Break the Sequential Dependency of LLM Inference Using LOOKAHEAD DECODING

Yichao Fu [1] **Peter Bailis** [2] **Ion Stoica** [3] **Hao Zhang** [1]

Abstract

Autoregressive decoding of large language models (LLMs) is memory bandwidth bounded, resulting in high latency and significant wastes of the parallel processing power of modern accelerators. Existing methods for accelerating LLM decoding often require a draft model (e.g., speculative decoding), which is nontrivial to obtain and unable to generalize. In this paper, we introduce LOOKAHEAD DECODING, an exact, parallel decoding algorithm that accelerates LLM decoding without needing auxiliary models or data stores. It allows trading per-step log(FLOPs) to reduce the number of total decoding steps, is more parallelizable on single or multiple modern accelerators, and is compatible with concurrent memory-efficient attention (e.g., FlashAttention). Our implementation of LOOKAHEAD DECODING can speed up autoregressive decoding by up to 1.8x on MT-bench and 4x with strong scaling on multiple GPUs in code completion tasks. Our code is avialable at `https://github.com/hao-ai-lab/LookaheadDecoding`

1. Introduction

Large language models (LLMs) are transforming the AI industry. As they are increasingly integrated into diverse applications such as search (Team et al., 2023) and chatbots (Ouyang et al., 2022), generating long sequences at *low-latency* using LLMs is becoming one significant requirement. However, current LLMs (Touvron et al., 2023a;b; Jiang et al., 2023; OpenAI, 2023) generate text based on *autoregressive decoding*, which falls short in efficiency, primarily for two reasons. First, autoregressive decoding generates only one token at a time. Hence, the overall generation time is proportional to the number of decoding steps. Second, each decoding step largely underutilizes the parallel processing capabilities of modern accelerators (e.g., GPUs). Given the pressing need for low latency in various applications, improving autoregressive decoding remains a central challenge.

Several approaches have been proposed – one such approach is *speculative decoding* (Chen et al., 2023a; Leviathan et al., 2023) and its variants (He et al., 2023; Stern et al., 2018; Cai et al., 2024; Li et al., 2024; Liu et al., 2023; Miao et al., 2024). These methods all follow a *guess-and-verify* approach: they use a draft model to speculate several subsequent tokens and then use the original (base) LLM to verify these tokens in parallel. Since the draft model requires much fewer resources and the cost of verifying multiple tokens in parallel is similar to the cost of generating a single token, these methods can achieve considerable speedups. However, their speedups are bounded by the *token acceptance rate* (§4.1), i.e., the fraction of tokens generated by the draft model that passes the verification test of the base model. This is because every token that fails verification needs to be regenerated by the base model. In the worst case, if most proposed tokens fail verification, these methods may slow down the decoding process. Therefore, achieving a high acceptance rate is essential for these methods. Unfortunately, training a draft model to achieve a high acceptance rate is non-trivial, and the trained draft model does not generalize across base models and datasets.

To address these problems, this paper develops LOOKAHEAD DECODING. We build upon a key observation: autoregressive decoding can be equivalently formulated as solving a non-linear system via the fixed point Jacobi iteration method (§2), which has been termed as *Jacobi decoding* (Santilli et al., 2023). Each Jacobi decoding step can generate multiple tokens in parallel at different positions. Although these tokens may appear at incorrect positions, we can leverage this parallel generation approach to have the LLM generate several disjoint *n-grams* in parallel in a single step. These n-grams could potentially be integrated into future parts of the generated sequence, pending verification by the base model to maintain the output distribution.

LOOKAHEAD DECODING takes advantage of the partic-

*Work done when Yichao Fu was visiting UCSD [1]UCSD [2]Google [3]UC Berkeley. Correspondence to: Hao Zhang <haozhang@ucsd.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

ular characteristics of autoregressive decoding, which is bounded by the memory bandwidth–as each generated token depends on all tokens before it–rather than compute, by using the available cycles to generate and verify n-grams (subsequent tokens) at virtually no additional cost. In a nutshell, LOOKAHEAD DECODING consists of a *lookahead branch* that generates n-grams and a *verification branch* that verifies n-grams, both executing in a single step. To improve efficiency, we use an n-gram pool to cache the historical n-grams generated so far. This way, LOOKAHEAD DECODING can significantly reduce the latency of LLM inference just by exploiting the compute resources that autoregressive decoding would leave unused. More importantly, LOOKAHEAD DECODING scales with the compute – we show that it can linearly reduce the number of decoding steps relative to the log(FLOPs) allocated per step.

We have implemented the algorithm in Python and CUDA, which is compatible with memory-efficient attention algorithms (e.g., FlashAttention (Dao, 2023)) and supports various sampling methods without changing the output distribution. We also scale it to multiple GPUs, resulting in *Lookahead Parallelism*. We evaluate LOOKAHEAD DECODING on the popular LLaMA-2 (Touvron et al., 2023b) models. It achieves 1.8x speedup on the challenging multi-turn chat dataset MT-Bench (Zheng et al., 2024) and up to 4x speedup in code completion tasks with Lookahead Parallelism on 8 GPUs. LOOKAHEAD DECODING showed significant potential in lowering the latency for latency-sensitive tasks. Our contributions are summarized as follows.

- We design LOOKAHEAD DECODING, a new lossless, parallel decoding algorithm to accelerate LLM inference without needing any auxiliary component.
- We reveal LOOKAHEAD DECODING's scaling behavior: it linearly reduces the number of decoding steps according to per-step $\log(\text{FLOPs})$. This enables tradeoffs between the number of decoding steps and per-step FLOPs, making it future-proof.
- We show it benefits from the latest memory-efficient attentions and is easily parallelizable by developing its distributed CUDA implementations.
- We evaluated LOOKAHEAD DECODING and demonstrate its effectiveness under different settings.

2. Background

In this section, we formulate both autoregressive and Jacobi decoding from the lens of solving nonlinear systems.

Causal Attention in Decoder Models. Most contemporary LLMs are composed of two core components: token-wise modules (including MLP and normalization (Ba et al., 2016; Zhang & Sennrich, 2019)) and attention (Vaswani et al., 2017) modules. Tokens interact with each other in the attention modules, while in other token-wise modules, they are processed without exchanging information with each other.

The attention layer encompasses three input elements: query $\mathbf{Q}$, key $\mathbf{K}$, and value $\mathbf{V}$, with the i-th token in each denoted as $\mathbf{Q}_i$, $\mathbf{K}_i$, and $\mathbf{V}_i$, respectively. The attention layer executes the following operation: $\mathbf{O} = \text{softmax}\left(\mathbf{Q}\mathbf{K}^T\right)\mathbf{V}$. A lower triangular mask applied to $\mathbf{Q}\mathbf{K}^T$ in causal attentions (specific to decoder models) ensures that $\mathbf{O}_i$ is calculated only from $\mathbf{Q}_i$ and $\mathbf{K}_j$, $\mathbf{V}_j$ where $j \le i$. Because all other layers in the LLM perform token-wise operations, for any given model input $\mathbf{x}$ and output $\mathbf{o}$, $\mathbf{o}_i$ (i-th token in $\mathbf{o}$) is exclusively influenced by $\mathbf{x}_j$ (j-th token in $\mathbf{x}$) where $j \le i$.

Autoregressive Decoding in LLMs. LLM takes tokens as input, which are represented in discrete integers and mapped to continuous tensors by an embedding layer. We notate $\mathbf{x} = (x_1, x_2, ..., x_s) \in \mathbb{N}^s$ of length s as the input of the model, and $\mathbf{x}^t_{1:m} = (x_1, x_2, ..., x_m)$ to denote a slice of $\mathbf{x}$ of length m at step t, following the representations from previous work (Santilli et al., 2023)). LLMs' output characterizes the probability distribution of the next token. The probability for the s-th token (i.e., the output of the $s-1$-th token) is decided by all previous input tokens, represented as $P_M(x_s|\mathbf{x}_{1:s-1})$. Then, the next token input x_s is obtained by sampling from $P_M(x_s|\mathbf{x}_{1:s-1})$ using different methods (e.g., greedy, top-K, and top-P (Kool et al., 2020; Holtzman et al., 2019)). When using greedy sampling, the next token is selected by applying an argmax function on P_M.

We define $\mathbf{x}^0$ as the prompt tokens given by the user. The LLM needs to generate an output sequence (of length m) from $\mathbf{x}^0$. Denote y_i as the token generated at step i. The autoregressive decoding process of m tokens with greedy sampling can be seen as solving the following m problems *one by one*. The following equations come from previous work (Santilli et al., 2023):

$$\begin{cases} y_1 = \text{argmax}\, P_M(y_1|\mathbf{x}^0) \\ y_2 = \text{argmax}\, P_M(y_2|y_1, \mathbf{x}^0) \\ ... \\ y_m = \text{argmax}\, P_M(y_m|\mathbf{y}_{1:m-1}, \mathbf{x}^0) \end{cases} \tag{1}$$

Guess-And-Verify Paradigm. The *Guess-And-Verify* decoding paradigm speculates multiple potential future tokens and subsequently confirms the correctness of these speculations within a single decoding step. Take speculative decoding with greedy sampling as an example: at step t, with the prompt $\mathbf{x}^0$ and tokens $\mathbf{y}_{1:t-1}$ generated so far, we can use a draft model to autoregressively generate a draft sequence $\mathbf{y}_{t:t+n-1}$ of length n. Because $\mathbf{y}_{t:t+n-1}$ is known a priori, we then use the LLM to solve Eqs 2 *in parallel*, obtaining $\mathbf{y}'_{t:t+n}$. Then, we verify if y_{t+i} is equal to y'_{t+i} for each i from $i = 0$ to $i = n-1$. If there is a match, we accept this token and proceed; otherwise, we stop checking and drop subsequent tokens. Finally, we update $\mathbf{y}$ with all accepted tokens.

$$\begin{cases} y'_t = \operatorname{argmax} P_M(y_t|\mathbf{y}_{1:t-1}, \mathbf{x}^0) \\ y'_{t+1} = \operatorname{argmax} P_M(y_{t+1}|\mathbf{y}_{1:t}, \mathbf{x}^0) \\ \ldots \\ y'_{t+n} = \operatorname{argmax} P_M(y_{t+n}|\mathbf{y}_{1:t+n-1}, \mathbf{x}^0) \end{cases} \tag{2}$$

As stated in §1, these approaches depend on a good draft model, which is hard to obtain and cannot generalize.

