Beyond Next-Token Prediction:
A Performance Characterization of
Diffusion versus Autoregressive Language Models

Minseo Kim[1]   **Coleman Hooper**[2]   **Aditya Tomar**[2]   **Chenfeng Xu**[2,5]
Mehrdad Farajtabar[6]   **Michael W. Mahoney**[2,3,4]   **Kurt Keutzer**[2]   **Amir Gholami**[2,3]

[1]Seoul National University   [2]University of California, Berkeley   [3]ICSI   [4]LBNL
[5]University of Texas at Austin   [6]Apple*

Abstract

Large Language Models (LLMs) have achieved state-of-the-art performance on a broad range of Natural Language Processing (NLP) tasks, including document processing and code generation. Autoregressive Language Models (ARMs), which generate tokens sequentially conditioned on all previous tokens, have been the predominant paradigm for LLMs. While these models have achieved high accuracy across a range of downstream tasks, they exhibit low arithmetic intensity due to the inherent sequential dependency in next-token prediction. Recently, Diffusion Language Models (DLMs) have emerged as a promising alternative architecture. DLMs generate output tokens *in parallel*, mitigating the limitations of sequential decoding. However, the performance implications of DLMs relative to commonly deployed ARMs are not fully understood. In this work, we present a comprehensive study of the performance characteristics of ARMs and DLMs, combining theoretical analysis with empirical profiling to characterize the trade-offs between these approaches. We show that although DLMs can achieve higher arithmetic intensity than ARMs by leveraging parallelism across token positions, they fail to scale effectively with longer contexts. We then explore block-wise decoding for DLMs, which decouples arithmetic intensity from sequence length and enables better scaling to long contexts (similar to ARMs). We also examine batched inference and find that ARMs exhibit superior throughput as they benefit more from parallelism across sequences in the batch. Finally, we highlight opportunities for accelerating DLM inference, emphasizing that reducing the number of sampling steps is key for open-source DLMs to achieve lower latency relative to ARMs.

1   Introduction

Large Language Models (LLMs) have ushered in a new era for machine learning and artificial intelligence, demonstrating broad applicability across instruction following, code generation, summarization, and multi-turn dialogue [24]. Modern LLMs are predominantly Autoregressive Language Models (ARMs): decoder-only Transformers trained with a next-token prediction objective under a causal attention mask [8, 35]. Although the training process is highly parallelizable thanks to teacher forcing, this objective imposes a strict sequential dependency at inference time. To mitigate this, various techniques have emerged, with speculative decoding being a prominent example that uses a smaller draft model to generate token bursts for parallel verification [2, 18, 20]. However, the drafting path remains sequential and offers limited speedups.

---

*advisory role

As a promising alternative, Diffusion Language Models (DLMs) have recently gained attention for their new possibilities in text generation. Originating from the broader success of diffusion models in modalities like image, audio, and video synthesis [4, 9, 13, 28], DLMs operate by iteratively denoising a sequence of random noise into coherent text [11, 21, 29]. This process updates all token positions *in parallel* at each step, thereby breaking the token-level sequential dependency inherent to autoregressive decoding, and showing promise as an architecture that better exploits parallelism along the sequence dimension. Closed-source DLMs have publicly emphasized substantial speedups (e.g., Google's experimental Gemini Diffusion [12] and Inception Labs' Mercury [16] claim up to *10×* faster and cheaper generation than strong autoregressive baselines), whereas current open-source DLMs often remain substantially slower than comparable ARMs in practice (see Section 3.3).

In this work, we present a systematic performance characterization of diffusion-based text generation, relative to standard autoregression, to identify directions for closing the gap with closed-source DLMs. Our contributions are threefold:

- **Unified analysis.** (Sections 3.2 and 3.3) We present profiling data along with roofline modeling analysis to characterize ARM prefill/decode and DLM variants across prompt and output lengths. Our analysis shows that DLMs have greater arithmetic intensity (AI) than ARMs during decoding, but scale poorly to long contexts due to their need to re-process the full sequence at each sampling step.

- **Block-wise decoding and batching.** (Sections 3.4 and 3.5) We demonstrate how block-wise DLMs reduce AI relative to naive DLM decoding, while maintaining higher AI than ARMs and scaling well to long context lengths. In contrast, for batched serving, ARMs exhibit superior throughput, as they better exploit sequence-level parallelism compared to DLMs (with or without block-wise decoding).

- **Directions to close the gap.** (Section 4) We surface concrete strategies to narrow the current gap with closed-source DLMs, highlighting how reducing the number of diffusion steps is critical for improving the latency of open-source DLMs relative to open-source ARMs.

2 Background

Here, we describe several lines of related background work.

Autoregressive and Diffusion Language Models. Autoregressive language models (ARMs), the foundation of modern LLMs, are trained with a next-token prediction objective under a causal mask, restricting attention to past tokens [1, 27]. While training can be efficiently parallelized, inference is inherently sequential because each new token depends on the entire previously generated prefix [31]. This sequential dependency creates a performance bottleneck, leading to low arithmetic intensity and a decode path that is memory-bandwidth-bound [17, 19]. To avoid recomputing activations for past tokens at every step, *KV caching* has become standard. By storing Key (K) and Value (V) vectors during decoding, it largely eliminates recomputation and accelerates autoregressive inference [26]. The trade-off is a substantial increase in memory footprint and bandwidth pressure [14].

