijing Jiaotong University, Beijing 100044, China. E-mail: ruimingtian@bjtu.edu.cn; lkli@bjtu.edu.cn.
* To whom correspondence should be addressed.
Manuscript received: 2025-07-09; revised: 2025-10-07; accepted: 2025-11-03

tokens per second[13]. These constraints make straightforward deployment on edge devices impractical.

Several techniques have been proposed to reduce model parameters and inference overhead[14], including pre-deployment optimizations and runtime optimizations. Pre-deployment optimizations, such as pruning and quantization, have been extensively reviewed. This survey instead focuses on runtime optimizations. At runtime, large LLMs require loading parameters into memory and performing intensive computations. Speculative decoding[15, 16] increases computational unit utilization, whereas model offloading reduces[17, 18] memory and computation demands. These techniques are particularly promising for enabling practical edge LLM inference. Speculative decoding addresses the sequential nature of autoregressive generation. It leverages smaller, faster draft models to propose candidate tokens, which are then efficiently verified in parallel by the large target model. It significantly reduces inference latency. Model offloading strategically partitions the LLM. It keeps frequently accessed or latency-critical components on the edge's high-speed computation and memory units and fetches larger, less active parts from nearby resources (e.g., low-speed computation and memory units, edge servers, or the cloud). This strategy alleviates memory and computation pressure on the device. Overall, this survey highlights these two techniques as promising paths toward efficient edge LLMs.

While several surveys cover efficient LLMs, they lack a dedicated focus on the unique constraints and opportunities of edge deployment[14, 19–23]. Reference [21] concentrates on common speculative decoding techniques. References [24, 25] focus on model offloading techniques. References [19, 20] cover a wide range of inference techniques, but do not emphasize edge-specific advances.

This survey aims to fill the gap by providing a comprehensive review with a focus on efficient inference techniques for LLMs deployed on edge platforms, emphasizing speculative decoding and model offloading. We systematically analyze the fundamental principles, recent algorithmic advancements, and edge-specific designs. Furthermore, we investigate the support for these technologies in different edge frameworks. We also highlight open challenges and future directions.

The remainder of this survey is structured as follows: Section 2 introduces the basic knowledge about LLMs' architecture and inference process, edge device features, and optimization strategies. Sections 3 and 4 provide a comprehensive review of speculative decoding and model offloading approaches, respectively. Section 5 discusses open challenges and future directions. Section 6 summarizes the main contributions of this survey.

2 Preliminary

2.1 LLMs

Language models capture the statistical structure of natural language and estimate the probability distribution of the next word. Nowadays, most powerful language models are deep learning models. Recently, researchers have found that enlarging deep learning models improves language modeling performance and unlocks the ability for multi-step problem-solving[4, 9]. Models at that scale are commonly referred to as large language models. They now facilitate many state-of-the-art systems in natural language processing and beyond.

LLMs' success is largely attributed to the Transformer architecture[26], which provides a scalable backbone for learning language's contextual representations from massive corpora. Mainstream LLMs are composed of multiple stacked Transformer decoder blocks, where each Transformer decoder block consists of a self-attention (SA) block and a feed-forward neural network (FNN). A typical LLM's computation flow is shown in Fig. 1. LLM first embeds the input to get queries Q, keys K, and values V matrices,

$$Q = XW^Q,\ K = XW^K,\ V = XW^V \tag{1}$$

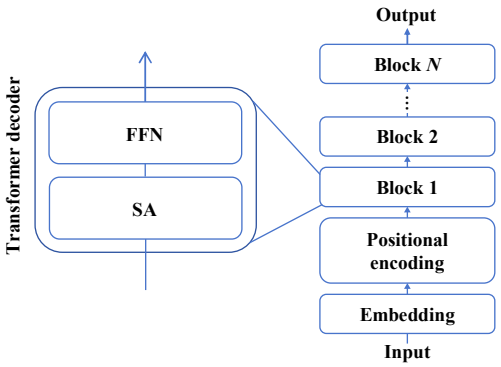


Fig. 1 Example of LLM architecture.

where X is the input token sequence, W^Q, W^K, and W^V are learnable projection matrices. It then encodes the input's position information. After that, each block first applies the self-attention to compute contextual relationships between all tokens in a sequence via Q, K, and V matrices,

$$\text{Attention}(Q,K,V) = \text{softmax}\left(\frac{QK^{\mathrm{T}}}{\sqrt{d_k}}\right)V \tag{2}$$

where d_k is the last dimension size of Q. Then the feed-forward network (FFN) applies linear transformations with an activation function to the attention result. There are some additional components, including layer normalization, residual connections, and positional embeddings. The implementation may have some differences in different LLMs, but the self-attention and feed-forward network are the most important components.

2.2 Autoregressive inference

Most LLMs adopt the autoregressive decoding method to generate the output. In autoregressive generation, tokens are produced sequentially. In each step, LLMs input both the original input and all tokens generated so far to get the entire context and predict the next token. As the sequence becomes longer, this repeated computation leads to a substantial increase in inference latency. To speed up, modern LLMs employ the key-value (KV) cache mechanism, which accelerates autoregressive decoding by storing the previously computed key (K) and value (V) within each self-attention layer. The model can directly reuse cached KV, avoiding redundant computation of calculated KV. With KV cache, the inference workflow is divided into two phases, as shown in Fig. 2.

Prefilling stage: For example, the input is "I am a bird". The LLM processes all the input tokens to produce the next token "I". During the process, the LLM also calculates intermediate features Q, K, and V. The K and V are cached for reuse in the decoding stage. The main computation at this stage comes from matrix-matrix multiplication, which is computation-bound because the computation complexity scales quadratically with sequence length.

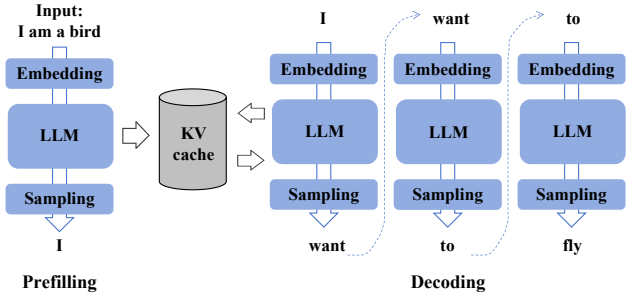


Fig. 2 LLM's autoregressive inference.

Decoding stage: The LLM iteratively generates tokens, and each step depends on prior outputs. For example, the LLM outputs "want", "to", and "fly" one by one. During decoding, it fetches the old KV cache, generates new output, and updates the KV cache. This phase is memory-bandwidth-bound. The reason is as follows: First, the sequential dependency can not be parallelized. Second, KV cache reads/writes dominate memory traffic. Third, the main computation at this stage comes from matrix-vector multiplication, which has low computational overhead and high access overhead.

The decoding stage constitutes most of the total inference latency, making it the most important optimization target.

2.3 Edge characteristics

We compare the hardware specifications between edge devices and data center graphics processing unit (GPUs) in Table 1. Typically, edge platforms, including commercial-level computers, mobile, IoT, and embedded devices, have several constraints as follows.

Memory limitations: dynamic random access memory (DRAM) capacities range from 16 GB (high-end smartphones) to less than 4 GB (low-end microcontrollers), while a 7B LLM needs over 14 GB of memory for half-precision floating-point precision, i.e., FP16 precision. The memory bandwidth is also limited, e.g., mobile's 60 GB/s vs. A100's 2 TB/s. Low memory bandwidth becomes a bottleneck for LLMs,

Table 1 Hardware comparison of edge devices and data center GPUs.

Device	Memory capacity (GB)	Memory bandwidth (GB/s)	Peak performance (FP16 TFLOPS)
Raspberry Pi 5	4	~17.0	~0.1
Snapdragon 8 Gen 3	8–16	~60.0	~3.0 (GPU)
NVIDIA Jetson TX2	8	59.7	1.3
NVIDIA Jetson Orin NX	16	102.4	~6.0
NVIDIA Jetson AGX Orin	32	204.8	~65.0
NVIDIA A100 (80 GB)	80	2039.0	624.0
NVIDIA H100 (80 GB)	80	3350.0	1979.0

since KV cache access during decoding requires high throughput.

Compute constraints: Edge devices have weak parallelization capabilities; e.g., the floating-point operations per second (FLOPS) of smartphones is on the order of 1 Tera FLOPS (TFLOPS) (for a high-end system on a chip (SoC)), which is significantly lower than data-center GPUs capable of hundreds of TFLOPS.

These constraints directly conflict with LLM demands: multi-GB model weights, 100+ GB/s KV cache bandwidth, and sustained compute for decoding.

Operational characteristics: Although edge deployment has several limitations, there are opportunities. On-device LLMs optimize memory and computing overhead for edge devices. What's more, edge deployment like an AI assistant commonly has no user-level concurrency, and edge devices can cooperate in typical environments. Researchers can improve the performance of edge LLMs using these properties.

