

Jupiter: Fast and Resource-Efficient Collaborative Inference of Generative LLMs on Edge Devices

Shengyuan Ye[◆], Bei Ouyang[◆], Liekang Zeng[◇], Tianyi Qian[◆], Xiaowen Chu[▲], Jian Tang[▲], Xu Chen^{◆*}

[◆]School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China

[◇]The Chinese University of Hong Kong, Hong Kong SAR, China

[▲]Data Science and Analytics Thrust, HKUST (Guangzhou), Guangzhou, China

[▲]Midea Group, China

{yeshy8, ouyb9, qianty}@mail2.sysu.edu.cn, zenglk3@gmail.com

xwchu@ust.hk, tangjian22@midea.com, chenxu35@mail.sysu.edu.cn

Abstract—Generative large language models (LLMs) have garnered significant attention due to their exceptional capabilities in various AI tasks. Traditionally deployed in cloud datacenters, LLMs are now increasingly moving towards more accessible edge platforms to protect sensitive user data and ensure privacy preservation. The limited computational resources of individual edge devices, however, can result in excessively prolonged inference latency and overwhelmed memory usage. While existing research has explored collaborative edge computing to break the resource wall of individual devices, these solutions yet suffer from massive communication overhead and under-utilization of edge resources. Furthermore, they focus exclusively on optimizing the prefill phase, neglecting the crucial autoregressive decoding phase for generative LLMs. To address that, we propose *Jupiter*, a fast, scalable, and resource-efficient collaborative edge AI system for generative LLM inference. *Jupiter* introduces a flexible pipelined architecture as a principle and differentiates its system design according to the differentiated characteristics of the prefill and decoding phases. For prefill phase, *Jupiter* submits a novel intra-sequence pipeline parallelism and develops a meticulous parallelism planning strategy to maximize resource efficiency; For decoding, *Jupiter* devises an effective outline-based pipeline parallel decoding mechanism combined with speculative decoding, which further magnifies inference acceleration. Extensive evaluation based on realistic implementation demonstrates that *Jupiter* remarkably outperforms state-of-the-art approaches under various edge environment setups, achieving up to $26.1\times$ end-to-end latency reduction while rendering on-par generation quality.

I. INTRODUCTION

The emergence of generative large language models (LLMs) has attracted widespread attention from both industry and academia owing to their exceptional capabilities in a wide range of artificial intelligence (AI) tasks. These models, widely deployed in cloud datacenters equipped with powerful server-grade GPUs, have driven increasing intelligent edge applications such as ChatBot [1] and smart-home AI agent [2]. While born on datacenter warehouse, there is an emerging trend of serving LLMs on more accessible edge platforms rather than uploading requests to remote clouds owned by commercial companies, owing to the sensitive and privacy-critical nature of user data. A recent survey [3] on LLM-based edge applications

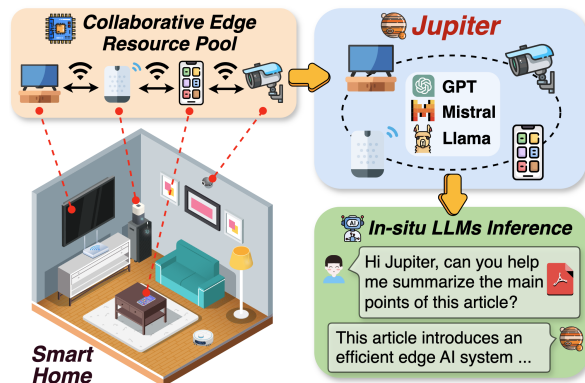


Fig. 1. Collaborative LLMs inference in smart home empowered by Jupiter.

has revealed that over 80% of industry experts believe personal LLMs should be fully or primarily hosted at the edge to ensure privacy-preserving model inference.

However, hosting computation-intensive and resource-hungry LLMs at the edge is significantly challenging due to the limited resources of individual edge devices, leading to prolonged inference latency and overwhelmed memory footprint. To address this, some pioneering research [4]–[7] alternatively observe that common edge environments, such as smart homes, usually comprise a variety of trusted idle devices in physical proximity (e.g., phones, laptops, and smart-home devices) that are often connected to the same local area network (LAN). This motivates us to consider them as a collaborative edge resource pool to facilitate in-situ expedited and resource-efficient LLM inference, as depicted in Fig. 1.

Motivated by above insight, researchers have proposed different ways for LLM inference with collaborative edge computing [8]–[10]. However, these approaches suffer from significant limitations. In contrast to datacenter deployment, in-situ LLM serving for edge applications mainly focuses on low latency in processing a single input sequence (e.g., for the purpose of context/intention-aware intelligent control or response in smart homes). Galaxy and its subsequent works [8], [9] borrow ideas from tensor parallelism [11] to achieve parallel acceleration of single-sequence inference across multiple edge devices. However, these methods neces-

*Corresponding authors.

sitate multiple tensor synchronizations at each decoder layer to ensure inference correctness, making communication latency a bottleneck under low-bandwidth edge environments. Other few works, such as EdgeShard [10], leverage pipelined architecture [12] to orchestrate collaborative edge devices. However, in single-sequence request scenarios, it ultimately degrades to sequential inference, failing to leverage the computational resources of multiple edge devices concurrently. Furthermore, all the aforementioned works have focused exclusively on optimizing the prefill phase, neglecting the decoding phase, which is also crucial for generative LLMs.

In this paper, we address the limitations of existing systems by introducing *Jupiter*, a fast, scalable, and resource-efficient collaborative edge AI system for generative LLM inference, guided by the following design goals: (1) Enable parallel acceleration of single-sequence inference during both prefill and decoding phases. (2) Reduce memory footprint per device by distributing LLM parameters across participating devices. (3) Minimize tensor exchanges to ensure robust inference performance in low-bandwidth edge environments.

To achieve aforementioned design goals, we eschew tensor parallelism but instead adopt a pipelined architecture as a principle to orchestrate collaborative edge devices. This architecture assembles the memory of multiple edge devices to support the target LLM while maintaining high communication efficiency, as devices exchange only a small set of activations with their neighbors. To enable resource-efficient pipeline parallel inference for single-sequence requests, we leverage the key property of generative LLMs and propose a novel intra-sequence pipeline parallelism method. To maximize resource utilization, we introduce a novel dynamic programming-based parallelism planning for optimal LLM and sequence partitioning, taking into account device heterogeneity, memory budget, and varying input lengths. To conquer the parallelization of the autoregressive decoding phase, we first incorporate the idea of speculative decoding into our collaborative inference system to enhance resource efficiency. Next, to further boost parallelism potential by leveraging multiple edge devices concurrently, we borrow the wisdom from human thinking and further introduce an outline-based pipeline parallel decoding method. Extensive evaluations on practical edge testbeds demonstrate that *Jupiter* achieves up to $26.1\times$ end-to-end latency reduction compared to baselines while maintaining significant scalability in bandwidth-limited environments.

In summary, this paper makes the following contributions.