Diffusion models generate samples by initializing from Gaussian noise and iteratively applying reverse sampling steps, gradually transforming noise into a clean sample from the data distribution [13]. This family of models has been successfully adopted across modalities, including text-conditioned image generation [28], audio synthesis [4], video generation [9], and even language. In language modeling, diffusion language models (DLMs) generate over a fixed-length sequence via iterative refinement, updating all token positions in parallel at each step [11, 21, 29]. Architecturally, DLMs typically adopt Transformer architectures with *bidirectional* (non-causal) attention, allowing each position to attend to the full context [23]. Early work focused on continuous DLMs that operate in embedding space [10, 11, 29]; more recently, masked diffusion models (MDMs), a class of discrete DLMs that operate directly on categorical token spaces, have emerged as competitive alternatives to ARMs [23, 34]. A key systems challenge for DLMs is that bidirectional attention is incompatible with standard KV caching, causing latency to increase substantially as context length grows (see Section 3.3). To mitigate this, *block-wise decoding* divides the sequence into blocks. Decoding proceeds in an autoregressive manner across blocks, while positions within each block are updated in parallel, enabling approximate KV caching of the prompt and non-active blocks. [15, 22, 33]. A separate line of work, speculative diffusion decoding, keeps the main model autoregressive while employing a diffusion model as the drafter [6].

Table 1: Asymptotic FLOPs, memory operations (MOPs), and arithmetic intensity (AI) for ARM prefill/decode and DLM variants. Here $L = L_p + L_g$ (prompt + generated length), B is batch size, d is hidden dimension, K is the number of diffusion steps, G is the block size, and $N = \lceil L_g/G \rceil$ is the number of blocks. The first and second terms in each entry correspond to linear/attention operations for FLOPs, and weight/activation loads for MOPs, respectively.

	ARM Prefill	ARM Decode	Block-wise DLM (KV Cache)	Naive DLM (no KV Cache)
FLOPs	$\mathcal{O}(BL_pd^2)$ $+\,\mathcal{O}(BL_p^2d)$	$\mathcal{O}(BL_gd^2)$ $+\,\mathcal{O}(BL_gL_pd)$	$\mathcal{O}(KBGd^2)$ $+\mathcal{O}(KBGLd)$	$\mathcal{O}(KBLd^2)$ $+\,\mathcal{O}(KBL^2d)$
MOPs	$\mathcal{O}(d^2)$ $+\,\mathcal{O}(BL_pd)$	$\mathcal{O}(L_gd^2)$ $+\,\mathcal{O}(BL_gL_pd)$	$\mathcal{O}(Kd^2)$ $+\,\mathcal{O}(KBLd)$	$\mathcal{O}(Kd^2)$ $+\,\mathcal{O}(KBLd)$
AI	$\approx \begin{cases} \mathcal{O}(BL_p), & L_p \ll d \\ \mathcal{O}(L_p), & L_p \gg d \end{cases}$	$\approx \begin{cases} \mathcal{O}(B), & L_p \ll d \\ \mathcal{O}(1), & L_p \gg d \end{cases}$	$\approx \begin{cases} \mathcal{O}(BG), & L_p \ll d \\ \mathcal{O}(G), & L_p \gg d \end{cases}$	$\approx \begin{cases} \mathcal{O}(BL), & L_p \ll d \\ \mathcal{O}(L), & L_p \gg d \end{cases}$

Arithmetic Intensity and Roofline Model. While many analyses focus solely on the raw FLOP count of an algorithm, this view can be misleading. Kernel runtime may instead be dictated by the number of memory operations required if the kernel is bottlenecked by memory traffic. Thus, considering arithmetic intensity (AI) relative to memory bandwidth is essential for understanding true system performance. AI quantifies how much computation is performed per byte moved; namely, the number of floating-point operations (FLOPs) per byte (or, approximately, per memory operation, MOP):

$$\text{Arithmetic Intensity (AI)} \; = \; \frac{\#\text{FLOPs}}{\#\text{Bytes moved}} \; \approx \; \frac{\#\text{FLOPs}}{\#\text{MOPs}}$$

Low AI indicates memory-bound behavior (limited by peak memory bandwidth), whereas high AI indicates compute-bound behavior (limited by peak FLOP/s).

One method to visualize bottlenecks across prompt lengths, KV caching, and batch sizes is the roofline model [32]. The roofline model provides a visual criterion for classifying programs as compute-bound or memory-bound via the hardware ceilings for compute and bandwidth. The ridge point is the knee where the bandwidth roof meets the compute roof, occurring at $\text{AI}_{\text{ridge}} = \frac{\text{peak compute}}{\text{peak memory bandwidth}}$ (FLOP/byte); workloads with $\text{AI} < \text{AI}_{\text{ridge}}$ are memory-bound, and those with $\text{AI} \geq \text{AI}_{\text{ridge}}$ are compute-bound.

3 Workload Analysis

In this section, we present our main results.

