BLOCK PARALLELISM: EFFICIENT LONG-CONTEXT DIFFUSION LANGUAGE MODEL TRAINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Block diffusion language models (BDLMs) combine autoregressive dependencies across blocks with parallel denoising within blocks, but long-context training is constrained by distributed attention communication and activation memory. Conventional context parallelism (CP) shards the combined clean-plus-corrupted sequence by position, communicating shared clean K/Vs together with block-specific corrupted K/Vs and their gradients. Our key observation is that the BDLM objective separates over target blocks and that diffusion-based speculative drafter losses have the same decomposition. We introduce block parallelism (BP), a new distributed parallelism dimension that assigns each corrupted-block computation to one rank. To scale BP to long contexts, we introduce context-sharded block parallelism (CSBP), which also shards the shared clean sequence across those ranks. CSBP keeps corrupted K/Vs and their gradients local, avoids replicated clean prefixes, and preserves BDLM training semantics. Applied to speculative drafters, it also avoids replicated block-local computation. On 16 H200 GPUs at 256K context, CSBP improves throughput over the best baseline by **1.18–1.45**× for supervised fine-tuning on 3B–26B-A4B models and **1.27–1.33**× for converting 27B autoregressive models to BDLMs, while matching or reducing peak HBM. For DiffusionGemma 26B-A4B, the fine-tuning speedup over the best baseline grows with context length, reaching **1.61**× at 512K. On eight H100 GPUs, CSBP accelerates Qwen3.8-27B DFlash2 drafter training by **2.48**× at 512K and **7.59**× at 1M. In 12-hour DiffusionGemma 26B-A4B SFT, CSBP achieves higher pass rates at every trained checkpoint on SWE-bench Verified and Terminal-Bench Lite.

1 INTRODUCTION

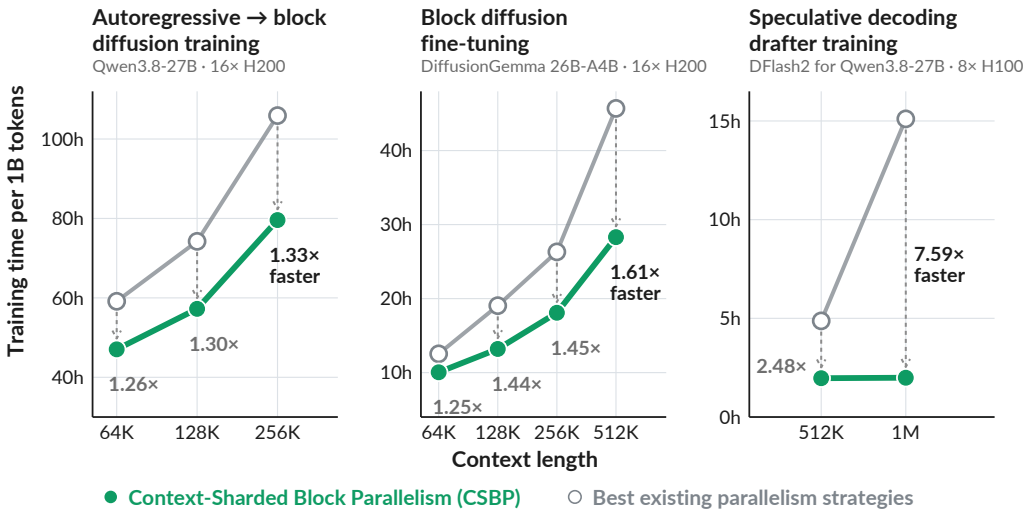

Figure 1: CSBP accelerates the frontier of long-context block diffusion training.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

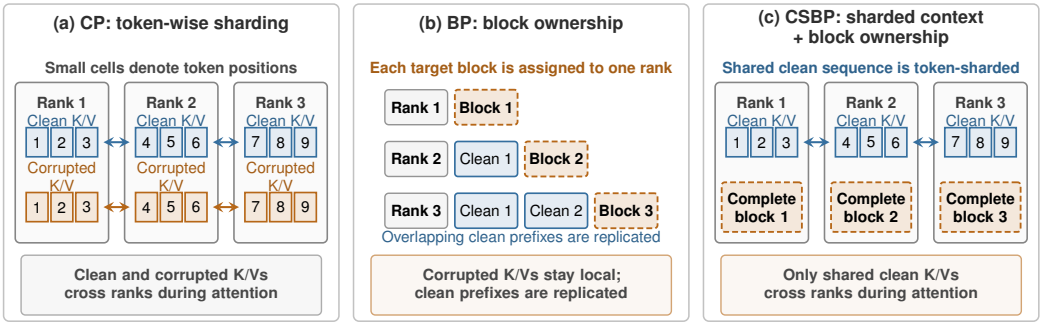

Figure 2: (a) Conventional CP shards by token position and exchanges both clean and corrupted K/Vs across ranks, despite corrupted K/Vs having no reuse across blocks. (b) BP keeps corrupted K/Vs local to the rank assigned that block, but replicates overlapping clean prefixes. (c) CSBP also shards the shared clean context, so only clean K/Vs and their gradients cross ranks during attention. Figure 3 details the resulting tensor communication.

Block diffusion language models, or BDLMs, offer a promising approach to combining strong language-model quality with efficient inference. They retain autoregressive dependencies across consecutive token blocks while generating the tokens within each block through parallel denoising. This combines the causal structure of autoregressive models with the parallelism of diffusion-based generation (Arriola et al., 2025; O'Donoghue & Flennerhag, 2026; Fu et al., 2026). Diffusion-based speculative decoding similarly uses a block-diffusion drafter to propose token blocks that an autoregressive target model verifies in parallel (Leviathan et al., 2023; Chen et al., 2026; Inco AI, 2026).

Scaling both uses of block diffusion to long contexts is important for applications such as coding agents and other long-running assistants, whose interaction histories grow throughout a task (Yang et al., 2024; Yao et al., 2023). At these lengths, BDLMs and diffusion-based speculative drafters face the same memory and attention bottlenecks. Existing long-context training systems commonly use context parallelism, which partitions sequence activations across devices so that each device stores and processes only part of the context. During attention, devices exchange information across these partitions, allowing local queries to incorporate relevant information from the full sequence (Liu et al., 2024; Jacobs et al., 2023).

Conventional context parallelism (CP) is poorly matched to block-diffusion training objectives because it partitions the combined clean-plus-corrupted representation by token position and performs distributed attention across those position shards. Clean K/Vs from earlier blocks can condition multiple later blocks, whereas corrupted K/Vs for a block are used only by queries from that block. At every layer, CP therefore exchanges both clean and corrupted K/Vs during forward and their corresponding gradients during backward. The corrupted K/Vs consume cross-rank bandwidth despite having no reuse across blocks (Figure 2(a)). For speculative drafters, CP also repeats each block-local drafter computation across context ranks.

The objective separates across target blocks. We exploit this separability by introducing block parallelism (BP), a new distributed parallelism dimension that assigns each complete corrupted-block computation to one rank. Different blocks run concurrently, while each block's corrupted Q/K/V tensors, activations, and gradients remain local (Figure 2(b)). For speculative-drafter training, the same assignment computes each draft block once rather than once per context rank.

BP alone replicates the long, overlapping clean prefixes used by different blocks. Context-sharded block parallelism (CSBP) removes this replication by combining BP with context sharding: BP assigns complete corrupted-block computations to ranks, while CP shards the shared clean sequence across those same ranks. Each clean K/V shard is reused by all assigned blocks that attend to it. The resulting gradient contributions are accumulated on the rank that stores the shard. CSBP thus distributes clean computation and storage while keeping corrupted K/Vs and their gradients local (Figure 2(c)).

We make four contributions.

- **Block parallelism.** We introduce target-block computation as a new distributed parallelism dimension for block-diffusion training objectives, including those used for BDLMs and diffusion-based speculative drafters. BP uses per-block loss separability to assign each complete corrupted-block computation, including its forward pass, loss, and backward pass, to one rank. Different blocks execute concurrently while their corrupted K/Vs and gradients remain local, eliminating block-specific cross-rank communication and replicated drafter computation.
- **Context-sharded block parallelism.** We introduce CSBP to scale block parallelism to long contexts without replicating the clean sequence. Over the same ranks, BP assigns complete corrupted-block computations while CP shards the shared clean-sequence computation. Each clean K/V shard is reused by all assigned blocks that attend to it. The resulting gradient contributions are accumulated on the rank that stores the shard. CSBP removes corrupted K/Vs from cross-rank attention communication and avoids replicating overlapping clean prefixes. We open-source our training infrastructure, which is highly optimized for training long-context diffusion language models and speculative drafters, with support for BP and CSBP.
- **Efficient long-context training.** Across six models, CSBP delivers 256K-context speedups of **1.18–1.45**$\times$ for SFT and **1.27–1.33**$\times$ for AR-to-BDLM conversion over throughput-optimized combinations of data, tensor, expert, and token-sharded context parallelism, while matching or reducing peak HBM (Table 1). The advantage grows with context length (Figure 4): at 512K, it reaches **1.19**$\times$ on NemotronDiffusion 14B and **1.61**$\times$ on DiffusionGemma 26B-A4B.
- **Faster speculative drafter training and stronger agent performance.** CSBP accelerates Qwen3.8-27B DFlash2 drafter training by **2.48**$\times$ at 512K and **7.59**$\times$ at 1M. Under matched 12-hour DiffusionGemma SFT budgets, CSBP achieves higher pass rates at every trained checkpoint on SWE-bench Verified and Terminal-Bench Lite.

2 Related Work

Long-context training parallelism. Megatron sequence parallelism reduces activation replication alongside tensor parallelism (Korthikanti et al., 2023). DeepSpeed-Ulysses redistributes sequence shards across attention heads (Jacobs et al., 2023), while Ring Attention overlaps K/V communication with attention computation (Liu et al., 2024). Striped Attention balances causal attention workloads, and LoongTrain combines head and context parallelism (Brandon et al., 2023; Gu et al., 2024).

Block diffusion and AR-to-BDLM adaptation. Block Diffusion combines autoregressive dependencies between blocks with parallel denoising within blocks, supporting variable-length generation and clean-prefix K/V caching (Arriola et al., 2025). Fast-dLLM v2 converts AR checkpoints using block-structured attention, complementary masking, and shifted prediction (Wu et al., 2025). DiffusionGemma adapts an AR model through denoising SFT followed by sampler distillation and reinforcement learning (DiffusionGemma Team et al., 2026).

Diffusion-based speculative decoding. Speculative decoding verifies draft tokens in parallel while preserving the AR target distribution (Leviathan et al., 2023). DFlash uses target-model features to draft an entire block in one pass (Chen et al., 2026). DFlash2 adds block-local dynamic convolutions and candidate-path selection to improve draft quality (Inco AI, 2026). CSBP instead accelerates block-separable training by separating shared context from block-local computation, preserving training objectives and inference protocols.

3 Background

3.1 Notation

Let a sequence $x = (x_1, \ldots, x_L)$ be divided into B contiguous blocks of $M = L/B$ tokens.

$$x = \left(x^{(1)}, \ldots, x^{(B)}\right), \qquad x^{(<b)} = \left(x^{(1)}, \ldots, x^{(b-1)}\right). \tag{1}$$

We use b to index blocks and r to index the P ranks in a parallel group. At each transformer layer (Vaswani et al., 2017), Q, K, and V denote the query, key, and value projections; semicolons inside brackets denote sequence concatenation.

3.2 Block diffusion language models

A block diffusion language model (BDLM) generates blocks from left to right while denoising the tokens within each block in parallel (Arriola et al., 2025).

$$p_\theta(x) = \prod_{b=1}^{B} p_\theta\left(x^{(b)} \mid x^{(<b)}\right), \tag{2}$$

where each conditional distribution is modeled by a diffusion process.

