SpecInfer: Accelerating Generative LLM Serving with Speculative Inference and Token Tree Verification

Xupeng Miao[♠], **Gabriele Oliaro**[♠], **Zhihao Zhang**[♠], **Xinhao Cheng, Zeyu Wang, Rae Ying Yee Wong, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar**[†], **Zhihao Jia**
Carnegie Mellon University [†]University of California San Diego

Abstract

The high computational and memory requirements of generative large language models (LLMs) make it challenging to serve them quickly and cheaply. This paper introduces SpecInfer, an LLM serving system that accelerates generative LLM inference with speculative inference and token tree verification. A key insight behind SpecInfer is to combine various collectively boost-tuned small language models to jointly 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 by 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.

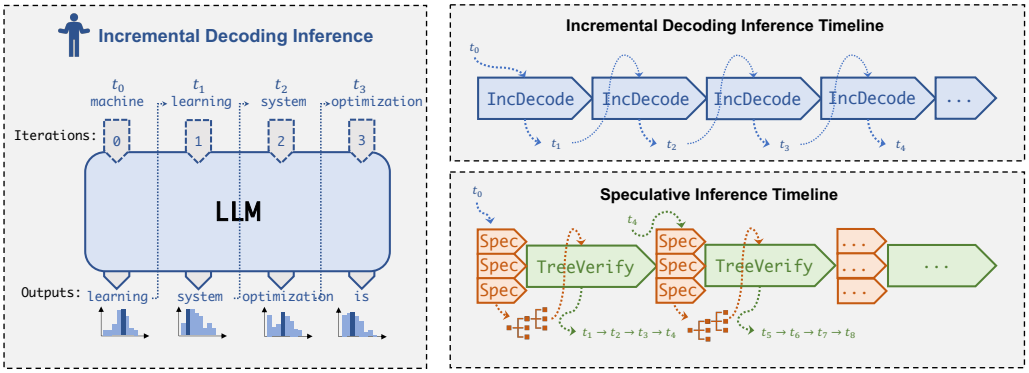
1 Introduction

Generative large language models (LLMs), such as ChatGPT [3] and GPT-4 [24], have demonstrated remarkable capabilities of creating natural language texts across various application domains, including summarization, instruction following, and question answering [43, 22]. However, it is challenging to quickly and cheaply serving these LLMs due to their large volume of parameters, complex architectures, and high computational requirements. For example, the GPT-3 architecture has 175 billion parameters, which require more than 16 NVIDIA 40GB A100 GPUs to store in single-precision floating points, and take several seconds to serve a single inference request [3].

A generative LLM generally takes input as 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.

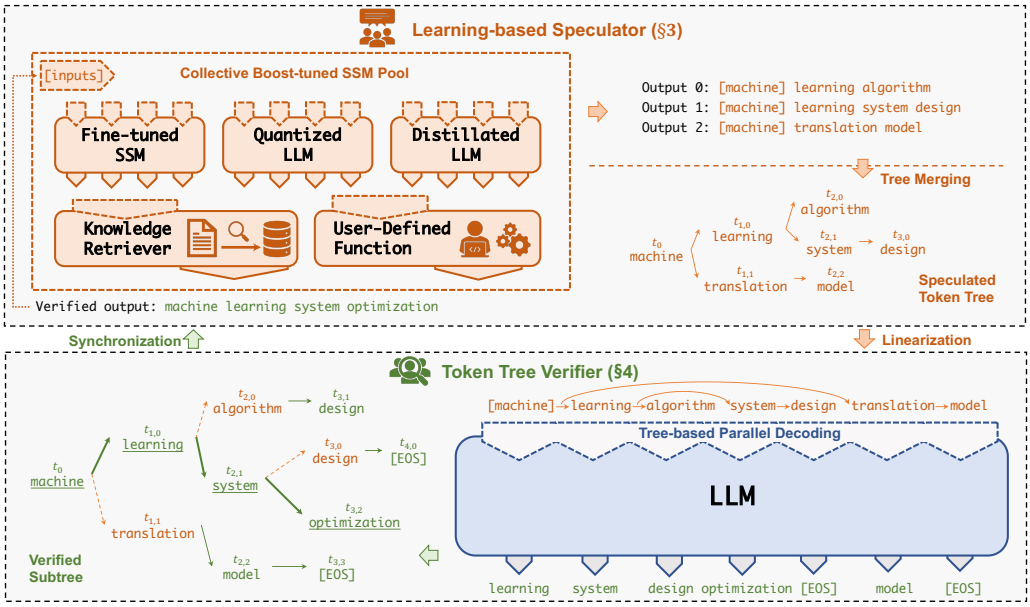
Existing LLM systems generally use an *incremental decoding* approach to serving a request where the system computes the activations for all prompt tokens in a single step and then iteratively decodes *one* new token using the input prompt and all previously generated tokens. This approach respects data dependencies between tokens, but achieves suboptimal 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 [36] 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 serving 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

[♠] Contributed equally.



