Lookahead: An Inference Acceleration Framework for Large Language Model with Lossless Generation Accuracy

Yao Zhao*
nanxiao.zy@antgroup.com
Ant Group
Hangzhou, China

Zhitian Xie
xiezhitian.xzt@antgroup.com
Ant Group
Hangzhou, China

Chen Liang
liangchen.liangche@antgroup.com
Ant Group
Hangzhou, China

Chenyi Zhuang
c@antgroup.com
Ant Group
Hangzhou, China

Jinjie Gu
jinjie.gujj@antgroup.com
Ant Group
Hangzhou, China

ABSTRACT

As Large Language Models (LLMs) have made significant advancements across various tasks, such as question answering, translation, text summarization, and dialogue systems, the need for accuracy in information becomes crucial, especially for serious financial products serving billions of users like Alipay. However, for a real-world product serving millions of users, the inference speed of LLMs becomes a critical factor compared to a mere experimental model.

Hence, this paper presents a generic framework for accelerating the inference process, resulting in a substantial increase in speed and cost reduction for our LLM-based scenarios, with lossless generation accuracy. In the traditional inference process, each token is generated sequentially by the LLM, leading to a time consumption proportional to the number of generated tokens. To enhance this process, our framework, named *lookahead*, introduces a *multi-branch* strategy. Instead of generating a single token at a time, we propose a Trie-based retrieval and verification mechanism to be able to accept several tokens at a forward step. Our strategy offers two distinct advantages: (1) it guarantees absolute correctness of the output, avoiding any approximation algorithms, and (2) the worst-case performance of our approach could be comparable with the performance of the conventional process. We conduct extensive experiments to demonstrate the significant improvements achieved by applying our inference acceleration framework. Our framework has been widely deployed in Alipay since April 2023, and obtained remarkable 2.66x to 6.26x speedup. Our code is available at https://github.com/alipay/PainlessInferenceAcceleration.

CCS CONCEPTS

• **Computing methodologies** → **Natural language generation**.

KEYWORDS

Large Language Model, Lossless generation accuracy, Inference Framework, Trie tree, Single-branch draft, Multi-branch draft

ACM Reference Format:
Yao Zhao, Zhitian Xie, Chen Liang, Chenyi Zhuang, and Jinjie Gu. 2024. Lookahead: An Inference Acceleration Framework for Large Language Model with Lossless Generation Accuracy. In *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '24), August 25–29, 2024, Barcelona, Spain.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3637528.3671614

1 INTRODUCTION

Large language models (LLMs) based on transformer architecture have exhibited remarkable performance across various benchmarks, rendering them highly valuable in numerous industries. However, despite their significant achievements in language-based tasks, LLMs still face challenges in terms of inference latency when employed in generative tasks. This drawback becomes particularly apparent in scenarios where step-wise decoding is implemented.

To further gain deeper insights into the factors affecting the LLMs' inference latency, we conduct a comprehensive theoretical analysis focused on a specific instance. Generating a single token with a 10B LLM (without sparse activation) necessitates reading approximately 20GB of memory and performing approximately 20G FLOPs. Although the computational load and memory access associated with the attention mechanism scale quadratically and linearly with sequence length, respectively, their impact can be considered negligible when the sequence length is less than half of the hidden dimension. For instance, in a GLM-10B model with a prompt length of 512, the memory required to read weights is 20.27GB, with additional memory for inputs and outputs totaling about 1.63GB. The computational FLOPs for matrix multiplication involving these weights are around 20.27T, whereas the combined FLOPs for attention and activation functions are approximately 0.405T. Given that an Nvidia A100 GPU provides a bandwidth of 2039 GB/s and a computational capacity of 312T FLOPs, the input/output (IO) time is estimated at 10ms (i.e., 20/2039), while the computation time is a marginal 0.06ms (i.e., 20/312,000). These calculations underscore that the IO time, rather than the computation time which is contingent on the hardware's FLOPs capacity, is the predominant factor influencing the overall inference latency of

*Corresponding author

KDD '24, August 25–29, 2024, Barcelona, Spain

ACM ISBN 979-8-4007-0490-1/24/08.
https://doi.org/10.1145/3637528.3671614

LLMs, due to its strong correlation with the model size and the available memory bandwidth.

Various techniques, such as quantization [1, 2], sparsity [3, 4], pruning [5, 6], distilling [7, 8], and tensor decomposition [9, 10], have been proposed to reduce the LLMs' size and the IO consumption time for predicting each token in LLMs. However, these techniques have been found to result in a degradation of accuracy. To address the challenge of predicting more tokens within a fixed IO consumption time, non-autoregressive neural machine translation (NAT) [11] and iterative parallel decoding [12] have been introduced and successfully applied in translation tasks. Unfortunately, this approach has shown limited effectiveness in question-answering scenarios.

Recently, speculative decoding with a draft model has become a popular strategy. However, this strategy necessitates extra training efforts [13–15] or a smaller auxiliary model capable of producing drafts [16–18], and they may worse the memory burden with additional parameters or models. In light of this, training and assist-model free strategies are proposed, such as LLMA [19] and LookaheadDecoding [20]. The LLMA algorithm relies on a text-matching technique, which is effective within document retrieval domains, tends to underperform in other applications. LookaheadDecoding, on the other hand, incorporates the Jacobi iteration for draft generation, which, despite its innovative design, may face computational bottlenecks that impair its overall effectiveness.

Table 1 summarizes the acceleration techniques discussed previously. To address their limitations, we introduce *Lookahead*, a groundbreaking framework that incorporates a trie-tree-based retrieving strategy and a multi-branch-based parallel Verification and Accept (VA) strategy. *Lookahead* leverages a trie tree to record the n-gram tokens of input prompts and generated responses. The draft is retrieved based on the provided context tokens, allowing for extremely fast draft generation. The draft tokens arranged in a logical tree structure are efficiently processed in the Verification and Accept (VA) process. Furthermore, we have implemented an adaptive strategy to optimize the retrieval process, effectively striking a balance between memory and computation requirements. It shows that *Lookahead* proves its superior performance in accelerating LLMs' inference, compared with the existing state-of-the-art (SOTA) acceleration method.

Being benefited from the superior performance and accessibility, our *Lookahead* framework has been widely employed in dozens of the real world scenarios of Alipay, including financial RAG, health suggestion, medical report summary, etc.

Moreover, to gain a wide range of applications, we have implemented our framework based on the transformers library of Hugging face[1], by extending a generation mode named *lookahead generation*, which supports the *greedy search* and *sample* generation strategy. We have also currently applied *Lookahead* to the most recent LLMs, such as GLM[23], Llama[24], OPT[25], GPT2[26], BLOOM[27], ChatGLM[28], Baichuan[29] and Qwen[30], InternLM[31], Mistral[32], Mixtral MoE[33], etc. The aforementioned models can be easily adapted to integrate *Lookahead*, our well-designed framework, with only minor code modifications of approximately 20 lines, which can be found in our repository.

[1]https://huggingface.co/

Table 1: Comparison of different acceleration methods.