Training samples continuous times t_b, not discrete timestep indices: standard BDLM uses uniform stratification over $[10^{-3}, 1)$ across batch/block pairs; Fast-dLLM v2 uses unstratified $\mathcal{U}[0, 1)$ draws. Each target block forms a corrupted copy $x_{t_b}^{(b)}$ conditioned on its clean prefix $x^{(<b)}$. The training objective is

$$\mathcal{L}_{\text{BDLM}} = \sum_{b=1}^{B} \mathbb{E}_{t_b,\, x_{t_b}^{(b)}} \left[-w(t_b) \log p_\theta\left(x^{(b)} \mid x_{t_b}^{(b)}, x^{(<b)}\right) \right]. \tag{3}$$

The expectation is over t_b and $x_{t_b}^{(b)} \sim q_{t_b}(\cdot \mid x^{(b)})$, where q_{t_b} is the corruption process and $w(t_b)$ is a noise-dependent loss weight. We denote the bth summand, including token normalization, by $\mathcal{L}_b$. Training observes the complete sequence and uses the clean ground-truth prefix for every target block.

The standard BDLM training computation evaluates all B block losses together by placing a length-L corrupted copy alongside a length-L clean copy, forming a length-$2L$ representation. It contains a block-specific corrupted copy and a reusable clean copy of every token. A block-causal attention mask gives the clean tokens in block b access to the preceding clean blocks and causal access within block b. The corrupted tokens in target block b attend bidirectionally within their own corrupted block and attend to the clean prefix $x^{(<b)}$, but do not use corrupted K/Vs from other blocks. Their visible keys and values are therefore $[K_b; K_{<b}^{\text{clean}}]$ and $[V_b; V_{<b}^{\text{clean}}]$.

The corrupted activations of block b affect only $\mathcal{L}_b$, while its clean representation can be reused by multiple later block losses. Corrupted computation is therefore block-specific, while clean computation is shared across block losses. All block losses share the model parameters.

3.3 Diffusion-based speculative drafter training

Diffusion-based speculative decoding uses a lightweight block-diffusion drafter to propose future tokens for parallel verification by an autoregressive target model (Leviathan et al., 2023; Chen et al., 2026). DFlash drafter training samples a set $\mathcal{S}$ of draft blocks. In each block, the anchor token is visible and the next $M - 1$ tokens are masked and predicted in one pass. Each draft block attends bidirectionally within itself and to the shared frozen target-model features before its anchor, but not to other draft blocks.

Blockwise objectives. Both BDLM and drafter losses separate over target blocks:

$$\mathcal{L} = \sum_{b \in \mathcal{S}} \mathcal{L}_b. \tag{4}$$

For BDLMs, $\mathcal{S} = \{1, \ldots, B\}$ and $\mathcal{L}_b$ is the summand of Equation 3. For drafters, $\mathcal{S}$ contains the sampled blocks and $\mathcal{L}_b$ is the normalized loss for one block (Appendix E). We use *corrupted* for a noised BDLM block or masked draft block and *clean* for the shared clean sequence or target-model features.

3.4 Context parallelism

Context parallelism (CP) uses token position as its ownership dimension, partitioning the attention input across P ranks so each rank stores approximately a $1/P$ fraction of its token activations while

the group jointly evaluates attention (Liu et al., 2024; Jacobs et al., 2023). BDLM training supplies the length-$2L$ clean-plus-corrupted representation as this attention input, so conventional CP partitions both copies by token position.

Let Q_r, K_r, and V_r denote the query, key, and value projections initially owned by rank r, and let d_h be the attention-head dimension. Writing $K = [K_1; \ldots; K_P]$ and $V = [V_1; \ldots; V_P]$, the attention output for the queries on rank r is

$$O_r = \mathrm{softmax}\left(\frac{Q_r K^\top}{\sqrt{d_h}} + \mathcal{M}_r\right) V, \tag{5}$$

where $\mathcal{M}_r$ contains the attention-mask rows for the queries on rank r.

Causal attention produces uneven work across sequence shards because later queries see longer prefixes. Zigzag partitioning pairs early and late chunks on each rank to balance their attention work (Dubey et al., 2024), while retaining the sequence shard as the unit of CP ownership.

4 Block Parallelism

4.1 Limitations of conventional context parallelism

Conventional CP is poorly matched to block diffusion training because it partitions the entire length-$2L$ clean-plus-corrupted representation by token position and performs distributed attention across those position shards. Both clean and corrupted K/Vs therefore participate in cross-rank attention communication. The corrupted K/Vs for block b are used only by queries from that block. CP exchanges these block-specific K/Vs during forward and returns their gradients to the ranks storing the position shards during backward, adding communication at every layer for data used by only one block. For speculative drafters, CP also repeats each block-local computation across context ranks (Appendix E).

Clean K/Vs are shared across later block losses. When $\mathcal{L}_b$ uses a clean K/V shard, attention backward produces a gradient contribution for that shard. This contribution is accumulated on the rank that stores the shard, where backward continues through the clean projections and earlier layers. Conventional CP therefore communicates the required clean-prefix K/Vs together with block-specific corrupted K/Vs and their gradients at every layer. It also distributes every target block across ranks, so each rank needs the union of K/V regions used by queries from every block and therefore does not preserve block-level communication locality.

4.2 Block-parallel execution

Our first insight is that we can eliminate this block-specific communication by exploiting the per-block separability of Equation 4 to expose target-block computation as a new distributed parallelism dimension. The corrupted activations of block b contribute only to $\mathcal{L}_b$, while the clean representations of its observed prefix can contribute to multiple block losses. BP assigns each complete corrupted-block computation to one rank. The assigned rank executes the corrupted forward pass, loss, and backward pass for block b while keeping the block's corrupted Q/K/V tensors, activations, and gradients local. Different ranks execute different block computations concurrently (Figure 2(b)).

Let $\mathcal{B}_r$ denote the blocks assigned to rank r; these sets partition $\mathcal{S}$. Rank r computes

$$\mathcal{L}_r = \sum_{b \in \mathcal{B}_r} \mathcal{L}_b. \tag{6}$$

Linearity of differentiation gives the global gradient identity

$$\sum_{r=1}^{P} \nabla_\theta \mathcal{L}_r = \sum_{b \in \mathcal{S}} \nabla_\theta \mathcal{L}_b = \nabla_\theta \mathcal{L}. \tag{7}$$

4.3 Context-sharded block parallelism

BP keeps corrupted-block computations local, but BP alone replicates clean prefixes across ranks. A rank can reuse the longest prefix required by its assigned blocks, but these prefixes overlap across

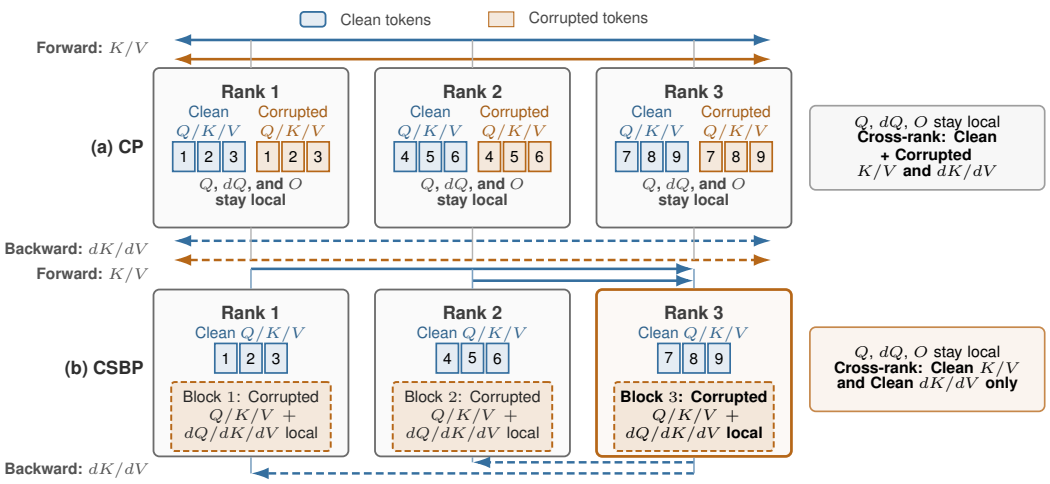

Figure 3: Conventional CP shards the combined clean-plus-corrupted sequence by token position. Clean and corrupted K/Vs cross ranks during forward, and clean and corrupted dK/dV cross ranks during backward; Q, dQ, and attention outputs remain on the ranks storing their queries. CSBP shards only the shared clean sequence, so only clean K/Vs and dK/dV cross ranks. The rank assigned each block retains its corrupted $Q/K/V$ and $dQ/dK/dV$ locally.

ranks, and ranks assigned later blocks may compute and store nearly the full clean sequence. Clean computation and activations can therefore be replicated across up to $\min(B, P)$ active ranks.

Our second insight is that we can scale BP without this replication because BP and CP operate on complementary parts of the computation over the same P ranks. BP assigns complete corrupted-block computations while CP shards the clean-sequence computation shared by those blocks. We call this context-sharded block parallelism (CSBP). Each rank executes a subset of complete corrupted-block computations and stores one clean-sequence shard (Figure 3). Context parallelism supplies the distributed clean K/Vs and accumulates their backward contributions, while BP keeps each corrupted-block computation on its assigned rank.

Forward pass. The rank assigned target block b computes

$$O_b = \mathrm{Attn}\big(Q_b, [K_b; K^{\mathrm{clean}}_{<b}], [V_b; V^{\mathrm{clean}}_{<b}]\big)\,, \tag{8}$$

where Attn denotes scaled dot-product attention. The rank holds Q_b, K_b, and V_b locally, while CP supplies the sharded clean-prefix K/Vs. Online softmax combines the local within-block contribution with the distributed clean-prefix contributions (Milakov & Gimelshein, 2018; Dao et al., 2022).

The target-block computations reuse overlapping prefixes of the clean sequence. During one context-parallel attention pass, rank r applies each clean K/V shard supplied by CP to its clean queries and to every $b \in \mathcal{B}_r$ whose prefix contains tokens from that shard. The length-L distributed clean K/V stream therefore serves every local block whose clean prefix includes that shard. Algorithm 1 gives the complete per-layer computation in the same notation.

Backward pass. The rank assigned block b backpropagates $\mathcal{L}_b$, computing the gradients of Q_b, K_b, and V_b locally together with contributions to the clean-prefix K/V gradients. Exact CP backward accumulates the clean K/V gradient contributions from all clean and assigned corrupted-block queries on the rank storing each clean shard. Backward then continues through the clean projections and earlier layers. Corrupted Q/K/V gradients remain on the ranks assigned those blocks, while clean-activation gradients accumulate on the ranks storing the clean shards. Together, these contributions give the global parameter gradient in Equation 7. Algorithm 2 gives the corresponding backward pass. Implementations 1 and 2 give the production forward and backward fast paths.

Later target blocks and clean-sequence chunks have longer prefixes and require more attention work. We devise a dual-end strategy that assigns target blocks b and $B + 1 - b$ to the same rank. We

use the same early–late pairing for target blocks and zigzag clean-sequence sharding over the same ranks (Dubey et al., 2024). Appendix B shows that this common pairing exactly balances target-block and clean causal attention for equal partitions and handles non-divisible block counts.

4.4 Complexity and sources of improvement

CSBP preserves its exact computation and gradients (Proposition 1) while eliminating cross-rank communication for corrupted K/Vs and avoiding replicated clean prefixes.

Computation. Conventional CP and CSBP evaluate the same valid attention pairs, so both require $\Theta(L^2)$ total attention work and $\Theta(L^2/P)$ work per rank under balanced assignment. CSBP changes where this work is executed. Each corrupted block is evaluated on one rank, while CP supplies the shared clean K/Vs. The gain comes from keeping block-specific computation local and eliminating its cross-rank communication while preserving the same asymptotic attention work.

