SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification

Xupeng Miao*
Carnegie Mellon University
Pittsburgh, PA, USA
xupeng@cmu.edu

Gabriele Oliaro*
Carnegie Mellon University
Pittsburgh, PA, USA
goliaro@cs.cmu.edu

Zhihao Zhang*
Carnegie Mellon University
Pittsburgh, PA, USA
zhihaoz3@andrew.cmu.edu

Xinhao Cheng*
Carnegie Mellon University
Pittsburgh, PA, USA
xinhaoc@andrew.cmu.edu

Zeyu Wang
Carnegie Mellon University
Pittsburgh, PA, USA
zeyuwang@alumni.cmu.edu

Zhengxin Zhang
Tsinghua University
Beijing, China
zhang-zx21@mails.tsinghua.edu.cn

Rae Ying Yee Wong
Stanford University
Stanford, CA, USA
raewong@stanford.edu

Alan Zhu
Carnegie Mellon University
Pittsburgh, PA, USA
aczhu@andrew.cmu.edu

Lijie Yang
Carnegie Mellon University
Pittsburgh, PA, USA
lijiey@andrew.cmu.edu

Xiaoxiang Shi
Shanghai Jiao Tong University
Shanghai, China
lambda7shi@sjtu.edu.cn

Chunan Shi
Peking University
Beijing, China
spirited_away@pku.edu.cn

Zhuoming Chen
Carnegie Mellon University
Pittsburgh, PA, USA
zhuominc@andrew.cmu.edu

Daiyaan Arfeen
Carnegie Mellon University
Pittsburgh, PA, USA
marfeen@andrew.cmu.edu

Reyna Abhyankar
University of California, San Diego
San Diego, CA, USA
vabhyank@ucsd.edu

Zhihao Jia
Carnegie Mellon University
Pittsburgh, PA, USA
zhihao@cmu.edu

Abstract

This paper introduces SpecInfer, a system that accelerates generative large language model (LLM) serving with *tree-based* speculative inference and verification. The key idea behind SpecInfer is leveraging small speculative models to predict the LLM's outputs; the predictions are organized as a token tree, whose nodes each represent a candidate token sequence. The correctness of all candidate token sequences represented by a token tree is verified against the LLM in parallel using a novel tree-based parallel decoding mechanism. SpecInfer uses an LLM as a token tree verifier instead of an incremental decoder, which significantly reduces the end-to-end latency and computational requirement for serving generative LLMs while provably preserving model quality. Our evaluation shows that SpecInfer outperforms existing LLM serving systems by 1.5-2.8× for distributed LLM inference and by 2.6-3.5× for offloading-based LLM inference, while preserving the same generative performance. SpecInfer is publicly available at https://github.com/flexflow/FlexFlow/

Keywords: large language model serving, speculative decoding, token tree verification

ACM Reference Format:
Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. 2024. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '24), April 27-May 1, 2024, La Jolla, CA, USA.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3620666.3651335

1 Introduction

Generative large language models (LLMs), such as ChatGPT [3] and GPT-4 [33], have proven to be powerful in various application domains, including question answering, program synthesis, and task automation [26, 56]. However, it is challenging to quickly and cheaply serve these LLMs

[*] Equal contribution.

ASPLOS '24, April 27-May 1, 2024, La Jolla, CA, USA

ACM ISBN 979-8-4007-0386-7/24/04.
https://doi.org/10.1145/3620666.3651335

due to their large volume of parameters, complex architectures, and high computational requirements. For example, the largest GPT-3 architecture has 175 billion parameters, which requires more than eight NVIDIA 40GB A100 GPUs to store in half-precision floating points, and takes several seconds to serve a single inference request [3].

An LLM generally takes as input a sequence of tokens, called *prompt*, and generates subsequent tokens one at a time, as shown in Figure 1a. The generation of each token in the sequence is conditioned on the input prompt and previously generated tokens and does not consider future tokens. This approach is also called *autoregressive* decoding because each generated token is also used as input for generating future tokens. This dependency between tokens is crucial for many NLP tasks that require preserving the order and context of the generated tokens, such as text completion [55].

Existing LLM systems generally use an *incremental decoding* approach to serving a request where the system computes the activations for all prompt tokens in a single step and then iteratively decodes *one* new token using the input prompt and all previously generated tokens [27]. This approach respects data dependencies between tokens, but achieves sub-optimal runtime performance and limited GPU utilization, since the degree of parallelism within each request is greatly limited in the incremental phase. In addition, the attention mechanism of Transformer [48] requires accessing the keys and values of all previous tokens to compute the attention output of a new token. To avoid recomputing the keys and values for all preceding tokens, today's LLM systems use a caching mechanism to store their keys and values for reuse in future iterations. For long-sequence generative tasks (e.g., GPT-4 supports up to 32K tokens in a request), caching keys and values introduces significant memory overhead, which prevents existing systems from serving a large number of requests in parallel due to the memory requirement of caching their keys and values.

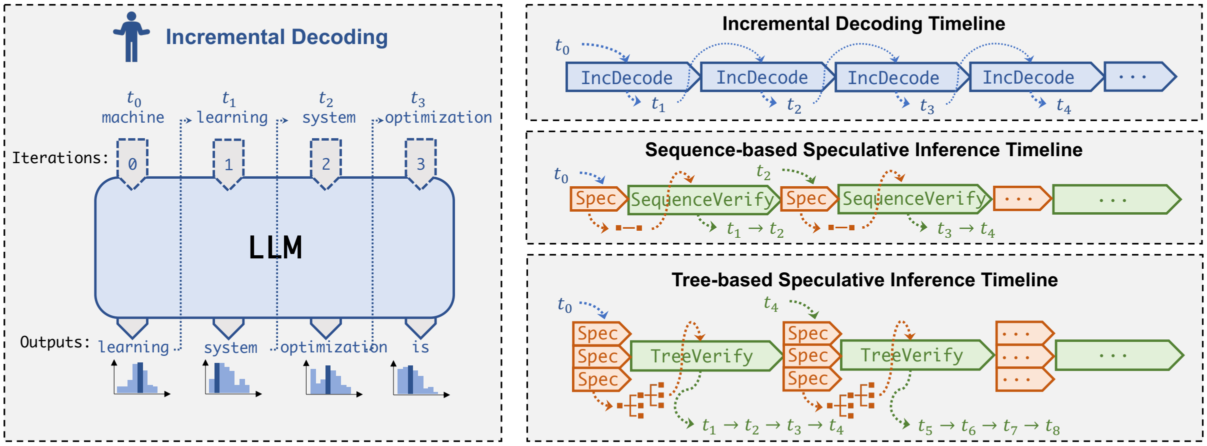
Motivated by the idea of speculative execution in processor optimizations [13, 42], recent work introduces *sequence-based speculative inference*, which leverages a *small speculative model* (SSM) to generate a sequence of tokens and uses an LLM to examine their correctness in parallel [5, 22, 25, 44, 51]. These attempts only consider a token sequence generated by a single SSM for speculation, which cannot align well with an LLM due to the model capacity gap between them, since SSMs are generally orders of magnitude smaller than the LLM to maintain low memory and runtime overheads.

This paper introduces SpecInfer, a system that improves the end-to-end latency and computational efficiency of LLM serving with *tree-based speculative inference and verification*. Figure 1b illustrates a comparison between existing incremental decoding, sequence-based speculative inference, and our tree-based speculative inference. A key insight behind SpecInfer is to simultaneously consider a diversity of speculation candidates (instead of just one as in existing approaches) to maximize speculative performance. These candidates are organized as a *token tree*, whose nodes each represents a sequence of speculated tokens. The correctness of *all* candidate token sequences is verified against the LLM in parallel, which allows SpecInfer to significantly increase the number of generated tokens in an LLM decoding step. Compared with sequence-based speculative inference, leveraging tree structures can significantly improve the success rate of verifying a token (e.g., from 52-57% to 96-97% for stochastic decoding as shown in Table 1). However, realizing this improvement requires addressing two unique challenges. Next, we elaborate on these challenges and the main ideas SpecInfer uses to address them.

First, SpecInfer must explore an extremely large search space of candidate token sequences to maximize speculative performance. While the idea of speculative execution has been widely deployed in a variety of optimization tasks in computer architecture and systems, including branch prediction in modern pipelined processors and value prediction for pre-fetching memory and files [13, 42], the search space considered by SpecInfer is significantly larger due to two reasons: (1) modern LLMs generally involve very large vocabularies, and (2) maximizing speculative performance requires predicting multiple future tokens (instead of just the next token). For example, all LLMs in the OPT model family consider 50,272 different possible tokens in their vocabulary, while SpecInfer can correctly predict the next 4 tokens on average. Achieving this goal requires considering a search space of $50272^4 \approx 6 \times 10^{18}$ different combinations of tokens.

SpecInfer leverages existing distilled, quantized, and/or pruned variants of an LLM, which we call small speculative models (SSMs), to guide speculation. A key challenging of using SSMs for speculative inference is that the alignment between an SSM and an LLM is inherently bounded by the model capacity gap, since an SSM is generally 100-1000× smaller than an LLM. Instead of using a single SSM for sequence-based speculation, SpecInfer maximizes speculative performance by simultaneously considering a variety of token sequences organized in a tree structure for a given input prompt. SpecInfer introduces an *expansion*- and a *merge*-based mechanism for constructing token trees by exploiting diversity within a single SSM and across multiple SSMs, respectively.

A second challenge SpecInfer must address is verifying the speculated tokens. Many LLM applications perform *stochastic decoding*, which samples the next token from a probability distribution instead of deterministically generating a token. To preserve an LLM's generative performance, SpecInfer must guarantee that its tree-based speculative inference and verification mechanism generates the next token by following the *exact same* probability distribution as incremental decoding. To achieve this goal, we propose *multi-step speculative sampling*, a new sampling approach for SSMs that guarantees equivalence while maximizing the number of



(a) Incremental decoding.

(b) Timeline Comparison.

Figure 1. Comparing the incremental decoding approach used by existing LLM serving systems, the sequence-based speculative inference approach, and the tree-based speculative inference approach used by SpecInfer.

speculated tokens that can be verified. To minimize the token tree verification cost, SpecInfer introduces a *tree-based parallel decoding* mechanism, *simultaneously* verifying all tokens of a token tree against the LLM's output in a *single* LLM decoding step.

By leveraging tree-based speculative inference and verification, SpecInfer accelerates both distributed LLM inference across multiple GPUs and offloading-based LLM inference on one GPU. Our evaluation shows that SpecInfer outperforms existing LLM serving systems by 1.5-2.8× for distributed LLM inference and by 2.6-3.5× for offloading-based LLM inference, while preserving the same generative accuracy.

To summarize, we make the following contributions:

- We present SpecInfer, a tree-based speculative inference and verification system for LLM serving.
- To maximize speculative performance, we propose a merge- and an expansion-based method to construct token trees by exploiting diversity within and across SSMs, respectively.
- To minimize verification cost, we introduce a tree-based parallel decoding mechanism to simultaneously verify all tokens of a token tree.
- We evaluate SpecInfer and show that it outperforms existing systems by up to 2.8× for distributed inference and by up to 3.5× for offloading-based inference.