2.4 Optimization strategies

To overcome edge limitations, two techniques are very suitable.

Speculative decoding aims to tackle the memory bandwidth limitations[15]. It mitigates decoding latency by using a small draft model to propose token sequences rapidly. The target LLM then verifies multiple tokens in parallel, reducing decoding steps. This method is ideal for edge batch-1 workloads.

Model offloading aims to tackle the memory size and computation capacity limitations[27]. It partitions a large model and offloads portions of the parameters and computations to other local storage/computational units or remote devices. This enables the host device to run a larger model and improves inference speed.

3 Speculative Decoding

Speculative decoding[15] shifts the paradigm of autoregressive inference in LLMs. This section introduces its core mechanism, evolution, edge-specific innovations, integration in mainstream frameworks.

3.1 Core mechanism

Speculative decoding consists of two steps: token generation and verification. The core idea involves employing a smaller language model (draft model) to quickly predict several tokens. The target language model then validates these predicted tokens in parallel. This approach enables the target LLM to generate multiple tokens at once, instead of only one token per step in the traditional autoregressive decoding approach. As shown in Fig. 3, in the drafting phase, the drafter autoregressively outputs "want", "to", and "jump" in sequence. The input and output are then concatenated into "I am a bird, I want to jump". Finally, the target LLM takes the whole sequence as input, verifies "want", "to", and "jump", and revises "jump" to "fly". We formally introduce the drafting and decoding phase in the following.

Drafting phase: At each decoding step, starting from the current context $x_1, x_2, \ldots, x_t$, the method first employs a computationally efficient draft model to predict γ future tokens $y_1, y_2, \ldots, y_\gamma$,

$$y_i \sim q(\cdot \mid x_{<t}, y_{<i}),\ i = 1, 2, \ldots, \gamma \tag{3}$$

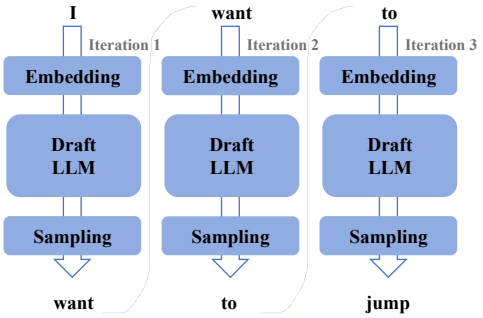
where $q(\cdot \mid x_{<t}, y_{<i})$ denotes the probability distribution of y_i. The draft model is typically smaller and more efficient than the target model.

Verification phase: The target model processes the sequence $x_{<t}, y_{<\gamma+1}$ in a single forward pass and computes the following distributions:

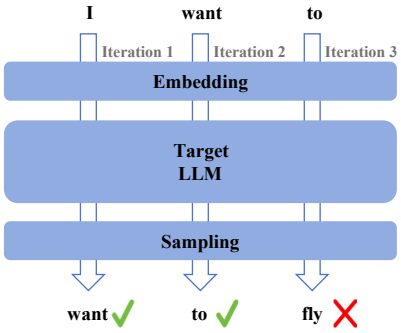
$$p_i(\cdot) \triangleq p(\cdot \mid x_{<t}, y_{<i}),\ i = 1, 2, \ldots, \gamma+1 \tag{4}$$

The verification has two steps:

(1) Token acceptance. First, it computes the



(a) Phase 1: Draft tokens autoregressively



(b) Phase 2: Verify tokens in parallel

Fig. 3 Overview of speculative decoding.

acceptance probability $\alpha_i = \min\left(1, \frac{p_i(y_i)}{q_i(y_i)}\right)$. Second, token y_i is accepted if $u_i < \alpha_i$, where $u_i \sim$ Uniform(0, 1). Third, if token y_i is rejected, the algorithm discards all tokens $y_{\geqslant i}$ and samples a replacement token $\tilde{y}_i$ from the residual distribution and stops,

$$\tilde{y}_i \sim p_{\text{resid}}(\cdot) = \frac{\max(0,\ p_i(\cdot) - q_i(\cdot))}{\sum_{y'} \max(0,\ p_i(y') - q_i(y'))} \tag{5}$$

This sampling strategy has been proven to guarantee that the output follows the target model's distribution[15].

(2) Additional sampling. Finally, if all draft tokens are accepted, the target model samples an additional token $y_{\gamma+1}$ from $p_{\gamma+1}(\cdot)$.

3.2 Drafting

The two key factors in drafting are accuracy and latency, and various strategies aim to strike a balance between these factors.

3.2.1 Model-based methods

Most drafting methods employ language models to generate draft tokens. There are two main paradigms: independent drafters and dependent drafters.

Independent drafter: The drafter model is a small LLM independent of the larger verification LLM. SpecDec[28] proposes to use an independent model for drafting. The model is a trained-from-scratch non-autoregressive Transformer that generates multiple tokens in a single step. SpecDec++[29] boosts speculative decoding via adaptive candidate token lengths. It incorporates an acceptance predictor into the draft model and stops drafting when the rejection likelihood exceeds a specified threshold.

Instead of training a drafter model from scratch, some works utilize a smaller LLM from the same family for drafting[15, 30, 31]. For instance, LAMDA-8B can be used as a drafter model and LAMDA-137B as the target model, achieving an acceptance ratio of 0.74[15]. Using a drafter and a target model from the same off-the-shelf LLM family has several advantages, including no training cost and higher alignment between the drafter and verifier.

Recent works focus on aligning the drafter to the target model. A well-aligned drafter has a higher token acceptance rate. The knowledge distilling technique is widely used[32−35]. DistillSpec[32] utilizes on-policy data generated from the draft model and tailors the divergence function to the task and decoding strategy. SD^2[33] leverages self-data distillation and fine-grained weight sparsity to produce efficient, well-aligned draft models. The approach in Ref. [35] employs collaborative interaction between student and teacher models to dynamically generate training data with an aligned inference distribution.

Dependent drafter: It is also called self-drafting. Although independent drafters have simpler architectures, they do not fully use the intermediate output in the drafting process, leading to information waste. To address this issue, dependent drafters leverage the target model itself to generate draft tokens[36−41].

One prominent strategy modifies the model architecture to generate tokens in parallel. For example, Blockwise Decoding[36] and Medusa[39] achieve this by adding additional FFN heads to the Transformer decoder. These lightweight layers minimize computational overhead while accelerating draft token generation. Another approach in this category is multi-token prediction (MTP)[42], which modifies the model to predict multiple future tokens simultaneously during both training and inference.

Other works adopt early exiting and layer skipping techniques to accelerate the drafting process. PPD[43] proposes a subprocesses exit early method during decoding, allowing drafting tokens ahead of time. Similarly, Ref. [44] introduces an adaptive layer-skipping mechanism during inference to improve drafting efficiency. These methods demonstrate that modifying the self-drafting model can improve generation speed without relying on independent drafters.

3.2.2 N-gram-based methods

Some approaches exploit N-gram patterns to generate draft tokens at low cost[45−47]. LOOKAHEAD[45] utilizes a trie tree to cache n-gram tokens from context. This enables rapid draft retrieval by leveraging preceding tokens. ANPD[45] utilizes an adaptive n-gram component during drafting, dynamically adjusting to the interactive context. N-Grammys[47] extracts N-grams obtained from the model weights and the context and exploits the top-k candidates.

3.2.3 Retrieval-based methods

Other approaches retrieve context or a historical corpus to assist draft token generation[48−52]. LLMA[48] applies retrieval-augmented generation (RAG) to extract text from the reference documents and verifies it in a single step. REST[49] retrieves draft tokens from a

comprehensive datastore to improve the performance. Speculative RAG[50] accelerates RAG by using a large generalist LLM to verify multiple drafts produced by a smaller, distilled specialist LLM in parallel. RAPID[51] improves long-context inference speed and quality by an LLM-based RAG drafter. RaLMSpec[52] proposes a framework to accelerate iterative RAG using speculative retrieval with parallel verification.

3.3 Verification

Once draft tokens are generated, the speculative decoding organizes them and leverages the target model to verify structured tokens. To accelerate the verification process, various structures are proposed, including sequences, trees, and graphs, as illustrated in Fig. 4.

Drafts are naturally organized as sequences. With the sequence structure, target models have to verify one sequence at a time. Fast Inference[15] proposes speculative sampling to improve the acceptance rate while maintaining the output distribution. To further accelerate verification, SpecInfer[16] organizes multiple draft sequences into a token tree. It then proposed a tree-based masking attention algorithm to verify drafts in parallel, as shown in Fig. 5. The method significantly improves verification efficiency compared to sequence-based. GSD[53] observes that candidate drafts often share common token sequences and employs directed acyclic graphs to structure drafts. It merges recurring token sequences and further reduces the computational overhead.