Communication. Conventional CP applies distributed attention to the combined length-$2L$ clean-plus-corrupted representation. CSBP communicates only clean K/Vs, while corrupted K/Vs remain on their assigned ranks. Global attention requires the full length-L clean sequence. For local patterns such as sliding-window attention, complete-block ownership lets each rank communicate only the clean rows needed by its blocks. In backward, corrupted K/V gradients remain on their assigned ranks, and clean K/V gradients return to the ranks storing those shards. CSBP therefore removes the length-L corrupted component from communication in both forward and backward.

Memory. Each CSBP rank stores one length-L/P clean shard and the activations for its assigned corrupted blocks. With balanced block assignments, each rank stores approximately L/P clean tokens and L/P corrupted tokens, giving approximately $2L/P$ token activations and matching conventional CP to first order. Pure BP can store a clean prefix approaching length L on ranks assigned later blocks. CSBP reduces this clean-prefix component to L/P, providing up to a $P\times$ reduction for a rank whose longest required prefix approaches L. CSBP therefore retains CP's activation scaling without pure BP's overlapping-prefix replication.

Speculative drafters. For speculative drafter training, the BP component of CSBP assigns each draft block to one rank, so its block-local path runs once rather than once per context rank. Target-feature processing remains context sharded. Appendix E formalizes this execution-count reduction and gives the corresponding loss and gradient decomposition.

5 Experiments

5.1 Experimental setup

Models and hardware. We evaluate three workloads: SFT on NemotronDiffusion 3B, 8B, and 14B and DiffusionGemma 26B-A4B; AR-to-BDLM conversion on Qwen3.5-27B and Qwen3.8-27B with Fast-dLLM v2; and DFlash2 drafter training on Qwen3.8-27B and Muse-Glimmer-30B. BDLM fine-tuning and AR-to-BDLM conversion use 16 NVIDIA H200 GPUs (141 GB each); DFlash2 uses eight H100 GPUs (80 GB each).

Comparisons and metrics. Each best baseline is the fastest feasible tested non-BP configuration over batch size and standard parallelism dimensions, with token-sharded CP when needed. We profile-optimize it, share applicable kernel and layout optimizations, and tune CSBP independently. Compared methods use identical objectives and target blocks. All runs use BF16. We report useful MFU and maximum allocated HBM; DFlash2 counts supervised draft-target tokens but not replicated work. Profiles use fixed reported context lengths; equal-wall-clock runs use dataset lengths. Appendix F gives full details.

5.2 Effectiveness of CSBP

CSBP improves throughput across all evaluated BDLM training settings at 256K context (Table 1). It delivers 1.18–$1.45\times$ speedup for SFT and 1.27–$1.33\times$ for AR-to-BDLM conversion, while matching

or reducing peak HBM. The gains extend from NemotronDiffusion to DiffusionGemma 26B-A4B and both Qwen backbones, demonstrating consistent benefits across the evaluated model families.

Table 1: CSBP accelerates training at 256K context.

Training workload	Model	Best baseline topology	Ours topology	Tok/s	Speedup	MFU (%)	Peak HBM (GiB)
BDLM fine-tuning	NemotronDiffusion 3B	DP4/TP1/CP4	DP4/TP1/CP4/BP4	13,434 / **16,027**	**1.19**×	31.7 / **37.8**	57.1 / **57.0**
BDLM fine-tuning	NemotronDiffusion 8B	DP4/TP1/CP4	DP4/TP1/CP4/BP4	9,668 / **11,379**	**1.18**×	32.3 / **38.0**	97.5 / 97.5
BDLM fine-tuning	NemotronDiffusion 14B	DP2/TP1/CP8	DP2/TP1/CP8/BP8	7,884 / **9,301**	**1.18**×	33.1 / **39.0**	91.1 / **90.6**
BDLM fine-tuning	DiffusionGemma 26B-A4B	DP1/TP1/EP2/CP8	DP1/TP1/EP2/CP8/BP8	10,557 / **15,359**	**1.45**×	12.7 / **18.4**	118.1 / **113.6**
AR-to-BDLM conversion	Qwen3.8-27B	DP1/TP2/CP8	DP1/TP2/CP8/BP8	2,623 / **3,490**	**1.33**×	20.1 / **26.8**	120.7 / **111.1**
AR-to-BDLM conversion	Qwen3.5-27B	DP1/TP2/CP8	DP1/TP2/CP8/BP8	2,641 / **3,351**	**1.27**×	20.3 / **25.7**	120.7 / **111.1**

Pairs: best baseline / ours. Bold marks the better value.

5.3 Diffusion-based speculative drafter training

On eight H100 GPUs (Table 2), CSBP produces larger DFlash2 speedups than the BDLM training workloads because conventional CP replicates the block-local decoder, vocabulary loss, and selector across context ranks, whereas CSBP assigns each draft block to one rank. At 512K, matched DP4/CP2 configurations isolate a $2.48\times$ gain from removing this duplication. The 1M $7.59\times$ system-level gain also includes four data-parallel replicas enabled by CSBP's lower per-replica cost. At 512K, CSBP also cuts exposed attention communication from 105.4 to 7.5 ms per GPU because each block stays on one rank (Appendix E). At 256K, where full replicas fit, DP8 remains 19.2% faster because it avoids sequence-parallel overhead.

Table 2: CSBP accelerates long-context DFlash2 drafter training.

Model	Context length	Best baseline topology	Ours topology	Tok/s	Speedup	MFU (%)	Peak HBM (GiB)
Qwen3.8-27B	256K	DP8	DP4/CP2/BP2	**167,228** / 140,350	0.84×	**38.4** / 32.2	63.5 / **48.3**
Qwen3.8-27B	512K	DP4/CP2	DP4/CP2/BP2	57,027 / **141,222**	**2.48**×	13.1 / **32.4**	63.3 / **54.6**
Qwen3.8-27B	1M	DP1/CP8	DP4/CP2/BP2	18,389 / **139,541**	**7.59**×	4.2 / **32.0**	**53.7** / 67.1
Muse-Glimmer-30B	256K	DP4/CP2	DP4/CP2/BP2	62,698 / **98,046**	**1.56**×	18.5 / **28.9**	63.8 / **57.5**
Muse-Glimmer-30B	512K	DP2/CP4	DP4/CP2/BP2	29,493 / **98,785**	**3.35**×	8.7 / **29.1**	**63.8** / 65.6
Muse-Glimmer-30B	1M	DP1/CP8	DP2/CP4/BP4	12,842 / **61,742**	**4.81**×	3.8 / **18.2**	63.3 / **55.7**

Pairs: best baseline / ours. Tok/s and MFU count supervised draft-target tokens. Bold marks the better value.

5.4 BDLM fine-tuning scaling with context length

Longer contexts strengthen the throughput advantage of CSBP (Figure 4). Extending context from 64K to 512K increases speedup from $1.10\times$ to $1.19\times$ for NemotronDiffusion 14B and from $1.25\times$ to $1.61\times$ for DiffusionGemma 26B-A4B. The gains persist through 512K, consistent with avoiding corrupted K/V communication becoming more valuable as sequences grow.

Context sharding is what makes these long-context gains practical. Without it, BP replicates overlapping clean prefixes: NemotronDiffusion 14B pure BP is slower and uses more memory at 64K, then runs out of memory at 128K; DiffusionGemma 26B-A4B pure BP runs out of memory at both 64K and 128K under the matched native objective. CSBP fits and accelerates training in these cases (Appendix I).

5.5 Equal-wall-clock downstream performance

We finally test whether CSBP's higher training throughput translates into faster downstream progress under a fixed wall-clock budget. We fine-tune DiffusionGemma 26B-A4B with the best baseline and CSBP for 12 hours and evaluate checkpoints every three hours on SWE-bench Verified and Terminal-Bench Lite (Figure 5). We train with LoRA adapters on eight H100 GPUs, comparing CP8

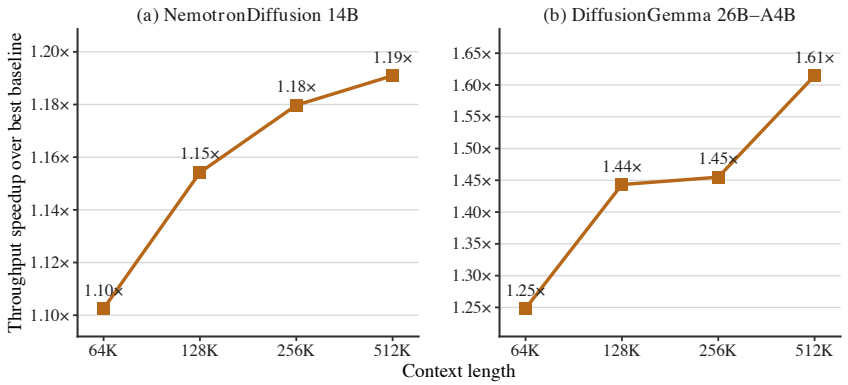

Figure 4: Longer contexts amplify CSBP's throughput advantage.

with CSBP at CP8/BP8. Appendix G provides full data and training details. CSBP achieves a higher pass rate at every trained checkpoint on both benchmarks.

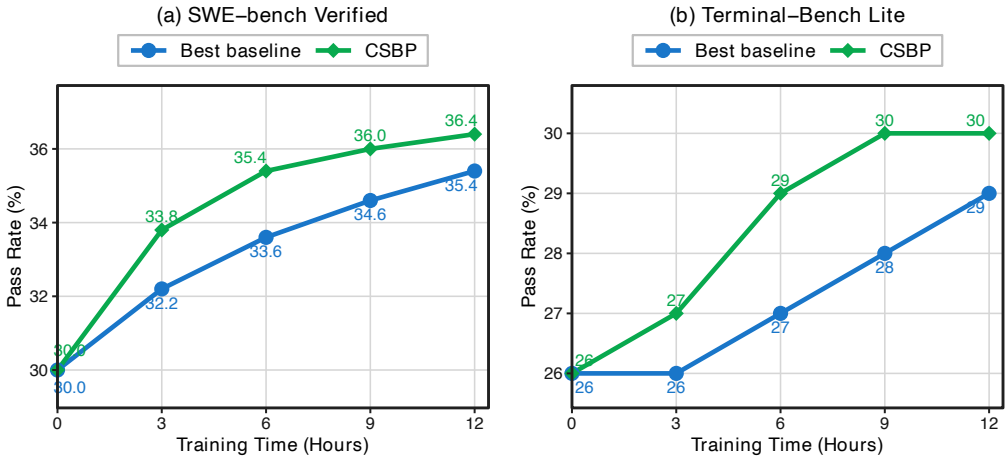

Figure 5: Equal-wall-clock downstream performance. Checkpoints from the best baseline and CSBP are evaluated every three hours on (a) SWE-bench Verified and (b) Terminal-Bench Lite.

6 Limitations

CSBP requires known training targets and accelerates training, not generation of unknown future tokens.

7 Conclusion

We introduce block parallelism (BP), which assigns complete target-block computations to ranks, and context-sharded BP (CSBP), which also shards the shared clean context for long-context training. This alignment keeps corrupted K/Vs and their gradients local, avoids clean-prefix and drafter-computation replication, and preserves computation and gradients. CSBP delivers up to **1.61**× BDLM fine-tuning and **7.59**× drafter-training speedups and higher pass rates under fixed training budgets. Block-aware parallelism improves long-context BDLM and diffusion-based speculative-drafter training without changing either objective. BP and CSBP complement data, tensor, and expert parallelism.

AI USE STATEMENT

We used AI tools for manuscript editing and LaTeX formatting. All AI-generated outputs were reviewed by the authors and edited for accuracy and clarity. The authors take full responsibility for the final manuscript, implementation, and reported results.

ETHICS STATEMENT

This work improves the computational efficiency of training long-context block-diffusion models and speculative drafters. Our experiments use previously released coding and terminal-interaction datasets and benchmarks. Lower training costs can support broader model development, but may also enable more capable systems whose code generation or tool use could cause harm. The reported evaluations measure efficiency and task completion rather than security, alignment, or deployment readiness. Real-world use should therefore incorporate sandboxing, least-privilege tool access, monitoring, and separate assessments of misuse, privacy, and dataset risks.