- Through extensive measurement studies on existing edge collaborative LLM inference systems, we advocate a pipelined architecture as a principle to orchestrate edge devices for fast generative LLM inference.
- We address the challenge of pipelined parallel acceleration during the prefill phase by proposing an intra-sequence pipeline parallelism method, combined with meticulous parallelism planning to maximize resource efficiency.
- We achieve parallel acceleration of the autoregressive decoding phase by integrating speculative decoding into our collaborative inference system and introducing an outline-

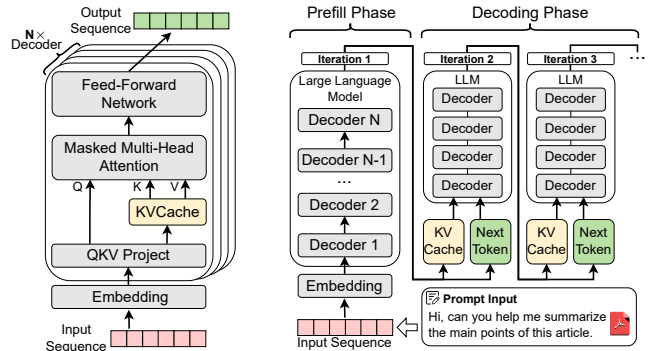


Fig. 2. Left: The architecture of a decoder-based LLM. Right: An instance of prefill and autoregressive decoding phases during generative LLMs inference.

based pipeline parallel decoding method that efficiently utilizes multiple edge devices concurrently.

- We implement *Jupiter* and evaluate it in realistic edge testbeds. Experimental results show up to $26.1\times$ latency reduction over the state-of-the-art methods, while maintaining significant scalability in bandwidth-limited environments.

II. MOTIVATION AND PRELIMINARIES

A. Decoder-Based Generative LLMs Architecture

Decoder-based LLMs, such as GPT-3 [13], LLaMa [14], and Mistral [15], have revolutionized natural language processing by enabling tasks ranging from simple text generation to complex problem-solving and conversational AI. In this paper, we focus on deploying these powerful generative models on collaborative edge devices.

As shown in Fig. 2(Left), a typical decoder-based LLM comprises input embeddings and multiple sequentially stacked decoder layers. Each decoder layer contains several key modules: (1) *QKV Project* takes the input tokens and transforms them into three distinct representations: queries (Q), keys (K), and values (V). (2) *Masked Multi-Head Attention (MHA)* performs self-attention for each head independently, concatenates their outputs, and processes them through a final linear layer. Masking ensures each position only attends to previous positions, maintaining the autoregressive property. (3) *Feed-Forward Network (FFN)* involves two linear operations that first expand the hidden size to a larger dimension and then compress it back to its original size. (4) *KVCache* serves as a dynamic repository where the keys and values of all the previous tokens are typically memoized, allowing models to access and reuse previously computed information expediently.

B. Generative LLMs Inference and KV Caching

Text generation with LLMs involves two main phases:

- 1) *Prefill Phase*: The initial delay after submitting a prompt input (i.e., time-to-first-token) is the processing time during the prefill phase, which occurs only once per input sequence. As illustrated in Fig. 2(Right), the LLM first takes the prompt sequence input and predicts a new token that serves as the initial token for the decoding phase. The intermediate states, including keys and values for each decoder layer, are stored in the *KVCache* for reuse in subsequent iterations.

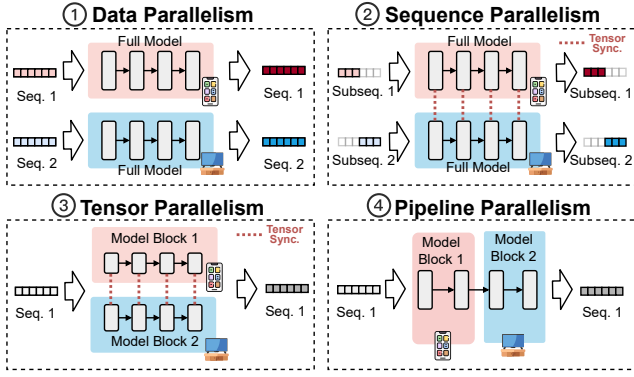


Fig. 3. Different parallel inference methods for LLMs.

Existing studies [16], [17] indicate that longer prompts often improve response quality and coherence, driving the ongoing effort to build LLMs that can accept increasingly longer inputs. However, long contexts pose a challenge to response-generation latency during the prefill phase, since the computational load for processing long contexts grows super-linearly with context length due to self-attention mechanism.

2) *Autoregressive Decoding Phase*: As shown in Fig. 2(Right), the newly generated token from prefill phase is then fed back into the decoding phase as input, creating an autoregressive process for token generation. To generate a new token that aligns with the context, LLMs must compute its relationship with all previous tokens. The *KVCache* stores the previously computed keys and values of these tokens, enabling their direct reuse without recomputation in each iteration. Decoding phase is repeated until a stop token is generated or the maximum sequence length is reached.

For tasks requiring the generation of numerous tokens, the decoding phase can dominate the inference process. The challenge with the decoding phase is its autoregressive nature, which makes it challenging to run in parallel and therefore hard to accelerate with multiple edge devices.

C. Collaborative Edge Computing for Generative LLMs

1) *Parallel Inference Methods for LLMs*: To fully harness the potential of collaborative edge devices for serving generative LLMs, the key question is the choice of parallelism method. We illustrate different parallelism strategies in Fig. 3. Specifically, ① *Data Parallelism (DP)* partitions workloads across the sample dimension, with each device maintaining a full replica of the model and independently performing inferences. However, for single-sequence requests, DP fails to leverage multiple edge devices concurrently. ② *Sequence Parallelism (SP)* requires each device to maintain a full model replica and partitions the input sequence into subsequences for parallel operation, but it necessitates two all-gather synchronizations per decoder layer to ensure consistent inference results. ③ *Tensor Parallelism (TP)* partitions LLM weights across devices, with each hosting a subset of parameters. However, it requires two all-reduce synchronizations per decoder, one after the MHA module and another after the FFN module. ④ *Pipeline Parallelism (PP)* horizontally partitions the LLM into consecutive stages along the layer dimension,

TABLE I
ANALYSIS ON COMPLEXITY OF VARIOUS PARALLELISM METHODS.

Features	① DP	② SP	③ TP	④ PP	Jupiter
Model Memory Usage (per device)	$\mathcal{O}(P)$	$\mathcal{O}(P)$	$\mathcal{O}(\frac{P}{N})$	$\mathcal{O}(\frac{P}{N})$	$\mathcal{O}(\frac{P}{N})$
Comp. Latency (per sequence)	$\mathcal{O}(C)$	$\mathcal{O}(\frac{C}{N})$	$\mathcal{O}(\frac{C}{N})$	$\mathcal{O}(C)$	$\mathcal{O}(\frac{C}{N})$
Comm. Volume (per sequence)	None	$2LSH$	$4LSH$	$(N-1)SH$	

TABLE II
COMM.-TO-COMP. RATIO OF VARIOUS PARALLELISM METHODS.

Model Name	Network Bandwidth	Communication-to-Computation Ratio				
		SP	TP	DT [9]	Galaxy [8]	Jupiter
Llama2-7B	100Mbps	8.16	6.96	3.48	5.19	0.08
	1Gbps	0.92	0.88	0.45	0.69	0.01
Llama2-13B	100Mbps	5.71	6.06	3.03	4.63	0.05
	1Gbps	0.73	0.81	0.38	0.56	0.01

with each stage mapped to a distinct edge device. Multiple input sequences are injected into the pipeline concurrently to increase parallelism. However, facilitating parallel inference for a single sequence remains challenging.

We analyze and summarize the model memory usage, computation latency, and total communication volume for various parallelism methods in single sequence inference, as detailed in Table I. P represents the total number of LLM parameters, C denotes the total floating-point operations for single sequence inference, L indicates the number of decoder layers, S stands for the input sequence length, and H is the size of the hidden state of LLM. We can observe that *PP* is far more communication-efficient than *SP* and *TP* ($L \gg N$) because each device only needs to exchange a subset of output activations with neighboring workers, eliminating the need for multiple tensor synchronizations at each layer.

