Chipmunk: Training-Free Acceleration of Diffusion Transformers with Dynamic Column-Sparse Deltas

Anonymous Authors[1]

Abstract

Diffusion Transformers (DiTs) excel in generating high-quality images and videos but suffer from redundant computations at inference, increasing costs. Observing that only a small fraction (5-25%) of activations in attention and MLP layers account for 70-90% of the change across inference steps, we introduce Chipmunk, a dynamic sparsity method that recomputes only these rapidly changing activations while caching the remainder. Dynamic sparsity, however, poses system-level challenges, specifically GPU tensor core underutilization and additional runtime overhead from computing sparsity patterns and managing cached activations. To maximize GPU efficiency and approximation quality, Chipmunk employs voxel-based token reordering and efficient column-sparse kernels, achieving a 9.3x kernel speedup at 93% sparsity. Chipmunk also overlaps sparsity pattern computation and cache updates with ongoing computation to mask overhead latency. Chipmunk achieves up to 2.16x speedup on HunyuanVideo and 1.41x on FLUX.1-dev. Furthermore, we show that Chipmunk can be stacked on top of full step caching, achieving a 3.72x speedup on HunyuanVideo, a 2.67x speedup on WAN2.1, and a 2.56x speedup on FLUX.1-dev with minimal quality impact.

1. Introduction

Diffusion Transformers (DiTs) have emerged as state-of-the-art (SOTA) models for generating high-quality images and videos (Peebles & Xie, 2023; Zheng et al., 2024; Hong et al., 2022; Kong et al.; Labs, 2024; Chen et al., 2023). However, DiTs are increasingly constrained by their substantial computational requirements as sequence lengths and parameters scale up (Sun et al., 2024; Yao et al., 2024). For example, HunyuanVideo, with a 118k sequence length and 13B parameters, requires 18 minutes to generate a 5s video on an H100 GPU (Kong et al.). In particular, two major redundancies contribute significantly to unnecessary computation: the slow-changing latent vector (DiT model input) (Ma et al., 2024; Sun et al., 2024; Yuan et al., 2024) across generation steps, and the inherent sparsity in DiT activations (Li et al., 2022b; Liu et al., 2024b).

Existing approaches leverage these redundancies by caching per-step (Liu et al., 2024a), per-layer (Wimbauer et al., 2024), or per-token (Zou et al., 2025) outputs, or by computing sparse attention with static local window techniques (Zhang et al., 2025b; Yuan et al., 2024). In this paper, we ask whether we can achieve greater training-free acceleration by exploiting this redundancy *dynamically*, in the *most granular* manner possible.

We begin with an initial quantitative analysis of activation patterns in two state-of-the-art open source DiTs (HunyuanVideo and FLUX.1-dev). We find that over 90% of the variance in cross-step attention activation changes is explained by only 5-25% of the intermediate activation values (Table 1). That is, recomputing only the top 5-25% of attention interactions and reusing the rest from the previous step captures over 90% of the cross-step change. For MLPs, we find 15-25% of the intermediate activation values explain over 70% of the variance (Table 1).

Chipmunk. This finding motivates our method, Chipmunk, which caches intermediate activations and uses dynamic sparsity to recompute only the fastest-changing activations. We exploit a common computational structure of attention and MLPs–act$(a \mathbin{@} b) \mathbin{@} c$–both using a back-to-back matrix multiply to produce a linear combination of vectors in c (Section 2). Sparsity on the intermediate activations of both operations corresponds one-to-one with these individual vector contributions. Specifically, Chipmunk uses the magnitudes of intermediate activations to determine which vectors of c to recompute at each step, and uses cached values of the rest of the vectors from the previous step. For attention, Chipmunk chooses the top-k *column-chunks* of the attention matrix to recompute, and for MLP, Chipmunk computes an approximate difference against cached acti-