Methods	Accuracy-lossless	Training-or-Assis-Model-free	Multi-branch-draft	Low-cost-draft-generation
Quantization [1, 2]	×	-	-	-
Sparsity [3, 4]	×	-	-	-
Pruning [5, 6]	×	-	-	-
Distilling [7, 8]	×	×	-	-
Tensor-decomp [9, 10]	×	×	-	-
Early-exit [21, 22]	×	×	-	-
Block decoding [13]	✓	×	×	✓
Spec decoding [16]	✓	×	×	×
SpecInfer [18]	✓	×	✓	×
FREE [15]	✓	×	×	✓
LLMA [19]	✓	✓	×	✓
Medusa [14]	✓	×	✓	✓
LookaheadDecoding [20]	✓	✓	✓	×
Lookahead (ours)	✓	✓	✓	✓

Our contributions can be summarized as:

- We empirically quantify that the main bottleneck of LLM inference is the IO bandwidth, rather than the computation bound.
- We innovatively develop *Lookahead*, a framework that applies a hierarchical multi-branch draft strategy implemented with a trie tree to output more tokens per step than the traditional methods, in accelerating LLMs' inference.
- We extensively conduct experiments on both the industry and open source datasets and prove that *Lookahead* brings a significant improvement over the existing SOTA method in accelerating LLMs' inference.
- We elaborately adapt *Lookahead* to the most recent LLMs without any assistance of smaller models and have released our work with open source.

2 RELATED WORK

Recently, several strategies have been proposed and developed to enhance the inference speed of LLMs while maintaining the output quality within an acceptable range. One such strategy is the non-auto-regressive approach, specifically non-auto-regressive translation (NAT) [11], primarily employed in translation tasks [34–36]. However, it is essential to note that there are significant distinctions between translation tasks and general language model (LLM) scenarios, which may lead to subpar performance when applying the NAT strategy to LLM decoding.

To address this limitation, Huang et al. [35] introduce a layer-wise iterative method wherein each layer utilizes the decoding results and embeddings from the preceding layers. This approach involves training each layer through maximum likelihood estimation to predict the outcomes of subsequent decoding layers. On the other hand, Santilli et al. [12] formalized the standard greedy auto-regressive decoding strategy by employing a parallel Jacobi and Gauss-Seidel fixed-point iteration. It initializes the next tokens using special tokens and performs iterative decoding until convergence. However, all these methods may suffer from the risk of

accuracy degeneration, since the manipulated model deviates from its original version.

Recently, there has been a proposal for accuracy-lossless acceleration to enhance the auto-regressive decoding strategy as illustrated in Table 1. In this approach, a block-wise parallel decoding strategy was introduced by [13]. In this strategy, each subsequent token is independently and parallel predicted as a proposal using an additional transformer model, which consists of a multi-output feed-forward layer and can be fine-tuned or distilled for optimal performance. Then the proposals are directly compared against the output tokens generated by the original decoder. The longest verified tokens are then selected for acceptance as the current step's output. However, frequent failures during the verification process may be occur due to its reliance on a singular predictive branch. To overcome this drawback, Medusa [14] employs multiple heads to simultaneously predict drafts, enhancing the robustness of the process. On the other hand, FREE [15], uses the shallow layers of a model to generate drafts, instead of the final layer. to address this dependency. It proposals a synchronized parallel decoding strategy to ensure accuracy without loss. Despite this strategy's remarkable acceleration, more efforts are needed to train the extra layer.

To address the aforementioned issue, speculative decoding has been proposed [16–18, 37, 38]. These works utilize a smaller model as a draft predictor. For instance, the Bloom 7.1B model can be employed as a draft model for the larger 176B model. However, the proposed works above may face significant practical challenges. Primarily, the availability of a smaller-scale model, typically one-tenth the size of the larger model, is not always feasible for a range of model series, such as Llama-7B [24] and ChatGLM-6B [28]. Furthermore, larger models are often fine-tuned to specific applications, necessitating a parallel fine-tuning of the smaller helper model to produce comparable drafts. This requirement complicates the deployment and diminishes the system's accessibility.

To overcome this challenge, model-free prediction strategies have been introduced to achieve accurate predictions without relying on a specific model. One such strategy is presented in Ge et al. [39], which utilizes an input-guided method that copies content from the input sentence through prefix matching. Another strategy LLMA, proposed by Yang et al. [19], employs a prefix matching approach to retrieve content from either the input sentence or a document database. However, it is worth noting that the model-free prediction strategies mentioned above utilize tokens in a single draft manner, failing to fully utilize the GPUs. Recently, another training-free and assist-model-free method named LookaheadDecoding [20] explores the multi-branch strategy with employing Jacobi iteration and speculative decoding simultaneously. Nonetheless, this method incurs a substantial overhead associated with the generation of drafts, which consequently attenuates the potential for acceleration.

3 PRELIMINARY

3.1 Inference Speed

The inference speed V can be expressed as below.

$$V = \frac{L}{T} \propto \frac{L}{N \times t(l)} \tag{1}$$

$$N \propto \frac{L}{l} \tag{2}$$

Here, L denotes the overall generation tokens' length and T is the overall inference time, which is positively correlative to the overall consuming time for decoding: N indicates the overall decoding steps, $t(l)$ is the decoding time per step. It should be noted that $t(l)$ is nearly constant while l, namely the generated tokens' length per decoding step, is within a certain range, whose details and explanations will be introduced and discussed in the following subsection. Therefore, given the fixed L, the longer l is, the fewer N is needed, which in turn promises a higher inference speed V.

3.2 Step-Wise Decoding

Auto-regressive language models have been firstly introduced through following a step-wise decoding strategy: at each decoding step, the models concatenate the prompt sequence and the previous generated tokens and output the next single token using greedy-decoding, which selects the next token with the highest predicted probability over the vocabulary.

Though this strategy has been widely applied, the particular process promises only one single output token per decoding step ($l = 1$), which limits the overall inference speed.

3.3 Singe-Branch Strategy

Several most recent methodologies [13] [19] have been proposed to generate a sequence of tokens at each decoding step, with the purpose of promising a higher l to accelerate the LLMs' inference speed V. In these works, according to the prompt and the output tokens at the previous decoding step, a branch of tokens, named single-branch draft, have been obtained through the small model or the document reference, and efficiently validated by running the LLM in a single forward process.

Despite the single-branch strategy's success in accelerating LLMs' inference, the generated tokens' length per decoding step, namely l, cannot be guaranteed as long as we wish. The previous works [13] [19] empirically conclude that there is an upper limit for l and the inference speed V.

This can be explained through the successive validating mechanism: the validation process breaks once one token in the single-branch draft fails the validation, only the validated tokens in front of this failed token are accepted as the output. For the sake of brevity, we give a definition below:

Definition 1 Following the predicted next token, only the successive validated tokens from the beginning of the branch draft are kept and accepted as the output tokens, whose length is called the ***effective decoding length, EDL***.

Simply extending the single-branch draft's length over ***EDL*** not only fails to promise a longer output tokens' length per decoding step l and a higher inference speed V, but also wastes more computation. Apparently, how to achieve a longer ***EDL***, namely the ***effective decoding length***, is the key to further accelerate LLMs' inference.

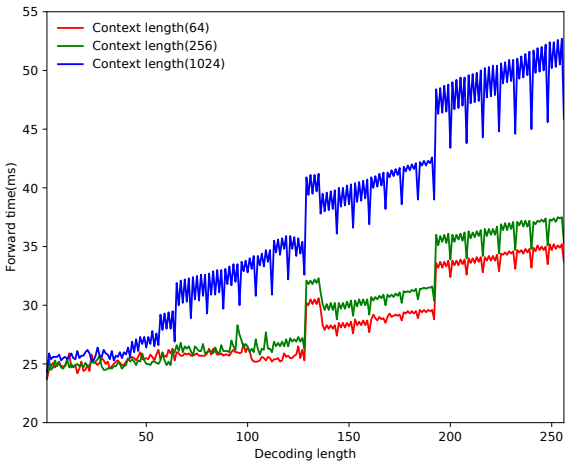

Figure 1: Decoding length's impact on the overall consuming time of LLMs' single forward process. Even the forward FLOPs is linear to the decoding length,

3.4 Discovering GPU's FLOPs Redundancy

To explore how far ***EDL*** can be extended at each decoding step, we conduct a case study through discussing the decoding length's impact on the overall consuming time of LLMs' single forward process, whose result can be found in Figure 1. It should be noted that we apply AntGLM-10B (which is trained from scratch with the GLM structure [23, 28]) model with single Nvidia A100 GPU as an example.