(a) Incremental decoding.

(b) Timeline Comparison.



(c) Speculative inference and token tree verification.

Figure 1: Comparing the incremental decoding approach used by existing LLM serving systems and the speculative inference and token tree verification approach used by SpecInfer.

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.

This paper introduces SpecInfer, an LLM serving system that improves the end-to-end latency and computational efficiency of generative LLM inference with *speculative inference* and *token tree verification*. A key insight behind the design of SpecInfer is to use an LLM as a token tree verifier instead of an incremental decoder. For a given sequence of tokens, SpecInfer uses a *learning-based speculator* that combines user-provided functions (e.g., document retriever) and multiple collectively *boost-tuned* small speculative models (SSMs) to jointly generate a token tree, whose nodes each represent a candidate token sequence The correctness of *all* token sequences represented by a token tree is then verified against the LLM's original output in parallel using a novel *tree-based parallel* decoding algorithm. This approach allows SpecInfer to opportunistically verify multiple tokens in a single decoding step as long as the speculated token tree overlaps with the LLM's output.

Compared to incremental decoding, SpecInfer's speculative inference and token tree verification introduce small computation and memory overheads for generating and verifying speculated token trees. However, by maximizing the number of tokens that can be successfully verified in a single LLM decoding step, SpecInfer greatly reduces the end-to-end inference latency and improves the

computational efficiency for serving generative LLMs. We evaluate SpecInfer on two LLM families (i.e., LLaMA [34] and OPT [44]) and five prompt datasets. Our evaluation shows that SpecInfer can reduce the number of LLM decoding steps by up to $4.4\times$ ($3.7\times$ on average) and reduce the end-to-end inference latency by up to $2.8\times$.

2 Overview

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**
    $t = \text{DECODE}(\text{LLM}, \mathcal{S})$
    $\mathcal{S}$.append(t)
    if $t = \langle\text{EOS}\rangle$ **then**
        Return $\mathcal{S}$

Algorithm 2 The speculative inference and token tree verification algorithm used by SpecInfer. SPECULATE takes the current token sequence $\mathcal{S}$ as an input and generates a speculated token tree $\mathcal{N}$. SpecInfer's use of an LLM is different from existing systems: the LLM takes a token tree $\mathcal{N}$ as an input and generates a token $\mathcal{O}(u)$ for each node $u \in \mathcal{N}$. Note that the TREEPARALLELDECODE function can generate all tokens in $\mathcal{O}$ in a single LLM decoding step (see Section 4). Finally, VERIFY examines the speculated token tree $\mathcal{N}$ against the LLM's output $\mathcal{O}$ and produces a sequence of verified tokens $\mathcal{V}$, which can be directly appended to the current token sequence $\mathcal{S}$.

Input: A sequence of input tokens $\mathcal{I}$
Output: A sequence of generated tokens
$\mathcal{S} = \mathcal{I}$
while true **do**
    $\mathcal{N} = \text{SPECULATE}(\mathcal{S})$
    $\mathcal{O} = \text{TREEPARALLELDECODE}(\text{LLM}, \mathcal{N})$
    $\mathcal{V} = \text{VERIFY}(\mathcal{O}, \mathcal{N})$
    for $t \in \mathcal{V}$ **do**
        $\mathcal{S}$.append(t)
        if $t = \langle\text{EOS}\rangle$ **then**
            return $\mathcal{S}$

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

Figure 1c shows an overview of our approach. SpecInfer 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 token sequence generated by the LLM using incremental decoding. As shown at the top of Figure 1c, the speculator combines (1) user-provided functions that predict future tokens based on heuristics and/or retrieval-augmented documents, and (2) multiple distilled and/or pruned versions of the LLM, which we call small speculative models (SSMs).

There are a number of ways to prepare SSMs for speculative inference. First, modern LLMs generally have many much 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 [44]. These pre-trained small models can be directly used as SSMs in SpecInfer. Second, to maximize the coverage of speculated token trees, in addition to using these pre-trained SSMs, SpecInfer also introduces a novel fine-tuning technique called *collective boost-tuning* to cooperatively fine-tune a set of SSMs by aligning their aggregated prediction with the LLM's output using adaptive boosting [13].

The speculator automatically combines the candidate token sequences predicted by individual SSMs to construct a token tree, as shown in Figure 1c. Since SpecInfer executes multiple SSMs in parallel, using more SSMs does not directly increase the speculative inference latency. However, using a large number of SSMs will result in a large token tree, which requires more memory and computation resources for verification. To address this challenge, SpecInfer uses a *learning-based* speculative scheduler to learn to decide which SSMs to use for a given input token sequence and the speculative configurations for these SSMs (e.g., the beam search width and depth when running an SSM using beam search).