REPRODUCIBILITY STATEMENT

Section 4 describes BP and CSBP, and Appendix A states the assumptions and establishes equivalence to the reference computation. Appendix C gives forward and backward schedules. Section 5.1 and Appendices F and G describe the experimental setup, training data, and evaluation; Appendices D and E detail the conversion and drafter objectives.

REFERENCES

Alpay Ariyak, Junda Zhang, Junxiong Wang, Shang Zhu, Federico Bianchi, Sanjana Srivastava, Ashwinee Panda, Siddhant Bharti, Chenfeng Xu, John Heo, Xiaoxia Shirley Wu, James Zou, Percy Liang, Leon Song, Ce Zhang, Ben Athiwaratkun, Zhongzhu Zhou, and Qingyang Wu. CoderForge-Preview: SOTA open dataset for training efficient agents, February 2026. URL https://www.together.ai/blog/coderforge-preview. Project core leads: Alpay Ariyak; Zhongzhu Zhou; Qingyang Wu.

Marianne Arriola, Aaron Kerem Gokaslan, Justin T. Chiu, Zhihan Yang, Zhixuan Qi, Jiaqi Han, Subham Sekhar Sahoo, and Volodymyr Kuleshov. Block diffusion: Interpolating between autoregressive and diffusion language models. In *International Conference on Learning Representations*, 2025. URL https://arxiv.org/abs/2503.09573.

William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431*, 2023. URL https://arxiv.org/abs/2311.09431.

Jian Chen, Yesheng Liang, and Zhijian Liu. DFlash: Block Diffusion for Flash Speculative Decoding. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2026. URL https://arxiv.org/abs/2602.06036.

Tianqi Chen, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. Training deep nets with sublinear memory cost. *arXiv preprint arXiv:1604.06174*, 2016. URL https://arxiv.org/abs/1604.06174.

Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023. URL https://arxiv.org/abs/2204.02311.

Neil Chowdhury, James Aung, Chan Jun Shern, Oliver Jaffe, Dane Sherburn, Giulio Starace, Evan Mays, Rachel Dias, Marwan Aljubeh, Mia Glaese, Carlos E. Jimenez, John Yang, Leyton Ho, Tejal Patwardhan, Kevin Liu, and Aleksander Madry. Introducing SWE-bench verified. OpenAI Blog, August 2024. URL https://openai.com/index/introducing-swe-bench-verified/.

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, volume 35, pp. 16344–16359, 2022. URL https://arxiv.org/abs/2205.14135.

DiffusionGemma Team, Adrien Ali Taïga, James Assiene, Daniele Calandriello, Rahma Chaabouni, João Gante, Tamara von Glehn, Nate Keating, Chris Knutsen, Martin Kukla, Tianlin Liu, Ivan Lobov, Ofir Nabati, João Gabriel Oliveira, Nicolas Perez-Nieves, Nastasia Prutianova, Bobak Shahriari, Jean Tarbouriech, Pavel Tyletski, Çağlar Ünlü, Cindy Wu, Glenn Cameron, Jerome Connor, Sertan Girgin, Maarten Grootendorst, Alon Levkovitch, Eliya Nachmani, Omar Sanseviero, Piotr Stanczyk, Quentin Berthet, Andrew Campbell, Clément Crepy, Valentin De Bortoli, Arnaud Doucet, Romuald Elie, Alexandre Galashov, Klaus Greff, Alexis Jacq, David Ruhe, Yu-Han Wu, Sebastian Flennerhag, Brendan O'Donoghue, George Scrivener, and Shantanu Thakoor. DiffusionGemma technical report. *arXiv preprint arXiv:2608.00146*, 2026. URL https://arxiv.org/abs/2608.00146.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex-Attention: A programming model for generating fused attention variants. In *Proceedings of Machine Learning and Systems*, volume 7, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/61a9278dfef5f871b5e472389f8d6fa1-Abstract-Conference.html.

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/2407.21783v1.

Yonggan Fu, Lexington Whalen, Abhinav Garg, et al. Nemotron-Labs-Diffusion: A tri-mode language model unifying autoregressive, diffusion, and self-speculation decoding. *arXiv preprint arXiv:2607.05722*, 2026. URL https://arxiv.org/abs/2607.05722.

Diandian Gu, Peng Sun, Qinghao Hu, Ting Huang, Xun Chen, Yingtong Xiong, Guoteng Wang, Qiaoling Chen, Shangchun Zhao, Jiarui Fang, Yonggang Wen, Tianwei Zhang, Xin Jin, and Xuanzhe Liu. LoongTrain: Efficient training of long-sequence LLMs with head-context parallelism. *arXiv preprint arXiv:2406.18485*, 2024. URL https://arxiv.org/abs/2406.18485.

Harbor Framework Team. Harbor: A framework for evaluating and optimizing agents and models in container environments, August 2026. URL https://doi.org/10.5281/zenodo.22052497. Version v0.22.0.

Inco AI. DFlash 2: Keep Drafting Parallel, August 2026. URL https://inco.ai/blog/dflash2/.

Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. DeepSpeed-Ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023. URL https://arxiv.org/abs/2309.14509.

Carlos E. Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R. Narasimhan. SWE-bench: Can language models resolve real-world GitHub issues? In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=VTF8yNQM66.

Vijay Anand Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models. In *Proceedings of Machine Learning and Systems*, volume 5, pp. 341–353, 2023. URL https://proceedings.mlsys.org/paper_files/paper/2023/hash/80083951326cf5b35e5100260d64ed81-Abstract-mlsys2023.html.

Dmitry Lepikhin, HyoukJoong Lee, Yuanzhong Xu, Dehao Chen, Orhan Firat, Yanping Huang, Maxim Krikun, Noam Shazeer, and Zhifeng Chen. GShard: Scaling giant models with conditional computation and automatic sharding. In *International Conference on Learning Representations*, 2021. URL https://arxiv.org/abs/2006.16668.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pp. 19274–19286, 2023. URL https://proceedings.mlr.press/v202/leviathan23a.html.

Shenggui Li, Yikai Zhu, Chao Wang, Fan Yin, Shuai Shi, Yubo Wang, Yi Zhang, Yingyi Huang, Haoshuai Zheng, and Yineng Zhang. SpecForge: Train speculative decoding models effortlessly. https://github.com/sgl-project/specforge, 2025.

Shenggui Li, Chao Wang, Yikai Zhu, Yubo Wang, Fan Yin, Shuai Shi, Yefei Chen, Xiaomin Dong, Qiaoling Chen, Jin Pan, et al. SpecForge: A flexible and efficient open-source training framework for speculative decoding. *arXiv preprint arXiv:2603.18567*, 2026.

Wanchao Liang, Tianyu Liu, Less Wright, Will Constable, Andrew Gu, Chien-Chin Huang, Iris Zhang, Wei Feng, Howard Huang, Junjie Wang, Sanket Purandare, Gokul Nadathur, and Stratos Idreos. TorchTitan: One-stop PyTorch native solution for production ready LLM pre-training. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=SFN6Wm7YBI.

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. In *International Conference on Learning Representations*, 2024. URL https://proceedings.iclr.cc/paper_files/paper/2024/hash/1119587863e78451f080da2a768c4935-Abstract-Conference.html.

Maxim Milakov and Natalia Gimelshein. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018. URL https://arxiv.org/abs/1805.02867.

NVIDIA. Context Parallel Package — Megatron Core. Software documentation, n.d. URL https://docs.nvidia.com/megatron-core/developer-guide/latest/user-guide/features/context_parallel.html. Accessed September 5, 2026.

Brendan O'Donoghue and Sebastian Flennerhag. DiffusionGemma: 4x faster text generation. Google Blog, June 2026. URL https://blog.google/innovation-and-ai/technology/developers-tools/diffusion-gemma-faster-text-generation/.

OpenThoughts-Agent team, Snorkel AI, Bespoke Labs. OpenThoughts-TBLite: A high-signal benchmark for iterating on terminal agents. https://www.openthoughts.ai/blog/openthoughts-tblite, February 2026.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, 2019. URL https://papers.neurips.cc/paper_files/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Xiaoxuan Peng, Kaiqi Zhang, Xinyu Lu, Boxi Cao, Yaojie Lu, Hongyu Lin, Xianpei Han, and Le Sun. LiteCoder-Terminal: Scaling long-horizon terminal environments for learning language agents. *arXiv preprint arXiv:2605.29559*, 2026. URL https://arxiv.org/abs/2605.29559.

Qwen Team. Qwen3.5: Towards native multimodal agents, February 2026a. URL https://qwen.ai/blog?id=qwen3.5.

Qwen Team. Qwen3.8-Max: A new bar for coding and cowork, August 2026b. URL https://qwen.ai/blog?id=qwen3.8.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. ZeRO: Memory optimizations toward training trillion parameter models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '20, pp. 1–16, 2020. doi: 10.1109/SC41405.2020.00024.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-LM: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. URL https://arxiv.org/abs/1909.08053.

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*, volume 30, pp. 5998–6008, 2017.

Xingyao Wang, Boxuan Li, Yufan Song, Frank F. Xu, Xiangru Tang, Mingchen Zhuge, Jiayi Pan, Yueqi Song, Bowen Li, Jaskirat Singh, Hoang H. Tran, Fuqiang Li, Ren Ma, Mingzhang Zheng, Bill Qian, Yanjun Shao, Niklas Muennighoff, Yizhe Zhang, Binyuan Hui, Junyang Lin, Robert Brennan, Hao Peng, Heng Ji, and Graham Neubig. OpenHands: An open platform for AI software developers as generalist agents. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=OJd3ayDDoF.

Chengyue Wu, Hao Zhang, Shuchen Xue, Shizhe Diao, Yonggan Fu, Zhijian Liu, Pavlo Molchanov, Ping Luo, Song Han, and Enze Xie. Fast-dLLM v2: Efficient block-diffusion LLM. *arXiv preprint arXiv:2509.26328*, 2025. URL https://arxiv.org/abs/2509.26328.

John Yang, Carlos E. Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. SWE-agent: Agent-computer interfaces enable automated software engineering. *arXiv preprint arXiv:2405.15793*, 2024. URL https://arxiv.org/abs/2405.15793.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik Narasimhan, and Yuan Cao. ReAct: Synergizing reasoning and acting in language models. In *International Conference on Learning Representations*, 2023. URL https://arxiv.org/abs/2210.03629.

Ted Zadouri, Markus Hoehnerbach, Jay Shah, Timmy Liu, Vijay Thakkar, and Tri Dao. FlashAttention-4: Algorithm and kernel pipelining co-design for asymmetric hardware scaling. *arXiv preprint arXiv:2603.05451*, 2026. URL https://arxiv.org/abs/2603.05451.

A CORRECTNESS OF CONTEXT-SHARDED BLOCK PARALLELISM

Proposition 1 (Exactness of CSBP). *Let $\{\mathcal{B}_r\}_{r=1}^{P}$ partition the B target blocks, and let the clean token rows be partitioned arbitrarily across the same P ranks. Suppose the reference computation uses any fixed layerwise attention mask under which clean queries attend only to clean rows and corrupted queries in block b attend only to clean rows and corrupted rows from block b. This includes models that mix global block-causal and sliding-window layers. Given any fixed input sequence, sampled diffusion times and corruptions, per-token model randomness, and loss normalization, CSBP computes, for every parameter value θ, the same global sampled loss and parameter gradient as the reference length-$2L$ BDLM computation, up to floating-point reduction order.*

Proof. Fix the input sequence, sampled diffusion times and corruptions, model randomness, and loss normalization. Let $\mathcal{L}_b^{\mathrm{ref}}(\theta)$ and $\mathcal{L}_b^{\mathrm{CSBP}}(\theta)$ be the resulting sampled losses for block b. For rank r, define

$$\mathcal{L}_r^{\mathrm{CSBP}} = \sum_{b\in\mathcal{B}_r} \mathcal{L}_b^{\mathrm{CSBP}}, \qquad \mathcal{L}^{\mathrm{CSBP}} = \sum_{r=1}^{P} \mathcal{L}_r^{\mathrm{CSBP}}, \qquad \mathcal{L}^{\mathrm{ref}} = \sum_{b=1}^{B} \mathcal{L}_b^{\mathrm{ref}}. \tag{9}$$

Let H_d^{clean} and $H_{b,d}^{\mathrm{corr}}$ denote the clean representations and the corrupted representations of block b after d of the model's D transformer layers. Thus H_0 is the embedding output and H_D enters the output projection. We use H_d to denote all clean and corrupted representations at depth d, and θ_d to denote the parameters of layer d for $1 \le d \le D$.