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

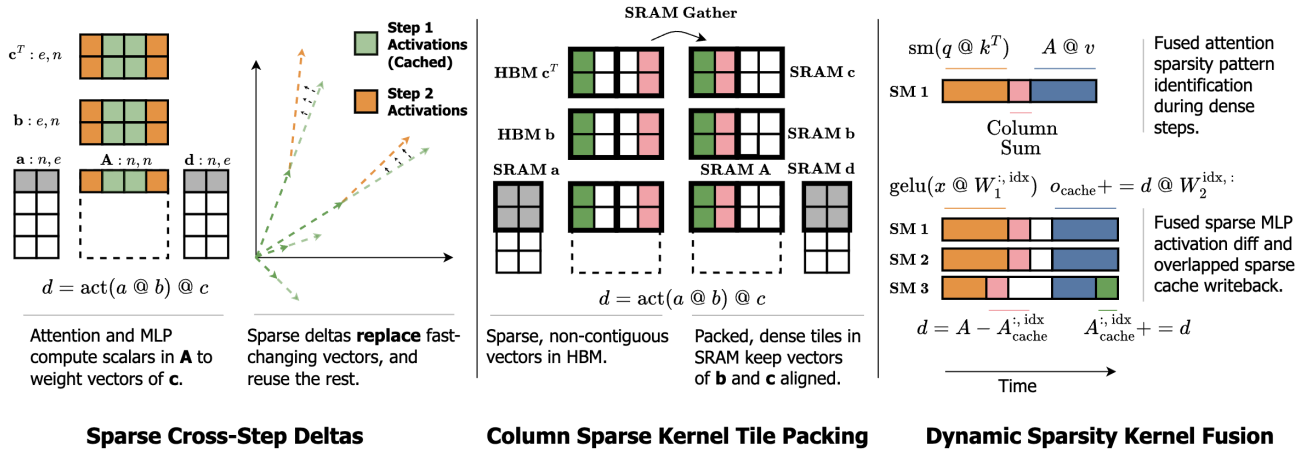

Figure 1. Left: Chipmunk uses intermediate activation sparsity to recompute only the fastest changing vectors in the output linear combinations of attention and MLPs. *Middle:* Column sparse kernels achieve low approximation error and hardware-efficiency by packing dense SRAM tiles for peak tensor core utilization. *Right:* Extra operations required to compute dynamic sparse deltas, such as sparsity pattern identification and cache updates, are fused with other operations to minimize overhead.

vations to identify the top-k column-chunks to recompute. This can be interpreted intuitively as computing sparse deviations from straight line paths in latent space (Fig. 1, left), where the recomputed vectors are dynamically allocated more sequential computation steps.

Systems Challenges. Dynamic sparsity introduces two systems challenges: (1) Sparse attention and MLP operations tend to underutilize tensor cores, which can make it challenging to achieve wall-clock speedup relative to dense baselines. (2) Computing dynamic sparsity patterns at runtime and caching activations introduces additional overhead.

To address these challenges, Chipmunk uses a voxel-based ordering of the pixels (Fig. 5, left) to regroup spatiotemporally local tokens into *column-wise* sparsity patterns (Fig. 1, middle). In particular, a contiguous group of tokens activates the *same* sparse set of individual keys/values (attention) or weights (MLPs). This corresponds to selecting columns of K^T/W_1 and rows of V/W_2.

This column-wise sparsity pattern admits efficient attention and MLP kernels using sparse gathers from GPU global memory (HBM) to packed dense tiles in GPU shared memory (SRAM), which can then fully utilize GPU tensor cores (Fig. 1, middle) (NVIDIA, 2024; Chen et al., 2021b; Li et al., 2022a; Ye et al., 2025). To reduce the overhead of computing dynamic sparsity patterns and maintaining an activation cache, Chipmunk uses custom kernels to overlap sparsity pattern identification and cache I/O operations with other computations (Fig. 1, right). Chipmunk's kernels scale linearly with sparsity with little overhead, achieving nearly-optimal speedup while providing 2x less approximation error than block sparsity. At 93% sparsity, our column-sparse attention kernel is 9.3x faster than FlashAttention-3 in the ThunderKittens library (Shah et al., 2024; Spector et al., 2024).

Evaluation. We evaluate Chipmunk on state-of-the-art text-to-video generation models (HunyuanVideo and WAN2.1) and one state-of-the-art text-to-image generation model (FLUX.1-dev).