SpecInfer's usage of the LLM is also different from that of existing LLM serving systems. Instead of using the LLM as an incremental decoding engine that predicts the next single token, SpecInfer uses the LLM as a token tree verifier that verifies whether the speculated token tree overlaps with the true token sequence. For each token, SpecInfer computes its activations by considering all of its ancestors in the token tree as its preceding tokens. For example, 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, $t_{1,0}$, and $t_{2,1}$ are $t_{3,0}$'s ancestors in the token tree. SpecInfer includes a novel tree-based parallel decoding algorithm to simultaneously verify *all* tokens in a speculated token tree in a single LLM decoding step.

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

Reduced memory accesses to LLM parameters. The performance of generative LLM inference is largely limited by GPU memory accesses. In 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 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 Speculative Inference

One major factor of SpecInfer is the design and implementation of the speculator. On the one hand, more accurate speculation can lead to speculated token trees with longer matching lengths, which in turn results in fewer LLM verification steps. On the other hand, due to the intrinsic expression dynamism where some phrases in a sentence are easier to speculate while others are more challenging, a fixed configuration to perform speculation (e.g., the beam width and depth when speculating using beam search) leads to suboptimal performance, since a very small speculation window may result in missed opportunities to match longer token sequences, while a very large speculation window may produce unnecessary tokens.

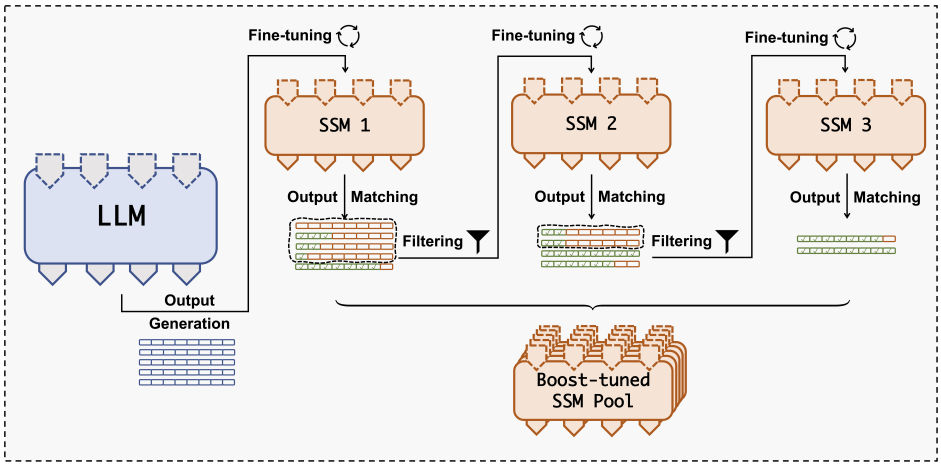


Figure 2: Illustrating SpecInfer's collective boost-tuning technique. When using a single SSM to generate token trees, SpecInfer can verified 2.6 tokens on average in each LLM decoding step. This is due to the misalignment between SSM 1 and the LLM on the first four token sequences. By collectively boost-tuning three SSMs, the average number of verified tokens per LLM decoding step is improved to 7.2.

SpecInfer includes two key techniques to address this challenge. First, to improve the speculative performance of a token tree, Section 3.1 introduces collective boost-tuning, a novel fine-tuning technique that aligns the aggregated prediction of a set of SSMs with the LLM's output using adaptive boosting. Second, to tackle the dynamism across different speculations, Section 3.2 presents a learning-based speculative scheduler that learns to discover the best speculative configuration for a given input token sequence and a set of SSMs.

3.1 Collective Boost-Tuning

As identified in previous works [21, 32], a key limitation of using a single SSM for speculative inference is that the alignment between the SSM and LLM is inherently bounded by the model capacity gap between the two models. Our preliminary exploration shows that using a larger model achieves better speculative performance but introduces additional memory overhead and inference latency to run the larger speculative model.

Consequently, SpecInfer uses an unsupervised approach to collectively fine-tuning a pool of SSMs to align their outputs with that of the LLM by leveraging the adaptive boosting technique, as shown in Figure 2. SpecInfer's SSMs are used to predict the next few tokens that will be generated by an LLM, therefore SpecInfer uses general text datasets (e.g., the OpenWebText corpus [14] in our evaluation) to adaptively align the aggregated output of multiple SSMs with the LLM in a fully unsupervised fashion. In particular, we convert a text corpus into a collection of prompt samples and uses 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 roughly 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. Note that using multiple SSMs increases the memory overhead for storing their parameters on GPUs. However, our evaluation shows that SpecInfer can achieve significant performance improvement by using SSMs 40-100$\times$ smaller than the LLM, making the overhead of hosting these SSMs negligible. In our evaluation, we perform collective boost-tuning offline on publicly available datasets.

3.2 Learning-based Speculative Scheduler

To discover an optimal configuration to launch multiple SSMs at each decoding step, we design a *learning-based* speculative scheduler that learns to decide which SSMs to use for a given input token sequence and the speculative configurations for these SSMs.