2 SpecInfer's Overview

Figure 2 shows an overview of SpecInfer, which includes a *learning-based speculator* that takes as input a sequence of tokens, and produces a *speculated token tree*. The goal of the speculator is to predict the LLM's output by maximizing the overlap between the speculated token tree and the tokens generated by the LLM using incremental decoding (Alg. 1).

Algorithm 1 The incremental decoding algorithm used in existing LLM serving systems.

```
Input: A sequence of input tokens $\mathcal{I}$
Output: A sequence of generated tokens
$\mathcal{S} = \mathcal{I}$
while true do
    $t = \textsc{Decode}(\text{LLM}, \mathcal{S})$
    $\mathcal{S}$.append($t$)
    if $t = \langle\text{EOS}\rangle$ then
        Return $\mathcal{S}$
```

There are several ways to prepare SSMs for speculative inference. First, modern LLMs generally have many smaller architectures pre-trained together with the LLM using the same datasets. For example, in addition to the OPT-175B model with 175 billion parameters, the OPT model family also includes OPT-125M and OPT-350M, two variants with 125 million and 350 million parameters, which were pre-trained using the same datasets as OPT-175B [57]. These pre-trained small models can be directly used as SSMs. Second, to improve the coverage of speculated tokens from SSMs, SpecInfer takes an expansion-based and a merge-based speculation method as shown at the top of Figure 2. The speculated tokens are organized in a token tree structure.

SpecInfer's usage of an LLM is also different from that of existing LLM serving systems. Instead of using the LLM as an incremental decoder that predicts the next single token, SpecInfer uses the LLM as a token tree verifier that verifies a speculated token tree against the LLM's output. For each token, SpecInfer computes its activations by considering all of its ancestors in the token tree as its preceding tokens. For example, in Figure 2, the attention output of the token $t_{3,0}$ is calculated based on sequence $(t_0, t_{1,0}, t_{2,1}, t_{3,0})$, where t_0,

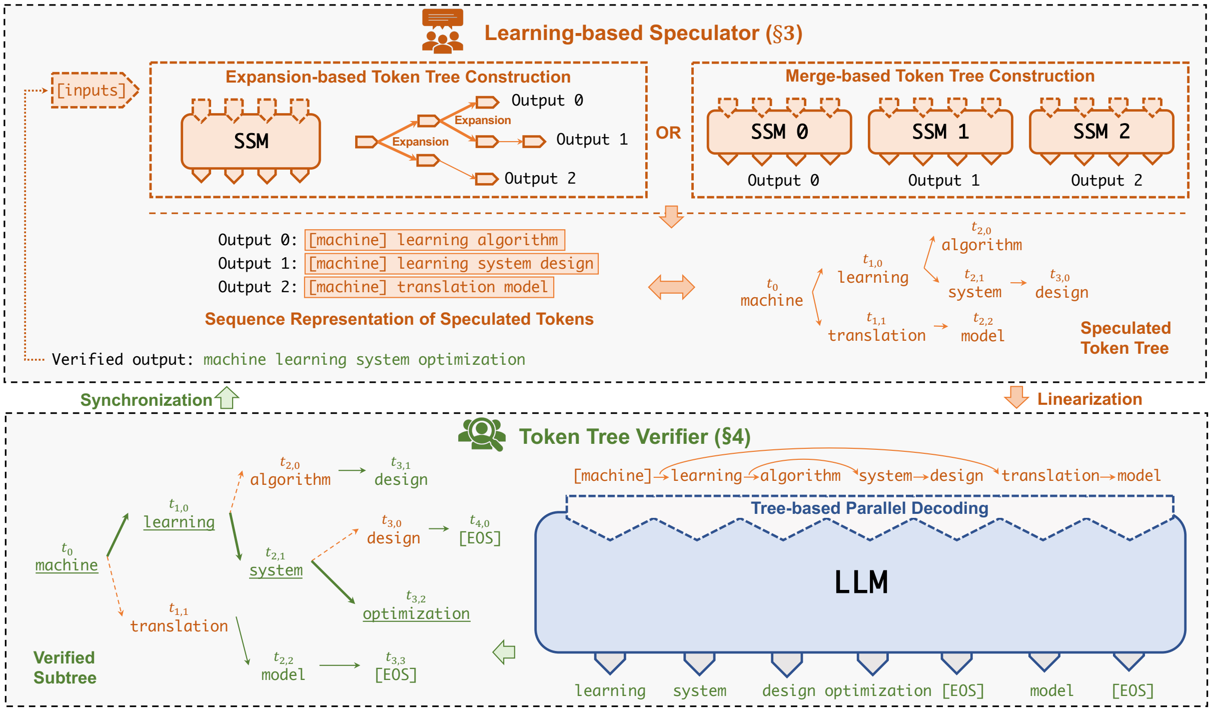


Figure 2. An overview of SpecInfer's tree-based speculative inference and verification mechanism.

$t_{1,0}$, and $t_{2,1}$ are $t_{3,0}$'s ancestors in the token tree. SpecInfer includes a novel tree-based parallel decoding mechanism to simultaneously verify *all* tokens of a token tree in a single LLM decoding step.

SpecInfer's speculative inference and token tree verification provide two key advantages over the incremental decoding approach of existing LLM inference systems.

Reduced memory accesses to LLM parameters. The performance of LLM inference is largely limited by accesses to GPU memory. In the existing incremental decoding approach, generating a single token requires accessing all parameters of an LLM. The problem is exacerbated for offloading-based LLM inference systems, which use limited computational resources such as a single commodity GPU to serve LLMs by utilizing CPU DRAM and persistent storage to save model parameters and loading these parameters to GPU's high bandwidth memory (HBM) for computation. Compared to the incremental decoding approach, SpecInfer significantly reduces accesses to LLM parameters whenever the overlap between a speculated token tree and the LLM's actual output is not empty. Reduced accesses to GPU device memory and reduced data transfers between GPU and CPU memory can also directly translate to decreased energy consumption, since accessing GPU HBM consumes two or three orders of magnitude more energy than floating point arithmetic operations.

Reduced end-to-end inference latency. Serving LLMs suffers from long end-to-end inference latency. For example, the GPT-3 architecture includes 175 billion parameters and requires many seconds to serve a request. In the existing incremental decoding approach, the computation for generating each token depends on the keys and values of all previously generated tokens, which introduces sequential dependencies between tokens and requires modern LLM serving systems to serialize the generation of different tokens for each request. In SpecInfer, LLMs are used as a verifier that takes a speculated token tree as an input and can simultaneously examine *all* tokens in the token tree by making a single verification pass over the LLM. This approach enables parallelization across different tokens in a single request and reduces the LLM's end-to-end inference latency.

3 Learning-based Speculator

Existing speculative decoding methods perform *sequence-based* speculation, where an SSM predicts a single sequence of tokens to be verified by an LLM. However, a key limitation of a single speculated sequence is that the probability of a successful alignment between the LLM and the speculated token sequence decays exponentially with the expected alignment length. This can be further exacerbated by the fact that the speculation only includes a single candidate token to verify per step, resulting in suboptimal speculative performance. On the other hand, by encouraging more diverse speculated candidates per step, the probability of a successful match per

Algorithm 2 The speculation and verification algorithm used by SpecInfer. Speculate takes the current token sequence $\mathcal{S}$ as an input and generates a speculated token tree $\mathcal{N}$. TreeParallelDecode generates a token $\mathcal{O}(u)$ for each node $u \in \mathcal{N}$. VerifyGreedy and VerifyStochastic examine $\mathcal{N}$ against $\mathcal{O}$ and produce a sequence of verified tokens $\mathcal{V}$ using greedy or stochastic sampling, respectively.

```
Input: A sequence of input tokens $\mathcal{I}$
Output: A sequence of generated tokens
$\mathcal{S} = \mathcal{I}$
while true do
    $\mathcal{N} = \textsc{Speculate}(\mathcal{S})$
    $\mathcal{O} = \textsc{TreeParallelDecode}(\text{LLM}, \mathcal{N})$
    if use greedy decoding then
        $\mathcal{V} = \textsc{VerifyGreedy}(\mathcal{O}, \mathcal{N})$
    else
        $\mathcal{V} = \textsc{VerifyStochastic}(\mathcal{O}, \mathcal{N})$
    for $t \in \mathcal{V}$ do
        $\mathcal{S}$.append($t$)
        if $t = \langle\text{EOS}\rangle$ then
            return $\mathcal{S}$

function VerifyGreedy($\mathcal{O}, \mathcal{N}$)
    $\mathcal{V} = \emptyset$, $u \leftarrow$ the root of token tree $\mathcal{N}$
    while $\exists v \in \mathcal{N}.p_v = u$ and $t_v = \mathcal{O}(u)$ do
        $\mathcal{V}$.append($t_v$)
        $u = v$
    $\mathcal{V}$.append($\mathcal{O}(u)$)
    return $\mathcal{V}$

function VerifyStochastic($\mathcal{O}, \mathcal{N}$)
    $\mathcal{V} = \emptyset$, $u \leftarrow$ the root of token tree $\mathcal{N}$
    while $u$ is a non-leaf node do
        $\mathcal{H} = \text{child}(u)$        ▷ The set of child nodes for $u$
        while $\mathcal{H}$ is not empty do
            $s \sim \text{rand}(\mathcal{H}), r \sim U(0,1), x_s = \mathcal{H}[s]$
            if $r \le P(x_s \mid u, \Theta_{LLM}) / P(x_s \mid u, \Theta_{SSM_s})$ then
                ▷ Token $x_s$ passes verification.
                $\mathcal{V}$.append($x_s$)
                $u = s$
                break
            else
                ▷ Normalize the residual $P(x \mid u, \Theta_{LLM})$
                $P(x \mid u, \Theta_{\text{LLM}}) := \text{norm}(\max(0, P(x \mid u, \Theta_{\text{LLM}}) - P(x \mid u, \Theta_{\text{SSM}_s})))$
                $\mathcal{H}$.pop($s$)
        if $\mathcal{H}$ is empty then
            break
    ▷ All SSMs fail verification; sample the next token
    $x_{\text{next}} \sim P(x \mid u, \Theta_{LLM})$
    $\mathcal{V}$.append($x_{\text{next}}$)
    return $\mathcal{V}$
```

Table 1. The success rate of verifying a token for LLaMA-7B using the top-k tokens derived from LLaMA-68M. The five prompt datasets are described in Section 6.1.

	Dataset	$k = 1$	$k = 2$	$k = 3$	$k = 4$	$k = 5$
Greedy decoding	Alpaca	68%	77%	81%	84%	85%
	CP	69%	79%	83%	86%	87%
	WebQA	62%	72%	77%	80%	82%
	CIP	70%	81%	85%	88%	89%
	PIQA	63%	75%	79%	83%	85%
Stochastic decoding	Alpaca	54%	81%	91%	95%	97%
	CP	56%	82%	92%	95%	97%
	WebQA	52%	80%	90%	94%	96%
	CIP	57%	84%	92%	95%	97%
	PIQA	55%	82%	91%	95%	97%

step (i.e., the token decoded by the LLM is in this candidate pool) can be greatly improved. To this end, SpecInfer aims to construct a tree of speculated candidates by exploiting diversity within a single SSM and across multiple SSMs. In particular, SpecInfer's *learning-based speculator* aggregates the predictions of one or multiple SSMs to maximize speculative performance while maintaining low memory overhead and inference latency. SpecInfer uses a *token tree* to organize the tokens produced by the speculator and introduces two methods for constructing token trees: *expansion*- and *merge-based* tree constructions.