- **Quality.** Chipmunk attains up to 92% attention sparsity across 44 of 50 generation steps on HunyuanVideo with minimal impact to VBench scores. On FLUX.1-dev, Chipmunk achieves 84% attention sparsity and 70% MLP sparsity for 44 of 50 steps without affecting ImageReward and CLIP scores.
- **Fast Generation.** Chipmunk alone achieves end-to-end generation speedups of 2.16x on HunyuanVideo and 1.41x on FLUX.1-dev.
- **Stacked Acceleration.** Chipmunk natively stacks with existing caching strategies, such as step caching, leading to further acceleration—achieving speedups of 3.72x on HunyuanVideo, 2.67x on WAN2.1, and 2.56x on FLUX.1-dev.

2. Background

Here we summarize the core aspects of Diffusion Transformers (DiTs) and GPU architectures relevant to Chipmunk. The key points are: (1) DiT inference computes latent-space paths from noise to output, and (2) GPU efficiency hinges on saturating tensor cores with large matrix multiplications.

Table 1. R^2 scores between true cross-step activation change and approximated change.

	Attention Active	Explained Change	MLP Active	Explained Change
HunyuanVideo	5%	92.4%	25%	70.8%
FLUX.1-dev	25%	90.7%	15%	69.0%

Diffusion Transformers (DiTs). Diffusion models reconstruct data from noise by reversing a forward noising process through iterative denoising steps with neural networks (Sohl-Dickstein et al., 2015). DiTs apply Vision Transformer architectures (Dosovitskiy et al., 2021) to diffusion tasks, modulating attention and MLP outputs based on timesteps and prompts (Peebles & Xie, 2023). DiTs represent images as token vectors derived from pixel patches, iteratively refining these latent vectors during inference. While single-step inference follows a straight line, multi-step inference paths adjust their direction iteratively, significantly increasing computational cost but improving generation quality (Song et al., 2023; Esser et al., 2024). A final important piece of the DiT architecture is the shared computational form of attention and MLP layers: both compute non-linear scalar coefficients to produce a linear combination of vectors (scaled rows of V in attention, and scaled rows of W_2 in MLPs).

GPU Architecture. GPUs accelerate attention and MLPs in DiTs with efficient matrix multiplications executed on Streaming Multiprocessors (SMs) with dedicated tensor cores and local SRAM (NVIDIA, 2024). Both kernels load data blocks from global memory to SRAM, perform matrix multiplications via tensor cores, and store results back in global memory. FlashAttention fuses both GEMMs of attention to minimize memory transfers (Dao et al., 2022). Achieving peak GPU performance demands keeping tensor cores continually supplied with large matrix tiles (at least 64x64), maximizing their utilization and throughput (Luo et al., 2024; Spector et al., 2024).

3. Method

We first quantify sparse activation changes across diffusion steps, motivating Chipmunk's unified caching and sparse recomputation strategy; next, we show column sparsity's hardware efficiency and low error; finally, we describe Chipmunk's algorithm that interleaves dense and sparse computation, stacking natively with coarser caching methods.

3.1. Motivation: Sparse Cross-Step Changes in Attention and MLPs

DiT inference shows sparse, slow-changing activations across diffusion steps. A natural question arises: how much change between steps can be captured by sparse recomputation? Formally, considering attention and MLP layers at consecutive steps $i = 1, 2$, outputs can be approximated using a binary mask M to identify rapidly changing activations:

$$\begin{aligned} \mathbf{o}_2^{\text{attn}} \approx \mathbf{o}_1^{\text{attn}} &- \left[\text{softmax}(q_1\, k_1^\top) *\ M\right] v_1 \\ &+ \left[\text{softmax}(q_2\, k_2^\top) *\ M\right] v_2, \\ \mathbf{o}_2^{\text{mlp}} \approx \mathbf{o}_1^{\text{mlp}} &- \left[\text{gelu}(x_1\, W_1) *\ M\right] W_2 \\ &+ \left[\text{gelu}(x_2\, W_1) *\ M\right] W_2 \end{aligned} \tag{1}$$

Empirical results (Table 1) indicate that recomputing only 5-25% of the fastest-changing activations explains most activation changes across steps. Motivated by this, Chipmunk employs unified caching and sparse recomputation strategies for both attention and MLP layers, recomputing only rapidly changing vectors.