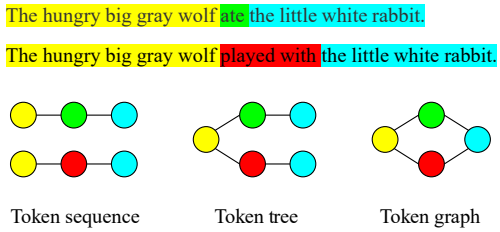


Fig. 4 Illustration of sequence-, tree-, and graph-based draft structure.

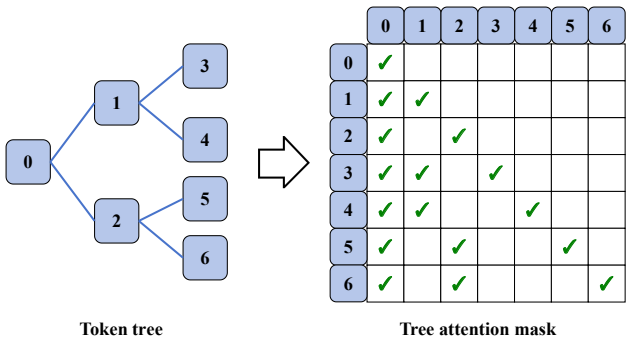


Fig. 5 Overview of tree-based verification.

3.4 Edge methods

While most existing studies explore speculative decoding on resource-rich cloud platforms using enterprise-grade computing cards (e.g., A100), an increasing number of works aim to optimize deployment on edge-grade (e.g., RTX 4090, Jetson TX2, and Snapdragon 888) platforms. We categorize edge methods into single-device and multi-device approaches. A summary of these methods is provided in Table 2.

Single-device: The limited GPU memory of edge devices forces existing systems to offload model weights to CPU memory, which incurs significant I/O latency during CPU-GPU communication. SpecOffload[54] uses latent GPU resources to embed speculative decoding into the offloading pipeline, accelerating inference with nearly zero additional cost. It interleaves target and draft model execution and introduces a planner to optimize parameter and tensor placement. To deploy on memory-constrained devices, SpecMemo[55] models speculative decoding's memory consumption to derive a memory budget while preserving speedup. It further introduces batched speculative decoding to scale across multiple constrained GPUs. S3D[56] proposes skipping simultaneous speculative decoding, which is based on simultaneous multi-token decoding and mid-layer skipping. It requires minimal architecture changes and training data. It theoretically calculates the optimal hyperparameters to balance speed and memory usage. To reduce the memory consumption and training cost for the draft model, PPD[57] proposes parallel prompt decoding, requiring only 0.0002% trainable parameters and 0.0004% runtime memory overhead. It maintains output quality while achieving high acceptance rates for long-distance predictions. It further proposes a dynamic sparse tree technique to adapt to heterogeneous hardware constraints. With parameter offloading on memory-constrained devices, hundreds of tokens can be processed within the same time as just one token. Thus, SpecExec[58] leverages this by optimizing draft tree structures for large token budgets, achieving acceptance of over 10 tokens per verification step. Ghidorah[59] exploits diverse processing units in edge devices to speed up inference. It introduces a

Table 2 Edge speculative decoding methods.

Method		Drafter	Target model	Experimental device	Feature	Max speedup
Single-device	SpecOffload[54]	Small LLM	Mixtral-8x (7B–22B)	RTX 4090	Offloading and pipeline	2.54×
	SpecMemo[55]	Self drafting	LLaMA-2-70B	Titan RTX, AMD MI250	Modeling memory footprint	2.00×
	S3D[56]	Self drafting	LLaMA-2 (7B–13B)	RTX 3060, A10G	Early exiting	2.00×
	PPD[57]	Self drafting	Vicuna (7B–13B)	RTX 4090, A100	Hardware-aware dynamic sparse tree	2.49×
	SpecExec[58]	Small LLM	LLaMA-2-70B, LLaMA-3-70B, Mixtral-8x-7B	RTX 4090/4060/3090/2080Ti, A100	Most probable continuations, huge draft tree	18.70×
	Ghidorah[59]	Self drafting	Vicuna-7B	NVIDIA Jetson NX	Hetero-core model parallelism, architecture-aware profiling	7.60×
	EdgeLLM[60]	Small LLM	Vicuna-13B, LLaMA-2-13B	Xiaomi 10/12, Jetson TX2/Orin	Continuous drafting, drafting-verification pipeline	9.30×
Multi-device	DSI[61]	Small LLM	Vicuna (7B–13B), Phi-3-14B	A100	Distributed speculative inference	1.92×
	AMUSD[62]	Small LLM	LLaMA-3-8B	A100	Asynchronous multi-device, speculative decoding	1.96×
	Dovetail[63]	Small LLM	LLaMA-2-chat-7B	RTX 2080 SUPER, RTX 3090	Drafting on GPU and verifying on CPU	2.77×
	SLED[64]	Small LLM	LLaMA (11B–70B)	Raspberry Pi 4b/5, Jetson Orin Nano, A100	Lightweight edge device support	2.20×
	SpecEdge[65]	Small LLM	Qwen-3 (14B–32B)	RTX 4090, A100	Proactive edge drafting	2.22×
	FSECD[66]	Small LLM	Vicuna (7B–13B), LLaMA-2-7B	Jetson Nano, RTX 2080 Ti, A100	Early exiting in target model	1.21×

hetero-core model parallelism architecture to adapt to resource requirements and an architecture-aware profiling method to determine an optimal partitioning strategy. EdgeLLM[60] proposes an efficient draft tree generation and verification method tailored for edge devices. It introduces a continuous drafting strategy and drafting-verification pipeline to fully utilize available computational resources.

Multi-device: Traditional speculative decoding techniques consider only a single device or GPU. Recent works propose extending them to multiple devices or GPUs. DSI[61] designs the first distributed speculative decoding algorithm to utilize multiple GPUs, which is faster than single-GPU speculative decoding. It alleviates the memory limitation of the edge device by distributing the draft and target model to different GPUs. AMUSD[62] parallelizes draft and verification phases on different GPUs via an asynchronous execution pipeline to maximize hardware utilization. It greatly reduces idle time and achieves faster inference. Old personal and edge devices from the pre-LLMs era have weak GPUs but relatively stronger CPUs. Thus, Dovetail[63] deploys the draft model on the GPU and the target model on the CPU. It designs the draft model and token tree structure to align with the heterogeneous hardware, achieving better performance than CPU-only and GPU-only methods. SLED[64] decouples drafting on local devices from verification on the server. It shares a single target model across devices to minimize server memory. SpecEdge[65] represents an edge-server cooperative speculative decoding framework, which only exchanges token outputs over the network. It employs proactive edge drafting when the server verifies received tokens to increase server-side throughput. Fast spec edge-cloud decoding (FSECD)[66] concurrently proposes the edge-server cooperative speculative decoding framework, which introduces early exits in the target model. Tokens are generated mid-verification to utilize idle time and enhance throughput.

Insights: Table 2 reported speedups span a wide range, from 1.2× to 18.7×. This variance mainly arises from differences in model alignment, hardware characteristics, task configurations, and baseline. Among these methods, SpecExec[58] reports the highest speedups, mainly due to its parameter offloading deployment setting. In this setting, the memory bottleneck is more pronounced, limiting overall throughput. This property makes the total inference time largely insensitive to the number of tokens processed per forward pass. SpecExec generates a larger token tree and achieves unusually high acceleration factors. Overall, speculative decoding speedup is a function of model alignment, hardware-software co-design, and workload dynamics. Jointly optimizing these factors will achieve better performance.

3.5 Trade-off analysis

While speculative decoding substantially improves

decoding speed by mitigating the memory bandwidth bottleneck, it introduces trade-offs in memory footprint, energy consumption, and latency. First, both self-drafting and small LLM-based methods incur additional memory capacity requirements, especially the small LLM-based methods. For instance, a configuration with a 7B draft model and a 13B target model increases the total memory footprint by around 50% compared to deploying the 13B model alone. This capacity overhead limits deployment on devices with tight memory budgets.

Moreover, speculative decoding may increase the energy consumption due to redundant computation. Although the total inference time may decrease, the energy per token does not always improve proportionally when verification rejects many draft tokens.

These trade-offs highlight that speculative decoding should be viewed not merely as a latency optimization technique but as part of a multi-objective optimization problem involving latency, energy, and memory.

3.6 Framework support

Mainstream edge LLM frameworks are often differentiated by their primary deployment targets. Frameworks such as llama.cpp[67], MNN[68] and MLC-LLM[69] prioritize efficiency on consumer-grade or resource-constrained devices. In contrast, high-performance frameworks like TensorRT-LLM[70] and vLLM[71] cater to powerful edge devices (e.g., NVIDIA Jetson series) and data center environments. Table 3 summarizes speculative decoding support across these frameworks, revealing variations driven by architectural trade‑offs and design objectives.