Definition 3.1 (Token Tree). A token tree $\mathcal{N}$ is a tree structure, where each node $u \in \mathcal{N}$ is labeled with a token t_u, and p_u represents u's parent node in the token tree. For each node u, S_u represents a sequence of tokens identified by concatenating S_{p_u} and $\{t_u\}$[1].

Expansion-based token tree construction. One approach to creating a token tree involves deriving *multiple* tokens from an SSM within a single decoding step. This approach is motivated by an important observation that when an SSM misaligns with an LLM (i.e., the two models select different top-1 tokens), the token selected by the LLM is generally among the top-k tokens from the SSM for very small values of k. Table 1 shows the success rate of verifying a token using the top-k tokens derived from an SSM, where a verification is successful if the token selected by the LLM is among the top-k tokens from the SSM. Compared to only using the top-1 token from an SSM, using the top-5 tokens can increase the success rate from 70% to 89% for greedy decoding and from 57% to 97% for stochastic decoding.

Directly selecting the top-k tokens at each step leads to an exponential increase in the number of potential token sequences, which substantially elevates inference latency and

[1] For the root node r, S_r represents the token sequence $\{t_r\}$.

memory overhead. Consequently, we adopt a *static* strategy that expands the token tree following a preset *expansion configuration* represented as a vector of integers $\langle k_1, k_2, ..., k_m \rangle$, where m denotes the maximum number of speculative decoding steps, and k_i indicates the number of tokens to expand for each token in the i-th step. For example, Figure 3 illustrates the expansion configuration $\langle 2, 2, 1 \rangle$, leading to four token sequences. Our evaluation (see Section 6.4) shows that even a simple strategy can generate highly accurate speculative results. We acknowledge that *dynamically* expanding a token tree from an SSM is an opening research problem beyond the scope of this paper, which we leave as future work.

Merge-based token tree construction. In addition to using a single SSM, SpecInfer can also combine multiple SSMs to jointly predict an LLM's output. SpecInfer uses an unsupervised method to *collectively boost-tune* a pool of SSMs to align their outputs with that of the LLM by leveraging adaptive boosting [12]. SpecInfer uses SSMs to predict the next few tokens that an LLM will generate, and uses general text datasets (e.g., the OpenWebText corpus [15] in our evaluation) to adaptively align the aggregated output of multiple SSMs with the LLM in a fully unsupervised fashion. In particular, SpecInfer converts a text corpus into a collection of prompt samples and use the LLM to generate a token sequence for each prompt. SpecInfer first fine-tunes one SSM at a time to the fullest and marks all prompt samples where the SSM and LLM generate identical subsequent tokens. Next, SpecInfer filters all marked prompt samples and uses all remaining samples in the corpus to fine-tune the next SSM to the fullest.

By repeating this process for every SSM in the pool, SpecInfer obtains a diverse set of SSMs whose aggregated output largely overlaps with the LLM's output on the training corpus. All SSMs have identical inference latency, and therefore running all SSMs on different GPUs in parallel does not increase the latency of speculative inference compared to using a single SSM. In addition, SpecInfer uses data parallelism to serve SSMs across multiple GPUs, and therefore using multiple SSMs does not increase the memory overhead on each GPU. In the case where multiple SSMs are employed, the output of each SSM is considered as a token tree, and SpecInfer performs *token tree merge* to aggregate all speculated tokens in a single tree structure.

Definition 3.2 (Token Tree Merge)**.** $\mathcal{M}$ is the tree merge of m token trees $\{\mathcal{N}_i\}$ $(1 \le i \le m)$ if and only if $\forall 1 \le i \le m, \forall u \in \mathcal{N}_i, \exists v \in \mathcal{M}$ such that $S_v = S_u$ and vice versa.

Intuitively, each token tree represents a set of token sequences. Merging multiple token trees produces a new tree that includes all token sequences of the original trees. For example, Figure 3 shows the token tree derived by merging four sequences of tokens. Each token sequence is identified by a node in the merged token tree.

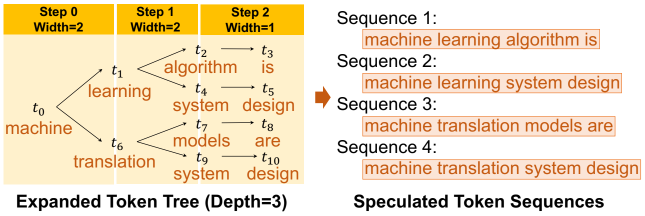


Figure 3. Illustration of token tree expansion.

Note that, in addition to boosting, there are several other ensemble learning methods (e.g., voting, bagging, and stacking) [14] that can be used to combine the outputs from multiple SSMs, and we leave the exploration as future work.

4 Token Tree Verifier

This section introduces SpecInfer's *token tree verifier*, which takes as input a token tree generated by the speculator and verifies the correctness of its tokens against an LLM's output. A key idea behind the design of SpecInfer is *simultaneously* verifying all sequences of a token tree against the original LLM's output by making a *single* pass over the LLM's parameters. This functionality allows SpecInfer to opportunistically decode *multiple* tokens (instead of a single token in incremental decoding), resulting in reduced memory accesses to the LLM's parameters. A challenge SpecInfer must address in token tree verification is efficiently computing the attention scores for *all* sequences of a token tree. To this end, we introduce *tree attention*, which generalizes the attention mechanism [48] from sequence to tree structure. In addition, we develop a *tree-based parallel decoding* mechanism that can decode *all* tokens in a token tree in parallel.

§4.1 and §4.2 describe tree attention and tree-based parallel decoding. §4.3 introduces the mechanism to verify a token tree against the LLM's output.

4.1 Tree Attention

Transformer-based language models use the attention mechanism to reason about sequential information [48]. LLMs generally use decoder-only, multi-head self-attention, which takes a single input tensor X and computes an output tensor O via scaled multiplicative formulations as follows.

$$Q_i = X \times W_i^Q, \qquad K_i = X \times W_i^K, \tag{1}$$

$$V_i = X \times W_i^V, \qquad A_i = \frac{(Q_i \times K_i^T)}{\sqrt{d}}, \tag{2}$$

$$H_i = \text{softmax}\big(\text{mask}(A_i)\big)V_i, \quad O = (H_1, ..., H_h)W^O \tag{3}$$

where Q_i, K_i, and V_i denote the query, key, and value tensors of the i-th attention head $(1 \le i \le h)$, W_i^Q, W_i^K, and W_i^V are the corresponding weight matrices. A_i is an $l \times l$ matrix that represents the attention scores between different tokens in the input sequence, where l is the sequence length. To preserve causality when generating tokens (i.e., a token in the sequence should not affect the hidden states of any preceding tokens), the following causal mask function is applied:

$$\text{mask}(A)_{jk} = \begin{cases} A_{jk} & j \geq k \\ -\infty & j < k \end{cases}. \qquad (4)$$

Intuitively, when computing the attention output of the j-th token in the sequence, all subsequent tokens should have an attention score of $-\infty$ to indicate that the subsequent tokens will not affect the attention output of the j-th token[2]. In Equation 3, H_i represents the output of the i-th attention head, and W_O is a weight matrix used for computing the final output of the attention layer.

Note that the attention mechanism described above applies only to a sequence of tokens. We generalize the attention mechanism to arbitrary tree structures.

Definition 4.1 (Tree Attention)**.** For a token tree $\mathcal{N}$ and an arbitrary node $u \in \mathcal{N}$, its tree attention is defined as the output of computing the original Transformer-based sequence attention on S_u (i.e., the token sequence represented by u):

$$\textsc{TreeAttention}(u) = \textsc{Attention}(S_u) \forall u \in \mathcal{N} \qquad (5)$$

For a given set of token sequences, since each sequence S is covered by a node of the merged token tree, performing tree attention on the token tree allows SpecInfer to obtain the attention output for *all* token sequences.

4.2 Tree-based Parallel Decoding

This section describes SpecInfer's *tree-based parallel decoding* mechanism for computing tree attention for *all* tokens in a token tree *in parallel*. A key challenge SpecInfer must address in computing tree attention is managing *key-value cache*. In particular, the attention mechanism of Transformer [48] requires accessing the keys and values of all preceding tokens to compute the attention output of each new token, as shown in Equation 3. To avoid recomputing these keys and values, today's LLM inference systems generally cache the keys and values of all tokens for reuse in future iterations, since the causal relation guarantees that a token's key and value remain unchanged in subsequent iterations (i.e., $\text{mask(A)}_{\text{jk}} = -\infty$ for any $j < k$). However, when computing tree attention, different sequences in a token tree may include conflicting key-value caches. For example, for the speculated token tree in Figure 4, two token sequences (t_2, t_3, t_4, t_5) and (t_2, t_3, t_8, t_9) have different keys and values for the third and fourth positions.

A straightforward approach to supporting key-value cache is employing the sequence-based decoding of existing LLM inference systems and using a different key-value cache for each sequence of a token tree, as shown on the left of Figure 4. However, this approach is computationally very expensive and involves redundant computation, since two token sequences sharing a common prefix have the same attention outputs for the common prefix due to the causal mask in Equation 3. In addition, launching one kernel for each token sequence introduces additional kernel launch overhead.

SpecInfer introduces two key techniques to realize tree-based parallel decoding.

Depth-first search to update key-value cache. Instead of caching the keys and values for individual token sequences of a token tree, SpecInfer reuses the same key-value cache across all token sequences by leveraging a *depth-first search* mechanism to traverse the token tree, as shown in Figure 4, where SpecInfer visits $t_2, t_3, ..., t_9$ by following a depth-first order to traverse the token tree and update the shared key-value cache. This approach allows SpecInfer to maintain the correct keys and values for all preceding tokens when computing the attention output of a new token.