The scheduler includes a matching length predictor and cost model. The matching length predictor takes as input the latest feature representation of the final hidden layer from the LLM and outputs a vector of continuous numbers, each corresponding to the expected matching length under a specific speculative configuration. SpecInfer uses a three-layer MLP as the neural architecture of the matching length predictor and considers a configuration space of beam search for each SSM, where the beam width $b \in [1, 2, 4]$ and the beam depth $d \in [1, 2, 4, 8, 16]$, therefore the MLP outputs a vector of 15 numbers, each represent the predicted matching length for a speculative configuration. The predictor is also trained on publicly available datasets in an offload fashion. Note that obtaining the input feature vector for the predictor does not involve extra cost as it's self-contained in the SpecInfer's verifier (see Section 4).

To achieve higher matching length per unit time, we define the following cost function:

$$cost(b, d \mid h) = \frac{f(b, d \mid h)}{L_{\text{verify}}(b, d) + L_{\text{speculate}}(b, d)}, \tag{1}$$

where b and d are the beam search width and depth, h is the input feature vector to the predictor, and $f(b, d \mid h)$ is the predicted matching length for the given speculative configuration (b, d) and current context h. $L_{\text{verify}}(b, d)$ and $L_{\text{speculate}}(b, d)$ are the estimated inference latency for the verifier and speculator, respectively, which are measured by profiling the SpecInfer runtime system. Using the cost function defined in Equation (1), SpecInfer chooses the configuration that minimizes the expected cost for each SSM:

$$(b, d) = \arg\max_{(b,d)} cost(b, d \mid h) \tag{2}$$

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 token sequences against a given LLM.

Token tree. SpecInfer uses a *token tree* to store the results generated by the learning-based speculator. Each token tree $\mathcal{N}$ is a tree structure, where each node $u \in \mathcal{N}$ is labelled by 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].

SpecInfer receives multiple token sequences generated by different SSMs, each of which can be considered as a token tree (with linear tree structure). SpecInfer first merges these token trees into a single tree structure.

Definition 4.1 (Tree Merge)**.** $\mathcal{M}$ is the tree merge of m token trees $\{\mathcal{N}_i\}$ $(1 \leq i \leq m)$ if and only if $\forall 1 \leq i \leq m, \forall u \in \mathcal{N}_i, \exists v \in \mathcal{M}$ 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.

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 architecture. Token tree verification allows SpecInfer to opportunistically decode multiple tokens (instead of a single token in the incremental decoding approach), resulting in reduced 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. SpecInfer performs *tree attention*, a fast and cheap approach to performing Transformer-based attention computation for a token tree, and a number of important system-level optimizations to address this challenge.

Section 4.1 describes tree attention, Section 4.2 introduces the mechanism SpecInfer uses to verify a token tree against an LLM's output, and Section 4.3 presents SpecInfer's optimizations to accelerate token tree verification.

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

4.1 Tree Attention

Transformer-based language models use the attention mechanism to reason about sequential information [36]. Modern LLMs generally use decoder-only, multi-head self-attention layers, each of which takes a single input tensor X and computes an output tensor O via scaled multiplicative formulations as follows.

$$Q_i = X \times W_i^Q, \qquad K_i = X \times W_i^K, \qquad V_i = X \times W_i^V, \tag{3}$$

$$A_i = \frac{(Q_i \times K_i^T)}{\sqrt{d}}, \quad H_i = \text{softmax}(\text{mask}(A_i))V_i, \quad O = (H_1, ..., H_h)W^O \tag{4}$$

where Q_i, K_i, and V_i denote the query, key, and value tensors of the i-th attention head ($1 \leq i \leq h$), 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 casual mask function is applied:

$$\text{mask}(A)_{jk} = \begin{cases} A_{jk} & j \geq k \\ -\infty & j < k \end{cases} \tag{5}$$

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 4, 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 to a sequence of tokens. Therefore, a straightforward approach to verifying a token tree is computing the attention scores for individual token sequences (i.e., S_u for all $u \in \mathcal{N}$). However, this approach is computationally very expensive and involves redundant computations, since two token sequences sharing a common prefix have the same attention outputs for the common prefix due to the casual mask in Equation 4. To address this issue, we generalize the attention mechanism to apply it to tree structures. For each node u in a token tree, its attention output is defined as the output of computing attention on S_u (i.e., the token sequence represented by u). Note that the semantic of SpecInfer's tree attention is different from prior tree-structured attention work, which we discuss in Section 7.

4.2 Verification

For a given speculated token tree $\mathcal{N}$, SpecInfer uses the tree attention mechanism described in Section 4.1 to compute an attention output for each node $u \in \mathcal{N}$. A key advantage of this approach is enabling SpecInfer to examine all tokens in parallel by visiting the LLM's parameters once. This parallel decoding procedure generates an output tensor $\mathcal{O}$ that includes a token for each node $u \in \mathcal{N}$. Algorithm 2 shows SpecInfer's verification process, which 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 $\mathcal{O}(v)$ as a verified node in $\mathcal{N}$ and terminates the verification process. Finally, all verified nodes are appended to the current generated token sequence $\mathcal{S}$. 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.