3.1 Experimental Setup

Models. We study the LLaMA-3-8B-Instruct model [8] (an ARM) and the LLaDA-8B-Instruct model [23] (a DLM) as representative baselines (as they have comparable model size and comparable downstream task performance as reported in [23]). For block-wise DLM experiments with approximate KV caching (outlined in Section 3.4), we adopt the LLaDA variant (dual cache) from Fast-dLLM [33], as the original LLaDA does not support KV caching.

Methodology. We extend prior analytical modeling frameworks [17, 30] to (i) decompose ARM inference into prefill and decode, and (ii) model DLM inference as multi-step refinement with optional block-wise KV reuse.

Hardware. We perform measurements using a single NVIDIA RTX A6000 (48GB) with FP16 inference under PyTorch/CUDA (unless otherwise stated). Roofline parameters follow the vendor specifications.

3.2 Asymptotic Analysis of Arithmetic Intensity in ARM and DLM

Table 1 shows asymptotic analysis of arithmetic intensity (AI) for ARMs and DLMs.

ARMs. Prefill shifts from *linear-projection*-dominated AI ($\mathcal{O}(BL_p)$) when $L_p \ll d$ to *attention*-dominated AI ($\mathcal{O}(L_p)$) when $L_p \gg d$. During the subsequent decode phase, each step involves small
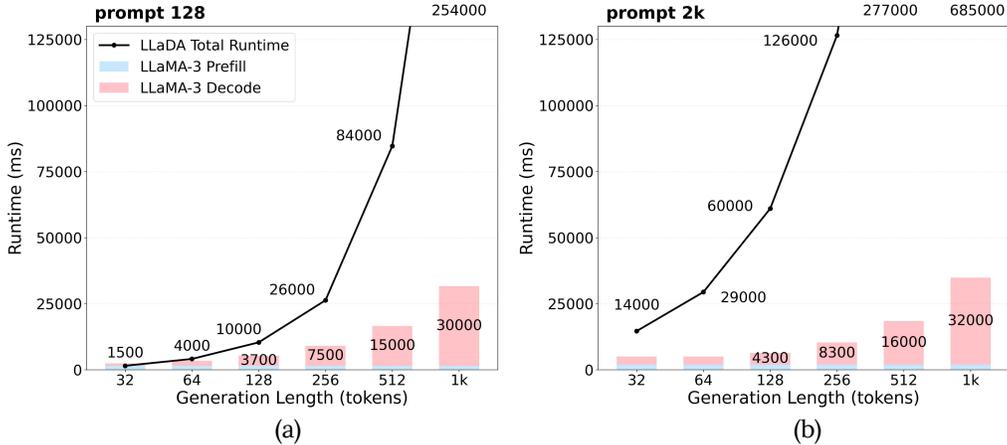
Figure 1: **Runtime comparison of ARM and DLM.** (a) End-to-end runtime vs. generation length L_g for a short context ($L_p = 128$); bars show ARM prefill/decode and the line shows naive DLM total. (b) Same as (a) but for a long context ($L_p = 2$k).
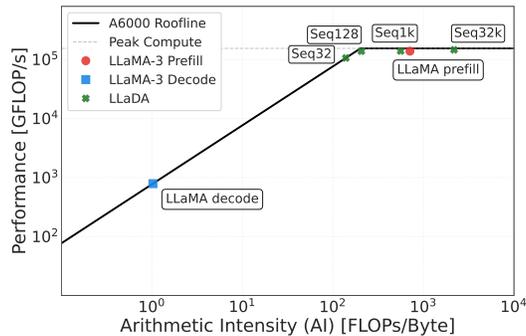

Figure 2: **Roofline comparison of ARM and DLM.** Roofline placement across sequence lengths for ARM prefill, ARM decode, and naive DLM, simulated for an NVIDIA RTX A6000 with a peak performance of 154.8 TFLOP/s and a ridge point at an AI of 201.6 FLOP/Byte.

matrix multiplications (*linear projections*) and retrieval of the growing KV cache for *attention*. As a result, the AI is initially $\mathcal{O}(B)$ when $L_p \ll d$, but it drops to $\mathcal{O}(1)$ once $L_p \gg d$, since the memory load from attention begins to dominate the fixed per-token compute; see also [30].

Block-DLMs. In a block-wise DLM with block size G, each refinement step updates a block of G tokens in parallel. The parallel computation within its linear layers is analogous to an ARM prefill operation on a sequence of length G. However, the attention mechanism is fundamentally distinct, as these G tokens attend to the KV cache representing the entire sequence L. The cost of initializing and periodically updating this cache via a full-attention pass is omitted, as it is a less dominant term compared to the cumulative cost of the K refinement steps.

Naive DLMs. In contrast, a naive (fully bidirectional) DLM processes the full sequence each step; the scale factor is the total length $L = L_p + L_g$, giving AI $\approx \mathcal{O}(BL)$ for $L_p \ll d$ and $\mathcal{O}(L)$ for $L_p \gg d$. Notably, when KV caching is absent, each sampling step recomputes the full sequence, akin to ARM prefill, and is therefore typically compute-bound.