In Figure 1, it is evident that as the decoding length increases in the LLM's single forward process, the time consumption remains relatively constant in the initial stage, given a fixed context length. This observation can be attributed to several factors. Firstly, certain time requirements, such as kernel launching, operator loading, reading, and verification, are fixed and dependent on the GPU bandwidth. These factors are unrelated to the computational complexity or decoding length. Secondly, when the decoding length is relatively short, the resulting small matrix blocks may not take full advantage of CUDA and Tensor cores, indicating that GPU shows its FLOPs redundancy in this scenario. Consequently, the differences in overall time consumption between decoding lengths of 16 and 128 in the LLM's single forward process is negligible, provided the context length is 256. This finding supports the concept that the decoding time per decoding step, represented as $t(l)$, remains nearly constant within a specific range of generated token lengths l. However, when the decoding length exceeds 128 while the context length is 256 in the LLM's single forward process, the larger matrix block size introduces a more complex calculation logic for the tensor cores. Consequently, the overall inference time shifts to the second stage, leading to a gradual increase in overall inference time.

Definition 2 We define the maximum decoding length within which the inference consuming time of the single forward process is nearly constant as the ***critical decoding length, CDL***.

It can be clearly seen that there is still a "sufficient gap" between ***EDL***s empirically concluded previously [13] [19] and the ***CDL*** illustrated in Figure 1. In this scenario, we come up with two questions:

1. How to extend ***EDL*** further to accelerate the LLMs' inference as much as possible?

2. How to fully leverage the capabilities of GPUs, particularly considering our preliminary finding that GPUs exhibit FLOPs redundancy when the decoding length is within the ***CDL***?

Motivated by these questions, we propose a framework called *Lookahead* that aims to achieve a longer ***EDL*** by optimizing GPU utilization. The subsequent section provides detailed insights into this framework.

4 METHODS

4.1 Overview of Lookahead

We construct a framework, *Lookahead*, to accelerate LLMs' inference through developing and applying the *multi-branch strategy*.

In contrast to the single-branch strategy such as [19], which only considers one available draft without considering the existence of other drafts, the multi-branch strategy retrieves multiple drafts simultaneously. These drafts are then efficiently decoded and validated in parallel through the Verification and Accept (VA) process, progressively. The VA process then identifies the correct sub-sequence for each draft and retains the longest sub-sequence as the output tokens. Figure 2 presents the progress overview including various draft retrieving and verifying strategies.

The execution of the multi-branch strategy is based on a fundamental fact that when considering a sequence of tokens, there may exist multiple sequences of successive following tokens, each representing a potential branch draft. These sequences are referred to as *multi-branch draft* in our work, whose details will be introduced in the next sub-section.

4.2 Multi-Branch Draft

While developing and applying the multi-branch strategy, a concern naturally arises regarding how to efficiently deal with as many drafts as possible within the constraints of limited computational resources, specifically the ***CDL***.

4.2.1 Parallel Multi-Branch Draft. The drafts organized in a simple and straight forward parallel manner are referred to as *parallel multi-branch draft*, whose details can be found in Figure 2 and 3.

4.2.2 Hierarchical Multi-Branch Draft. Fortunately, through careful observation of the context, we have noticed that certain branches in the drafts have common prefix tokens. For example, in Figure 2, the branches **[on, my, knee]** and **[on, a, table]** both share the prefix token **[on]**. By leveraging this observation, we can compress and organize the multi-branch draft using a hierarchical structure, allowing for the inclusion of additional branches while maintaining the same token capacity. To achieve this, we recursively merge the prefix token(s) shared by multiple branches, and then append the remaining tokens of each branch. We refer to these compressed and organized drafts as *hierarchical multi-branch draft*.

As illustrated in Figure 2 describing the process of an inference step, through the single-branch strategy, the single draft **[on, my, knee]** is appended to the next token **[sits]** directly for the single forward process in VA, which in turn gives the output **[on, a]**. Being benefited from the parallel multi-branch draft, two branches **[on, my, knee]** and **[on, a, table]** are organized in parallel and

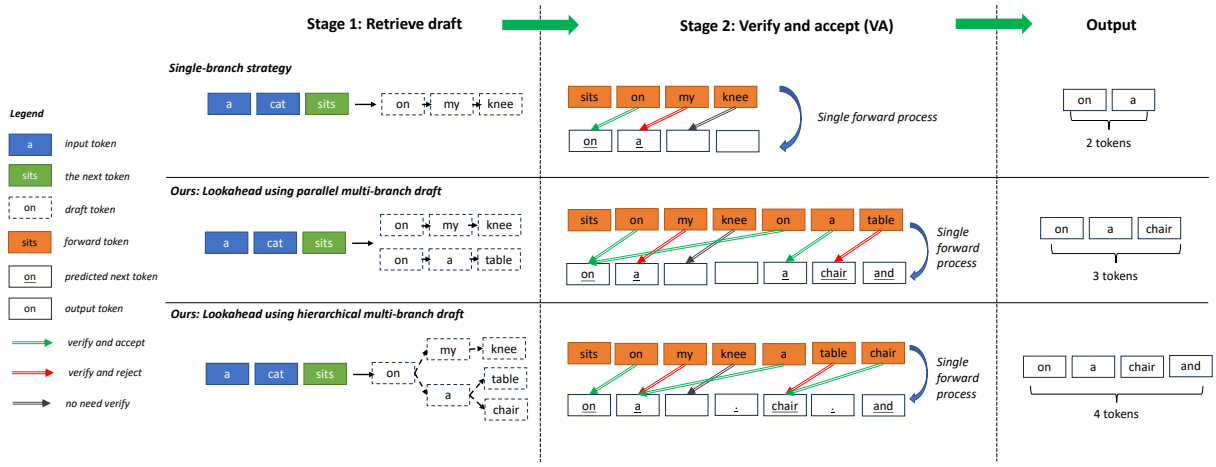

Figure 2: Overview of the drafts retrieving and the Verification and Accept (VA) process using various strategies.

appended to the next token **[sits]** for verifying, which outputs **[on, a, chair]**. By utilizing the hierarchical multi-branch draft, the shared prefix token **[on]** are merged, after which the token list **[my, knee, a, table]** are appended. By doing so, given the same decoding length, we are able to save an additional space to append one more token **[chair]** from another branch **[on, a, chair]**. Combined with the next token, we collect the token list **[sits, on, my, knee, a, table, chair]** for decoding through the single forward process and output **[on, a, chair, and]**. Therefore, compared to the single-branch strategy, our multi-branch strategy, particularly the hierarchical multi-branch draft, offers the advantage of retrieving multiple drafts, resulting in improved ***EDL*** from a statistical perspective and thus significantly enhancing LLMs' inference speed. As illustrated in Figure 3, the position IDs and causal masks in a transformer-based model [40] are also merged to align with the merged token list. By doing so, we are able to accommodate more branches using *hierarchical multi-branch draft*, compared to *parallel multi-branch draft*. We summarize the workflow of *Lookahead* with pseudo code in the Algorithm 1.

4.3 Trie-tree-based Draft Retrieval

To enable *hierarchical multi-branch draft*, we utilize a trie tree [41, 42], a data structure that is widely used for efficient retrieval and storage purposes and handles prefix matching by organizing nodes as individual characters or words. In our work, each node of the trie tree represents a token ID, and a path from the root to a leaf node represents a branch. The trie tree is initialized when a model is loaded, and it is alive until the model instance is shut down.