Jacobi Decoding. By notating $f(y_i, \mathbf{y}_{1:i-1}, \mathbf{x}^0) = y_i - \operatorname{argmax} P_M(y_i|\mathbf{y}_{1:i-1}, \mathbf{x}^0)$, we can transform Eqs 1 into the following non-linear system of equations (adapted from previous work (Song et al., 2021; Santilli et al., 2023)):

$$\begin{cases} f(y_1, \mathbf{x}^0) = 0 \\ f(y_2, y_1, \mathbf{x}^0) = 0 \\ \ldots \\ f(y_m, \mathbf{y}_{1:m-1}, \mathbf{x}^0) = 0 \end{cases} \tag{3}$$

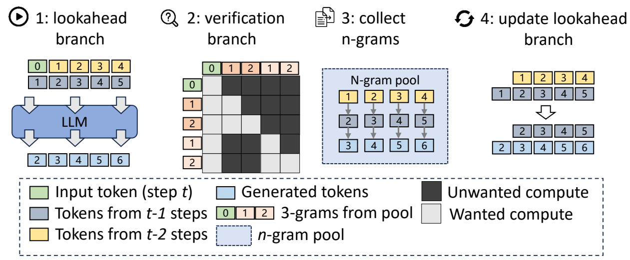
We can solve this non-linear system using Jacobi iteration by iteratively updating all y_i from a random initial guess $\mathbf{y}^0$, along the *trajectory* $\mathbf{y}^1, ..., \mathbf{y}^t, ...$, until converging to the fixed point solution $\mathbf{y}^m$. We detail this algorithm, termed as *Jacobi decoding* by previous work (Santilli et al., 2023), in Appendix Algorithm 2. This process guarantees to return the solution of all m variables y_i in *at most* m iterations, as the very first token of each Jacobi update matches autoregressive decoding. Sometimes, more than one token might be correctly generated in a single iteration, potentially reducing the number of decoding steps. It is worth noting that, as $\mathbf{y}^t$ is generated based on the past value $\mathbf{y}^{t-1}$ on the trajectory, any two adjacent tokens from $\mathbf{y}^{t-1}$ and $\mathbf{y}^t$ can form a meaningful 2-gram.

Limitations of Jacobi Decoding. Empirically, it has been observed from previous research (Santilli et al., 2023) and in our evaluations (Appendix F) that Jacobi decoding can hardly reduce decoding steps, even if it can generate multiple tokens per step. This is because the generated tokens are often put in the wrong positions of the sequence, and correctly placed tokens are frequently replaced by subsequent Jacobi iterations. These prevent it from achieving wall-clock speedup.

3. Lookahead Decoding

Lookahead Decoding leverages Jacobi decoding's ability to generate many tokens in one step but addresses its limitation. Fig. 1 illustrates its workflow, and Algorithm 1 shows its detail. The key design in Lookahead Decoding is to keep track of the trajectory of Jacobi decoding and generate n-grams from this trajectory. This is achieved by maintaining a fixed-sized 2D window, with the two dimensions corresponding to the sequence and the time axis,

Figure 1: Workflow of Lookahead Decoding with $W = 5$, $N = 3$, and $G = 2$. W, N, and G are parameters controlling Lookahead Decoding's performance, defined in §3.1 and §3.2. For each decoding step, we do the following. (1) Generate one token at each position in the *lookahead branch*; (2) Verify and accept 3-grams (searched from the 3-gram pool) with the *verification branch*; (3) Collect and cache newly generated 3-grams in the pool from lookahead branch trajectories. (4) Update the *lookahead branch* to maintain a fixed window size.



respectively, to generate multiple disjoint n-grams from the Jacobi iteration trajectory in parallel. We call this process the *lookahead branch*. The fixed-sized 2D window corresponds to the variable $\mathbf{w}$ in Algorithm 1, with a superscript indicating time axis and a subscript indicating sequence axis. In addition, Lookahead Decoding introduces an n-gram pool (i.e., $\mathbf{C}$ in Algorithm 1) to cache these n-grams generated along the trajectory. Promising n-gram candidates are verified later by a designed *verification branch* to preserve the LLM's output distribution; if passing verification, those disjoint n-grams are integrated into the sequence. We introduce the lookahead branch and verification branch in detail in the following sections.

3.1. Lookahead Branch

Lookahead Decoding uses a fixed-sized 2D window for efficient n-gram generation. In contrast to the original Jacobi decoding, which only uses the history tokens from the last step (or equivalently, it generates 2-grams), Lookahead Decoding generates many n-grams, with $n \geq 2$, in parallel by using the $n-1$ past steps' history tokens, effectively leveraging more information from the trajectory. The fixed-sized 2D window in the lookahead branch is characterized by two parameters: (1) W defines the *lookahead* size into future token positions to conduct parallel decoding; (2) N defines the *lookback* steps into the past Jacobi trajectory to retrieve n-grams. The 2D window is shaped $(N-1) \times W$ as the $\mathbf{w}$ in Algorithm 1.

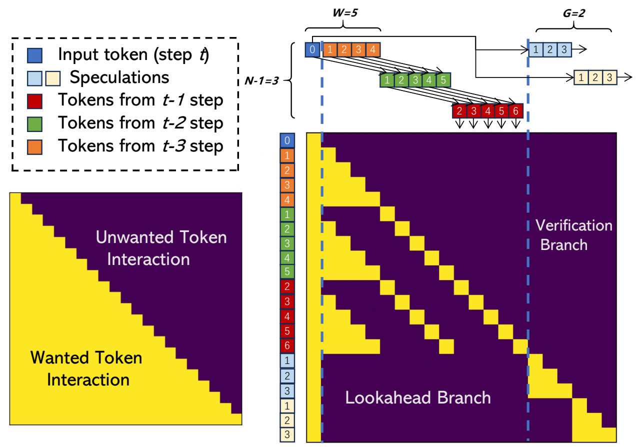
An example of the lookahead branch with $W = 5$ and $N = 4$ is in Fig. 2 (b), in which we look back $N - 1 = 3$ steps and look ahead 5 tokens for each step. The blue token with the digit 0 is the current step's (t) input, and the orange, green, and red tokens were generated in previous lookahead branches at steps $t-3$, $t-2$, and $t-1$, respec-

Algorithm 1 Lookahead decoding

1: **Input:** prompt $\mathbf{x}^0 = (x_1, x_2, ...x_n)$, model P_M, n-gram size N, window size W, max #speculations G, max steps m.
2: Initialize n-gram pool $\mathbf{C} \leftarrow \emptyset$
3: Initialize $\mathbf{o} \leftarrow \emptyset$
4: Randomly initialize 2D window $\mathbf{w}_{1:W}^{2-N:0}$
5: Set $\mathbf{o}_0 \leftarrow x_n$ {Initial Size of $\mathbf{o}$ is 1}
6: **for** $i = 1$ **to** m **do**
7:   **if** $size(\mathbf{o}) >= i + 1$ **then**
8:     Randomly set $\mathbf{w}_{1:W}^{i}$
9:     continue
10:   **end if**
11:   {Lookahead Branch}
12:   **for** $j = 1$ **to** W **do**
13:     $w_j^i \leftarrow \text{argmax}\, P_M(w_j^i | \mathbf{w}_j^{i+2-N:i-1}, \mathbf{w}_{2:j}^{i+1-N},$
14:     $\mathbf{o}_{1:i}, \mathbf{x}^0)$ {$\mathbf{w}_{2:j}^{i+1-N}$ is empty if $j = 1$}
15:   **end for**
16:   $\mathbf{w}_{1:W}^{i} = (w_1^i, w_2^i, ..., w_W^i)$
17:   {Verification Branch}
18:   $\mathbf{g} \leftarrow \emptyset$
19:   **for** $j = 1$ **to** G **do**
20:     $\mathbf{g}^i \leftarrow$ n-gram from $\mathbf{C}$ starting with $\mathbf{o}_{i-1}$
21:   **end for**
22:   {Verification Algorithm in Algo. 3 and Algo. 4.}
23:   $\mathbf{o}$.append(VERIFICATION$((x^0, \mathbf{o}_{1:i}), P_M, \mathbf{g})$)
24:   {Update n-gram pool}
25:   **for** $j = 1$ **to** W **do**
26:     add n-gram $\mathbf{w}_j^{i+1-N:i}$ to $\mathbf{C}$
27:   **end for**
28:   {Update Lookahead Branch}
29:   Remove $\mathbf{w}_{1:W}^{:size(\mathbf{o})-N}$
30: **end for**
31: **Output:** $\mathbf{o}_{1:m} = (y_1, y_2, ..., y_m)$

tively. The digit on each token shows its relative position to the current input (i.e., the blue one labeled as 0). In the present stage, we perform a modified Jacobi iteration to generate new tokens for all $W = 5$ positions, following the trajectory formed by the preceding 3 steps. Note that these newly generated tokens have not yet been merged into the generation sequence. For example, based on Fig. 2 (b), we will generate the next token from the path "blue 0 $\rightarrow$ orange 1 $\rightarrow$ green 2 $\rightarrow$ red 3" and so on. The generation process corresponds to L12-15 in Algorithm 1. Once generated, we collect and cache them in the n-gram pool ($n = 4$) – for instance, a 4-gram consists of the orange token at position 1, the green token at position 2, the red token at position 3, and a newly generated token (L25-27 in Algorithm 1).

The most outdated tokens in both dimensions (time and sequence) will be removed (L29 in Algorithm 1), and newly generated tokens will be appended to the lookahead branch to maintain a fixed window size for each step (L16 in Algorithm 1). For example, we will remove all orange and green tokens with position 1 in Fig. 2. We then form a new lookahead branch with green tokens with indices 2, 3, 4, 5, all red tokens, and all newly generated tokens for the next step.



(a) Causal Mask (b) Lookahead Decoding Mask

Figure 2: (a) Causal mask for decoder models. (b) Attention mask for LOOKAHEAD DECODING with $W = 5$, $N = 4$, and $G = 2$. Digits on tokens indicate relative positions.