3.3 Runtime and Roofline Analysis of ARM and DLM

We study end-to-end runtime for ARMs and DLMs as generation length L_g varies under two prompt lengths, L_p=128 (short) and L_p=2k (long). We use batch size B=1, do *not* use block-wise decoding and approximate caching, and set the number of diffusion steps to match the generation length, K=L_g, a common convention in diffusion language models. The roofline analysis assumes a peak FP16 performance of 154.8 TFLOP/s for the A6000 GPU (see Appendix A for details) and models LLaMA-3 with a 2k-token sequence length.
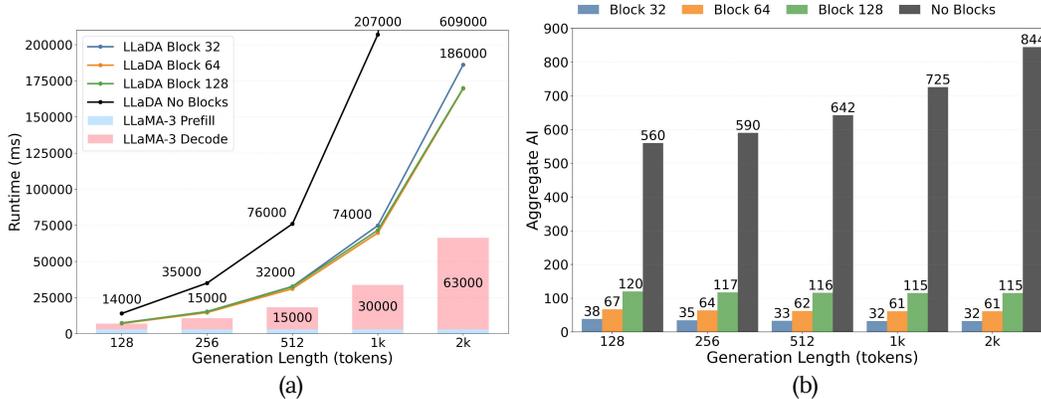
Figure 3: **Runtime and AI for naive and block-wise DLM with KV caching.** (a) End-to-end runtime vs. generation length L_g at a fixed prompt $L_p = 1k$; bars show ARM (for reference); lines show the naive DLM ("No Blocks") and block-wise DLM for block sizes $G \in \{32, 64, 128\}$ with diffusion steps $K = L_g$. (b) Aggregate AI vs. L_g for the naive and block-wise DLM configurations.

Figure 1 (a) shows latency for the short prompt. The ARM runtime decomposes into prefill (blue), which is flat in L_g, and decode (red), which grows modestly because KV caching prevents recomputation. The DLM is quite fast for short prompts and small L_g, thanks to parallel token updates, but its latency rises sharply as L_g grows. With a long prompt in Figure 1 (b), DLM latency is higher even at small L_g, since each sampling step recomputes the entire sequence, whereas ARM decode reuses the cached tokens from prior steps.

Figure 2 places these workloads on the roofline. ARM prefill (red) lies to the right of the ridge point and is *compute-bound*, reflecting large GEMMs over the full prompt. ARM decode (blue) sits on the bandwidth roof and is *memory-bound*, dominated by KV-cache traffic. For naive DLM (green), short sequences (e.g., up to 128 tokens) lie near the bandwidth roof, while longer sequences cross the ridge point and become *compute-bound* as full-sequence denoising dominates.

3.4 Implications of Block-wise Decoding with KV Cache

Block-wise decoding combines the strengths of ARMs and DLMs: it reuses the KV cache for the prompt and non-active blocks (ARM-like), while updating all tokens within the active block in parallel (DLM-like). The diffusion steps K are typically distributed evenly across blocks. We conduct a profiled analysis, where we fix the prompt length to $L_p = 1k$, consider block sizes $G \in \{32, 64, 128\}$, and set the number of diffusion steps equal to the generation length ($K = L_g$). Figure 3 (a) compares end-to-end latency versus generation length L_g, demonstrating how block-wise decoding with KV caching reduces latency by roughly $2\times$ to $3\times$, compared to naive DLM (labeled "No Blocks"). Moreover, Figure 3 (b) shows that this approach not only reduces the AI but also makes it invariant to the generation length, depending only on the block size G. This reduction in AI translates into lower latency in compute-bound settings (e.g., naive DLM inference in Fig. 3 (b)), and block-wise decoding reduces the required FLOPs relative to naive DLM (as highlighted in Table 1). Furthermore, block-wise decoding preserves (and can even improve) task accuracy despite these efficiency gains; we report accuracy results in Appendix B.

Note that even with KV caching, the DLM remains slower than the ARM at matched prompt and output lengths when K scales with L_g. For instance, the widely used configuration of $K = L_g$ [23, 34] implies that the model denoises and finalizes only one token per step on average, which is hugely inefficient. This underscores the importance of *reducing diffusion steps* via efficient multi-token denoising, a point we revisit in Section 4.