2) *Issues of Existing Collaborative Edge Inference Systems*: Most existing collaborative edge inference systems [8], [9] employ *TP* and *SP* as the primary principle for parallel LLMs inference. However, employing *TP* and *SP* involves multiple tensor synchronization in each decoder layer, resulting in significant communication overhead. Specifically, Table II summarizes the communication-to-computation latency ratios observed during single-sequence prefilling using various parallelism methods on an edge platform with four Jetson Xavier NX [18]. We observe that the ratio can reach up to 8.2 times for *TP* and *SP*-based methods under typical edge network bandwidth. Despite efforts by these systems to design sophisticated communication optimization techniques, our evaluation in §VI has revealed that communication time still remains a bottleneck under low-bandwidth edge environments. Besides the aforementioned methods, a few research efforts [10] have explored the use of *PP* to orchestrate edge devices. However, for single-sequence requests, these methods ultimately degrade to serial inference, preventing the concurrent utilization of multiple edge devices. Moreover, all of the aforementioned works have focused solely on optimizing the prefill phase, neglecting the autoregressive decoding phase, which is also a critical part of the generative LLMs' inference process.

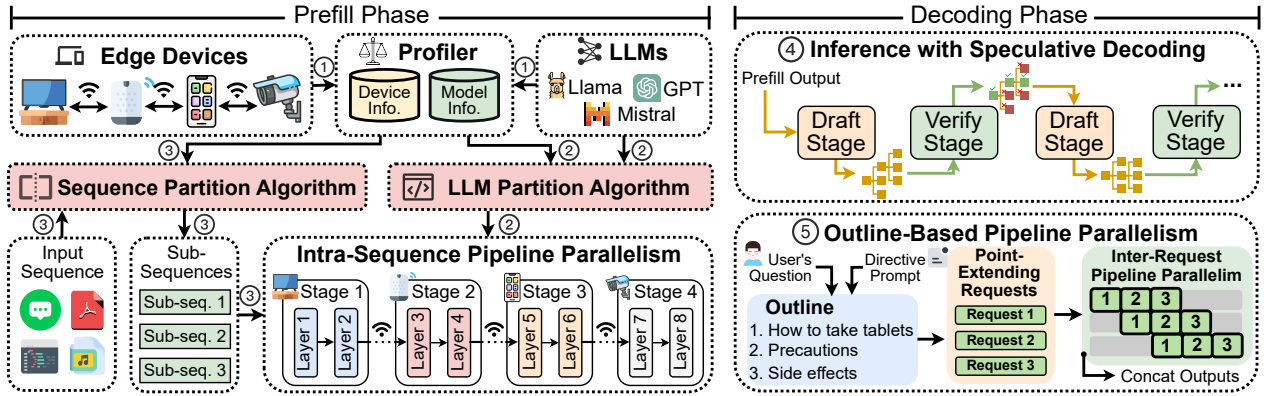


Fig. 4. Jupiter system overview.

D. Design Goal and Technical Challenges

As summarized earlier, existing collaborative edge systems for generative LLM inference exhibit limitations. Alternatively, we revisit these limitations and endeavor to propose a more fast, scalable and resource-efficient collaborative edge AI system, guided by the following design goals: (1) Support parallel acceleration of single-sequence inference during both the prefill and autoregressive decoding phases. (2) Achieve scalable memory footprint reduction per device by distributing the LLM parameters across participating devices. (3) Minimize tensor exchanges between devices to maintain high inference performance even in extremely low-bandwidth edge environments. To achieve the above design goals, instead *TP*, we adopt a pipelined architecture as a principle to orchestrate collaborative edge devices. Each device holds a portion of the parameters, enabling the collective memory of multiple devices to support the target LLM. Furthermore, pipelined architecture is highly communication-efficient, as devices exchange only a small set of intermediate activations with their neighbors. Despite the opportunities, adopting pipelined architecture still suffers from the following non-trivial challenges:

- *How to accelerate the prefill phase with pipeline parallelism?* The prefill phase is computation-intensive and incurs significant time-to-first-token latency, particularly for long prompts. This motivates us to augment available computing power with collaborative edge devices. However, traditional pipelined inference degrades to serial inference for single-sequence requests, necessitating the exploration of new parallel opportunities for our pipelined architecture.
- *How to accelerate the autoregressive decoding phase with pipeline parallelism?* Token-by-token autoregressive decoding is notorious for its prolonged inference process and the inherent difficulty in parallelization. Effectively leveraging edge resources in a pipeline manner poses a significant challenge to the decoding acceleration.

III. JUPITER SYSTEM OVERVIEW

Fig. 4 illustrates an overview of our proposed Jupiter system, which incorporates dedicated designs to optimize both the prefill and decoding phases. In the prefill phase, Jupiter profiler first conducts an LLM prefill process using calibration sequences with varying lengths on the edge devices

to record the run-time traces necessary for parallelism planning algorithm (①). LLM partition algorithm takes the target LLM and profiling results as inputs to generate an optimal pipelined partition, considering both the resource heterogeneity and memory budget of edge devices (②). The input sequence is processed by the sequence partition algorithm, which divides it into multiple sub-sequences. These sub-sequences are then concurrently fed into the inference pipeline, enabling resource-efficient intra-sequence parallel inference (③). Jupiter incorporates two modules designed to maximize the utilization of computational resources by fully exploiting the parallelism potential inherent in the decoding process. Specifically, we first incorporate speculative decoding into our collaborative edge inference system to enhance resource efficiency by processing multiple candidate tokens in parallel (④). Next, to further boost parallelism potential by leveraging multiple edge devices concurrently during the decoding phase, we introduce an outline-based pipeline parallel decoding method (⑤).

IV. PARALLEL ACCELERATION FOR PREFILL PHASE

A. Intra-Sequence Pipeline Parallelism for Generative LLMs

1) *Pipelined Inference for LLMs:* Jupiter adopt a pipelined architecture [10] as a principle to orchestrate collaborative edge devices, which involves partitioning the decoder layers of an LLM into multiple *stages*. Each stage contains a stage model composed of a set of consecutive decoder layers and is mapped to a separate edge device that performs the forward pass (FP) for the corresponding stage model. Single-sequence requests are prevalent in edge intelligence services; in this setup, only one device is active at any given time, as depicted in Fig. 5(①). Ideally, we aim for all edge devices to be active concurrently to fully exploit their resource potential.

2) *Opportunities of Intra-Sequence Parallel Inference:* To fully boost the parallel potential of our pipelined architecture, we propose to partition the input sequence into multiple sub-sequences along the sequence dimension and inject them into the pipeline concurrently to increase parallelism, as depicted in Fig. 5(②). Although the computation of the *QKV Project* and *FFN* modules in decoder layers depends on each individual sub-sequence, the *self-attention* operation in the *MHA* module requires calculating dependencies and relationships between

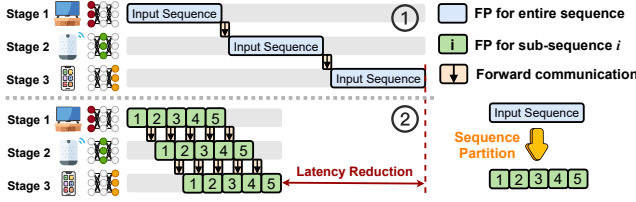


Fig. 5. An illustration of pipelined inference with three edge devices.