3.2. Verification Branch

The verification branch verifies n-grams to preserve the output distribution (L22 in Algorithm 1). We first discuss how to verify in greedy sampling (Algorithm 3). Recall in speculative decoding: the verification is performed by sending the draft tokens to the LLM to get an output for each draft token, then progressively checking if the last token's corresponding output, generated by the target LLM, exactly matches the draft token itself (§2). The verification branch in LOOKAHEAD DECODING resembles this process despite verifying *many draft n-gram* candidates ($\mathbf{g}$ in Algorithm 3) in parallel. In particular, we first look up from the n-gram pool ($\mathbf{C}$ in Algorithm 1) for "promising" n-grams – by checking if a n-gram starts with a token that exactly matches the last token of the current ongoing sequence. We then use the LLM to verify all these multiple n-grams in parallel, following a similar fashion as in speculative decoding but accepting a prefix of n-gram with the largest number of accepted tokens amongst all n-grams.

We next discuss how to support more advanced sampling. Previous research (Miao et al., 2024) has developed efficient tree-based verification for speculative decoding with sampling support, where multiple draft sequences derived from a token tree can be verified in parallel. However, it does not apply to LOOKAHEAD DECODING as our verification works on disjoint n-grams instead of trees. We improve it by progressively verifying along the n-gram length and removing n-grams with mismatched prefixes. Besides, speculative decoding style verification requires the probability distribution where the *draft token* is sampled to update the probability distribution when the draft token is rejected. Because we store all n-grams in a pool instead of discarding them each step, we would need huge memory to store the probability distributions (each of vocabulary size) for the entire n-gram pool. The key to overcoming this is to leverage the mechanism that the verification is indifferent to how draft tokens were sampled – different sampling methods (e.g., greedy sampling) only influence the acceptance rate but keep the output distribution. We can force greedy sampling at the n-gram generation (lookahead branch), in which the probability distribution degenerates into a one-hot vector. Hence, we only need to store which token has been selected. We elaborate on the approach in Algorithm 4, prove its correctness in Appendix B, and verify its quality and speedups in §5.3 and Appendix E.

It is expected to have an increasingly large n-gram cache, hence a growing verification branch as decoding progresses. We set a cap of G to limit the maximum number of promising candidates running in parallel in the verification branch to manage the verification cost. Empirically, we suggest setting G proportional to W to balance generation and verification. In practice, we simply set $G = W$.

3.3. Decode, Predict, and Verify in The Same Step

At execution, the lookahead and verification branches can be integrated into one decoding step to leverage parallel processing. This requires a designated attention mask, as shown in Fig. 2 (b). This attention mask is straightforwardly derived following the principle that each token is only visible to the tokens along the generation trajectory with a larger position index than itself (§2). For example, only the green token at position 5 and all orange tokens are visible to the red token 6. The tokens in the lookahead branch are not visible to the tokens in the verification branch, and vice versa.

Integration with FlashAttention. FlashAttention (Dao et al., 2022; Dao, 2023) can vastly accelerate the training and inference of LLMs by saving memory I/O on the slow memory hierarchy. It forces a causal mask (e.g., Fig. 2 (a)) to avoid all token interactions outside a lower triangular scope, which is not suitable for LOOKAHEAD DECODING as we take a more subtle attention mask (e.g., Fig. 2 (b)) for different W, N, and G. To solve this, we hardcode LOOKAHEAD DECODING's attention pattern with adjustable W, N, and G in FlashAttention. Applying FlashAttention to LOOKAHEAD DECODING brings about 20% end-to-end speedup compared to a straightforward implementation on top of native PyTorch in our experiments (§5.2).

3.4. Lookahead Parallelism

LOOKAHEAD DECODING is easy to parallelize on multiple GPUs for both lookahead and verification branches. Parallelizing the lookahead branch is achieved by noting that the lookahead computation is composed of several disjoint branches. For example, the branch with green 1 and red 2 tokens does not have interaction with the branch with the tokens green 3 and red 4 in Fig. 2 (b). We can put these disjoint branches onto different GPUs without introducing communication during the inference computation. Parallelizing the verification branch is done by assigning multiple n-gram candidates to different devices. Because the verification of each candidate, by design, is independent of others, this will not cause communication.

Fig. 3 shows an example of parallelizing the lookahead branch and verification branch in Fig. 2 (b) to four GPUs. This workload allocation will have the orange token 0,1,2,3 and the input token 0 be redundantly placed and computed. However, it can essentially save communication volume during the whole forward pass. We only need to synchronize the generated tokens on each device after the forward pass. We can further scale the W, N, and G with multiple GPUs' increased FLOPs to obtain a lower latency according to LOOKAHEAD DECODING's scalability (§4).

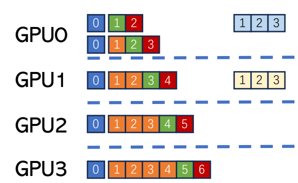


Figure 3: Distribute the workload of the lookahead branch and verification branch in Fig 2 (b) to 4 GPUs with lookahead parallelism, which can avoid communication during the forward pass.

We name this new parallelism as *lookahead parallelism* (LP). Unlike Pipeline Parallelism (PP) and Tensor Parallelisms (TP) that shard the model parameters or states across different GPUs (Narayanan et al., 2021; Shoeybi et al., 2019), LP maintains an entire copy of the model for each GPU (thus needing more memory) and allows distributing tokens to different GPUs. Hence, LP is advantageous in faster inference when memory is not a bottleneck as it introduces near-zero communication per step, while existing model parallelism methods involve a large communication overhead on the critical path of each decoding step. To sum up, PP and TP use multiple GPUs' memory to hold a larger model, and our proposed LP uses multiple GPUs' FLOPs to accelerate LLM decoding.

4. Scaling Law of LOOKAHEAD DECODING

Since LOOKAHEAD DECODING introduces flexible parameters W and N associated with the cost of each parallel decoding step. This section investigates the scaling law between compute FLOPs and the theoretical speedup, and compares it to speculative decoding.

4.1. Estimating Speedup for Speculative Decoding

Speculative decoding uses the draft model to speculate one token sequence at each step. We represent the probability of each token in the sequence passing the verification of the LLM by β (acceptance rate) and notate its expectation $E(\beta) = \alpha$. If we use the draft model to guess γ tokens per step, the expectation of the number of accepted tokens is denoted as (Leviathan et al., 2023):

$$E(\#tokens) = \frac{1 - \alpha^{\gamma+1}}{1 - \alpha}. \tag{4}$$

Instead of speculating one sequence every time, we would speculate b sequences. We assume that b sequences, each of γ tokens, are sampled as each token will have the same acceptance rate of β. Under this setting, the expectation of the number of accepted tokens is denoted as follows:

$$E(\#tokens) = (\gamma + 1) - \sum_{i=1}^{\gamma} (1 - \alpha^i)^b. \tag{5}$$

See derivations in Appendix C for Eq. 4 and Eq. 5. Note that when $b = 1$, Eq. 5 falls back to Eq. 4.

4.2. Estimating Speedup for LOOKAHEAD DECODING

We define the $\mathcal{S}$ = *step compression ratio* as the number of autoregressive steps divided by the number of LOOKAHEAD DECODING steps to generate the same length of the sequence. As the number of generated tokens equals the autoregressive steps, it can be denoted as:

$$\mathcal{S} = \frac{\#generated\ tokens}{\#\text{LOOKAHEAD DECODING}\ steps} \tag{6}$$

LOOKAHEAD DECODING speculates b sequences every time as in Eq. 5. In each step, we will search n-grams in the pool starting with the current input token and have at most G speculations of length $N - 1$. As we set $G = W$ (§3.2), we have $G = W = b$ and $N - 1 = \gamma$ using the notations in Eq. 5. Lookahead decoding differs from vanilla speculative decoding in verification. In speculative decoding, there is only one speculation (given by the draft model) verified by the LLM. But in lookahead decoding, all n-grams are *searched* from an n-gram pool and verified within a total number of G speculations. This search and selection should follow another distribution. We assume that, on average, for every f step, we have one good search/selection with $E(\#tokens)$ tokens accepted, and for the other $f - 1$ steps, we fall back to autoregressive decoding due to bad speculations. We use this f to bridge $\mathcal{S}$ and $E(\#tokens)$ per step as follows:

$$\mathcal{S} = (f - 1 + E(\#tokens))/f. \tag{7}$$

We can plot the curve indicated by our formulation with one specific setting as in Fig. 4 (b). We find that the trend of our empirical experiments (LLaMA-2-Chat-7B on MT-Bench with $G = W$ as in Fig. 4 (a)) aligns well with the formulation to some extent. From this formulation, we conclude that we can linearly reduce the number of decoding steps according to per-step $\log(b)$ given a large enough γ. In contrast to speculative decoding, LOOKAHEAD DECODING will not meet an upper bound indicated in Eq. 4 by simultaneously increasing γ and b. This reveals the *scaling law* of LOOKAHEAD DECODING to linearly reduce decoding steps according to per-step $\log(\text{FLOPs})$ given a large enough N, since per-step FLOPs is roughly proportional to the number of input tokens (i.e., $(W + G) * (N - 1)$). The scaling law also suggests LOOKAHEAD DECODING's strong scaling to multiple GPUs, in which we can obtain an even greater per-token latency reduction by using more FLOPs, which is advantageous for latency-sensitive tasks.

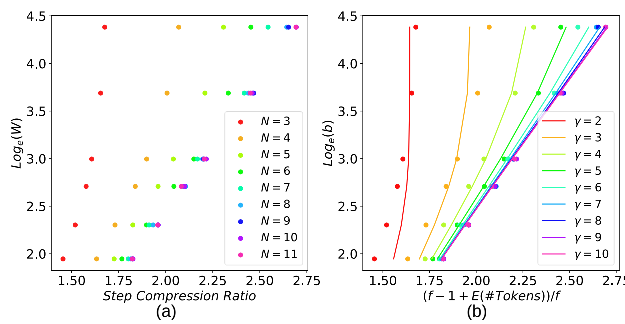


Figure 4: (a) Relation of W, N, G and $\mathcal{S}$ for LLaMA-2-Chat-7B on MT-Bench. (b) When we assume a setting with $\alpha = 0.425$ and $f = 3.106$, the trend of our formulation.

5. Evaluation Results

Model and testbed. We used various versions of the LLaMA-2 (Touvron et al., 2023b) and CodeLlama (Roziere et al., 2023) models, including the 7B, 13B, 34B, and 70B sizes, on two GPU setups *S1* and *S2*. *S1* is equipped with NVIDIA A100 GPUs with 80GB of memory. On *S1*, the 7B, 13B, and 34B models are deployed on a single A100, while the 70B model utilizes 2 A100s with pipeline parallelism supported by Accelerate (Gugger et al., 2022). *S2* is a DGX machine with 8 NVIDIA A100 GPUs with 40GB memory and NVLink. All models serve with FP16 precision and batch of 1 if not specified (Cai et al., 2024; He et al., 2023).

Datasets. We benchmarked LOOKAHEAD DECODING's performance across a broad spectrum of datasets and tasks. MT-Bench (Zheng et al., 2024) is a diverse set of multi-turn questions with many *unique tokens*. GSM8K (Cobbe et al., 2021) contains a set of math questions, in which we use the first 1k questions. HumanEval (Chen et al., 2021) covers both code completion and infilling tasks. We also test on MBPP (Austin et al., 2021) dataset for instruction-based

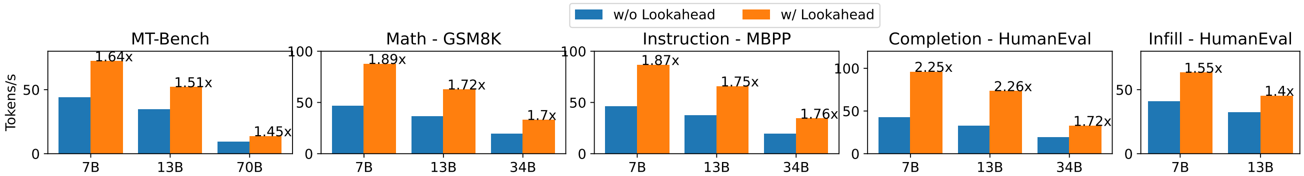


Figure 5: Throughput of LOOKAHEAD DECODING on various dataset without FlashAttention and distributed serving.

code generation, and on ClassEval (Du et al., 2023) for class-level code completion. To control generation length in code generation tasks, we set the maximum sequence length to 512 and 2,048 on HumanEval and ClassEval, respectively, aligned with prior setups (Ben Allal et al., 2022; Du et al., 2023). Tab. 1 lists detailed settings. In addition, we validate the effectiveness of sampling (§3.2) on XSum (Narayan et al., 2018) and CNN/Daily Mail (See et al., 2017) datasets.

Baseline Settings. Our primary baseline is HuggingFace's implementation of greedy search (Wolf et al., 2020). Additionally, we employ FlashAttention (Dao et al., 2022; Dao, 2023) as a stronger baseline to assess the performance of memory-efficient attention empowered LOOKAHEAD DECODING. In distributed settings, we evaluate LP against TP (supported by deepspeed (Aminabadi et al., 2022)) and PP (supported by accelerate (Gugger et al., 2022)). We measure the throughput of single batch inference against these baseline settings (Cai et al., 2024; He et al., 2023).

Table 1: Experimental settings for §5.1 and §5.2.

SERVER	PARALLEL.	MODEL	MODEL SIZE	DATASET
		LLAMA-2-CHAT	7B, 13B, 70B	MT-BENCH
S1	W/O LP	CODELLAMA	7B, 13B, 34B	HUMANEVAL
		CODELLAMA-INST	7B, 13B, 34B	MBPP, GSM8K
		LLAMA-2-CHAT	7B, 13B	MT-BENCH
S2	W/ LP	CODELLAMA	7B, 13B	HUMANEVAL
		CODELLAMA-PYTHON	7B, 13B	CLASSEVAL

5.1. End-to-end Performance

Fig. 5 shows the end-to-end performance of LOOKAHEAD DECODING when compared with HuggingFace's implementation of greedy search on *S1*. The used tasks and models are shown in Tab. 1. Across various datasets, LOOKAHEAD DECODING demonstrates a 1.4x-2.3x speedup. Generally, our method exhibits better performance in code completion tasks (e.g., 2.3x), given the higher occurrence of repetitive tokens during code completions, making predictions easier. Besides, smaller models also exhibit a higher speedup when compared to larger models. This is because LOOKAHEAD DECODING trades per-step FLOPs with a step compression ratio (§4). A larger model requires more FLOPs and quickly hits the GPU FLOPs cap compared to a smaller model. So, it shows a lower ability to compress decoding steps given the same GPU setting.

5.2. Performance with LP and FlashAttention

We evaluated the performance of LOOKAHEAD DECODING with LP and FlashAttention augmentation on *S2* with greedy search. The used tasks and models are shown in Tab. 1. The results for the 7B and 13B models are in Fig. 6 and Fig. 7, respectively. FlashAttention speeds up the PyTorch implementation of LOOKAHEAD DECODING by 20%. Notably, FlashAttention-integrated LOOKAHEAD DECODING shows 1.8x speedups for the 7B model on MT-Bench compared with autoregressive decoding with FlashAttention (i.e., 1.9x vs 1.07x in Fig. 6). We did a strong scaling of the workloads to multiple GPUs for distributed settings (i.e., increasing GPUs but not increasing workloads). The multiple GPU settings of both TP (w/ DeepSpeed) and PP (w/ Accelerate) bring slowdowns (i.e., less than 0.9x). The results echos DeepSpeed's documentation (dee, 2023). However, with LOOKAHEAD DECODING, we can further utilize the FLOPs of multiple GPUs to reduce the inference latency (e.g., 4x on ClassEval).

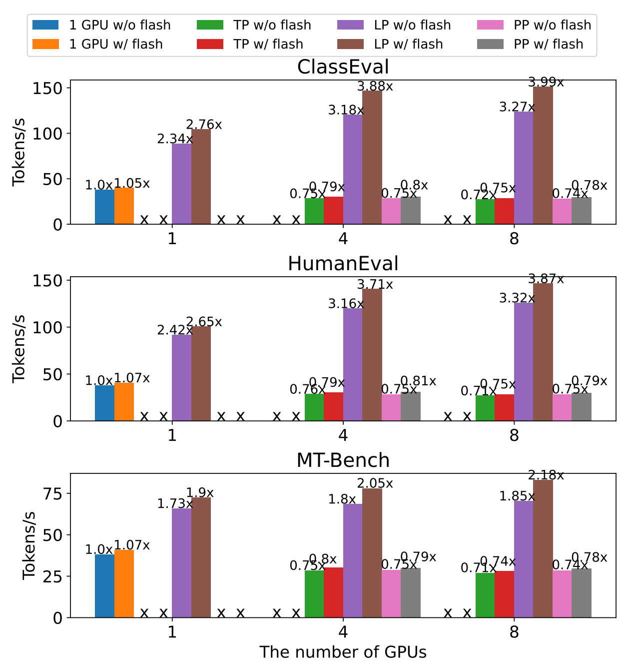


Figure 6: Throughput of LOOKAHEAD DECODING with multiple GPUs and FlashAttention for 7B models.

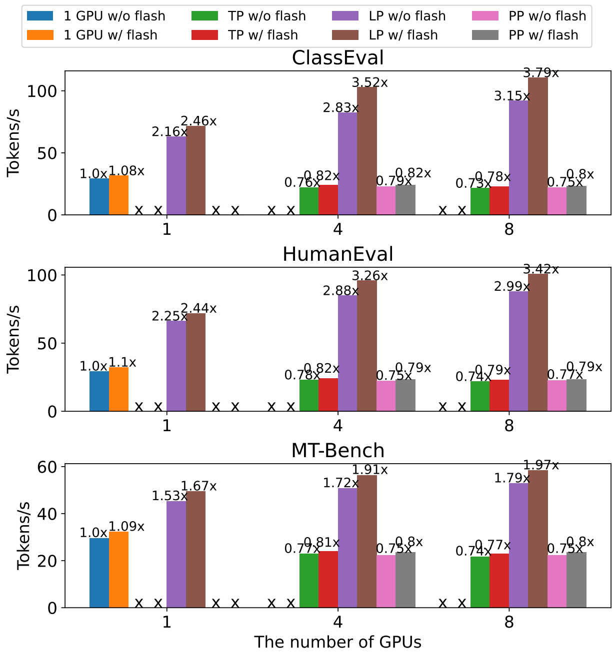


Figure 7: Throughput of LOOKAHEAD DECODING with multiple GPUs and FlashAttention for 13B models.

5.3. Generation Quality of LOOKAHEAD DECODING

We assess the generation quality of LOOKAHEAD DECODING on LLaMA-2-7B-Chat model with the prompts in Appendix D on summarization datasets (Chen et al., 2023a; Leviathan et al., 2023) in Tab. 2. Whether the sampling is activated, LOOKAHEAD DECODING can reserve the output distribution quality, which is evaluated in rouge-1, rouge-2, and rouge-L (Lin, 2004), while achieving 1.46x-1.60x speedups compared with autoregressive decoding. Using sampling gives smaller speedups as the acceptance ratio is lower according to the sampling verification algorithm 4, which aligns with the results in the previous research (Chen et al., 2023a; Leviathan et al., 2023). We further verify that using greedy sampling and advanced integrations will not change the generation quality in Appendix E.

Table 2: Sampling with LOOKAHEAD DECODING on CNN/Daily Mail and XSum. A temperature (Temp.) of 0.0 equals greedy search. "AR." is autoregressive and "LA." is LOOKAHEAD DECODING. Rouge scores, speedups against autoregressive, and compression ratio ($\mathcal{S}$) are reported.

DATASET	TEMP.	METHOD	ROUGE-1	ROUGE-2	ROUGE-L	SPEEDUPS	$\mathcal{S}$
CNN.	1.0	AR.	36.55	13.20	22.68	1.00x	1.00x
		LA.	36.53	13.27	22.71	1.46x	1.64x
	0.0	AR.	37.79	14.59	23.96	1.00x	1.00x
		LA.	37.79	14.59	23.96	1.57x	1.72x
XSUM	1.0	AR.	19.15	4.53	12.84	1.00x	1.00x
		LA.	19.20	4.53	12.87	1.50x	1.67x
	0.0	AR.	19.38	4.78	13.05	1.00x	1.00x
		LA.	19.39	4.79	13.06	1.60x	1.77x

5.4. Ablation Study

In this section, we study the importance of the lookahead and verification branch in achieving a high speedup. We experiment on LLaMA-2-7B-Chat and MT-Bench on *S1* with various settings. The results are shown in Tab. 3.

We ablate the *importance of lookahead branch* by comparing the performance of using a lookahead branch to the recent methods of using *prompts as reference* (Yang et al., 2023a; Saxena, 2023). This comparison assumes that LOOKAHEAD DECODING does not use the prompt to build the n-gram pool. We use the implementation in transformers v4.37 of prompt lookup as a baseline (②, with prompt_lookup_num_tokens=10). We also use prompt to build n-gram pool to augment LOOKAHEAD DECODING (③④⑥⑨). The results show that although using a minimal lookahead branch ($W = 1$) with various N, G settings (③④⑤⑥) can obtain a decent speedup on MT-Bench, it is still not as good as using balanced branches (⑧). We can find that prompt lookup can surpass *prompt as reference* implementation in LOOKAHEAD DECODING. This is because our method checks if n-gram starts with *one* token that exactly matches the last generated token while prompt lookup in transformers v4.37 checks *several* starting tokens for a better speculation.

We ablate the *importance of verification branch* by reporting the speedup of using a tiny verification branch and a large lookahead branch (⑦, $G = 1$) . It shows lower performance due to lower potential in accepting speculations compared with a balanced branches (⑧).

Besides, our evaluation shows that using *prompt as reference* can further boost LOOKAHEAD DECODING (⑧ and ⑨). We have integrated them in our implementation.

Table 3: Compare the effectiveness of both lookahead and verification branch on MT-Bench on A100. FlashAttention is activated. We show the speedups against autoregressive decoding and the compression ratio ($\mathcal{S}$).

TAG	SETTING (N, W, G)	*Prompt as Ref.*	SPEEDUPS	$\mathcal{S}$
①	AUTOREGRESSIVE	✗	1.00x	1.00
②	PROMPT LOOKUP	✓	1.44x	1.55
③	(10, 1, 3)	✓	1.36x	1.45
④	(5, 1, 10)	✓	1.36x	1.51
⑤	(5, 1, 30)	✗	1.04x	1.12
⑥	(5, 1, 30)	✓	1.46x	1.59
⑦	(5, 30, 1)	✗	1.61x	1.79
⑧	(5, 15, 15)	✗	1.78x	1.96
⑨	(5, 15, 15)	✓	**1.88x**	**2.05**

5.5. Discussion and Limitation

The main limitation of LOOKAHEAD DECODING is that it requires extra computations. Our experimental results show that on A100, the configuration in Tab. 4 works near optimally in most cases for single batch serving. Because the per-step FLOPs are roughly proportional to the number

Table 4: Good Config. of LOOKAHEAD DECODING on A100 GPUs with $G = W$.

MODEL	WINDOW SIZE (W)	N-GRAM SIZE (N)
7B	15	5
13B	10	5
34B	7	5

of per-step input tokens, which is $(W + G) * (N - 1)$. If we ignore the attention cost's increase with sequence length, the 7B, 13B, and 34B models require 120x, 80x, and 56x extra FLOPs per step, respectively. Since the LLM decoding is memory bandwidth-bound rather than compute-bound, these extra FLOPs only turn into a limited wall-clock slowdown for each step.

Given this, LOOKAHEAD DECODING needs large surplus FLOPs to obtain high speedups. Running in compute-bound environments (e.g., serving with a large batch size) may cause slowdowns. Another example is shown in Fig. 8, where lower speedup is observed when the GPU's cap FLOPs is smaller (e.g., on RTX 3090 GPUs).

Based on §4, we need to exponentially increase the per-step FLOPs to obtain a linear reduction in decoding steps. Hence, the setting in Tab. 4 faces a diminishing return. However, when FLOPs are not rich, we see that a gentle speedup (e.g., 30% on RTX 3090 and $> 50\%$ on A100) on MT-Bench easily achievable, as in Fig. 8, which is a free lunch that requires no extra model, training, or changing the output distribution.

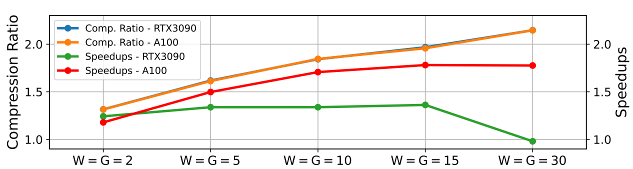


Figure 8: Compression ratio($\mathcal{S}$) and speedups of LOOKAHEAD DECODING on RTX 3090 and A100 with $N = 5$, all with FlashAttention. The blue and orange curves of $\mathcal{S}$ overlap as the device does not affect the ratio.

6. Related Work

Lookahead Methods. A few previous works have proposed "lookahead" methods in various areas. In speech recognition, researchers have proposed two efficient look-ahead pruning algorithms to speed up speech recognition (Ortmanns & Ney, 2000). In natural language generation, researchers have used lookahead method to enhance generation quality by anticipating future outputs (Lu et al., 2022; Wan et al., 2023; Ribeiro et al., 2023; Cui & Sachan, 2023; Qi et al., 2020). Lookahead has been proposed in optimizing deep learning (Zhang et al., 2019), sequential Monte Carlo (Lin et al., 2013), and Bayesian optimization (Wu & Frazier, 2019). Our proposed lookahead decoding differs from these methods in that our method is designed for accelerating LLM decoding, and we do not change the output distribution.

Speculative Decoding. Speculative decoding (Stern et al., 2018; Leviathan et al., 2023; Chen et al., 2023a) is an effective method to tackle the slow speed of LLMs' autoregressive decoding, as summarized in recent surveys (Xia et al., 2024; Miao et al., 2023). Unlike non-autoregressive generation methods (Xiao et al., 2023; Guo et al., 2020; Su et al., 2021; Wang et al., 2022; Li et al., 2022), speculative decoding is mainly for augmenting pre-trained LLMs for faster generation speed without changing their output distribution. Recent researchers have proposed different methods to draft speculations efficiently for further verification. Applying finetuning (Xia et al., 2023; Kim et al., 2024; Miao et al., 2024; Zhou et al., 2023; Liu et al., 2023), ready-made generators (Leviathan et al., 2023; Spector & Re, 2023; Chen et al., 2023b; Sun et al., 2024; He et al., 2023), or trained feedforward layers (Stern et al., 2018; Cai et al., 2024; Li et al., 2024) all show significant potential in generating high-quality drafts. Other methods (Yang et al., 2023b; Zhang et al., 2023; Hooper et al., 2023) depend on layer-skipping or early-exiting to speed up. Parallel Jacobi decoding (Santilli et al., 2023) and lookahead decoding use the original LLM to generate and verify the draft by adapting attention masks. However, lookahead decoding will cache the drafts and verify them flexibly with multiple branches, showing more considerable speedups in LLM generations. SpecInfer (Miao et al., 2024), Medusa (Cai et al., 2024), and Eagle (Li et al., 2024) use tree verification to do multiple branch verification. However, lookahead decoding verifies disjoint n-grams in parallel, more adapting to the n-gram generation by its lookahead branch.

7. Conclusion

In this paper, we present LOOKAHEAD DECODING to parallelize the autoregressive decoding of LLMs without changing the output distribution. It shows notable speedup without a draft model and can linearly decrease the decoding steps with exponential investment in per-step FLOPs.

Acknowledgements

We would like to thank Richard Liaw, Yang Song, and Lianmin Zheng for providing insightful feedback. We greatly thank the anonymous reviewers for their insightful and constructive suggestions.

Impact Statement

The rapid advancement of Large Language Models (LLMs) underscores the necessity for low latency. The inefficiency of autoregressive decoding in LLMs is in its underutilization of GPU FLOPs and the large number of decoding steps. LOOKAHEAD DECODING addresses this issue by linearly reducing decoding steps with per-step $\log(\text{FLOPs})$, significantly decreasing the decoding steps and effectively harnessing the parallel processing capabilities of GPUs. Importantly, LOOKAHEAD DECODING does not require a finely-tuned draft model; it preserves the output distribution and exhibits broad generalizability across diverse models and datasets.

The efficiency of LLMs, as enhanced by LOOKAHEAD DECODING, is anticipated to have profound ethical impacts and societal implications. By speeding up LLM decoding, LOOKAHEAD DECODING could potentially democratize access to these powerful tools, thereby fostering innovation across various domains. Thus, the development of LOOKAHEAD DECODING highlights the need for robust ethical guidelines and regulatory oversight in the deployment of efficient LLMs.

References

Automatic tensor parallelism for huggingface models, 2023. URL https://www.deepspeed.ai/tutorials/automatic-tensor-parallelism.

Aminabadi, R. Y., Rajbhandari, S., Awan, A. A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, 2022.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

Ben Allal, L., Muennighoff, N., Kumar Umapathi, L., Lipkin, B., and von Werra, L. A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness, 2022.

Cai, T., Li, Y., Geng, Z., Peng, H., Lee, J. D., Chen, D., and Dao, T. Medusa: Simple llm inference acceleration framework with multiple decoding heads. *arXiv preprint arXiv:2401.10774*, 2024.

Chen, C., Borgeaud, S., Irving, G., Lespiau, J.-B., Sifre, L., and Jumper, J. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023a.

Chen, M., Tworek, J., Jun, H., Yuan, Q., de Oliveira Pinto, H. P., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., Ryder, N., Pavlov, M., Power, A., Kaiser, L., Bavarian, M., Winter, C., Tillet, P., Such, F. P., Cummings, D., Plappert, M., Chantzis, F., Barnes, E., Herbert-Voss, A., Guss, W. H., Nichol, A., Paino, A., Tezak, N., Tang, J., Babuschkin, I., Balaji, S., Jain, S., Saunders, W., Hesse, C., Carr, A. N., Leike, J., Achiam, J., Misra, V., Morikawa, E., Radford, A., Knight, M., Brundage, M., Murati, M., Mayer, K., Welinder, P., McGrew, B., Amodei, D., McCandlish, S., Sutskever, I., and Zaremba, W. Evaluating large language models trained on code, 2021.

Chen, Z., Yang, X., Lin, J., Sun, C., Huang, J., and Chang, K. C.-C. Cascade speculative drafting for even faster llm inference. *arXiv preprint arXiv:2312.11462*, 2023b.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems. *arXiv preprint arXiv:2110.14168*, 2021.

Cui, P. and Sachan, M. Adaptive and personalized exercise generation for online language learning. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 10184–10198, 2023.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations*, 2023.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022.

Du, X., Liu, M., Wang, K., Wang, H., Liu, J., Chen, Y., Feng, J., Sha, C., Peng, X., and Lou, Y. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861*, 2023.

Gugger, S., Debut, L., Wolf, T., Schmid, P., Mueller, Z., Mangrulkar, S., Sun, M., and Bossan, B. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate, 2022.

Guo, J., Zhang, Z., Xu, L., Wei, H.-R., Chen, B., and Chen, E. Incorporating bert into parallel sequence decoding with adapters. *Advances in Neural Information Processing Systems*, 33:10843–10854, 2020.

He, Z., Zhong, Z., Cai, T., Lee, J. D., and He, D. Rest: Retrieval-based speculative decoding. *arXiv preprint arXiv:2311.08252*, 2023.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2019.

Hooper, C., Kim, S., Mohammadzadeh, H., Genc, H., Keutzer, K., Gholami, A., and Shao, S. Speed: Speculative pipelined execution for efficient decoding. *arXiv preprint arXiv:2310.12072*, 2023.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Kim, S., Mangalam, K., Moon, S., Malik, J., Mahoney, M. W., Gholami, A., and Keutzer, K. Speculative decoding with big little decoder. *Advances in Neural Information Processing Systems*, 36, 2024.

Kool, W., van Hoof, H., and Welling, M. Ancestral gumbel-top-k sampling for sampling without replacement. *Journal of Machine Learning Research*, 21(47):1–36, 2020. URL http://jmlr.org/papers/v21/19-985.html.

Leviathan, Y., Kalman, M., and Matias, Y. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pp. 19274–19286. PMLR, 2023.

Li, P., Li, L., Zhang, M., Wu, M., and Liu, Q. Universal conditional masked language pre-training for neural machine translation. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 6379–6391, 2022.

Li, Y., Wei, F., Zhang, C., and Zhang, H. Eagle: Speculative sampling requires rethinking feature uncertainty. *arXiv preprint arXiv:2401.15077*, 2024.

Lin, C.-Y. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out*, pp. 74–81, Barcelona, Spain, July 2004. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/W04-1013.

Lin, M., Chen, R., and Liu, J. S. Lookahead strategies for sequential monte carlo. *Statistical Science*, 28(1):69–94, 2013.

Liu, X., Hu, L., Bailis, P., Stoica, I., Deng, Z., Cheung, A., and Zhang, H. Online speculative decoding. *arXiv preprint arXiv:2310.07177*, 2023.

Lu, X., Welleck, S., West, P., Jiang, L., Kasai, J., Khashabi, D., Le Bras, R., Qin, L., Yu, Y., Zellers, R., et al. Neurologic a* esque decoding: Constrained text generation with lookahead heuristics. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 780–799, 2022.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Jin, H., Chen, T., and Jia, Z. Towards efficient generative large language model serving: A survey from algorithms to systems. *arXiv preprint arXiv:2312.15234*, 2023.

Miao, X., Oliaro, G., Zhang, Z., Cheng, X., Wang, Z., Zhang, Z., Wong, R. Y. Y., Zhu, A., Yang, L., Shi, X., 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*, pp. 932–949, 2024.

Narayan, S., Cohen, S. B., and Lapata, M. Don't give me the details, just the summary! Topic-aware convolutional neural networks for extreme summarization. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, 2018.

Narayanan, D., Shoeybi, M., Casper, J., LeGresley, P., Patwary, M., Korthikanti, V., Vainbrand, D., Kashinkunti, P., Bernauer, J., Catanzaro, B., et al. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2021.

OpenAI. Gpt-4 technical report, 2023.

Ortmanns, S. and Ney, H. Look-ahead techniques for fast beam search. *Computer Speech & Language*, 14(1):15–32, 2000.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., et al. Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35:27730–27744, 2022.

Qi, W., Gong, Y., Yan, Y., Jiao, J., Shao, B., Zhang, R., Li, H., Duan, N., and Zhou, M. Prophetnet-ads: A looking ahead strategy for generative retrieval models in sponsored search engine. In *Natural Language Processing and Chinese Computing: 9th CCF International Conference, NLPCC 2020, Zhengzhou, China, October 14–18, 2020, Proceedings, Part II 9*, pp. 305–317. Springer, 2020.

Ribeiro, L. F., Bansal, M., and Dreyer, M. Generating summaries with controllable readability levels. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pp. 11669–11687, 2023.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Ruan, J. T., Sabir, F., and Chopra, P. Best prompting practices for using the llama 2 chat llm through amazon sagemaker jumpstart, November 2023. URL https://aws.amazon.com/cn/blogs/machine-learning/best-prompting-practices-for-using-the-llama-2-chat-llm-through-amazon-sagemaker-jumpstart/.

Santilli, A., Severino, S., Postolache, E., Maiorca, V., Mancusi, M., Marin, R., and Rodola, E. Accelerating transformer inference for translation via parallel decoding. In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12336–12355, Toronto, Canada, July 2023. Association for Computational Linguistics. URL https://aclanthology.org/2023.acl-long.689.

Saxena, A. Prompt lookup decoding, November 2023. URL https://github.com/apoorvumang/prompt-lookup-decoding/.

See, A., Liu, P. J., and Manning, C. D. Get to the point: Summarization with pointer-generator networks. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 1073–1083, 2017.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019.

Song, Y., Meng, C., Liao, R., and Ermon, S. Accelerating feedforward computation via parallel nonlinear equation solving. In *International Conference on Machine Learning*, pp. 9791–9800. PMLR, 2021.

Spector, B. and Re, C. Accelerating llm inference with staged speculative decoding. *arXiv preprint arXiv:2308.04623*, 2023.

Stern, M., Shazeer, N., and Uszkoreit, J. Blockwise parallel decoding for deep autoregressive models. *Advances in Neural Information Processing Systems*, 31, 2018.

Su, Y., Cai, D., Wang, Y., Vandyke, D., Baker, S., Li, P., and Collier, N. Non-autoregressive text generation with pre-trained language models. In *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pp. 234–243, 2021.

Sun, Z., Suresh, A. T., Ro, J. H., Beirami, A., Jain, H., and Yu, F. Spectr: Fast speculative decoding via optimal transport. *Advances in Neural Information Processing Systems*, 36, 2024.

Team, G., Anil, R., Borgeaud, S., Wu, Y., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023a.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023b.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Wan, D., Liu, M., Mckeown, K., Dreyer, M., and Bansal, M. Faithfulness-aware decoding strategies for abstractive summarization. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, pp. 2864–2880, 2023.

Wang, Y., He, S., Chen, G., Chen, Y., and Jiang, D. Xlm-d: Decorate cross-lingual pre-training model as non-autoregressive neural machine translation. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 6934–6946, 2022.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., Davison, J., Shleifer, S., von Platen, P., Ma, C., Jernite, Y., Plu, J., Xu, C., Scao, T. L., Gugger, S., Drame, M., Lhoest, Q., and Rush, A. M. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pp. 38–45, Online, October 2020. Association for Computational Linguistics. URL https://www.aclweb.org/anthology/2020.emnlp-demos.6.

Wu, J. and Frazier, P. Practical two-step lookahead bayesian optimization. *Advances in neural information processing systems*, 32, 2019.

Xia, H., Ge, T., Wang, P., Chen, S.-Q., Wei, F., and Sui, Z. Speculative decoding: Exploiting speculative execution for accelerating seq2seq generation. In *Findings of the Association for Computational Linguistics: EMNLP 2023*, pp. 3909–3925, 2023.

Xia, H., Yang, Z., Dong, Q., Wang, P., Li, Y., Ge, T., Liu, T., Li, W., and Sui, Z. Unlocking efficiency in large language model inference: A comprehensive survey of speculative decoding. *arXiv preprint arXiv:2401.07851*, 2024.

Xiao, Y., Wu, L., Guo, J., Li, J., Zhang, M., Qin, T., and Liu, T.-y. A survey on non-autoregressive generation for neural machine translation and beyond. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

Yang, N., Ge, T., Wang, L., Jiao, B., Jiang, D., Yang, L., Majumder, R., and Wei, F. Inference with reference: Lossless acceleration of large language models. *arXiv preprint arXiv:2304.04487*, 2023a.

Yang, S., Lee, G., Cho, J., Papailiopoulos, D., and Lee, K. Predictive pipelined decoding: A compute-latency trade-off for exact llm decoding. *arXiv preprint arXiv:2307.05908*, 2023b.

Zhang, B. and Sennrich, R. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019.

Zhang, J., Wang, J., Li, H., Shou, L., Chen, K., Chen, G., and Mehrotra, S. Draft & verify: Lossless large language model acceleration via self-speculative decoding. *arXiv preprint arXiv:2309.08168*, 2023.

Zhang, M., Lucas, J., Ba, J., and Hinton, G. E. Lookahead optimizer: k steps forward, 1 step back. *Advances in neural information processing systems*, 32, 2019.

Zhang, P., Zeng, G., Wang, T., and Lu, W. Tinyllama: An open-source small language model. *arXiv preprint arXiv:2401.02385*, 2024.

Zheng, L., Chiang, W.-L., Sheng, Y., Zhuang, S., Wu, Z., Zhuang, Y., Lin, Z., Li, Z., Li, D., Xing, E., et al. Judging llm-as-a-judge with mt-bench and chatbot arena. *Advances in Neural Information Processing Systems*, 36, 2024.

Zhou, Y., Lyu, K., Rawat, A. S., Menon, A. K., Rostamizadeh, A., Kumar, S., Kagy, J.-F., and Agarwal, R. Distillspec: Improving speculative decoding via knowledge distillation. *arXiv preprint arXiv:2310.08461*, 2023.

A. Algorithms

Algorithm 2 Jacobi decoding with greedy sampling (adapted from previous work (Santilli et al., 2023))

Input: prompt $\mathbf{x}^0$, model P_M, generation length m
Initialize $\mathbf{y}^0 = (y_1^0, y_2^0, ..., y_m^0)$
Initialize $\mathbf{o} \leftarrow \emptyset$
for $i = 1$ **to** m **do**
    if size($\mathbf{o}$)$\geq$i **then**
        $\mathbf{y}^i = \mathbf{y}^{i-1}$
        continue
    end if
    $\mathbf{y}_{1:m}^i \leftarrow \mathrm{argmax}(P_M(\mathbf{y}_{1:m}^i | \mathbf{y}_{1:m}^{i-1}, \mathbf{x}^0))$
    $\mathbf{o}$.append(AcceptedTokens($\mathbf{y}^i$, $\mathbf{y}^{i-1}$, $\mathbf{o}$))
end for
Output: $\mathbf{o} = (y_1, y_2, ..., y_m)$

Algorithm 3 Greedy Verification with LOOKAHEAD DECODING

```
input prefill x^0, model P_M, n-grams g^i with i ∈ [1, G]
output o {accepted tokens of length 1 to N}
function GreedyVerification(x^0, P_M, g)
    V, D, o ← ∅, ∅, ∅
    for i = 1 to G do
        V.append(g^i_{2:}) {each is a n-1 gram}
        D.append(P_M(g^{i'}_{2:}, x_{next} | g^i_{2:}, x^0))
        {obtain last token of x^0 and all g^i_{2:}'s outputs – totally N probability distributions}
    end for
    for i = 1 to N − 1 do
        j ← 1
        is_accept ← 0
        P ← D[1]_i {D[1] is a series of N probability distributions;all D[j]_i should be the same as different distributions
        are removed; size(D)> 0 is guaranteed}
        while j ≤ size(V) do
            s_j ← V[j]_i
            if s_j = argmax P then
                {accepted, update all potential speculations and probabilities}
                o.append(s_j)
                is_accept ← 1
                V_new, D_new ← ∅, ∅
                for k = j to size(V) do
                    if s_j=V[k]_i then
                        V_new.append(V[k])
                        D_new.append(D[k])
                    end if
                end for
                V, D ← V_new, D_new
                break
            else
                {rejected, go to next speculation }
                j ← j + 1
            end if
        end while
        if is_accept then
            continue
        else
            {guarantee one step movement}
            o.append(argmax P)
            break
        end if
    end for
    if is_accept then
        o.append(argmax D[1]_N)
    end if
    return o
end function
```

Algorithm 4 Sample Verification with LOOKAHEAD DECODING

```
input prefill $\mathbf{x}^0$, model $P_M$, n-grams $\mathbf{g}^i$ with $i \in [1, G]$
output $\mathbf{o}$ {accepted tokens of length 1 to N}
function SampleVerification($\mathbf{x}^0$, $P_M$, $\mathbf{g}$)
    $\mathbf{V}, \mathbf{D}, \mathbf{o} \leftarrow \emptyset, \emptyset, \emptyset$
    for $i = 1$ to $G$ do
        $\mathbf{V}$.append($\mathbf{g}^i_{2:}$){each is a n-1 gram}
        $\mathbf{D}$.append($P_M(\mathbf{g}^{i'}_{2:}, \mathbf{x}_{next} | \mathbf{g}^i_{2:}, \mathbf{x}^0)$)
        {obtain last token of $\mathbf{x}^0$ and all $\mathbf{g}^i_{2:}$'s outputs – totally N probability distributions}
    end for
    for $i = 1$ to $N - 1$ do
        $j \leftarrow 1$
        $is\_accept \leftarrow 0$
        $\mathcal{P}_j \leftarrow \mathbf{D}[j]_i$ {$\mathbf{D}[j]$ is a series of N probability distributions;all $\mathbf{D}[j]_i$ should be the same; size($\mathbf{D}$)$>$ 0 is guaranteed}
        while $j \leq$ size($\mathbf{V}$) do
            $s_j \leftarrow \mathbf{V}[j]_i$
            sample $r \sim U(0, 1)$
            if $r \leq \mathcal{P}_j(s_j)$ then
                {accepted, update all potential speculations and probabilities}
                $\mathbf{o}$.append($s_j$)
                $is\_accept \leftarrow 1$
                $\mathbf{V}_{new}, \mathbf{D}_{new} \leftarrow \emptyset, \emptyset$
                for $k = j$ to size($\mathbf{V}$) do
                    if $s_j$=$\mathbf{V}[k]_i$ then
                        $\mathbf{V}_{new}$.append($\mathbf{V}[k]$)
                        $\mathbf{D}_{new}$.append($\mathbf{D}[k]$)
                    end if
                end for
                $\mathbf{V}, \mathbf{D} \leftarrow \mathbf{V}_{new}, \mathbf{D}_{new}$
                break
            else
                {rejected, go to next speculation }
                $\mathcal{P}_j(s_j) = 0$
                $\mathcal{P}_{j+1} =$ norm($\mathcal{P}_j$)
                $j \leftarrow j + 1$
            end if
        end while
        if $is\_accept$ then
            continue
        else
            {guarantee one step movement}
            sample $\mathbf{x}_{next} \sim \mathcal{P}_j$
            $\mathbf{o}$.append($\mathbf{x}_{next}$)
            break
        end if
    end for
    if $is\_accept$ then
        $\mathbf{o}$.append(sample $\mathbf{x}_{next} \sim \mathbf{D}[1]_N$)
    end if
    return $\mathbf{o}$
end function
```

B. Proof: Output distribution preserved disjoint n-gram verification

The sampling verification in LOOKAHEAD DECODING is adapted from the algorithm in Specinfer but with all speculations generated by the greedy sample. It does not change the output distribution from a fundamental point that how the draft model generates speculations is unimportant.

Theorem A For a given LLM, prompt and previously generated tokens $\mathbf{x} = (x_1, x_2, ..., x_i)$, and G speculations $\mathbf{s} = (s_1, s_2, ..., s_G)$ of next token x_{i+1}. Each speculation token is sampled by a greedy sample (i.e., probability of 1). We use $P(v|\mathbf{x})$ to represent the probability of $x_{i+1} = v$ sampled by the LLM and use $Q(v|\mathbf{x})$ to represent the probability of $x_{i+1} = v$ sampled by our proposed algorithm 4. We use $P(v)$ and $Q(v)$ for short. We need to prove $P(v) = Q(v)$ for any G, and any v and s_j from the full vocabulary V.

Proof. The proof of this part corresponds to line 14 to line 44 in algorithm 4. Given speculations $\mathbf{s}$, we use $a_j(v)$ to represent the probability that the token v is accepted by the j-th speculation (line 18-line 30), and $r_j(s_j)$ is the probability that the token s_j is rejected by the j-th speculation (line 30-line 35). Moreover, $a'_{G+1}(v)$ is the probability of being accepted by the sampling at line 41. For simplicity, we use a_j to represent $a_j(v)$, a'_j to represent $a'_j(v)$, and use r_j to represent $r_j(s_j)$. We use $\mathcal{P}_1$ to represent the probability distribution obtained in line 13 and $\mathcal{P}_j$ to present the updated probability before the j-th speculation. We have $\mathcal{P}_1(v) = P(v)$ as $\mathcal{P}_1(v)$ is never updated. We define $Q_G(v)$ is the probability of $x_{i+1} = v$ sampled by algorithm 4 when we have G speculations. Then we should have:

$$Q_G(v) = a_1 + r_1 a_2 + r_1 r_2 a_3 + ... + a_G \prod_{k=1}^{G-1} r_k + a'_{G+1} \prod_{k=1}^{G} r_k$$

We define the j-th speculation's token as s_j. We use induction to prove $Q_G(v) = P(v)$ for any $G \geq 1$, any $v \in V$, and any $s_j \in V$ with $1 \leq j \leq G$:

1) When $G = 1$, we have $Q_G(v) = a_1 + r_1 a'_2$. The initial guess s_1 can be either the same at v or be different from v.

   (1) When $s_1 = v$, a_1 equals $\mathcal{P}_1(v)$ at line 17, which is the same as $P(v)$ as it is never updated. Upon this, we have $r_1 = 1 - a_1 = 1 - \mathcal{P}_1(v) = 1 - P(v)$. And, a'_2 is the updated probability $\mathcal{P}_2(v)$ at line 42. Since $\mathcal{P}_2(v)$ is set to zero once rejected at line 32, $a'_2 = 0$. In this case, $Q_G(v) = P(v) + (1 - P(v)) * 0 = P(v)$.

   (2) When $s_1 \neq v$, a_1 should be 0 even if s_i is accepted. Moreover, we have $r_1 = 1 - \mathcal{P}_1(s_1)$. Then $\mathcal{P}_2(v)$ is updated to $\frac{\mathcal{P}_1(v)}{1-\mathcal{P}_1(s_1)}$ at lines 32 and 33. Then $a'_2 = \mathcal{P}_2(v)$. In this case, $Q_G(v) = 0 + r_1 * \frac{P(v)}{r_1} = P(v)$.

2) When $G = g$ holds, which means $Q_g(v) = a_1 + r_1 a_2 + ... + a_g \prod_{k=1}^{g-1} r_k + a'_{g+1} \prod_{k=1}^{g} r_k = P(v)$ for any $s_j, v \in V$, $1 \leq j \leq g$.

   We prove $Q_{g+1}(v) = Q_g(v) - a'_{g+1} \prod_{k=1}^{g} r_k + a_{g+1} \prod_{k=1}^{g} r_k + a'_{g+2} \prod_{k=1}^{g+1} r_k = P(v)$ for the same $s_j, v \in V$, $1 \leq j \leq g$, and any $s_{g+1} \in V$.

   (1) When $s_g \neq v$, we have $a_g = 0$. If $\mathcal{P}_g(v) = 0$, we have all $a'_{g+1} = 0$, $a_{g+1} = 0$ and $a'_{g+2} = 0$. It ensures that $Q_{g+1}(v) = Q_g(v) - 0 + 0 + 0 = Q_g(v) = P(v)$.
   If $\mathcal{P}_g(v) \neq 0$, $\mathcal{P}_{g+1}(v) = \frac{\mathcal{P}_g(v)}{r_g} = \frac{\mathcal{P}_1(v)}{\prod_{k=1}^{g} r_k}$ since $s_g \neq v$ by observation.

   Then we have $Q_{g+1}(v) = Q_g(v) - \mathcal{P}_{g+1}(v) \prod_{k=1}^{g} r_k + a_{g+1}(v) \prod_{k=1}^{g} r_k + a'_{g+2} \prod_{k=1}^{g+1} r_k = Q_g(v) - \frac{\mathcal{P}_1(v)}{\prod_{k=1}^{g} r_k} \prod_{k=1}^{g} r_k + a_{g+1} \prod_{k=1}^{g} r_k + a'_{g+2} \prod_{k=1}^{g+1} r_k = Q_g(v) - P(v) + a_{g+1}(v) \prod_{k=1}^{g} r_k + a'_{g+2} \prod_{k=1}^{g+1} r_k$. Here we have another two cases:

   ① If $s_{g+1} = v$, $a_{g+1} \prod_{k=1}^{g} r_k = \frac{\mathcal{P}_1(v_i=v)}{\prod_{k=1}^{g} r_k} \prod_{k=1}^{g} r_k = P(v)$ and $a'_{g+2} = \mathcal{P}_{g+2}(v) = 0$. We have $Q_{g+1}(v) = Q_g(v) - P(v) + P(v) + 0 = Q_g(v) = P(v)$.

② If $s_{g+1} \neq v$, $a_{g+1} = 0$. $a'_{g+2} \prod_{k=1}^{g+1} r_k = \mathcal{P}_{g+2}(v) \prod_{k=1}^{g+1} r_k = \frac{\mathcal{P}_1(v)}{\prod_{k=1}^{g+1} r_k} \prod_{k=1}^{g+1} r_k = P(v)$. So $Q_{g+1}(v) = Q_g(v) - P(v) + 0 + P(v) = Q_g(v) = P(v)$

(2) When $s_g = v$, $\mathcal{P}_{g+1}(v)$ is set to zero at line 32 after this step. In this case, we have $a'_{g+1} = \mathcal{P}_{g+1}(v) = 0$, $a_{g+1} = \mathcal{P}_{g+1}(v) = 0$ and $a'_{g+2} = 0$. It makes that $Q_{g+1}(v) = Q_g(v) - 0 + 0 + 0 = Q_g(v) = P(v)$

□

This part of the proof guarantees that from line 14 to line 44 in Algorithm 4, any new token appended to $\mathbf{o}$ can follow the original distribution of the LLM. Line 21 to line 28 guarantees that sequences in $\mathbf{V}$ share the same prefix of length $i-1$ in every iteration. This further guarantees that $\mathcal{P}$ from $\mathbf{D}[j]_i$ is the same for all j, follows the wanted distribution. Thus, the correctness of the whole sampling algorithm is proved.

C. Derivation of Expectation of The Number of Accepted Tokens

We first start with single-candidate speculation. We need to obtain the probability of accepting i tokens as $P(\#accepted\ tokens = i)$ for all possible i. Since the speculation's length is γ, the probability of accepting i tokens with $i \geq \gamma + 2$ is 0. $P(\#accepted\ tokens = 1)$ is the probability of the first token being rejected, which is $1 - \alpha$. The probability $P(\#accepted\ tokens = i) = P(\#accepted\ tokens = i-1) * \alpha$, for all $i \leq \gamma$. The probability $P(\#accepted\ tokens = \gamma + 1)$ is accepting all tokens, which is α^γ. Thus we have the following, which is Eq. 4:

$$\begin{aligned} E(\#tokens) &= \sum_{i=1}^{\gamma+1} i * P(\#accepted\ tokens = i) \\ &= 1*(1-\alpha) + 2*(1-\alpha)*\alpha + ... + (\gamma+1)*\alpha^\gamma \\ &= (1-\alpha) + (2\alpha - 2\alpha^2) + (3\alpha^2 - 3\alpha^3) + ... + (\gamma+1)\alpha^\gamma \\ &= 1 + (-\alpha + 2\alpha) + (-2\alpha^2 + 3\alpha^2) + (-3\alpha^3 + 4\alpha^3) + ... + (\gamma+1)\alpha^\gamma \\ &= 1 + \alpha + \alpha^2 + \alpha^3 + ... + \alpha^\gamma \\ &= \frac{1-\alpha^{\gamma+1}}{1-\alpha} \end{aligned} \tag{8}$$

We then investigate the case of speculations with a batch size of b. We need to obtain the probability of accepting i tokens as $P(\#accepted\ tokens = i)$. Since all speculations' length is γ, the probability of accepting a tokens with $a \geq \gamma + 2$ is 0. We use p_i to denote $(1-\alpha^i)^b$, which is the probability that at most i tokens are accepted in all b speculations. For all $i \leq \gamma$, we should have $P(\#accepted\ tokens = i) = p_i - p_{i-1}$. And, the probability $P(\#accepted\ tokens = \gamma + 1)$ should be $(1 - p_\gamma)$. Thus we have the following, which is Eq. 5:

$$\begin{aligned} E(\#tokens) &= \sum_{i=1}^{\gamma+1} i * P(\#accepted\ tokens = i) \\ &= \sum_{i=1}^{\gamma} i(p_i - p_{i-1}) + (\gamma+1)*(1-p_\gamma) \\ &= (p_1 - p_0) + (2p_2 - 2p_1) + (3p_3 - 3p_2) + ... + (\gamma+1)(1-p_\gamma) \\ &= -p_0 + (p_1 - 2p_1) + (2p_2 - 3p_2) + (3p_3 - 4p_3)... + (\gamma+1) \\ &= -p_0 - p_1 - p_2 - ... - p_\gamma + (\gamma+1) \\ &= (\gamma+1) - \sum_{i=1}^{\gamma}(1-\alpha^i)^b \end{aligned} \tag{9}$$

D. Prompt for LLaMA-2-Chat on Summarization Tasks

We use the following as the prompt for summarization task, modified from (Ruan et al., 2023).

➤ Prompt:
[INST] <<SYS>>
You are an intelligent chatbot. Answer the questions only using the following context:
{Original Text}
Here are some rules you always follow:
- Generate human readable output, avoid creating output with gibberish text.
- Generate only the requested output, don't include any other language before or after the requested output.
- Never say thank you, that you are happy to help, that you are an AI agent, etc. Just answer directly.
- Generate professional language typically used in business documents in North America.
- Never generate offensive or foul language.
<< /SYS>>
Briefly summarize the given context. [/INST]
Summary:

E. Verification of Generation Quality with LOOKAHEAD DECODING

Evaluation of LOOKAHEAD DECODING's Generation Quality with Sampling on MT-Bench. We further compared the performance of LLaMA-2-7b-chat with LOOKAHEAD DECODING and autoregressive decoding on MT-Bench as in Tab. 5 as a supplement to the comparisons in §5.3, using MT-Bench score as a better metric. Sampling is activated, and the default setting (e.g., temperature) in MT-Bench is used. The MT-Bench score is preserved in an acceptable interval by activating LOOKAHEAD DECODING.

Table 5: Compare LOOKAHEAD DECODING and autoregressive decoding on MT-Bench.

	LOOKAHEAD DECODING	AUTOREGRESSIVE DECODING
SCORE	6.51	6.47

Generation Quality with Greedy Search is not changed. Theoretically, LOOKAHEAD DECODING does not change the output generation of greedy search due to the verification mechanism. However, LOOKAHEAD DECODING's output does not perfectly align with the huggingface's implementation of greedy search in practice. We attribute this discrepancy to numerical accuracy issues. To substantiate this claim, we compared the output results as follows. We use the LLaMA-2-7b-Chat model's single precision (FP32) inference with huggingface's greedy search on 160 turns on the MT-Bench dataset as a baseline. With single precision inference, the outputs of LOOKAHEAD DECODING (on 1GPU, 4GPUs, and 8GPUs) are the same as the output of the baseline. With half-precision (FP16) inference, huggingface's greedy search has 35 out of 160 (w/o FlashAttention) and 42 out of 160 (w/ FlashAttention) answers not perfectly aligned with the baseline output. In contrast, LOOKAHEAD DECODING and its integration with FlashAttention and multi-GPU inference has 35-44 results different from the baseline output under different settings. We claim this result can show that LOOKAHEAD DECODING can retain the output distribution using a greedy search within the numerical error range (not worse than huggingface's half-precision inference). Besides, Tab. 2 also strengthens the statements for greedy search.

Generation Quality with LP and FlashAttention Augmentation is not changed. We verify that FlashAttention and LP Support will not change the compression ratio ($\mathcal{S}$) of vanilla LOOKAHEAD DECODING. We compared each 18 generations of LOOKAHEAD DECODING w/ FlashAttention and w/o FlashAttention (7B and 13B model on MT-Bench, HumanEval, and ClassEval); the average $\mathcal{S}$ w/ FlashAttention is 3.267 while w/o FlashAttention is 3.259, with less than 0.3% differences. We also compared 6 generations of LOOKAHEAD DECODING on a single GPU and 12 generations with LP (7B model on MT-Bench, HumanEval, and ClassEval, both with $N = 5$, $W = 15$, and $G = 15$). The average $\mathcal{S}$ on a single GPU is 2.558, while on multiple GPUs, it is 2.557, with less than 0.1% differences. We claim that our advanced support does not change $\mathcal{S}$.

F. Comparing LOOKAHEAD DECODING with Speculative Decoding and Jacobi Decoding

We evaluated the speedup of LOOKAHEAD DECODING (LD) against autoregressive decoding (AD), speculative decoding (SD), Medusa, Parallel Jacobi decoding (PJ), Parallel GS-Jacobi decoding (PGJ) on A100 GPU in Tab 6. PJ and PGJ are Jacobi decoding variants proposed in previous research (Santilli et al., 2023). Speculative decoding needs a carefully selected or tuned draft model to achieve speedup, and our selection (i.e., Tinyllama 1b (Zhang et al., 2024) and Vicuna 7b (Zheng et al., 2024)) only shows a slowdown. PJ gives slowdowns as it needs a large number of tokens as input per step (i.e., the per-step cost is very high). PGJ shows gentle speedup (i.e., 1.1x-1.2x) as it differs from PJ in that it only uses b tokens generated from the last step as input. Medusa shows the largest speedup. We use greedy search in these settings.

Table 6: Compare LOOKAHEAD DECODING with other methods.

MODEL & DATASET	AD	LD	MEDUSA	SD (TINYLLAMA 1B)	SD (VICUNA 7B)	PJ	PGJ - B=5	PGJ - B=16
VICUNA-7B ON MT-BENCH	1.0X	1.7X	2.3X	0.7X	-	0.4X	1.1X	1.1X
VICUNA-7B ON HUMANEVAL	1.0X	2.1X	2.5X	0.8X	-	0.7X	1.1X	1.1X
VICUNA-13B ON MT-BENCH	1.0X	1.6X	2.4X	0.9X	0.7X	0.3X	1.2X	1.2X
VICUNA-13B ON HUMANEVAL	1.0X	2.0X	2.6X	1.0X	0.8X	0.6X	1.2X	1.2X