3.5 Implications of Inference with Different Batch Sizes

Real-world applications often serve multiple users concurrently, making scalability with respect to the batch size B a critical metric. We therefore measure throughput (tokens/s) as a function of B, fixing the generation length at $L_g = 128$ and evaluating two prompt categories: short ($L_p = 128$) and long ($L_p = 2k$). We set the block size to $G = 32$ and use $K = L_g$ for the block-wise DLM. All
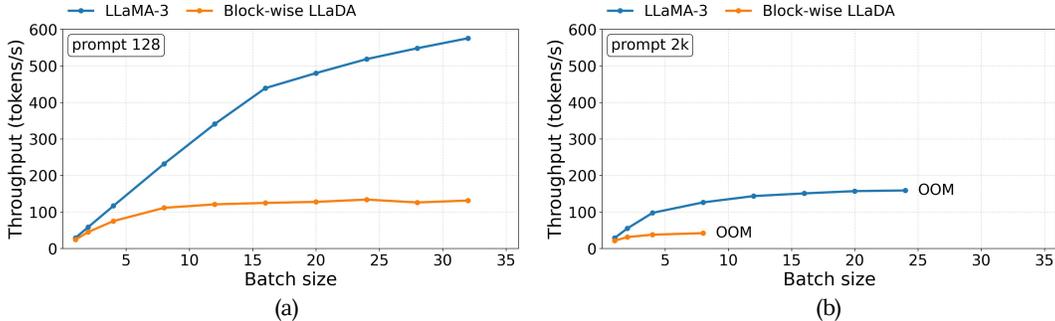
Figure 4: **Throughput vs. batch size for ARM and block-wise DLM.** Throughput (tokens/s) as a function of batch size B for (a) short prompt ($L_p = 128$) and (b) long prompt ($L_p = 2\text{k}$). The generation length is fixed at $L_g = 128$, and OOM denotes out-of-memory.

experiments in this section are run on a single NVIDIA A100 (80 GB) GPU to accommodate the memory demands of batched inference. We omit the naive DLM, as prior work shows it does not benefit from batching due to its consistently *compute-bound* bidirectional full attention [33].

Figure 4 indicates that for both ARMs and DLMs, throughput scaling with batch size diminishes as prompt length L_p increases. As L_p grows, the memory-intensive task of reading a large *KV cache* becomes the dominant part of the workload, limiting the potential throughput gains from batching. Despite this shared trend, ARMs consistently exhibit better batching scalability than DLMs across all prompt lengths. For example, at $L_p = 128$, ARM throughput scales effectively up to a batch size of 16, whereas DLM throughput already plateaus around 8 and remains much lower in absolute terms. This disparity arises because the ARM's lightweight, memory-bound decoding effectively leverages batch parallelism, while the DLM's throughput is constrained by two factors: its compute-heavy block generation leads to an earlier performance plateau, and the high cumulative cost of iterative refinement results in a lower overall peak.

The plots also reveal memory limitations. For the block-wise DLM, out-of-memory (OOM) errors appear at smaller B as L_p grows. Even though decoding proceeds block by block, attention within each block is bidirectional and requires the KV cache for the full sequence length $L = L_p + L_g$ to be resident. Consequently, peak memory scales with $B \times L$, causing longer prompts to hit OOM at smaller batch sizes.

4 Discussion and Future Work

Autoregressive language models (ARMs) have driven major progress in NLP, but their next-token objective imposes sequential dependencies that increase latency and can degrade quality by limiting bidirectional context. Diffusion language models (DLMs) are a promising alternative, as they have potential to decode many tokens in parallel, while conditioning on the full context. We have presented a comprehensive analysis of the performance trade-offs between open-source ARMs and DLMs.

Our analysis shows that a naive DLM exhibits substantial latency overheads relative to ARMs as the context length increases. To ameliorate this, block-wise decoding with approximate KV caching offers an effective hybrid approach. This method achieves increased arithmetic intensity (AI) by exploiting intra-block parallelism, while still scaling well to long contexts, similar to ARMs. However, in batched serving, the block-wise DLM's parallel generation saturates the GPU at smaller batch sizes than ARMs. Combined with the DLM's high cost of iterative refinement, this results in ARMs providing distinct throughput advantages in the large-batch regime. Recent closed-source systems [12, 16, 25] report striking latency gains at comparable quality, even faster than ARMs. Based on our analysis, we therefore conclude that two core bottlenecks are responsible for the slowdown of current open-source DLMs: poor scaling with context length and high refinement costs.

For the scaling problem, Section 3.4 confirms that block-wise decoding with approximate KV caching [15, 22, 33] is an indispensable strategy. Yet, to truly rival ARM speed, the most critical path forward involves reducing the number of sampling steps K. In common configurations, K scales with the generation length L_g, leading to high per-output costs and compute saturation, as highlighted in Section 3.5. Recent approaches attempt to cut steps without quality loss, such as

multi-token finalization with autoregressive guidance [15], confidence-based early finalization [33], and distillation into few-step or even one-step student models [5]. Reducing sampling steps is particularly promising in *small-batch* scenarios, where a DLM's throughput could surpass an ARM's, as mentioned in Appendix C. Finally, orthogonal system-level optimizations can offer additional speedups. These include compute-focused methods like low-precision execution as well as sparse attention methods that utilize only the most salient cache entries.

Acknowledgements

References

[1] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D. Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, volume 33, pages 1877–1901, 2020.

[2] Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023. URL https://arxiv.org/abs/2302.01318.

[3] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Giri Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Michael Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Phil Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Guss, Alex Nichol, Igor Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Anna Carr, Jan Leike, Joshua Achiam, Vedant Misra, Kamil Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Ryan Kaplan, Sam McCandlish, Dario Amodei, and Ilya Sutskever. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021. URL https://arxiv.org/abs/2107.03374.

[4] Nanxin Chen, Yu Zhang, Heiga Zen, Ron J. Weiss, Mohammad Norouzi, and William Chan. Wavegrad: Estimating gradients for waveform generation. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2009.00713.

[5] Tianqi Chen, Shujian Zhang, and Mingyuan Zhou. DLM-One: Diffusion language models for one-step sequence generation. *arXiv preprint arXiv:2506.00290*, 2025. URL https://arxiv.org/abs/2506.00290.

[6] Jacob K. Christopher, Brian R. Bartoldson, Tal Ben-Nun, Michael Cardei, Bhavya Kailkhura, and Ferdinando Fioretto. Speculative diffusion decoding: Accelerating language generation through diffusion. In *Proceedings of the 2025 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 12042–12059, Albuquerque, New Mexico, 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.naacl-long.601. URL https://aclanthology.org/2025.naacl-long.601/.

[7] Karl Cobbe, Vineet Kosaraju, Mohammad Bavarian, Mark Chen, Heewoo Jun, Lukasz Kaiser, Matthias Plappert, Jerry Tworek, Jacob Hilton, Reiichiro Nakano, Christopher Hesse, and John Schulman. Training verifiers to solve math word problems. In *Proceedings of the 2021*

Conference on Empirical Methods in Natural Language Processing (EMNLP), 2021. URL `https://arxiv.org/abs/2110.14168`.

[8] Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024. URL `https://arxiv.org/abs/2407.21783`.

[9] Patrick Esser, Johnathan Chiu, Parmida Atighehchian, Jonathan Granskog, and Anastasis Germanidis. Structure and content-guided video synthesis with diffusion models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, 2023. URL `https://openaccess.thecvf.com/content/ICCV2023/html/Esser_Structure_and_Content-Guided_Video_Synthesis_with_Diffusion_Models_ICCV_2023_paper.html`.

[10] Zhujin Gao, Junliang Guo, Xu Tan, Yongxin Zhu, Fang Zhang, Jiang Bian, and Linli Xu. Empowering diffusion models on the embedding space for text generation. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics (NAACL)*, 2024.

[11] Shansan Gong, Mukai Li, Jiangtao Feng, Zhiyong Wu, and Lingpeng Kong. Diffuseq: Sequence to sequence text generation with diffusion models. In *International Conference on Learning Representations*, 2023. URL `https://openreview.net/forum?id=D8DUDJQKZg`.

[12] Google DeepMind. Gemini Diffusion. `https://deepmind.google/models/gemini-diffusion/`, 2025.

[13] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. In *Advances in Neural Information Processing Systems*, 2020.

[14] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W Mahoney, Yakun S Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. *Advances in Neural Information Processing Systems*, 37:1270–1303, 2024.

[15] Zhanqiu Hu, Jian Meng, Yash Akhauri, Mohamed S Abdelfattah, Jae-sun Seo, Zhiru Zhang, and Udit Gupta. Accelerating diffusion language model inference via efficient kv caching and guided diffusion. *arXiv preprint arXiv:2505.21467*, 2025. URL `https://arxiv.org/abs/2505.21467`.

[16] Samar Khanna, Siddhant Kharbanda, Shufan Li, Harshit Varma, Eric Wang, Sawyer Birnbaum, Ziyang Luo, Yanis Miraoui, Akash Palrecha, Stefano Ermon, Aditya Grover, and Volodymyr Kuleshov. Mercury: Ultra-fast language models based on diffusion. *arXiv preprint arXiv:2506.17298*, 2025. URL `https://arxiv.org/abs/2506.17298`.

[17] Sehoon Kim, Coleman Hooper, Thanakul Wattanawong, Minwoo Kang, Ruohan Yan, Hasan Genc, Grace Dinh, Qijing Huang, Kurt Keutzer, Michael W. Mahoney, et al. Full stack optimization of transformer inference: A survey. *arXiv preprint arXiv:2302.14017*, 2023. URL `https://arxiv.org/abs/2302.14017`.

[18] Sehoon Kim, Karttikeya Mangalam, Suhong Moon, Jitendra Malik, Michael W. Mahoney, Amir Gholami, and Kurt Keutzer. Speculative decoding with big little decoder. In *Advances in Neural Information Processing Systems*, volume 36, pages 39236–39256, 2023.

[19] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W. Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization. In *Proceedings of the 41st International Conference on Machine Learning*, volume 235, pages 23901–23923. PMLR, 2024.

[20] Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

[21] Xiang Lisa Li, John Thickstun, Ishaan Gulrajani, Percy S. Liang, and Tatsunori B. Hashimoto. Diffusion-LM improves controllable text generation. In *Advances in Neural Information Processing Systems*, volume 35, pages 4328–4343, 2022.