Before and after each step consisting of the draft retrieving process and the VA process as illustrated in Figure 2, a global trie tree will be updated through multiple procedures, which will be introduced in the sub-section 4.3.1. During the draft retrieving process, the trie tree will be retrieved to provide the drafts, whose details will be introduced in the sub-section 4.3.2.

4.3.1 Trie Tree Updating. We apply the branch inserting, branch eliminating and node pruning to update the trie tree.

Algorithm 1 Lookahead with multi-branch draft

Input: decoding length L_d, branch length L_b, minimal count without re-retrieving N_m, node capacity of trie tree N_{max}, start token id S
Output: O

initialization: trie tree $\mathcal{T}$
repeat
  initialization: output id list O=[S], KV cache C_{kv} = []
  tokenize a query to token list T
  for $i \leftarrow$ 1 to len(T)-1 **do** ▷ branch inserting
    insert $T[i : i + L_b]$ into $\mathcal{T}$
    if Node count of T > N_m **then**
      do node pruning
    end if
  end for
  repeat
    for $j \leftarrow len(O)$ to 1 **do** ▷ trie tree retrieving
      obtain a prefix token list $T_p \leftarrow O[-j :]$
      match a sub trie tree $\mathcal{T}_s$ from $\mathcal{T}$ with T_p
      if Node count of T_s < N_m **then**
        continue
      end if
      select L_d-largest frequency nodes N_{max} from $\mathcal{T}_s$
    end for
    prepare token ids T_{fp}, position ids P_{fp} and causal masks M_{fp} for N_{max}
    get next tokens prediction via $LLM(C_{kv}, T_{fp}, P_{fp}, M_{fp})$
    verify each branches and accept verified tokens T_v
    append T_v to O
    incrementally put branches of O to $\mathcal{T}$
    rearrange KV cache C_{kv} with accepted tokens
  until meet stopping criteria
  eliminate frequency of branches from the current prompt
until all queries are processed

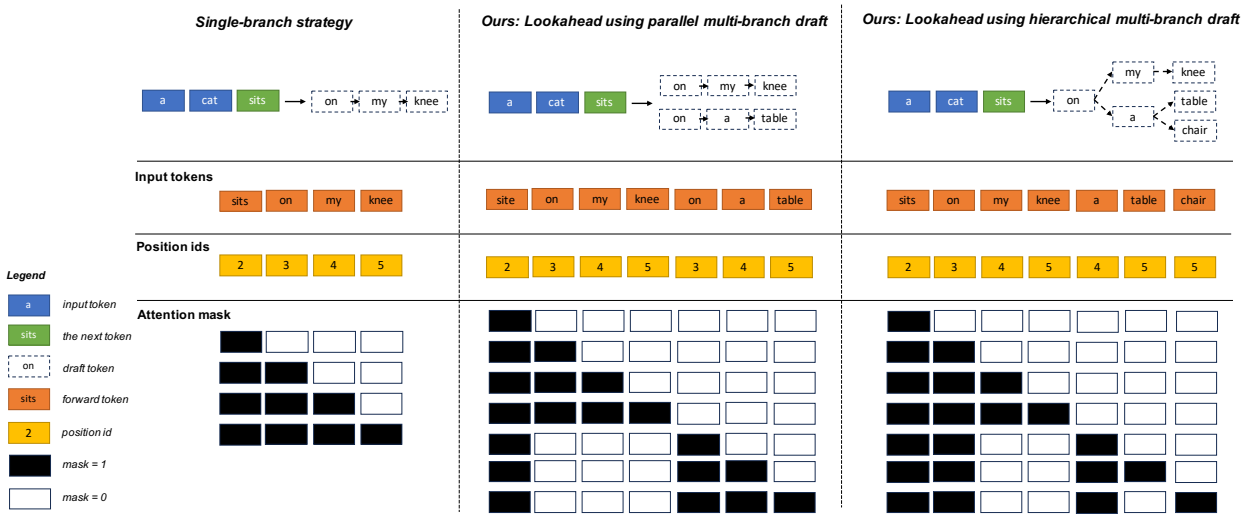

Figure 3: The input ids, position ids and causal masks for forwarding using various strategies.

Prompt branch Inserting. Considering that in several typical scenarios (e.g., RAG, Summary), the output are often derived from the prompt, we thus manipulate the prompt into the branches and insert them into the trie tree.

Generated branch Inserting. Different from LLMA which only uses branches from prompts, we also reuse the branch from generated tokens. We find that, towards various prompts, there may be similar outputs, e.g., "keep exercising" in answering "how to keep healthy" and "how to be stronger". Moreover, we also find that some tokens may be repeated in a response, from example, "numbers.push_back" may occur several times in a code-generation response. To benefit from the repeat, we put the generated branches into the trie tree on-the-fly instead of the final step.

Branch Eliminating. When the current generation in answering the prompt is finished after multiple steps, the branches derived from this prompt are eliminated, considering that these branches may not be relevant to the generation in answering the other prompts.

Node Pruning. To maintain the trie tree within a moderate size, in case that the oversized trie tree results in high memory consumption and slow retrieval performance, we develop a pruning strategy. To achieve this, we decay the branch frequency and remove the nodes with frequency less than 1, when the trie tree exceeds a predetermined capacity. By doing so, we are able to optimize memory consumption and improve retrieval performance in our framework.

The analysis of the above updating procedures will be discussed in the experiment section.

4.3.2 Trie Tree Retrieving. **Multi-stage Retrieval.** We can extract a sub tree from a trie tree by providing a prefix, which is essentially a list of tokens. The sub tree is also a trie tree and can be directly used for the hierarchical multi-branch draft. The length of the prefix influences the number and relevance of the retrieved branches. Short prefixes yield a greater number of branches, while longer prefixes result in more related branches. To strike a balance between the count and correlation of the branches, we adopt a multi-stage retrieval strategy inspired by [19]. Specifically, we begin by attempting to match a longer prefix. If the number of tokens associated with the matched branches is significantly smaller than the ***CDL***, we reduce the length of the prefix and retry the matching process until we obtain a substantial number of tokens linked to matched branches. If the count of the matched branches falls below a predefined threshold, we utilize all of them for the VA process. However, if the count exceeds a given size, we choose the tokens with the highest frequency.

Branch Weighting. For token sorting by frequency, we employ a weighted scheme that considers both prompt and response frequencies. Intuitively, branches from an input prompt may be more related to current generation than that of other responses. To prioritize the importance of branches derived from the input prompt, we amplify their frequency by a significant factor.

5 EXPERIMENTS

5.1 Experimental Setup

We conduct a solid experiment to evaluate the efficacy, robustness and practicality of our *Lookahead* framework in accelerating LLMs' inference. Our inference environment includes a server with 32-core CPU, 64 GB host memory, and various GPUs. Considering the actual industry scenarios that *Lookahead* is applied, AntRAG is chosen as the evaluation dataset. AntRAG dataset is an internal benchmark dataset that is meticulously collected from a real-life product system in Alipay. The dataset collection process involves submitting user queries to a search engine, which then retrieves the most relevant document alongside the original query. These query-document pairs are combined to form prompts. To ensure the dataset's quality and coherence, we employ a rigorous process of removing any duplicate or unrelated prompts, resulting in a refined and comprehensive dataset. Furthermore, to validate the robustness and practicality of *Lookahead* in other scenarios, we additionally conduct experiments utilizing the Dolly dataset

(an open domain QA dataset) [2], GSM8k dataset (a dataset of 8.5K high quality linguistically diverse grade school math word problems) [3], and HumanEval-x dataset (a dataset of Python, C++, Java, JavaScript, and Go code tasks) [4]. Detailed information regarding the datasets, base models and devices utilized can be found in the Appendix. For inference speed evaluation, each dataset's test set is utilized, with the corresponding development set employed for warm-up purposes. It is important to clarify that this warm-up procedure is implemented solely to ensure the convergence and accuracy of performance metrics, as detailed in the Appendix, and is not included in real-world deployment scenarios. In evaluating the candidate methods, which all purport lossless accuracy in generation, inference speed, quantified as output tokens generated per second, serves as the sole metric. Optimal hyper-parameters are determined through a grid search, with the grid size deliberately set as multiples of four to enhance time efficiency.