1. **Reference dependencies.** The length-$2L$ construction and the attention mask at layer d determine the visible K/V rows. Let $S_{q,d}^{\mathrm{clean}}$ be the clean positions visible to a clean query at position q. For corrupted query u in block b, let $S_{b,u,d}^{\mathrm{corr}}$ and $S_{b,u,d}^{\mathrm{clean}}$ be its visible corrupted and clean positions, respectively. By assumption, $S_{b,u,d}^{\mathrm{corr}}$ contains positions only from block b. Under global block-causal attention, these sets contain the full clean prefix and the permitted within-block corrupted rows; sliding-window attention intersects the same sets with the layer's window. The reference attention is

$$O_{q,d}^{\mathrm{clean,ref}} = \mathrm{Attn}\Big(Q_{q,d}^{\mathrm{clean}}, K_{S_{q,d}^{\mathrm{clean}},d}^{\mathrm{clean}}, V_{S_{q,d}^{\mathrm{clean}},d}^{\mathrm{clean}}\Big), \tag{10}$$

$$O_{b,u,d}^{\mathrm{corr,ref}} = \mathrm{Attn}\Big(Q_{b,u,d}^{\mathrm{corr}}, [K_{S_{b,u,d}^{\mathrm{corr}},d}^{\mathrm{corr}}; K_{S_{b,u,d}^{\mathrm{clean}},d}^{\mathrm{clean}}], [V_{S_{b,u,d}^{\mathrm{corr}},d}^{\mathrm{corr}}; V_{S_{b,u,d}^{\mathrm{clean}},d}^{\mathrm{clean}}]\Big). \tag{11}$$

   Let $O_{b,d}^{\mathrm{corr,ref}}$ collect these outputs over the corrupted queries u in block b. Equation 11 then implies, for every $Z \in \{Q, K, V\}$,

$$\frac{\partial O_{b,d}^{\mathrm{corr,ref}}}{\partial Z_{b',d}^{\mathrm{corr}}} = 0 \qquad (b' \neq b). \tag{12}$$

2. **Exact sharded attention.** Let C_r be the set of clean token positions stored on rank r. The clean shards satisfy

$$\bigcup_{r=1}^{P} C_r = \{1, \dots, L\}, \qquad C_r \cap C_{r'} = \varnothing \quad (r \neq r'). \tag{13}$$

   Consider one query vector $\mathbf{q}$. Let S be its visible key positions, partitioned into disjoint nonempty sets $S_1, \dots, S_J$. For the key and value rows $\mathbf{k}_i$ and $\mathbf{v}_i$, let $a_i = \mathbf{q}\mathbf{k}_i^{\top}/\sqrt{d_h}$ and $a_{\max} = \max_{i\in S} a_i$. Since the sets S_j partition S,

$$\mathrm{Attn}(\mathbf{q}, K_S, V_S) = \frac{\displaystyle\sum_{j=1}^{J}\sum_{i\in S_j} e^{a_i - a_{\max}}\mathbf{v}_i}{\displaystyle\sum_{j=1}^{J}\sum_{i\in S_j} e^{a_i - a_{\max}}}. \tag{14}$$

Online softmax evaluates these partitioned sums exactly (Milakov & Gimelshein, 2018; Dao et al., 2022). For a clean query at position q, the nonempty sets $C_r \cap S^{\text{clean}}_{q,d}$ partition its visible K/V rows. For corrupted query u in block b, the local set $S^{\text{corr}}_{b,u,d}$ together with the nonempty clean sets $C_r \cap S^{\text{clean}}_{b,u,d}$ partitions its visible K/V rows. CSBP therefore supplies exactly the rows in Equations 10 and 11.

3. **Forward induction.** The common inputs and embedding computation give $H_0^{\text{CSBP}} = H_0^{\text{ref}}$. Assume $H_{d-1}^{\text{CSBP}} = H_{d-1}^{\text{ref}}$ for some $1 \le d \le D$. Their Q/K/V projections agree, and Equation 14 gives

$$\begin{aligned} O^{\text{clean,CSBP}}_{q,d} &= O^{\text{clean,ref}}_{q,d} \qquad \forall q, \\ O^{\text{corr,CSBP}}_{b,d} &= O^{\text{corr,ref}}_{b,d} \qquad \forall b. \end{aligned} \tag{15}$$

The remaining operations in layer d are unchanged, so $H_d^{\text{CSBP}} = H_d^{\text{ref}}$. Induction over $d = 1, \dots, D$ gives $H_D^{\text{CSBP}} = H_D^{\text{ref}}$.

4. **Loss.** The output projection and block losses are unchanged. The forward induction therefore gives $\mathcal{L}_b^{\text{CSBP}} = \mathcal{L}_b^{\text{ref}}$ for every block. Since $\{\mathcal{B}_r\}_{r=1}^P$ partitions the target blocks,

$$\mathcal{L}^{\text{CSBP}} = \sum_{r=1}^{P} \sum_{b \in \mathcal{B}_r} \mathcal{L}_b^{\text{CSBP}} = \sum_{b=1}^{B} \mathcal{L}_b^{\text{ref}} = \mathcal{L}^{\text{ref}}. \tag{16}$$

5. **Backward pass.** The equal final representations and common loss functions give the same derivatives at the output of the model. Let $\mathcal{L}$ denote the global sampled loss of either execution. For $Z \in \{Q, K, V\}$, Equation 12 and the chain rule give

$$\frac{\partial \mathcal{L}}{\partial Z^{\text{corr}}_{b,d}} = \sum_{b'=1}^{B} \left(\frac{\partial O^{\text{corr}}_{b',d}}{\partial Z^{\text{corr}}_{b,d}} \right)^{\top} \frac{\partial \mathcal{L}}{\partial O^{\text{corr}}_{b',d}} = \left(\frac{\partial O^{\text{corr}}_{b,d}}{\partial Z^{\text{corr}}_{b,d}} \right)^{\top} \frac{\partial \mathcal{L}}{\partial O^{\text{corr}}_{b,d}}. \tag{17}$$

Thus corrupted Q/K/V derivatives depend only on the queries from the same block and are computed on the rank assigned that block. Let $Z^{\text{clean}}_{r,d}$ denote the clean K/V rows at positions C_r, and let $O_{q,d}$ denote the attention output for query row q. The chain rule gives

$$\frac{\partial \mathcal{L}}{\partial Z^{\text{clean}}_{r,d}} = \sum_{q} \left(\frac{\partial O_{q,d}}{\partial Z^{\text{clean}}_{r,d}} \right)^{\top} \frac{\partial \mathcal{L}}{\partial O_{q,d}}, \qquad Z \in \{K, V\}, \tag{18}$$

where the sum is over all clean and corrupted query rows. A term is zero when the query cannot attend to a position in C_r. Exact CP backward accumulates this sum on rank r. For every clean query row q,

$$\frac{\partial \mathcal{L}}{\partial Q^{\text{clean}}_{q,d}} = \left(\frac{\partial O^{\text{clean}}_{q,d}}{\partial Q^{\text{clean}}_{q,d}} \right)^{\top} \frac{\partial \mathcal{L}}{\partial O^{\text{clean}}_{q,d}}, \tag{19}$$

which is computed on the rank storing the query. Because the two executions have the same forward values and output derivatives, Equations 17–19 give the same Q/K/V derivatives. Backpropagation through the unchanged remainder of layer d therefore satisfies

$$\begin{aligned} \frac{\partial \mathcal{L}^{\text{CSBP}}}{\partial H_{d-1}^{\text{CSBP}}} &= \left(\frac{\partial H_d^{\text{CSBP}}}{\partial H_{d-1}^{\text{CSBP}}} \right)^{\top} \frac{\partial \mathcal{L}^{\text{CSBP}}}{\partial H_d^{\text{CSBP}}} \\ &= \left(\frac{\partial H_d^{\text{ref}}}{\partial H_{d-1}^{\text{ref}}} \right)^{\top} \frac{\partial \mathcal{L}^{\text{ref}}}{\partial H_d^{\text{ref}}} = \frac{\partial \mathcal{L}^{\text{ref}}}{\partial H_{d-1}^{\text{ref}}}, \\ \nabla_{\theta_d} \mathcal{L}^{\text{CSBP}} &= \left(\frac{\partial H_d^{\text{CSBP}}}{\partial \theta_d} \right)^{\top} \frac{\partial \mathcal{L}^{\text{CSBP}}}{\partial H_d^{\text{CSBP}}} \\ &= \left(\frac{\partial H_d^{\text{ref}}}{\partial \theta_d} \right)^{\top} \frac{\partial \mathcal{L}^{\text{ref}}}{\partial H_d^{\text{ref}}} = \nabla_{\theta_d} \mathcal{L}^{\text{ref}}. \end{aligned} \tag{20}$$

Because the output projection is identical, its parameter gradient and derivative with respect to H_D agree, providing the base case. Reverse induction through $d = D, \ldots, 1$ gives equal transformer-layer gradients and equal derivatives with respect to H_0. The common embedding computation then gives equal embedding gradients. Finally, linearity of differentiation and Equation 9 give

$$\sum_{r=1}^{P} \nabla_\theta \mathcal{L}_r^{\mathrm{CSBP}} = \nabla_\theta \mathcal{L}^{\mathrm{CSBP}} = \nabla_\theta \mathcal{L}^{\mathrm{ref}}. \tag{21}$$

Equations 16 and 21 hold for every sampled realization. Taking expectations also gives the same BDLM objective and objective gradient. □

Communication consequence. Under CSBP, only the required clean K/V rows and their gradients cross ranks during attention, while corrupted Q/K/V tensors and their gradients remain on the rank assigned their block.

B Load Balancing

Both attention workloads in CSBP increase with block position. Later target blocks attend to longer clean prefixes, and later clean query blocks attend to longer causal prefixes. Because both costs grow with the same block index, the same early–late pairing balances them simultaneously.

Assume B equal-sized blocks of $M = L/B$ tokens. Let W_b^{target} and W_b^{clean} denote the valid query–key pairs for target-block and clean causal attention at block b. The M target queries in block b attend to $(b-1)M$ clean-prefix keys and M keys from their own corrupted block, for bM visible keys. The M clean queries attend to the $b-1$ preceding clean blocks and causally within their own block. Their work is therefore

$$W_b^{\mathrm{target}} = bM^2, \qquad W_b^{\mathrm{clean}} = (b-1)M^2 + \frac{M(M+1)}{2}. \tag{22}$$

Dual-end BP pairs target block b with block $B+1-b$. Clean-sequence zigzag sharding uses the same early–late pairing (Dubey et al., 2024). The work of each pair is

$$W_b^{\mathrm{target}} + W_{B+1-b}^{\mathrm{target}} = (B+1)M^2, \qquad W_b^{\mathrm{clean}} + W_{B+1-b}^{\mathrm{clean}} = BM^2 + M. \tag{23}$$

Both sums are independent of b, so every early–late pair carries the same amount of target-block work and the same amount of clean causal work. When B is divisible by $2P$, assigning $B/(2P)$ pairs to each rank gives

$$W_r^{\mathrm{target}} = \frac{B(B+1)M^2}{2P}, \qquad W_r^{\mathrm{clean}} = \frac{L(L+1)}{2P}, \qquad 1 \le r \le P. \tag{24}$$

Using the same pairs for target-block placement and clean zigzag sharding gives every rank equal target-block and clean causal attention work. Attention backward uses the same valid query–key pairs, so these pair counts remain balanced. When exact divisibility does not hold, we distribute the complete pairs as evenly as possible and assign any unpaired middle block to the currently least-loaded rank.