[22] Xinyin Ma, Runpeng Yu, Gongfan Fang, and Xinchao Wang. dkv-cache: The cache for diffusion language models. *arXiv preprint arXiv:2505.15781*, 2025. URL `https://arxiv.org/abs/2505.15781`.

[23] Shen Nie, Fengqi Zhu, Zebin You, Xiaolu Zhang, Jingyang Ou, Jun Hu, Jun Zhou, Yankai Lin, Ji-Rong Wen, and Chongxuan Li. Large language diffusion models. *arXiv preprint arXiv:2502.09992*, 2025. URL `https://arxiv.org/abs/2502.09992`.

[24] OpenAI. GPT-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023. URL `https://arxiv.org/abs/2303.08774`.

[25] Akash Palrecha. Inception partners with amazon to provide mercury diffusion llms on bedrock and sagemaker jumpstart. https://www.inceptionlabs.ai/inception-amazon-bedrock, August 2025.

[26] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Anselm Levskaya, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. Efficiently scaling transformer inference. *arXiv preprint arXiv:2211.05102*, 2022. URL `https://arxiv.org/abs/2211.05102`.

[27] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. OpenAI Blog, 2019. URL `https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf`.

[28] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with CLIP latents. *arXiv preprint arXiv:2204.06125*, 2022. URL `https://arxiv.org/abs/2204.06125`.

[29] Robin Strudel, Corentin Tallec, Florent Altché, Yilun Du, Yaroslav Ganin, Arthur Mensch, Will Sussman Grathwohl, Nikolay Savinov, Sander Dieleman, Laurent Sifre, and Rémi Leblond. Self-conditioned embedding diffusion for text generation. *arXiv preprint arXiv:2211.04236*, 2022. URL `https://arxiv.org/abs/2211.04236`.

[30] Rishabh Tiwari, Haocheng Xi, Aditya Tomar, Coleman Hooper, Sehoon Kim, Maxwell Horton, Mahyar Najibi, Michael W. Mahoney, Kurt Keutzer, and Amir Gholami. Quantspec: Self-speculative decoding with hierarchical quantized kv cache. *arXiv preprint arXiv:2502.10424*, 2025. URL `https://arxiv.org/abs/2502.10424`.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[32] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[33] Chengyue Wu, Hao Zhang, Shuchen Xue, Zhijian Liu, Shizhe Diao, Ligeng Zhu, Ping Luo, Song Han, and Enze Xie. Fast-dllm: Training-free acceleration of diffusion llm by enabling kv cache and parallel decoding. *arXiv preprint arXiv:2505.22618*, 2025. URL `https://arxiv.org/abs/2505.22618`.

[34] Jiacheng Ye, Zhihui Xie, Lin Zheng, Jiahui Gao, Zirui Wu, Xin Jiang, Zhenguo Li, and Lingpeng Kong. Dream 7b. `https://hkunlp.github.io/blog/2025/dream`, 2025. Project blog.

[35] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, et al. OPT: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022. URL `https://arxiv.org/abs/2205.01068`.

Table 2: DLM performance on GSM8K and HumanEval with and without block-wise decoding. We report results for block sizes $G \in \{64, 32, 16, 8\}$ and the non-block-wise baseline (None). The generation length is fixed to $L_g = 128$. GSM8K is reported as exact-match (EM) accuracy, and HumanEval is reported as pass@1.

Block Size G	GSM8K (4-shot, EM)	HumanEval (0-shot, pass@1)
None	63.2	13.4
64	73.8	34.1
32	73.6	35.4
16	75.7	38.4
8	75.0	38.4

A   RTX A6000 Roofline Parameters

We derive the roofline parameters for the NVIDIA RTX A6000 (GA102). We assume *dense* FP16 Tensor Core computation (without 2:4 structured sparsity) and use the GPU boost clock. Hardware specifications (SM count, Tensor Cores per SM, boost clock, memory bandwidth, and FP16 per-Tensor Core throughput) are taken from the GA102 architecture whitepaper. [2]

Peak FP16 Performance.

$$\text{Peak FP16 (dense)} = 84\,\text{SM} \times 4\,\text{Tensor Core/SM} \times 128\,\text{FMA/(cycle} \cdot \text{Tensor Core)}$$
$$\times\, 1.8\,\text{GHz} \times 2\,\text{FLOP/FMA} \approx 154.8\,\text{TFLOP/s}.$$

Ridge Point.   With memory bandwidth $768\,\text{GB/s}$, the ridge arithmetic intensity is

$$\text{AI}_{\text{ridge}} = \frac{154.8\,\text{TFLOP/s}}{768\,\text{GB/s}} \approx 201.6\,\text{FLOP/Byte}.$$

B   Impact of Block-wise Decoding on Generation Quality

As discussed in Section 3.4, block-wise decoding combined with KV caching provides systematic efficiency benefits by reducing redundant computation, thereby lowering latency under compute-bound DLM inference. Following the evaluation protocol of prior work [33], we evaluate block-wise DLMs on representative benchmarks, including the GSM8K math benchmark [7] and the HumanEval coding benchmark [3].