3.2. Hardware-Aware Sparse Deltas

Chipmunk employs column sparsity to efficiently map sparse computations into compacted dense computations while maintaining low approximation error (Fig. 5, Table 4). While block sparsity provides high hardware efficiency by skipping entire tiles during computation, it suffers from approximately double the approximation error compared to column sparsity (Table 4, Fig. 5). Chipmunk achieves efficient computation with column sparsity by using sparse gathers that pack sparse activations from high-bandwidth memory (HBM) into dense shared memory (SRAM) tiles, thus maintaining high tensor core utilization and competitive speedups (Fig. 1). Column sparsity uses a granularity of $[C, 1]$, meaning each token chunk of size C selectively activates individual keys/values or neurons, as opposed to MoE models that route individual tokens to entire contiguous expert chunks. Lastly, reordering tokens into video voxel or image patch chunks further improves the approximation quality of sparse attention by exploiting local token similarities, achieving a 1.2x reduction in unexplained variance (Fig. 5). Please see Appendix B for a full description of Chipmunk's kernels.

3.3. Chipmunk Algorithm

Chipmunk interleaves three types of steps: dense, sparse, and skipped.

Dense Steps: Refresh cached activations and dynamically identify sparsity patterns.

Method	Efficiency		VBench Dimension		
	Speedup ↑	Latency (s) ↓	Total ↑	Quality ↑	Semantic ↑
HunyuanVideo, T=50 (720 x 1280 x 129)					
Hunyuan	1x	1030s	83.24	85.09	75.82
STA	1.79x	575s	82.46	**84.63**	73.83
Chipmunk	**2.16x**	**477s**	**82.94**	84.60	**76.3**
Step Caching (TeaCache)	3.69x	279s	80.79	82.87	72.5
Chipmunk+Step Cache	**3.72x**	**277s**	**82.5**	**84.23**	**75.6**
WAN2.1, T=50 (720 x 1280 x 121)					
WAN2.1	1x	1357s	81.47	83.57	73.08
Step Caching (TeaCache)	2.0x	678s	81.17	83.24	72.87
Chipmunk-56+Step Cache	2.20x	616s	**81.73**	**83.74**	73.69
Chipmunk-73+Step Cache	**2.67x**	**508s**	81.11	82.88	**74.05**

Table 2. Performance comparison of various methods across different datasets for video generation. *Note*: Chipmunk-X denotes a sparsity level of $X\%$ to assess the speed-quality tradeoff.

Sparse Steps: Compute sparse activation deltas based on cached activations, recomputing only the most significantly changing vectors.

Skipped Steps: Fully reuse cached outputs without additional computation.

Chipmunk is parameterized by a step schedule, attention sparsity, and MLP sparsity levels, complementing coarser caching techniques like step- and token-level caching. Please see Appendix C for a full description of the algorithm.

4. Experiments

We evaluate Chipmunk on state-of-the-art text-to-video and text-to-image tasks, summarizing our setup (D.1), quantitative performance (Tables 2, 6; D.2), and qualitative results (Fig. 9; D.3).

4.1. Setup

We assess Chipmunk using three DiT models: HunyuanVideo, WAN2.1 (text-to-video), and FLUX.1-dev (text-to-image), all at default 50-generation steps on H100-SXM5 GPUs. Baselines include TeaCache (Liu et al., 2024a), ToCa (Zou et al., 2025), STA (Zhang et al., 2025b), and DiTFastAttn (Yuan et al., 2024). We use standard metrics: VBench for videos and ImageReward (ImRe) and CLIP for images. Chipmunk's hyperparameters (attention/MLP sparsity and step schedule) are chosen to capture 95% of cross-step activation changes.

4.2. Quantitative Results

Chipmunk achieves superior efficiency-quality tradeoffs across tasks (Table 2). For equivalent acceleration, Chipmunk preserves higher quality compared to TeaCache due to granular vector caching. Holding quality constant, Chipmunk outperforms STA, ToCa, and DiTFastAttn in speed due to granular, dynamic sparsity identification. Sparsity adjustments can also modulate Chipmunk's speed-quality balance.

4.3. Qualitative Results

Qualitatively, Chipmunk maintains detailed visual fidelity, effectively preserving critical elements like moving hands or intricate prompt details (Fig. 9). Failure modes include slightly unfocused backgrounds in videos and small differences in image details compared to references, reflecting the FLOPs reduction.