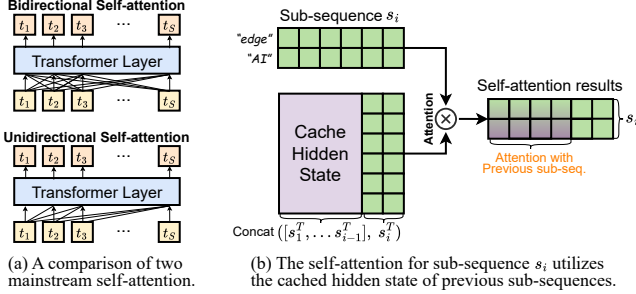


Fig. 6. An illustration of opportunities of intra-sequence parallel inference.

tokens across the entire input sequence, making intra-sequence pipeline parallelism a non-trivial task.

In this work, we leverage the key observation that the causal decoder, the predominant architecture used in current generative LLMs, employs a unidirectional attention mask to ensure that each input token can only attend to previous tokens. Consequently, for an input sequence (t_1, t_2, \dots) , the computation of self-attention of a token t_i involves only the preceding tokens t_1, \dots, t_{i-1} , as depicted in Fig. 6(a). This property brings opportunities for pipeline parallelism within a single input sequence. Specifically, we partition an input sequence into M sub-sequences: (s_1, s_2, \dots) , where each sub-sequence consists of a set of consecutive tokens. We then inject all sub-sequences into the pipeline sequentially. During inference, each stage caches the hidden states of every sub-sequence. When computing for s_i , we utilize the cached hidden states of s_1, \dots, s_{i-1} to ensure correct self-attention results for s_i , as depicted in Fig. 6(b).

B. Resource-Efficient Parallelism Planning

In this section, we detail the parallelism planning for optimal partitioning of LLMs and input sequences.

1) *Selecting Optimal LLMs Partition:* To enable pipelined inference, we need to partition the target LLM into multiple stages and map them to edge devices. As with any pipeline, the steady-state throughput is determined by the execution time of the slowest stage. Consequently, we endeavor to partition the LLM into balanced stages. We consider an LLM consisting of L layers (embedding, decoder and output head, etc.) and denote \mathcal{D} as an ordered set of all devices involved in parallel inference. $\mathcal{D}_n = \{d_1, \dots, d_n\}$ denotes the subset of first n device in \mathcal{D} . $A(i \rightarrow j, \mathcal{D}_n)$ denote the time taken by the slowest stage in the optimally balanced sub-pipeline between layer i to j with \mathcal{D}_n . The goal of our algorithm is to estimate $A(1 \rightarrow L, \mathcal{D})$. To solve this balanced partitioning problem, we break the pipeline into sub-pipelines and leverage the

idea of dynamic programming. The formula of the dynamic programming algorithm can be written as:

$$A(1 \rightarrow y, \mathcal{D}_n) = \min_{1 \leq l < y} \max \begin{cases} A(1 \rightarrow l, \mathcal{D}_{n-1}), \\ T(l+1 \rightarrow y, d_n), \end{cases} \quad (1)$$

where $T(i \rightarrow j, n) = \sum_{l=i}^j t_l^n$ represents the time taken by device d_n to process layers i through j . t_l^n is averaged from profiling on physical edge devices using a calibration dataset. If the memory required for the stage model spanning layers i through j (including LLM parameters and $KVCache$) exceeds the memory budget of device d_n , then $T(i \rightarrow j, n) = +\infty$.

2) *Selecting Optimal Sequence Partition:* As previously discussed, we can partition the input sequence into multiple sub-sequences to boost parallelism. However, partitioning the input sequence is not trivial for the following reasons: (1) Increasing the number of sub-sequences can efficiently boost parallelism and reduce pipeline bubbles. However, this results in shorter sub-sequences, which may underutilize the mobile accelerator (e.g., GPU, NPU), potentially leading to longer processing times. (2) Partitioning the sequence into sub-sequences of equal length for pipelining is not optimal. As previously discussed, the self-attention of sub-sequence s_i depends on previous s_1, \dots, s_{i-1} . Consequently, later sub-sequences carry a heavier computational load than earlier ones. (3) The input sequence lengths vary across different requests, necessitating the determination of the optimal partitioning strategy for each length to accommodate varying tasks.

To overcome above challenges, we design a sophisticated sequence partition algorithm to achieve optimal pipeline efficiency. We denote the maximum length of the input sequence that the target LLM can process as S_{max} . To avoid underutilizing devices, we first profile the accelerator utilization on each device at various input sequence lengths ($< S_{max}$). We then decide the minimum sub-sequence length, denoted as b , that prevents underutilization of all mobile accelerators.

After optimal LLM partitioning, we obtain a partitioned LLM with multiple stage models, each mapped to an edge device and having exact same computational latency. We use h_i to represent the inference latency of sub-sequence s_i at each stage, as shown in Fig. 7(Left). As detailed earlier, in our intra-sequence parallel inference, the inference latency of each sub-sequence s_i depends on s_1, \dots, s_{i-1} and itself. We use $q(x, y)$ to denote the inference latency for a sub-sequence of length x , given the total length y of its previous sub-sequences. Thus, h_i can be expressed as $h_i = q(|s_i|, \sum_{j=1}^{i-1} |s_j|)$. We meticulously profile $q(x, y)$ on a realistic edge platform under various sequence lengths x and y . The profiling overhead can be linearly reduced by conducting concurrent profiling on multiple devices and approximating results through interpolation.

The goal of our algorithm is to find optimal partition schemes for sequences of varying lengths ($< S_{max}$), ensuring that the inference latency of each sub-sequence is as balanced as possible while each sub-sequence length exceeds b to avoid device underutilization. We leverage dynamic programming to achieve the goal. We denote $W(i \rightarrow j, k)$ as the inference latency of the slowest sub-sequence in the optimal partitioning

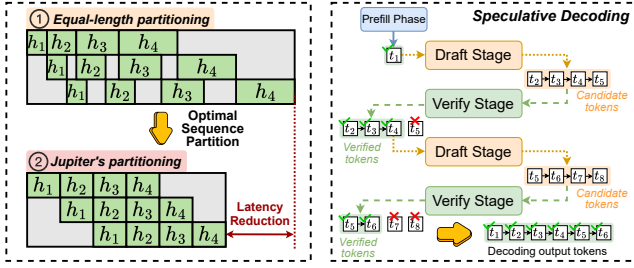


Fig. 7. Left: Comparison between equal-length and Jupiter's partition. Right: An illustration of the decoding phase with Speculative Decoding.

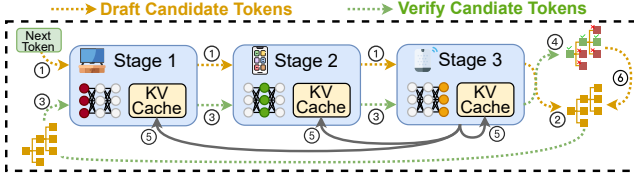


Fig. 8. A workflow of our collaborative inference with speculative decoding. of the sequence from token i to j into k sub-sequences. The formula of the dynamic programming can be written as:

$$W(1 \rightarrow y, k) = \min_{1 \leq l < y} \max \begin{cases} W(1 \rightarrow l, k-1), \\ T^*(y-l, l). \end{cases} \quad (2)$$

$$T^*(y-l, l) = \begin{cases} +\infty, & \text{if } y-l < b, \\ q(y-l, l), & \text{else.} \end{cases} \quad (3)$$

When solving for Eq. 2, the sentence length y is iterated from 1 to S_{max} , and k is iterated from 1 to $4|\mathcal{D}|$. We set the maximum number of sub-sequences to $4|\mathcal{D}|$, which effectively boosts the pipeline parallelism while preventing excessive planning algorithm runtime. For each sentence length, we record the optimal balanced partitioning strategy for dividing the sentence into a varying number of sub-sequences.