5.2 Results

5.2.1 Lookahead's superior performance in accelerating inference speed. Table 2 exhibits the inference speeds achieved by different acceleration methods, whose mean values are then selected as the performance indicator for each acceleration method.

As depicted in Table 2, our *Lookahead* obtain significant improvement over other methods, on various models, datasets and devices. The average inference speed of AntGLM-10B towards AntRAG is recorded at 52.4 tokens/s on A100 GPU. However, with the incorporation of LLMA, this speed is elevated to 165.4 tokens/s, resulting in a notable 3.16 times speed improvement. Our *Lookahead* with hierarchical multi-branch draft via trie tree propels the average inference speed even further, culminating in a 5.36 times speed-up, surpassing LLMA by 70% in terms of acceleration. *Lookahead* also obtains similar acceleration with AntRAG dataset on less-powerful devices, such as A10 and V100, which are widely used for deployment. Despite of RAG scenarios, *Lookahead* also consistently demonstrates its remarkable superiority and applicability across diverse datasets, especially with speedup of 3.92 on the HumanEval-x dataset, further emphasizing its practicality in real-world scenarios.

5.2.2 Hyper-parameters in multi-branch draft. In continuation of AntGLM-10B, we delve deeper into the analysis of the decoding and branch lengths, two key hyper-parameters within our *Lookahead* framework. In particular, we empirically examine their impact on the inference speed, as illustrated in Figure 4.

Generally speaking, it can be observed that as the decoding and branch lengths increase, there is an upward trend in the overall inference speed. As mentioned, with the single-branch strategy, LLMA is unable to further enhance the inference speed while its branch length surpasses 25. This limitation arises due to its inability to ensure a longer ***EDL***. By implementing the multi-branch strategy, *Lookahead* is able to retrieve multiple branch drafts and thus guarantee a longer ***EDL*** after the VA process, as illustrated in Figure 5. Consequently, this leads to a significantly higher inference speed, while maintaining the same branch length as LLMA in Figure 4. Furthermore, the improved ***EDL*** as shown in Figure 5

[2]https://huggingface.co/datasets/databricks/databricks-dolly-15k
[3]https://huggingface.co/datasets/gsm8k
[4]https://huggingface.co/datasets/THUDM/HumanEval-x

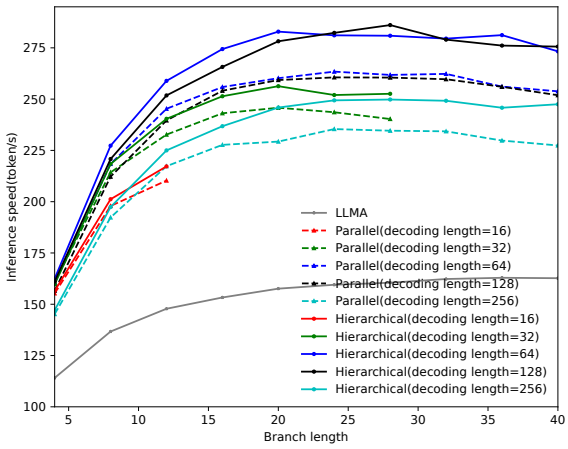

Figure 4: The decoding and branch length's impact on the LLM's inference speed using various accelerations.

also provides more headroom for increasing the branch length and thereby enhancing the upper limit of the inference speed.

In addition, Figure 5 shows that being benefited from dealing with more branch drafts, *Lookahead* using the hierarchical multi-branch draft promises the advantage of a longer ***EDL*** and consequently improved inference speed, compared to using the parallel multi-branch draft with the identical branch and decoding lengths. It should be noted that in Figure 5, improving the decoding length promises a longer ***EDL*** using *Lookahead*, however the oversized decoding length that surpasses the ***CDL*** fails to promise a higher inference speed, due to a more complex calculation logic for the tensor cores that has been introduced in the preliminary.

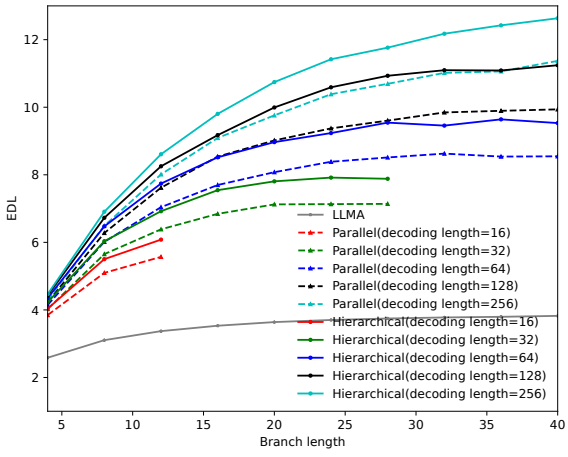

Figure 5: The decoding and branch length's impact on the *effective decoding length*, *EDL* using various accelerations.

5.2.3 Hyper-parameters in trie tree updating. Table 3 showcases the inference speeds of *Lookahead*, which exhibit variability depending on different procedures used for updating the trie tree.

Table 2: Inference Speed(token/s) of different methods on various models, datasets and devices. As LookaheadDecoding only offers implementation for Llama, we do not conduct experience on AntGLM-10B Model.

model	dataset	device	transformers	vLLM	LLMA	LookaheadDecoding	lookahead(ours)
AntGLM-10B	AntRAG	A100-80G	52.4	52.06(x0.99)	165.4(x3.16)	-	280.9(x5.36)
AntGLM-10B	AntRAG	A10	20.3	20.29(x1.00)	59.5(x2.93)	-	105.1(x5.18)
AntGLM-10B	AntRAG	V100-32G	27.3	27.28(x1.00)	64.3(x2.36)	-	118.9(x4.36)
Llama-7B	Dolly	A100-80G	50.4	91.04(x1.81)	60.7(x1.20)	66.7(x1.32)	106.8(x2.12)
Llama-7B	Dolly	A10	31.4	32.58(x1.04)	35.8(x1.14)	38.2(x1.22)	55.7(x1.77)
Llama-7B	GSM8k	A100-80G	41.4	92.09(x2.22)	69.0(x1.67)	94.3(x2.28)	111.3(x2.69)
Llama-7B	GSM8k	A10	31.4	32.73(x1.04)	41.1(x1.31)	48.9(x1.56)	68.1(x2.17)
Llama-7B	HumanEval-x	A100-80G	51.1	90.47(x1.77)	66.5(x1.30)	88.3(x1.73)	161.5(x3.16)
Llama-7B	HumanEval-x	A10	30.9	32.46(x1.05)	42.1(x1.36)	46.0(x1.49)	89.6(x2.90)
Llama-13B	Dolly	A100-80G	39.9	51.67(x1.29)	59.0(x1.48)	45.2(x1.13)	84.6(x2.12)
Llama-13B	Dolly	V100-32G	20.5	22.07(x1.08)	23.9(x1.17)	23.2(x1.13)	35.2(x1.72)
Llama-13B	GSM8k	A100-80G	42.9	52.06(x1.21)	51.8(x1.21)	64.8(x1.51)	103.4(x2.41)
Llama-13B	GSM8k	V100-32G	22.0	22.43(x1.02)	25.8(x1.17)	27.8(x1.26)	45.6(x2.07)
Llama-13B	HumanEval-x	A100-80G	35.0	51.49(x1.47)	57.7(x1.65)	66.1(x1.89)	137.3(x3.92)
Llama-13B	HumanEval-x	V100-32G	21.5	22.33(x1.04)	28.8(x1.34)	27.5(x1.28)	57.0(x2.65)

These procedures include branch inserting with prompt and/or output, branch eliminating, and node pruning. In comparison to the results obtained without a specific procedure, *Lookahead* attains its optimal performance when all the aforementioned procedures are applied to dynamically update the trie tree, highlighting the necessity of these procedures in updating the trie tree.