High-performance frameworks like TensorRT-LLM[70] and vLLM[71] offer the most extensive support, targeting scenarios that demand maximum throughput. TensorRT-LLM is the most versatile, with native implementations for nearly all mainstream algorithms, from standard draft-model approaches[15] to dependent drafters, like Medusa[39], EAGLE[72], and MTP[73]. Its attention operators feature custom masks to fully support tree-based verification and advanced features like Paged KV cache[71]. vLLM, while also supporting a wide range of dependent drafters, prioritizes high-batch-size optimization. Consequently, it does not implement tree-based attention, restricting its Medusa and EAGLE implementations to verifying a single draft sequence at a time.

Resource-constrained frameworks adopt selective implementation strategies. MLC-LLM[69] is a universal compilation framework that focuses on providing robust support for dependent drafter methods, such as Medusa and EAGLE. Its key advantage lies in a specialized tree-attention operator for its Paged KV cache. llama.cpp[67] is widely used on consumer CPUs, offering a straightforward implementation of standard and N-gram-based speculative decoding. It utilizes its batch processing APIs and a unified KV cache design to implement tree decoding. MNN[68] targets memory-limited mobile devices by eschewing standard speculative decoding, instead implementing the lightweight Lookahead[46] and MTP to minimize memory overhead while maintaining acceleration benefits.

Table 3 Support for speculative decoding variants in mainstream frameworks. The symbols ✓ and ○ denote full and partial support, respectively. Here, "partial support" (○) indicates implementations that lack tree-based attention, thus restricting verification to a single draft sequence at a time.

Framework	Speculative decoding variant				
	Standard	N-gram/ Lookahead	Medusa	EAGLE	MTP
TensorRT-LLM[70]	✓	✓	✓	✓	✓
vLLM[71]	✓	✓	○	○	✓
MLC-LLM[69]	✓	−	✓	✓	−
llama.cpp[67]	✓	✓	−	−	−
MNN[68]	−	✓	−	−	✓

4 Model Offloading

Given the massive number of parameters and the growing KV-cache during auto-regressive decoding in LLMs, deploying the full model on a single device is often infeasible. Offloading addresses this limitation by partitioning the model or distributing the KV-cache across different storage devices, while spreading the workload across heterogeneous devices (e.g., edge and cloud). These devices collaborate by coordinating computation and exchanging intermediate data over networks or high-speed interconnects, subject to device capabilities, bandwidth, and latency constraints.

In general, offloading for LLM inference can be categorized into two types (Fig. 6): (1) Single-device offloading (Section 4.1), where heterogeneous on-board processors (e.g., CPU, GPU, neural processing unit (NPU)) within a single device collaboratively execute inference tasks to optimize resource utilization

and latency. (2) Multi-device collaboration (Section 4.2), where the inference workload is partitioned and executed across multiple devices. This includes both coordination among resource-constrained end devices through model or pipeline partitioning, and joint execution between edge and cloud with intermediate activations exchanged.

4.1 Single-device offloading

The primary bottleneck for running LLMs on edge devices is the mismatch between compute and memory. GPUs provide the necessary computation but are constrained by limited video random access memory (VRAM), which is consumed by the model weights and KV cache, while CPUs offer abundant DRAM but lack the computational power. Thus, extensive research has focused on GPU–CPU collaborative inference. However, as edge devices aim to run larger models, DRAM also becomes insufficient due to the growing KV-cache during auto-regressive generation. In this case, further collaboration between VRAM, DRAM, and external storage is required.

To enable KV-cache-accelerated inference of larger models on edge devices with consumer-grade GPUs, researchers offload both computation and caching from GPU VRAM to the CPU and system memory. A naive strategy is to compute the *Q*/*K*/*V* projections on the GPU, store KV cache in DRAM, perform attention on the CPU, and return to the GPU for the following calculations. While this reduces VRAM usage, it introduces CPU bottlenecks, GPU idle time, and heavy IO traffic. FastDecode[17] mitigates this by splitting each batch into two sub-batches and pipelining GPU (for *Q*/*K*/*V* calculation) and CPU (for attention matrix calculation) work across layers, yet bubbles remain, and placing all KV in DRAM underutilizes GPU memory. NEO[18] adopts asymmetric offloading. It retains part of the KV cache and some attention on the GPU and employs an online scheduler. Given request rate, prompt length, and expected output length, it greedily solves a constrained problem to maximize throughput without violating end-to-end latency bounds. vLLM[71] introduces the PagedAttention mechanism, which manages the KV-cache in a paged manner to reduce GPU memory fragmentation and enable more efficient utilization of VRAM. On edge devices, long-context inference faces the challenge that the KV cache grows with the context length. To address this, some studies dynamically discard part of

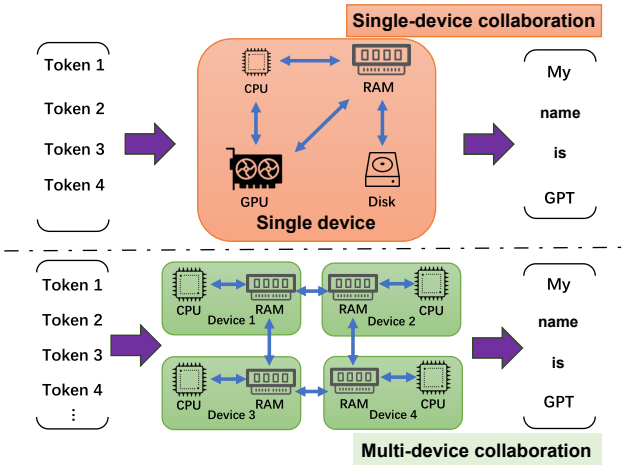


Fig. 6 Single-device collaboration and multi-device collaboration.

the KV cache during autoregressive generation, enabling larger-scale inference under limited storage.

While most existing studies focus on standard Transformer models, mixture-of-experts (MoE) models introduce unique structural characteristics that pose additional challenges. Simply offloading expert parameters to CPU memory while caching only a small subset in GPU memory results in low cache hit rates and increased inference latency. To overcome this, Zhang et al.[74] leveraged expert activation path statistics to achieve efficient deployment and scheduling of expert modules across CPU and GPU. MoEpic[75] predicts and prefetches the experts to be activated in the next layer during each inference step. Since the top segments of cached experts do not require reloading, the loading time is reduced, enabling efficient overlap between computation and data transfer. Building on this mechanism, MoEpic introduces a divide-and-conquer algorithm based on fixed-point iteration to achieve adaptive cache configuration.

When demands exceed the available VRAM and DRAM, even after using both, solutions often rely on external storage. This creates an opportunity to place model weights on slower but high-capacity storage devices. DeepSpeed-ZeRO[76] Inference adopts such an approach by storing model weights persistently in DRAM or non-volatile memory express (NVMe) and streaming each layer into GPU memory on demand. However, this strategy does not fully exploit the cooperative potential of DRAM, NVMe, and VRAM. To address this, FlexGen[77] formulates offloading optimization as a linear programming problem,

deriving near-optimal strategies that maximize throughput and deliver significant improvements over DeepSpeed-ZeRO Inference and Hugging Face Accelerate.

4.2 Multi-device collaboration

Devices participating in offloading are organized into two types: the first type is edge-side devices (non-enterprise-grade devices, including smartphones, personal computers, and embedded systems), and the second type is cloud-side devices (enterprise-grade devices such as large-scale GPU computing clusters).

Multi-device collaborative inference can be achieved through two principal methods. The first is to orchestrate multiple idle, trusted edge devices into a heterogeneous edge-computing pipeline to share compute and memory pressure. The second is to offload part of the workload to the cloud to reduce the edge-side burden while avoiding excessive load on the cloud, thereby maintaining adequate throughput.

In multi-device edge settings, pipelined inference can exploit idle on-device compute. The key objectives are either maximizing throughput or minimizing end-to-end latency. However, heterogeneous compute, memory capacities, and inter-device bandwidth make uniform partitioning ineffective and cause a combinatorial explosion in the search space of feasible assignments. PipeEdge[78] partitions at the layer granularity. Each device hosts a subset of layers, and memory limits, compute capability, and link bandwidth. The resource utilization is optimized via a dynamic-programming formulation. Focusing instead on sparse-request regimes where end-to-end latency dominates, Matching[27] models the assignment as a matching problem to minimize bubble time by jointly accounting for computation and transfer costs. From a tensor-parallel perspective, Galaxy[79] shards attention by heads so that each device generates a partial token and then sum-reduces across devices; once a token is reduced, it immediately proceeds to the FFN, overlapping communication with computation. Galaxy minimizes the per-layer makespan under memory constraints and uses a two-stage heuristic search to approach the latency optimum.