Upon the completion of dynamic programming process, we need to select the optimal number of sub-sequences k for each input length y . We observe from Fig. 7(Left) that the total inference latency for a sequence of length y partitioned into k sub-sequences can be estimated by:

$$\text{Latency} = \sum_{i=1}^k h_i + (|\mathcal{D}| - 1) \times W(1 \rightarrow y, k). \quad (4)$$

We choose k to minimize Eq. 4 for each sequence length y .

3) **Complexity**: The time complexity for our optimal LLM partition is $\mathcal{O}(L^2|\mathcal{D}|)$, while for the optimal sequence partition it is $\mathcal{O}(S_{max}^2|\mathcal{D}|)$. In our experiments, the entire planning process is completed in under five minutes on an edge device. Notably, parallelism planning is a one-shot offline process and its outputs can be stored and reused. The overhead of it can be amortized across thousands of inference iterations.

V. COLLABORATIVE INFERENCE FOR DECODING PHASE

The decoding latency of LLMs has become a substantial obstacle to high-quality human-computer interactions. This latency stems from the token-by-token generation required by autoregressive decoding, causing significant delays in producing long outputs. To accelerate LLM decoding, an intuitive way involves leveraging idle computational resources to enhance parallelism. Jupiter introduces two parallel decoding strategies to accelerate the decoding phase.

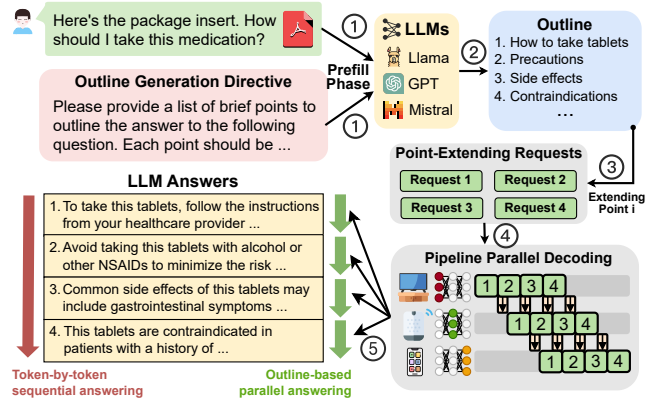


Fig. 9. An illustration of our outline-based pipeline parallel decoding.

A. Collaborative Inference with Speculative Decoding

Speculative decoding [19]–[23] has emerged as a promising paradigm for accelerating decoding. In each decoding step, speculative decoding first drafts multiple candidate tokens efficiently, speculating on future decoding steps. This is achieved using extra lightweight heads atop current LLM backbone (*Self-Drafting*) or a small independent draft model (*Independent Drafting*). These candidate tokens are then verified in parallel by original LLM to ensure the outputs align with the original distribution, as illustrated in Fig. 7(Right). By leveraging parallel token generation, speculative decoding significantly reduces the total number of decoding steps required.

Jupiter incorporates the idea of *Self-Drafting* from speculative decoding into our collaborative edge inference system to enhance resource efficiency during decoding phase. Fig. 8 illustrated our workflow. Specifically, the next token produced during the prefill phase is fed into the LLM for a forward pass to obtain logits (1). The logits will be processed by the FFN heads incorporated atop the LLM to generate multiple candidate tokens in parallel (2). The candidate tokens will be transferred from the final stage back to the initial stage and fed into the LLM for a forward pass. The intermediate results of all candidate tokens will be precisely stored in the *KVCache* (3). The posterior probability of each speculative candidate will be evaluated to determine whether it should be accepted or rejected (4). Subsequently, the final stage will inform all stages of the rejected candidate tokens, directing them to remove these tokens from the *KVCache* (5). We extract the logits of the accepted tokens from the output produced in step (3). These logits are processed by the draft heads to generate a new batch of candidate tokens for the next decoding iteration (6). Our workflow design and implementation minimize redundant LLM calls, ensuring that each draft and verification process requires only one LLM inference. Jupiter supports the flexible plug-and-play replacement of various self-drafting speculative decoding algorithms. In our evaluation, we incorporate the *token tree*-based speculative decoding algorithm proposed by Medusa [23].

B. Outline-Based Pipeline Parallel Decoding

Incorporating speculative decoding with collaborative inference enables parallel processing of multiple candidate tokens,

thus attaining improved device utilization. However, merely applying speculative decoding on pipelined inference yet operates serially, failing to leverage multiple devices concurrently.

To fully exploit idle computational resources at the edge, we intend to further explore the parallel potential during the decoding phase. Towards that, we borrow wisdom from human thinking, which usually organizes thoughts against questions first and then responds point-by-point. This routine, in many situations, is more common and efficient than purely sequential answering and has been experimentally verified by many recent explorations [24]–[26]. These works typically guide LLMs in generating an explicit chain or tree-like thought process, subsequently eliciting high-quality answers.

Inspired by them, we introduce an outline-based pipeline parallel decoding method, with its workflow illustrated in Fig. 9. Specifically, we first concatenate a crafted outline generation directives with the user’s question for prefilling (①) to guide the LLM in organizing its thoughts and generating an outline of the answers (②). The prefill for static guide prompts can be performed offline in advance and cached into the *KVCache*. Next, we package each point from the outline into separate point-extending requests. Each request guides the LLM to expand solely on that specific point (③). These requests are then injected into the collaborative inference pipeline concurrently for efficient pipeline parallelism (④). The *KVCache* of input sequence generated during the prefill phase will be shared across all requests, thereby eliminating redundant computations. Finally, after all point-extending requests are finished, we will concatenate the outputs from each request to obtain the final answer (⑤).

Note that different tasks (e.g., document summarization, and question answering) can utilize distinct outline generation directives in generating outlines appropriate for parallel inference. For tasks requiring step-by-step reasoning with chained logical dependencies, such as math and coding, or tasks needing only short answers, the outline-based parallel decoding method may not generate high-quality responses. Therefore, our system design includes outline-based parallel decoding as a flexible, pluggable module. For problem types less suited to it, the system can automatically decide or let the user choose whether to use it, thus avoiding unsatisfactory results. In these cases, our inference system automatically defaults to speculative decoding for sequential answering, and experimental evaluations demonstrate that Jupiter can still achieve outstanding performance in latency reduction.

VI. IMPLEMENTATION AND EVALUATION

A. Experimental Setups

1) *Models and Datasets*: We evaluate Jupiter using 2 LLMs from the Llama2 series [14], specifically Llama2-7B and Llama2-13B (both with INT4 quantization). We employ 3 recent assistant-style datasets: LiMA [27], Vicuna-80 [28], and WizardLM [29]. LiMA is used to evaluate inference latency, and all datasets are utilized for assessing generation quality.

TABLE III
SPECIFICATIONS OF EDGE DEVICES IN EXPERIMENTS.

Edge Device	GPU Processor	Memory	Power
Jetson Xavier NX [18]	384-core NVIDIA Volta	8GB	20W
Jetson TX2 [30]	256-core NVIDIA Pascal	8GB	20W
Jetson Nano [31]	128-core NVIDIA Maxwell	8GB	10W

2) *Edge Environment Setup*: We use three heterogeneous off-the-shelf edge devices, as listed in Table III, in our experiments. We evaluate Jupiter’s performance in two realistic edge environments, incorporating both homogeneous and heterogeneous configurations. *Homogeneous Environment A* consists of 4×NX, while *Heterogeneous Environment B* comprises 1×NX, 2×TX2, and 1×Nano. We adjust the device-to-device communication bandwidth to simulate the diverse network conditions within realistic edge environments.

3) *Baseline Methods*: We compare Jupiter with five state-of-the-art parallel LLMs inference method:

- *Sequence Parallelism (SP)* [17] is pioneering work that proposes *SP* for distributed LLM execution in datacenters.
- *Megatron-LM (M-LM)* [11] is pioneering work that proposes *TP* for distributed LLM execution in datacenters.
- *DeTransformer (DT)* [9] is a *TP*-based collaborative edge inference system that strikes a trade-off between communication overhead and inference accuracy by reducing the frequency of tensor synchronization. In our evaluation, we selected the decoupled layers to be half of the total layers.
- *Galaxy* [8] is a collaborative edge inference system that employs *TP* across MHA and FFN modules, with *SP* applied to the connecting operations. It employs fine-grained overlapping of comm. and comp. to mitigate inference latency.
- *EdgeShard* [10] is a collaborative edge inference system that employs pipelined architecture to orchestrate edge devices.

Given that existing inference systems do not optimize the decoding phase, we equipped them with the *naive* token-by-token sequence generation method. During the decoding phase, *SP* will degrade to single-device inference due to the sequence length being one.

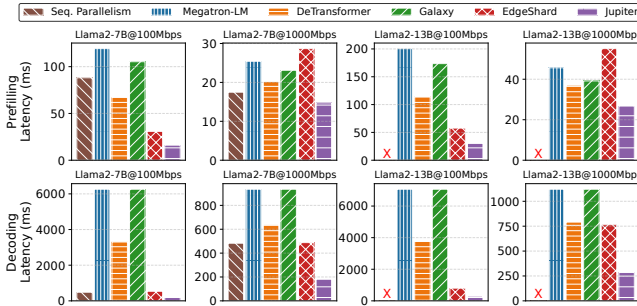
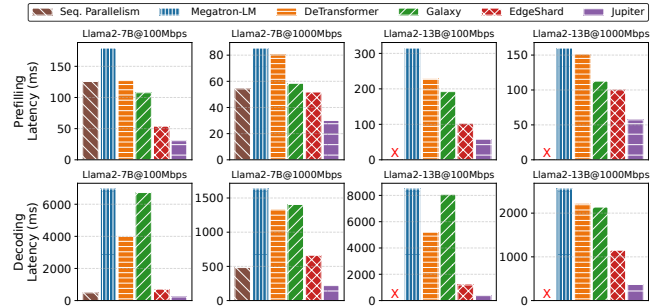
B. End-to-End Performance

Table IV summarizes the end-to-end generation latency of Jupiter and baselines. We sample prompts from LiMA dataset with an average sequence length of 260 tokens and set the maximum generation length to 64 tokens. The results indicate that Jupiter consistently outperforms all baselines across various models, edge environments, and edge network bandwidths. Specifically, when compared to *TP*-based parallel inference methods like M-LM, DT and Galaxy, Jupiter achieves up to 26.1× latency reduction. When compared to *SP*-based parallel inference method like *SP*, Jupiter achieve up to 3.3× latency reduction. *SP* degrades to single-device inference during the decoding phase, avoiding significant communication overhead but wasting resources on idle devices. Additionally, *SP* requires each device to accommodate all parameters, causing out-of-memory (OOM) issues with a 13B model. When compared to pipelined methods like EdgeShard, Jupiter achieves up to a 2.7× reduction in

TABLE IV

END-TO-END GENERATION LATENCY (IN SECONDS) FOR LIMA DATASET INCLUDING THE PREFILL AND DECODING PHASES UNDER VARIOUS SETTINGS.

Edge Environment	Network Bandwidth	Llama2-7B						Llama2-13B					
		SP	M-LM	DT	Galaxy	EdgeShard	Jupiter	SP	M-LM	DT	Galaxy	EdgeShard	Jupiter
Homo. Env. A	100Mbps	53.5	431.2	228.5	427.6	42.2	16.5	OOM	503.4	270.1	496.5	66.2	26.3
	500Mbps	37.4	106.9	66.4	103.9	39.0	15.2	OOM	130.1	83.4	125.0	63.4	25.2
	1Gbps	35.4	66.4	46.1	65.0	38.6	14.9	OOM	83.4	60.1	81.3	63.1	24.9
Hetero. Env. B	100Mbps	63.1	491.2	288.6	458.3	59.3	22.4	OOM	624.5	391.2	566.4	102.4	38.8
	500Mbps	47.0	167.0	126.4	142.9	56.1	21.4	OOM	251.2	204.5	208.0	99.7	37.3
	1Gbps	44.8	126.4	106.2	104.9	55.7	20.9	OOM	204.5	181.2	165.7	98.3	36.8

Fig. 10. Evaluate in Homogeneous Environment A. The average per-token processing/generation latency in prefill/decoding phase. \times indicates OOM.Fig. 11. Evaluate in Heterogeneous Environment B. The average per-token processing/generation latency in prefill/decoding phase. \times indicates OOM.

latency by fully leveraging the computational resources of multiple devices concurrently. *Jupiter*'s resource-efficient parallelism planning accounts for the computational resources of heterogeneous devices, consistently outperforming baselines in heterogeneous environment B and achieving a $2.6\times$ to $21.9\times$ latency reduction.

C. Phase-Wise Analysis

We further investigate the performance improvements in the prefill and decoding phases separately. In Fig. 10 and 11 we report the average per-token processing latency during the prefill phase and generation latency during the decoding phase in homogeneous and heterogeneous environments, respectively. For the prefill phase, *Jupiter* achieves a $1.4\times$ to $7.4\times$ reduction in latency compared to the baselines. Despite efforts by state-of-the-art edge inference systems like DT and Galaxy to design sophisticated communication optimization techniques, such as fine-grained communication-computation overlapping, these methods still perform poorly in bandwidth-constrained edge networks. For the decoding phase, *Jupiter* significantly outperforms the baselines, achieving a $2.9\times$ to $33.2\times$ reduction in latency. In TP and SP-based parallel architectures, the high communication-to-computation ratio is further exacerbated during token-by-token autoregressive generation, while pipelined architectures fail to concurrently utilize the computational resources of multiple devices. These issues collectively amplify the severe resource wastage problem during decoding phase. In contrast, *Jupiter*'s system design fully boosts parallelism potential during autoregressive generation, significantly accelerating the decoding phase.

D. Scalability

We analyze the scalability of *Jupiter* on a 4-node homogeneous Jetson Xavier NX cluster. We use the same set of input prompts as in §VI-B, with a maximum generation length of 64 tokens. The results are summarized in Fig. 12. We observe that *Jupiter* exhibits substantial scalability even under a bandwidth-limited (100Mbps) edge environment. When compared to existing state-of-the-art collaborative edge inference frameworks, *Jupiter* can achieve up to $23.7\times$ latency reduction. The high communication-to-computation ratio of these frameworks makes it challenging to scale resource-efficiently in bandwidth-constrained edge environments. The scalability analysis indicates that our *Jupiter* framework enables the addition of more edge devices to aggregate computational resources, allowing for parallel acceleration of inference and leveraging collective memory to support larger LLMs.