Table 3: The inference speed of different updating procedures was evaluated, considering various conditions. In the evaluation, "W/o prompt" and "W/o output" refer to the absence of branching from the prompt and output, respectively. "W/o Pruning" signifies the exclusion of node count reduction in an oversized trie tree through pruning, "W/o eliminating" denotes the retention of branches derived from the prompt before processing the subsequent query.

Condition	W/o prompt	W/o output	W/o pruning	W/o elimin	Lookahead
Token/s	234.5	202.0	221.4	234.2	280.9

The capacity of the trie tree has a significant impact on the acceleration performance. To determine the optimal capacity, we conduct empirical experiments, as outlined in Table 4. Instead of using a fixed value, we normalize the node capacity in relation to the decoding length, which allows us to identify a suitable hyperparameter that performs well across different decoding lengths. Based on our findings, we conclude that node capacities of 16 and 32 times the decoding length yield the best results. Consequently, we have set the default capacity in our framework to be 16 times the decoding length. We also examine the retrieving and updating time of trie tree with different capacity, the time is negligible when compared with the forward time.

5.2.4 Lookahead's efficient memory usage. Figure 5 shows the peak GPU memory consumption for various decoding lengths (1, 2, 4, 8, 16, 32, 64, 128). The decoding length of 1 represents the experience

Table 4: Inference Speed and retrieving/updating time of different capacity of trie tree. n*DL denotes the capicity is n times the decoding length.

Capacity	1*DL	2*DL	4*DL	8*DL	16*DL	32*DL	64*DL	128*DL
Token/s	254.5	258.7	268.8	273.3	280.7	280.8	279.5	278.3
Retrieve(ms)	0.81	1.35	1.22	1.46	1.02	1.29	1.75	2.38
Update(ms)	0.10	0.12	0.11	0.11	0.12	0.12	0.15	0.16

where no *Lookahead* is used, resulting in a GPU memory consumption of 20.25 GiB. Surprisingly, even with a decoding length of 128, the use of *Lookahead* only leads to a negligible increase in GPU memory consumption of 0.6%. This minimal increase in memory usage is practically negligible and has no significant impact on real-world applications. Furthermore, we have also evaluated the CPU memory consumption of the trie tree. With the AntRAG dataset, the trie tree only utilizes a mere 260 MiB of memory. This value is negligible when compared to the total memory capacity of a mainstream server.

Table 5: Inference peak memory of different decoding lengths.

Decoding length	1	2	4	8	16	32	64	128
Memory(GiB)	20.25	20.27	20.27	20.27	20.27	20.29	20.38	20.39

5.2.5 Lookahead's side note. Theoretically, Lookahead will not intervene the original output. However, due to the operators' non-deterministic characteristics and the low precision data format (e.g. fp16) employed by the LLMs in practice, Lookahead applying different operators may result in minor inconsistent outputs rarely, compared to the original output. We carried out additional experiments, to illustrate the effects of these inconsistencies on

Table 7: Inference Speed of product scenarios. The speed is measured by mean latency (seconds) of a query.

Scenario	Baseline	Lookahead	Speedup
CHART2JSON	12.97	2.07	6.26x
Citizen Biz Agent	1.67	0.32	5.21x
Enterprise Info QA	14.01	2.74	5.11x
Health Suggestion	12.41	2.66	4.66x
Medical Report Summary	3.33	1.25	2.66x

Table 8: Inference Latency with Lookahead for vLLM.

Scenario	vLLM	vLLM+lookahead	speedup
Virtual Human Script	5.109	3.203	1.60

the metrics of accuracy. It can be concluded from the comparative experiments that Lookahead doesn't change the baseline model's output accuracy with low precision data format, in terms of MMLU, CEval and so on.

Table 6: Accuracy with low precision data format(fp16).

Dataset	Original(%)	Lookahead(%)
mmlu	48.19	48.20(+0.01)
mmlu-humanities	52.69	52.72(+0.03)
mmlu-stem	37.94	37.95(+0.01)
mmlu-social-science	55.82	55.76(-0.06)
mmlu-other	51.61	51.69(+0.08)
ceval	35.71	35.77(+0.06)
ceval-stem	32.58	32.58(+0.0)
ceval-social-science	40.27	40.57(+0.30)
ceval-humanities	36.95	36.95(+0.0)
ceval-other	36.01	36.01(+0.0)
ceval-hard	28.44	28.44(+0.0)

Generally speaking, Lookahead leverages a hierarchical multi-branch strategy to generate more tokens (statistical expectation in comparison to the baseline) per decoding step, with only a marginal increase in time consumed. Therefore, in the worst-case, Lookahead not only outputs a single token as the conventional process, but also requires slight additional time consumption, due to draft retrieving, forward validation and key-value cache withdraw. Through our benchmark with forced no draft acceptation to deliberately simulate this worst-case scenario, Lookahead preserved 87.5% speed of no lookahead inference with AntGLM-10B and 91.9% with Llama-13B. However, according to our examination, the worst-case mentioned above happens rarely.

5.3 Online Deployment

Our framework has been widely used in real-world scenarios of Alipay since **April 2023**, due to its accessibility, accuracy-lossless and remarkable acceleration. To be concise, we only report 5 typical scenarios in Table 7. CHART2JSON is a scenario that uses a multi-modal model to convert a chart image to structured content with JSON format, it can achieve extraordinary acceleration due to plenty of template tokens in JSONs. Citizen biz agent, enterprise info QA and health suggestion are RAG scenarios, they are used for answering user questions with reference documents. Medical report summary is a scenario to summarize the content (texts after OCR) of a medical report image. With the assistance of our lookahead framework, the latency of all scenarios is decreased significantly while the generation results are the same as the original step-by-step strategy.

Meanwhile, we have integrated lookahead into other frameworks and obtained additional improvement. vLLM [43] is a widely used framework with state-of-the-art serving throughput, however we may suffer from its huge latency even when only one query is processing. We have implemented lookahead based on vLLM for the single-query situation and obtain 1.6 times acceleration on a real-life scenario about script generation for virtual human, shown in table 8.

6 CONCLUSION

In our work, we empirically quantify that the main bottleneck of LLM inference is the IO bandwidth, rather than the FLOPs. Inspired by this, to take full advantage of the GPU's FLOPs redundancy, we innovatively develop *Lookahead*, a generic framework that applies a hierarchical multi-branch draft strategy implemented with a trie tree to output more tokens per step than the traditional methods. We conduct extensive experiments and demonstrate that *Lookahead* gains a substantial inference acceleration and cost reduction, with lossless generation accuracy. By simply adapting our framework to the latest LLMs, we have achieved a wide range of applications and promising prospects and will soon release our work with open source.

REFERENCES

[1] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. Awq: Activation-aware weight quantization for llm compression and acceleration. *ArXiv*, abs/2306.00978, 2023.

[2] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *ArXiv*, abs/2210.17323, 2022.