Running LLMs at the edge keeps data local and strengthens privacy, but tight memory and compute budgets restrict model size and throughput. By contrast, cloud data centers can host larger models and execute them quickly. But centralized resources saturate under surging demand, and introduce non-negligible communication latency. These complementary strengths and weaknesses motivate collaborative edge-cloud inference. The core challenge is task partitioning and placement. The planner must jointly optimize end-to-end response time for individual requests, cloud-side throughput and load, and the network traffic incurred by exchanging intermediate representations, while safeguarding accuracy, especially when on-device models are compressed or otherwise approximated to meet energy and memory constraints.

A straightforward collaborative offloading strategy is layer-wise partitioning: a prefix of layers is executed on the device, and the resulting intermediate representations are transmitted to the cloud to complete the remaining computation. This design can trade cloud-side throughput against end-to-end latency and, importantly, requires no weight fine-tuning or retraining, making it flexible across models. Λ-Split[80] instantiates this idea by dividing a large model into three segments, with two residing on the device and one in the cloud. During inference, a local segment A produces intermediate states that the cloud segment processes for the bulk of computation; the results then return to a local segment B to generate the final output. Because neither raw inputs nor final outputs appear in the cloud, privacy is preserved. To further reduce transfer time, KV caching and layer-dependent activation memory (LDM) caching are leveraged to shrink the volume of intermediate data exchanged. However, the split points are chosen manually based on heuristics. To automate partitioning under system constraints, EdgeShard[81] formulates partition-and-placement as a constrained optimization problem over the resources. Using a dynamic programming approach, it yields principled split policies that either maximize cloud-side throughput or minimize per-request latency.

The same idea naturally extends to personalized fine-tuning settings. Under parameter-efficient fine-tuning (PEFT), most backbone weights are frozen and only small adapter modules are retrained, meaning personalized models largely share the same backbone. Leveraging this property, Edge-LLM[82] places PEFT adapters on the device while hosting the frozen backbone in the cloud. During generation, the device performs preprocessing (word embeddings and positional encodings), sends the embedded sequence to

the cloud for backbone inference, and then receives intermediate features to finish the adapter inference locally (the Infer Adapter stage). It also supports continual fine-tuning, allowing different edge sites to update adapters independently with local data. PrivateLoRA[83], instead of keeping the entire low-rank adaptation (LoRA) module on the device, adopts a three-segment LoRA decomposition that reduces inter-tier transmission overhead and significantly shrinks the on-device LoRA footprint.

Complementary to layer-wise partitioning, a second offloading paradigm is small-large model cooperation. A compact edge model handles requests by default, and the cloud-hosted large model is consulted only when necessary. Ce-collm[84] exemplifies this idea with early-exit models. The layers before the exit point are placed locally, and the remaining layers run in the cloud. If the local confidence is high, the device returns the result; otherwise, it uploads shallow-layer activations (rather than the exit-point state) to enable asynchronous transmission while local computation continues. The trade-off is flexibility, and inserting reliable exit points typically requires model-specific fine-tuning or retraining. PerLLM[85] deploys a lightweight model at the edge and a large model in the cloud. It formulates the request assignment as a combinatorial multi-armed bandit problem, optimizing for latency, compute, and bandwidth. This problem is solved by a constraint-satisfying upper confidence bound policy, which distributes deadline-aware requests across servers. Finally, an edge-first strategy[86] keeps full-precision inference at the edge and uses a faster (potentially slightly less accurate) cloud inference path only under edge overload. The strategy mitigates long-haul transmission delays. The objective is to minimize average latency while controlling tail latency, yielding an efficient task allocation.

As summarized in Table 4, offloading strategies for LLM inference can be broadly divided into single-device and multi-device approaches, each balancing compute–memory tradeoffs through different coordination and scheduling mechanisms.

Single-device methods (e.g., FastDecode, Neo, PageAttention, HAQ-edge, MOEpic, DeepSpeed, and FlexGen) focus on intra-device heterogeneity between GPU, CPU, DRAM, and NVMe. FastDecode and Neo adopt heterogeneous pipelines: both partition tasks, so GPUs handle compute-intensive kernels while CPUs process memory-bound components, but FastDecode further integrates performance modeling and hybrid-precision quantization, while Neo emphasizes asymmetric scheduling and load-aware dynamic balancing[87]. PageAttention and HAQ-edge address memory optimization: PageAttention implements paged KV storage to reduce fragmentation and redundancy, whereas HAQ-edge combines Hessian-based mixed quantization with CPU–GPU cooperative inference for low-bit precision efficiency. MOEpic tackles MoE challenges through expert vertical

Table 4 Overview of collaboration strategies for LLM serving.

Method		Device	Collaboration mode	Optimization goal
Single-device	FastDecode[17]	GPU↔CPU	Heterogeneous decoding pipeline	Throughput, GPU utilization, transfer overhead
	NEO[18]	GPU↔CPU	KV-cache offloading	Memory, batch size, throughput, latency
	PagedAttention (vLLM) [71]	Single GPU	In-GPU KV paging management	Fragmentation, redundancy, throughput
	HAQ-Edge[74]	GPU↔CPU	Expert offloading (activation path)	Memory, latency
	MoEpic[75]	GPU↔CPU memory	Expert offloading (activation path)	Latency, cache hit rate
	DeepSpeed[76]	NVMe↔DRAM↔GPU	Layer-wise weight streaming	Throughput, model size
	FlexGen[77]	GPU↔CPU↔NVMe	KV-cache & weight scheduling	Throughput, batch size
Multi-device	PipeEdge[78]	Edge↔Edge	Layer-wise pipeline parallelism	Latency, model size
	Match-LLM[27]	Edge↔Edge	Pipeline parallelism	Latency
	Galaxy[79]	Multi-GPU (edge)	Tensor parallelism	Latency
	Λ-Split[80]	Device↔Cloud	Three-way partition	Transfer latency
	EdgeShard[81]	Device↔Cloud / Edge↔Edge	Segmented model placement	Latency, throughput
	Edge-LLM[82]	Device↔Cloud	Local fine-tuning, cloud freezing	Latency
	PrivateLoRA[83]	Device↔Cloud	Three-stage LoRA decomposition	Communication, latency
	CE-CoLLM[84]	Device↔Cloud	Early exit	Latency, cloud throughput
	PerLLM[85]	Edge↔Cloud	Small-large model collaboration	Throughput, energy
	Edge-First[86]	Edge↔Cloud	Small-large model collaboration	Latency, edge load, energy

splitting and hierarchical memory allocation, while DeepSpeed and FlexGen extend to tiered memory hierarchies (GPU–CPU–NVMe). DeepSpeed applies ZeRO-Inference for layered offloading with parallelism optimizations, whereas FlexGen formulates linear programming-based placement and scheduling with quantized weight and cache streaming to maximize throughput under tight resources.

Multi-device methods (e.g., PipeEdge, Matching Game, Galaxy, Λ-Split, EdgeShard, PrivateLoRA, CE-CoLLM, and Edge-first) emphasize inter-device collaboration and scheduling. PipeEdge and Matching Game optimize pipeline parallelism, with PipeEdge performing layer-wise partitioning and dynamic programming scheduling, and Matching Game modeling computation-communication tradeoffs as a stable matching problem. Galaxy introduces hybrid model parallelism (Tensor+Sequence) with heterogeneity-aware workload division, while Λ-Split and EdgeShard extend to edge-cloud collaboration, using layer partitioning and adaptive device selection for privacy-preserving or resource-aware execution. PrivateLoRA focuses on personalized low-rank adaptation with compressed activation transfer, reducing communication cost by over 95 CE-CoLLM implements latency-aware early exiting and adaptive context management to dynamically switch between cloud and edge modes, while Edge-first introduces active inference-based scheduling, leveraging free-energy minimization to optimize task offloading, resource allocation, and communication latency.

5 Open Challenge and Future Direction

Despite significant advancements in speculative decoding and model offloading, the tailored optimization for edge LLMs still faces several open challenges. Key challenges and promising future directions include:

Multi-device collaboration: Current multi-device speculative decoding relies on basic token-level draft-verify patterns, failing to leverage intermediate features like attention states and hidden representations. This results in suboptimal utilization of distributed resources and limited parallelism. Key research directions should explore feature-level speculation with middle-layer features and hierarchical verification schemes for heterogeneous devices. Moving beyond token-passing to feature-aware collaboration is crucial for unlocking the full potential of multi-device edge inference.

Decoding strategy: The sequential nature of LLM's basic autoregressive decoding method limits inference speed. While speculative decoding helps, its effectiveness is constrained by limited edge resources. Small draft models yield poor acceptance rates, while larger ones consume more memory and compute resources. Future efforts may explore more decoding methods. Multi-token decoding[36, 42] generates several tokens per forward pass to reduce computational costs. Non-autoregressive[88] decoding predicts multiple tokens simultaneously in one step, offering massive latency reductions and is even faster than multi-token decoding. The primary challenge for these methods is maintaining high-quality output.