C CSBP ALGORITHM AND EXECUTION SCHEDULE

Algorithm 1 summarizes one CSBP layer. Each rank stores one clean-sequence shard and the complete corrupted blocks in $\mathcal{B}_r$. All ranks execute concurrently: CP supplies the clean K/V rows required by the rank's queries, while each block's corrupted Q/K/V tensors remain local. Global attention requires all clean K/V shards; sliding-window attention requires only the clean rows used by the rank's clean queries and assigned blocks.

Algorithm 1 CSBP forward pass on rank r

Input: Clean shard (Q_r, K_r, V_r) and $\{(Q_b, K_b, V_b) : b \in \mathcal{B}_r\}$
Output: O_r and $\{O_b : b \in \mathcal{B}_r\}$
Exchange the required clean K/V rows to form $(K_r^{\mathrm{req}}, V_r^{\mathrm{req}})$ on rank r *(all clean rows for global attention)*

Clean-sequence attention (CP)
Compute clean-query attention under the block-causal mask
$O_r \leftarrow \mathrm{softmax}\big(Q_r (K_r^{\mathrm{req}})^\top / \sqrt{d_h} + \mathcal{M}_r\big)\, V_r^{\mathrm{req}}$

Assigned corrupted blocks (BP)
for all $b \in \mathcal{B}_r$ **do**
    Compute within-block attention from the local Q_b, K_b, V_b
    Compute clean-prefix attention from the required rows in $K_r^{\mathrm{req}}, V_r^{\mathrm{req}}$
    Merge the two outputs and LSEs by online softmax to obtain
    $O_b \leftarrow \mathrm{Attn}\big(Q_b, [K_b; K_{<b}^{\mathrm{clean}}], [V_b; V_{<b}^{\mathrm{clean}}]\big)$ *(Equation 8)*
end for
return O_r and $\{O_b : b \in \mathcal{B}_r\}$

Algorithm 2 gives the exact backward pass. Rank r keeps the corrupted Q/K/V gradients for its assigned blocks, while CP sums the clean K/V gradients contributed by all clean and corrupted queries.

Algorithm 2 CSBP attention backward on rank r

Input: Saved forward tensors and $\{\mathcal{L}_b : b \in \mathcal{B}_r\}$
Output: (dQ_r, dK_r, dV_r) and $\{(dQ_b, dK_b, dV_b) : b \in \mathcal{B}_r\}$

Clean-query backward (CP)
Backpropagate O_r; retain dQ_r and the clean K/V contributions

Assigned corrupted blocks (local backward)
for all $b \in \mathcal{B}_r$ **do**
    Backpropagate $\mathcal{L}_b$ through O_b
    $(dQ_b, dK_b, dV_b) \leftarrow \nabla_{(Q_b, K_b, V_b)} \mathcal{L}_b$ *(kept on rank r)*
    Add this block's clean-prefix K/V gradients to the CP reduction
end for

Clean-gradient reduction (CP backward)
Return the clean K/V contributions to the ranks storing their shards
for $Z \in \{K, V\}$ **do**
    $$dZ_r \leftarrow \sum_q \left(\frac{\partial O_q}{\partial Z_r}\right)^\top \frac{\partial \mathcal{L}}{\partial O_q}$$ *(Equation 18)*
end for
return (dQ_r, dK_r, dV_r) and $\{(dQ_b, dK_b, dV_b) : b \in \mathcal{B}_r\}$

Efficient fused operator. Implementation 1 shows the production forward path for global attention. It starts an asynchronous all-gather of clean K/Vs, then computes all local corrupted blocks with one block-diagonal attention launch while communication runs. After the all-gather, one masked fused-attention launch computes the clean-K/V contribution for every query. The mask uses each gathered key's original clean token position, so the kernel consumes the gathered K/Vs directly without a reordering copy.

For sliding-window attention, the same schedule replaces the full all-gather with a variable-size exchange containing only the clean rows required by the rank's clean queries and assigned blocks. The backward pass returns gradients for those rows to the ranks storing the clean shards. Complete block ownership and the aligned clean-sequence assignment make most required rows local.

Each attention launch returns a normalized output O_j and row-wise log-sum-exp (LSE) z_j. For two disjoint key sets, the CUDA kernel sets $m = \max(z_1, z_2)$, $w_j = e^{z_j - m}$, and computes

$$O = \frac{w_1 O_1 + w_2 O_2}{w_1 + w_2}, \qquad z = m + \log(w_1 + w_2). \tag{25}$$

This is Equation 14 applied to the two attention launches. The implementation stores the numerator, m, and the normalizer $w_1 + w_2$ in FP32 until the final division.

```
@cuda_kernel
def merge_lse_kernel(numerator, m, normalizer,
                     O_block, z_block):
    # One thread handles one (corrupted-query row, head-dimension) element.
    q, d = cuda_grid_2d()
    if q >= len(z_block):
        return
    m_new = max(m[q], z_block[q])
    w_acc = exp(m[q] - m_new) if isfinite(m[q]) else 0.0
    w_block = exp(z_block[q] - m_new) if isfinite(z_block[q]) else 0.0
    numerator[q, d] = (w_acc * numerator[q, d]
                       + w_block * O_block[q, d])
    if d == 0:
        normalizer[q] = w_acc * normalizer[q] + w_block
        m[q] = m_new


def csbp_collective_forward(Q, K_block, V_block,
                            K_clean_r, V_clean_r, masks, scale):
    # 1. Start the clean K/V all-gather.
    work, gathered = async_all_gather(K_clean_r, V_clean_r)

    # 2. One block-diagonal launch covers every block owned by this rank.
    O_block, LSE_block = block_diagonal_attention(
        Q[:masks.corrupted_rows], K_block, V_block, scale)

    # 3. Use the gathered clean K/Vs directly in one masked launch.
    work.wait()
    K_clean, V_clean = view_all_gathered_clean_kv(gathered)
    clean_mask = block_causal_mask(masks)
    O_clean, LSE_clean = masked_clean_attention(
        Q, K_clean, V_clean, clean_mask, scale)

    # 4. Custom CUDA kernels apply the LSE rule above in FP32.
    numerator = O_clean.float()
    m = LSE_clean.float()
    normalizer = isfinite(m).float()
    launch(merge_lse_kernel, numerator, m, normalizer,
           O_block, LSE_block)
    O, LSE = empty_outputs(Q)
    launch(normalize_output_kernel,
           numerator, m, normalizer, O, LSE)
    save_for_backward(Q, K_block, V_block, K_clean, V_clean, O, LSE)
    return O
```

Implementation 1: CSBP global-attention forward schedule and CUDA LSE merge.

Implementation 2 shows the global-attention fused packed backward. One masked kernel reuses the saved output and LSE to compute gradients for the local corrupted K/Vs and gathered clean K/Vs together. The corrupted gradients remain local, while only the clean K/V gradients are reduce-scattered to their CP ranks.

```
def csbp_fused_backward(dO, saved, masks, scale):
    Q, K_block, V_block, K_clean, V_clean, O, LSE = saved

    # 1. Pack local corrupted K/Vs before the gathered clean K/Vs.
    K_packed, V_packed = pack_local_clean_kv(
        K_block, V_block, K_clean, V_clean)

    # 2. One masked backward reuses the exact forward output and LSE.
    dQ, dK_packed, dV_packed = masked_attention_backward(
        Q, K_packed, V_packed, O, LSE, dO,
        masks.packed, scale)

    # 3. Split local gradients from the clean gradients that communicate.
    dK_block, dK_clean = split_local_clean_gradients(
        dK_packed, masks.corrupted_rows)
    dV_block, dV_clean = split_local_clean_gradients(
        dV_packed, masks.corrupted_rows)

    # 4. Return each rank's clean K/V-gradient shard.
    dK_clean_r, dV_clean_r = reduce_scatter_clean_gradients(
        dK_clean, dV_clean)
    return dQ, dK_block, dV_block, dK_clean_r, dV_clean_r
```

Implementation 2: Global-attention fused CSBP backward. One masked kernel computes the local corrupted-block and clean-context gradients.

D AR-TO-BDLM CONVERSION METHODOLOGY

Fast-dLLM v2 adapts a pretrained causal LM to block diffusion while retaining next-token prediction (Wu et al., 2025). It samples a blockwise binary mask $m^{(0)}$ and constructs a second noisy view with the complementary mask $m^{(1)} = 1 - m^{(0)}$ on every supervised token. If $h_i^{(v)}$ is the hidden representation at position i in view $v \in \{0, 1\}$, let $m_i^{(v)} = 1$ when x_i is masked and let $\mathcal{S}$ contain the supervised positions other than the first token. The sampled objective is

$$\widehat{\mathcal{L}}_{\text{AR-to-BDLM}} = -\sum_{v=0}^{1} \sum_{i \in \mathcal{S}} m_i^{(v)} \log p_\theta\left(x_i \mid h_{i-1}^{(v)}\right), \qquad m_i^{(0)} + m_i^{(1)} = 1. \tag{26}$$

The complementary views supervise every eligible next token exactly once. The one-token shift uses the preceding query row to predict x_i, matching a causal LM head.

The objective preserves the two properties used by block parallelism. Each view uses the length-$2L$ clean-plus-corrupted computation, and queries in block b use corrupted K/Vs only from block b of that view and clean K/Vs from preceding clean blocks,

$$O_{b,v} = \text{Attn}\big(Q_{b,v}, [K_{b,v}; K^{\text{clean}}_{<b,v}], [V_{b,v}; V^{\text{clean}}_{<b,v}]\big). \tag{27}$$

The two views are separate batch elements and introduce no cross-view attention. The label shift also leaves these attention dependencies unchanged. We assign each shifted loss term to the block containing its prediction row, and group the terms from both views into the corresponding block loss. The objective therefore retains the decomposition $\widehat{\mathcal{L}}_{\text{AR-to-BDLM}} = \sum_{b=1}^{B} \mathcal{L}_b$.

CSBP therefore applies directly. BP assigns each complete corrupted computation and loss for block b in both views to one rank, while CP shards the clean sequence of both views across the same ranks. Corrupted Q/K/V tensors and their gradients remain on the rank assigned the block, while clean K/V gradients are accumulated on the ranks storing their CP shards. Proposition 1 applies to each view, and summing the complementary-view losses preserves its loss and gradient equalities.

E DFLASH2 DRAFTER TRAINING METHODOLOGY

DFlash trains a lightweight block-diffusion drafter for speculative decoding (Chen et al., 2026). Given a clean sequence x, hidden features H^{tar} are extracted from a frozen autoregressive target model. For each sampled block b, the token at position a_b is visible and the next $M-1$ tokens are masked and predicted in one pass. Queries attend bidirectionally within the block and to target-feature K/Vs preceding a_b; different draft blocks do not attend to one another.

Let $\widetilde{x}^{(b)} = (x_{a_b}, [\texttt{MASK}], \ldots, [\texttt{MASK}])$ denote the input for block b. One draft-attention layer computes

$$O_b = \mathrm{Attn}\big(Q_b, [K_b; K^{\mathrm{tar}}_{<a_b}], [V_b; V^{\mathrm{tar}}_{<a_b}]\big)\,. \tag{28}$$

Following DFlash2 (Inco AI, 2026), our drafter uses the architecture in the open-source SpecForge implementation (Li et al., 2025). It retains this one-pass backbone and uses block-local two-tap dynamic convolutions and a candidate-path selector. Its sampled objective is the sum of a draft-token loss and a selector loss for each block:

$$\widehat{\mathcal{L}}_{\mathrm{DFlash2}} = \sum_{b\in\mathcal{S}} \mathcal{L}_b, \qquad \mathcal{L}_b = \frac{1}{Z}\left(\mathcal{L}_b^{\mathrm{draft}} + \alpha_{\mathrm{sel}}\mathcal{L}_b^{\mathrm{sel}}\right), \tag{29}$$

where $\mathcal{S}$ is the set of sampled draft blocks, Z is the sum of valid token weights, and α_{sel} weights the selector term. This is an instance of the blockwise objective in Equation 4.

This blockwise objective makes CSBP directly applicable. Let $\mathcal{B}_r$ be the blocks assigned to rank r; these sets are disjoint and together contain every sampled block. Since $\mathcal{L}_b$ depends only on block b and its target-feature prefix, each rank evaluates its assigned terms, and

$$\sum_{r=1}^{P} \nabla_\theta \left(\sum_{b\in\mathcal{B}_r} \mathcal{L}_b \right) = \nabla_\theta \widehat{\mathcal{L}}_{\mathrm{DFlash2}}. \tag{30}$$

Thus, assigning complete blocks to ranks preserves the loss and gradient up to floating-point reduction order. Let c_b denote the block-local drafter work for block b. Conventional CP carries every sampled block on every context rank, whereas CSBP partitions them across ranks:

$$W_{\mathrm{CP}}^{\mathrm{local}} = \sum_{r=1}^{P} \sum_{b\in\mathcal{S}} c_b = P \sum_{b\in\mathcal{S}} c_b, \tag{31}$$

$$W_{\mathrm{CSBP}}^{\mathrm{local}} = \sum_{r=1}^{P} \sum_{b\in\mathcal{B}_r} c_b = \sum_{b\in\mathcal{S}} c_b. \tag{32}$$

Each DFlash2 draft block has its own activations and loss, and draft blocks do not attend to one another. Target-feature K/Vs remain context sharded, but the computation on a block's draft activations is independent of every other draft block. This block-local path includes the decoder-side computation, output-head recomputation, vocabulary objective, and selector. Because the sets $\mathcal{B}_r$ partition $\mathcal{S}$, CSBP exploits this block independence through complete block ownership: it assigns the entire path to one rank, so the work counted by c_b is performed once rather than once per context rank.

At 512K, removing two-way replication reduces measured GEMM work by $1.92\times$, and higher execution efficiency raises the throughput gain to $2.48\times$. At 1M, removing eight-way replication permits four data-parallel replicas, producing the $7.59\times$ aggregate throughput gain. Unlike useful MFU, which excludes replicated computation, executed-GEMM utilization (all executed GEMM FLOP/s relative to BF16 peak) is 28.7%/37.0% at 512K and 35.8%/36.5% at 1M for baseline/CSBP. At 512K, CSBP cuts exposed attention communication from 105.4 to 7.5 ms per GPU because each block stays on one rank, avoiding communication of corrupted K/Vs and their gradients and explaining its higher utilization. The near-equal 1M values show that the combined gain comes from reduced executed work and increased data parallelism.

F ADDITIONAL EXPERIMENTAL SETUP

Workloads and matching. We evaluate SFT on NemotronDiffusion 3B, 8B, and 14B (Fu et al., 2026) and DiffusionGemma 26B-A4B (O'Donoghue & Flennerhag, 2026); AR-to-BDLM conversion

on the text backbones of Qwen3.5-27B (Qwen Team, 2026a) and Qwen3.8-27B (Qwen Team, 2026b) with the Fast-dLLM v2 objective (Wu et al., 2025); and long-context DFlash2 drafters for Qwen3.8-27B and Muse-Glimmer-30B using the SpecForge framework and open-source implementation (Inco AI, 2026; Li et al., 2026; 2025). In every comparison, the best baseline and CSBP use the same objective and target blocks. SFT and AR-to-BDLM conversion include every eligible target block; DFlash2 uses the blocks sampled by its training objective.

Hardware and configurations. BDLM fine-tuning and AR-to-BDLM conversion use two nodes with eight NVIDIA H200 GPUs per node and 141 GB of HBM3e per GPU. DFlash2 uses one node with eight NVIDIA H100 GPUs and 80 GB of HBM3 per GPU. For each setting, the best baseline is the highest-throughput feasible configuration tested without BP, selected over global batch size and data, tensor (Shoeybi et al., 2019), expert (Lepikhin et al., 2021), context (Liu et al., 2024; Jacobs et al., 2023), and sequence parallelism (Korthikanti et al., 2023). When CP is required, it uses the token-position sharding found in distributed-training libraries (NVIDIA, n.d.; Liang et al., 2025). We profile-optimize baseline execution, share applicable kernel and layout optimizations with CSBP, and tune CSBP topology and batch size independently.

Implementation and optimization. We use PyTorch (Paszke et al., 2019), FlashAttention-4 (Zadouri et al., 2026), FlexAttention (Dong et al., 2025), and NCCL in BF16. Our token-chunked linear cross-entropy loss uses cuBLAS projections and fused CUDA softmax/CE kernels, avoiding full-sequence logits materialization. BDLM fine-tuning and AR-to-BDLM conversion runs use activation checkpointing (Chen et al., 2016) and DeepSpeed ZeRO Stage 2 (Rajbhandari et al., 2020); the throughput-maximizing DFlash2 configuration uses fused AdamW without activation checkpointing.

Metrics and sequence lengths. We report useful MFU (Chowdhery et al., 2023) and maximum peak allocated HBM across GPUs. DFlash2 throughput and useful MFU count supervised draft-target tokens; computation replicated across context ranks consumes runtime without increasing either metric. Every throughput and memory profile uses sequences at exactly the reported context length. The equal-wall-clock experiments use the variable sequence lengths of their respective datasets.

G DiffusionGemma SFT Training Data and Evaluation Details

SWE-bench Verified. For the software-engineering comparison in Figure 5, the best baseline and CSBP train on identical ordered streams of successful CoderForge-Preview trajectories (Ariyak et al., 2026), formatted for the OpenHands agent interface (Wang et al., 2025). Trajectories in the training stream have a mean sequence length of 65,536.5 tokens and a maximum length of 131,072 tokens. We evaluate the base model at 0 hours and checkpoints at 3, 6, 9, and 12 hours, with one attempt on each of the 500 SWE-bench Verified tasks (Jimenez et al., 2024; Chowdhury et al., 2024). The harness supplies the issue and repository state; success requires both the designated `FAIL_TO_PASS` and `PASS_TO_PASS` tests to pass.

Terminal-Bench Lite. The best baseline and CSBP train on identical ordered streams of quality-filtered Terminus-2 trajectories from LiteCoder-Terminal-SFT (Peng et al., 2026). Trajectories in the training stream have a mean sequence length of 65,551 tokens and a maximum length of 131,072 tokens. We evaluate the same checkpoint schedule with Harbor's Terminus-2 agent (Harbor Framework Team, 2026) on all 100 OpenThoughts-TBLite tasks (OpenThoughts-Agent team, Snorkel AI, Bespoke Labs, 2026), with one attempt per task. A task passes only when its benchmark verifier returns success.

Matched wall-clock protocol. Within each workload, the best baseline and CSBP use the same model, training corpus, optimizer settings, checkpoint schedule, and evaluation configuration; only the distributed training method changes. We report pass rate as the percentage of tasks solved by the single attempt. Evaluation tasks and verifier outputs do not contribute training gradients. Both sources apply benchmark decontamination: CoderForge excludes matching SWE-bench Verified repository/commit pairs and problem statements, while LiteCoder-Terminal-SFT filters 13-gram overlaps with Terminal-Bench queries (Ariyak et al., 2026; Peng et al., 2026).

SFT optimization. The CoderForge-Preview and LiteCoder-Terminal-SFT runs use a global batch size of one example and one gradient-accumulation step per update. We train rank-16 LoRA adapters with $\alpha = 32$ on eight H100 GPUs. The baseline uses CP8, while CSBP uses CP8/BP8. We optimize with DeepSpeed ZeRO-2 and FusedAdam using a peak learning rate of 10^{-5}, Adam coefficients $(\beta_1, \beta_2) = (0.95, 0.99)$, $\epsilon = 10^{-8}$, weight decay 10^{-4}, gradient clipping at 1.0, and a cosine schedule that decays the learning rate to 10^{-6}.

H End-to-End Numerical Agreement

Table 3 compares the strongest non-BP baseline with CSBP using full-depth BF16 runs at 256K context on sixteen H200 GPUs. The NemotronDiffusion and DiffusionGemma rows evaluate BDLM fine-tuning, while the Qwen rows evaluate Fast-dLLM v2 AR-to-BDLM conversion. Each pair uses the same model, random seed, generated inputs, objective, and optimizer configuration. The resulting loss differences are small and consistent with floating-point reduction order.

Table 3: End-to-end loss agreement between the strongest non-BP baseline and CSBP at 256K context.

Model	Context	Best baseline topology	Ours topology	Absolute loss difference	Relative loss difference
NemotronDiffusion 3B	256K	DP4/TP1/CP4	DP4/TP1/CP4/BP4	2.83×10^{-3}	2.11×10^{-4}
NemotronDiffusion 8B	256K	DP4/TP1/CP4	DP4/TP1/CP4/BP4	2.59×10^{-3}	1.97×10^{-4}
NemotronDiffusion 14B	256K	DP2/TP1/CP8	DP2/TP1/CP8/BP8	9.54×10^{-7}	7.18×10^{-8}
DiffusionGemma 26B-A4B	256K	DP1/TP1/EP2/CP8	DP1/TP1/EP2/CP8/BP8	6.51×10^{-3}	2.11×10^{-4}
Qwen3.8-27B	256K	DP1/TP2/CP8	DP1/TP2/CP8/BP8	7.21×10^{-6}	5.11×10^{-7}
Qwen3.5-27B	256K	DP1/TP2/CP8	DP1/TP2/CP8/BP8	9.50×10^{-5}	6.69×10^{-6}

I Pure Block Parallelism Ablation

Sharding the clean context makes BP practical at long sequence lengths (Table 4). For NemotronDiffusion 14B at 64K, CSBP is $1.46\times$ faster than pure BP and uses 89.8 rather than 110.7 GiB of HBM. At 128K, CSBP still fits while pure BP runs out of memory. DiffusionGemma 26B-A4B shows the same need for context sharding even earlier: pure BP runs out of memory at both 64K and 128K, where CSBP fits and outperforms pure CP. Block ownership removes corrupted K/V communication; context sharding prevents replicated clean prefixes from erasing that benefit.

Table 4: Context sharding enables block parallelism at long contexts.

Model	Context	Method	Topology	Global batch	Tok/s	Peak HBM (GiB)
NemotronDiffusion 14B	64K	Best baseline	DP8/TP1/CP2	8	20,092	89.5
		Pure BP	DP8/TP1/BP2	8	15,223	110.7
		Ours	DP8/TP1/CP2/BP2	8	**22,152**	89.8
NemotronDiffusion 14B	128K	Best baseline	DP4/TP1/CP4	4	13,114	90.0
		Pure BP	DP4/TP1/BP4	4	OOM	OOM
		Ours	DP4/TP1/CP4/BP4	4	**15,136**	90.1
DiffusionGemma 26B-A4B	64K	Best baseline	DP4/TP1/EP2/CP2	8	22,154	109.4
		Pure BP	DP4/TP1/EP2/BP2	8	OOM	OOM
		Ours	DP4/TP1/EP2/CP2/BP2	8	**27,653**	113.5
DiffusionGemma 26B-A4B	128K	Best baseline	DP2/TP1/EP2/CP4	4	14,584	112.0
		Pure BP	DP2/TP1/EP2/BP4	4	OOM	OOM
		Ours	DP2/TP1/EP2/CP4/BP4	4	**21,049**	113.6

J Load Balancing Analysis

Load balancing delivers substantial gains without increasing memory use (Table 5). Replacing contiguous block assignment with early–late pairing improves throughput by $1.34\times$ for Nemotron-Diffusion 3B and $1.28\times$ for DiffusionGemma 26B-A4B at 256K. Peak HBM is unchanged for NemotronDiffusion and falls by 6.5 GiB for DiffusionGemma. Only the block assignment changes; the objective and remaining training settings are fixed. These gains show that equal block counts are not enough: balancing attention work across ranks is important to realizing CSBP's throughput benefit.

Table 5: Load balancing accelerates CSBP at 256K context.

Model	Topology	Global batch	ms/step	Tok/s	MFU (%)	Peak HBM (GiB)	Speedup
NemotronDiffusion 3B	DP4/TP1/CP4/BP4	4	88,950.4 / **66,500.3**	11,788 / **15,768**	27.8 / **37.2**	57.0 / 57.0	**1.34**×
DiffusionGemma 26B-A4B	DP1/TP1/EP2/CP8/BP8	2	43,722.0 / **34,134.6**	11,991 / **15,359**	14.4 / **18.4**	120.1 / **113.6**	**1.28**×

Pairs: contiguous / dual-end. Speedup: dual-end / contiguous. Bold marks the better value.

K AR-to-BDLM Conversion Results

CSBP's benefits extend to converting autoregressive models into BDLMs (Table 6). Under the Fast-dLLM v2 objective, both Qwen backbones train faster and use less peak HBM at every evaluated context length. Speedup grows from 1.26–$1.27\times$ at 64K to 1.27–$1.33\times$ at 256K, where CSBP also saves 9.6 GiB of HBM.

Table 6: CSBP accelerates AR-to-BDLM conversion from 64K to 256K context.

Model	Context	Best baseline topology	Ours topology	Tok/s	Speedup	MFU (%)	Peak HBM (GiB)
Qwen3.8-27B	64K	DP1/TP2/CP8	DP1/TP2/CP8/BP8	4,697 / **5,905**	**1.26**×	22.3 / **28.0**	64.2 / **62.5**
Qwen3.8-27B	128K	DP1/TP2/CP8	DP1/TP2/CP8/BP8	3,742 / **4,853**	**1.30**×	21.4 / **27.8**	83.0 / **78.3**
Qwen3.8-27B	256K	DP1/TP2/CP8	DP1/TP2/CP8/BP8	2,623 / **3,490**	**1.33**×	20.1 / **26.8**	120.7 / **111.1**
Qwen3.5-27B	64K	DP1/TP2/CP8	DP1/TP2/CP8/BP8	4,709 / **5,967**	**1.27**×	22.3 / **28.3**	64.2 / **62.5**
Qwen3.5-27B	128K	DP1/TP2/CP8	DP1/TP2/CP8/BP8	3,722 / **4,850**	**1.30**×	21.3 / **27.8**	83.0 / **78.3**
Qwen3.5-27B	256K	DP1/TP2/CP8	DP1/TP2/CP8/BP8	2,641 / **3,351**	**1.27**×	20.3 / **25.7**	120.7 / **111.1**

Pairs: best baseline / ours. Bold marks the better value.

L Effectiveness Across Different Block Sizes

CSBP maintains its advantage across an eightfold range of block sizes (Table 7). From 128 to 1,024 tokens per block, speedup stays within 1.14–$1.15\times$ for NemotronDiffusion 14B and 1.33–$1.44\times$ for DiffusionGemma 26B-A4B.

Table 7: CSBP sustains speedups across block sizes at 128K context.

Model	Block size	Best baseline topology	Ours topology	Global batch	ms/step	Tok/s	MFU (%)	Peak HBM (GiB)	Speedup
NemotronDiffusion 14B	128	DP4/TP1/CP4	DP4/TP1/CP4/BP4	4	39,187.3 / **34,279.4**	13,379 / **15,295**	34.3 / **39.2**	**90.0** / 90.1	**1.14**×
NemotronDiffusion 14B	256	DP4/TP1/CP4	DP4/TP1/CP4/BP4	4	39,979.2 / **34,637.4**	13,114 / **15,136**	33.7 / **38.9**	**90.0** / 90.1	**1.15**×
NemotronDiffusion 14B	512	DP4/TP1/CP4	DP4/TP1/CP4/BP4	4	38,852.1 / **34,197.3**	13,494 / **15,331**	34.7 / **39.4**	**90.0** / 90.1	**1.14**×
NemotronDiffusion 14B	1,024	DP4/TP1/CP4	DP4/TP1/CP4/BP4	4	39,201.0 / **34,392.4**	13,374 / **15,244**	34.5 / **39.3**	**90.0** / 90.1	**1.14**×
DiffusionGemma 26B-A4B	128	DP2/TP1/EP2/CP4	DP2/TP1/EP2/CP4/BP4	4	33,271.3 / **24,844.4**	15,758 / **21,103**	12.6 / **16.9**	**111.8** / 113.6	**1.34**×
DiffusionGemma 26B-A4B	256	DP2/TP1/EP2/CP4	DP2/TP1/EP2/CP4/BP4	4	35,949.5 / **24,907.5**	14,584 / **21,049**	11.7 / **16.9**	**112.0** / 113.6	**1.44**×
DiffusionGemma 26B-A4B	512	DP2/TP1/EP2/CP4	DP2/TP1/EP2/CP4/BP4	4	33,319.8 / **24,829.4**	15,735 / **21,116**	12.7 / **17.0**	**112.1** / 113.7	**1.34**×
DiffusionGemma 26B-A4B	1,024	DP2/TP1/EP2/CP4	DP2/TP1/EP2/CP4/BP4	4	33,033.5 / **24,850.7**	15,871 / **21,097**	12.9 / **17.1**	**112.3** / 113.7	**1.33**×

Pairs: best baseline / ours. Bold marks the better value.

M SAVINGS BREAKDOWNS

Table 8 separates attention communication from end-to-end step time. NemotronDiffusion 14B uses full attention, so keeping corrupted K/Vs local halves its logical attention traffic; overlap limits the reduction in exposed attention communication to 20.4%. DiffusionGemma 26B-A4B has an additional architecture-specific opportunity: 25 of its 30 attention layers use a 1,024-token sliding window. The CP8 baseline is the highest-throughput conventional CP configuration found by our profile-guided tuning procedure and uses the same applicable kernel and layout optimizations as CSBP. Its standard token-position sharding distributes every corrupted block across ranks and therefore does not preserve sliding-window locality in cross-rank communication. CSBP's complete-block ownership creates this locality and directly enables window-aware exchange. By aligning complete-block assignments with clean-sequence ownership, most required clean rows are local in the 25 sliding-window layers and only the remaining rows cross ranks; only the five global-attention layers exchange complete clean K/V shards. This reduces logical attention traffic by 93.5% and exposed attention communication by 93.4%.

Table 8: Attention and step-time breakdown at 256K: CP8 baseline vs. CSBP.

Measure	NemotronDiffusion 14B CP8 → CP8/BP8	DiffusionGemma 26B-A4B CP8 → CP8/BP8
Step time (s)	66.50 → 56.37 (1.18×)	49.66 → 34.13 (1.45×)
Forward GPU span (s)	17.57 → 13.00 (−26.0%)	10.99 → 5.83 (−47.0%)
Backward GPU span (s)	49.46 → 45.02 (−9.0%)	36.50 → 26.62 (−27.1%)
Attention traffic (GiB/GPU)	210.00 → 105.00 (−50.0%)	433.13 → 28.30 (−93.5%)
Exposed attention comm. (s/GPU)	3.034 → 2.414 (−20.4%)	10.939 → 0.723 (−93.4%)

The 93% reductions apply to attention communication, not the complete training step. DiffusionGemma's step time falls by 31.3%, while dense and expert GEMM durations remain nearly unchanged. Both forward and backward become faster, yielding 1.18× and 1.45× end-to-end speedup for NemotronDiffusion 14B and DiffusionGemma 26B-A4B, respectively.

N COMPLETE H200 PROFILES

CSBP improves the throughput–memory trade-off across the tested models and context lengths (Table 9). The independently selected configurations deliver speedups up to 1.61×. Even when throughput is nearly tied, context sharding can substantially reduce memory: for NemotronDiffusion 3B at 128K, CSBP stays within 1% of the DP16 baseline's throughput while saving 41 GiB of peak HBM. This case highlights another practical benefit of CSBP: retaining throughput while freeing memory.

Table 9: Throughput and memory gains across long-context training configurations.

Model	Context	Ours topology	Best baseline topology	Global batch	Tok/s	MFU (%)	Peak HBM (GiB)	HBM saved (GiB)	Speedup
NemotronDiffusion 3B	128K	DP8/TP1/CP2/BP2	DP16/TP1	8 / 16	28,891 / 29,086	37.5 / 37.7	56.5 / 97.5	41.0	0.99×
NemotronDiffusion 3B	256K	DP4/TP1/CP4/BP4	DP4/TP1/CP4	4	16,027 / 13,434	37.8 / 31.7	57.0 / 57.1	0.1	**1.19**×
NemotronDiffusion 3B	512K	DP2/TP1/CP8/BP8	DP2/TP1/CP8	2	8,352 / 6,895	37.4 / 30.8	58.1 / 61.7	3.6	**1.21**×
NemotronDiffusion 8B	64K	DP8/TP1/CP2/BP2	DP8/TP1/CP2	16	29,829 / 26,669	37.8 / 33.8	96.9 / 96.4	−0.5	**1.12**×
NemotronDiffusion 8B	128K	DP8/TP1/CP2/BP2	DP8/TP1/CP2	8	19,653 / 16,580	38.5 / 32.5	96.9 / 96.4	−0.5	**1.19**×
NemotronDiffusion 8B	256K	DP4/TP1/CP4/BP4	DP4/TP1/CP4	4	11,379 / 9,668	38.0 / 32.3	97.5 / 97.5	0.0	**1.18**×
NemotronDiffusion 8B	512K	DP2/TP1/CP8/BP8	DP2/TP1/CP8	2	6,254 / 5,202	38.2 / 31.8	98.5 / 99.5	1.0	**1.20**×
NemotronDiffusion 14B	64K	DP8/TP1/CP2/BP2	DP8/TP1/CP2	8	22,152 / 20,092	38.8 / 35.2	89.8 / 89.5	−0.3	**1.10**×
NemotronDiffusion 14B	128K	DP4/TP1/CP4/BP4	DP4/TP1/CP4	4	15,136 / 13,114	38.9 / 33.7	90.1 / 90.0	−0.1	**1.15**×
NemotronDiffusion 14B	256K	DP2/TP1/CP8/BP8	DP2/TP1/CP8	2	9,301 / 7,884	39.0 / 33.1	90.6 / 91.1	0.5	**1.18**×
NemotronDiffusion 14B	512K	DP1/TP2/CP8/BP8	DP1/TP2/CP8	1	5,052 / 4,242	37.7 / 31.6	71.9 / 72.7	0.8	**1.19**×
DiffusionGemma 26B-A4B	64K	DP4/TP1/EP2/CP2/BP2	DP4/TP1/EP2/CP2	8	27,653 / 22,154	16.0 / 12.8	113.5 / 109.4	−4.1	**1.25**×
DiffusionGemma 26B-A4B	128K	DP2/TP1/EP2/CP4/BP4	DP2/TP1/EP2/CP4	4	21,049 / 14,584	16.9 / 11.7	113.6 / 112.0	−1.6	**1.44**×
DiffusionGemma 26B-A4B	256K	DP1/TP1/EP2/CP8/BP8	DP1/TP1/EP2/CP8	2	15,359 / 10,557	18.4 / 12.7	113.6 / 118.1	4.6	**1.45**×
DiffusionGemma 26B-A4B	512K	DP1/TP2/CP8/BP8/SP	DP1/TP2/CP8/SP	1	9,815 / 6,079	20.3 / 12.6	106.5 / 127.4	20.9	**1.61**×

Pairs: ours / best baseline. HBM saved: best baseline minus ours.