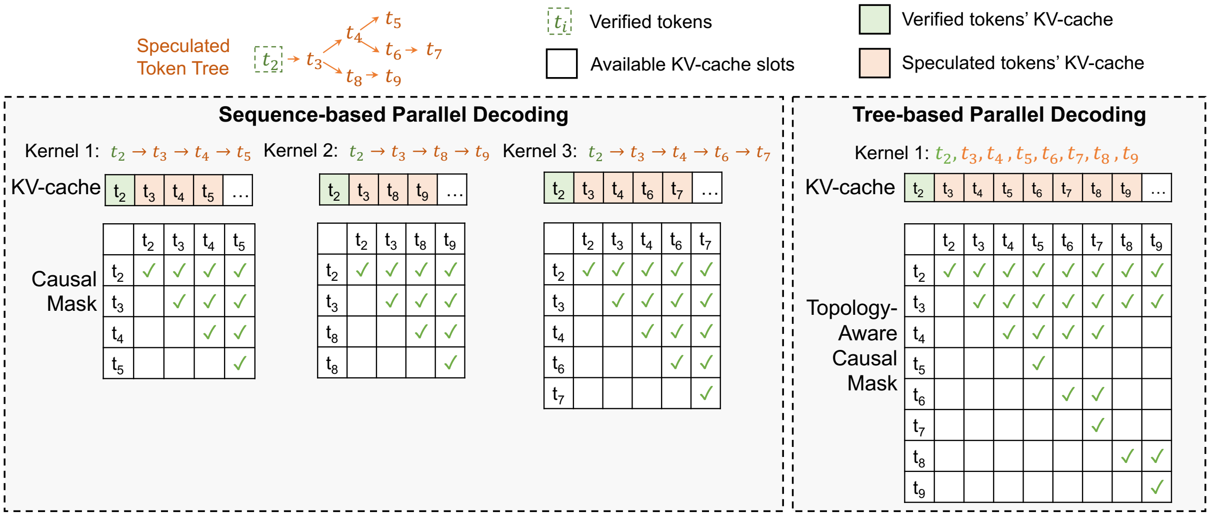
Topology-aware causal mask. A straightforward approach to computing tree attention is calculating the tree attention output for individual tokens by following the depth-first order described earlier. However, this approach would result in high GPU kernel launch overhead since each kernel only computes tree attention for one token sequence. In addition, executing these kernels in parallel requires additional GPU memory to store their key-value caches separately due to cache conflict. A key challenge that prevents SpecInfer from batching multiple tokens is that the attention computation for different tokens requires different key-value caches and therefore cannot be processed in parallel.

We introduce *topology-aware casual mask* to fuse tree attention computation of all tokens in a single kernel. To batch attention computation, SpecInfer uses a tree topology instead of the original sequence topology to store the keys and values of all tokens in a token tree in the key-value cache. For example, to compute tree attention for the speculated token tree shown in Figure 4, SpecInfer takes both verified tokens (i.e., t_2) and all speculated tokens (i.e., $t_3, t_4, ..., t_9$) as inputs. This approach allows SpecInfer to fuse the attention computation into a single kernel but also results in attention scores that violate the causal dependency (e.g., t_7's attention computation uses all previous tokens, including t_5 which is not in t_7's token sequence). To fix the attention scores for these pairs, SpecInfer updates the causal mask based on the token tree's topology. This approach computes the exact same attention output as incremental decoding, while resulting in much fewer kernel launches compared to sequence-based decoding.

4.3 Token Verification

For a given speculated token tree $\mathcal{N}$, SpecInfer uses tree-based parallel decoding (see Section 4.2) to compute its tree attention and generate an output tensor $\mathcal{O}$ that includes a

[2] Note that we use $-\infty$ (instead of 0) to guarantee that the softmax's output is 0 for these positions.


Speculated Token Tree
$t_2 \to t_3$; $t_3 \to t_4$; $t_4 \to t_5$; $t_4 \to t_6 \to t_7$; $t_3 \to t_8 \to t_9$
t_i Verified tokens
Available KV-cache slots
Verified tokens' KV-cache
Speculated tokens' KV-cache
Sequence-based Parallel Decoding
Kernel 1: $t_2 \to t_3 \to t_4 \to t_5$
Kernel 2: $t_2 \to t_3 \to t_8 \to t_9$
Kernel 3: $t_2 \to t_3 \to t_4 \to t_6 \to t_7$
KV-cache
Causal Mask
Tree-based Parallel Decoding
Kernel 1: $t_2, t_3, t_4, t_5, t_6, t_7, t_8, t_9$
KV-cache
Topology-Aware Causal Mask

KV-cache					
Kernel 1	t_2	t_3	t_4	t_5	…
Kernel 2	t_2	t_3	t_8	t_9	…

	t_2	t_3	t_4	t_5	t_6	t_7
Kernel 3 KV-cache	t_2	t_3	t_4	t_6	t_7	…

	t_2	t_3	t_4	t_5
t_2	✓	✓	✓	✓
t_3		✓	✓	✓
t_4			✓	✓
t_5				✓

	t_2	t_3	t_8	t_9
t_2	✓	✓	✓	✓
t_3		✓	✓	✓
t_8			✓	✓
t_8				✓

	t_2	t_3	t_4	t_6	t_7
t_2	✓	✓	✓	✓	✓
t_3		✓	✓	✓	✓
t_4			✓	✓	✓
t_6				✓	✓
t_7					✓

KV-cache	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	…

	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9
t_2	✓	✓	✓	✓	✓	✓	✓	✓
t_3		✓	✓	✓	✓	✓	✓	✓
t_4			✓	✓	✓	✓		
t_5				✓				
t_6					✓	✓		
t_7						✓		
t_8							✓	✓
t_9								✓

Figure 4. Comparing SpecInfer's tree-based parallel decoding with existing sequence-based decoding.

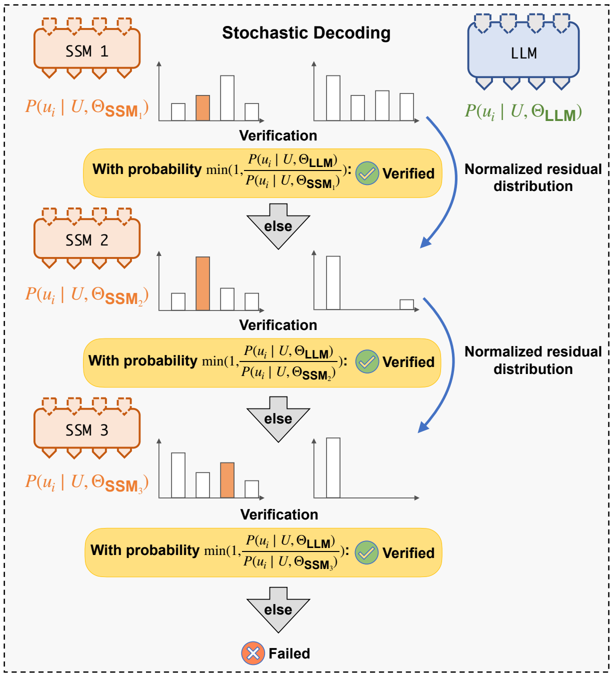

Stochastic Decoding
SSM 1
$P(u_i \mid U, \Theta_{\text{SSM}_1})$
LLM
$P(u_i \mid U, \Theta_{\text{LLM}})$
Verification
With probability $\min(1, \frac{P(u_i \mid U, \Theta_{\text{LLM}})}{P(u_i \mid U, \Theta_{\text{SSM}_1})})$: Verified
Normalized residual distribution
else
SSM 2
$P(u_i \mid U, \Theta_{\text{SSM}_2})$
Verification
With probability $\min(1, \frac{P(u_i \mid U, \Theta_{\text{LLM}})}{P(u_i \mid U, \Theta_{\text{SSM}_2})})$: Verified
Normalized residual distribution
else
SSM 3
$P(u_i \mid U, \Theta_{\text{SSM}_3})$
Verification
With probability $\min(1, \frac{P(u_i \mid U, \Theta_{\text{LLM}})}{P(u_i \mid U, \Theta_{\text{SSM}_3})})$: Verified
else
Failed

Figure 5. Illustrating the multi-step speculative sampling mechanism for verifying LLMs with stochastic sampling.

token for each node $u \in \mathcal{N}$. Next, SpecInfer's *token tree verifier* examines the correctness of speculated tokens against the LLM. SpecInfer supports both greedy and stochastic sampling as shown in Algorithm 2.

Greedy decoding. Many LLM applications generate tokens using *greedy decoding*, which greedily selects the token with the highest likelihood in each decoding step. The VerifyGreedy function in Algorithm 2 shows how SpecInfer verifies a speculated token tree $\mathcal{N}$ with greedy decoding. SpecInfer starts from the root of $\mathcal{N}$ and iteratively examines a node's speculated results against the LLM's original output. For a node $u \in \mathcal{N}$, SpecInfer successfully speculates its next token if u includes a child node v (i.e., $p_v = u$) whose token matches the LLM's output (i.e., $t_v = \mathcal{O}(u)$). In this case, SpecInfer finishes its verification for node u and moves on to examine its child v. When the node u does not include a child that contains the LLM's output, SpecInfer adds $\mathcal{O}(u)$ as a verified node in $\mathcal{N}$ and terminates the verification process. Finally, all verified nodes are appended to the current generated token sequence $\mathcal{V}$. Token tree verification allows SpecInfer to opportunistically decode multiple tokens (instead of a single token in the incremental decoding approach), while preserving the same generative performance as incremental decoding.

Stochastic decoding. To improve the diversity of generated tokens, many LLM applications perform *stochastic decoding*, which samples a token from a probability distribution $P(u_i|u_0, ..., u_{i-1}; \Theta_{LLM})$, where $U = u_0, ..., u_{i-1}$ are previously generated tokens, u_i is the next token to generate, and Θ_{LLM} represents a parameterized LLM.

To verify a speculated token tree with stochastic decoding, we introduce a *multi-step speculative sampling* (MSS) algorithm to conduct verification, whose pseudocode code is shown in the VerifyStochastic function in Algorithm 2 and illustrated in Figure 5. Our method provably preserves an LLM's generative performance as incremental decoding while optimizing the number of speculated tokens that can be verified. Theorem 4.2 proves its correctness.

Theorem 4.2. *For a given LLM and m SSMs (i.e., SSM_1,...,SSM_m, let $P(u_i|U; \Theta_{LLM})$ be the probability distribution of sampling* *a token using stochastic decoding, where $U = u_0, ..., u_{i-1}$ are previously generated tokens, u_i is the next token to generate, Θ_{LLM} represents the parameterized LLM.*

Let $P_{\text{SpecInfer}}(u_i|U; \Theta_{LLM}, \{\Theta_{SSM_j}\})$ be the probability distribution of sampling token u_i using SpecInfer's multi-step speculative sampling (see the VerifyStochastic *function in Algorithm 2), where Θ_{SSM_j} is the j-th parameterized SSM. Then $\forall U, u_i, \Theta_{LLM}, \Theta_{SSM_j}$ we have*

$$P(u_i \mid U; \Theta_{LLM}) = P_{\text{SpecInfer}}(u_i \mid U; \Theta_{LLM}, \{\Theta_{SSM_j}\}) \tag{6}$$

A proof of this theorem is presented in [28].

We acknowledge that a more straightforward approach to preserving the probability distribution of stochastic decoding is directly sampling the next token $x \sim P(u_i \mid U; \Theta_{LLM})$ and examining whether x is a child node of u_{i-1} in the speculated token tree. We call this approach *naive sampling* (NS) and show that SpecInfer's multi-step speculative sampling has a uniformly lower rejection probability than naive sampling.