[3] Yucheng Lu, Shivani Agrawal, Suvinay Subramanian, Oleg Rybakov, Chris De Sa, and Amir Yazdanbakhsh. Step: Learning n: M structured sparsity masks from scratch with precondition. *ArXiv*, abs/2302.01172, 2023.

[4] Aojun Zhou, Yukun Ma, Junnan Zhu, Jianbo Liu, Zhijie Zhang, Kun Yuan, Wenxiu Sun, and Hongsheng Li. Learning n: m fine-grained structured sparse neural networks from scratch. *arXiv preprint arXiv:2102.04010*, 2021.

[5] Qingru Zhang, Simiao Zuo, Chen Liang, Alexander Bukharin, Pengcheng He, Weizhu Chen, and Tuo Zhao. Platon: Pruning large transformer models with upper confidence bound of weight importance. In *International Conference on Machine Learning*, pages 26809–26823. PMLR, 2022.

[6] François Lagunas, Ella Charlaix, Victor Sanh, and Alexander M Rush. Block pruning for faster transformers. *arXiv preprint arXiv:2109.04838*, 2021.

[7] Hyojin Jeon, Seungcheol Park, Jin-Gee Kim, and U Kang. Pet: Parameter-efficient knowledge distillation on transformer. *Plos one*, 18(7):e0288060, 2023.

[8] Cheng-Yu Hsieh, Chun-Liang Li, Chih-Kuan Yeh, Hootan Nakhost, Yasuhisa Fujii, Alexander Ratner, Ranjay Krishna, Chen-Yu Lee, and Tomas Pfister. Distilling step-by-step! outperforming larger language models with less training data and smaller model sizes. *arXiv preprint arXiv:2305.02301*, 2023.

[9] Xindian Ma, Peng Zhang, Shuai Zhang, Nan Duan, Yuexian Hou, Ming Zhou, and Dawei Song. A tensorized transformer for language modeling. *Advances in neural information processing systems*, 32, 2019.

[10] Maolin Wang, Yu Pan, Xiangli Yang, Guangxi Li, and Zenglin Xu. Tensor networks meet neural networks: A survey. *arXiv preprint arXiv:2302.09019*, 2023.

[11] Jiatao Gu, James Bradbury, Caiming Xiong, Victor O. K. Li, and Richard Socher. Non-autoregressive neural machine translation. *ArXiv*, abs/1711.02281, 2017.

[12] Andrea Santilli, Silvio Severino, Emilian Postolache, Valentino Maiorca, Michele Mancusi, Riccardo Marin, and Emanuele Rodolà. Accelerating transformer inference for translation via parallel decoding, 2023.

[13] Mitchell Stern, Noam Shazeer, and Jakob Uszkoreit. Blockwise parallel decoding for deep autoregressive models. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 10107–10116, Red Hook, NY, USA, 2018. Curran Associates Inc.

[14] Tianle Cai, Yuhong Li, Zhengyang Geng, Hongwu Peng, Jason D. Lee, Deming Chen, and Tri Dao. Medusa: Simple llm inference acceleration framework with multiple decoding heads, 2024.

[15] Sangmin Bae, Jongwoo Ko, Hwanjun Song, and Se-Young Yun. Fast and robust early-exiting framework for autoregressive language models with synchronized parallel decoding. In Houda Bouamor, Juan Pino, and Kalika Bali, editors, *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 5910–5924, Singapore, December 2023. Association for Computational Linguistics.

[16] Heming Xia, Tao Ge, Si-Qing Chen, Furu Wei, and Zhifang Sui. Speculative decoding: Lossless speedup of autoregressive translation, 2023.

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

[18] Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, Xinhao Cheng, Zeyu Wang, Zhengxin Zhang, Rae Ying Yee Wong, Alan Zhu, Lijie Yang, Xiaoxiang Shi, Chunan Shi, Zhuoming Chen, Daiyaan Arfeen, Reyna Abhyankar, and Zhihao Jia. Specinfer: Accelerating generative large language model serving with tree-based speculative inference and verification, 2024.

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

[20] Yichao Fu, Peter Bailis, Ion Stoica, and Hao Zhang. Breaking the sequential dependency of llm inference using lookahead decoding, November 2023.

[21] Xiaonan Li, Yunfan Shao, Tianxiang Sun, Hang Yan, Xipeng Qiu, and Xuanjing Huang. Accelerating bert inference for sequence labeling via early-exit. *arXiv preprint arXiv:2105.13878*, 2021.

[22] Ji Xin, Raphael Tang, Jaejun Lee, Yaoliang Yu, and Jimmy Lin. Deebert: Dynamic early exiting for accelerating bert inference. *arXiv preprint arXiv:2004.12993*, 2020.

[23] Zhengxiao Du, Yujie Qian, Xiao Liu, Ming Ding, Jiezhong Qiu, Zhilin Yang, and Jie Tang. Glm: General language model pretraining with autoregressive blank infilling. In *Annual Meeting of the Association for Computational Linguistics*, 2021.

[24] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models, 2023.

[25] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona T. Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. Opt: Open pre-trained transformer language models. *ArXiv*, abs/2205.01068, 2022.

[26] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.

[27] Teven Le Scao, Angela Fan, and etc. Bloom: A 176b-parameter open-access multilingual language model. *ArXiv*, abs/2211.05100, 2022.

[28] Aohan Zeng, Xiao Liu, Zhengxiao Du, Zihan Wang, Hanyu Lai, Ming Ding, Zhuoyi Yang, Yifan Xu, Wendi Zheng, Xiao Xia, Weng Lam Tam, Zixuan Ma, Yufei Xue, Jidong Zhai, Wenguang Chen, P. Zhang, Yuxiao Dong, and Jie Tang. Glm-130b: An open bilingual pre-trained model. *ArXiv*, abs/2210.02414, 2022.

[29] Ai Ming Yang et al. Baichuan 2: Open large-scale language models. *ArXiv*, abs/2309.10305, 2023.

[30] Jinze Bai, Shuai Bai, Yunfei Chu, Zeyu Cui, Kai Dang, Xiaodong Deng, Yang Fan, Wenhang Ge, Yu Han, Fei Huang, Binyuan Hui, Luo Ji, Mei Li, Junyang Lin, Runji Lin, Dayiheng Liu, Gao Liu, Chengqiang Lu, K. Lu, Jianxin Ma, Rui Men, Xingzhang Ren, Xuancheng Ren, Chuanqi Tan, Sinan Tan, Jianhong Tu, Peng Wang, Shijie Wang, Wei Wang, Shengguang Wu, Benfeng Xu, Jin Xu, An Yang, Hao Yang, Jian Yang, Jian Yang, Shusheng Yang, Yang Yao, Bowen Yu, Yu Bowen, Hongyi Yuan, Zheng Yuan, Jianwei Zhang, Xing Zhang, Yichang Zhang, Zhenru Zhang, Chang Zhou, Jingren Zhou, Xiaohuan Zhou, and Tianhang Zhu. Qwen technical report. *ArXiv*, abs/2309.16609, 2023.

[31] InternLM Team. Internlm: A multilingual language model with progressively enhanced capabilities. https://github.com/InternLM/InternLM, 2023.

[32] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023.

[33] Albert Q. Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, Gianna Lengyel, Guillaume Bour, Guillaume Lample, Lélio Renard Lavaud, Lucile Saulnier, Marie-Anne Lachaux, Pierre Stock, Sandeep Subramanian, Sophia Yang, Szymon Antoniak, Teven Le Scao, Théophile Gervet, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mixtral of experts, 2024.

[34] Jungo Kasai, Nikolaos Pappas, Hao Peng, James Cross, and Noah A. Smith. Deep encoder, shallow decoder: Reevaluating non-autoregressive machine translation, 2021.