As shown in Table 2, block-wise decoding consistently improves accuracy across both math and coding tasks compared to the naive DLM baseline (None). While the block size G is a hyperparameter whose optimal value may vary across tasks, all block-wise configurations outperform the naive DLM in our experiments.

Two factors likely contribute to this behavior. First, block-wise decoding mitigates the excessive generation of end-of-text (|EOT|) tokens. In naive DLMs, generation can collapse because the model can "cheat" by repeatedly filling |EOT| from later positions, since this is easy, leading to degenerate outputs. Block-wise decoding helps alleviate this issue by enforcing parallel generation within the active block. Second, block-wise decoding introduces a weak left-to-right inductive bias at the block level. This ordering constraint can be beneficial for reasoning tasks, where a left-to-right structure naturally aligns with step-by-step reasoning processes.

C   Implications of Reduction in Sampling Steps

As noted in our discussion, reducing the number of sampling steps K is a critical path toward making DLMs competitive. To quantify this potential, we conduct a batch scalability experiment where we mechanically reduce the number of sampling steps K from our default block-wise DLM configuration

---

[2]NVIDIA Ampere GA102 Architecture Whitepaper: `https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf`.
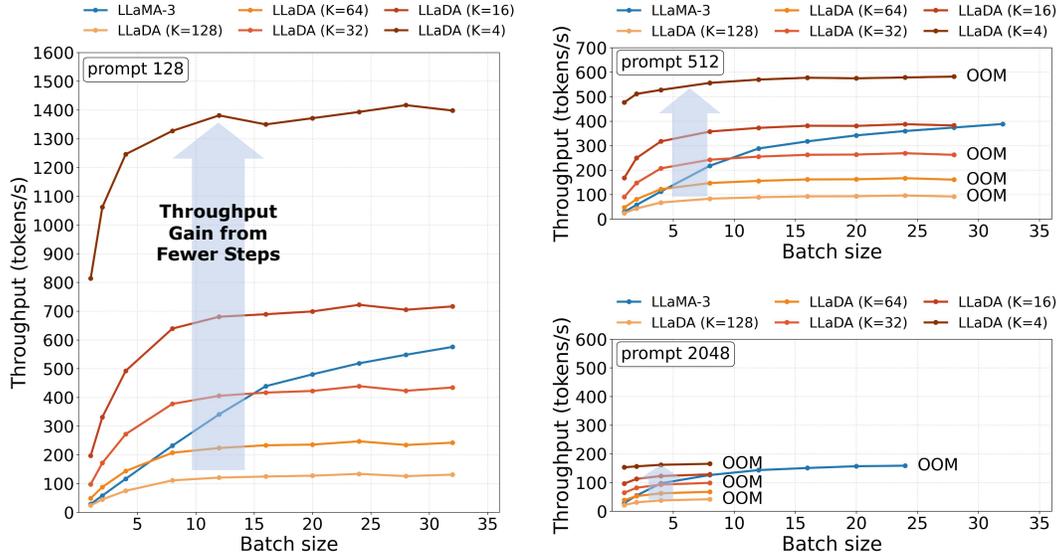
Figure 5: **Throughput vs. batch size for ARM and block-wise DLM with a varying number of sampling steps.** The plots show the throughput of ARM and block-wise DLM for prompt length $L_p \in \{128, 512, 2048\}$ with generation length $L_g = 128$ and block size $G = 32$. For LLaDA, K denotes the number of sampling steps, which is varied from the baseline of 128 down to 4.

Table 3: Block-wise DLM performance on GSM8K and HumanEval benchmarks across different sampling steps K (from 128 down to 4). Generation length is fixed at $L_g = 128$ and block size $G = 32$. GSM8K is reported as exact-match (EM) accuracy, and HumanEval is reported as pass@1.

Sampling Steps K	GSM8K (4-shot, EM)	HumanEval (0-shot, pass@1)
128	73.6	35.4
64	70.6	32.3
32	61.7	17.7
16	29.7	9.1
4	6.1	3.7

(block size $G = 32$). Specifically, we reduce K from its baseline of 128 (where $K = L_g$) down to 64, 32, 16, and 4. For example, when $K = 4$, the generation length $L_g = 128$ is divided into 4 blocks of 32 tokens each, and each block is denoised in a single refinement step.

In addition to throughput analysis, we also evaluate accuracy degradation when reducing steps without any further techniques. We again follow the evaluation setup of prior work [33] and test on representative benchmarks: the GSM8K math benchmark [7] and the HumanEval coding benchmark [3]. As shown in Table 3, naively reducing the number of steps results in severe accuracy drops, underscoring the necessity of advanced methods such as confidence-based multi-token finalization, autoregressive guidance, or distillation.

Figure 5 shows the results, with memory footprint remaining unchanged since only the number of iterations is reduced. For instance, with a 128-token prompt, we observe that although performance still plateaus around a batch size of 8, the peak throughput scales nearly proportionally as K decreases. These findings reinforce our earlier hypothesis in Section 4: in the *small-batch* regime where DLMs can still benefit from batching, reducing the number of sampling steps (especially via methods that finalize multiple tokens per step) has the potential to enable DLMs to surpass ARMs in throughput, provided quality degradation is mitigated by advanced techniques.