E. Decoding Speedup and Generation Quality Assessment

1) *Decoding Speedup Analysis*: We investigate the performance boost of each decoding optimization module on homogeneous edge environment A. In our evaluation, we incorporate the *token tree*-based speculative decoding algorithm proposed by Medusa [23], which utilizes five draft heads with top-1 prediction. We conduct an ablation study to assess the contributions of speculative decoding and outline-based pipeline parallelism, as depicted in Table V. We observe that both modules achieve effective decoding acceleration, with an overall speedup ratio of up to $3.9\times$.

2) *Generation Quality Assessment*: Previous works [21]–[23] have shown that speculative decoding achieves nearly lossless generation results compared to naive token-by-token sequence generation, as its verification process will correct

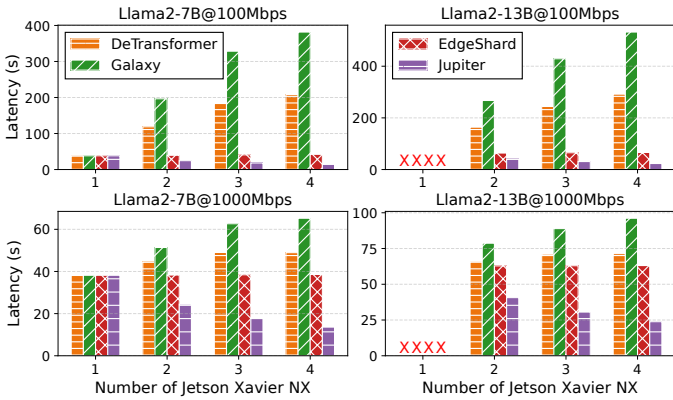


Fig. 12. End-to-end inference latency with a varying number of Jetson Xavier NX under 100Mbps and 1Gbps network bandwidths. × indicates OOM.

the output distribution. In this section, we separately assess the generation quality of our outline-based pipeline parallel decoding method. We select the widely adopted LLM-based evaluation framework FastChat [32] to compare the answer quality of naive token-by-token sequence generation (*naive generation*) and our outline-based parallel generation. FastChat, empowered by GPT-4o, will assign a quality score between 1 and 10 for each answer. As summarized in Table VI, we compared the generation quality of naive generation and our method on the Vicuna-80, WizardLM, and LiMA datasets. We observe that our outline-based pipeline parallel decoding significantly reduces latency while maintaining comparable generation quality to that of naive generation. However, across all three datasets, the overall generation quality of our outline-based parallel decoding method is slightly lower than that of naive generation. Therefore, we conducted further experiments on the Vicuna-80 dataset to analyze the reasons for the lower quality. We manually selected five question categories from the Vicuna-80 dataset: *Generic*, *Knowledge*, *Counterfactual*, *Coding*, and *Math*, and evaluated them using FastChat, as shown in Table VII. Our results show that for *Generic*, *Knowledge*, *Counterfactual* questions, our outline-based method achieved comparable or superior generation quality. However, for tasks requiring step-by-step reasoning with chained logical dependencies, such as *Coding* and *Math*, our outline-based parallel decoding exhibited lower generation quality. Therefore, our system design incorporates outline-based parallel decoding as a flexible, pluggable module. For problem types less suited to it, the system can automatically decide or let the user choose whether to use it, thus avoiding unsatisfactory answers.

VII. RELATED WORK

A. Collaborative Edge Computing for DNN Inference

CoEdge and DeepThings [33], [34] enable the distributed execution of CNN-based inference applications on resource-constrained edge clusters. Galaxy and DeTransformer [8], [9], [35] utilize *TP* to accelerate transformer inference with collaborative edge devices. EdgeShard [10] orchestrates edge devices in a pipelined manner for sequential inference.

TABLE V
SPEEDUP OVER NAIVE SEQUENTIAL GENERATION. SD: SPECULATIVE DECODING. OP: OUTLINE-BASED PARALLEL DECODING.

Model	Speedup Over Naive			
	Naive	Jupiter w/o OP	Jupiter w/o SD	Jupiter
Llama2-7B	1.0×	1.8×	2.3×	3.6×
Llama2-13B	1.0×	2.0×	2.4×	3.9×

TABLE VI
OVERALL ANSWERS QUALITY OF NAIVE AND JUPITER'S METHOD.

Method	Vicuna-80		WizardLM		LiMA	
	Llama2-7B	Llama2-13B	Llama2-7B	Llama2-13B	Llama2-7B	Llama2-13B
Naive	7.05	7.19	6.26	6.85	6.72	6.77
Jupiter	6.59	6.85	6.21	6.70	6.20	6.25

TABLE VII
ANSWERS QUALITY ON VARIOUS QUESTION CATEGORIES IN VICUNA-80.

Model	Method	Quality on different question categories				
		Generic	Knowledge	Counterfactual	Coding	Math
Llama2-7B	Naive	6.17	7.17	6.75	7.07	4.33
	Jupiter	7.33	7.17	6.75	4.07	3.16
Llama2-13B	Naive	7.25	7.33	7.25	7.07	4.83
	Jupiter	7.58	7.67	7.75	4.29	2.33

B. Parallel LLMs Execution Architectures

DP [36], [37] is the most extensively used distributed execution method in datacenter. Gpipe [12] and subsequent works [38], [39] leverage *PP* to address the memory challenges associated with executing LLMs that contain billions of parameters. Megatron [11] initially introduced *TP* as a principle to parallelize the execution of the transformer-based LLMs. To overcome the limitations of handling long sequences, [17], [40], [41] borrow the concept of intra-sequence parallelism from recurrent neural networks, splitting long sequences across multiple devices for concurrent execution.

C. Speculative Decoding for LLMs Inference Acceleration

Blockwise decoding [20] is a pioneering work proposing the *Draft-then-Verify* paradigm. [21], [22], [42] further unleashes its potential and utilizes independent lightweight LLMs to perform the drafting task both accurately and efficiently. While leveraging an external drafter model offers considerable advantages, obtaining an appropriate draft model remains challenging. To address that, numerous studies suggest leveraging the target LLM itself for efficient self-drafting [23], [43], [44].

VIII. CONCLUSION

This paper introduces Jupiter, a fast and scalable collaborative edge inference framework for generative LLMs. Jupiter employs a communication-efficient and resource-scalable pipelined architecture, combined with sophisticated system design, to parallelize and accelerate the prefill and decoding phases. Our extensive evaluation demonstrates that Jupiter achieves up to 26.1× end-to-end generation latency reduction compared to state-of-the-art methods.

REFERENCES

- [1] OpenAI, "Chatgpt: Openai language model (gpt-4)," <https://www.openai.com/chatgpt>, 2024, accessed: 2024-07-22.
- [2] E. King, H. Yu, S. Lee, and C. Julien, "Sasha: creative goal-oriented reasoning in smart homes with large language models," *Proceedings of the ACM on IMWUT*, vol. 8, no. 1, pp. 1–38, 2024.
- [3] Y. Li, H. Wen, W. Wang, X. Li, Y. Yuan, G. Liu, J. Liu, W. Xu, X. Wang, Y. Sun *et al.*, "Personal llm agents: Insights and survey about the capability, efficiency and security," *arXiv preprint arXiv:2401.05459*, 2024.
- [4] S. Ye, L. Zeng, Q. Wu, K. Luo, Q. Fang, and X. Chen, "Eco-fl: Adaptive federated learning with efficient edge collaborative pipeline training," in *Proceedings of the 51st International Conference on Parallel Processing*, 2022, pp. 1–11.
- [5] S. Ye, L. Zeng, X. Chu, G. Xing, and X. Chen, "Asteroid: Resource-efficient hybrid pipeline parallelism for collaborative dnn training on heterogeneous edge devices," in *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, 2024, pp. 312–326.
- [6] L. Zeng, S. Ye, X. Chen, and Y. Yang, "Implementation of big ai models for wireless networks with collaborative edge computing," *IEEE Wireless Communications*, vol. 31, no. 3, pp. 50–58, 2024.
- [7] L. Zeng, S. Ye, X. Chen, X. Zhang, J. Ren, J. Tang, Y. Yang, and S. Xuemin (Sherman), "Edge graph intelligence: Reciprocally empowering edge networks with graph intelligence," *IEEE Communications Surveys & Tutorials*, vol. 27, 2025.
- [8] 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 *IEEE INFOCOM 2024-IEEE Conference on Computer Communications*. IEEE, 2024.
- [9] Y. Wei, S. Ye, J. Jiang, X. Chen, D. Huang, J. Du, and Y. Lu, "Communication-efficient model parallelism for distributed in-situ transformer inference," in *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2024, pp. 1–6.
- [10] M. Zhang, J. Cao, X. Shen, and Z. Cui, "Edgeshard: Efficient llm inference via collaborative edge computing," *arXiv preprint arXiv:2405.14371*, 2024.
- [11] D. Narayanan, M. Shoyebi, J. Casper, P. LeGresley, M. Patwary, V. Korthikanti, D. Vainbrand, P. Kashinkunti, J. Bernauer, B. Catanzaro *et al.*, "Efficient large-scale language model training on gpu clusters using megatron-lm," in *SC*, 2021, pp. 1–15.
- [12] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu *et al.*, "Gpipe: Efficient training of giant neural networks using pipeline parallelism," *NeurIPS*, vol. 32, 2019.
- [13] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners," *NeurIPS*, vol. 33, pp. 1877–1901, 2020.
- [14] 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.
- [15] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. d. l. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier *et al.*, "Mistral 7b," *arXiv preprint arXiv:2310.06825*, 2023.
- [16] I. Beltagy, M. E. Peters, and A. Cohan, "Longformer: The long-document transformer," *arXiv preprint arXiv:2004.05150*, 2020.
- [17] S. Li, F. Xue, C. Baranwal, Y. Li, and Y. You, "Sequence parallelism: Long sequence training from system perspective," in *The 61st Annual Meeting Of The Association For Computational Linguistics*, 2023.
- [18] "Jetson-nx," <https://developer.nvidia.com/blog/jetson-xavier-nx-the-worlds-smallest-ai-supercomputer>, 2019.
- [19] 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," *arXiv preprint arXiv:2401.07851*, 2024.
- [20] M. Stern, N. Shazeer, and J. Uszkoreit, "Blockwise parallel decoding for deep autoregressive models," *NeurIPS*, vol. 31, 2018.
- [21] Y. Leviathan, M. Kalman, and Y. Matias, "Fast inference from transformers via speculative decoding," in *International Conference on Machine Learning*. PMLR, 2023, pp. 19274–19286.
- [22] 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.
- [23] 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," *arXiv preprint arXiv:2401.10774*, 2024.
- [24] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou *et al.*, "Chain-of-thought prompting elicits reasoning in large language models," *NeurIPS*, vol. 35, pp. 24824–24837, 2022.
- [25] S. Yao, D. Yu, J. Zhao, I. Shafraan, T. Griffiths, Y. Cao, and K. Narasimhan, "Tree of thoughts: Deliberate problem solving with large language models," *NeurIPS*, vol. 36, 2024.
- [26] J. Long, "Large language model guided tree-of-thought," *arXiv preprint arXiv:2305.08291*, 2023.
- [27] C. Zhou, P. Liu, P. Xu, S. Iyer, J. Sun, Y. Mao, X. Ma, A. Efrat, P. Yu, L. Yu *et al.*, "Lima: Less is more for alignment," *NeurIPS*, vol. 36, 2024.
- [28] W.-L. Chiang, Z. Li, Z. Lin, Y. Sheng, Z. Wu, H. Zhang, L. Zheng, S. Zhuang, Y. Zhuang, J. E. Gonzalez *et al.*, "Vicuna: An open-source chatbot impressing gpt-4 with 90%* chatgpt quality," See <https://vicuna.lmsys.org> (accessed 14 April 2023), vol. 2, no. 3, p. 6, 2023.
- [29] C. Xu, Q. Sun, K. Zheng, X. Geng, P. Zhao, J. Feng, C. Tao, and D. Jiang, "Wizardlm: Empowering large language models to follow complex instructions," *arXiv preprint arXiv:2304.12244*, 2023.
- [30] "Jetson-tx2," <https://developer.nvidia.com/embedded/jetson-tx2>, 2017.
- [31] "Jetson-nano," <https://developer.nvidia.com/embedded/jetson-nano-developer-kit>, 2019.
- [32] L. Zheng, W.-L. Chiang, Y. Sheng, S. Zhuang, Z. Wu, Y. Zhuang, Z. Lin, Z. Li, D. Li, E. Xing *et al.*, "Judging llm-as-a-judge with mt-bench and chatbot arena," *Advances in Neural Information Processing Systems*, vol. 36, 2024.
- [33] L. Zeng, X. Chen, Z. Zhou, L. Yang, and J. Zhang, "Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices," *IEEE/ACM Transactions on Networking*, vol. 29, no. 2, pp. 595–608, 2020.
- [34] Z. Zhao, K. M. Barijough, and A. Gerstlauer, "Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, 2018.
- [35] J. Du, Y. Wei, S. Ye, J. Jiang, X. Chen, D. Huang, and Y. Lu, "Co-designing transformer architectures for distributed inference with low communication," *IEEE Transactions on Parallel and Distributed Systems*, 2024.
- [36] M. Li, D. G. Andersen, A. J. Smola, and K. Yu, "Communication efficient distributed machine learning with the parameter server," *NeurIPS*, vol. 27, 2014.
- [37] S. Rajbhandari, J. Rasley, O. Ruwase, and Y. He, "Zero: Memory optimizations toward training trillion parameter models," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2020, pp. 1–16.
- [38] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM symposium on operating systems principles*, 2019, pp. 1–15.
- [39] S. Fan, Y. Rong, C. Meng, Z. Cao, S. Wang, Z. Zheng, C. Wu, G. Long, J. Yang, L. Xia *et al.*, "Dapple: A pipelined data parallel approach for training large models," in *PPoPP*, 2021, pp. 431–445.
- [40] Z. Li, S. Zhuang, S. Guo, D. Zhuo, H. Zhang, D. Song, and I. Stoica, "Terapepe: Token-level pipeline parallelism for training large-scale language models," in *International Conference on Machine Learning*. PMLR, 2021, pp. 6543–6552.
- [41] R. Ma, X. Yang, J. Wang, Q. Qi, H. Sun, J. Wang, Z. Zhuang, and J. Liao, "Hpipe: Large language model pipeline parallelism for long context on heterogeneous cost-effective devices," in *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 6: Industry Track)*, 2024, pp. 1–9.
- [42] H. Xia, T. Ge, P. Wang, S.-Q. Chen, F. Wei, and Z. Sui, "Speculative decoding: Exploiting speculative execution for accelerating seq2seq generation," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 3909–3925.
- [43] 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 *ASPLOS, Volume 3*, 2024, pp. 932–949.
- [44] A. Santilli, S. Severino, E. Postolache, V. Maiorca, M. Mancusi, R. Marin, and E. Rodolà, "Accelerating transformer inference for translation via parallel decoding," *arXiv preprint arXiv:2305.10427*, 2023.