Memory scheduling: With the rapid evolution of large language models and their increasingly complex architectures, edge inference demands more refined and comprehensive memory scheduling strategies to maximize the utilization of limited hardware resources, thereby enabling the deployment of larger-scale models. Meanwhile, the flexible selective eviction of KV cache introduces new design spaces and research opportunities for memory scheduling.

Energy overhead: Offloading model parameters or KV cache to external storage, such as solid state drives (SSDs), enables edge devices to host larger-scale models. However, the per-access energy cost of external storage is significantly higher than that of memory access[89]. Speculative decoding requires the execution of both a draft and a target model, which may lead to increased peak power consumption and inefficient resource utilization if not carefully managed. Existing research has primarily focused on designing strategies to accelerate inference, while systematic exploration of how to reduce energy consumption without sacrificing inference efficiency remains limited.

Pre-deployment optimizations: Pre-deployment techniques, such as quantization, pruning, and distillation, can interact with speculative decoding and offloading methods. But methods like quantization may cause distribution shifts that harm consistency. Joint optimization with speculative decoding could enable lighter yet accurate inference. Understanding this interplay could guide the holistic design of efficient edge LLMs.

6 Conclusion

This survey reviews efficient inference techniques for

LLMs deployed on resource-constrained edge devices, focusing on two key strategies: speculative decoding and model offloading. These strategies address challenges in computational power, memory, and bandwidth by optimizing token generation, model partitioning, and model coordination. Despite significant progress, challenges remain in efficiency around inference speed and energy consumption. Future advances in these areas will enhance LLM capabilities on edge devices.

Acknowledgment

This work was supported by the National Key R&D Program of China (No. 2022YFC3801300), the National Natural Science Foundation of China (Nos. U22A2031, 62172250, and 62402028), and the Tsinghua University-Fuzhou Joint Institute for Data Technology.

References

[1] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al., LLaMA: Open and efficient foundation language models, arXiv preprint arXiv: 2302.13971, 2023.

[2] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al., LLAMA 2: Open foundation and fine-tuned chat models, arXiv preprint arXiv: 2307.09288, 2023.

[3] A. Grattafiori, A. Dubey, A. Jauhri, A. Pandey, A. Kadian, A. Al-Dahle, A. Letman, A. Mathur, A. Schelten, A. Vaughan, et al., The LlaMA 3 herd of models, arXiv preprint arXiv: 2407.21783, 2024.

[4] DeepSeek-AI, DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning, arXiv preprint arXiv: 2501.12948, 2025.

[5] DeepSeek-AI, DeepSeek-V3 technical report, arXiv preprint arXiv: 2412.19437, 2024.

[6] Gemma Team, Gemma: Open models based on Gemini research and technology, arXiv preprint arXiv: 2403.08295, 2024.

[7] Gemma Team, Gemma 2: Improving open language models at a practical size, arXiv preprint arXiv: 2408.00118, 2024.

[8] Gemma Team, Gemma 3 technical report, arXiv preprint arXiv: 2503.19786, 2025.

[9] OpenAI, ChatGPT, https://openai.com/index/chatgpt/, 2022.

[10] Microsoft, Microsoft copilot, https://copilot.microsoft.com/, 2025.

[11] Google, Google Gemini, https://gemini.google.com/, 2025.

[12] H. Wen, Y. Li, G. Liu, S. Zhao, T. Yu, T. J. J. Li, S. Jiang, Y. Liu, Y. Zhang, and Y. Liu, AutoDroid: LLM-powered task automation in android, in *Proc. 30th Annu. Int. Conf. Mobile Computing and Networking*, Washington, DC, USA, 2024, pp. 543–557.

[13] X. Li, Z. Lu, D. Cai, X. Ma, and M. Xu, Large language models on mobile devices: Measurements, analysis, and insights, in *Proc. Workshop on Edge and Mobile Foundation Models*, Tokyo, Japan, 2024, pp. 1–6.

[14] Y. Zheng, Y. Chen, B. Qian, X. Shi, Y. Shu, and J. Chen, A review on edge large language models: Design, execution, and applications, *ACM Comput. Surveys*, vol. 57, no. 8, p. 209, 2025.

[15] Y. Leviathan, M. Kalman, and Y. Matias, Fast inference from transformers via speculative decoding, in *Proc. 40th Int. Conf. Machine Learning*, Honolulu, HI, USA, 2023, pp. 19274–19286.

[16] X. Miao, G. Oliaro, Z. Zhang, X. Cheng, Z. Wang, Z. Zhang, R. Y. Y. Wong, A. Zhu, L. Yang, X. Shi, et al., SpecInfer: Accelerating large language model serving with tree-based speculative inference and verification, in *Proc. 29th ACM Int. Conf. Architectural Support for Programming Languages and Operating Systems*, La Jolla, CA, USA, 2024, pp. 932–949.

[17] J. He and J. Zhai, FastDecode: High-throughput GPU-efficient LLM serving using heterogeneous pipelines, arXiv preprint arXiv: 2403.11421, 2024.

[18] X. Jiang, Y. Zhou, S. Cao, I. Stoica, and M. Yu, NEO: Saving GPU memory crisis with CPU offloading for online LLM inference, arXiv preprint arXiv: 2411.01142, 2024.

[19] Z. Zhou, X. Ning, K. Hong, T. Fu, J. Xu, S. Li, Y. Lou, L. Wang, Z. Yuan, X. Li, et al., A survey on efficient inference for large language models, arXiv preprint arXiv: 2404.14294, 2024.

[20] Y. Hu, Z. Liu, Z. Dong, T. Peng, B. McDanel, and S. Q. Zhang, Speculative decoding and beyond: An in-depth survey of techniques, arXiv preprint arXiv: 2502.19732, 2025.

[21] H. Xia, Z. Yang, Q. Dong, P. Wang, Y. Li, T. Ge, T. Liu, W. Li, and Z. Sui, Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding, in *Proc. Findings of the Association for Computational Linguistics*: *ACL*, Bangkok, Thailand, 2024, pp. 7655–7671.

[22] B. Xiao, X. Yu, W. Ni, X. Wang, and H. V. Poor, Over-the-air federated learning: Status quo, open challenges, and future directions, *Fundam. Res.*, vol. 5, no. 4, pp. 1710–1724, 2025.

[23] M. Wang, L. Zhou, X. Huang, and W. Zheng, Towards federated learning driving technology for privacy-preserving micro-expression recognition, *Tsinghua Science and Technology*, vol. 30, no. 5, pp. 2169–2183, 2025.

[24] Z. Yuan, Y. Shang, Y. Zhou, Z. Dong, Z. Zhou, C. Xue, B. Wu, Z. Li, Q. Gu, Y. J. Lee, et al., LLM inference unveiled: Survey and roofline model insights, arXiv preprint arXiv: 2402.16363, 2024.

[25] H. Li, Y. Li, A. Tian, T. Tang, Z. Xu, X. Chen, N. Hu, W. Dong, Q. Li, and L. Chen, A survey on large language model acceleration based on KV cache management, arXiv preprint arXiv: 2412.19442, 2024.

[26] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones,

A. N. Gomez, Ł. Kaiser, and I. Polosukhin, Attention is all you need, in *Proc. 31st Int. Conf. Neural Information Processing Systems*, Long Beach, CA, USA, 2017, pp. 6000–6010.

[27] B. Picano, D. T. Hoang, and D. N. Nguyen, A matching game for LLM layer deployment in heterogeneous edge networks, *IEEE Open J. Commun. Soc.*, vol. 6, pp. 3795–3805, 2025.

[28] H. Xia, T. Ge, P. Wang, S. Q. Chen, F. Wei, and Z. Sui, Speculative decoding: Exploiting speculative execution for accelerating Seq2seq generation, in *Proc. Findings of the Association for Computational Linguistics*: *EMNLP*, Singapore, 2023, pp. 3909–3925.

[29] K. Huang, X. Guo, and M. Wang, SpecDec++: Boosting speculative decoding via adaptive candidate lengths, arXiv preprint arXiv: 2405.19715, 2024.

[30] C. Chen, S. Borgeaud, G. Irving, J. B. Lespiau, L. Sifre, and J. Jumper, Accelerating large language model decoding with speculative sampling, arXiv preprint arXiv: 2302.01318, 2023.

[31] Z. Sun, A. T. Suresh, J. H. Ro, A. Beirami, H. Jain, and F. Yu, SpecTr: Fast speculative decoding via optimal transport, in *Proc. 37th Int. Conf. Neural Information Processing Systems*, New Orleans, LA, USA, 2023, p. 1314.

[32] Y. Zhou, K. Lyu, A. S. Rawat, A. K. Menon, A. Rostamizadeh, S. Kumar, J. F. Kagy, and R. Agarwal, DistillSpec: Improving speculative decoding via knowledge distillation, in *Proc. 12th Int. Conf. Learning Representations*, Vienna, Austria, https://dblp.org/rec/conf/iclr/ZhouLRMRKKA24.html?view=bibtex, 2024.

[33] M. Lasby, N. Sinnadurai, V. Manohararajah, S. Lie, Y. Ioannou, and V. Thangarasa, SD^2: Self-distilled sparse drafters, arXiv preprint arXiv: 2504.08838, 2025.

[34] W. Xu, R. Han, Z. Wang, L. T. Le, D. Madeka, L. Li, W. Y. Wang, R. Agarwal, C. Y. Lee, and T. Pfister, Speculative knowledge distillation: Bridging the teacher-student gap through interleaved sampling, in *Proc. 13th Int. Conf. Learning Representations*, Singapore, Singapore, https://dblp.org/rec/conf/iclr/XuH0LM0WALP25.html?view=bibtex, 2025.

[35] L. Gui, B. Xiao, L. Su, and W. Chen, Boosting lossless speculative decoding via feature sampling and partial alignment distillation, arXiv preprint arXiv: 2408.15562, 2024.

[36] M. Stern, N. Shazeer, and J. Uszkoreit, Blockwise parallel decoding for deep autoregressive models, in *Proc. 32nd Int. Conf. Neural Information Processing Systems*, Montréal, Canada, 2018, pp. 10107–10116.

[37] A. Santilli, S. Severino, E. Postolache, V. Maiorca, M. Mancusi, R. Marin, and E. Rodola, Accelerating transformer inference for translation via parallel decoding, in *Proc. 61st Annu. Meeting of the Association for Computational Linguistics*, Toronto, Canada, 2023, pp. 12336–12355.

[38] C. Hooper, S. Kim, H. Mohammadzadeh, H. Genc, K. Keutzer, A. Gholami, and Y. Sophia Shao, SPEED: Speculative pipelined execution for efficient decoding, in *Enhancing LLM Performance*: *Efficacy, Fine-Tuning, and Inference Techniques*, P. Passban, A. Way, and M. Rezagholizadeh, eds. Cham, Switzerland: Springer, 2025, pp. 19–32.

[39] T. Cai, Y. Li, Z. Geng, H. Peng, J. D. Lee, D. Chen, and T. Dao, Medusa: Simple LLM inference acceleration framework with multiple decoding heads, in *Proc. 41st Int. Conf. Machine Learning*, Vienna, Austria, https://dblp.org/rec/conf/icml/CaiLGPLCD24.html?view=bibtex, 2024.

[40] T. Ge, H. Xia, X. Sun, S. Q. Chen, and F. Wei, Lossless acceleration for Seq2seq generation with aggressive decoding, arXiv preprint arXiv: 2205.10350, 2022.

[41] C. Du, J. Jiang, Y. Xu, J. Wu, S. Yu, Y. Li, S. Li, K. Xu, L. Nie, Z. Tu, et al., GliDe with a CaPE: A low-hassle method to accelerate speculative decoding, in *Proc. 41st Int. Conf. Machine Learning*, Vienna, Austria, https://dblp.org/rec/conf/icml/Du0XWY0LXNTY24.html?view=bibtex, 2024.

[42] F. Gloeckle, B. Y. Idrissi, B. Rozière, D. Lopez-Paz, and G. Synnaeve, Better & faster large language models via multi-token prediction, in *Proc. 41st Int. Conf. Machine Learning*, Vienna, Austria, https://dblp.org/rec/conf/icml/GloeckleIRLS24.html?view=bibtex, 2024.

[43] S. Yang, G. Lee, J. Cho, D. Papailiopoulos, and K. Lee, Predictive pipelined decoding: A compute-latency trade-off for exact LLM decoding, arXiv preprint arXiv: 2307.05908, 2024.

[44] J. Zhang, J. Wang, H. Li, L. Shou, K. Chen, G. Chen, and S. Mehrotra, Draft & verify: Lossless large language model acceleration via self-speculative decoding, in *Proc. 62nd Annu. Meeting of the Association for Computational Linguistics*, Bangkok, Thailand, 2024, pp. 11263–11282.

[45] J. Ou, Y. Chen, and W. Tian, Lossless acceleration of large language model via adaptive N-gram parallel decoding, in *Proc. 2024 Conf. North American Chapter of the Association for Computational Linguistics*: *Human Language Technologies* (*Volume 1*: *Long Papers*), Mexico City, Mexico, 2024, pp. 10–22.

[46] Y. Fu, P. Bailis, I. Stoica, and H. Zhang, Break the sequential dependency of LLM inference using lookahead decoding, in *Proc. 41st Int. Conf. Machine Learning*, Vienna, Austria, 2024, pp. 14060–14079.

[47] L. Stewart, M. Trager, S. K. Gonugondla, and S. Soatto, The N-Grammys: Accelerating autoregressive inference with learning-free batched speculation, in *Proc. 4th NeurIPS Efficient Natural Language and Speech Processing Workshop*, Vancouver, Canada, 2024, pp. 322–335.

[48] N. Yang, T. Ge, L. Wang, B. Jiao, D. Jiang, L. Yang, R. Majumder, and F. Wei, Inference with reference: Lossless acceleration of large language models, arXiv preprint arXiv: 2304.04487, 2023.

[49] Z. He, Z. Zhong, T. Cai, J. D. Lee, and D. He, REST: Retrieval-based speculative decoding, in *Proc. 2024 Conf. North American Chapter of the Association for Computational Linguistics*: *Human Language Technologies* (*Volume 1*: *Long Papers*), Mexico City, Mexico, 2024, pp. 1582–1595.

[50] Z. Wang, Z. Wang, L. T. Le, H. S. Zheng, S. Mishra, V. Perot, Y. Zhang, A. Mattapalli, A. Taly, J. Shang, et al., Speculative RAG: Enhancing retrieval augmented generation through drafting, in *Proc. 13th Int. Conf. Learning Representations*, Singapore, https://dblp.org/rec/conf/iclr/00020LZMP0MTSLP25.html?view=bibtex, 2025.

[51] G. Chen, Q. Feng, J. Ni, X. Li, and M. Q. Shieh, Long-context inference with retrieval-augmented speculative decoding, arXiv preprint arXiv: 2502.20330, 2025.

[52] Z. Zhang, A. Zhu, L. Yang, Y. Xu, L. Li, P. M. Phothilimthana, and Z. Jia, Accelerating iterative retrieval-augmented language model serving with speculation, in *Proc. 41st Int. Conf. Machine Learning*, Vienna, Austria, https://dblp.org/rec/conf/icml/ZhangZYXLPJ24.html?view=bibtex, 2024.

[53] Z. Gong, J. Liu, Z. Wang, P. Wu, J. Wang, X. Cai, D. Zhao, and R. Yan, Graph-structured speculative decoding, in *Proc. Findings of the Association for Computational Linguistics*: *ACL*, Bangkok, Thailand, 2024, pp. 11404–11415.

[54] X. Zhuge, X. Shen, Z. Wang, F. Dang, X. Ding, D. Li, Y. Han, T. Hao, and Z. Yang, SpecOffload: Unlocking latent GPU capacity for LLM inference on resource-constrained devices, arXiv preprint arXiv: 2505.10259, 2025.

[55] S. Yildirim and D. Chen, SpecMemo: Speculative decoding is in your pocket, arXiv preprint arXiv: 2506.01986, 2025.

[56] W. Zhong and M. Bharadwaj, S3D: A simple and cost-effective self-speculative decoding scheme for low-memory GPUs, arXiv preprint arXiv: 2405.20314, 2024.

[57] H. M. Chen, W. Luk, K. F. C. Yiu, R. Li, K. Mishchenko, S. I. Venieris, and H. Fan, Hardware-aware parallel prompt decoding for memory-efficient acceleration of LLM inference, arXiv preprint arXiv: 2405.18628, 2024.

[58] R. Svirschevski, A. May, Z. Chen, B. Chen, Z. Jia, and M. Ryabinin, SpecExec: Massively parallel speculative decoding for interactive LLM inference on consumer devices, in *Proc. 38th Int. Conf. Neural Information Processing Systems*, Vancouver, Canada, 2024, p. 522.

[59] J. Wei, Y. Huang, Y. Zhou, J. Jiang, J. Du, and Y. Lu, Ghidorah: Fast LLM inference on edge with speculative decoding and hetero-core parallelism, arXiv preprint arXiv: 2505.23219, 2025.

[60] D. Xu, W. Yin, H. Zhang, X. Jin, Y. Zhang, S. Wei, M. Xu, and X. Liu, EdgeLLM: Fast on-device LLM inference with speculative decoding, *IEEE Trans. Mob. Comput.*, vol. 24, no. 4, pp. 3256–3273, 2025.