4.3 Optimizations

This section describes a number of system-level optimizations in SpecInfer to accelerate token tree verification.

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

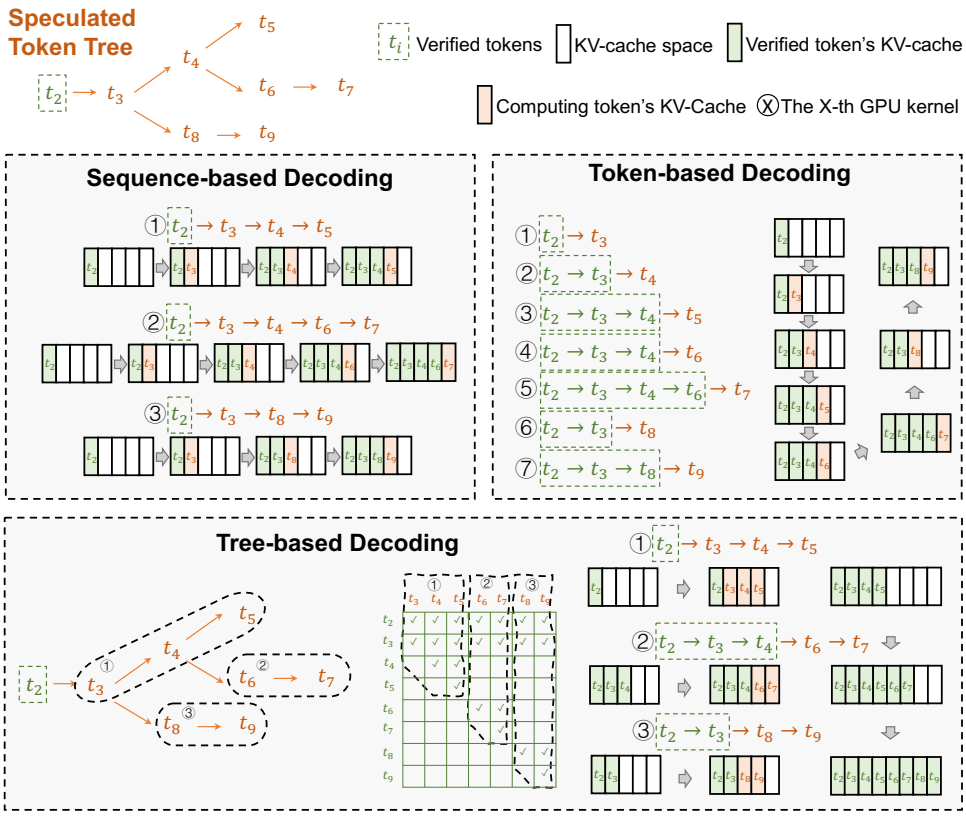


Figure 3: Comparing SpecInfer's tree-based parallel decoding with sequence and token-based decoding.

Depth-first search to update key-value cache. As shown in Equation 4, the attention mechanism of Transformer [36] requires accessing the keys and values of all preceding tokens to compute the attention output of each new token. 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 casual relation guarantees that a token's key and value remain unchanged in subsequent iterations.

A key challenge SpecInfer must address in verifying a token tree is that different sequences of in the token tree may include conflicting key-value caches. For the speculated token tree at the top of Figure 3, 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 straight forward approach to supporting key-value cache is employing the sequence-based decoding of existing LLM inference systems and have a different key-value cache for each sequence of a token tree, as shown in the top-left of Figure 3. However, this approach requires multiple replicas of key-value caches for verifying different sequences and introduces redundant computations since sequences in a token tree may share common prefixes.

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 the top-right of Figure 3, where the arrows indicate how the key-value cache is updated when decoding different tokens. By following a depth-first order to traverse the token tree and update the shared key-value cache, SpecInfer is able to maintain the correct keys and values for all preceding tokens when computing the attention output of a new token.

Tree-based parallel decoding. Existing LLM inference systems use an incremental decoding approach that decodes a single token in each iteration during the generative phase. Therefore, a similar approach for computing tree attention is iteratively calculating the attention output for individual tokens in the token tree 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 a single token. A key challenge that prevents SpecInfer from batching multiple

tokens is that the attention computation for different tokens require different key-value caches and therefore cannot be processed in parallel. For example, the token-based decoding in Figure 3 shows the key-value caches needed for each token.

SpecInfer uses a *tree-based parallel decoding* algorithm to opportunistically batch multiple tokens in a token tree. Specifically, SpecInfer leverages the casual mask of generative LLM inference and groups multiple tokens into a single kernel if each token is the subsequent token's parent. For example, a depth-first search to traverse the token tree in Figure 3 is $(t_3, t_4, t_5, t_6, t_7, t_8, t_9)$. Instead of launching 7 individual kernels to compute the tree attention for these tokens, SpecInfer groups them into three kernels: (t_3, t_4, t_5), (t_6, t_7), and (t_8, t_9), within each of which a token is a child of the previous token. To batch attention computation, SpecInfer uses the key-value cache of the kernel's last token (i.e., t_5 for the first kernel), which results in attention scores that violate the casual dependency. SpecInfer then fixes the attention scores for these pairs. This approach computes the exact same attention output as incremental decoding, while achieving much fewer kernel launches compared to the sequence and token-based decoding mechanism.

5 Discussion

5.1 Overheads of Speculative Inference and Token Tree Verification

SpecInfer accelerates generative LLM inference at the cost of memory and computation overheads. This section analyzes these overheads and show that they are generally one or two orders of magnitude smaller than the memory and computation cost of performing LLM inference using incremental decoding.

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 small models, which collectively speculate the LLM's output. Our evaluation shows that SpecInfer can achieve significant performance improvement by using speculative models 40-100$\times$ smaller than the LLM. As a result, hosting each small speculative model (SSM) increases the overall memory requirement by 1-2%. 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 storing the keys, values, and attention scores for all tokens in a token tree. Due to the necessity for supporting very long sequence length in today's LLM serving, we observe that the memory overhead associated with the token tree is negligible compared to the key-value cache. For example, GPT-4 supports processing up to 32K tokens in a single request; our evaluation shows that a token tree of size 32 or 64 already allows SpecInfer to match xxx tokens on average.

Computation overhead. Similarly, the computation overhead introduced by speculation inference and verification also comes from two aspects. First, SpecInfer needs to run multiple SSMs in the incremental-decoding mode to generate candidate token sequences. SpecInfer processes the SSMs in parallel across GPUs to minimize the latency for generating a speculated token tree. Our evaluation shows that the latency of running a SSM in the incremental-decoding mode is 3.7$\times$ better than that of an LLM. Second, SpecInfer verifies a token tree by computing the attention outputs for all token sequences of the 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.

5.2 Applications

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

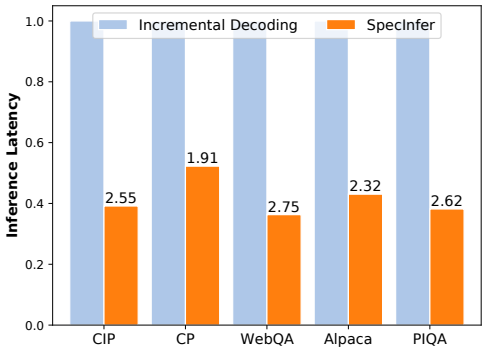


Figure 4: Comparing the end-to-end inference latency of incremental decoding and SpecInfer on five prompt datasets. We use LLaMA-7B as the LLM and all SSMs are derived from LLaMA-160M. The performance is normalized by incremental decoding, and the numbers on the SpecInfer bars indicate the speedups over incremental decoding.

Distributed generative 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. 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 generative 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 for LLM inference, SpecInfer verification mechanism can increase the communication granularity and reduce the number of LLM decoding steps.

Offloading-based generative LLM inference. Another practical scenario where SpecInfer's techniques can help is to help reduce the end-to-end inference latency for offloading-based generative LLM serving systems, 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 [30]. By opportunistically verifying multiple tokens, SpecInfer can effectively reduce the number of LLM decoding steps and the overall communication between CPU DRAM and GPU HBM.

6 Evaluation

6.1 Implementation

SpecInfer was implemented on top of FlexFlow [19, 35], a distributed multi-GPU runtime for DNN computation. FlexFlow exposes an API that allows the user to define a DNN model in terms of its layers. The user can also provide a parallelization plan, specifying the degree of data, model, and pipeline parallelism of each layer. During the training phase, FlexFlow can automatically discover the best parallelization plan, which can then be saved and reused in the inference stage.

Internally, FlexFlow represents a DNN 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 (layers) to the lowest (tasks). Tasks are also the unit of parallelization; they are non-preemptible, and are executed asynchronously.

6.2 Experimental Setup

Datasets. We evaluate SpecInfer on five conversational datasets, namely Chatbot Instruction Prompts (CIP) [25], ChatGpt Prompts (CP) [23], WebQA [1], Alpaca [33, 27], and PIQA [2]. We

only use the prompts/questions from these datasets to form our input prompts to simulate the real-world conversation trace. We randomly selected at most 1000 prompts from each dataset in our evaluation.

Models. To test our system against mainstream generative LLMs, we evaluate our results using two publicly available language models: OPT [44] and LLaMA [34]. More specifically, we select OPT-13B and LLaMA-7B as the LLMs and collectively boost-tune SSMs from OPT-125M and LLaMA-160M. The pre-trained model parameters for OPT-13B, LLaMA-7B, and OPT-125M were directly acquired from their HuggingFace repositories [17]. We didn't find a publicly available pre-trained version of small LLaMA models, and therefore trained a LLaMA-160M from scratch for one epoch using the Wikipedia dataset [10], which took approximately 35 hours on a single NVIDIA A100 GPU. We also used the OpenWebText Corpus [14] to (1) collectively boost-tune multiple SSMs for speculative inference, and (2) collect training data for the learning-based speculative scheduler. Section 6.4 and Section 6.5 report our evaluation on these two components.

Platform. The experiments were conducted on an AWS `g4dn.12xlarge` instance, each of which is equipped with four NVIDIA T4 16GB GPUs, 48 CPU cores, and 192 GB DRAM. The LLMs used in our evaluation do not fit on a single T4 GPU. Therefore SpecInfer performs LLM inference in single-precision floating points and serves the LLMs across the four GPUs using pipeline model parallelism. SpecInfer serves each SSM on a dedicated GPU and runs these SSMs in parallel for a given sequence of tokens.

6.3 End-to-end Performance

We compare the end-to-end inference latency between incremental decoding and SpecInfer on the five prompt datasets. For each prompt dataset, we measured the inference latency of the two approaches on up to 1000 prompts and reported the average inference latency. Figure 4 shows the results. Compared to incremental decoding, SpecInfer reduces the inference latency by 1.9 - 2.7$\times$ while generating the exact same sequence of tokens as incremental decoding for all prompts. The performance improvement is mostly realized by SpecInfer's ability to verify multiple tokens in a single LLM decoding step. Next, we evaluate how collective boost-tuning and the learning-based speculative scheduler help improve SpecInfer's inference performance.

6.4 Collective Boost-Tuning

In this section, we demonstrate the effectiveness of collective boost-tuning in terms of improving the average number of verified tokens in each LLM decoding step. For both the OPT and LLaMA experiments, we fine-tuned four SSMs over the OpenWebText Corpus using collective boost-tuning on top of the pre-trained OPT-125M and LLaMA-160M models, which provides a collection of five SSMs (including the base SSM) in each experiment. As shown in Figure 5 and Figure 6, the average number of tokens verified by SpecInfer in each LLM decoding step increases consistently across all five datasets due to better alignment between the LLM and our tuned collection of SSMs. Table 1 and Table 2 further list the corresponding values and show an overall improvement of 26.4% and 24.8% respectively compared to using only a single pre-trained SSM.

Table 1: Average number of tokens verified by SpecInfer in a decoding step. We used OPT-13B as the LLM and used different numbers of collectively boost-tuned SSMs, all of which were derived from OPT-125M. The beam depth is 16 for all SSMs.

# SSMs	1	2	3	4	5
CIP	3.00	3.39	3.52	3.58	**3.74**
CP	2.95	3.35	3.49	3.52	**3.68**
WebQA	2.51	2.92	3.04	3.09	**3.20**
Alpaca	3.33	3.89	4.06	4.17	**4.35**
PIQA	2.75	3.14	3.26	3.31	**3.43**
Avg	2.91	3.34	3.47	3.53	**3.68**

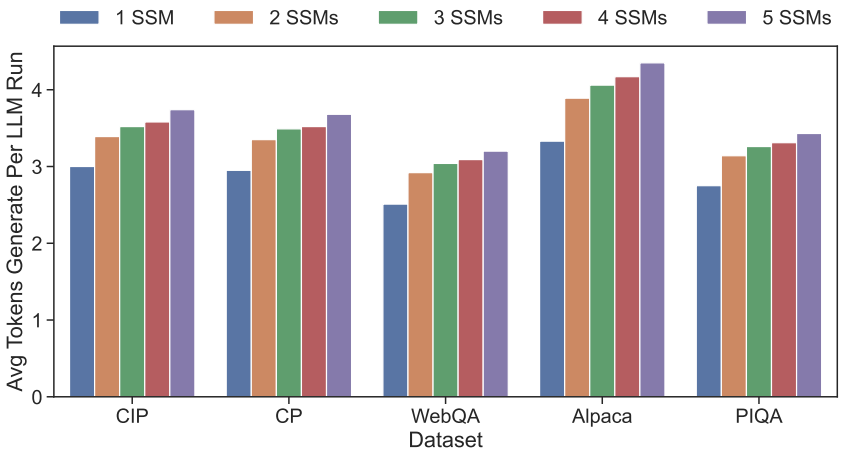


Figure 5: Average number of tokens verified by SpecInfer in each LLM decoding step over five datasets. We use a fixed speculation length of 16 for all the SSMs in this experiment. We used OPT-13B as the LLM and used four SSMs boost-tuned from OPT-125M.

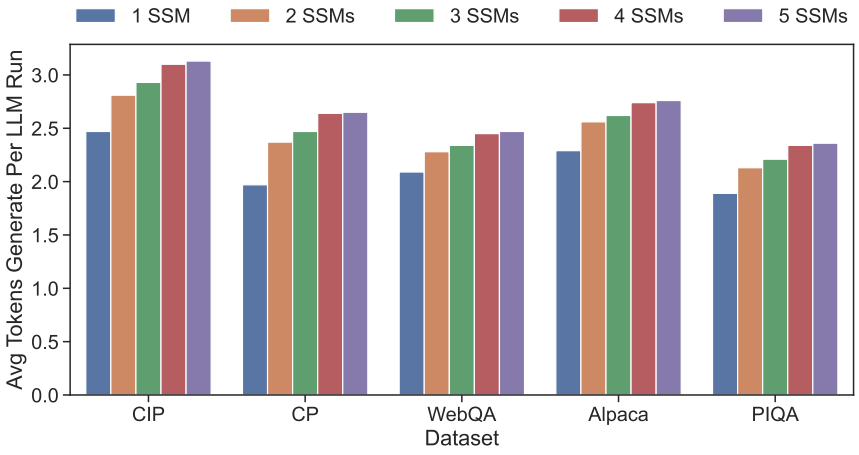


Figure 6: Average number of tokens verified by SpecInfer in each LLM decoding step over five datasets. We use a fixed speculation length of 16 for all the SSMs in this experiment. We used LLaMA-7B as the LLM and used four SSMs boost-tuned from LLaMA-160M.

Table 2: Average number of tokens verified by SpecInfer in a decoding step. We used LLaMA-7B as the LLM and used different numbers of collectively boost-tuned SSMs, all of which were derived from LLaMA-160M. The beam depth is 16 for all SSMs.

# SSMs	1	2	3	4	5
CIP	2.47	2.81	2.93	3.10	**3.13**
CP	1.97	2.37	2.47	2.64	**2.65**
WebQA	2.09	2.28	2.34	2.45	**2.47**
Alpaca	2.29	2.56	2.62	2.74	**2.76**
PIQA	1.89	2.13	2.21	2.34	**2.36**
Avg	2.14	2.43	2.51	2.65	2.67

Table 3: The number of LLM runs and SSM runs with or without the presence of the matching length predictor. When there is no predictor, we use a fixed speculation length of 16 for all the SSMs in this experiment. LLM: OPT-13B, SSMs: OPT-125M

	LLM run		SSM run	
	w/ predictor	w/o predictor	w/ predictor	w/o predictor
CIP	8812	8449	56401	135184
CP	3625	3462	23172	55392
WebQA	12624	12080	74953	193280
Alpaca	11123	10684	72863	170944
PIQA	12625	11560	74548	184960

6.5 Learning-based Speculative Scheduler

For the learning-based speculative scheduler, we demonstrate some preliminary results on the matching length predictor in this section. We use a three-layer MLP with a hidden feature size of 64 as our predictor. We train the predictor on 200K samples over the OpenWebText corpus. The labels are generated using OPT-13B as the LLM and OPT-125M as the SSM. As shown in Table 3, using the predictor can achieve similar LLM runs while reducing the SSM runs significantly due to dynamic speculation length. Nevertheless, there is still plenty of space to improve the predictor as the optimal SSM run would be the average matching length times the LLM run.

7 Related Work

Transformer-based [36] generative LLMs have demonstrated significant potential in numerous human-level language modeling tasks by continuously increasing their sizes [28, 31, 9, 7]. As GPT-3 [3] becomes the first model to surpass 100B parameters, multiple LLMs (>100B) have been released, including OPT-175B [44], Bloom-176B [29], 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. [41] 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, 15], 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 [21, 39, 32, 5, 20]. There are three 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 collective boost-tuning to adapt different SSMs to align with an LLM under different scenarios, which largely increases the coverage of the speculated token trees produced by SpecInfer. Third, an additional challenge SpecInfer has to address is deciding the speculative configuration for a given speculation task. SpecInfer leverages an important observation that the tokens generated by an LLM involve diverse difficulties to speculate, and uses a learning-based speculator to learn to decide which SSMs to use and the speculative configurations for them.

Prior work has also introduced a variety of techniques to optimize ML computations on modern hardware platforms. For example, TVM [6] and Ansor [45] automatically generate efficient kernels for a given tensor program. TASO [18] and PET [38] automatically discover graph-level transformations to optimize the computation graph of a neural architecture. 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. 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 [40, 12, 26, 42, 8]. Recent work further explores a variety of structured pruning techniques for accelerating Transformer-based architectures [11, 37, 16]. A key difference between SpecInfer and these piror 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.

8 Conclusion

This paper introduces SpecInfer, an LLM serving system that accelerates generative LLM inference with speculative inference and token tree verification. A key insight behind SpecInfer is to combine various collectively boost-tuned versions of small language models to efficiently predict the LLM's outputs. SpecInfer significantly reduces the memory accesses to the LLM's parameters and the end-to-end LLM inference latency.

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] Wikimedia Foundation. Wikimedia downloads.

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

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

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

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

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

[16] 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.

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

[18] 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.

[19] 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.

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

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

[22] 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.

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

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

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

[26] 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.

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

[28] 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.

[29] 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.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

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

[44] 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.

[45] 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.