Theorem 4.3. *Let $P\left(reject \mid MSS, U, \Theta_{LLM}, \{\Theta_{SSM_j}\}\right)$ denote the probability of rejecting speculation following multi-step speculative sampling with abbreviation $P(reject \mid MSS)$, and $P\left(reject \mid NS, U, \Theta_{LLM}, \{\Theta_{SSM_j}\}\right)$ the probability of rejecting speculation following Naive Sampling (NS) with abbreviation $P(reject \mid NS)$. Then $\forall U, \Theta_{LLM}, \{\Theta_{SSM_j}\}$, we have*

$$P(reject \mid MSS) \leq P(reject \mid NS)$$

We present a proof of Theorem 4.3 in [28].

Note that prior work has introduced single-step speculative sampling for sequence-based speculative inference [5, 25]. Different from these approaches, SpecInfer leverages token trees for improving speculative performance, which requires a different verification algorithm. As a result, SpecInfer performs multi-step verification (see VerifyStochastic in Algorithm 2) across all branches of a token to maximize the success rate while preserving equivalence as incremental decoding. The proposed MSS algorithm not only works for merge-based method with multiple SSMs, but also supports expansion-based method with one SSM and top-k sampling.

5 System Design and Implementation

This section describes the design and implementation of SpecInfer's distributed runtime system (§5.1 and §5.2), analyzes the computation and memory overheads of speculation and verification (§5.3), and introduces potential LLM applications that can benefit from SpecInfer's techniques (§5.4).

5.1 SpecInfer's Runtime Design

Figure 6 shows the workflow for one iteration of speculative inference and verification. SpecInfer's *request manager* receives LLM serving requests and schedules these requests for serving by adapting the *iteration-level scheduling* policy from Orca [55]. Specifically, SpecInfer iteratively selects requests from a pool of pending requests and performs one iteration of speculative inference and token tree verification for the selected requests. Since SSMs are small and can fit in one GPU, SpecInfer equally distributes GPUs across SSMs and serves these SSMs using data parallelism. For example, Figure 6 shows how SpecInfer serves two SSMs and four requests (i.e., r_1, r_2, r_3, and r_4) on four GPUs. The SSM-generated tokens are sent back to the request manager, which produces a speculated token tree for each request using the tree merge algorithm introduced in §4.

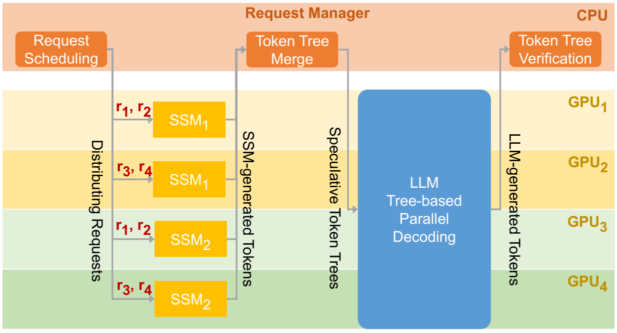


Figure 6. SpecInfer's workflow for one iteration of speculative inference and verification. SpecInfer uses data parallelism to serve SSMs, and combine tensor model parallelism and pipeline model parallelism for serving an LLM.

SpecInfer serves an LLM using the hybrid parallelization strategy introduced in Megatron-LM [41], which uses tensor model parallelism for parallelizing each Transformer layer across GPUs within a node, and uses pipeline model parallelism for partitioning Transformer layers across nodes. All GPUs perform the tree-based parallel decoding (see §4.2) to compute tree attention scores and send the LLM-generated tokens back to the request manager, which finally verifies the speculated tokens against the LLM's output (see §4.3).

Note that the overhead introduced by the request manager (i.e., request scheduling, token tree merge, and verification) is negligible compared to the execution time of LLM inference. In addition, SpecInfer's request manager and GPU workers only communicate tokens and do not transfer the vector representations of these tokens, which again introduces negligible communication overheads.

Continuous batching. SpecInfer uses *continuous batching* introduced in Orca [55] to serve multiple LLM inference requests in parallel. Specifically, SpecInfer schedules LLM execution at the granularity of iterations instead of requests. After each LLM decoding iteration, SpecInfer checks each request's status and sends the generated results of all finished requests to the client. This design also allows SpecInfer to start processing newly arrived requests without waiting for all current requests to complete.

5.2 SpecInfer's Implementation

SpecInfer was implemented on top of FlexFlow [21, 47], a distributed multi-GPU runtime for DNN computation. FlexFlow exposes an API that allows users to define a DNN model in terms of its layers. It is compatible with PyTorch's model definition due to the alignment of underlying operators. For example, the open-source LLMs from HuggingFace [19] can be directly imported into SpecInfer for serving without modification. Users can also provide a parallelization strategy, specifying the degree of data, model, and pipeline parallelism for each layer. A DNN is represented as a computational graph where each node is a region of memory, and each edge is an operation on one or more regions. Operations can be represented using three levels of abstraction: layers, operators, and tasks. The FlexFlow compiler transforms the computational graph from the highest abstractions (i.e., layers) to the lowest (i.e., tasks). Tasks are also the unit of parallelization; they are non-preemptible, and are executed asynchronously.

CUDA kernel optimizations. Directly launching cuBLAS and cuDNN kernels for calculating attention results in high kernel launch overhead and does not leverage the shared memory available on modern GPUs. To address this inefficiency, SpecInfer uses a customized kernel built on top of FasterTransformer [32] for computing attention. Within this kernel, each thread block computes a single head for a single request. The process begins with loading the query tensor into GPU shared memory accessible by all threads within a thread block. Each thread then performs a segment of the query/key product and broadcasts the result to other threads for computing the max query/key product and exponential sum. To support tree-based parallel decoding, SpecInfer computes all tokens within a tree in parallel and leverages the topology-aware causal mask to preserve casuality.

5.3 Overhead of Speculation and Verification

SpecInfer accelerates generative LLM inference at the cost of additional memory and computation overheads. This section analyzes these overheads and shows that they are generally one or two orders of magnitude smaller than the memory and computation cost of executing LLM inference.

Memory overhead. The memory overhead of SpecInfer's speculation-verification approach comes from two aspects. First, in addition to serving an LLM, SpecInfer also needs to allocate memory for saving the parameters of one or multiple SSMs, which collectively speculate the LLM's output. Our evaluation shows that SpecInfer can achieve significant performance improvement by using SSMs 100-1000× smaller than the LLM. As a result, hosting each SSM increases the overall memory requirement by less than 1%. A second source of memory overhead comes from the token tree verification engine, which verifies an entire token tree instead of decoding a single token. Therefore, additional memory is needed for caching the keys and values, and storing the attention scores for all tokens. Due to the necessity for supporting very long sequence length in today's LLM serving, we observe that the memory overhead associated with token tree is negligible compared to key-value cache.

Computation overhead. Similarly, the computation overhead introduced by speculation and verification also comes from two aspects. First, SpecInfer needs to run SSMs in the incremental-decoding mode to generate candidate tokens. When multiple SSMs are employed, SpecInfer processes these SSMs in parallel across GPUs to minimize speculation latency. Second, SpecInfer verifies a token tree by computing the attention outputs for an entire token tree, most of which do not match the LLM's output and therefore are unnecessary in the incremental-decoding inference. However, the key-value cache mechanism of existing LLM inference systems prevents them from serving a large number of requests in parallel, resulting in under-utilized computation resources on GPUs when serving LLMs in incremental decoding. SpecInfer's token tree verification leverages these under-utilized resources and therefore introduces negligible runtime overhead compared to incremental decoding.

5.4 Applications

Our speculative inference and token tree verification techniques can be directly applied to a variety of LLM applications. We identify two practical scenarios where LLM inference can significantly benefit from our techniques.

Distributed LLM inference. The memory requirements of modern LLMs exceed the capacity of a single compute node with one or multiple GPUs, and the current approach to addressing the high memory requirement is distributing the LLM's parameters across multiple GPUs [29]. For example, serving a single inference pipeline for GPT-3 with 175 billion parameters requires more than 16 NVIDIA A100-40GB GPUs to store the model parameters in single-precision floating points. Distributed LLM inference is largely limited by the latency to transfer intermediate activations between GPUs for each LLM decoding step. While SpecInfer's approach does not directly reduce the amount of inter-GPU communications, its verification mechanism can increase the communication granularity and reduce the number of decoding steps.

Offloading-based LLM inference. Another practical scenario that can benefit from SpecInfer's techniques is offloading-based LLM inference, which leverages CPU DRAM to store an LLM's parameters and loads a subset of these parameters to GPUs for computation in a pipeline fashion [40]. By opportunistically verifying multiple tokens, SpecInfer can reduce the number of LLM decoding steps and the overall communication between CPU DRAM and GPU HBM.

6 Evaluation

6.1 Experimental Setup

LLMs. To compare the runtime performance of SpecInfer with existing LLM serving systems, we evaluate these systems using two publicly available LLM families: OPT [57] and LLaMA [46]. More specifically, we select LLaMA-7B, OPT-13B, OPT-30B, and LLaMA-65B as the LLMs, and LLaMA-68M and OPT-125M as the SSMs. The pre-trained model parameters for the LLMs and SSMs were obtained from their HuggingFace repositories [19], and we describe how SpecInfer collectively boost-tunes multiple SSMs in [28].

Datasets. We evaluate SpecInfer on five datasets: Chatbot Instruction Prompts (CIP) [34], ChatGPT Prompts (CP) [30], WebQA [1], Alpaca [36, 45], and PIQA [2]. We only use the prompts/questions from these datasets to form our input prompts to simulate real-world conversation traces.

Platform. The experiments were conducted on two AWS `g5.12xlarge` instances, each of which is equipped with four NVIDIA A10 24GB GPUs, 48 CPU cores, and 192 GB DRAM. Nodes are connected by 100 Gbps Ethernet.

Our experiments use the expansion-based method (see Section 3) for constructing token trees and use the expansion configuration $\langle 1, 1, 3, 1, 1, 1, 1, 1 \rangle$, which provides good results for our benchmarks. We analyze the impact of expansion configurations in §6.4, evaluate tree-based parallel decoding and multi-step speculative sampling in §6.5 and §6.6, and finally compares the expansion- and merge-based tree construction methods in [28].

6.2 Distributed LLM Inference

We compare the end-to-end distributed LLM inference performance among SpecInfer, vLLM [24], HuggingFace Text Generation Inference (TGI) [18], and FasterTransformer [32] on LLaMA-7B, OPT-30B, and LLaMA-65B. For LLaMA-7B and OPT-30B, all systems serve the two LLMs in half-precision floating points across one and four A10 GPUs using tensor model parallelism. LLaMA-65B do not fit on four GPUs on a single node, therefore both FasterTransformer and SpecInfer serve it on eight A10 GPUs on two nodes by combining tensor model parallelism within each node and pipeline model parallelism across nodes. vLLM and HuggingFace TGI do not support pipeline model parallelism and cannot serve an LLM on multiple nodes.

To rule out potential effects of our system implementation, we also evaluate SpecInfer with two additional configurations. First, SpecInfer with *incremental decoding* evaluates the runtime performance of our implementation when the speculator generates empty token trees, and the verifier verifies exactly one token in each decoding step. Second, SpecInfer with *sequence-based speculative inference* serves as a reference for existing speculative inference system and is enabled by using a single pre-trained SSM and sequence-based decoding.

We use prompts from the five datasets described in §6.1. For each prompt, we let all systems generate up to 128 new tokens and report the average per-token latency in Figure 7. Note that SpecInfer may generate more than 128 new tokens since the verifier can verify multiple tokens in each iteration. In this case, we truncate SpecInfer's output to 128 tokens. SpecInfer with incremental decoding achieves on-par performance as existing systems. This is because all systems use the same strategies to parallelize LLM inference across GPUs and use the same kernel libraries (i.e., cuDNN, cuBLAS, and cuTLASS) to execute inference computation on GPUs. With tree-based speculative inference and verification, SpecInfer outperforms incremental decoding systems by 1.5-2.5× for single-node, multi-GPU inference and by 2.4-2.8× for multi-node, multi-GPU inference, while generating the exact same sequence of tokens as incremental decoding for all prompts. The speedup comes from leveraging spare GPU resources to perform tree-based parallel decoding while maintaining the same per-iteration latency as incremental decoding.

Compared to sequence-based speculative inference, SpecInfer's tree-based approach further reduces LLM serving latency by 1.2-1.5×. The improvement is achieved by (1) leveraging token trees to optimize speculative performance, (2) using tree-based parallel decoding to verify an entire token tree in parallel, and (3) performing multi-step speculative sampling to improve verification performance. We further evaluates these aspects in §6.4, §6.5, and §6.6.

Note that SpecInfer's performance improvement over existing systems reduces as the batch size (i.e., number of concurrent requests) increases. This is because SpecInfer leverages spare GPU resources to perform tree-based parallel decoding while maintaining the same per-iteration latency as incremental decoding. A larger batch size introduces more parallelizable computation for incremental decoding, and thus less spare GPU resources that can be leveraged by SpecInfer. On the flip side, larger batch sizes also increase the end-to-end latency of each request, as shown in Figure 7. Overall, SpecInfer is most beneficial for *low-latency* LLM inference.

6.3 Offloading-based LLM Inference

Another important application of SpecInfer is offloading-based LLM inference, where the system offloads an LLM's parameters to CPU DRAM and loads a subset of these parameters to GPUs for inference computation in a pipeline fashion. We compare the end-to-end offloading-based LLM inference performance between SpecInfer and FlexGen [39] using a single 24GB A10 GPU and two LLMs (i.e., OPT-13B and OPT-30B), both of which exceed the memory capacity of an A10 GPU and requires offloading for serving. Both SpecInfer and FlexGen retain all model parameters on CPU DRAM. During computation, the demand weights are loaded from the CPU

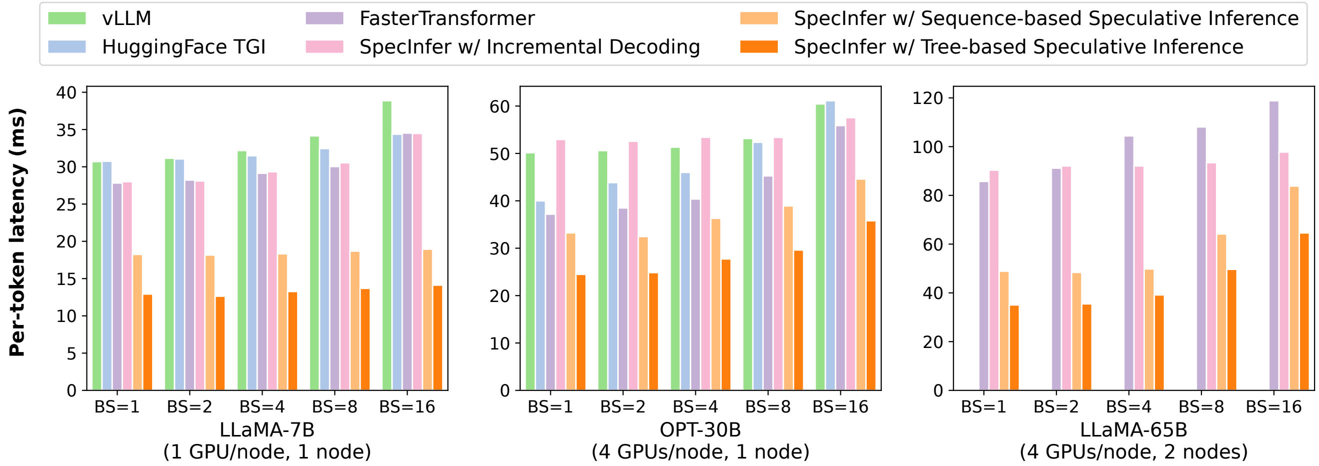


Figure 7. Comparing the end-to-end inference latency of SpecInfer with existing systems. Numbers in parenthesis show the number of GPUs and compute node used to serve each LLM. All systems parallelize LLM inference by combining tensor model parallelism (within a node) and pipeline parallelism (across nodes).

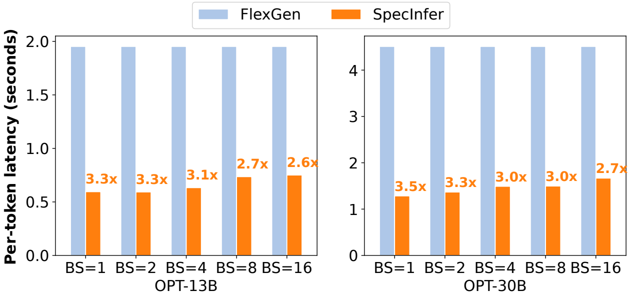


Figure 8. Comparing the end-to-end offloading-based inference latency of FlexGen and SpecInfer. Both FlexGen and SpecInfer perform model offloading to serve OPT-13B and OPT-30B models on a single 24GB A10 GPU.

to the GPU. Figure 8 shows the results. Compared to FlexGen, SpecInfer reduces the per-token latency by 2.6-3.5×. Since offloading-based LLM inference is mostly bottlenecked by the communication between CPU DRAM and GPU HBM for loading an LLM's parameters, SpecInfer's improvement over existing systems is achieved by opportunistically verifying multiple tokens, which in turn reduces the number of LLM decoding steps and data transfers between CPU and GPU.

6.4 Token Tree Construction

This section evaluates the expansion-based token tree construction mechanism. We first study how token tree width affects SpecInfer's speculative performance. In this experiment, we use LLaMA-7B and LLaMA-68M as the LLM and SSM, and use the expansion configuration $\langle 1, 1, k, 1, 1, 1, 1 \rangle$ (i.e., expanding at the third token), where k is the token tree width. Figure 9 shows the cumulative distribution function (CDF) of the average number of verified tokens per decoding

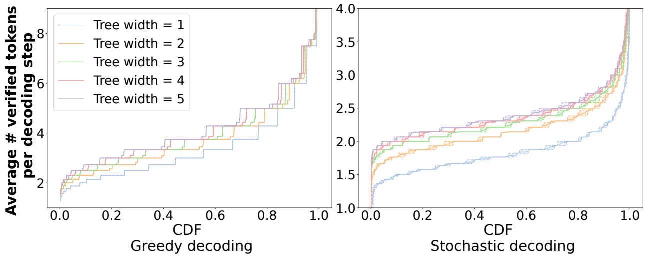


Figure 9. Comparing speculative performance of SpecInfer with different token tree structures.

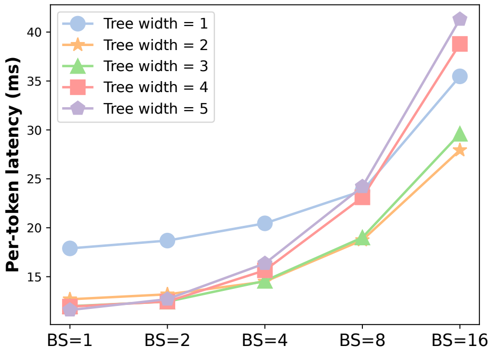


Figure 10. SpecInfer's end-to-end inference latency with different tree widths. We use LLaMA-7B and LLaMA-68M as the LLM and SSM.

step for all prompts in the Alpaca dataset [45]. Compared to sequence-based speculation (i.e., tree width = 1), leveraging token trees can reduce LLM decoding steps by 1.2-1.5× for greedy decoding and by 1.3-1.4× for stochastic decoding.

A larger token tree width reduces the LLM decoding steps to process a request at the cost of increased verification

Table 2. Average number of tokens verified by SpecInfer in a decoding step. We use LLaMA-7B and LLaMA-68M as the LLM and SSM, and use different tree widths for constructing a token tree. The speculation length is 8.

		Token tree width				
	Dataset	1	2	3	4	5
Greedy decoding	Alpaca	2.95	3.07	3.21	3.33	**3.43**
	CP	2.58	3.24	3.46	3.59	**3.69**
	WebQA	2.27	2.69	2.86	2.98	**3.07**
	CIP	2.73	3.40	3.62	3.79	**3.91**
	PIQA	2.18	2.80	2.97	3.10	**3.21**
Stochastic decoding	Alpaca	1.79	2.11	2.26	2.32	**2.38**
	CP	1.69	1.99	2.15	2.23	**2.28**
	WebQA	1.64	1.93	2.08	2.15	**2.21**
	CIP	1.72	2.05	2.19	2.28	**2.29**
	PIQA	1.67	1.93	2.08	2.15	**2.21**

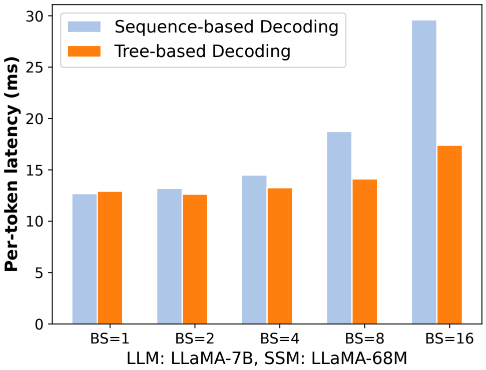


Figure 11. Comparing SpecInfer's tree-based parallel decoding with the sequence-based decoding mechanism employed by existing LLM inference systems.

overhead, since SpecInfer must verify more tokens. Figure 10 compares the end-to-end inference latency of SpecInfer using different tree widths. For small batch sizes (i.e., BS = 1 and 2), using a large tree width can consistently reduce per-token latency, since SpecInfer can leverage sparse GPU resources to verify more tokens in parallel while maintaining the same per-iteration latency. For large batch sizes (i.e., BS ≥ 4), using a large tree width increases the latency to verify a token tree due to less sparse GPU resources that can be leveraged by SpecInfer, and a tree width of 2 or 3 achieves the best performance by striking a perfect balance between speculative performance and verification latency.

6.5 Tree-based Parallel Decoding

We now evaluate the effectiveness of SpecInfer's tree-based parallel decoding mechanism, which decodes all tokens of a token tree in parallel. As a comparison, all existing LLM inference systems use sequence-based decoding, which requires decomposing a token tree into multiple sequences of tokens and processing these sequences using separate resources due to potential key-value cache conflicts (see §4.2). As shown in Figure 11, SpecInfer's tree-based parallel decoding achieves on-par performance as existing sequence-based decoding mechanism for small batch sizes and outperforms it by up to 1.8× for large batch sizes. The improvement is realized by (1) eliminating redundant attention computation for sequences with a shared prefix, and (2) fusing tree attention of all tokens in a single kernel through the topology-aware casual mask (see §4.2).

