Imperial College
London

MEng Individual Project

Imperial College London

Department of Computing

Parallel Prompt Decoding: A Cost-Effective and Memory-Efficient Approach for Accelerating LLM Inference

Author:
Hao (Mark) Chen

Supervisor:
Prof. Wayne Luk
Dr. Hongxiang Fan

Second Marker:
Dr. Roberto Bondesan

June 17, 2024

Abstract

The auto-regressive decoding of Large Language Models (LLMs) results in significant overheads in their hardware performance. While recent research has investigated various speculative decoding techniques for multi-token generation, these efforts have primarily focused on improving processing speed such as throughput. Crucially, they often neglect other metrics essential for real-life deployments, such as memory consumption and training cost. To overcome these limitations, we propose a novel parallel prompt decoding method that requires only 0.0002% additional trainable parameters compared to the total trainable parameters. By predicting multiple prompt tokens in parallel for approximating outputs generated at future timesteps, *PPD* partially recovers the missing conditional dependency information for multi-token generation, achieving up to 28% higher acceptance rate for long-range prediction as compared to the state-of-the-art parallel decoding method. Furthermore, we present a hardware-aware dynamic sparse tree technique that adaptively optimizes this decoding scheme to fully leverage the computational capacities on different GPUs. Through extensive experiments across LLMs ranging from MobileLlama to Vicuna-13B on a wide range of benchmarks, our approach demonstrates up to 2.49× speedup and maintains a minimal runtime memory overhead of just 0.0004%. More importantly, our parallel prompt decoding can serve as an orthogonal optimization for synergistic integration with existing speculative decoding, showing up to 1.22× further speed improvement.

Acknowledgements

I would like to begin by expressing my sincere gratitude to my amazing supervisors, Prof. Wayne Luk and Dr. Hongxiang Fan, for their invaluable guidance and support throughout this project. I am also grateful to the teams at Samsung AI Center and Hong Kong Polytechnic University for providing the necessary hardware platforms and their insightful advice. Finally, I would like to extend a special thank you to Suki Yu for her unwavering support.

Contents

1 Introduction **4**
- 1.1 Motivation . . . 4
- 1.2 Proposed approach: Parallel Prompt Decoding . . . 5
- 1.3 Challenges and contributions . . . 6

2 Preliminaries **7**
- 2.1 LLM Basics . . . 7
  - 2.1.1 Transformer Architecture . . . 7
  - 2.1.2 Types of Language Models . . . 8
  - 2.1.3 Lifecycle of Foundation Models . . . 8
- 2.2 LLM Inference . . . 9
  - 2.2.1 Prefill Phase . . . 9
  - 2.2.2 Autoregeressive Sampling Phase . . . 9
  - 2.2.3 Key-Value Cache . . . 10
  - 2.2.4 Accelerators . . . 10
  - 2.2.5 Batch Inference . . . 11
- 2.3 Challenges of LLM inference . . . 11
- 2.4 Full Parameter Fine-Tuning . . . 11
  - 2.4.1 Transfer Learning . . . 11
  - 2.4.2 Limitations of FPFT . . . 12
  - 2.4.3 Memory-Efficient Fine-Tuning . . . 12
- 2.5 Parameter-Efficient Fine-Tuning . . . 12
  - 2.5.1 Objectives . . . 12
  - 2.5.2 Adapter-based Tuning . . . 13
  - 2.5.3 Low-Rank Adaptation . . . 13
  - 2.5.4 Prompt Tuning . . . 14
  - 2.5.5 Prefix Tuning . . . 14
- 2.6 Summary . . . 15

3 Related Work **16**
- 3.1 System-level Optimization . . . 16
- 3.2 Autoregressive Algorithm-level Optimization . . . 16
  - 3.2.1 Early Exiting . . . 16
  - 3.2.2 Cascade Inference . . . 17
  - 3.2.3 Speculative Decoding . . . 17
- 3.3 Non-Autoregressive Generation . . . 18
  - 3.3.1 Overview . . . 18
  - 3.3.2 Adapter-based Method . . . 19
  - 3.3.3 Jacobi Decoding . . . 19
  - 3.3.4 Limitations of NAD . . . 20
- 3.4 Summary . . . 20

4 Parallel Prompt Decoding **21**
- 4.1 Prompt Tokens . . . 21
- 4.2 Ensemble Prompt Tokens . . . 22
- 4.3 Training . . . 23
  - 4.3.1 Random Insertion of Prompt Tokens . . . 23

4.3.2 Knowledge Distillation 23
4.4 Extensions 24
4.4.1 Prefix Tuning + Prompt Token 24
4.4.2 Custom Decoding Heads + Prompt Token 25
4.4.3 Attention Masking for EPTs 25
4.4.4 Aggregation Method for EPTs 26
4.4.5 Multi-exit Ensemble 26
4.5 Inference 27
4.5.1 Candidate Prediction 27
4.5.2 Candidate Verification 28
4.5.3 Candidate Acceptance 29
4.5.4 Custom KV Cache Implementation 30

5 Dynamic Sparse Tree **31**
5.1 Top-k Candidate 31
5.2 Tree Attention 31
5.3 Full Sparse Tree: A Naive Implementation 32
5.4 Dynamic Sparse Tree: Be Hardware-Aware 33
5.5 Customized Attention Mask for Tree Attention 33
5.6 Construction Algorithm for Dynamic Sparse Tree 33
5.6.1 Problem Formulation 33
5.6.2 Solve the Optimization Problem 34
5.6.3 Hardware-Aware Tree Construction 35

6 Experiments **36**
6.1 Evaluation Setup 36
6.1.1 Models and testbeds 36
6.1.2 Training 36
6.1.3 Datasets 36
6.1.4 Additional Experimental Details 37
6.2 Speedup Ratios Compared to Other Parallel Decoding Methods 37
6.3 Long-range Token Prediction 38
6.4 Inference Memory Efficiency and Synergistic Integration 40
6.4.1 Memory efficiency 40
6.4.2 *PPD* + Speculative Decoding 41
6.5 Comparison of Throughputs on Different GPUs 41
6.6 Training Loss 41
6.7 Comparison of Training of *PPD* with Medusa 43
6.8 Ablation Study 44
6.8.1 Dynamic Sparse Tree 44
6.8.2 Hardware-aware Tree Size 44
6.8.3 Effect of EPTs on Prediction Accuracy 44
6.8.4 Impact of Various Hyperparameters on Prediction Accuracy 45
6.8.5 Prefix Tuning + Prompt Token 46
6.8.6 Custom Decoding Heads + Prompt Token 46
6.8.7 Attention Masking for EPTs 46
6.8.8 Aggregation Method for EPTs 47
6.8.9 Multi-exit Ensemble 47
6.9 Summary 47

7 Conclusion **48**
7.1 Summary 48
7.2 Limitations 48
7.3 Ethical Issues 49
7.4 Future Work 49
7.4.1 Further Reduce the Training Cost 49
7.4.2 Minimize the Computational Overhead 49
7.4.3 Marry Acceleration with Downstream Task Customization 50
7.4.4 Combination of *PPD* with Other Acceleration Methods 50

Chapter 1

Introduction

1.1 Motivation

Large language models (LLMs) [1, 2] are increasingly gaining influence in the landscape of artificial intelligence (AI) due to their exceptional capability in solving a wide range of language tasks, including machine translation, natural language understanding, summarization, question answering, etc. These models are characterized by both their impressive parameter sizes and their ability to perform certain tasks at a level comparable to humans. Open AI's GPT-3 model [3], for instance, boasts an extraordinary 175 billion parameters and possesses the capability to produce textual content indistinguishable from that created by humans. LLMs' presence extends beyond Natural Language Processing (NLP), influencing a wide range of fields such as code development [4], circuit design [5], and music composition [6].

However, the exceptional success of LLMs has also introduced a number of challenges, with the most significant being their considerable computational demands during inference. Due to their unprecedented model size and complexity, the inference process for LLMs imposes significant demands on computational resources, memory capacity, and energy consumption. Among the various challenges faced by LLMs, one of the most critical is the limited parallelism inherent in autoregressive generation. [7]. Autoregressive generation, the de facto approach employed in LLM inference, suffers from inadequate hardware performance due to its inherent sequential nature [8]. In autoregressive generation, each step produces only one single token, leading to a situation where the latency of LLMs' request is dominated by the response's length. Each decoding step has a strong dependency on the previous step so it fails to utilize the parallel processing capabilities of contemporary GPUs, frequently leading to sub-optimal GPU utilization. Such inefficiencies pose significant challenges for many real-life LLM applications that need to generate sequences with minimal delay.

A possible solution to the problem is speculative decoding [9]. Speculative decoding employs a guess-and-verify framework, whereby a smaller draft model predicts the next few tokens and the

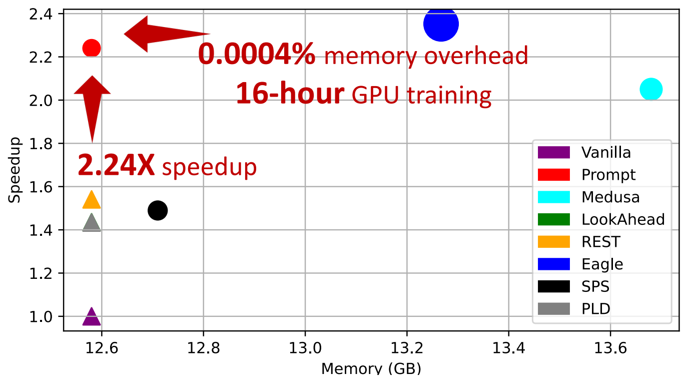


Figure 1.1: Comparison of memory consumption, speedup, and training cost (GPU hours), evaluating on MT-Bench with Vicuna-7B. The diameter of circles represents the GPU hours required for training.

original LLM concurrently verifies the draft. The effectiveness of this method is limited by the ability of the draft model to recover the same token distribution as the original LLM. If the draft token acceptance rate is low, a great portion of the computation will be spent on the verification of wrong tokens, potentially slowing the decoding process significantly. Unfortunately, developing a draft model that consistently achieves a high acceptance rate is often complex, and draft models often struggle to perform uniformly across different base models and datasets. Additionally, the extra runtime memory overhead for executing draft models poses a significant barrier to the broader adoption of speculative decoding, particularly in edge and mobile environments where memory capacity is limited. Considering the growing need for user privacy and personalization, deploying LLMs on devices urges a more memory- and cost-efficient solution for accelerating LLM inference. Recent efforts have explored the possibility of generating multiple tokens in parallel without relying on a separate transformer draft model [10]. Approaches such as inserting additional decoding heads [11] and retrieving frequently used tokens [12] are employed to enhance performance. However, these methods either aggressively assume conditional independence among the tokens generated in a single step [11, 12], or use placeholder tokens (*e.g.*, [PAD] token) that do not convey enough contextual information [10]. Therefore, they often suffer from low acceptance rates or degradation in output quality due to the lack of sufficient conditional information during inference.

1.2 Proposed approach: Parallel Prompt Decoding

To alleviate the complexity and overhead associated with the use of draft models while maintaining a high acceptance rate, we propose *Parallel Prompt Decoding* (*PPD*), a novel architecture-agnostic and memory-efficient framework that adopts prompt tuning for non-autoregressive LLM inference. *PPD* introduces the use of prompt tokens, the meticulously trained embeddings, for multi-token prediction. Specifically, these trained prompt tokens are appended to the original input sequence in parallel, enabling the concurrent generation of multiple output tokens in a single forward pass and facilitating parallel execution of verification and prediction to ensure the quality of generated tokens.

The key intuition of *PPD* lies in the observation that if trained properly, prompt tokens appended to the input could approximate tokens generated at future timesteps, hence partially recovering the missing conditional dependency information for multi-token generation. By positioning trained prompt tokens at proper places, *PPD* achieves up to 28% higher acceptance rate when predicting long-range tokens as compared to Medusa. This is one of the most **exciting** features of *PPD*, as it significantly enhances the performance on long-range predictions while using very few additional trainable parameters.

To further increase the token acceptance rate, we generate multiple candidate continuations with each prompt token and use them in combination with a tree attention mask [13] to minimize the computation and memory overhead. The capability of *PPD* to use low-cost prompt tokens for accurate multi-token prediction forms the foundation for accelerating LLM inference. As shown in Figure 1.1, *PPD* achieves a comparable speedup to the state-of-the-art speculative decoding approaches with negligible memory overhead and reduced training cost. Moreover, to facilitate the optimized implementation of *PPD* across different hardware platforms, we propose a hardware-aware dynamic sparse tree technique that adaptively refines the prompt structure during runtime based on the computational and memory resources available on the specific hardware.

To demonstrate the effectiveness of our approach, we evaluate *PPD* on MobileLLaMA [14], Vicuna-7b and Vicuna-13b [15]. Running on a single A100-40GB GPU, our method achieves a speedup ratio from **2.12**× to **2.49**× across a diverse range of popular datasets including MT-Bench, HumanEval, and GSM8K. Our experiments demonstrate that *PPD* not only achieves comparable throughput to the state-of-the-art speculative decoding method, but it also manages this with significantly fewer trainable parameters—specifically, **0.0002%** of trainable parameters compared to the total number of parameters—and incurs only a minimal memory overhead (**0.0004%**). *PPD* is extremely cost-efficient and memory-efficient. The training of prompt tokens can be completed in **8 hours** using four GeForce RTX 3090 GPUs and in just **4 hours** on four A100-40GB GPUs, compared to the 1-2 days required for Eagle [16]. Furthermore, since *PPD* does not require the modification of the original LLM nor the addition of extra networks, it is highly adaptable and orthogonal to other decoding techniques. For instance, it can be effectively combined with a draft model to further reduce inference latency.

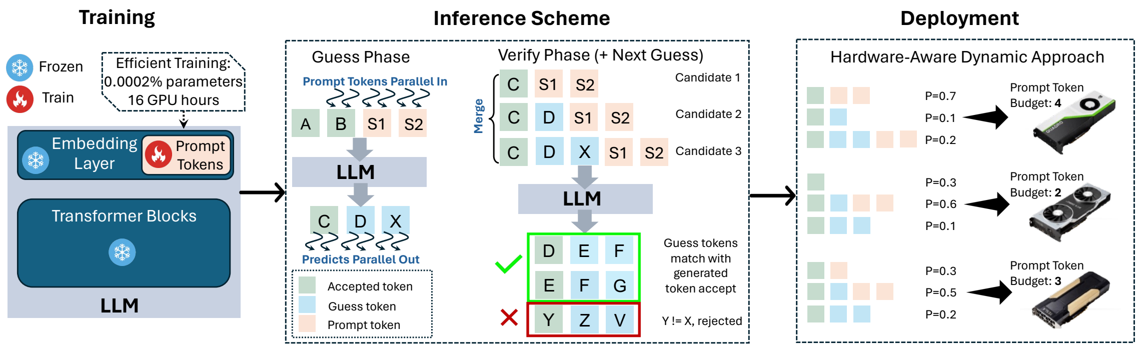


Figure 1.2: Overview of *PPD* The left section shows the location of trainable parameters and the middle section displays the combined **guess-and-verify** process during inference. The "prompt token" denotes the special token with separately trained embeddings to perform parallel prediction.

1.3 Challenges and contributions

We summarize the challenges and our proposed solutions below:

Low acceptance rate of current parallel decoding methods. To address this, we introduce *Parallel Prompt Decoding* (*PPD*), which uses cost-effective prompt tokens for non-autoregressive LLM inference, achieving a high acceptance rate for long-distance token prediction while maintaining output quality.

High training cost and memory overhead of speculative decoding methods. Our method eliminates the need for a separate draft model by training only additional word embeddings, significantly reducing training costs and memory overhead.

Compute resource constraints of LLM inference. We developed a hardware-aware dynamic sparse tree technique that adaptively optimizes the prompt structure of *PPD* at runtime, based on available computational and memory resources, facilitating efficient deployment on various hardware platforms.

We have open-sourced the implementation of *PPD*, along with comprehensive evaluations on various models and benchmarks. Our experiments demonstrate that *PPD* achieves significant speed improvements with negligible memory overhead and reduced training costs.

Our approach can be categorized as parallel decoding, with three novel features distinguishing it from other methods: *1)* *PPD* trains the embeddings of parameterized ensemble prompt tokens, *2)* it utilizes a dynamic sparse tree, adapting its structure at every inference step, and *3)* we propose a hardware-aware algorithm for designing a dynamic sparse tree tailored to each hardware platform.

Chapter 2

Preliminaries

Firstly, we introduce some relevant background information which the problem statement and the proposed approach are based upon. Section 2.1 explains the basics of LLMs and the lifecycle of foundation models. After that, Section 2.2 introduces process of LLM inference and explain the current challenges, which forms the motivation of our method. Finally, Section 2.4 and Section 2.5 discusses and compares the two popular families of fine-tuning methods.

2.1 LLM Basics

2.1.1 Transformer Architecture

The Transformer-based LLMs, pivotal in the advancement of NLP, represents a significant shift in the way natural languages are generated and processed. The Transformer model [17] is fundamentally different from the previous sequence-to-sequence models as its attention-based mechanism enables the effective capture of long-range dependencies in a textual context. The Transformer model has the following components:

1. **Embedding and encoding**. The embedding layer is a learned function that maps discrete input tokens into continuous vector representations. The word embeddings encapsulate the semantic relationship between words, representing the meaning of each token. Differing from recurrent neural networks (RNNs), Transformers have no information of the order of a sequence of tokens. To mitigate this limitation, the architecture adds positional encodings to the word embeddings, providing positional information of each token for the subsequent layers. The original Transformer paper uses sin and cosine functions for the positional encoding:

$$\mathrm{PE}_{(pos,2i)} = \sin\left(\frac{pos}{10000^{2i/d_{\mathrm{model}}}}\right)$$
$$\mathrm{PE}_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{2i/d_{\mathrm{model}}}}\right)$$

2. **Multi-head self-attention**. The self-attention component is the core of the Transformer model, assigning different weights to different input tokens when making predictions for each output token. The attention function could be described by the following formula:

$$\mathrm{Attention}(Q,K,V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$
$$\mathrm{MultiHead}(Q,K,V) = \mathrm{Concat}(head_1, head_2, \ldots, head_n)W$$

   Here, Q, K, and V are the query, key, and value matrices, obtained by a learned transformation of the input x. d_k is the dimension of the key vector. W is a learned linear transformation. In multi-head self-attention, the self-attention function is executed with disjoint sets of parameters in parallel. The outcomes are then concatenated and then transformed through a matrix. Its purpose is to identify different patterns and relationships in the sequence, enriching the overall representation.

For the decoder block, masking is applied in self-attention to prevent the previous tokens from attending to subsequent tokens. The masking mechanism, together with shifting the output embeddings by one position, makes sure the prediction for a token at position i only depends on the tokens on the left of position i.

3. **Feed-Forward Network (FFN)**. FFN, positioned in every layer of the Transformer model, contributes greatly to the model's computational demand. An FFN generally comprises two linear transformations, separated by a non-linear ReLU function. This can be mathematically represented as:

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

W_1, W_2, b_1, b_2 are all learned parameters. Following the calculation of the multi-head attention, where information from different segments of the text is aggregated, the FFN processes this combined information independently at each position. This capacity for parallel processing is a fundamental strength of the Transformer model, facilitating efficient handling of sequences. The combination of FFN and self-attention empowers the Transformer model to capture a variety of linguistic contexts.

2.1.2 Types of Language Models

There are conventionally three types of language models:

- **Encoder-only**. Encoder-only language models, like BERT [18], produce vector representations of the input sequence, but they do not generate text directly. The primary objective of encoder-only models is to identify and comprehend patterns and semantics within the input data, making them useful in text classification and sentiment analysis.
- **Decoder-only**. Decoder-only language models, such as GPT-3 [3], are the standard autoregressive models that sequentially generate text. These models produce both contextual embeddings of the input sequence and the probability distribution over the next token. They are well-suited for tasks that involve text generation and completion.
- **Encoder-Decoder**. Encoder-Decoder language models, such as T5 [19], encompass a two-part structure: an encoder that converts input text into vector representations and a decoder that utilizes these representations to generate output text. This architecture is particularly adept at handling sequence-to-sequence tasks, such as machine translation, where the goal is to transform an input sequence into a new, contextually related output sequence.

This project primarily focuses on the Decoder-only models and Encoder-Decoder models as our objective is to accelerate the inference of text generation tasks.

2.1.3 Lifecycle of Foundation Models

Foundation models are any models that are first trained on expansive datasets to gain general capability, often using self-supervision, and then adapted to a wide range of downstream tasks. They are named for their pivotal role and the need for further tuning. With the appearance of models like BERT [18], DALL-E [20], GPT-3 [3], foundation models marked a paradigm shift in the field of machine learning [21]. Key stages of the development and deployment of foundation models are listed below:

- **Data Creation and Curation**: Data creation is intrinsically a human-centric process as all data are collected from human activities. The collected data are then filtered and preprocessed to form datasets.
- **Pre-training**: Foundation models are trained on curated datasets with pre-training objectives such as Language Modelling [3], Masked Language Modeling [18], and Prefix Language Modeling [19]. The choice of the pre-training objective hugely influences the data efficiency and hence the performance of the foundation model.

- **Fine-tuning**: Fine-tuning is a critical step before deployment. This stage involves not only modifying the foundation model to ensure accurate and relevant output for specific tasks like question answering but also calibrating it to avoid the generation of unethical or inappropriate content.
- **Inference**: During deployment, the model is used for inference. There are a few important objectives of efficient LLM serving: 1) Latency and response time, 2) Throughput, and 3) Hardware compatibility and acceleration.

Various optimization techniques are proposed to target each of the 4 stages above. In this project, while we aim to accelerate the inference stage, techniques from fine-tuning stage are applied. We carefully design our approach to make sure it is agnostic to the data stage and pre-training stage of the model used.

2.2 LLM Inference

2.2.1 Prefill Phase

LLM inference contains two distinct phases with different characteristics. The first phase is the prefill phase, or the prompt phase, where all user-provided prompt tokens are processed in parallel by the model to produce intermediate states. The intermediate states contains the vector representations of the keys and values, which will be used later to generate new tokens.

2.2.2 Autoregeressive Sampling Phase

Following the prefill phase is the autoregressive sampling phase, or the token-generation phase, where the LLM predicts the next token based on the given tokens. Assume the given sequence is $x_{1:L}$ and L is the sequence length, the joint probability of the sequence could be expressed using the chain rule of probability:

$$p(x_{1:L}) = \prod_{i=1}^{L} p(x_i | x_{1:i-1})$$

To generate the next token x_{L+1}, we could sample it from the conditional probability:

$$x_{L+1} \sim p(x_{L+1} | x_{1:L})^{\frac{1}{T}}$$

$T \leq 0$ is the temperature hyperparameter that determines how random the generation process is. When $T = 0$, the most probable token is deterministically chosen. When $T = 1$, the next token is sampled exactly following the conditional probability. When $T = 1$, the next token is randomly selected among the entire vocabulary. The distribution is re-normalized to make sure its summation is equal to 1.

For the generation of the token x_{L+2}, we first obtain a new sequence $x_{1:L+1}$ by appending the newly generated token x_{L+1} and then repeat the sampling process. The iterative process continues until a stopping criteria is met. This generation process is called autoregressive generation, as shown in Algorithm 1.

Here, $sample_y$ is a sampling function based on the conditional probability, usually generated by the LLM decoder. A frequently used stopping condition in token generation is the production of a special token, denoted as EOS, marking the end of the sequence. In LLM inference, the autoregressive decoding plays a pivotal role in producing text that not only exhibits coherence but also aligns with the context. This method effectively conditions the generation of each new token on a detailed comprehension of the entire sequence of tokens already generated. Most analyses of the performance of Transformer-based LLM inference in the community, which includes evaluating the floating-point operations per second (FLOPS), and the input/output and memory demands, are based on auto-regressive decoding algorithm.

When the generated sequence is sufficiently long, the inference latency is dominated by the autoregressive sampling phase. Consequently, various methods have been proposed to accelerate this phase. These methods are discussed in more detail in Chapter 3.

Algorithm 1 Autoregressive Generation for LLM Inference

```
1: Initial sequence $x_{1:L}$ from a given context or user input
2: for $i = (L + 1)$ to MAX_LEN do
3:     Predict the next token $x_i = sample_y(p(y|x_{1:i-1}))$
4:     $x_{1:i} \leftarrow$ Append $x_i$ to $x_{1:i-1}$
5:     if $x_{1:i}$ meets stopping criteria then
6:         break
7:     end if
8: end for
9: return $x_{1:L+N}$
```

2.2.3 Key-Value Cache

The concept of a Key-Value (KV) cache is an important aspect in LLM inference, particularly in improving the efficiency and reducing cost. A KV cache in LLMs is a mechanism that stores in cache previously computed key and value pairs used in the attention mechanism of models. The KV cache is initially populated in the prefill phase and persists throughout the token-generation phase. During autoregressive decoding, the self-attention requires the key-value pairs for each token currently in the sequence. By caching these pairs, the model can refer back to previously computed information without recalculation, thus saving computational resources. However, since the size of KV cache increases sequence length, it could also put pressure on memory consumption. It is worth noting that KV cache only happens in the decoder.

2.2.4 Accelerators

The remarkable progress of LLMs can be significantly attributed to advancements in accelerators, which provides the necessary hardware for effective model execution. The use of Graphics Processing Units (GPUs), dominates the field of deep learning due to their exceptional ability to perform parallel processing. Different from traditional Central Processing Units (CPUs) tailored for general-purpose sequential tasks, GPUs are built with thousands of efficient cores in order to execute a large number of operations in parallel. Thus, GPUs are well-suited for tasks that needs parallel computation, such as matrix multiplication, video decoding , and deep learning.

Streaming Multiprocessors (SMs) are the building blocks of GPUs. Each SM has several cores and shares one instruction unit. Independent threads run one SM in parallel. The shared memory (SRAM) within each SM facilitates data exchange and synchronization between threads, while the high-bandwidth memory (HBM) enables quicker data transfers and alleviates memory access bottlenecks in large-scale calculations. With these components, GPUs excel in handling the high demand of LLMs for computational resources and data transfer. Advanced GPUs, like NVIDIA A100 Tensor Core GPU [22], offer increased memory bandwidth, and specialized computing units like Tensor Cores for matrix math operations. The support for a full range of data types and precision allows for a balance between computational speed and precision, a key aspect in optimizing LLMs.

Programming languages for GPUs, such as NVIDIA's CUDA, provide custom control over thread operations, enabling maximal exploitation of GPU parallelism. This has facilitated the development of GPU applications, contributing to a vibrant software ecosystem and advancing LLM research.

In addition to GPUs, a number of hardware platforms is used for LLM deployment, including ASICs [23], TPUs [24], FPGAs [25], and other AI chips from various manufacturers. While these hardware platforms offer distinct advantages in specific scenarios, the extensive body of research and development around GPUs positions them as a central reference point for understanding LLM inference methodologies.

Software-hardware co-design is an engineering methodology where software and hardware components of a system are designed in a coordinated manner, rather than independently. This approach aims to optimize system performance, efficiency, and flexibility by allowing hardware and software to be more closely integrated and tailored to each other's capabilities and requirements. When developing acceleration algorithms for LLM inference, the constraints and strengths of the hardware should also be taken into account.

2.2.5 Batch Inference

Accelerators like GPU are massively parallel computing units, so it is important for LLMs to fully utilize the parallel processing power to increase throughput. Batching is a simple and effective way to increase parallelism, and GPU utilization. Many input sequences are collected and batched to process at once, efficiently reducing memory bandwidth use and increasing compute utilization. There are several ways to implement batching for LLM inference.

- **Static batching**. Static batching is the naive implementation of batching and the size of the batch remains constant until the completion of inference. Multiple prompts from clients are batched and processed by the LLM, only returning the output sequences if the inference for all prompts is complete. Static batching could either be done at the client side or the server side. Static batching could be inefficient as the inference of some requests might finish at earlier time steps and these requests have to wait for others, leading to under-utilization of compute resources. Static batching is only desirable if the output sequence length is uniform, for example, in a classification task.

- **Continuous batching**. To address the limitations of static batching, continuous batching [26] is proposed. In continuous batching, instead of waiting until all requests in a batch have completed generation, the server uses iteration-level scheduling to determine the batch size dynamically per iteration. A new request could be added to the batch once an old request has finished, leading to saturated GPU utilization. Unfortunately, the detailed implementation is more complicated as the computational patterns in prefill phase and token generation phase are different, making it more difficult to add new requests to a in-process batch.

2.3 Challenges of LLM inference

Several studies [7, 27, 28] have analysed and pointed out the computational difficulties of LLM inference. Relating to what we have discussed above, we summarize the key challenges of LLM inference:

1. **Low parallelizability**. Due to the sequential nature of the autoregressive generation stage, inference must produce output token by token. Although batching could partially address this problem by generating output sequence for multiple prompts at once, only throughput increases and the latency is still high. This problem is especially significant for latency-sensitive applications like chatbots.

2. **Large memory footprints**. The large size of both the trained model parameters and the transient state needed (KV cache) contributes to the large memory footprints during LLM inference. While model parameters are usually loaded in the prefill stage, transient state are stored during decoding time and grows with sequence length. KV cache also increases with batch size. With a batch size of 512, the KV cache could grow to 3TB, 3 times the model size [28]. In general, at small batch sizes and sequence length, the time to load model parameters is much larger than that to load KV cache, and vice versa.

3. **Scalability**. The inference cost scales quadratically with input sequence length [7]. Hence, there is usually a limit on the context length.

2.4 Full Parameter Fine-Tuning

As mentioned in Subsection 2.1.3, foundation models are generally not production-ready as the pre-training objectives are usually different from the objectives of the downstream applications. The fine-tuning phase serves to align the pre-trained capabilities of foundation models and the goals of downstream tasks.

2.4.1 Transfer Learning

Different from retraining model based on new data for new tasks, the concept of model fine-tuning, which hinges on transfer learning, efficiently utilizes the common capabilities learned by the foundation model. The foundation model parameters are used as efficient initialization for new tasks to reduce the end-to-end training costs and risks of insufficient data.

In short, transfer learning is a machine learning method where a model trained on one task is re-purposed on a different but related task. Transfer learning is not a new concept. It was proposed decades ago and used in a wide range of tasks, such as climate models [29], disease prediction [30], and computer vision [31].

2.4.2 Limitations of FPFT

As the name suggests, Full-Parameter Fine-Tuning (FPFT) modifies all pre-trained parameters. It is a simple and straightforward way to implement fine-tuning. However, it faces the following key challenges, which justifies the employment of additional fine-tuning optimizations:

- **Computational costs**. FPFT requires substantial computational resources as all model parameters are optimized. Foundation models with good performance are usually large, encompass billions of parameters. The computational overhead is even more severe in resource-constrained settings [32] and in applications that have multiple downstream tasks.
- **Storage overhead**. Distinct copies of model parameters need to be saved for different tasks. This adds significant maintenance costs to model deployment.
- **Overfitting Risk**. When the size of training datasets is limited, FPFT is prone to overfitting, where model generalize poorly to unseen data while performing exceptionally well on the training data.
- **Catastrophic Forgetting**. Fine-tuning on new data might lead to a phenomenon called "Catastrophic Forgetting" where the model lose its previously learned capabilities. If the model forgets foundational or general knowledge learned during pre-training, its ability to generalize to a broader range of scenarios or datasets can be compromised.
- **Model Stability**. Fine-tuning might compromise the stability of the model. The process might distort the patterns initially learned by the model in ways that are neither desirable nor easily understandable, resulting in a loss of consistency in the model's overall performance.

As will be discussed in greater detail in Section 2.5, Parameter-Efficient Fine-Tuning (PEFT) techniques focuses on solving some of the challenges listed above.

2.4.3 Memory-Efficient Fine-Tuning

To reduce the cost of FPFT, many recent works [33, 34, 35] target at minimizing memory consumption. Selective Fine-Tuning [33] selectively retains a subset of transient activations with non-zero gradients from the forward pass to optimizes memory consumption. This approach only reduces two-thirds of the GPU memory used. Another method called LOMO (Low-Memory Optimization) [34] proposes a new optimizer that combines the gradient computation and parameter update in one single trainig step, effectively achieving constant memory usage for gradient tensors. As a result, the FPFT of a 65B model only needs less than 192GB GPU memory using LOMO. MeZo [35] estimates the gradients with enhanced zeorth-order method using only two forward passes and fine-tunes LLMs with no memory overhead as compared to inference. MeZo is able to train a 30B parameter model on one GPU with 55GB memory using FPFT. These techniques effectively lower the cost and barrier for practitioners to conduct FPFT.

2.5 Parameter-Efficient Fine-Tuning

2.5.1 Objectives

An alternative method to fine-tune an LLM to a specific downstream task is Parameter-Efficient Fine-Tuning (PEFT). PEFT encompasses a range of methods that only updates a small subset of model parameters during model fine-tuning, with the remaining model parameters frozen.

As compared to FPFT, PEFT has much fewer parameters to modify, hence reducing the computation cost by a large margin. A recent study [36] has shown that FPFT takes around 3-5 times the time cost of Low-Rank Adaptation (LoRA) fine-tuning, which is a technique within the category of PEFT methods. PEFT also consumes much less GPU memory than FPFT. Another study [37] has shown that PEFT methods, as compared to FPFT, decrease the trainable parameters between 140 and 280 times and the training time between 32% and 44%. Moreover, PEFT enables LLMs to adapt to new tasks without catastrophic forgetting [38].

On the other hand, FPFT has a greater learning capacity and expressive power than PEFT due to its larger number of trainable parameters. In general, FPFT converges faster than PEFT [39]. Moreover, FPFT could lead to better performance than PEFT [36]. However, this phenomenon is heavily task-dependent and it is reported that PEFT performs better than FPFT on tasks with a lack of language diversity [37]. The performance difference between FPFT and PEFT decreases as the size of the training data decreases [37]. It is also shown that as the model size scales up, the performance gap between FPFT and PEFT closes and PEFT methods are able to match the good performance of FPFT [40].

From a high-level view, PEFT could be grouped into four categories: low-rank adaptation, adapter-based tuning, prompt tuning, and prefix tuning. We will dive into each one of them in the following subsections.

2.5.2 Adapter-based Tuning

In adapter-based tuning, additional lightweight layers, called adapters, are injected into the original layers of pre-trained models. During fine-tuning, only the parameters of the adapters are trainable, while the original parameters remain frozen [41, 42]. As compared to vanilla fine-tuning of deep networks where only the top layer is modified, adapter-based tuning is a more general approach to modify the architecture. There are two types of adapters: series adapters and parallel adapters.

Series adapters insert additional learnable modules sequentially inside the Transformer layers. For each Transformer layer, two adapter modules are added after the attention and FFN layers [41]. The adapter module first project the input into a smaller dimension, apply a non-linear transformation, and then project the result back to its original dimension. The dimension projection is added to effectively limit the number of additional parameters. This module can be mathematically expressed as:

$$H_{out} \leftarrow H_{out} + f(H_{out}W_{down})W_{up}$$

Here, H_{out} is the output of a specific layer. W_{down} is a matrix that down-projects the input, while W_{up} up-projects the intermediate result back to its original dimension. f is a non-linear function. A skip connection is added to avoid overfitting.

Parallel adapters [43] are proposed to unify different PEFT strategies, including Prefix Tuning, sequential adapters, and LoRA. The parallel adapters could be formulated as:

$$H_{out} \leftarrow H_{out} + f(H_{in}W_{down})W_{up}$$

where H_{in} and H_{out} are the input and output of a specific layer. Multi-head parallel adapter enhances the parallel adapter by applying parallel adapters to change the head attention outputs. Scaled parallel adapter is another variant that incorporates elements from LoRA into adapters.

2.5.3 Low-Rank Adaptation

Despite their usefulness, adapter-based methods add to inference latency as they extend the model depth. To address the limitations of adapters, Low-Rank Adaptation (LoRA) [44] was proposed, which does not introduce latency overhead during inference. LoRA was inspired by the observation that a very low "intrinsic rank" exists in pre-trained language model and fine-tuning in this low dimension is as effective as the full parameter space [45]. From a mathematical perspective, the original model parameter matrix W_0 is modified to $W_0 + \Delta W$ where ΔW is the learned update. LoRA decomposes ΔW into two low-rank matrices $A \in R^{m\times r}$ and $B \in R^{r\times n}$ such that $\Delta W = A{\cdot}B$. The rank r is much smaller than both d and k to reduce the fine-tuning computational cost. LoRA can be considered as an example of Reparameterization-based learning. During fine-tuning, only the small matrices A and B are optimized, while the large model weight matrix are kept fixed.

Since LoRA still needs to modify the parameters of the low-rank matrices for all layers of the model during finetuning, additional techniques were proposed to further improve the efficiency. LoRA-FA [46] holds the 'projection-down' matrix A frozen, while optimizing the 'projection-up' matrix B. This approach only modifies the weight in the low-rank space, so there is no need to store the full-rank input activations. LoraHub [47] extends the application of LoRA by investigating its composability for generalizing across various tasks. It integrates LoRA modules, each trained on different tasks, to achieve good performance on new, unseen tasks. One limitation of LoRA is that the rank used in the fine-tuning process is static and not subject to adjustment during training. To overcome this, Dylora is introduced [48]. Dylora is designed to train LoRA blocks across multiple ranks instead of just a single fixed rank. This method saves the effort of searching for the optimal ranks, as it organizes the representations learned by the adapters according to their respective ranks.

2.5.4 Prompt Tuning

Prompt tuning, or soft prompting, [40], is a simple yet effective method for learning prompts to re-purpose the pre-trianed language models.

Prompt tuning draws inspiration from prompt design [3], which shows promising capabilities to modulate the behavior of LLMs by simply applying prompt templates to the input sequence. These templates usually contain task descriptions and crafted example answers. Instead of requiring a different model for every task, a single LLM can perform different tasks with just different prompt templates. However, the performance of prompt design is not comparable to that of model tuning because it is difficult to manually design the optimal task description and the model's input is also limited. Although enhanced prompt design methods are proposed, such as a search algorithm over the discrete space of words [49], there is still a gap in the performance as compared to model tuning.

Prompt tuning proposed a trainable version of prompt design methods. For each downstream task, an additional k trainable tokens are added to the input text. During fine-tuning, while the entire pre-trained model is frozen, the trainable tokens are updated to learn the optimal prompt. Given the input X and prompt tokens P, the probability of the model output Y is expressed as $P_{\theta,\theta_P}(Y|[P;X])$. θ, the original model parameters, remains fixed, while θ_P are updated during fine-tuning. As compared to prompt design where prompt tokens are selected, prompt tuning uses a fixed set of special tokens as the prompt and select the optimal embeddings of these tokens through training.

As compared to adapter-based tuning and LoRA, prompt tuning generally requires much fewer additional trainable parameters. It only needs additional EP parameters, where E is the embedding dimension and P is the prompt length. Despite its simplicity, prompt tuning is especially effective, matching the performance of standard model tuning for the 11B T5 model [40]. It is reported that the gap between standard model tuning and prompt tuning closes as the model size scales up.

Here are the key hyperparameters that influence the performance of prompt tuning:

- **Prompt length**. To achieve good performance, it is generally preferrable to have the prompt length larger than one. However, The performance gain vanishes once the prompt length exceeds a certain threshold.
- **Prompt initialization**. The random uniform embedding initialization generally performs worse than the copied initialization from the model's vocabulary. Unfortunately, there is currently no effective method to interpret the learned prompt.
- **Pre-training objectives**. Some foundation models are pre-trained with objectives (span corruption e.g.) such that they do not see natural input text during training. For these pre-trained models, prompt tuning might not be as effective because the decoder input cannot be modified.

2.5.5 Prefix Tuning

Different from prompt tuning which only adds trainable vectors to the input embedding layer, prefix tuning [50] adds an array of prefix tokens to every layer in the LLM. Each set of prefix tokens correspond to a single downstream task. Prefix tuning is generally more expressive than prompt tuning and has a greater learning capability. The prefix tuning could be formulated as:

$$H_{out} = Attn(H_{in}W_Q, [P_K; H_{in}W_K], [P_V; H_{in}W_V])$$

Here, H_{out} and H_{in} are the input and output of the attention layer. P_K and P_V are the trainable prefix to the keys and the values respectively.

2.6 Summary

In this chapter, we summarize the basics of LLM inference and various techniques for fine-tuning LLMs. We also introduce the computational challenges inherent in LLM inference. In the next chapter, we will discuss strategies at different abstraction layers to address these challenges.

Chapter 3

Related Work

To address the challenges faced by LLM inference mentioned in Section 2.3, various optimization techniques are proposed. In this chapter, we aim to explain different methods for LLM inference acceleration. At a high level, the techniques for efficient LLM inference can be grouped into two categories: 1) system-level optimization, and 2) algorithm-level optimization. Both system-level and algorithm-level techniques are essential for enhancing the efficiency of LLM inference, as they address different aspects of the problem. Although our proposed solution is more related to the decoding algorithm of LLM, it is beneficial to briefly discuss system-level optimizations in Section 3.1 to offer a comprehensive view of the optimization landscape. The primary focus is on the algorithm-level optimization techniques, which are further classified into autoregressive optimization methods 3.2 and non-autoregressive optimization methods 3.3.

3.1 System-level Optimization

The performance of LLM inference can be optimized through better hardware and system infrastructure. Deja Vu [51] introduces the concept of contextual sparsity, which uses a small set of attention modules and MLP to approximate the result of a dense model. They show that accurate predictors can be trained to identify contextual sparsity dynamically and the inference speed can be increased with kernel fusion and memory coalescing. Orca [26], as mentioned in batch inference, decide the batch size dynamically using iteration-level scheduling. FlexGen [52] develops a high-throughput LLM inference engine using constrained resources, such as limited GPU memory. This method employs a linear programming approach to optimize tensor operations and integrates the storage and compute resources from different media like disk, GPU, and CPU. To increase the batch size as large as possible, weights and attention cache are quantized to 4 bits with insignificant accuracy drop. FlexGen speeds up the inference process of OPT-175B on a single 16GB GPU with a throughput of 1 tokens/s. DeepSpeed-Inference [53] proposes a multi-GPU inference solution for both dense and sparse Transformer models when they are stored in the combined GPU memory. Moreover, a heterogeneous architecture is developed to employ CPU and NVMe memory, in addition to GPU memory, leading to high-throughput inference of large models which the GPU memory only cannot contain. Flash-Decoding [54] is motivated by the problem that during inference, the batch size for long contexts needs to be small to fit in GPU memory, leading to low parallelization and low GPU utilization. As a solution, the key/values sequence is split into pieces, for which the attention is calculated in parallel. The splits are aggregated at the end to produce the final result. It is reported that Flash-Decoding is able to achieve up to 8 times faster generation for long contexts.

3.2 Autoregressive Algorithm-level Optimization

3.2.1 Early Exiting

Early exiting [55] is first introduced to reduce the latency and energy consumption of very deep neural network architectures by skipping the computation of deeper layers. This method is based on the intuition that feature representations learned at an early layer of a neural network is often sufficient to model the target distribution, especially when the input sample has a simple structure.

Internal classifiers are added to predict the exit position. The concept of early exiting is then applied to the context of LLMs [56, 57, 58]. Since these methods dynamically decide the amount of computation per request to achieve a low amortized inference cost, they are also called by adaptive computation [59].

A number of early-exit criterion are proposed, including entropy [60], softmax scores [61], convergence of intermediate predictions [62], and more complex methods that combine confident scores from consecutive layers [63]. However, the insufficient information from the internal representations might lead to sub-optimal exit decisions made [64], making it difficult to devise an efficient exit criteria for all Transformer models. The performance improvement of early-exit methods are further limited by their incompatibility with batch inference and KV cache, the two critical methods for efficient LLM serving [65]. When processing a batch of requests, the computation time of early-exit methods is bottlenecked by the most computation-heavy request. Another challenge arises with the use of KV cache, which needs to be modified if tokens generated at different time steps exit at different positions of the decoder.

3.2.2 Cascade Inference

Cascade inference is an approach motivated by the fact that different inference requests vary in complexity. This method involves assembling a collection of LLMs of different sizes, each possessing distinct capabilities, to handle requests of different difficulties. During serving, tasks considered less complex are allocated to smaller LLMs to minimize the response time, dynamically allocating computational resources based on the specific demands of each request. For instance, CascadeBERT [66] arrange a chain of models of different depths in a cascading manner, employing internal classifiers to select models to use on the fly according to request difficulties. Tabi [67] employs a similar technique to serve discriminative models. FrugalGPT [68] assign queries to different LLM APIs based on a learned policy. To further decrease the inference cost, FrugalGPT compresses and concatenates queries and cache LLM responses locally.

In general, cascade inference is an effective approach to reduce inference latency but the practical implementation is difficult due to the complexity in designing a dispatching policy that guarantees the inference quality [64].

3.2.3 Speculative Decoding

The decoding step during autoregressive LLM inference, which involves deciding the next token to generate, can be conceptualized as running a program that contains conditional branches in its control flow. Hence, speculative execution, a popular method to speed up instruction execution, can be transferred to the context of LLMs. Speculative decoding is proposed [9]. In speculative decoding, a smaller draft model is used to produce a draft output of length K autoregressively. Then the draft is verified by the larger, more capable target model. Only the draft tokens that preserve the distribution of the target model are accepted and kept, while the other tokens are discarded. The algorithm implementation is shown in Algorithm 2. It is worth noting that generation process of draft models is still in the auto-regressive manner.

This approach relies on two important intuitions: 1) a small draft model is able to produce the same output as the large target model for a significant portion of input sequences; 2) the latency of verifying a draft of length K in parallel is comparable to that of generating a single token from the target model. The first intuition underlines the potential of using smaller draft models to reduce the inference latency, given their ability to frequently generate correct tokens. The second intuition ensures that the additional computational overhead of speculative decoding remains reasonably low. Furthermore, unlike early exiting and cascade inference, speculative decoding guarantees the output quality using the verification step.

Building on the speculative decoding scheme, various studies have been conducted to further optimize its inference speed. To improve the accuracy of the draft model and its token acceptance rate, *Eagle* [16] incorporates the hidden features into the draft model's forward pass. To enhance token drafting with retrieval-augmented generation [69], *Rest* [12] introduce retrieval-based speculative decoding tailored for specific scenarios. *SpecInfer* [13] adopts a tree-based speculative inference and verification scheme, improving the diversity of speculation candidates. *Sequoia* [70] optimizes the sparse tree structure by considering the capability of the underlying hardware platforms.

Algorithm 2 Speculative Decoding

```
1: Given draft length $K$, minimum target sequence length $T$, target model $M_t$, draft model $M_d$,
   and initial prompt sequence $x_1$, ..., $x_t$
2: Initialize $n \leftarrow t$
3: while $n < T$ do
4:     for $l = 1$ to $K$ do
5:         Sample draft autoregressively $\tilde{x}_1$, ..., $\tilde{x}_l$ from $M_d$
6:     end for
7:     In parallel, verify the draft $\tilde{x}_1$, ..., $\tilde{x}_K$ using $M_t$ and obtain $K+1$ boolean values $b_1$, ...,
   $b_{K+1}$
8:     for $l = 1$ to $K+1$ do
9:         if $l \leq K$ and $b_l$ then
10:            $x_{n+l} \leftarrow \tilde{x}_l$ and $n \leftarrow n+1$
11:        else
12:            sample $x_{n+l}$ and exit the loop
13:        end if
14:    end for
15:    $n \leftarrow n+1$
16: end while
17: return $x_{1:T}$
```

However, there are still several challenges when using speculative decoding for LLM inference [11]. First, it remains difficult to develop a systematic way of designing an ideal draft model that is both computation-efficient and capable enough. Second, additional complexity, in particular for distributed systems, is introduced by maintaining multiple LLMs in a cooperative manner. Lastly, if multiple samples are required, there is extra computational overhead with speculative decoding.

3.3 Non-Autoregressive Generation

3.3.1 Overview

Given the low parallelism causes by the autoregressive decoding mechanism, one line of work to improve LLM inference latency is to find alternatives to the autoregressive decoding paradigm and produce multiple tokens in parallel. Non-Autoregressive Decoding (NAD)[1] aims to relax the constraints of token dependencies and assume a certain level of conditional independence during decoding. The early work of NAD focuses on a specific application, such as machine translation [71]. Despite its improved latency, the major challenge with parallel decoding is the quality degradation of the generated tokens caused by the lack of information on target dependency of target tokens [72]. Various methods are proposed to address this challenge. For instance, semi-autoregressive translation model [73] strikes a balance between the two paradigms, generating multiple target tokens in parallel in one step and autoregressively depending on these generated tokens for the next step. Iteration-based methods [74], on the other hand, generate tokens in parallel in a single step and refine the generated target through iterations. DePA [75] improves the translation quality by modeling conditional dependencies in the target.

Inspired by the success of NAD in the field of neural machine translation, researchers has been exploring the possibilities of NAD in the general context of text generation. In this project, we focus on the methods proposed for general-purpose NAD text generation, without the need of pre-training. At a high level, we categorize the techniques in two groups: 1) adapter-based methods, and 2) Jacobi decoding. Adapter-based methods modify the architectures of LLMs by inserting adapter-like modules and carry out fine-tuning to enable the LLMs to generate tokens non-autoregressively. On the other hand, Jacobi decoding does not need further training, reformulating the decoding algorithm based on Jacobi iteration. Both methods are explained in greater detail in Subsection 3.3.2, 3.3.3.

[1]Recently, an alternative name for NAD is proposed, which is parallel decoding. We used these two terms interchangeably throughout the report.

3.3.2 Adapter-based Method

Blockwise parallel decoding [8] modifies the original LLM by inserting a single multi-output feed-forward layer at the top to make token predictions at different positions in parallel. Then the new LLM can be re-purposed by either full-parameter fine-tuning or adapter-based fine-tuning. This method involves a three-stage non-autoregressive token generation process:

- **Predict**. k tokens (y_1, ..., y_k) are generated in parallel by the multi-output feed-forward layer. It is assumed that these k tokens are conditionally independent from each other. The prediction process takes 1 forward pass.
- **Verify**. The prompt combined with generated k tokens is passed to the original model for verification. The largest k' is found such that y_1, ..., $y_{k'}$ follows the distribution of the original model. The verification process takes another invocation of LLM inference.
- **Accept**. The target output is extended with y_1, ..., $y_{k'}$ and we go back to the **Predict** step if the generation is not finished yet. This step requires trivial computation.

The **Verify** step effectively guarantees the generated output follows the same distribution as the autoregressive version of the LLM, ensuring the prediction quality. However, in one generation step, two inference runs are required. Hence, unless the accepted token length k' is sufficiently large, the latency improvement will be limited. To address this problem, the **Verify** step and **Accept** step are combined into a single **Verify-Predict** step. In the **Verify-Predict** step, all possible target sequences from the last step are passed to the LLM, which verify and produce tokens for each sequence simultaneously. This is essentially very similar to speculative execution of branches in the instructions, where the program state is rolled back if a wrong branch prediction happens.

To further speed up the inference, the authors propose that the output sequences do not need to follow exactly the same distribution as an autoregressive model. Techniques like top-k selection, distance-based selection, and minimum block size can be used to provide an approximate output distribution.

Another adapter-based method is Medusa [11], which adopts a token generation approach similar to the Blockwise parallel decoding but uses a different token acceptance mechanism. Medusa adds and trains feed-forward layers on top of the base LLM with the parameters of the original model fixed. Like the Blockwise parallel decoding, Medusa also combines the token generation step with verification step but uses the tree-based attention mechanism from [76].

3.3.3 Jacobi Decoding

To avoid resource-intensive fine-tuning and modification of LLM structure, Jacobi decoding [77] provides an alternative implementation of NAD. In Jacobi decoding, the greedy autoregressive decoding is reformulated as solving a system of nonlinear equations, which could be done in parallel. Despite its simplicity, the fixed-point iteration methods proposed by Jacobi decoding mathematically ensures the convergence of the parallel decoding to an autoregressive output. The greedy autoregressive sampling approach can be expressed as:

$$y_i = argmax\, p_\theta(y_i|x, y_{1:i-1}) \tag{3.1}$$

Here, y_i is the generated token at position i and x is the user-provided input. We can extend Equation 3.1 to all generated tokens:

$$\begin{aligned} y_1 &= argmax\, p_\theta(y_1|x) \\ y_2 &= argmax\, p_\theta(y_2|x, y_1) \\ &\vdots \\ y_m &= argmax\, p_\theta(y_m|x, y_{1:m}) \end{aligned} \tag{3.2}$$

This system of equations has m non-linear equations with m unknowns (y_1, ..., y_m). Notice that the sequential autoregressive generation basically solves the system from top to bottom, one equation at a time by substitution. To allow for parallel solving, an initial draft solution, $\tilde{y}_1$, ...,

$TIldey_m$, is proposed and refined through iterations. In the original paper, a naive initial draft of $[PAD]$ tokens are used. The solution is guaranteed to converge and a common convergence criterion is obtaining the same results across consecutive iterations. Since this process is equivalent to Jacobi iteration, it is named Jacobi decoding. Although computational overhead is incurred due to longer input sequence length, the parallel processing power of GPUs can potentially hide the additional latency.

In practice, it is observed that Jocabi decoding often has trouble with positioning tokens correctly in the sequence, leading to the convergence speed degrading to the worst case, which is the auto-regressive generation case. To address this problem, *Lookahead Decoding* [78] improves upon this method by generating parallel n-grams and employing a caching memory pool. To capture more information while using multiple special tokens at distinct positions, *PaSS* [79] trains additional tokens with embedding layers for parallel decoding. To enhance token drafting with retrieval-augmented generation [69], *Rest* [12] introduce retrieval-based Jocabi decoding tailored for specific scenarios.

3.3.4 Limitations of NAD

In conclusion, the key challenge of NAD is how to balance the trade-off between prediction quality and speed. While methods like Jacobi decoding are guaranteed to generate the same results as the autoregressive decoding process, they might not improve the inference latency significantly enough. Blockwise parallel decoding attempts to relax the quality requirement by adopting approximate acceptance schemes, but the prediction accuracy drops as a result. Moreover, the adapter-based methods need to modify the model architectures, making them less effective in cases where the model is not directly accessible or the training resources are limited. Hence, we think a new NAD method is needed to address these challenges.

3.4 Summary

In this chapter, we discuss various techniques to optimize LLM inference performance. We believe NAD is one of the most promising methods to address the limited parallel computation in LLM inference. To overcome the challenges faced by the current NAD method, we will propose a new NAD approach in the next chapter.

Chapter 4

Parallel Prompt Decoding

In this chapter, we introduce Parallel Prompt Decoding (*PPD*), a method designed to enable non-autoregressive multi-token generation using a variant of prompt tuning. We cover the core components of *PPD*, its training procedures, and the inference implementation. Key innovations include the strategic placement of prompt tokens, the introduction of Ensemble Prompt Tokens (EPTs), and advanced training techniques such as random insertion and knowledge distillation. Our approach provides insights into alternative uses of prompt embeddings, exploring their potential to enhance representation power by filling in missing conditional dependency information in non-autoregressive generation.

4.1 Prompt Tokens

PPD trains embeddings for prompt tokens[1] rather than developing a separate model. Thus, the prompt tokens are the key component of *PPD* to realize multi-token generation. Initially introduced in [80] to adapt LLMs for specific tasks, prompt tokens are typically prepended to the input, with outputs generated in an autoregressive manner. Consequently, the produced logits corresponding to the prompt tokens are usually ignored, as show in Figure 4.1, due to the lack of semantic interpretation.

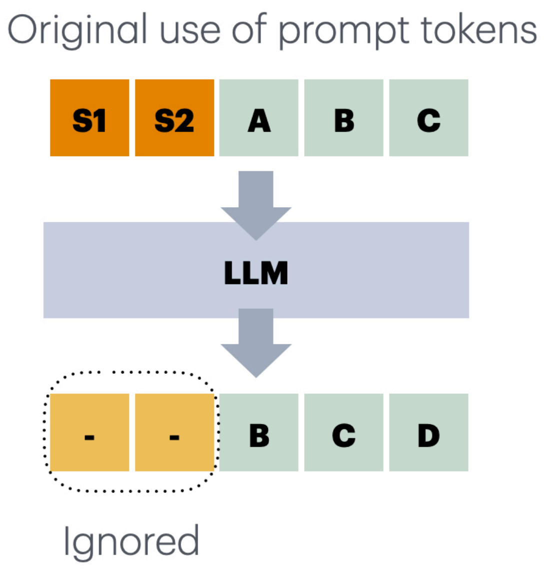


Figure 4.1: 'S1' and 'S2' are prompt tokens prepended to the input sequence "A B C". The tokens labeled '-' correspond to logits generated from the prompt tokens. These logits are ignored and not used.

[1]In this report, a "prompt token" denotes the special token with separately trained embeddings to perform parallel prediction.

In this work, we propose a novel approach of utilizing prompt tokens by strategically positioning them at locations where tokens are anticipated to be generated, allowing an innovative approximation approach for non-autoregressive generation. Figure 4.2 illustrates the basic use of prompt tokens.

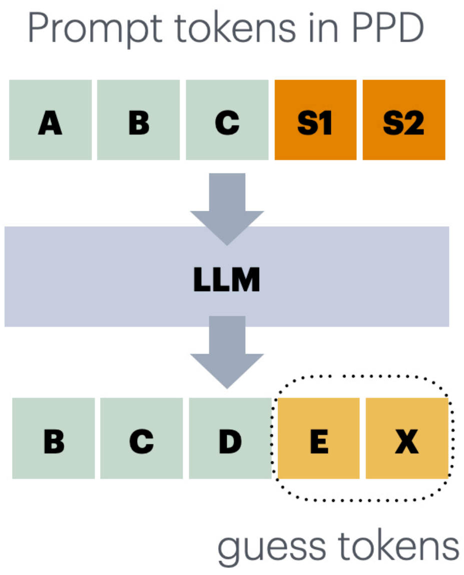


Figure 4.2: In this example, we add prompt tokens 'S1' and 'S2' at positions where we anticipate generating specific tokens. For instance, 'S1' stands in for the missing letter 'D', and 'S2' for the missing letter 'E'. The goal is to train the embeddings of 'S1' and 'S2' so they can approximate these missing tokens through the LLM's decoder layers by taking in semantic information from the input tokens.

This strategic placement allows for an innovative approximation approach. For adapter-based parallel decoding techniques [8, 11] that presume complete conditional independence among tokens decoded in a single step, the exact conditional probability p is approximated by:

$$p(y_{i+k+1}|x, y_{1:i+k}) = p_\theta(y_{i+k+1}|x, y_{1:i})$$

where $k > 0$ indicates the token distance, x is the given input tokens, and $y_{1:i}$ is the generated sequence with length i.

However, we observe that as k increases, the gap between the actual probability and its approximation expands, primarily due to the absence of relevant conditional dependencies. We propose that prompt tokens can bridge this gap by more accurately modeling the conditional probability as:

$$p(y_{i+k+1}|x, y_{1:i+k}) = p_\theta(y_{i+k+1}|x, y_{1:i}, t_{i+1:i+k})$$

where t_i is the prompt token that is i token distance away. Through this forward pass in the decoder layers, these causally linked prompt tokens facilitate the flow of information along the sequence of speculative tokens, thus restoring the conditional probability.

4.2 Ensemble Prompt Tokens

Inspired by prompt ensembling [80], which uses multiple prompts to generate diverse responses and averaging these to derive a single answer, we introduce the concept of ensemble prompt token (EPT). This additional abstraction allows us to decouple each prompt token from the fixed embedding dimension. For every prompt token, there exist multiple corresponding EPTs, each with its distinct embedding. Figure 4.3 illustrates the use of EPTs.

We modify the attention mask to ensure that each n^{th} EPT only depends on the corresponding n EPTs from preceding prompt tokens. This selective visibility is maintained for both training and inference, where the speculative token for each prompt token is determined by averaging the

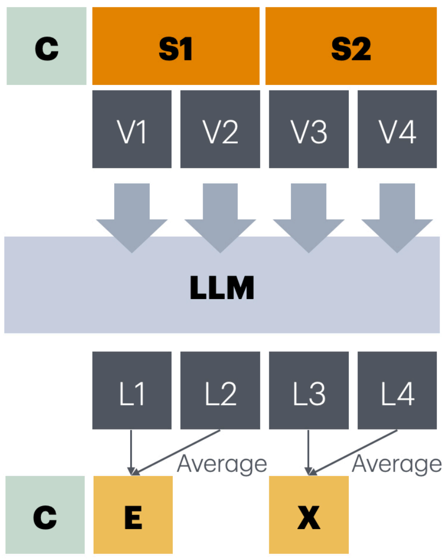


Figure 4.3: Here, each prompt token is associated with two distinct embeddings rather than just one. Specifically, 'V1' and 'V2' are the embeddings of EPTs for the prompt token 'S1', while 'V2' and 'V3' are for 'S2'. To determine the tokens corresponding to 'S1', we average the logits 'L1' and 'L2', which are generated from 'V1' and 'V2', respectively.

logits of its EPTs. The use of EPTs not only enables direct and flexible control over the trainable parameters, but also leads to an increase in prediction accuracy. The approximate probability is expressed as:

$$p(y_{i+k+1}|x, y_{1:i+k}) = \frac{1}{n}\sum_{j=1}^{n} p_\theta(y_{i+k+1}|x, y_{1:i}, v^j_{i+1:i+k})$$

where v^j_{i+m} denotes the j^{th} EPT at a token distance of m.

The space complexity of the trainable parameters is represented as $O(n \cdot d_{\text{emb}})$, where n is the number of virtual tokens per prompt token and d_{emb} is the embedding dimension. In practice, n is 3 to 4 orders of magnitude smaller than d_{emb}, so the space complexity simplifies to $O(d_{\text{emb}})$.

4.3 Training

The correct training approach for prompt tokens is crucial for the effectiveness of *PPD*. During training, only the embeddings of prompt tokens are changed, with the parameters of the original LLM[2] remaining frozen. We adopt the following two training techniques:

4.3.1 Random Insertion of Prompt Tokens

Randomly inserting prompt tokens throughout the input sequence reduces contextual bias associated with appending them only at the end. [3] This bias tends to restrict the learning of prompt tokens to a narrow vocabulary range, such as `<eos>` and punctuation. Random placement helps broaden the predictive capacity of prompt tokens.

Moreover, the random insertion of prompt tokens allows *PPD* to work effectively across a variety of context window lengths.

4.3.2 Knowledge Distillation

To align the predictive behavior of prompt tokens with the original LLM, we employ the knowledge distillation approach. Instead of using hard labels, prompt tokens are trained against the logits

[2]The "original LLM" denotes the LLM to accelerate.

[3]Since we only require the logits generated by the prompt tokens, this method is essentially equivalent to randomly truncating the input sequence and appending prompt tokens at the end.

produced by the original LLM. The loss function is formulated as:

$$L_{PD} = \frac{1}{N}\sum_{i=1}^{N} D_{KL}(P_i \parallel Q_i) \cdot \alpha^i \tag{4.1}$$

where P_i represents the predicted probability distribution of the i^{th} prompt token, Q_i denotes the corresponding probability distribution generated by the original LLM, and α is the decay ratio. This decay ratio is applied to balance the influence of predictions at different token positions, following the methodology used in Medusa [11].

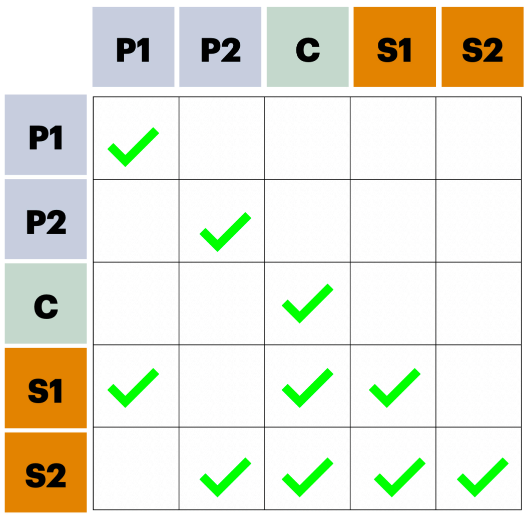
Additionally, knowledge distillation serves another crucial functionality: it can generate training datasets for *PPD* when the original model's training dataset is unavailable, inaccessible, or too small. By leveraging the logits produced by the original LLM, we can create a sufficiently large training set that aligns with the original LLM to effectively train *PPD*.

4.4 Extensions

In this section, we discuss the possible extensions to *PPD*. We examine the effectiveness of these extensions in Chapter 6. We only include the extensions in the final method used if they prove to be helpful.

4.4.1 Prefix Tuning + Prompt Token

Prefix tuning [50], similar to prompt tuning, provides a parameter-efficient approach to finetune a pretrained model. Unlike prompt tuning, it modifies the KV cache of every attention layer by prepending trained vectors. We hypothesize that the combination of prefix tuning and prompt tokens can lead to greater learning capacity and higher prediction accuracy. This hypothesis is based on the intuition that prompt tokens should see a different context than the input tokens when predicting long-range tokens. For example, if the input sequence is "Once upon a time," then enhancing the input with a prompt template might provide more suitable semantic context for long-range prediction. An enhanced input like "Predict the next-next token. Once upon a time" might empower the prompt token to predict the correct next-next token. Prefix tuning serves as the prompt template to enhance the hidden states visible to the prompt tokens.



	P1	P2	C	S1	S2
P1	✓				
P2		✓			
C			✓		
S1	✓		✓	✓	
S2		✓	✓	✓	✓

Figure 4.4: 'P1' is the prefix token for the prompt token 'S1' and 'P2' for 'S2'. 'C' is the input token. The green tick means visibility during attention calculation. For instance, 'S1' can see 'P1' but cannot see 'P2'. 'C' does not see any prefix tokens so the generated output corresponding to 'C' is not altered by the use of prefix tuning.

To retain the original model's distribution, we modify the attention mask so that prefix tokens are only visible to prompt tokens. This ensures that we can generate outputs that preserve the original model's distribution. We posit that prompt tokens at different positions should see different contexts so we allow a prompt token at a specific position to see a distinct set of prefix tokens, as shown in Figure 4.4.

4.4.2 Custom Decoding Heads + Prompt Token

It has been demonstrated that a fine-tuned decoding head alone can effectively predict long-range tokens [8, 11]. Thus, we hypothesize that combining a separately fine-tuned decoding head with prompt tokens might further enhance the potential of *PPD*. As shown in Figure 4.5, we trained a separate decoding head to transform only the hidden states of prompt tokens into logits. A key distinction from Medusa is that this decoding head is responsible for generating tokens at multiple positions, rather than just one.

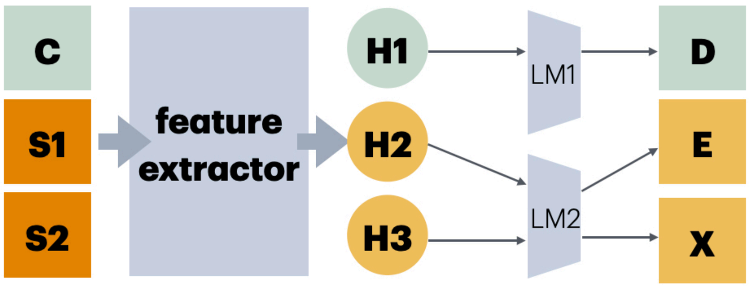


Figure 4.5: Custom decoding head with *PPD*. The feature extractor refers to the LLMs without the decoding heads. 'H1' is the generated hidden state for the input token 'C'. 'H2' is the hidden state for prompt token 'S1' and 'H3' for 'S2'. 'LM1' is the original LLM's decoding head and it takes in the hidden states of input tokens. 'LM2' is the custom decoding heads for *PPD* and only takes in the hidden states of prompt tokens.

We propose two training methods. In the first method, the custom decoding head and prompt tokens are trained together from scratch in a single stage. In the second method, the prompt tokens are initially trained for 2 epochs, followed by training both the prompt tokens and the decoding head with a smaller learning rate in a two-stage process.

4.4.3 Attention Masking for EPTs

In this report, we proposed a specialized attention mask for EPTs to achieve the effect of prompt ensemble. However, there are alternative masking strategies available. Here, we describe and compare three types of attention masks that we implemented and experimented with.

Ensemble Attention Masking

The ensemble attention masking is the masking strategy we previously described. In this approach, EPTs are divided into n disjoint groups, where n is the number of EPTs per prompt token. All k^{th} EPTs across prompt tokens are placed in the same group. An EPT v in group i can only attend to EPTs that meet the following two criteria: 1) they must belong to group i, and 2) their position ids must be smaller than the position id of v. Since this masking strategy effectively averages the results of disjoint groups of EPTs, we refer to it as the "ensemble attention masking". Figure 4.6 provides an example of the ensemble attention masking.

Decoder-like Attention Masking

Decoder-like attention masking is a simple strategy where EPTs can only attend to EPTs with smaller position ids. This results in a triangular-shaped attention mask, similar to the one used in decoder layers, hence the name "decoder-like attention masking". Figure 4.7 provides an example of this masking strategy.

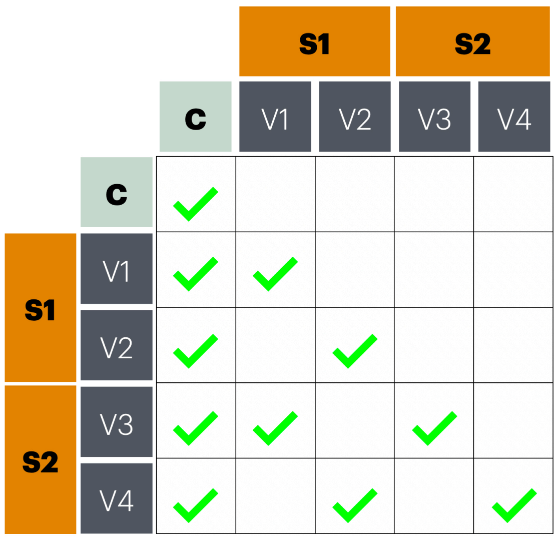


Figure 4.6: Ensemble Attention Mask. 'C' is an input token. 'V1' and 'V2' are the EPTs for prompt token 'S1' and 'V3' and 'V4' for 'S2'.

Encoder-like Attention Masking

In encoder-like attention masking, an EPT corresponding to a prompt token P can attend to all EPTs with smaller position IDs as well as all EPTs associated with P. This allows EPTs to see both preceding and succeeding EPTs, similar to the token visibility in an encoder layer, hence the name "encoder-like attention masking". Figure 4.8 illustrates this masking strategy.

4.4.4 Aggregation Method for EPTs

In addition to simply averaging the logits from EPTs, we explored more advanced aggregation methods. For instance, we applied learned weights to aggregate the logits. The final logit p can be expressed as:

$$p = \sum_{i=1}^{n} w_i \cdot p_i$$

where n is the number of EPTs and w_i is the learned scalar weight for the i^{th} EPT.

4.4.5 Multi-exit Ensemble

While using EPTs for prompt ensemble improves prediction accuracy, it also increases input length, resulting in higher computational overhead and forward pass latency. To address this, we propose the use of a multi-exit ensemble method. In multi-exit ensemble, the hidden states of a prompt token from the last k decoder layers are extracted and averaged to produce the final hidden state, which is then decoded by the decoding head into a guess token[4], as illustrated in Figure 4.9. This approach achieves prompt ensemble without the associated computational costs.

The hypothesis is that taking the hidden states from the last few decoder layers for ensemble might work because these layers capture increasingly abstract and high-level representations of the input sequence. By averaging the hidden states from multiple layers, we can combine diverse but complementary information, leading to a more robust and accurate final hidden state. Additionally, since the final layers are closest to the output, they are more likely to contain refined and contextually relevant information, making the ensemble more effective.

[4]A "guess token", also referred to as a "candidate token", is a draft token generated from a prompt token from the forward pass of the LLM.

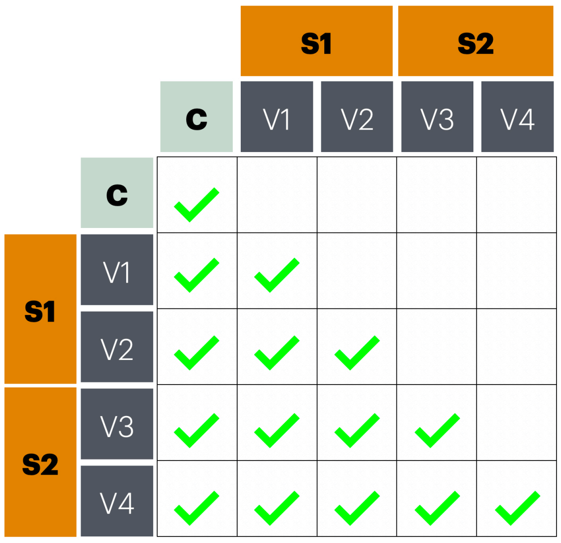


Figure 4.7: Decoder-like Attention Mask. 'C' is an input token. 'V1' and 'V2' are the EPTs for prompt token 'S1' and 'V3' and 'V4' for 'S2'.

4.5 Inference

During inference, our method integrates three substeps into a single decoding step, following the **guess-and-verify** strategy:

1. **candidate prediction.** We generate guess tokens, which are to be verified in the next decoding step.
2. **candidate verification.** We verify the guess tokens from the previous decoding step based on the outcome of the LLM forward pass.
3. **candidate acceptance.** We add newly accepted tokens to the input and update KV cache accordingly.

We now explain the details of each substep.

4.5.1 Candidate Prediction

Candidate prediction shows two distinct patterns for the initial and subsequent predictions. For the initial prediction, with no guess tokens, there is only one candidate continuation requiring the appending of prompt tokens. Figure 4.2 illustrates the prediction phase of this initial prediction. Guess tokens 'E' and 'X' are produced from the logits generated from prompt tokens 'S1' and 'S2' respectively.

In the prediction phases following the initial prediction, guess tokens are generated after each accepted candidate continuation. Since the length of accepted guess tokens can only be determined in the subsequent verification step, multiple candidate continuations are fed into the LLM. For example, Figure 4.10 illustrates the second prediction phase following the initial prediction phase shown in Figure 4.2. In this example, 'E' and 'X' are the guess tokens generated at token distances[5] of 1 and 2, respectively. There are three possible correct candidate continuations:

1. Both 'E' and 'X' are correct predictions.
2. 'E' is a correct prediction, but 'X' is incorrect.
3. Neither 'E' nor 'X' are correct.

Thus, three distinct candidates, each appended with a unique set of prompt tokens, are passed into the LLM for verification and to predict the next set of guess tokens.

[5]The "token distance" is the number of tokens between the last input token and the predicted token.

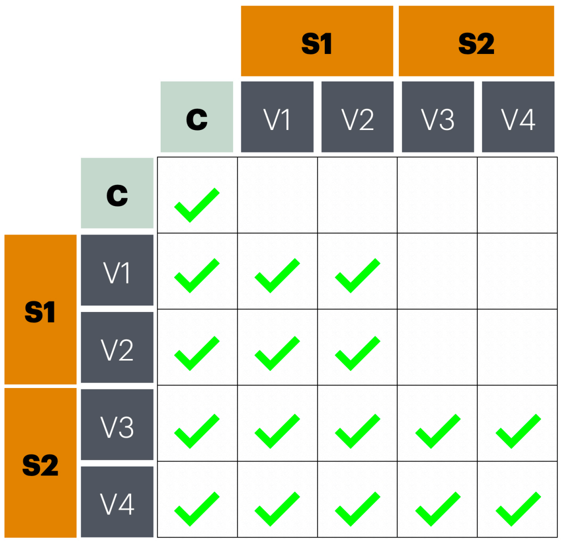


Figure 4.8: Encoder-like Attention Mask. 'C' is an input token. 'V1' and 'V2' are the EPTs for prompt token 'S1' and 'V3' and 'V4' for 'S2'.

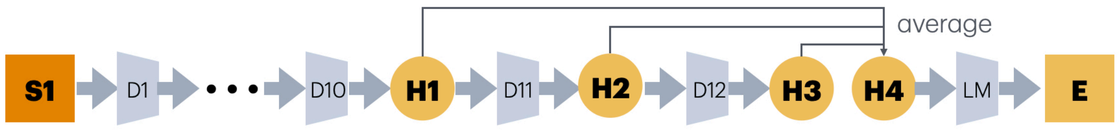


Figure 4.9: Multi-exit ensemble. 'D1', 'D10', 'D11', and 'D12' are the decoder layers in order. 'S1' is a prompt token and 'H1', 'H2', 'H3' are the corresponding hidden states from the last 3 decoder layers. 'H4' is obtained from averaging these 3 hidden states. The decoding head 'LM' translates 'H4' into a token 'E'.

4.5.2 Candidate Verification

Candidate verification and candidate prediction are completed in a single forward pass of the LLM. Two different verification schemes, exact matching [78] and typical acceptance [11], are implemented.

Exact Matching

Exact matching [78, 8] aims to achieve lossless acceleration of the LLM by ensuring that the generated outputs are identical to those of the original LLM. In speculative decoding, this verification process involves passing guess tokens to the LLM, which then produces an output token for each token. The output token corresponding to the last guess token is progressively checked to ensure it exactly matches the guess token. We adopt this verification scheme for *PPD*. See Algorithm 3 for the detailed scheme.

For example, Figure 4.10 illustrates the verification process using exact matching. The verification outcomes for the three candidates are summarized as follows:

- **Candidate 1.** Candidate 1 is accepted because it does not contain any guess token.
- **Candidate 2.** Candidate 2 is accepted because the guess token 'E' matches 'E' generated from the input token 'D'.
- **Candidate 3.** Candidate 3 is rejected because the guess token 'X' does not match 'F' generated from the input token 'E'.

Typical Acceptance

We adopt typical acceptance [11] to address the inefficiencies of rejection sampling in speculative decoding, especially at higher sampling temperatures. Traditional rejection sampling is designed

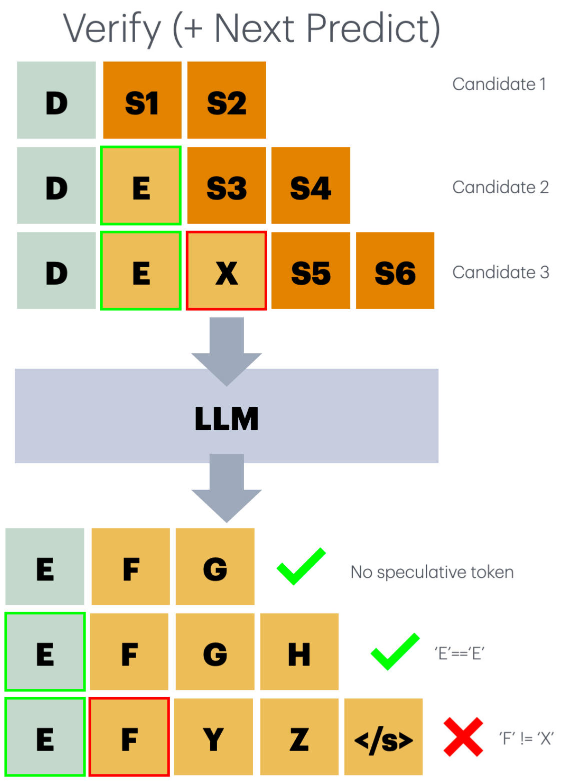


Figure 4.10: Verify and Predict combined in one single forward pass.

to produce diverse outputs matching the original model's distribution, but it becomes inefficient as the sampling temperature increases. If the draft model is identical to the original and uses greedy decoding, all outputs will be accepted, which is desirable. However, at high temperatures, the independent sampling of draft and original models leads to a high probability of misalignment between distributions of the draft and original model, which leads to unjustified rejection of drafts.

In contrast, typical acceptance leverages the prediction probabilities from the original model to select plausible candidates that are not highly improbable. Typical acceptance is based on 2 key insights:

1. Tokens with relatively high probability are usually meaningful.
2. When entropy is high, diverse plausible continuations should be accepted.

This method sets an acceptance threshold based on the probability distribution of the original model. Specifically, a candidate sequence $x_{n+1:n+K+1}$ given input sequence $x_{1:n}$ is accepted if:

$$p_o(x_{n+k} \mid x_{n+1:n+K+1}) > \min(\epsilon, \delta \exp(-H(p_o(\cdot \mid x_{1:n}))))$$

where p_o is the probability generated by the original LLM, ϵ, δ are hyperparameters provided by the user, and $H(\cdot)$ is the entropy function.

After the forward pass of the original LLM, the first token is verified using greedy decoding and hence, always accepted. Subsequent tokens are evaluated using the typical acceptance criterion.

4.5.3 Candidate Acceptance

Based on the outcomes of the verification stage, the longest accepted candidate is chosen as the final correct continuation and added to the output sequence. The corresponding guess tokens are saved for the next decoding step. The KV cache is then updated to reflect the state of the current output sequence. For instance, candidate 2 is accepted in Figure 4.10. The new output sequence will be "D E F" while the guess tokens are "G H". Note that 'F' is accepted despite not being verified because it is generated from a correct continuation. The KV cache now contains the keys and values of "D E F".

Algorithm 3 Exact Matching Verification

```
1: Given candidate continuations $c_1 ... c_K$, target model $M_t$
2: Initialize Set $accepted \leftarrow \{\}$
3: In parallel, pass the candidates $c_1 ... c_K$ to $M_t$ and obtain $K$ output sequences $s_1$, ..., $s_K$
4: for $k = 1$ to $K$ do
5:     for $l = 1$ to $K + 1$ do
6:         Let the sequences of token for $c_k$ be $cseq_{1:l_k}$ where $l_k$ is the length of the sequence
7:         Let the sequences of token for $s_k$ be $tseq_{1:l_k}$ where $l_k$ is the length of the sequence
8:         if $cseq_{2:l_k} == tseq_{1:l_k-1}$ then
9:             Add $c_k$ to $accepted$
10:        end if
11:    end for
12: end for
13: return $accepted$
```

4.5.4 Custom KV Cache Implementation

PPD requires a different implementation of the KV cache compared to conventional inference. In conventional inference, the KV cache is simply appended to, whereas *PPD* needs to modify the KV cache. This is because, during the forward pass of the LLM, the key and value states of guess tokens are also added to the KV cache, but only a subset needs to be retained.

The Hugging Face implementation of the KV cache uses a list of tensors. However, a list is not optimized for parallel modifications. Therefore, we preallocate a tensor with an additional dimension to serve as the KV cache. Modifications can then be efficiently performed using advanced indexing of the tensor. The tensor is kept on GPUs, while its metadata, such as cache length, is kept on the CPU for maximum efficiency. We optimize performance by pruning the KV cache at the conclusion of each decoding step rather than right after each attention computation, thereby reducing read/write overhead.

Chapter 5

Dynamic Sparse Tree

In this chapter, we explore the use of customized tree attention during inference and introduce a hardware-aware dynamic sparse tree to optimize the effectiveness of *PPD*. We adapt the shape of the Dynamic Sparse Tree based on hardware constraints and optimize the prediction power of *PPD*. By formulating a constrained optimization problem, we propose a systematic solution to maximize the expected acceptance length.

5.1 Top-k Candidate

In Chapter 4, we discuss candidate generation, where only the most probable candidate at each position is used. However, to leverage the entire probability distribution predicted by a prompt token, the top-k most likely tokens can be selected as candidates for a single position. Previous studies [11] have found that sampling multiple candidates leads to increased acceptance lengths during decoding. Hence, during inference, *PPD* selects multiple candidates from a single prompt token. As shown in Figure 5.1, three guess tokens are derived from a single prompt token, and if any of the three tokens are accepted, a correct prediction is achieved at this position.

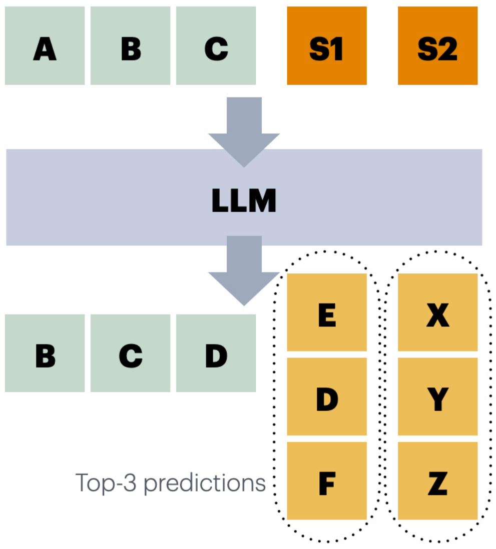


Figure 5.1: Top-3 candidate prediction. 'E', 'D', 'F' are the top-3 tokens predicted from 'S1' while 'X', 'Y', 'Z' are the top-3 tokens predicted from 'S2'.

5.2 Tree Attention

The top-k candidate generation results in a tree-structured input for verification, as shown in Figure 5.2. In this candidate tree, each node corresponds to a possible valid candidate continuation.

For instance, the node 'B' in Figure 5.2 represents the continuation "A B". Nodes at the same depth are generated from the same prompt token at a specific token distance. For example, nodes 'B' and 'b' at depth 1 are the top-2 predictions from the prompt token at 1 token distance. For verification, the candidate tree is flattened in a level-by-level order and passed to the LLM. The resulting logits are then used to decide the accepted candidate based on either exact matching or typical acceptance.

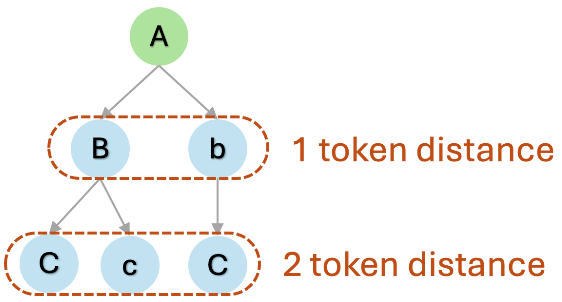


Figure 5.2: Tree-structured input. 'A' is an accepted token. 'B' and 'b' are the guess tokens derived from the top-2 predictions of a prompt token at a 1 token distance. 'C' and 'c' are the guess tokens derived from the top-2 predictions of a prompt token at a 2 token distance. The linearized input sequence for this tree is "A B b C c C".

However, the increased number of candidates leads to computational overhead. To minimize the computation demands, *PPD* utilizes a specialized tree attention to process multiple candidates within a single decoding step efficiently, similar to other parallel decoding methods [11, 13]. In tree attention, input tokens are organized into a tree structure, with each node representing a token, and each path corresponding to a candidate sequence. A token attends only to its ancestors and itself, with the attention mask adjusted accordingly. Nodes at the same tree depth i correspond to the top-k predictions of the i_{th} prompt token. This structure enables parallel processing of multiple candidates without expanding the batch size.

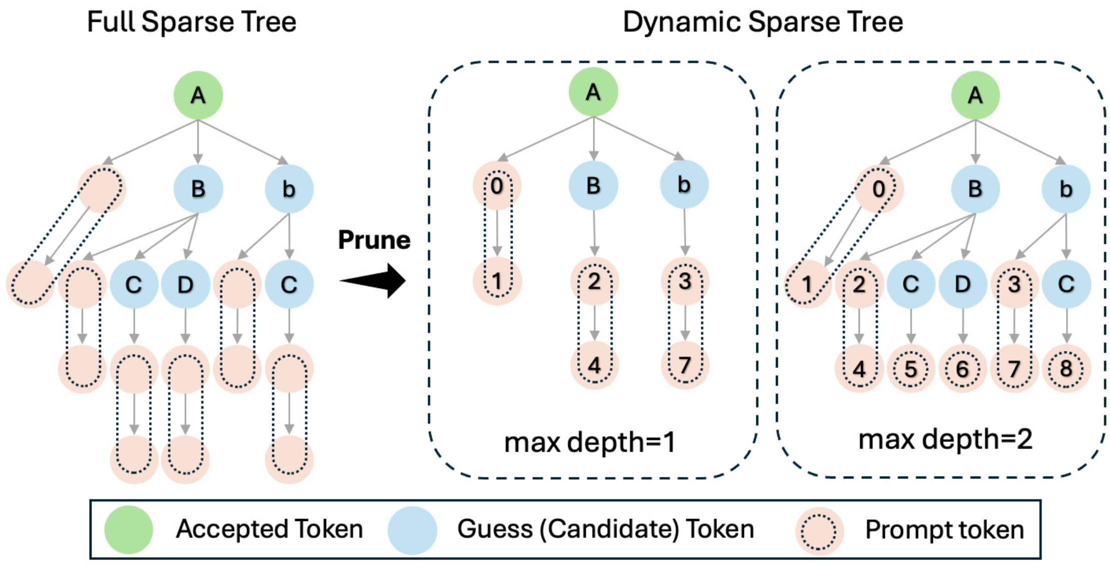


Figure 5.3: Full Sparse Tree vs Dynamic Sparse Tree.

5.3 Full Sparse Tree: A Naive Implementation

Various tree structures have been proposed [76, 11, 70]. In particular, *PPD* employs a sparse tree [11, 70], designed to be unbalanced to prioritize candidates with higher prediction accuracy. One key distinction from the sparse tree used in previous work is the appending of a sequence of prompt tokens to each tree node as shown in Figure 5.3. Due to the hardware constraint, the size of the sparse tree, which is the sum of candidate nodes and prompt token nodes, must be limited to prevent significant forward pass latency overhead. Therefore, to maximize the amortized acceptance length across the decoding steps, a careful balance between these two types of nodes is essential.

The acceptance length of the current decoding step increases with the number of candidate nodes. On the other hand, the maximum tree depth for the next decoding step is equal to the number of prompt tokens at the currently accepted candidate node, as each tree level directly corresponds to an individual prompt token. Each decoding step accepts precisely one candidate path, corresponding to a single candidate node; in the worst-case scenario, the root node - which represents the one predicted token from the original LLM - is guaranteed to be accepted.

A straightforward solution is to append the maximal number of prompt tokens to each candidate node in the sparse tree. This solution is simple to implement, and we call it the Full Sparse Tree method. The left figure in Figure 5.3 shows an example of a full sparse tree.

5.4 Dynamic Sparse Tree: Be Hardware-Aware

Despite its simplicity, the Full Sparse Tree method suffers from inflexibility and limited adaptability to hardware with restricted computational resources. To address these issues, we propose the Dynamic Sparse Tree method, which adapts the tree shape based on the available hardware resources. Instead of appending a uniform number of prompt tokens to every candidate node, we allocate them based on each candidate's probability. This approach is based on the observation that candidates are accepted at different probabilities; thus, reducing the number of prompt tokens for less likely candidates can maximize the overall number of tree nodes. Consequently, the maximum depth of the tree varies with each decoding step, making the sparse tree dynamic, as its structure changes each step based on the number of prompt tokens at the accepted candidate.

A dynamic sparse tree consists of multiple trees with different maximum depths for candidate tokens. The right figure in Figure 5.3 illustrates an example of a dynamic sparse tree with a total of 2 trained prompt tokens. For each candidate token, we append either 1 or 2 prompt tokens. At least one prompt token is appended to each guess token to ensure there are guess tokens generated at this decoding step. The maximum number of prompt tokens appended is equal to the total number of prompt tokens trained. In this example, for the dynamic sparse tree at a maximum depth of 2, if the continuation `"A B C"` is accepted, then the dynamic sparse tree at the next decoding step will have a maximum depth of 1 since the corresponding candidate token has only 1 prompt token. On the other hand, all possible continuations for the dynamic sparse tree at a maximum depth of 1 lead to a maximum depth of 2 in the next decoding step because all candidate tokens have 2 prompt tokens appended.

5.5 Customized Attention Mask for Tree Attention

During inference, we linearize the dynamic sparse tree into 1 input sequence. To preserve the tree-structured visibility during attention calculation, we adopt a customized attention mask. Unlike the default causal attention mask which has a triangular shape, our customized attention mask has a special shape that depends on the tree structure as shown in Figure 5.4. As the dynamic sparse tree adjusts its shape, the attention mask also changes correspondingly.

5.6 Construction Algorithm for Dynamic Sparse Tree

5.6.1 Problem Formulation

We aim to construct a dynamic sparse tree which maximizes the amortized number of tokens generated with a limit on the number of candidate nodes and prompt token nodes. We first define the tree construction algorithm as a constrained optimization problem and then propose a pruning algorithm to solve it.

Definition 5.6.1. Let m be the maximum number of prompt tokens. The dynamic sparse tree T can exist in m distinct states, each denoted by T_k for the k_{th} state s_k. Define $\mathrm{C}(T_k)$ as the subtree

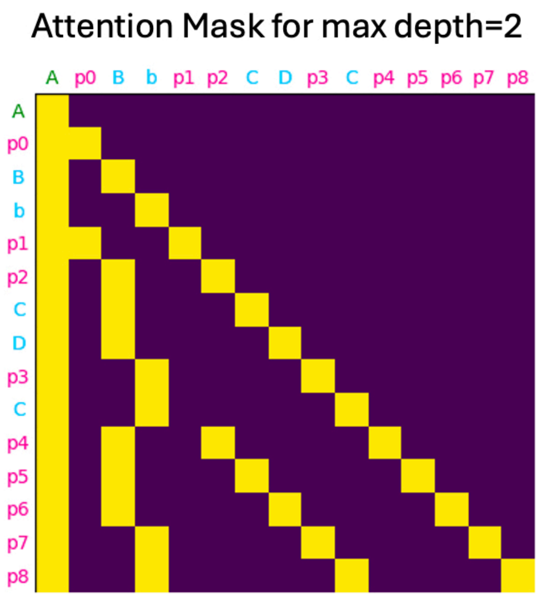


Figure 5.4: Customized Attention Mask for the Dynamic Sparse Tree from Figure 5.3 at max depth=2. "p{k}" refers to the prompt token with the label "k". Yellow block means the token is visible while purple block means the token is invisible.

of T_k consisting only of candidate nodes, excluding any prompt token nodes. The maximum depth of subtree $\mathrm{C}(T_k)$ is k.

Proposition 5.6.1. For a dynamic sparse tree state T_k with candidate nodes v, where each v follows a path $\mathrm{Path}(v)$ from the root, and the acceptance probability p_k at each path position k, the expected number of tokens $f(T_k)$ generated is given by:

$$f(T_k) = \sum_{v \in \mathrm{C}(T_k)} \prod_{i \in \mathrm{Path}(v)} p_i$$

where $\prod_{i \in \mathrm{Path}(v)} p_i$ represents the contribution of node v to the expected number of tokens.

5.6.2 Solve the Optimization Problem

We then propose an approximation of the amortized number of tokens generated by considering the tokens generated at the current and the next decoding step.

Proposition 5.6.2. The expected total number of tokens $F(T_k)$ generated for the dynamic sparse tree state $F(T_k)$ at the current and the next decoding step is given by:

$$F(T_k) = f(T_k) + \sum_{i=1}^{m} p(s_i|s_k) f(T_i)$$

where $p(s_i|s_k)$ represents the state transition probability from state s_k to state s_i.

We are now ready to introduce Proposition 5.6.3, which we use in the pruning algorithm.

Proposition 5.6.3. For a dynamic sparse tree state T_k with candidate subtree $c_k = \mathrm{C}(T_k)$, the change in expected total tokens $F(T_k)$ due to the removal of a prompt token at candidate node c is given by:

$$\Delta F = p(c) \cdot (f(T_i) - f(T_{i-1}))$$

where $p(c)$ is the acceptance probability of candidate c, i denotes the number of prompt tokens prior to removal. We assume that $i > 1$.

To construct an approximately optimal dynamic sparse tree with specified numbers of candidate and prompt token nodes, the process includes:

1. **Optimal Candidate Trees:** Constructing trees using only candidate nodes at varying depths, employing the algorithm from Medusa [11] and Sequoia [70] to maximize $f(T_k)$ as stated in Proposition 5.6.1.

2. **Appending Prompt Tokens:** Attaching the maximum allowable prompt tokens to each candidate node from the first step.

3. **Greedy Removal of Prompt Tokens:** Removing prompt tokens greedily to minimize ΔF (Proposition 5.6.3), continuing until the desired prompt token count is reached.

We now introduce the formulation of the real amortized number of tokens generated.

Proposition 5.6.4. The amortized number of tokens $R(T_k)$ generated for the dynamic sparse tree state $F(T_k)$ is given by:

$$R(T) = \sum_{i=1}^{m} p(s_i) f(T_i)$$

where $p(s_i)$ is the steady-state probability of state s_i, and f is the function defined in Proposition 5.6.1.

5.6.3 Hardware-Aware Tree Construction

All the probabilities used in above can be approximated on a validation dataset. The dynamic sparse tree construction algorithm can now be formulated as finding the dynamic sparse tree T with n_c candidate tokens and n_p prompt tokens to maximize $R(T)$:

$$c(n_c, n_p) = \max_{T, |C(T)|=n_c, |T|=n_c+n_p} R(T)$$

For a fixed tree size n, we explore all combinations of n_c and n_p where $n = n_c + n_p$, to identify the dynamic sparse tree that maximizes $R(T_k)$. To determine the optimal tree size n, we define two key functions:

1. Acceptance length $\tau(n)$ (hardware-independent),

2. Forward pass latency $L_{fp}(n)$ (hardware-dependent).

The theoretical speedup ratio for a tree of size n is given by:

$$\text{Speedup}(n) = \frac{\tau(n)}{L_{fp}(n)}$$

Both functions are approximated using a validation dataset. Notably, function 1) requires a single evaluation, whereas function 2) needs evaluation on each type of hardware. We then find the value of n that maximizes Speedup(n).

Chapter 6

Experiments

In this chapter, we evaluate *PPD* across diverse benchmarks using a range of performance metrics. Through extensive experiments across LLMs from MobileLLaMA to Vicuna-13B, *PPD* achieves notable speedups of up to 2.49x, while maintaining an exceptionally low runtime memory overhead of just 0.0004%. Additionally, we demonstrate that *PPD* achieves 28% higher prediction accuracy and exhibits strong synergistic capabilities with other speculative decoding methods.

6.1 Evaluation Setup

6.1.1 Models and testbeds

We conducted all the experiments using MobileLLaMA-1.4B [14], Vicuna-7B and Vicuna-13B [15]. We used 3 prompt tokens and 1 EPT per prompt token for all inference experiments. The inference throughputs of the models are evaluated on a single NVIDIA A100 GPU with 40GB of memory and a GeForce RTX 4090. Previous parallel decoding methods test the inference throughput using a batch size of 1 [11, 78] so we followed the same setting for inference. For the baseline models, we chose the Huggingface's implementation of Vicuna models and the leading parallel decoding methods including Medusa [11], *LOOKAHEAD DECODING* [78], REST [12], and PLD [81].

6.1.2 Training

All the trainable parameters of the original LLM were frozen. The v1.3 versions of the Vicuna models (7B, 13B), fine-tuned from Llama-2 models with a sequence length of 2048, were utilized. During training, we used 4-bit quantized LLMs. For both models, the embeddings of the prompt tokens were trained using the distillation logits from the ShareGPT dataset [82] over two epochs. We used a batch size of 4, a maximum context window length of 1024, a cosine learning rate scheduler, and an initial learning rate of 0.01, without any warmup steps. The batch size, initial learning rate, and warmup steps are determined through a preliminary, albeit not exhaustive, hyperparameter search. The maximum context window length is chosen based on GPU memory constraints and training time considerations. We used the Accelerate framework for distributed training [83]. We trained 3 special tokens and set the decay ratio to 0.8. We found that, given the limited tree size and the tree construction algorithm used, utilizing more than 3 special tokens does not further contribute to the expected acceptance length. The decay ratio follows the setting of Medusa [11]. We initialize the embeddings of the prompt tokens with the normal text token embeddings.

6.1.3 Datasets

We assess the throughput performance of *PPD* across various tasks and datasets. Specifically, we evaluated *PPD* using the MT-bench dataset [84] in both non-greedy and greedy settings (temperature=0), applying the same temperature values as those found in the default MT-bench configuration. MT-bench features a diverse collection of 80 multi-turn questions designed to evaluate various aspects of language model performance. MT-bench contains manually crafted challenging questions to differentiate model performance across common use cases. The questions are categorized into 8 topics, including writing, roleplay, extraction, reasoning, math, coding, stem, and

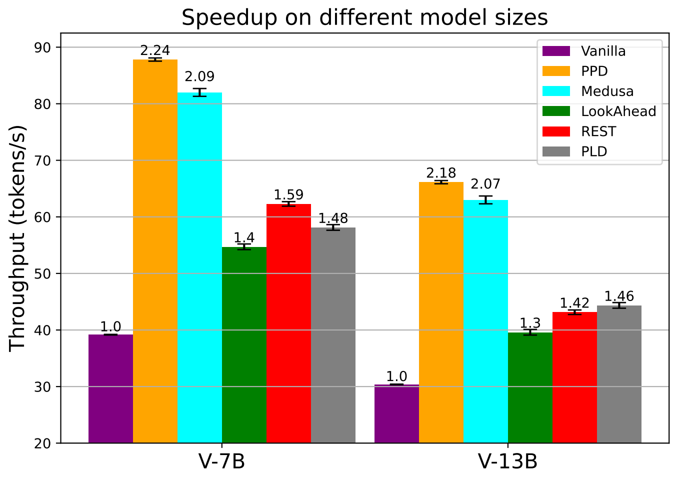


Figure 6.1: Comparative evaluation of throughput speedup between *PPD* and other parallel decoding methods. The experiments were conducted using the MT-Bench dataset, with the temperature set to MT-Bench's default configuration for Medusa and *PPD*. We found that *PPD* is the best-performing parallel decoding methods in terms of wall-time speedup ratios, which is one of the most **exciting** results in this report.

humanities. This comprehensive set of categories ensures a thorough assessment of the model's capabilities across different types of interactions and tasks. We used the GSM8K [85] and HumanEval [4] datasets only in the greedy setting. The GSM8K dataset consists of grade school math problems, and we used the first 500 questions of the test split for our evaluations. HumanEval includes coding tasks, for which we set a maximum new token limit of 512 to control the length of the generated sequences. We used the Alpaca [86] dataset as the validation dataset to produce the latency and accept lengths used for dynamic sparse tree construction.

6.1.4 Additional Experimental Details

For the throughput experiments, each result is obtained by averaging three separate runs. The standard deviations of these runs are reported as error bars in the bar charts. To ensure a fair comparison in our comparative experiments, we maintained consistent hardware settings and software versions.

We selected 3 prompt tokens because adding more would not further increase the expected acceptance length due to the tree size limit. The number of EPTs per prompt token was optimized to maximize throughput.

In Fig. 1.2, the temperature settings for *PPD*, Eagle [16], and Medusa [11] follow the default configuration, while the other models use a greedy setting (temperature=0). This choice is based on findings that retrieval-based methods perform significantly worse in non-greedy settings. Similarly, `LOOKAHEAD DECODING` [78], REST [12], and PLD [81] in Fig. 6.1 also use a temperature setting of 0 for the same reasons.

6.2 Speedup Ratios Compared to Other Parallel Decoding Methods

We compare the speedup ratios and other performance metrics of *PPD* with leading parallel decoding methods on MT-Bench in non-greedy settings in Figure 6.1 and Table 6.1. *PPD* achieves throughput up to 7.3% higher than Medusa while maintaining the same output quality, achieving about the same score on MT-Bench. The memory overhead of *PPD* is negligible, primarily because it arises solely from additional embeddings, which are insignificant compared to the original model size. The reasons for the increase in speedup ratios are two-fold. Firstly, *PPD* produces candidate tokens with a higher acceptance rate than Medusa when utilizing a sparse tree of the same size. Notably, *PPD* continues to achieve a comparable or slightly better acceptance rate even when employing a much smaller sparse tree – ranging from one-third to half the size. Secondly, *PPD* benefits from lower forward pass latency due to its ability to use smaller sparse tree sizes and hence shorter input lengths. *PPD* also eliminates the computational overhead associated with separate decoding heads. The speedup improvements are particularly pronounced in the Vicuna-7B models, likely because the larger forward pass latency in the Vicuna-13B models hides the computational overhead introduced by Medusa heads.

Model	Method	T	τ	L_{fp} (s)	Quality	P_{tr} (%)	S_{tr}	S_{input}
M	Vanilla	50.2	1.00	**0.020**	-	NA	NA	1
	PPD	**108.7**	**2.43**	0.022	*Same*	**4.50**e^{-4}	(10,84,89)	(40,285,285)
V-7B	Vanilla	39.2	1.00	**0.026**	**5.99**	NA	NA	1
	Medusa	82.0	2.51	0.0307	5.98	8.07	63	63
	PPD	**88.0**	**2.54**	0.029	5.93	**1.82**e^{-4}	(10,33,34)	(40,105,105)
V-13B	Vanilla	30.4	1.00	**0.0330**	**6.38**	NA	NA	1
	Medusa	63.4	**2.59**	0.0408	-	5.52	63	63
	PPD	**66.1**	2.44	0.0379	6.32	**7.87**e^{-5}	(10,20,20)	(40,60,60)

Table 6.1: Comparative performance metrics of MobileLLaMA (M) for greedy setting, Vicuna-7B (V-7B) and Vicuna-13B (V-13B) for non-greedy setting using different decoding methods. The table details throughput (T in tokens/s), average accept lengths (τ in tokens), forward pass latency (L_{fp} in seconds), quality scores on MT-benchmark, percentages of additional trainable parameters (P_{tr}) and input lengths (S_{input}) after the prefilling phase. *PPD* employs a dynamic sparse tree with variable tree sizes (S_{tr}), represented as tuples. *Same* means the output matches with that of the original LLM.

Figure 6.2 displays the throughput of *PPD* on MT-Bench, HumanEval, and GSM8K with temperature equal to 0. *PPD* achieves consistent wall-time speedup ratios from 2.12× to 2.49× on different GPUs. In general, *PPD* performs better in coding and math reasoning tasks, achieving speedups between 2.21× and 2.49×. This can be attributed to the fact that both code and math equations often contain fixed patterns and repetitive symbols, which narrows the range of plausible candidates and simplifies the prediction. We also found that with typical acceptance, the speedup increases with temperature. Another notable trend is that the speedup ratios for larger models, like Vicuna-13B, are limited as compared smaller models. This observation echoes the results presented in *LOOKAHEAD Decoding* [78]. Both *PPD* and *LOOKAHEAD Decoding* aim to generate more tokens per step at the expense of increased computation. For larger models, which require more computational resources, the size of the sparse tree must be limited to avoid the increased latency that results from reaching the GPU's utilization cap. Consequently, the acceptance lengths per step are reduced, resulting in lower speedups. One potential solution is to insert the prompt tokens only at the final few decoder layers, a strategy we plan to explore in future work.

6.3 Long-range Token Prediction

For a specific sparse tree, the accumulative accuracy provides a theoretical upper bound for the number of generated tokens per step and the maximum possible speedup ratio. Hence, maximizing accumulative accuracy is crucial for the effectiveness of *PPD*. Figure 6.3 demonstrates the accumulative accuracy of the tokens predicted at various positions. We summarize the following three key insights from the results.

***PPD* excels at predicting more distant tokens.** As depicted in Figure 6.3a, *PPD* consistently outperforms Medusa in accuracy across all token positions. The accuracy gap between *PPD* and Medusa widens with the increased distance from the last prompt token (*e.g.*, the top-10 accuracy difference is 0.01 for the 'next next' word versus 0.1 for the 'next next next next' word). This improvement can be attributed to *PPD*'s ability to partially recover conditional dependency information through causally connected prompt tokens.

***PPD* performs well at generating a broader array of plausible token candidates.** For example, in predicting the token at position 3, the top-10 candidates exhibit an accuracy improvement of 0.1 over Medusa, compared to only 0.02 for the top-1 candidate. This demonstrates

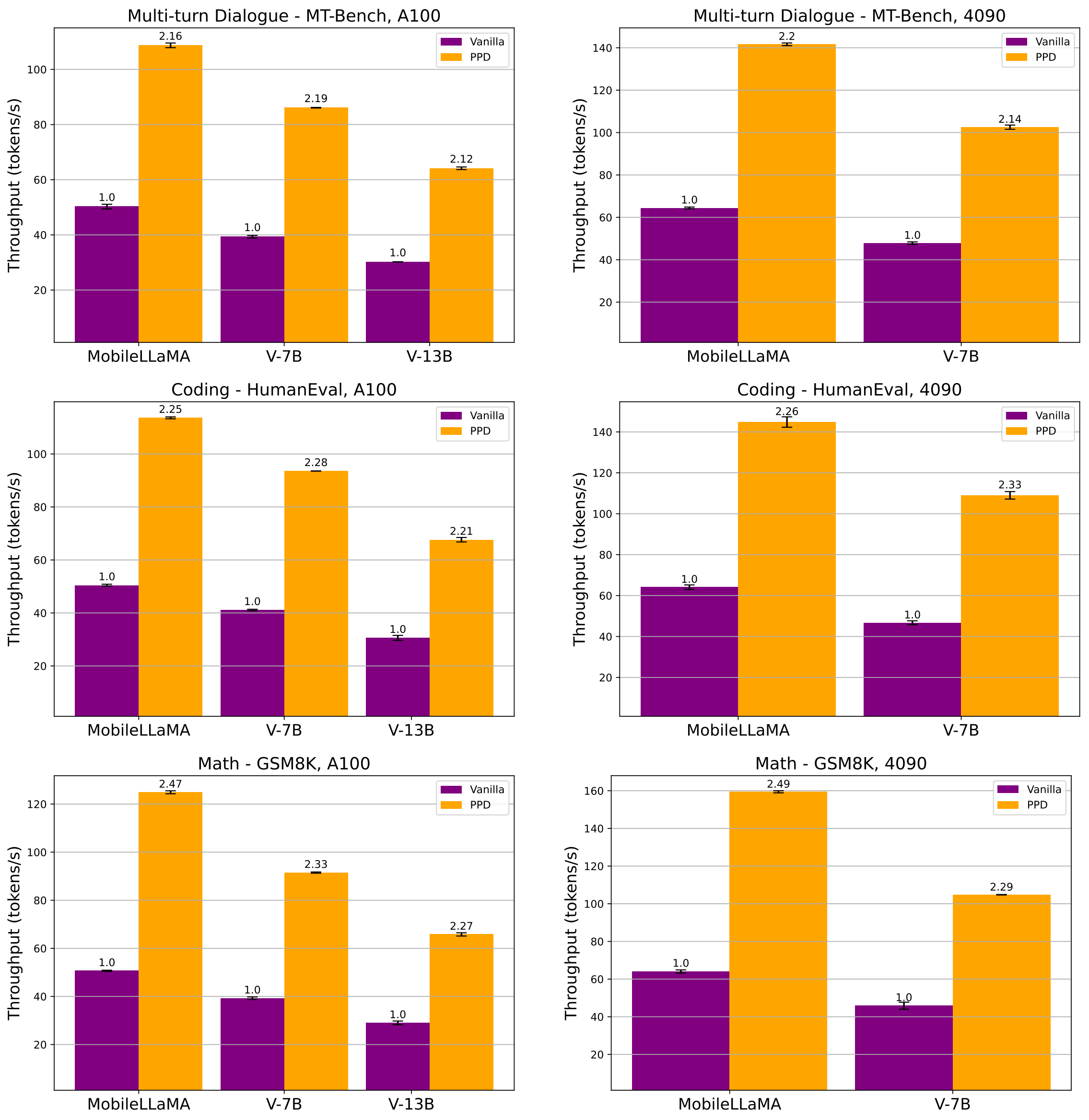
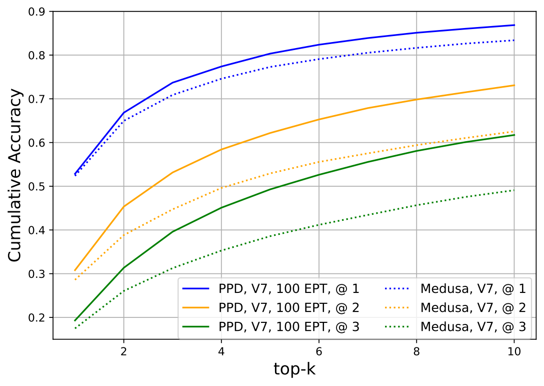


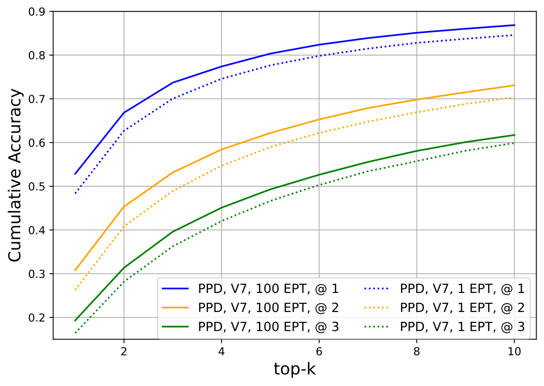
Figure 6.2: Throughput of *PPD* and vanilla models across different tasks. The temperature for experiments are set to 0 and the generated output of *PPD* exactly matches that of the original LLM. We do not show results of Vicuna-13B on RTX 4090 as it does not fit into the GPU memory.

the value of using tree attention and the largest viable tree size during inference, as multiple candidate continuations further boost accuracy improvement.

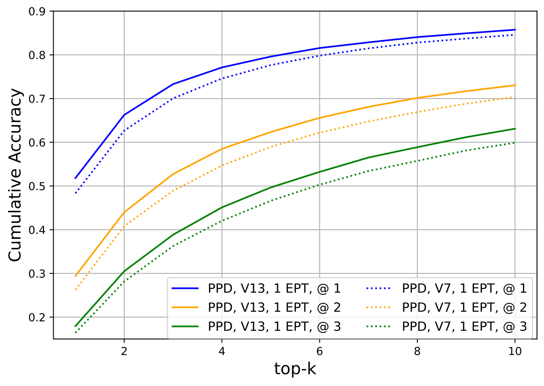
Multiple EPTs per prompt token and larger model sizes yield modest improvements in prediction accuracy. Figure 6.3b shows that using 100 EPTs per prompt token leads to accuracy improvement, ranging from 0.018 to 0.044. This highlights the need to optimize the number of EPTs to balance accuracy gains with computational costs. Figure 6.3c displays that *PPD* with Vicuna-13B outperforms Vicuna-7B with an accuracy gain of 0.011-0.034. This increase is due to Vicuna-13B's greater embedding dimensions and deeper layers, which enhance the expressive power of prompt tokens. However, these gains are modest and can be offset by the increased computational burden of larger model.



(a) PD vs. Medusa



(b) 100 EPT vs. 1 EPT



(c) 13b vs. 7b

Figure 6.3: Accumulative accuracy comparisons across different model configurations and prediction distances. 'V7' for Vicuna-7B, and 'V13' for Vicuna-13B. The notation '@i' refers to a token distance of i. '100 EPT' represents 100 EPTs per prompt token. Accumulative accuracy is defined as top-k accuracy (*e.g.*, a prediction is correct if the top-k candidates contain the ground truth). These measurements were obtained from the Alpaca Eval dataset with a maximum of 20 steps.

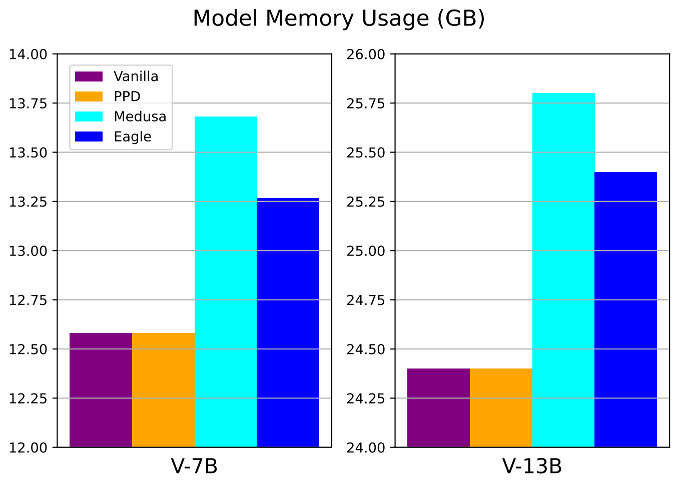


Figure 6.4: Model memory usage.

6.4 Inference Memory Efficiency and Synergistic Integration

6.4.1 Memory efficiency

The memory overhead of *PPD* is just 0.004% of Medusa's and 0.007% of speculative methods like Eagle. This efficiency arises from using embeddings, which are significantly smaller than decoding heads and draft models, both of which scale with vocabulary size. As depicted in Figure 6.4, *PPD* utilizes approximately the same amount of memory as the vanilla model, whereas speculative decoding and parallel decoding methods incur significantly more noticeable memory overhead.

6.4.2 *PPD* + Speculative Decoding

PPD can be easily integrated with speculative decoding [87] to speed up the draft model using prompt tokens. We applied *PPD* to Vicuna-68M [88] and used it as the draft model for Vicuna-7B. This combination resulted in a speedup of up to 1.22× for speculative decoding on Vicuna-7B compared to using speculative decoding alone.

6.5 Comparison of Throughputs on Different GPUs

In this section, we evaluate the performance of *PPD* on 2 different GPUs to gain insights into its scalability. Table 6.2 compares the throughput of *PPD* between the A100 and RTX 4090 GPUs across different tasks. For all tasks where data is available, the RTX 4090 shows higher throughput than the A100. The throughput ratio (4090 over A100) ranges between 1.15 and 1.30, indicating that *PPD* on the RTX 4090 is consistently faster.

Task	Model	A100 Throughput (tokens/s)	4090 Throughput (tokens/s)	4090 Faster By (times)
Multi-turn Dialogue	MobileLLaMA	108.7	141.6	1.30
Multi-turn Dialogue	V-7B	86.1	102.5	1.19
Multi-turn Dialogue	V-13B	64.1	-	N/A
Coding	MobileLLaMA	113.6	144.8	1.28
Coding	V-7B	93.5	108.9	1.16
Coding	V-13B	67.6	-	N/A
Math	MobileLLaMA	124.9	159.4	1.28
Math	V-7B	91.5	104.7	1.15
Math	V-13B	65.8	-	N/A

Table 6.2: Comparison of Throughput of *PPD* between A100 and RTX 4090 across different tasks and models. V-13B model does not fit into the memory of RTX 4090 so its results are not included.

The throughput improvement of the RTX 4090 over the A100 shows low variance across different tasks, indicating a consistent performance advantage. However, the RTX 4090's smaller memory capacity limits its ability to handle larger models like the V-13B, which is a significant consideration for high-end inference tasks. A possible solution is to partition the LLM on multiple GPUs for training [89].

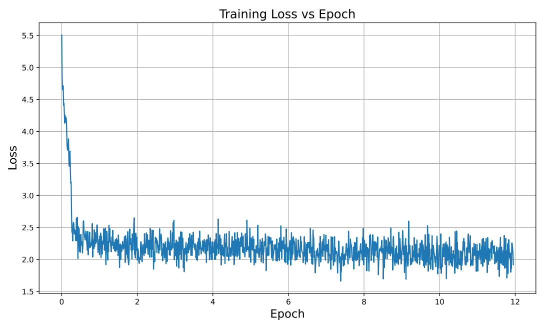
The difference in the performance can be partially accounted by the hardware architecture differences. The RTX 4090 has a significantly higher number of tensor cores (512) and a higher clock speed (2235 MHz) compared to the A100 (432 tensor cores, 1095 MHz clock rate), contributing to the faster computation speed. Given that *PPD* trades computation for step compression, the inference with *PPD* is likely to be compute-bound and have a high arithmetic intensity, which could benefit more from the architectural features of the RTX 4090.

In conclusion, GPUs with more computational resources tend to achieve higher throughputs with *PPD*, whereas GPUs with larger memory capacities offer advantages for inference with large models.

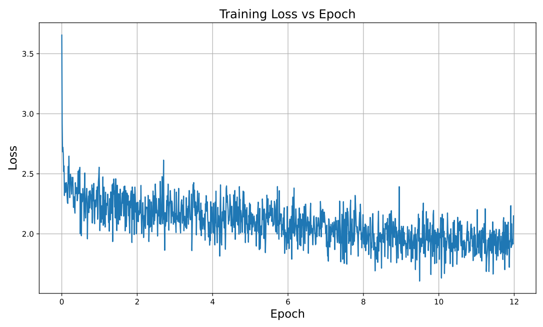
6.6 Training Loss

We study the training loss of *PPD* with different EPTs and loss functions. When knowledge distillation is applied, the loss function used is the KL divergence. Without knowledge distillation, the cross entropy loss function is utilized.

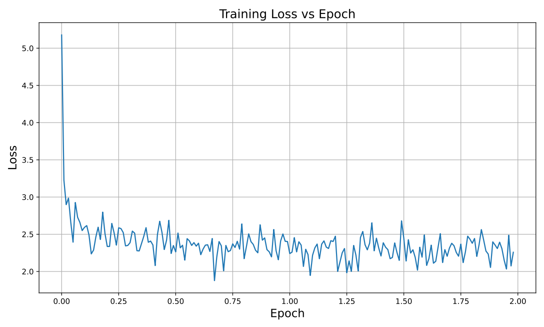
Figure 6.5a and Figure 6.5c shows that, with 1 EPT, the initial loss is quite high, starting above 5. There is a sharp decrease in loss within the first epoch, dropping below 2. After this initial



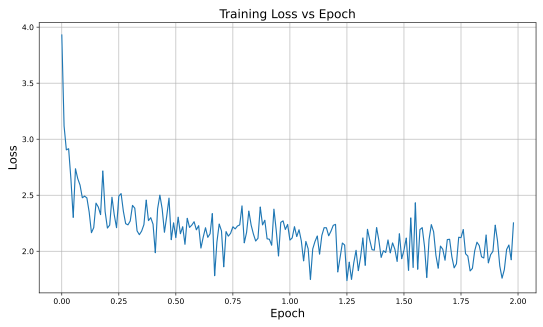
(a) 3 prompt tokens, 1 EPTs, KD



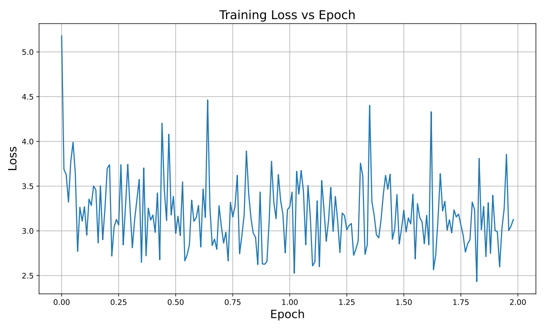
(b) 3 prompt tokens, 100 EPTs, KD



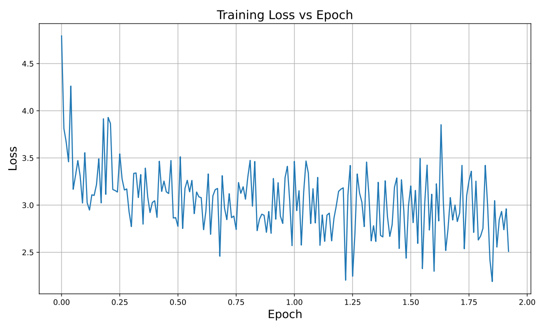
(c) 2 prompt tokens, 1 EPTs, KD



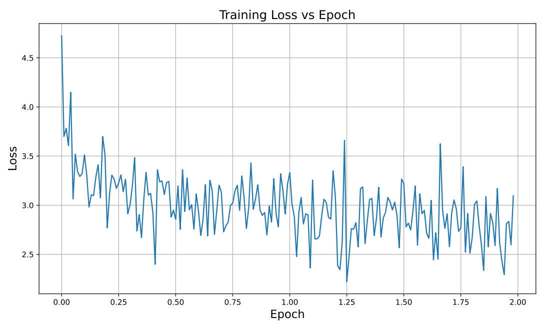
(d) 2 prompt tokens, 100 EPTs, KD



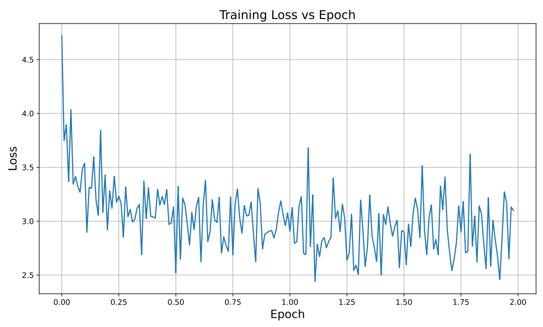
(e) 2 prompt tokens, 1 EPTs, without KD



(f) 2 prompt tokens, 10 EPTs, without KD



(g) 2 prompt tokens, 50 EPTs, without KD



(h) 2 prompt tokens, 100 EPTs, without KD

Figure 6.5: Training Loss. 'KD' means the model is trained with knowledge distillation, while 'without KD' means the model is trained with hard labels. All experiments follow the same training hyperparameters.

drop, the loss stabilizes and oscillates around a value slightly below 2 for the remainder of the training epochs (up to epoch 12). The loss oscillations remain within a narrow range, indicating consistent performance. The fluctuation can be attributed to the insertion of prompt tokens at random positions, while the initial sharp decrease is due to the prompt tokens quickly adapting to reasonable embeddings.

On the other hand, Figure 6.5b and Figure 6.5d, with 100 EPTs, shows the initial loss starting below 3, significantly lower than *PPD* with 1 EPT. Similarly, there is a sharp decrease within the first epoch, with the loss dropping to around 2.5. However, unlike *PPD* with 1 EPT, the loss continues to decrease gradually over the epochs, showing a downward trend. Similarly, Figure 6.5e, Figure 6.5f, Figure 6.5g, and Figure 6.5h show that with an increased number of EPTs, the final training loss decreases and the magnitude of fluctuations diminishes. This suggests that increasing the number of EPTs enhances the model's learning capacity and more effectively reduces training loss over time.

When comparing training with knowledge distillation to training without it, we found that knowledge distillation helps stabilize the training process and also leads to lower training loss. This can be attributed to the fact that knowledge distillation provides a smoother and more informative training signal by leveraging the logits of the original LLM, aligning the prompt tokens more effectively and preventing overfitting.

6.7 Comparison of Training of *PPD* with Medusa

In this section, we compare the training performance of *PPD*, with Medusa. Table 6.3 provides a detailed comparison of the training performance between *PPD* and Medusa. We provide the detailed analysis of each metric in the following discussion.

Method	T_{tr} (minute)	M_{tr} (MiB)	T_{fp} (sec)	T_{bp} (sec)	@1 Top-10	@2 Top-10	@3 Top-10
PPD	15780	18545	0.21	1.65	0.837	0.690	0.583
Medusa	8908	10163	0.18	0.83	0.784	0.567	0.431

Table 6.3: Comparison of Training Performance of *PPD* with Medusa. Both models are trained for 1 epoch using 2 RTX 4090 GPUs with the same training hyperparameters. T_{tr} stands for the total training time, T_{fp} for the time of one forward pass, and T_{bp} for the time of one backward pass. M_tr represents the memory usage during training. @1 Top-10, @2 Top-10, and @3 Top-10 are the top-10 prediction accuracies on the evaluation dataset at the end of 1 training epoch at a token distance of 1, 2, 3 respectively.

Training Time and Memory Usage. The total training time for *PPD* is significantly higher (1.77 $\times$) than that for Medusa, with *PPD* requiring 15780 minutes compared to Medusa's 8908 minutes. Additionally, *PPD* consumes more memory (1.82 $\times$) during training, with a usage of 18545 MiB, whereas Medusa utilizes 10163 MiB. This discrepancy in training time and memory usage can be attributed to the different positions of the trainable components of *PPD* and Medusa in the LLM. Specifically, *PPD* needs to train embeddings, which necessitates the computation and storage of all intermediate gradients from the embedding layer to the last decoder layer. In contrast, Medusa adds decoding heads at the end of the LLM backbone, so it only requires the computation of gradients of the feed-forward layer in the decoding heads. Hence, *PPD* incurs higher computational and storage overhead in training.

Forward and Backward Pass Times. Following the previous discussion, it is expected that *PPD* takes a longer backpropagation time due to the increased gradient computation required. Specifically, *PPD* takes 1.98 $\times$ longer for backpropagation. Additionally, it is worth noting that *PPD* also requires a slightly longer forward pass time (1.16 $\times$), likely due to the extended input length resulting from the insertion of prompt tokens. We discuss possible ways to reduce the computational demand in Section 7.4.

Prediction Accuracy. Despite the higher training costs, *PPD* demonstrates superior performance in terms of prediction accuracy. Across token distances of 1, 2, and 3, *PPD* achieves a performance improvement range of 1.07 to 1.35 times over Medusa. These results indicate that *PPD* demonstrates better prediction accuracy across different token distances, echoing our finding in Section 6.3.

6.8 Ablation Study

6.8.1 Dynamic Sparse Tree

Figure 6.6a shows that dynamic sparse trees consistently achieve longer acceptance lengths compared to static and random ones across varying sizes. The acceptance length for dynamic sparse trees shows a steady increase as the tree size extends, suggesting its good scalability. The convergence of dynamic and static sparse trees at larger sizes suggests a structural similarity emerging from constraints in tree depth and tree node count.

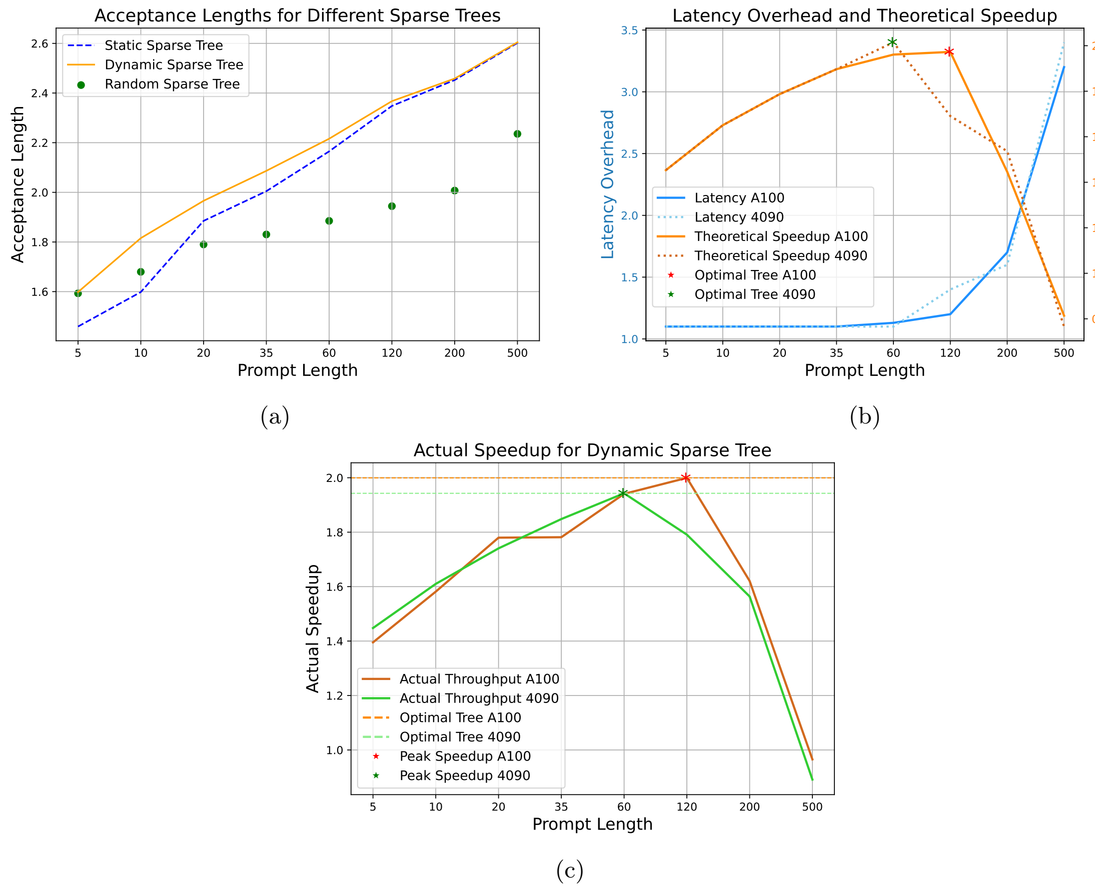


Figure 6.6: Evaluation of Dynamic Sparse Tree Performance. The static sparse trees in (a) always use the largest possible prompt tokens for each candidate. The theoretical speedup in (b) is calculated as the ratio of acceptance lengths (hardware-independent) to latency overhead (hardware-dependent). The optimal tree size is obtained from the peak value of the theoretical speedup. The latency in (b) is obtained from inference on the same prompt for 512 forward passes. (c) shows the actual speedup obtained by running inference on different GPUs with different tree lengths on Alpaca Eval dataset.

6.8.2 Hardware-aware Tree Size

Figure 6.6b presents the theoretical speedup across different GPUs. Figure 6.6c validates that the optimal sparse tree size, derived from theoretical speedup models, indeed results in the greatest actual speedup observed.

6.8.3 Effect of EPTs on Prediction Accuracy

Table 6.4 presents the prediction accuracy of *PPD* using different EPTs. The results indicate that increasing the number of EPTs generally enhances the prediction accuracy of PPD, particularly for long-range token predictions. Higher EPT numbers (e.g., 100 and 50) consistently produce better prediction accuracy compared to lower EPT numbers.

EPT	@1 Top-1	@1 Top-5	@2 Top-1	@2 Top-5
100	0.506	0.794	0.276	0.602
50	0.502	0.791	0.281	0.604
20	0.501	0.791	0.276	0.607
10	0.494	0.786	0.273	0.600
5	0.499	0.787	0.265	0.596
2	0.486	0.777	0.259	0.583
1	0.472	0.771	0.248	0.576

Table 6.4: Prediction Accuracy of *PPD* with different EPTs. '@i' denotes a token distance of i. 'Top-k' denotes the top-k prediction accuracy. The results are obtained on Alpaca dataset with 20 steps.

6.8.4 Impact of Various Hyperparameters on Prediction Accuracy

Table 6.5 summarizes our results with different settings. We analyze the effect of each factor on the prediction accuracy in the following discussion.

EPT	KD	Epoch	Batch	@1 Top-1	@1 Top-5	@2 Top-1	@2 Top-5
100	True	1	4	0.504	0.793	0.273	0.598
100	True	2	4	0.512	0.797	0.288	0.611
100	True	6	4	0.520	0.802	0.302	0.620
100	True	8	4	0.524	0.804	0.307	0.619
100	True	10	4	0.523	0.804	0.305	0.623
100	True	12	4	0.525	0.805	0.308	0.625
100	False	12	4	0.506	0.794	0.276	0.602
100	True	12	1	0.530	0.809	0.309	0.626
1	True	12	1	0.484	0.775	0.259	0.581
1	True	2	4	0.474	0.773	0.247	0.574
1	True	6	4	0.480	0.773	0.250	0.580
1	True	8	4	0.484	0.778	0.257	0.583
1	True	10	4	0.482	0.777	0.257	0.584
1	True	12	4	0.485	0.779	0.261	0.586
1	False	12	4	0.472	0.771	0.248	0.576

Table 6.5: Prediction Accuracy for *PPD* with and without knowledge distillation (KD) for different EPTs, epochs and batch size.

Training Epochs

We first investigates the effect of the number of training epochs on prediction accuracy. For models using 100 EPTs with KD enabled and a batch size of 4, we observe a steady improvement in prediction accuracy as the number of epochs increases. Specifically, the Top-1 accuracy at 1 token distance increases from 0.504 at 1 epoch to 0.525 at 12 epochs, while the Top-5 accuracy at 1 token distance improves from 0.793 to 0.805. Similarly, Top-1 accuracy at 2 token distance increases from 0.273 to 0.308, and Top-5 accuracy at 2 token distance improves from 0.598 to 0.625 over the same range of epochs. This trend demonstrates the positive impact of prolonged training on the performance of *PPD* when KD is applied.

Knowledge Distillation

When KD is not applied, as shown for 100 EPTs at 12 epochs with a batch size of 4, the performance metrics are generally lower. The improvement on prediction accuracy with KD is up to 12%. This suggests that KD contributes significantly to prediction accuracy for *PPD*.

Effect of Batch Size

We also examine the impact of batch size on the prediction accuracy. For the model trained with 100 EPTs, KD enabled, and 12 epochs, reducing the batch size from 4 to 1 results in a slight improvement in prediction accuracy up to 1%.

6.8.5 Prefix Tuning + Prompt Token

Table 6.6 compares the prediction accuracy of *PPD* with and without the application of prefix tuning. The results show that the models without prefix tuning outperform those with prefix tuning up to 28%, which suggests that, in this setup, prefix tuning does not enhance the prediction accuracy of *PPD*. Instead, it appears to degrade performance, potentially due to the complexity introduced by modifying the KV cache of attention layers with the prefix token. Unlike prompt tokens, prefix tokens do not interact with input tokens, meaning they do not change dynamically through the transformer layers based on the input context. This lack of interaction and dynamic adjustment could be a factor contributing to the decreased prediction accuracy observed with prefix tuning.

Prefix Tuning	@1 Top-1	@1 Top-5	@2 Top-1	@2 Top-5
False	0.485	0.779	0.261	0.586
True	0.412	0.738	0.204	0.541

Table 6.6: Prediction Accuracy of *PPD* with and without prefix tuning. 1 EPT is used for all models and 1 prefix token is used for prefix tuning.

6.8.6 Custom Decoding Heads + Prompt Token

Table 6.7 presents the prediction accuracy of *PPD* with and without a custom decoding head. When trained using the single-stage method, *PPD* with the custom decoding head shows a 12%-21% decrease in prediction accuracy compared to the vanilla *PPD* without the custom decoding head. This suggests that the single-stage approach does not result in stable or effective training.

Method Name	@1 Top-1	@1 Top-5	@2 Top-1	@2 Top-5
PPD without custom decoding head	0.485	0.779	0.261	0.586
PPD with custom decoding head (1-stage)	0.385	0.614	0.229	0.482
PPD with custom decoding head (2-stage)	0.506	0.795	0.276	0.602

Table 6.7: Prediction Accuracy of *PPD* with and without custom decoding head. 1 EPT is used for all models. 1-stage and 2-stage refer to the training strategies of custom decoding head.

In contrast, the two-stage training method results in a limited improvement of 2.1%-4.3% in prediction accuracy compared to the vanilla *PPD*. This suggests that adding a custom decoding head may not be necessary, given the additional trainable parameters and the limited improvement in prediction accuracy.

6.8.7 Attention Masking for EPTs

The results in Table 6.8 indicate that the ensemble attention mask outperforms the other masking strategies. In comparison, the *PPD* with decoder attention mask shows 4.9%-8.0% lower prediction accuracy. The *PPD* with encoder attention mask also underperforms in prediciton accuracy relative to the ensemble attention mask by 3.7%-7.2%.

Method Name	@1 Top-1	@1 Top-5	@2 Top-1	@2 Top-5
PPD with ensemble attention mask	0.506	0.794	0.276	0.602
PPD with decoder attention mask	0.465	0.755	0.262	0.572
PPD with encoder attention mask	0.473	0.765	0.256	0.573

Table 6.8: Prediction Accuracy of *PPD* with different attention masking strategies for EPTs. 100 EPT is used for all models.

These results suggest that the ensemble attention mask is the most effective strategy among the three, likely due to its ability to effectively average the votes of disjoint groups of EPTs, thereby improving prediction accuracy. The decoder-like and encoder-like attention masks, while simpler, do not provide the same level of performance, indicating that the structure and specificity of the ensemble attention mask better facilitate accurate long-range token prediction. Additionally, ensemble attention masking is more sparse, which offers greater potential for optimization.

6.8.8 Aggregation Method for EPTs

The results in Table 6.9 show the prediction accuracy of *PPD* with two different aggregation methods for EPTs: simple averaging and learned weights. When using learned weights to aggregate logits, the model shows a slight decrease of 0.6%-9.4% in prediction accuracy.

Aggregation Method	@1 Top-1	@1 Top-5	@2 Top-1	@2 Top-5
Average	0.506	0.794	0.276	0.602
Learned Weight	0.503	0.779	0.250	0.576

Table 6.9: Prediction Accuracy of *PPD* with different aggregation methods for EPTs. 100 EPT is used for all models.

These results suggest that while learned weights provide a more flexible aggregation method, they do not necessarily lead to improved prediction accuracy in this context. The simplicity and stability of the averaging method appear to offer better performance, possibly due to the additional complexity and potential overfitting introduced by learning the weights.

6.8.9 Multi-exit Ensemble

Table 6.10 shows the comparison of prediction accuracy of *PPD* with and without multi-exit ensemble. The results indicate that the introduction of multi-exit ensemble with both 2 and 3 exits results in a 7%-18% decrease in prediction accuracy compared to the vanilla *PPD* model without multi-exit.

Method Name	@1 Top-1	@1 Top-5	@2 Top-1	@2 Top-5
PPD without multi-exit	0.485	0.779	0.261	0.586
PPD with 3 exits	0.422	0.723	0.214	0.517
PPD with 2 exits	0.420	0.723	0.213	0.518

Table 6.10: Prediction Accuracy of *PPD* with and without multi-exit ensemble. 1 EPT is used for all models. k exits refer to the number of exits used.

These findings suggest that the multi-exit ensemble approach, as implemented, does not enhance prediction accuracy and instead leads to a notable decrease in performance. This may be due to the averaging of hidden states from multiple layers introducing noise or reducing the specificity of the representations needed for accurate prediction. Further refinement of the multi-exit ensemble may be necessary to achieve the desired improvements in accuracy.

6.9 Summary

In this chapter, we demonstrate the effectiveness of *PPD* in terms of speedup, inference memory, and synergistic power. We also identify its limitation in slower training time compared to adapter-based methods. The extensions of *PPD* are examined in detail in the ablation study.

Chapter 7

Conclusion

7.1 Summary

We introduced *PPD*, a memory-efficient, cost-effective, and powerful parallel decoding method that incorporates a hardware-aware dynamic sparse tree. Utilizing specially trained prompt tokens to predict long-range tokens accurately, *PPD* achieves a speedup of up to 2.49× in inference while employing only 0.0002% trainable parameters compared to the total number of parameters and without incorporating new models or architectural components. Our technique stands out for three key features:

- **Orthogonal Optimization**: Orthogonal to speculative decoding, *PPD* provides the potential for synergistic integration.
- **Memory Efficiency**: With a minimal runtime memory overhead of just 0.0004%, *PPD* is highly suitable for edge and mobile settings.
- **Training Efficiency**: The training process is efficient, requiring only 16 hours on a single A100-40GB GPU.

We believe that *PPD* offers a novel perspective on the capabilities of parallel decoding. In future work, it could be synergized with other speculative or parallel decoding techniques to expedite inference even further.

7.2 Limitations

Despite its efficiency, we have identified the following limitations of *PPD*:

1. **Low prediction accuracy for very small models.** We found that for very small models like Vicuna-68M [88], which only has 2 decoder layers and an embedding dimension of less than 1000, *PPD* suffers from low prediction accuracy. This is because the embedding dimension determines the expressive power of a prompt token, and the transformer architecture's depth is crucial for efficient information flow to the prompt tokens.
2. **GPU compute resource constraint.** Since *PPD* trades additional compute resources for increased throughput, its effectiveness depends on the availability of idle GPU compute resources. On a GPU with limited compute resources, the speedup ratios achieved by *PPD* are expected to decrease.
3. **Extended input length.** The improvement in acceptance length with *PPD* is not as significant as the gain in prediction accuracy compared to Medusa. This is because *PPD* must reserve a substantial portion of the input for prompt tokens, which limits the size of the sparse tree that can be used.

7.3 Ethical Issues

In this paper, we proposed *PPD* to accelerate LLMs easily and cheaply. Since *PPD* reduces the time required for handling a single inference request, it could bring down the cost of deploying LLMs for both the companies and the public. This might lead to increased accessibility of LLM services. Moreover, latency-sensitive applications like chatbots will benefit greatly from the usage of *PPD* as it reduces the inference latency greatly, providing better use experience.

While *PPD* aims to make AI more accessible, there may still be a digital divide where certain communities lack the necessary infrastructure, such as stable internet connections or modern hardware, to fully benefit from these advancements. This could further widen the gap between technology-privileged and underserved populations. On the other hand, *PPD* might be misused by malicious parties to manipulate the output of the original LLM, resulting in the generation of unreliable information and fake data.

While the project does not involve human participants, it does involve human-provided data. The dataset used for fine-tuning, shareGPT, comes from users who voluntarily contribute their data. In this training dataset, all conversations are anonymised, conforming to the regulation of GDPR. In the wider context of chatbots, problems like performance disparities between demographic groups and social stereotypes might exist. However, since the project focuses on the acceleration of LLM inference, it is not directly related to these issues.

7.4 Future Work

PPD has shown great promise in accelerating LLM inference in a cost-effective and memory-efficient manner. It also demonstrates strong synergistic potential with fine-tuning methods across various domains. Below, we discuss several possible directions for future work.

7.4.1 Further Reduce the Training Cost

Although *PPD* generally requires much less training time than speculative decoding methods, it does not have a strong advantage over other parallel decoding methods in terms of training cost. Efforts can be made to further reduce the training cost of *PPD*. For instance, Test-Time Prompt Tuning [90] proposes optimizing the prompt at test-time, a methodology that could be applied to *PPD* in a zero-shot manner, eliminating the training phase entirely. Observations indicate that the training loss of *PPD* rapidly decreases to a stable value within the first few training samples, suggesting that it does not require a large amount of data to provide reasonable predictions. Another approach is to train *PPD* for fewer epochs and periodically adapt the prompt tokens at test-time.

7.4.2 Minimize the Computational Overhead

While *PPD* accelerates LLM inference by compressing the steps in autoregressive generation, it introduces computational overhead and additional latency for each decoding step. Given that this overhead increases with model depth, we could strategically select where in the LLM to insert prompt tokens. Speculative Streaming [91] suggests that embeddings need not be inserted at the start of the LLM. Instead, adding embeddings at one of the middle or last layers could reduce computation. The hidden states of input tokens may carry more contextual information and meaningful representation in the last few layers, making the interaction of prompt tokens with input tokens more significant in these layers. This late-entry approach can lead to faster inference and training. Another alternative is to early-exit the hidden states of the prompt tokens if they already incorporate meaningful information.

Although multi-exit ensemble methods did not yield positive results in our preliminary experiments, they could potentially be enhanced with learned weightings and other advanced techniques to improve performance.

In our proposed construction algorithm for dynamic sparse trees, we currently prioritize empirical hardware latency as the primary constraint. In future work, we aim to incorporate more exact measures like the Floating-point operations per second (FLOPS) into our optimization framework, thereby solving the optimization problem under a limited FLOPS budget.

7.4.3 Marry Acceleration with Downstream Task Customization

This report primarily examines the application of *PPD* for accelerating LLM inference while maintaining the original model distribution. In different contexts, however, preserving the original LLM's distribution may not be necessary or even desirable. For instance, there might be scenarios where customizing the LLM for a specific downstream task is more beneficial. By employing alternative acceptance schemes or verification strategies, *PPD* can be effectively utilized to modify the model distribution while achieving accelerated inference.

7.4.4 Combination of *PPD* with Other Acceleration Methods

Our preliminary attempts have demonstrated that *PPD* can synergize effectively with speculative decoding methods to achieve enhanced acceleration. To further increase throughput, we propose developing a unified framework that integrates *PPD* with other speculative methods. A critical challenge will be designing a specialized attention mechanism to efficiently integrate the draft trees of *PPD* with that of the speculative decoding methods.

In addition, *PPD* can benefit from enhancements such as KV compression, low-precision quantization, cascade inference, or batch processing. Exploring the intersection of these acceleration techniques presents a rich area for future research. Particularly promising is the combination of *PPD* with KV compression, given its potential to optimize the size of the KV cache and increase idle compute resources.

Bibliography

[1] OpenAI. GPT-4 Technical Report-annotated. 2023 -12-19.

[2] Touvron H, Lavril T, Izacard G, Martinet X, Lachaux MA, Lacroix T, et al. LLaMA: Open and Efficient Foundation Language Models. 2023.

[3] Brown TB, Mann B, Ryder N, Subbiah M, Kaplan J, Dhariwal P, et al. Language Models are Few-Shot Learners. 2020 -07-22.

[4] Chen M, Tworek J, Jun H, Yuan Q, de Oliveira Pinto HP, Kaplan J, et al. Evaluating Large Language Models Trained on Code. 2021.

[5] Chang K, Wang Y, Ren H, Wang M, Liang S, Han Y, et al. ChipGPT: How far are we from natural language hardware design. 2023.

[6] Hussain AS, Liu S, Sun C, Shan Y. M^{2}UGen: Multi-modal Music Understanding and Generation with the Power of Large Language Models; 2024.

[7] Kaddour J, Harris J, Mozes M, Bradley H, Raileanu R, McHardy R. Challenges and Applications of Large Language Models; 2023.

[8] Stern M, Shazeer N, Uszkoreit J. Blockwise Parallel Decoding for Deep Autoregressive Models. 2018 -11-07.

[9] Chen C, Borgeaud S, Irving G, Lespiau JB, Sifre L, Jumper J. Accelerating Large Language Model Decoding with Speculative Sampling. arXiv preprint arXiv:230201318. 2023.

[10] Santilli A, Severino S, Postolache E, Maiorca V, Mancusi M, Marin R, et al. Accelerating Transformer Inference for Translation via Parallel Decoding.

[11] Cai T, Li Y, Geng Z, Peng H, Lee JD, Chen D, et al. Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads. 2024.

[12] He Z, Zhong Z, Cai T, Lee JD, He D. REST: Retrieval-Based Speculative Decoding; 2023.

[13] Miao X, et al. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In: ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS); 2024. .

[14] Chu X, Qiao L, Lin X, Xu S, Yang Y, Hu Y, et al. MobileVLM : A Fast, Strong and Open Vision Language Assistant for Mobile Devices. arXiv preprint arXiv:231216886. 2023.

[15] Chiang WL, Li Z, Lin Z, Sheng Y, Wu Z, Zhang H, et al. Vicuna: An Open-Source Chatbot Impressing GPT-4 with 90%* ChatGPT Quality. 2023 March. Available from: https://lmsys.org/blog/2023-03-30-vicuna/.

[16] Li Y, Wei F, Zhang C, Zhang H. EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty. In: International Conference on Machine Learning; 2024. .

[17] Vaswani A, Shazeer N, Parmar N, Uszkoreit J, Jones L, Gomez AN, et al. Attention Is All You Need. 2023.

[18] Devlin J, Chang MW, Lee K, Toutanova K. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding; 2019.

[19] Raffel C, Shazeer N, Roberts A, Lee K, Narang S, Matena M, et al. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. 2023.

[20] Ramesh A, Pavlov M, Goh G, Gray S, Voss C, Radford A, et al. Zero-Shot Text-to-Image Generation. 2021.

[21] Bommasani R, Hudson DA, Adeli E, Altman R, Arora S, von Arx S, et al. On the Opportunities and Risks of Foundation Models. 2022.

[22] Nvidia. NVIDIA A100 Tensor Core GPU Architecture UNPRECEDENTED ACCELERATION AT EVERY SCALE ii NVIDIA A100 Tensor Core GPU Architecture.

[23] Peng H, Davidson S, Shi R, Song SL, Taylor M. Chiplet Cloud: Building AI Supercomputers for Serving Large Generative Language Models; 2024.

[24] Jouppi NP, Kurian G, Li S, Ma P, Nagarajan R, Nai L, et al. TPU v4: An Optically Reconfigurable Supercomputer for Machine Learning with Hardware Support for Embeddings. 2023.

[25] Yemme A, Garani SS. A Scalable GPT-2 Inference Hardware Architecture on FPGA. In: 2023 International Joint Conference on Neural Networks (IJCNN); 2023. p. 1-8.

[26] Yu GI, Jeong JS, Kim GW, Kim S, Chun BG. Orca: A Distributed Serving System for Transformer-Based Generative Models. In: 16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Carlsbad, CA: USENIX Association; 2022. p. 521-38. Available from: https://www.usenix.org/conference/osdi22/presentation/yu.

[27] Weng L. Large Transformer Model Inference Optimization. Lil'Log. 2023 Jan. Available from: https://lilianweng.github.io/posts/2023-01-10-inference-optimization/.

[28] Pope R, Douglas S, Chowdhery A, Devlin J, Bradbury J, Levskaya A, et al. Efficiently Scaling Transformer Inference. 2022.

[29] Ma Y, Gong W, Mao F. Transfer learning used to analyze the dynamic evolution of the dust aerosol. Journal of Quantitative Spectroscopy and Radiative Transfer. 2015;153:119-30. Topical issue on optical particle characterization and remote sensing of the atmosphere: Part II. Available from: https://www.sciencedirect.com/science/article/pii/S0022407314004154.

[30] Ogoe H, Visweswaran S, Lu X, Gopalakrishnan V. Knowledge transfer via classification rules using functional mapping for integrative modeling of gene expression data. BMC bioinformatics. 2015 07;16:226.

[31] Farhadi A, Forsyth D, White R. Transfer Learning in Sign language. In: 2007 IEEE Conference on Computer Vision and Pattern Recognition; 2007. p. 1-8.

[32] Chen C, Zhang P, Zhang H, Dai J, Yi Y, Zhang H, et al. Deep Learning on Computational-Resource-Limited Platforms: A Survey. Mobile Information Systems. 2020 03;2020:1-19.

[33] Simoulin A, Park N, Liu X, Yang G. Memory-Efficient Selective Fine-Tuning. In: Workshop on Efficient Systems for Foundation Models @ ICML2023; 2023. Available from: https://openreview.net/forum?id=zaNbLceVwm.

[34] Lv K, Yang Y, Liu T, Gao Q, Guo Q, Qiu X. Full Parameter Fine-tuning for Large Language Models with Limited Resources; 2023.

[35] Malladi S, Gao T, Nichani E, Damian A, Lee JD, Chen D, et al. Fine-Tuning Language Models with Just Forward Passes. 2024.

[36] Sun X, Ji Y, Ma B, Li X. A Comparative Study between Full-Parameter and LoRA-based Fine-Tuning on Chinese Instruction Data for Instruction Following Large Language Model; 2023.

[37] Razuvayevskaya O, Wu B, Leite JA, Heppell F, Srba I, Scarton C, et al. Comparison between parameter-efficient techniques and full fine-tuning: A case study on multilingual news article classification. 2023.

[38] Pfeiffer J, Kamath A, Rücklé A, Cho K, Gurevych I. AdapterFusion: Non-Destructive Task Composition for Transfer Learning; 2021.

[39] Ding N, Qin Y, Yang G, Wei F, Yang Z, Su Y, et al. Parameter-efficient fine-tuning of large-scale pre-trained language models. Nature Machine Intelligence. 2023;5(3):220-35. Available from: https://doi.org/10.1038/s42256-023-00626-4.

[40] Lester B, Al-Rfou R, Constant N. The Power of Scale for Parameter-Efficient Prompt Tuning. 2021:3045-59. ID: ctx44609084940001591.

[41] Houlsby N, Giurgiu A, Jastrzebski S, Morrone B, de laroussilhe Q, Gesmundo A, et al. Parameter-Efficient Transfer Learning for NLP. International Conference on Machine Learning, Vol 97. 2019;97. PT: C; CT: 36th International Conference on Machine Learning (ICML); CY: JUN 09-15, 2019; CL: Long Beach, CA; UT: WOS:000684034302095.

[42] Hu Z, Wang L, Lan Y, Xu W, Lim EP, Bing L, et al. Llm-adapters: An adapter family for parameter-efficient fine-tuning of large language models. arXiv preprint arXiv:230401933. 2023.

[43] He J, Zhou C, Ma X, Berg-Kirkpatrick T, Neubig G. Towards a Unified View of Parameter-Efficient Transfer Learning; 2022.

[44] Hu EJ, Shen Y, Wallis P, Allen-Zhu Z, Li Y, Wang S, et al. LoRA: Low-Rank Adaptation of Large Language Models. In: International Conference on Learning Representations; 2022. Available from: https://openreview.net/forum?id=nZeVKeeFYf9.

[45] Aghajanyan A, Zettlemoyer L, Gupta S. Intrinsic Dimensionality Explains the Effectiveness of Language Model Fine-Tuning; 2020.

[46] Zhang L, Zhang L, Shi S, Chu X, Li B. LoRA-FA: Memory-efficient Low-rank Adaptation for Large Language Models Fine-tuning; 2023.

[47] Huang C, Liu Q, Lin BY, Pang T, Du C, Lin M. LoraHub: Efficient Cross-Task Generalization via Dynamic LoRA Composition; 2023.

[48] Valipour M, Rezagholizadeh M, Kobyzev I, Ghodsi A. DyLoRA: Parameter Efficient Tuning of Pre-trained Models using Dynamic Search-Free Low-Rank Adaptation; 2023.

[49] Shin T, Razeghi Y, au2 RLLI, Wallace E, Singh S. AutoPrompt: Eliciting Knowledge from Language Models with Automatically Generated Prompts; 2020.

[50] Li XL, Liang P. Prefix-Tuning: Optimizing Continuous Prompts for Generation. 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Acl-Ijcnlp 2021), Vol 1. 2021:4582-97. PT: C; CT: Joint Conference of 59th Annual Meeting of the Association-for-Computational-Linguistics (ACL) / 11th International Joint Conference on Natural Language Processing (IJCNLP) / 6th Workshop on Representation Learning for NLP (RepL4NLP); CY: AUG 01-06, 2021; CL: ELECTR NETWORK; SP: Assoc Computat Linguist, Apple, Bloomberg Engn, Facebook AI, Google Res, Amazon Science, ByteDance, Megagon Labs, Microsoft, Baidu, DeepMind, Tencent, IBM, Alibaba Grp, Duolingo, Naver, Babelscape, Bosch, LegalForce, Adobe ETS; UT: WOS:000698679200153.

[51] Liu Z, Wang J, Dao T, Zhou T, Yuan B, Song Z, et al. Deja Vu: Contextual Sparsity for Efficient LLMs at Inference Time. In: Krause A, Brunskill E, Cho K, Engelhardt B, Sabato S, Scarlett J, editors. Proceedings of the 40th International Conference on Machine Learning. vol. 202 of Proceedings of Machine Learning Research. PMLR; 2023. p. 22137-76. Available from: https://proceedings.mlr.press/v202/liu23am.html.

[52] Sheng Y, Zheng L, Yuan B, Li Z, Ryabinin M, Chen B, et al. FlexGen: high-throughput generative inference of large language models with a single GPU. In: Proceedings of the 40th International Conference on Machine Learning. ICML'23. JMLR.org; 2023. .

[53] Aminabadi RY, Rajbhandari S, Awan AA, Li C, Li D, Zheng E, et al. DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. SC '22. IEEE Press; 2022. .

[54] Tri Dao FMGS Daniel Haziza. Flash-Decoding for long-context inference | PyTorch; 2023. Available from: https://pytorch.org/blog/flash-decoding/.

[55] Teerapittayanon S, McDanel B, Kung HT. BranchyNet: Fast Inference via Early Exiting from Deep Neural Networks; 2017.

[56] Liu W, Zhou P, Zhao Z, Wang Z, Deng H, Ju Q. FastBERT: a Self-distilling BERT with Adaptive Inference Time; 2020.

[57] Liao K, Zhang Y, Ren X, Su Q, Sun X, He B. A Global Past-Future Early Exit Method for Accelerating Inference of Pre-trained Language Models. In: Toutanova K, Rumshisky A, Zettlemoyer L, Hakkani-Tur D, Beltagy I, Bethard S, et al., editors. Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Online: Association for Computational Linguistics; 2021. p. 2013-23. Available from: https://aclanthology.org/2021.naacl-main.162.

[58] He X, Keivanloo I, Xu Y, He X, Zeng B, Rajagopalan S, et al. Magic pyramid: Accelerating inference with early exiting and token pruning. In: NeurIPS 2021 Workshop on Efficient Natural Language and Speech Processing; 2021. Available from: https://www.amazon.science/publications/magic-pyramid-accelerating-inference-with-early-exiting-and-token-pruning.

[59] Schuster T, Fisch A, Jaakkola T, Barzilay R. Consistent Accelerated Inference via Confident Adaptive Transformers; 2021.

[60] Xin J, Tang R, Lee J, Yu Y, Lin J. DeeBERT: Dynamic Early Exiting for Accelerating BERT Inference; 2020.

[61] Schwartz R, Stanovsky G, Swayamdipta S, Dodge J, Smith NA. The Right Tool for the Job: Matching Model and Instance Complexities; 2020.

[62] Zhou W, Xu C, Ge T, McAuley J, Xu K, Wei F. BERT Loses Patience: Fast and Robust Inference with Early Exit; 2020.

[63] Zhang Z, Zhu W, Zhang J, Wang P, Jin R, Chung TS. PCEE-BERT: Accelerating BERT Inference via Patient and Confident Early Exiting. In: Carpuat M, de Marneffe MC, Meza Ruiz IV, editors. Findings of the Association for Computational Linguistics: NAACL 2022. Seattle, United States: Association for Computational Linguistics; 2022. p. 327-38. Available from: https://aclanthology.org/2022.findings-naacl.25.

[64] Miao X, Oliaro G, Jin H, Zhang Z, Cheng X, Chen T, et al. Towards Efficient Generative Large Language Model Serving: A Survey from Algorithms to Systems. Journal of the ACM. 2018;37:32.

[65] Corro LD, Giorno AD, Agarwal S, Yu B, Awadallah A, Mukherjee S. SkipDecode: Autoregressive Skip Decoding with Batching and Caching for Efficient LLM Inference; 2023.

[66] Li L, Lin Y, Chen D, Ren S, Li P, Zhou J, et al. CascadeBERT: Accelerating Inference of Pre-trained Language Models via Calibrated Complete Models Cascade. 2021.

[67] Wang Y, Chen K, Tan H, Guo K. Tabi: An Efficient Multi-Level Inference System for Large Language Models; 2023. p. 233-48.

[68] Chen L, Zaharia M, Zou J. FrugalGPT: How to Use Large Language Models While Reducing Cost and Improving Performance; 2023.

[69] Karpukhin V, Oğuz B, Min S, Lewis P, Wu L, Edunov S, et al. Dense Passage Retrieval for Open-Domain Question Answering. In: Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP); 2020. p. 6769-81.

[70] Chen Z, May A, Svirschevski R, Huang Y, Ryabinin M, Jia Z, et al. Sequoia: Scalable, Robust, and Hardware-aware Speculative Decoding. arXiv preprint arXiv:240212374. 2024.

[71] Gu J, Bradbury J, Xiong C, Li VOK, Socher R. Non-Autoregressive Neural Machine Translation; 2018.

[72] Xiao Y, Wu L, Guo J, Li J, Zhang M, Qin T, et al. A Survey on Non-Autoregressive Generation for Neural Machine Translation and Beyond. IEEE Transactions on Pattern Analysis and Machine Intelligence. 2023;45(10):11407-27.

[73] Wang C, Zhang J, Chen H. Semi-Autoregressive Neural Machine Translation; 2018.

[74] Lee J, Mansimov E, Cho K. Deterministic Non-Autoregressive Neural Sequence Modeling by Iterative Refinement; 2018.

[75] Zhan J, Chen Q, Chen B, Wang W, Bai Y, Gao Y. DePA: Improving Non-autoregressive Machine Translation with Dependency-Aware Decoder; 2023.

[76] Miao X, Oliaro G, Zhang Z, Cheng X, Wang Z, Zhang Z, et al. SpecInfer: Accelerating Large Language Model Serving with Tree-based Speculative Inference and Verification. In: Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3. ASPLOS '24. New York, NY, USA: Association for Computing Machinery; 2024. p. 932–949. Available from: https://doi.org/10.1145/3620666.3651335.

[77] Santilli A, Severino S, Postolache E, Maiorca V, Mancusi M, Marin R, et al. Accelerating Transformer Inference for Translation via Parallel Decoding.

[78] Fu Y, Bailis P, Stoica I, Zhang H. Breaking the Sequential Dependency of LLM Inference Using Lookahead Decoding; 2023. Available from: https://lmsys.org/blog/2023-11-21-lookahead-decoding/.

[79] Monea G, Joulin A, Grave E. PaSS: Parallel Speculative Sampling. arXiv preprint arXiv:231113581. 2023.

[80] Lester B, Al-Rfou R, Constant N. The Power of Scale for Parameter-Efficient Prompt Tuning.

[81] Saxena A. Prompt Lookup Decoding; 2023. Available from: https://github.com/apoorvumang/prompt-lookup-decoding/.

[82] ShareGPT. ShareGPT; 2023. Available from: https://huggingface.co/datasets/Aeala/ShareGPT_Vicuna_unfiltered.

[83] Gugger S, Debut L, Wolf T, Schmid P, Mueller Z, Mangrulkar S, et al. Accelerate: Training and inference at scale made simple, efficient and adaptable. 2022.

[84] Zheng L, Chiang WL, Sheng Y, Zhuang S, Wu Z, Zhuang Y, et al. Judging LLM-as-a-Judge with MT-Bench and Chatbot Arena. In: Advances in Neural Information Processing Systems (NeurIPS); 2023. .

[85] Cobbe K, Kosaraju V, Bavarian M, Chen M, Jun H, Kaiser L, et al. Training Verifiers to Solve Math Word Problems. arXiv preprint arXiv:211014168. 2021.

[86] Dubois Y, Li X, Taori R, Zhang T, Gulrajani I, Ba J, et al. AlpacaFarm: A Simulation Framework for Methods that Learn from Human Feedback. 2023.

[87] Kim S, Mangalam K, Moon S, Malik J, Mahoney MW, Gholami A, et al. Speculative Decoding with Big Little Decoder. Advances in Neural Information Processing Systems (NeurIPS). 2024;36.

[88] Yang S, Huang S, Dai X, Chen J. Multi-Candidate Speculative Decoding. arXiv preprint arXiv:240106706. 2024.

[89] Kuleshov V. MiniLLM: Large Language Models on Consumer GPUs. GitHub; 2023. https://github.com/kuleshov/minillm.

[90] Shu M, Nie W, Huang DA, Yu Z, Goldstein T, Anandkumar A, et al. Test-Time Prompt Tuning for Zero-Shot Generalization in Vision-Language Models. ArXiv. 2022;abs/2209.07511. Available from: https://api.semanticscholar.org/CorpusID:252284022.

[91] Bhendawade N, Belousova I, Fu Q, Mason H, Rastegari M, Najibi M. Speculative Streaming: Fast LLM Inference without Auxiliary Models; 2024.