[35] Chenyang Huang, Hao Zhou, Osmar R Zaiane, Lili Mou, and Lei Li. Non-autoregressive translation with layer-wise prediction and deep supervision. *ArXiv*, abs/2110.07515, 2021.

[36] Chitwan Saharia, William Chan, Saurabh Saxena, and Mohammad Norouzi. Non-autoregressive machine translation with latent alignments. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1098–1108, Online, November 2020. Association for Computational Linguistics.

[37] Zhihao Zhang, Alan Zhu, Lijie Yang, Yihua Xu, Lanting Li, Phitchaya Mangpo Phothilimthana, and Zhihao Jia. Accelerating retrieval-augmented language model serving with speculation, 2024.

[38] Jimin Hong, Gibbeum Lee, and Jaewoong Cho. A simple framework to accelerate multilingual language model for monolingual text generation, 2024.

[39] Tao Ge, Heming Xia, Xin Sun, Si-Qing Chen, and Furu Wei. Lossless acceleration for seq2seq generation with aggressive decoding, 2022.

[40] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Neural Information Processing Systems*, 2017.

[41] Rene De La Briandais. File searching using variable length keys. In *IRE-AIEE-ACM Computer Conference*, 1959.

[42] Paolo Ferragina, Marco Frasca, Giosuè Cataldo Marinò, and Giorgio Vinciguerra. On nonlinear learned string indexing. *IEEE Access*, 11:74021–74034, 2023.

[43] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Haotong Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. *Proceedings of the 29th Symposium on Operating Systems Principles*, 2023.

[44] Gyeong-In Yu and Joo Seong Jeong. Orca: A distributed serving system for transformer-based generative models. In *USENIX Symposium on Operating Systems Design and Implementation*, 2022.

[45] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

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

A DATASET, MODEL AND DEVICE SUMMARY

The experimental setup, including the datasets, models, and devices employed, is delineated in Table 9, Table 10, and Table 11, respectively. Specifically, the Llama2-7b-chat and Llama2-13b-chat models are utilized for the Dolly and GSM8k datasets, while the CodeLlama-7b and CodeLlama-13b models, which are fine-tuned on code dataset with Llama models, are applied to the HumanEval-x dataset. The prepossessed datasets are available in our repository.

Table 9: Summary of datasets.

Dataset	Split	#samples	#tokens(prompt)	#tokens(answer)
AntRAG	dev	7,605	243.1	81.9
	test	1,000	241.0	82.0
Dolly	dev	13,850	298.3	101.8
	test	1,000	301.5	104.8
GSM8k	dev	7,792	66.9	130.8
	test	1,000	67.9	131.9
HumanEval-x	dev	410	187.0	125.9
	test	410	139.8	82.1

Table 10: Summary of models.

Model	AntGLM-10B	Llama-7B	Llama-13B
Params	10.14B	6.74B	12.71B
Vocab size	115328	32000	32000
Layer	48	32	40
Hidden size	4096	4096	5120
Attention head	32	32	40
MLP size	16384	11008	13824

Table 11: Summary of devices.

Device	A100-80G	A10	V100-32G
Memory	80G HBM2e	24G GDDR6	32G HBM2
Memory bandwidth	2,039GB/s	600GB/s	900GB/s
FLOPs(FP16)	312T	125T	125T

B BATCH INFERENCE

The experiments described were conducted under single-query conditions (i.e., a batch size of 1). It is essential to recognize that batch inference (i.e., batch size > 1) incurs greater computational demand than single-query inference. To ascertain the computational threshold for batch inference, we evaluated the forward time across varying batch sizes, context lengths, and decoding lengths, as depicted in Figure 6. It can be seen that while the batch size is 4 for instance, an increase in decoding length from 5 to 32 does not substantially augment the forward time. Moreover, forward time still remains manageable for a decoding length increasing from 5 to 16 while the batch size is 16. The GPUs still show their redundancies in these scenarios and lookahead in batch inference could still be applied as an effective approach.

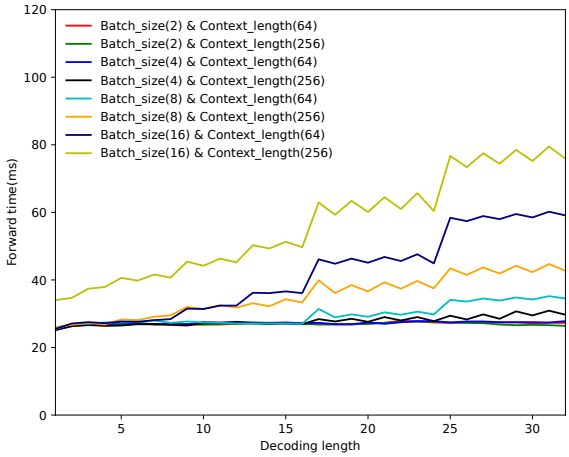

Figure 6: GPU Memory of various batch size, context length and decoding length.

To implement *lookahead* within batch inference parameters, one must navigate variances in KV cache sequence length and attention masks, with implementation specifics available in the repository. To our knowledge, this constitutes the first batch implementation of speculative decoding methodologies. As evidenced in Table 12, *lookahead* retains its efficacy even with non-unitary batch sizes, demonstrating a notable increase in inference speed relative to traditional step-by-step and LLMA-based batch inference approaches. Despite the less pronounced speedup compared to unitary batch scenarios, two primary factors contribute to this: first, large batch sizes decrease the GPU's redundancy, particularly with extensive context and decoding lengths, thereby capping potential speed enhancements. Second, the KV cache lengths' heterogeneity within a batch introduces extra computational overhead in attention operations, diminishing inference velocity.

Nonetheless, it is critical to emphasize that Lookahead is tailored for time-sensitive environments rather than those sensitive to throughput. Future work includes optimizing throughput by incorporating continuous batching [44] and high-efficiency attention mechanisms [43, 45].

Table 12: Inference Speed of different batch sizes with the AntRAG dataset.

Method	Batch size=2		Batch size=4	
	token/s	speedup	token/s	speedup
Baseline	68.0	1.00x	88.2	1.00x
LLMA	185.2	2.72x	214.0	2.43x
Lookahead	285.5	4.20x	299.5	3.40x

C WARMUP ANALYSIS

To accurately measure inference speed, a warm-up strategy is employed where responses from the development set are preloaded into the trie tree. As illustrated in Figure 7, inference speed is positively correlated with the number of warm-up samples. Given that a model may process hundreds of thousands of requests over its lifespan, it is reasonable to infer that the average speed would exceed that observed during the initial thousands of requests.

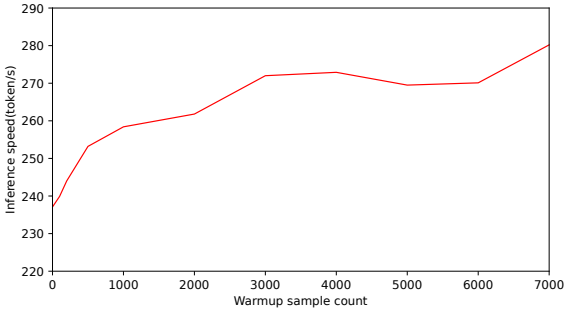

Figure 7: Inference speed with various warm-up sample count.

D PROMPT TEMPLATES.

Similar to [46], we construct a prompt for the Dolly dataset with the template in Table 13.

Table 13: Prompt templates of Dolly.

Prompt Type	Propmt Template
W/ reference	Below is an instruction that describes a task, paired with an input that provides further context. Write a response that appropriately completes the request. ### Instruction: {instruction} ### Input: {reference} ### Response:
W/o reference	Below is an instruction that describes a task. Write a response that appropriately completes the request. ### Instruction: {instruction} ### Response: