dInfer: An Efficient Inference Framework for Diffusion Language Models

YUXIN MA

Overview

- Intro & Motivation
- Framework Architecture
  - Design and optimizations of each module
  - Other optimizations
- Evaluation & Speed Results
- Optimizations after Technical Report
- Conclusion & Future Directions

Overview

- Intro & Motivation
- Framework Architecture
  - Design and optimizations of each module
  - Other optimizations
- Evaluation & Speed Results
- Optimizations after Technical Report
- Conclusion & Future Directions

Limitations of AR & Potential of dLLMs

- **Autoregressive (AR) LLMs:** Sequential token generation (left-to-right). Inherently constrained by latency (speed per token).
- **Diffusion LLMs (dLLMs):** Leverage iterative denoising for **parallel generation**.
- **The Potential:** Higher throughput and lower cost, especially for long sequence generation.

Bottlenecks of dLLM Inference

1. High Computational Cost:

- Iterative denoising requires multiple forward passes.
- Each step involves a large number of tokens, significantly increasing the total compute load.

2. Parallel Decoding Complexity:

- Naively increasing parallelism often leads to a quick degradation in the quality of the generated text.
- This is due to complex dependencies between predicted tokens.

3. Lack of Infrastructure :

- There is no standardized, unified inference framework (the "vLLM for dLLMs").
- This makes fair comparisons and widespread adoption nearly impossible.

Non-Block Diffusion & Block Diffusion

Non-Block Diffusion: Fully Bidirectional

Every single token attends to all other tokens—past, present, and future—during every denoising step.

Block Diffusion: Intra-Block Bidirectional, Inter-Block Causal

Tokens only attend to content within their current block and previous completed blocks.

Non-Block Diffusion & Block Diffusion

Non-Block Diffusion:

The naive decoding approach must perform a full sequence forward pass over the entire sequence for every single denoising iteration.

Direct KV-Cache reuse is impossible due to the full bidirectionality invalidating all previous states.

- Some works attempt to use an approximated KV-Cache (assuming distant tokens are stable between steps).
- This approach often degrades model performance and still necessitates periodic full sequence cache refreshes, limiting actual speed gains.

Block Diffusion: Efficient Caching, variable length

During decoding, only the current block requires a forward computation.

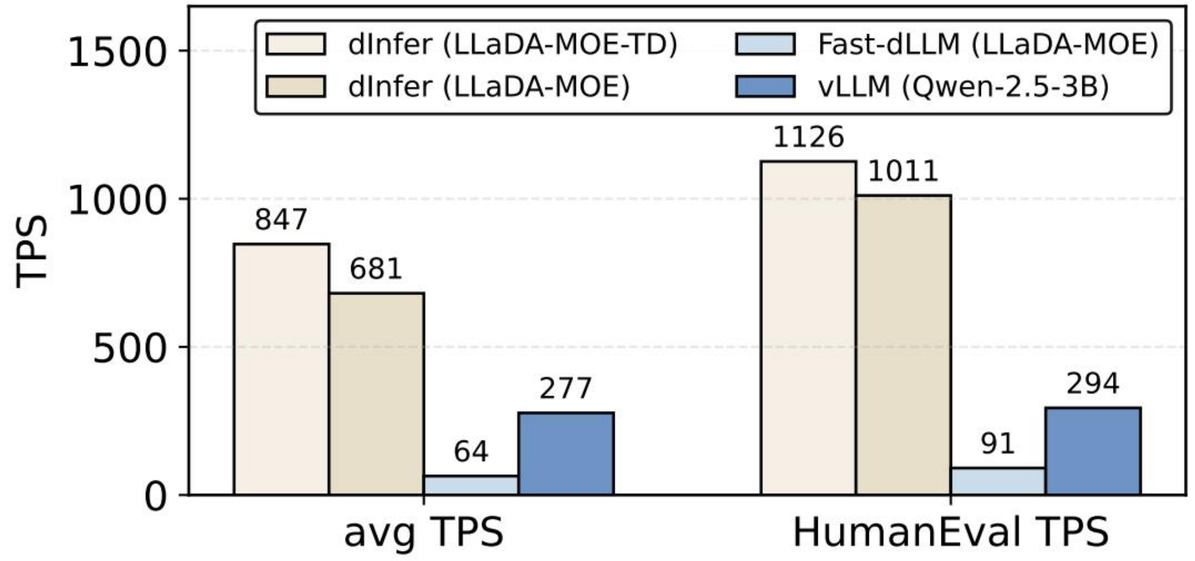
The KV states generated by all completed blocks can be fully cached and reused throughout the remaining decoding process.

dInfer's Goal

1. Be the first modular, highly efficient, and extensible inference framework for dLLMs.

2. Unlock the full parallel potential of dLLMs by addressing both algorithmic and systemic bottlenecks.

- dInfer drastically accelerates performance, achieving speeds significantly **faster than optimized AR models (Qwen + vLLM)** in latency-critical Batch Size = 1 scenarios.



Overview

- Intro & Motivation
- Framework Architecture
    - Design and optimizations of each module
    - Other optimizations
- Evaluation & Speed Results
- Optimizations after Technical Report
- Conclusion & Future Directions

Framework Architecture

The inference pipeline is decomposed into four independent components, allowing for flexible algorithm combination.

- **Decoder:** Logic layer that selects the next set of tokens based on logits.
- **Iteration Manager:** Controls the denoising steps, optimizing the number and sequence of iterations.
- **Model Runner:** Supports SOTA dLLMs like LLaDA-MoE, LLaDa2-mini, LLaDa2-flash.
- **KV-Cache Manager:** Manages the challenging dynamic KV-Cache updates specific to bidirectional dLLMs.

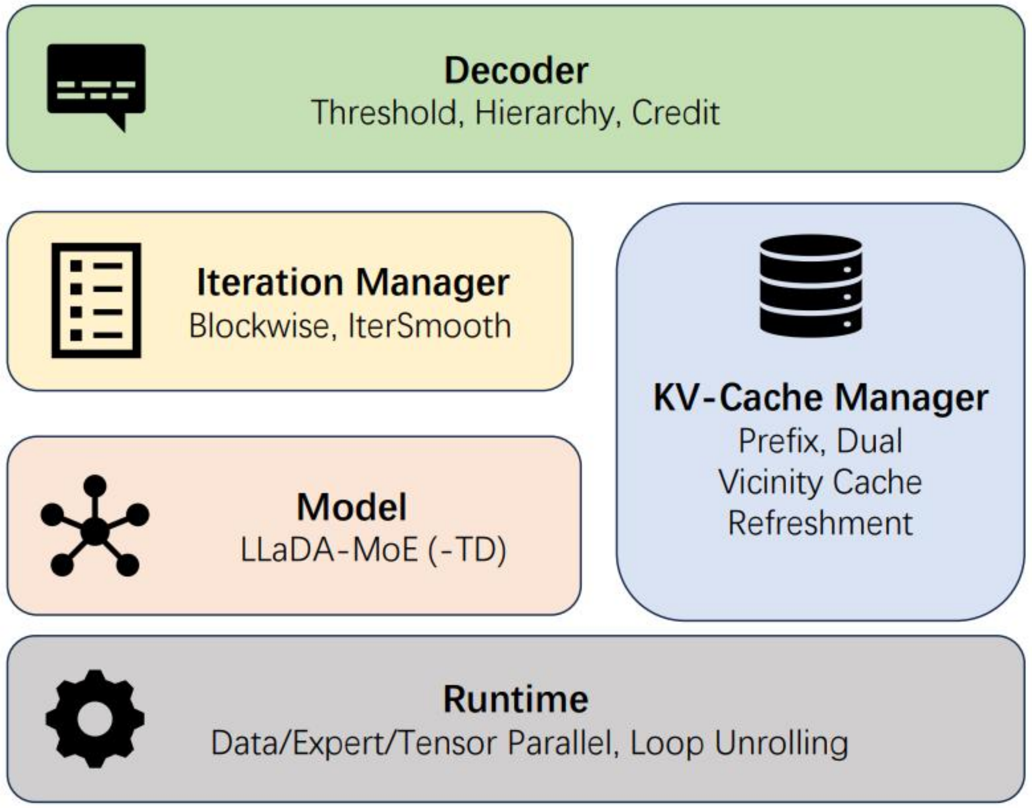
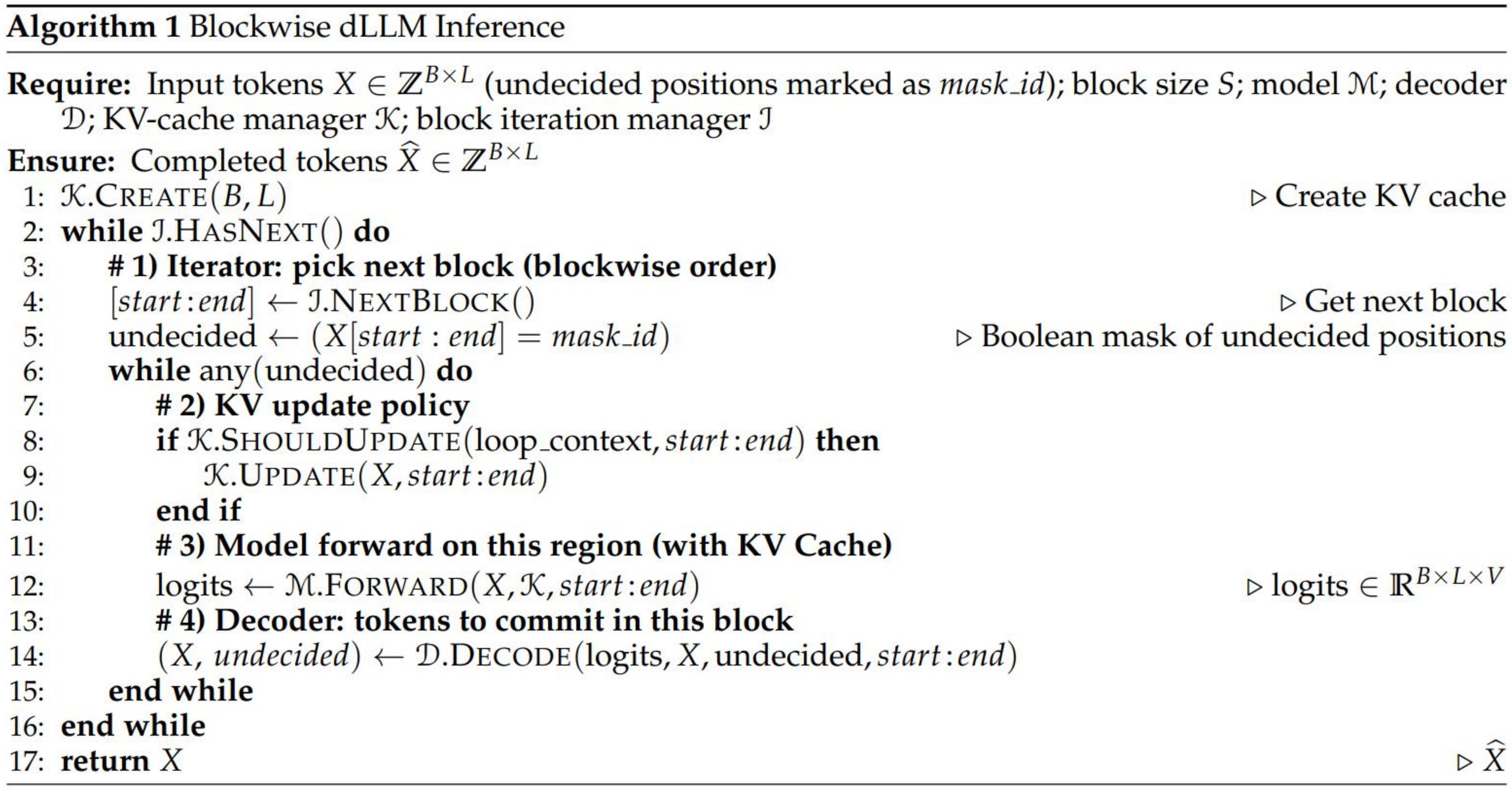


Figure 2: The architecture of the dInfer framework

Framework Architecture



Algorithm 1 Blockwise dLLM Inference

```
Require: Input tokens $X \in \mathbb{Z}^{B \times L}$ (undecided positions marked as $mask\_id$); block size $S$; model $\mathcal{M}$; decoder $\mathcal{D}$; KV-cache manager $\mathcal{K}$; block iteration manager $\mathcal{I}$
Ensure: Completed tokens $\widehat{X} \in \mathbb{Z}^{B \times L}$
$\mathcal{K}$.CREATE$(B, L)$                                    ▷ Create KV cache
while $\mathcal{I}$.HASNEXT() do
    # 1) Iterator: pick next block (blockwise order)
    $[start : end] \leftarrow \mathcal{I}$.NEXTBLOCK()             ▷ Get next block
    undecided $\leftarrow (X[start : end] = mask\_id)$    ▷ Boolean mask of undecided positions
    while any(undecided) do
        # 2) KV update policy
        if $\mathcal{K}$.SHOULDUPDATE(loop_context, $start : end$) then
            $\mathcal{K}$.UPDATE$(X, start : end)$
        end if
        # 3) Model forward on this region (with KV Cache)
        logits $\leftarrow \mathcal{M}$.FORWARD$(X, \mathcal{K}, start : end)$   ▷ logits $\in \mathbb{R}^{B \times L \times V}$
        # 4) Decoder: tokens to commit in this block
        $(X, undecided) \leftarrow \mathcal{D}$.DECODE(logits, $X$, undecided, $start : end$)
    end while
end while
return $X$                                                      ▷ $\widehat{X}$
```

Decoder - Decoding Strategy

By default, LLaDA uses **threshold decoding**, we propose 2 new decoding strategy to improve TPF(Token per Forward).

Hierarchical Decoding employs a divide-and-conquer approach, recursively partitioning large masked regions.

- This method breaks down local dependencies between tokens, resulting in a more stable and efficient parallel prediction process.

Credit Decoding introduces a "credit score" mechanism. Tokens that show stable, consistent predictions across multiple iterations earn credit.

- Prioritizes the commitment of already-certain tokens, avoiding redundant re-computation caused by single-iteration fluctuations.

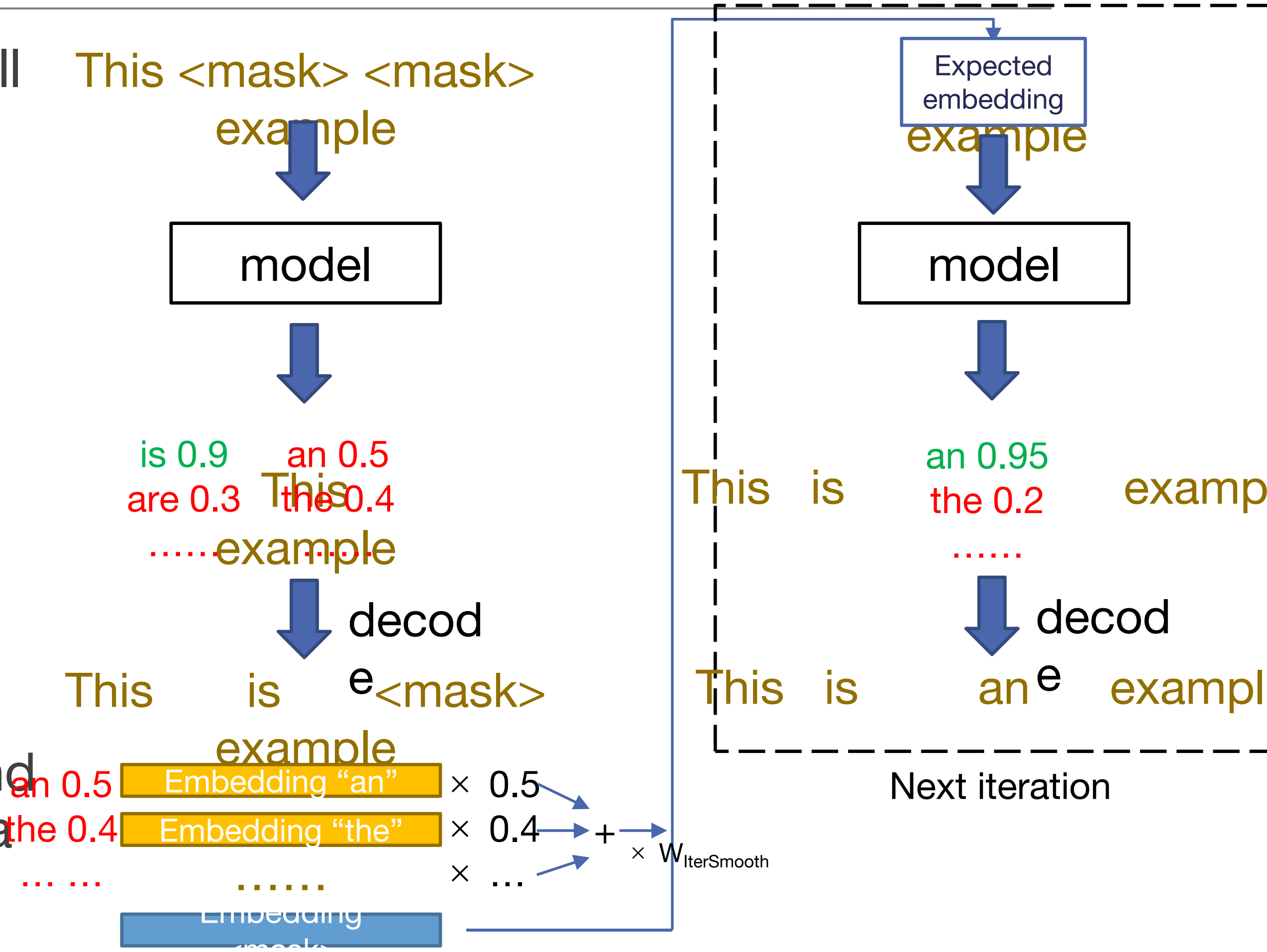
Iteration Manager - IterSmooth

Traditional greedy decoding discards the full probability distribution, only retaining the single argmax token. This is **wasted information**.

IterSmooth reuse the discarded logits distribution to improve model performance and decoding speed(TPF).

- IterSmooth Converts the **soft logits** into an Expected Embedding and injects this into subsequent iteration.

This leverages "wasted" information to significantly **enhance token confidence** and increase the number of tokens decoded in a single step.



KV-Cache - Vicinity Cache Refresh

For non-block diffusion dLLMs, the bidirectional attention mechanism means that the representation of all tokens changes during each denoising step. Consequently, the standard AR static KV-Cache cannot be directly used.

However, related research has shown that the KV-Cache states of tokens far away from the currently updated token remain approximately similar across successive diffusion steps.

We propose **Vicinity Cache Refresh** to maintain high accuracy while maximizing cache reuse.

- During a decoding step, we perform a full Cache Refresh not only on the current decoding block but also on a defined **vicinity** (a short span of tokens) both **before and after** the current block.
- This strategy ensures that the cached content is never too far from the updated tokens, effectively guaranteeing accuracy while drastically reducing the computation required compared to a full refresh.

Model Runner

TP/EP

Torch compile

Cuda graph

……

Post Training-Trajectory Distillation

Train the model to ”jump” between non-consecutive states within an optimal generation trajectory, thereby decoding multiple tokens in a single forward pass.

- Improve TPF while maintaining model performence

Loop unrolling

After compiling and capturing the diffusion forward into a CUDA Graph, a critical API call latency exists. This gap occurs between the Python host calling the CUDA API to launch the graph and the actual kernel execution starting on the device.

We unroll the diffusion loop to execute multiple forward steps without checking for branching, so there is no CUDA synchronization between the unrolled steps.

This technique effectively hides the API latency and minimizes synchronization costs, ensuring the GPU remains continuously utilized and maximizing the inference speed.

Overview

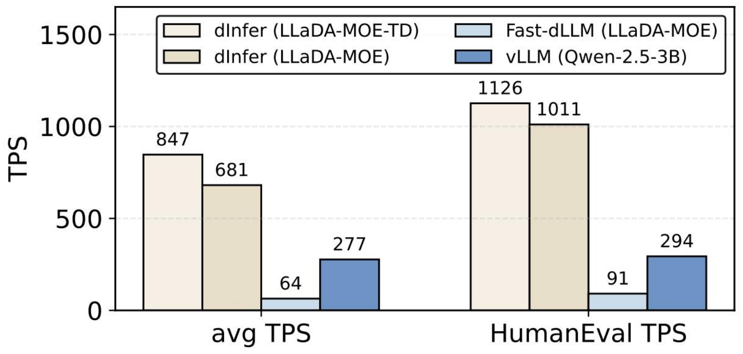
- Intro & Motivation
- Framework Architecture
    - Design and optimizations of each module
    - Other optimizations
- Evaluation & Speed Results
- Optimizations after Technical Report
- Conclusion & Future Directions

Evaluation & Speed Results

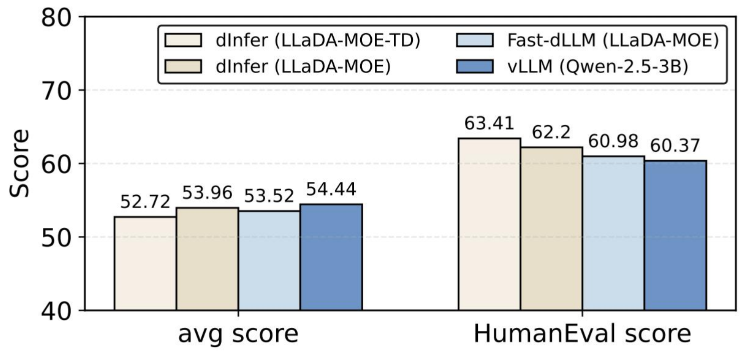
Significantly higher TPS on single sequence (on LLaDA-moe 7BA1B)

- 10x speedup over previous SOTA dLLM (Fast-dLLM)
- 2-3x faster than SOTA AR (vLLM + Qwen2.5-3B)
- Single-batch throughput reached 1126 TPS on HumanEval when combined with TD post training.

Model performance (accuracy) of LLaDA-moe was maintained or slightly improved due to robust decoding strategies.



(a). TPS



(b). Model performance

Evaluation & Speed Results

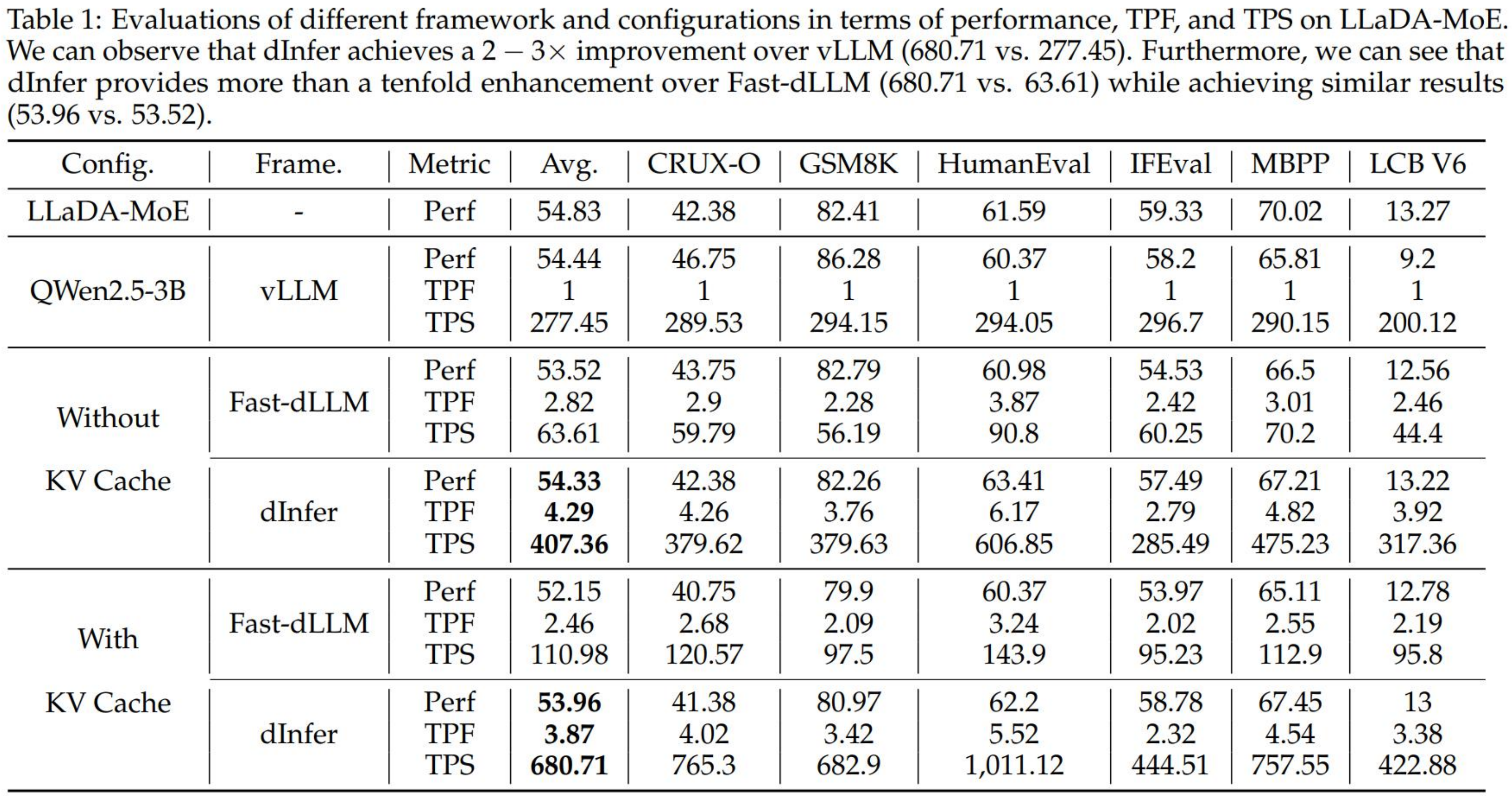


Table 1: Evaluations of different framework and configurations in terms of performance, TPF, and TPS on LLaDA-MoE. We can observe that dInfer achieves a $2-3\times$ improvement over vLLM (680.71 vs. 277.45). Furthermore, we can see that dInfer provides more than a tenfold enhancement over Fast-dLLM (680.71 vs. 63.61) while achieving similar results (53.96 vs. 53.52).

Config.	Frame.	Metric	Avg.	CRUX-O	GSM8K	HumanEval	IFEval	MBPP	LCB V6
LLaDA-MoE	-	Perf	54.83	42.38	82.41	61.59	59.33	70.02	13.27
QWen2.5-3B	vLLM	Perf	54.44	46.75	86.28	60.37	58.2	65.81	9.2
		TPF	1	1	1	1	1	1	1
		TPS	277.45	289.53	294.15	294.05	296.7	290.15	200.12
Without KV Cache	Fast-dLLM	Perf	53.52	43.75	82.79	60.98	54.53	66.5	12.56
		TPF	2.82	2.9	2.28	3.87	2.42	3.01	2.46
		TPS	63.61	59.79	56.19	90.8	60.25	70.2	44.4
	dInfer	Perf	**54.33**	42.38	82.26	63.41	57.49	67.21	13.22
		TPF	**4.29**	4.26	3.76	6.17	2.79	4.82	3.92
		TPS	**407.36**	379.62	379.63	606.85	285.49	475.23	317.36
With KV Cache	Fast-dLLM	Perf	52.15	40.75	79.9	60.37	53.97	65.11	12.78
		TPF	2.46	2.68	2.09	3.24	2.02	2.55	2.19
		TPS	110.98	120.57	97.5	143.9	95.23	112.9	95.8
	dInfer	Perf	**53.96**	41.38	80.97	62.2	58.78	67.45	13
		TPF	**3.87**	4.02	3.42	5.52	2.32	4.54	3.38
		TPS	**680.71**	765.3	682.9	1,011.12	444.51	757.55	422.88

Evaluation & Speed Results

Result of LLaDA-moe-TD:

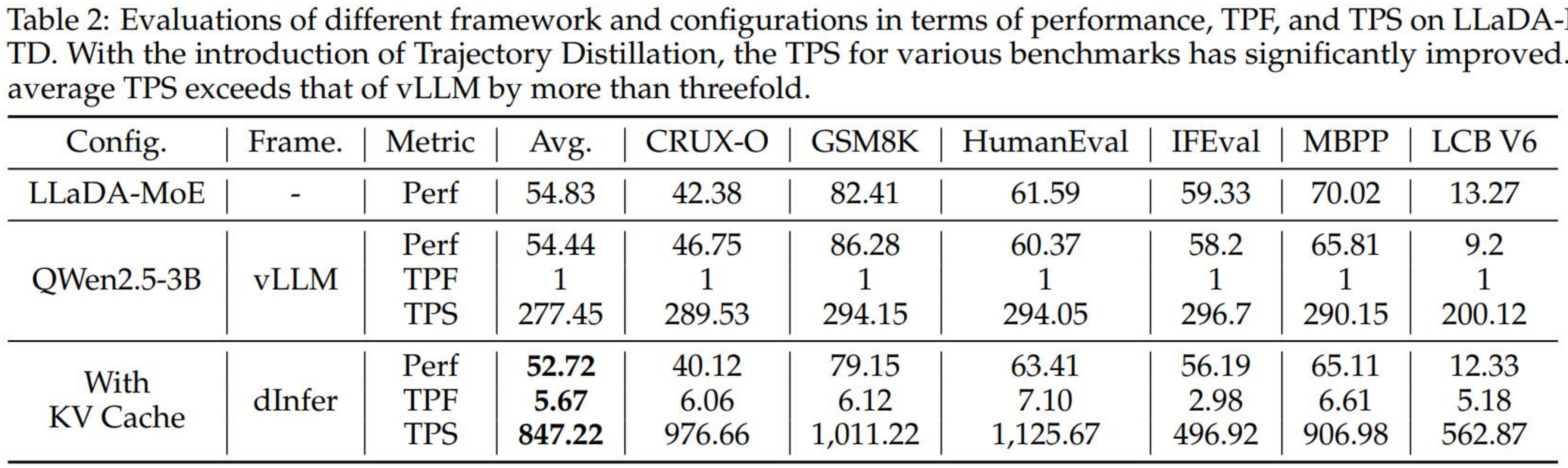


Table 2: Evaluations of different framework and configurations in terms of performance, TPF, and TPS on LLaDA-MoE-TD. With the introduction of Trajectory Distillation, the TPS for various benchmarks has significantly improved. The average TPS exceeds that of vLLM by more than threefold.

Config.	Frame.	Metric	Avg.	CRUX-O	GSM8K	HumanEval	IFEval	MBPP	LCB V6
LLaDA-MoE	-	Perf	54.83	42.38	82.41	61.59	59.33	70.02	13.27
QWen2.5-3B	vLLM	Perf	54.44	46.75	86.28	60.37	58.2	65.81	9.2
		TPF	1	1	1	1	1	1	1
		TPS	277.45	289.53	294.15	294.05	296.7	290.15	200.12
With KV Cache	dInfer	Perf	**52.72**	40.12	79.15	63.41	56.19	65.11	12.33
		TPF	**5.67**	6.06	6.12	7.10	2.98	6.61	5.18
		TPS	**847.22**	976.66	1,011.22	1,125.67	496.92	906.98	562.87

Overview

- Intro & Motivation
- Framework Architecture
    - Design and optimizations of each module
    - Other optimizations
- Evaluation & Speed Results
- Optimizations after Technical Report
- Conclusion & Future Directions

Optimizations after Technical Report

After the technical report, we are continuously improving dInfer. We have extended dInfer to fully support and optimize newer Block Diffusion models, specifically the LLaDA2-Mini(16BA1B) and LLaDA2-Flash (100BA6B) models.

We use multiple methods to improve inference speed.

- Use cross-block cache update strategy to save forward steps
- Switch to SGLang backend and apply other optimizations to reduce forward cost
- Adopt dParallel[1] post-training method to further improve TPF

Based on these optimizations we now achieve **>1000 TPS** even on larger model sizes.

[1] Chen, Zigeng, Gongfan Fang, Xinyin Ma, Ruonan Yu, and Xinchao Wang. "dParallel: Learnable Parallel Decoding for dLLMs." *arXiv preprint arXiv:2509.26488* (2025).

Conclusion & Future Directions

DInfer successfully solved some core pain points of dLLM’s inference speed. It proves that non-autoregressive models can be a superior solution in some scenes.

Our GitHub repo: *https://github.com/inclusionAI/dInfer*

Future Directions

- Batch inference
- Decoding strategy
- ……

Thank You & Q&A