Table 3. Average number of tokens verified by SpecInfer in a stochastic decoding step with different sampling algorithms. We use LLaMA-7B and LLaMA-68M as the LLM and SSM. Each token tree has a width of 5 and a depth of 8.

	Naive Sampling	Multi-Step Spec. Sampling	Improvement
Alpaca	1.87	2.38	1.27×
CP	1.80	2.28	1.26×
WebQA	1.73	2.21	1.28×
CIP	1.79	2.29	1.28×
PIQA	1.73	2.21	1.28×

6.6 Multi-Step Speculative Sampling

This section evaluates how our multi-step speculative sampling (MSS) and the VerifyStochastic algorithm improves the speculative performance of SpecInfer when performing stochastic decoding. We use naive sampling as a baseline where SpecInfer directly samples the next token from the LLM and examines whether the sampled token is included in the speculated token tree (see §4.3). Since different sampling algorithms involve identical speculation and verification overheads, we focus on the average number of tokens that can be verified in each stochastic decoding step in this experiment. Table 3 shows the results. Compared to naive sampling, MSS can consistently improve the number of verified tokens by 1.2-1.3× on average across a variety of prompt datasets, while guaranteeing the same output distribution with the LLM.

7 Related Work

Transformer-based LLMs have demonstrated significant potential in numerous human-level language modeling tasks by continuously increasing their sizes [7, 9, 37, 43, 48]. As GPT-3 becomes the first model to surpass 100B parameters [3], multiple LLMs (>100B) have been released, including OPT-175B [57], Bloom-176B [38], and PaLM [7]. Recent work has proposed a variety of approaches to accelerating generative LLM inference, which can be categorized into two classes.

Lossless acceleration. Prior work has explored the idea of using an LLM as a verifier instead of a decoder to boost inference. For example, Yang et al. [53] introduced *inference with reference*, which leverages the overlap between an LLM's output and the references obtained by retrieving documents, and checks each reference's appropriateness by examining the decoding results of the LLM. Motivated by the idea of speculative execution in processor optimizations [4, 16], recent work proposed *speculative decoding*, which uses a small language model to produce a sequence of tokens and examines the correctness of these tokens using an LLM [5, 22, 25, 44, 51]. There are two key differences between SpecInfer and these prior works. First, instead of only considering a single sequence of tokens, SpecInfer generates and verifies a token tree, whose nodes each represent a unique token sequence. SpecInfer performs tree attention to compute the attention output of these token sequences in parallel and uses a novel tree-based decoding algorithm to reuse intermediate results shared across these sequences. Second, prior attempts generally consider a single small language model for speculation, which cannot align well with an LLM due to the model capacity gap between them. SpecInfer introduces two novel speculation methods, including 1) expanding from a single SSM and 2) merging from multiple fine-tuned SSMs, and the generated token tree largely increases the coverage of the LLM's output.

Prior work has also introduced a variety of techniques to optimize ML computations on modern hardware platforms. For example, TVM [6] and Ansor [58] automatically generate kernels for a given tensor program. TASO [20] and PET [50] automatically discover graph-level transformations to optimize the computation graph of a DNN. SpecInfer's techniques are orthogonal and can be combined with these systems to accelerate generative LLM computation, which we believe is a promising avenue for future work.

Lossy acceleration. BiLD [23] is a speculative decoding framework that uses a single SSM to accelerate LLM decoding. Unlike the systems mentioned above, the acceleration is lossy: speed comes at the cost of a possible degradation in the generated tokens. Another line of research leverages model compression to reduce LLM inference latency while compromising the predictive performance of the LLM. For example, prior work proposed to leverage weight/activation quantization of LLMs to reduce the memory and computation requirements of serving these LLMs [8, 11, 35, 52, 54]. Recent work further explores a variety of structured pruning techniques for accelerating Transformer-based architectures [10, 17, 49]. A key difference between SpecInfer and these prior works is that SpecInfer does not directly reduce the computation requirement for performing LLM inference, but instead reorganizing LLM inference computation in a more parallelizable way, which reduces memory accesses and inference latency at the cost of manageable memory and computation overheads.

Tree-structured attention. Nguyen et al. [31] introduced *tree-structured attention*, a technique that lets a Transformer model capture the hierarchical composition of input text by running the model on the text's parse tree. To process with attention, it uses a one-on-one mapping to encode and decode the tree. There are two key differences from SpecInfer's tree-based decoding. First, SpecInfer uses a tree to combine candidate sequences to condense prefixes, whereas Nguyen et al. represent a single sequence with its parse tree. SpecInfer does not incorporate parse tree into the LLM, but accelerates inference by verifying decoded sequences in parallel. Second, SpecInfer's attention outputs a token sequence, not a tree.

Multi-sample decoding techniques. Like tree-based speculative inference, *beam search*, *top-k sampling*, and *top-p sampling* consider multiple candidate token sequences at each step and can prune low-probability options. However, tree-based decoding in SpecInfer speculatively predicts and verifies multiple candidates in parallel against an LLM to reduce decoding iterations and latency, leveraging small speculative models (SSMs). In contrast, beam search and top-k/top-p sampling are decoding strategies applied directly to the LLM's output probabilities to generate high-probability sequences without reducing decoding steps. SpecInfer supports beam search, top-k sampling, and top-p sampling. These techniques are orthogonal decoding optimizations and can be combined with tree-based speculative decoding.

8 Conclusion

This paper introduces SpecInfer, a system that accelerates generative LLM inference with tree-based speculative inference and verification. A key insight behind SpecInfer is to simultaneously consider a diversity of speculation candidates to efficiently predict the LLM's outputs, which are organized as a token tree and verified against the LLM in parallel using a tree-based parallel decoding mechanism. SpecInfer significantly reduces the memory accesses to the LLM's parameters and the end-to-end LLM inference latency for both distributed and offloading-based LLM inference.

Acknowledgement

We thank Tianqi Chen, Bohan Hou, Hongyi Jin, the anonymous ASPLOS reviewers, and our shepherd Shan Lu for their feedback on this work. This research is partially supported by NSF awards CNS-2147909, CNS-2211882, and CNS-2239351, and research awards from Amazon, Cisco, Google, Meta, Oracle, Qualcomm, and Samsung.

A Artifact Appendix

A.1 Abstract

The artifact contains the code to run SpecInfer, as well as the datasets and scripts that can be used to reproduce the experiments in the paper.

A.2 Artifact check-list (meta-information)

- **Algorithm:** Tree-based Speculative Inference
- **Program:** spec_infer.cc, incr_decoding.cc
- **Compilation:** CMake
- **Run-time environment:** CUDA, NCCL, MPI, UCX, Python3.
- **Hardware:** Two AWS g5.12xlarge instances, each with 4 NVIDIA A10 24GB GPUs, 48 CPU cores, and 192 GB DRAM.
- **Metrics:** End to-end average latency
- **Output:** End-to-end latency
- **Experiments:** Server-grade GPU inference, Offloading-based inference
- **How much disk space required (approximately)?:** 350GB per node
- **How much time is needed to prepare workflow (approximately)?:** 2h
- **How much time is needed to complete experiments (approximately)?:** 6h
- **Publicly available?:** Yes
- **Code licenses (if publicly available)?:** Apache License v2.0
- **Data licenses (if publicly available)?:** LLAMA is under the GNU license and OPT is under a Non-commercial license
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.10854410.

A.3 Description

A.3.1 How to access.

The artifact is released on Github: https://github.com/goliaro/specinfer-ae. The repository contains SpecInfer's source code, and the instructions to build the framework. We also provide scripts to reproduce the experiments from the paper. To clone the repository, use the following command (make sure to pass the `-recursive` flag):

```
git clone --recursive \
https://github.com/goliaro/specinfer-ae.git
```

A.3.2 Hardware dependencies.

We run out experiments on two AWS g5.12xlarge instances, each with 4 NVIDIA A10 24GB GPUs, 48 CPU cores, and 192 GB DRAM. We provide instructions to create and setup the instances.

A.3.3 Software dependencies.

The following software is required: CUDA 12.1, NCCL, Rust, CMake and Python3. Further, UCX and MPI are required for the multinode experiments. Additional Python dependencies are listed here: https://github.com/flexflow/FlexFlow/blob/inference/requirements.txt. We recommend using the `Deep Learning OSS Nvidia Driver AMI GPU PyTorch 2.1.0 (Ubuntu 20.04)` AMI, and provide scripts and a conda environment to install all the remaining dependencies.

A.3.4 Models.

We use the following LLM/SSM models for our experiments (for each model, we specify in parentheses the corresponding HuggingFace repository): LLaMA-68M (JackFram/llama-68m), LLaMA-7B (huggyllama/llama-7b), LLaMA-65B (huggyllama/llama-65b), OPT-125M (facebook/opt-125m), OPT-13B (facebook/opt-13b), OPT-125M (facebook/opt-30b). You can download all these models with the script:

```
./download_models.sh
```

A.4 Installation

To reproduce the experiments, you will need access to two AWS g5.12xlarge instances (or other machines with the same GPU/CPU/network specs). If you are using the preconfigured instances we provided, you can skip this step.

Launching the instances. Launch two AWS g5.12xlarge instances using the `Deep Learning OSS Nvidia Driver AMI GPU PyTorch 2.1.0 (Ubuntu 20.04)` AMI. Make sure to place the instances in a placement group that utilizes the cluster strategy to achieve low-latency network performance. Attach the same security group to all instances and add an inbound rule in the security group to allow all incoming traffic from the same security group. For example, you can add the following rule: Type: All TCP, Source: Anywhere-IPv4.

Installing the prerequisites. After gaining access to the AWS instances, install the prerequisites by following the steps below. First, activate the conda shell support by running `conda init bash`, and then restarting the shell session. Next, create the conda environment with all the required dependencies by running:

```
conda env create -f FlexFlow/conda/flexflow.yml
conda activate flexflow
```

Multinode setup. Download and build UCX by running the `install_ucx.sh` script. Next, if you are running SpecInfer on two AWS instances, you will need to configure MPI so that the two instances are mutually accessible. Pick a main node, and create a SSH key pair with:

```
    ssh-keygen -t ed25519
```

Append the contents of the public key (`~/.ssh/id_ed25519.pub`) to the `~/.ssh/authorized_keys` file on BOTH the main and secondary machine. Note that if the `.ssh` folder or the `authorized_keys` file do not exist, you will need to create them manually. Finally, create a file at the path `~/hostfile` with the following contents:

```
<main_node_private_ip> slots=4
<secondary_node_private_ip> slots=4
```

replacing <main_node_private_ip> and <secondary_node_private_ip> with the private IP addresses of the two machines, and the number of slots with the number of GPUs available (if you are using the recommended AWS instances, you will use a value of 4). You can find each machine's private IP address by running the command (and use the first IP value that is printed):

```
    hostname -I
```

Install SpecInfer. To install SpecInfer, run the script:

```
./install_specinfer.sh
```

A.5 Basic Test

To ensure that SpecInfer is installed correctly and is functional, run the `basic_test.sh` script. This script will test the basic incremental decoding and speculative inference functionalities, on both single and multi nodes. It will also test the support for offloading. The test passes if it prints the "Test passed!" message.

A.6 Experiment workflow

The artifact comes with scripts to gather the data that can be used to reproduce the results from the paper. It also comes with scripts that can be used to convert the output data into CSV format for plotting.

Running Experiments. We run the following two experiments to evaluate SpecInfer under different hardware setups. The output data will be saved to the `FlexFlow/inference/output` path.

- **Server-grade GPU evaluation.** This experiment tests the performance of SpecInfer on server-grade GPUs. The LLMs and SSMs are loaded in GPU memory, and we measure the end-to-end inference latency using 1 node, and 2 nodes. In the single node case, we measure the performance using 1 GPU, or 4 GPUs. In the multinode case, we use 4GPUs per node. The experiments use LLAMA-7B, OPT-30B and LLAMA-65B as the LLMs, and LLAMA-68M and OPT-125M as SSMs. The experiment runs SpecInfer in three different modes: incremental decoding, sequence-based speculative decoding, and tree-based speculative decoding. The former two are used to obtain data for the ablation study, and the latter is the novel inference mode proposed by SpecInfer, and will be deployed by the user. To run the server-grade GPU evaluation, run:

  ```
  ./server_gpu_experiments.sh
  ```

- **Offloading evaluation.** This experiment tests the performance of SpecInfer when loading only a subset of parameters in GPU memory, while offloading the remaining ones on CPU DRAM. This technique is used to perform inference when the target model is larger than the available GPU memory. In the experiment, SpecInfer uses a single GPU and swaps the model's weights to and from the CPU. To run the offloading evaluation, run:

  ```
  ./offloading_experiments.sh
  ```

Third-party frameworks. Please follow the vLLM, FasterTransformer, and HuggingFace TGI, and FlexGen official documentation to reproduce the performance of the third-party frameworks under the experiment scenarios.

Output data. The scripts above will generate data at the `FlexFlow/inference/output` path. For each scenario, a `.txt` file contains the generated output for each prompt, and a `.out` file contains the stdout logs. The quality of the generated output can be evaluated visually and compared with the output from third-party inference frameworks. We provide scripts to parse the raw output data and generate CSV files that can be used to generate the paper's figures. The README provides all details on the scripts and the mapping between CSV files and figures.

A.7 Evaluation and expected results

The data from the CSV files should show similar performance to the figures from the paper. Some variability is to be expected, but overall, SpecInfer should behave according to Figures 7-11 from the paper.

A.8 Experiment customization

Users can edit the configuration parameters from the evaluation scripts to change various parameters, such as the number of GPUs/CPUs, GPU/CPU memory, batch size, LLM/SSM models used, prompt dataset, full vs. half-precision, and the maximum number of tokens to generate.

References

[1] Jonathan Berant, Andrew Chou, Roy Frostig, and Percy Liang. Semantic parsing on Freebase from question-answer pairs. In *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, pages 1533–1544, Seattle, Washington, USA, October 2013. Association for Computational Linguistics.

[2] Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language. In *Thirty-Fourth AAAI Conference on Artificial Intelligence*, 2020.

[3] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[4] F Warren Burton. Speculative computation, parallelism, and functional programming. *IEEE Transactions on Computers*, 100(12):1190–1193, 1985.

[5] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

[6] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, 2018.

[7] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *arXiv preprint arXiv:2204.02311*, 2022.

[8] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in Neural Information Processing Systems*, 35:30318–30332, 2022.

[9] Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pages 5547–5569. PMLR, 2022.

[10] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot, 2023.

[11] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate quantization for generative pre-trained transformers. In *International Conference on Learning Representations*, 2023.

[12] Yoav Freund, Robert Schapire, and Naoki Abe. A short introduction to boosting. *Journal-Japanese Society For Artificial Intelligence*, 14(771-780):1612, 1999.

[13] Freddy Gabbay and Avi Mendelson. *Speculative execution based on value prediction*. Citeseer, 1996.

[14] Mudasir A Ganaie, Minghui Hu, AK Malik, M Tanveer, and PN Suganthan. Ensemble deep learning: A review. *Engineering Applications of Artificial Intelligence*, 115:105151, 2022.

[15] Aaron Gokaslan*, Vanya Cohen*, Ellie Pavlick, and Stefanie Tellex. Openwebtext corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

[16] John L Hennessy and David A Patterson. *Computer architecture: a quantitative approach*. Elsevier, 2011.

[17] Itay Hubara, Brian Chmiel, Moshe Island, Ron Banner, Joseph Naor, and Daniel Soudry. Accelerated sparse neural training: A provable and efficient method to find n: m transposable masks. *Advances in Neural Information Processing Systems*, 34:21099–21111, 2021.

[18] HuggingFace. Large language model text generation inference. https://github.com/huggingface/text-generation-inference. (Accessed on 08/09/2023).

[19] Hugging Face Inc. Hugging face. https://huggingface.co, 2023.

[20] Zhihao Jia, Oded Padon, James Thomas, Todd Warszawski, Matei Zaharia, and Alex Aiken. Taso: optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 47–62, 2019.

[21] Zhihao Jia, Matei Zaharia, and Alex Aiken. Beyond data and model parallelism for deep neural networks. In *Proceedings of the 2nd Conference on Systems and Machine Learning*, SysML'19, 2019.

[22] Joao Gante. Assisted generation: a new direction toward low-latency text generation, 2023.

[23] Sehoon Kim, Karttikeya Mangalam, Suhong Moon, John Canny, Jitendra Malik, Michael W. Mahoney, Amir Gholami, and Kurt Keutzer. Big little transformer decoder, 2023.

[24] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Yu, Joseph E Gonzalez, Hao Zhang, and Ion Stoica. vllm: Easy, fast, and cheap llm serving with pagedattention. *See https://vllm.ai/ (accessed 9 August 2023)*, 2023.

[25] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. *arXiv preprint arXiv:2211.17192*, 2022.

[26] Jiachang Liu, Dinghan Shen, Yizhe Zhang, Bill Dolan, Lawrence Carin, and Weizhu Chen. What makes good in-context examples for gpt-3? *arXiv preprint arXiv:2101.06804*, 2021.

[27] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Hongyi Jin, Tianqi Chen, and Zhihao Jia. Towards efficient generative large language model serving: A survey from algorithms to systems. *arXiv preprint arXiv:2312.15234*, 2023.

[28] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative llm serving with speculative inference and token tree verification. *arXiv preprint arXiv:2305.09781*, 2023.

[29] Xupeng Miao, Chunan Shi, Jiangfei Duan, Xiaoli Xi, Dahua Lin, Bin Cui, and Zhihao Jia. Spotserve: Serving generative large language models on preemptible instances. *arXiv preprint arXiv:2311.15566*, 2023.

[30] MohamedRashad. Chatgpt-prompts. https://huggingface.co/datasets/MohamedRashad/ChatGPT-prompts, 2023.

[31] Xuan-Phi Nguyen, Shafiq Joty, Steven Hoi, and Richard Socher. Tree-structured attention with hierarchical accumulation. In *International Conference on Learning Representations*, 2020.

[32] NVIDIA. Fastertransformer. https://github.com/NVIDIA/FasterTransformer. (Accessed on 08/09/2023).

[33] OpenAI. Gpt-4 technical report, 2023.

[34] Alessandro Palla. chatbot instruction prompts. https://huggingface.co/datasets/alespalla/chatbot_instruction_prompts, 2023.

[35] Gunho Park, Baeseong Park, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. nuqmm: Quantized matmul for efficient inference of large-scale generative language models. *arXiv preprint arXiv:2206.09557*, 2022.

[36] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. Instruction tuning with gpt-4. *arXiv preprint arXiv:2304.03277*, 2023.

[37] Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.

[38] Teven Le Scao, Angela Fan, Christopher Akiki, Ellie Pavlick, Suzana Ilić, Daniel Hesslow, Roman Castagné, Alexandra Sasha Luccioni, François Yvon, Matthias Gallé, et al. Bloom: A 176b-parameter open-access multilingual language model. *arXiv preprint arXiv:2211.05100*, 2022.

[39] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.

[40] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. High-throughput generative inference of large language models with a single gpu, 2023.

[41] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism, 2020.

[42] James E Smith. A study of branch prediction strategies. In *25 years of the international symposia on Computer architecture (selected papers)*, pages 202–215, 1998.

[43] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye,

George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

[44] Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*, 31, 2018.

[45] Rohan Taori, Ishaan Gulrajani, Tianyi Zhang, Yann Dubois, Xuechen Li, Carlos Guestrin, Percy Liang, and Tatsunori B. Hashimoto. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

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

[47] Colin Unger, Zhihao Jia, Wei Wu, Sina Lin, Mandeep Baines, Carlos Efrain Quintero Narvaez, Vinay Ramakrishnaiah, Nirmal Prajapati, Pat McCormick, Jamaludin Mohd-Yusof, Xi Luo, Dheevatsa Mudigere, Jongsoo Park, Misha Smelyanskiy, and Alex Aiken. Unity: Accelerating DNN training through joint optimization of algebraic transformations and parallelization. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 267–284, Carlsbad, CA, July 2022. USENIX Association.

[48] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[49] Hanrui Wang, Zhekai Zhang, and Song Han. Spatten: Efficient sparse attention architecture with cascade token and head pruning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 97–110. IEEE, 2021.

[50] Haojie Wang, Jidong Zhai, Mingyu Gao, Zixuan Ma, Shizhi Tang, Liyan Zheng, Yuanzhi Li, Kaiyuan Rong, Yuanyong Chen, and Zhihao Jia. PET: Optimizing tensor programs with partially equivalent transformations and automated corrections. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*, pages 37–54. USENIX Association, July 2021.

[51] Heming Xia, Tao Ge, Si-Qing Chen, Furu Wei, and Zhifang Sui. Speculative decoding: Lossless speedup of autoregressive translation.

[52] Guangxuan Xiao, Ji Lin, Mickael Seznec, Julien Demouth, and Song Han. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.

[53] Nan Yang, Tao Ge, Liang Wang, Binxing Jiao, Daxin Jiang, Linjun Yang, Rangan Majumder, and Furu Wei. Inference with reference: Lossless acceleration of large language models. *arXiv preprint arXiv:2304.04487*, 2023.

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

[55] Gyeong-In Yu, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. Orca: A distributed serving system for Transformer-Based generative models. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 521–538, Carlsbad, CA, July 2022. USENIX Association.

[56] Haoyu Zhang, Jianjun Xu, and Ji Wang. Pretraining-based natural language generation for text summarization. *arXiv preprint arXiv:1902.09243*, 2019.

[57] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

[58] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. Ansor: Generating high-performance tensor programs for deep learning. In *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, pages 863–879, 2020.