[61] N. Timor, J. Mamou, D. Korat, M. Berchansky, O. Pereg, M. Wasserblat, T. Galanti, M. Gordon-Kiwkowitz, and D. Harel, Distributed speculative inference (DSI): Speculation parallelism for provably faster lossless language model inference, in *Proc. 13th Int. Conf. Learning Representations*, Singapore, https://dblp.org/rec/conf/iclr/TimorMKBPWGGH25.html?view=bibtex, 2025.

[62] B. McDanel, AMUSD: Asynchronous multi-device speculative decoding for LLM acceleration, in *Proc. 2025 IEEE Int. Symp. Circuits and Systems*, London, UK, 2025, pp. 1–5.

[63] L. Zhang, Z. Zhang, B. Xu, S. Mei, and D. Li, Dovetail: A CPU/GPU heterogeneous speculative decoding for LLM inference, arXiv preprint arXiv: 2412.18934, 2024.

[64] X. Li, D. Spatharakis, S. Ghafouri, J. Fan, and D. Nikolopoulos, SLED: A speculative LLM decoding framework for efficient edge serving, arXiv preprint arXiv: 2506.09397, 2025.

[65] J. Park, S. Cho, and D. Han, SpecEdge: Scalable edge-assisted serving framework for interactive LLMs, arXiv preprint arXiv: 2505.17052, 2025.

[66] Y. Venkatesha, S. Kundu, and P. Panda, Fast and cost-effective speculative edge-cloud decoding with early exits, arXiv preprint arXiv: 2505.21594, 2025.

[67] G. Gerganov, D. Devesa, X. S. Nguyen, J. Gäßler, S. Skjæret, J. Bolz, D. Bevenius, J. Van Bortel, Kawrakow, compilade, et al., llama.cpp: Inference of LLaMA model in pure C/C++, https://github.com/ggerganov/llama.cpp, 2025.

[68] X. Jiang, H. Wang, Y. Chen, Z. Wu, L. Wang, B. Zou, Y. Yang, Z. Cui, Y. Cai, T. Yu, et al., MNN: A universal and efficient inference engine, in *Proc. 3rd Machine Learning and Systems*, Austin, TX, USA, 2020, pp. 1–13.

[69] MLC Team, MLC LLM: Universal LLM deployment engine, https://github.com/mlc-ai/mlc-llm, 2025.

[70] NVIDIA, TensorRT-LLM: A TensorRT toolbox for optimized large language model inference, https://github.com/NVIDIA/TensorRT-LLM, 2024.

[71] W. Kwon, Z. Li, S. Zhuang, Y. Sheng, L. Zheng, C. H. Yu, J. Gonzalez, H. Zhang, and I. Stoica, Efficient memory management for large language model serving with PagedAttention, in *Proc. 29th Symp. Operating Systems Principles*, Koblenz, Germany, 2023, pp. 611–626.

[72] Y. Li, F. Wei, C. Zhang, and H. Zhang, EAGLE: Speculative sampling requires rethinking feature uncertainty, in *Proc. 41st Int. Conf. Machine Learning*, Vienna, Austria, https://dblp.org/rec/conf/icml/LiW0024.html?view=bibtex, 2024.

[73] N. Sun, M. Backes, and Y. Zhang, MTP: Multi-token prediction for enhanced decoding efficiency, arXiv preprint arXiv: 2404.19841, 2024.

[74] T. Zhang, N. Li, X. Yuan, W. Xu, Q. Chen, S. Guo, and H. Zhang, Efficient edge LLMs deployment via HessianAware quantization and CPU GPU collaborative, arXiv preprint arXiv: 2508.07329, 2025.

[75] J. Yan, J. Liu, H. Xu, and L. Huang, Accelerating mixture-of-expert inference with adaptive expert split mechanism, arXiv preprint arXiv: 2509.08342, 2025.

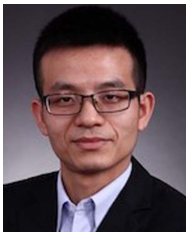
[76] R. Y. Aminabadi, S. Rajbhandari, A. A. Awan, C. Li, D. Li, E. Zheng, O. Ruwase, S. Smith, M. Zhang, J. Rasley, et al., DeepSpeed-inference: Enabling efficient inference of transformer models at unprecedented scale, in *Proc. Int. Conf. High Performance Computing, Networking, Storage and Analysis*, Dallas, TX, USA, 2022, pp. 1–15.

[77] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, FlexGen: High-throughput generative inference of large language models with a single GPU, in *Proc. 40th Int. Conf.*

Machine Learning, Honolulu, HI, USA, 2023, pp. 31094–31116.
[78] Y. Hu, C. Imes, X. Zhao, S. Kundu, P. A. Beerel, S. P. Crago, and J. P. Walters, PipeEdge: Pipeline parallelism for large-scale model inference on heterogeneous edge devices, in *Proc. 25th Euromicro Conf. Digital System Design*, Maspalomas, Spain, 2022, pp. 298–307.
[79] S. Ye, J. Du, L. Zeng, W. Ou, X. Chu, Y. Lu, and X. Chen, Galaxy: A resource-efficient collaborative edge AI system for in-situ transformer inference, in *Proc. IEEE Conf. Computer Communications*, Vancouver, Canada, 2024, pp. 1001–1010.
[80] S. Ohta and T. Nishio, A-Split: A privacy-preserving split computing framework for cloud-powered generative AI, arXiv preprint arXiv: 2310.14651, 2023.
[81] M. Zhang, X. Shen, J. Cao, Z. Cui, and S. Jiang, EdgeShard: Efficient LLM inference via collaborative edge computing, *IEEE Internet Things J.*, vol. 12, no. 10, pp. 13119–13131, 2025.
[82] F. Cai, D. Yuan, Z. Yang, and L. Cui, Edge-LLM: A collaborative framework for large language model serving in edge computing, in *Proc. 2024 IEEE Int. Conf. Web Services*, Shenzhen, China, 2024, pp. 799–809.
[83] Y. Wang, Y. Lin, X. Zeng, and G. Zhang, PrivateLoRA for efficient privacy preserving LLM, arXiv preprint arXiv: 2311.14030, 2023.
[84] H. Jin and Y. Wu, CE-CoLLM: Efficient and adaptive large language models through cloud-edge collaboration, in *Proc. 2025 IEEE Int. Conf. Web Services* (*ICWS*), Helsinki, Finland, 2025, pp. 316–323.
[85] Z. Yang, Y. Yang, C. Zhao, Q. Guo, W. He, and W. Ji, PerLLM: Personalized inference scheduling with edge-cloud collaboration for diverse LLM services, arXiv preprint arXiv: 2405.14636, 2024.
[86] Y. He, J. Fang, F. R. Yu, and V. C. Leung, Large language models (LLMs) inference offloading and resource allocation in cloud-edge computing: An active inference approach, *IEEE Trans. Mob. Comput.*, vol. 23, no. 12, pp. 11253–11264, 2024.
[87] T. Fu, Z. Yang, Z. Ye, C. Ma, Y. Han, Y. Luo, X. Wang, and Z. Wang, A survey on the scheduling of DL and LLM training jobs in GPU clusters, *Chin. J. Electron.*, vol. 34, no. 3, pp. 881–905, 2025.
[88] Y. Xiao, L. Wu, J. Guo, J. Li, M. Zhang, T. Qin, and T. Y. Liu, A survey on non-autoregressive generation for neural machine translation and beyond, *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 45, no. 10, pp. 11407–11427, 2023.
[89] K. Kyung, S. Yun, and J. H. Ahn, SSD offloading for LLM mixture-of-experts weights considered harmful in energy efficiency, *IEEE Comput. Archit. Lett.*, vol. 24, no. 2, pp. 265–268, 2025.

Guanyu Cai received the BEng degree from Central South University, China in 2022, and the MEng degree from Tsinghua University, China in 2025. He is currently a PhD candidate at School of Software, Tsinghua University, China. His research interests include Internet of Things (IoTs) and mobile computing.

Ruiming Tian is currently an undergraduate student in software engineering at Beijing Jiaotong University, China. His research interests focus on large language models at the edge and optimization techniques for accelerating deep learning inference.



Jiliang Wang received the BEng degree in computer science and technology from University of Science and Technology of China in 2022 and the PhD degree in computer science and engineering from Hong Kong University of Science and Technology, China in 2011. He is currently an associate professor at School of Software, Tsinghua University, China. His research interests include IoTs and mobile computing.

Lang Yang is an undergraduate student at School of Software, Tsinghua University, China. His research interests include efficient large language model inference and edge AI.

Yunzhe Jia is currently an undergraduate student at School of Software, Tsinghua University, Beijing, China. His research interests include efficient inference of large language models on edge devices.

Lingkun Li received the PhD degree in computer science from Michigan State University, USA in 2022. He is currently a lecturer at Beijing Jiaotong University, China. His research interests include mobile computing, operating systems, and IoTs.