5. Conclusion

We introduce Chipmunk, a training-free approach that accelerates DiT inference by dynamically exploiting the inherent sparsity and slow-changing patterns of intermediate activations. By leveraging the shared computational structure of attention and MLPs, Chipmunk selectively recomputes only rapidly changing activations, caching the remainder for reuse, significantly reducing computational load with minimal impact on visual quality. Chipmunk's speedups are most pronounced in large, compute-bound DiTs, while smaller models or those with compact matrix shapes may see less benefit due to overhead from managing sparsity. Models with fewer inference steps also yield modest acceleration, as caching opportunities diminish with larger activation changes. Future work should explore integrating sparse recomputation during training to allow models to dynamically target the fastest-changing activations and investigate combining Chipmunk with per-layer and per-token caching methods.

References

Arora, S., Eyuboglu, S., Zhang, M., Timalsina, A., Alberti, S., Zinsley, D., Zou, J., Rudra, A., and Ré, C. Simple linear attention language models balance the recall-throughput tradeoff, 2025. URL https://arxiv.org/abs/2402.18668.

Chen, B., Dao, T., Winsor, E., Song, Z., Rudra, A., and Ré, C. Scatterbrain: Unifying sparse and low-rank attention approximation, 2021a. URL https://arxiv.org/abs/2110.15343.

Chen, J., Yu, J., Ge, C., Yao, L., Xie, E., Wu, Y., Wang, Z., Kwok, J., Luo, P., Lu, H., and Li, Z. Pixart-α: Fast training of diffusion transformer for photorealistic text-to-image synthesis, 2023. URL https://iclr.cc/media/iclr-2024/Slides/18231.pdf.

Chen, Z., Qu, Z., Liu, L., Ding, Y., and Xie, Y. Efficient tensor core-based gpu kernels for structured sparsity under reduced precision. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, 2021b. Association for Computing Machinery. ISBN 9781450384421. doi: 10.1145/3458817.3476182. URL https://doi.org/10.1145/3458817.3476182.

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T., Hawkins, P., Davis, J., Mohiuddin, A., Kaiser, L., Belanger, D., Colwell, L., and Weller, A. Rethinking attention with performers, 2022. URL https://arxiv.org/abs/2009.14794.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale, 2021. URL https://iclr.cc/virtual/2021/poster/3013.

Elhage, N., Nanda, N., Olsson, C., Henighan, T., Joseph, N., Mann, B., Askell, A., Bai, Y., Chen, A., Conerly, T., DasSarma, N., Drain, D., Ganguli, D., Hatfield-Dodds, Z., Hernandez, D., Jones, A., Kernion, J., Lovitt, L., Ndousse, K., Amodei, D., Brown, T., Clark, J., Kaplan, J., McCandlish, S., and Olah, C. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

Esser, P., Kulal, S., Blattmann, A., Entezari, R., Müller, J., Saini, H., Levi, Y., Lorenz, D., Sauer, A., Boesel, F., Podell, D., Dockhorn, T., English, Z., Lacey, K., Goodwin, A., Marek, Y., and Rombach, R. Scaling rectified flow transformers for high-resolution image synthesis, 2024. URL https://proceedings.mlr.press/v235/esser24a.html.

Fedus, W., Zoph, B., and Shazeer, N. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity, 2022. URL https://arxiv.org/abs/2101.03961.

Ho, J., Jain, A., and Abbeel, P. Denoising diffusion probabilistic models, 2020. URL https://arxiv.org/abs/2006.11239.

Hong, W., Ding, M., Zheng, W., Liu, X., and Tang, J. Cogvideo: Large-scale pretraining for text-to-video generation via transformers, 2022. URL https://arxiv.org/abs/2205.15868.

Huang, Z., He, Y., Yu, J., Zhang, F., Si, C., Jiang, Y., Zhang, Y., Wu, T., Jin, Q., Chanpaisit, N., Wang, Y., Chen, X., Wang, L., Lin, D., Qiao, Y., and Liu, Z. Vbench: Comprehensive benchmark suite for video generative models, 2023. URL https://openaccess.thecvf.com/content/CVPR2024/html/Huang_VBench_Comprehensive_Benchmark_Suite_for_Video_Generative_Models_CVPR_2024_paper.html.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mixtral of experts, 2024. URL https://arxiv.org/abs/2401.04088.

Kitaev, N., Łukasz Kaiser, and Levskaya, A. Reformer: The efficient transformer, 2020. URL https://arxiv.org/abs/2001.04451.

Kong, W., Tian, Q., Zhang, Z., Min, R., Dai, Z., Zhou, J., Xiong, J., Li, X., Wu, B., Zhang, J., et al. Hunyuanvideo: A systematic framework for large video generative models. *arXiv preprint arXiv:2412.03603*. URL https://ui.adsabs.harvard.edu/abs/2024arXiv241203603K/abstract.

Labs, B. F. Flux. https://github.com/black-forest-labs/flux, 2024.

Li, S., Osawa, K., and Hoefler, T. Efficient quantized sparse matrix operations on tensor cores. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15. IEEE, November 2022a. doi: 10.1109/sc41404.2022.00042. URL http://dx.doi.org/10.1109/SC41404.2022.00042.

Li, Z., You, C., Bhojanapalli, S., Li, D., Rawat, A. S., Reddi, S. J., Ye, K., Chern, F., Yu, F., Guo, R., et al. The lazy neuron phenomenon: On emergence of activation sparsity in transformers. *arXiv preprint arXiv:2210.06313*, 2022b.

Liu, F., Zhang, S., Wang, X., Wei, Y., Qiu, H., Zhao, Y., Zhang, Y., Ye, Q., and Wan, F. Timestep embedding tells: It's time to cache for video diffusion model. *arXiv preprint arXiv:2411.19108*, 2024a.

Liu, J., Ponnusamy, P., Cai, T., Guo, H., Kim, Y., and Athiwaratkun, B. Training-free activation sparsity in large language models. *arXiv preprint arXiv:2408.14690*, 2024b.

Liu, X., Gong, C., and Liu, Q. Flow straight and fast: Learning to generate and transfer data with rectified flow, 2022. URL https://iclr.cc/virtual/2023/oral/12626.

Liu, Z., Wang, J., Dao, T., Zhou, T., Yuan, B., Song, Z., Shrivastava, A., Zhang, C., Tian, Y., Re, C., and Chen, B. Deja vu: Contextual sparsity for efficient llms at inference time, 2023. URL https://arxiv.org/abs/2310.17157.

Lu, C., Zhou, Y., Bao, F., Chen, J., Li, C., and Zhu, J. Dpm-solver: A fast ode solver for diffusion probabilistic model sampling in around 10 steps, 2022. URL https://arxiv.org/abs/2206.00927.

Luo, W., Fan, R., Li, Z., Du, D., Wang, Q., and Chu, X. Benchmarking and dissecting the nvidia hopper gpu architecture, 2024. URL https://ieeexplore.ieee.org/document/10579250.

Ma, X., Fang, G., and Wang, X. Deepcache: Accelerating diffusion models for free. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pp. 15762–15772, 2024.

NVIDIA. Gpu performance background user's guide, 2024. URL https://docs.nvidia.com/deeplearning/performance/dl-performance-gpu-background/index.html.

Peebles, W. and Xie, S. Scalable diffusion models with transformers, 2023. URL https://openaccess.thecvf.com/content/ICCV2023/html/Peebles_Scalable_Diffusion_Models_with_Transformers_ICCV_2023_paper.html.

Radford, A., Kim, J. W., Hallacy, C., Ramesh, A., Goh, G., Agarwal, S., Sastry, G., Askell, A., Mishkin, P., Clark, J., Krueger, G., and Sutskever, I. Learning transferable visual models from natural language supervision, 2021. URL https://arxiv.org/abs/2103.00020.

Salimans, T. and Ho, J. Progressive distillation for fast sampling of diffusion models, 2022. URL https://arxiv.org/abs/2202.00512.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL https://tridao.me/blog/2024/flash3/.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer, 2017. URL https://arxiv.org/abs/1701.06538.

Sohl-Dickstein, J., Weiss, E. A., Maheswaranathan, N., and Ganguli, S. Deep unsupervised learning using nonequilibrium thermodynamics, 2015. URL https://proceedings.mlr.press/v37/sohl-dickstein15.html.

Song, J., Meng, C., and Ermon, S. Denoising diffusion implicit models, 2022. URL https://arxiv.org/abs/2010.02502.

Song, Y., Dhariwal, P., Chen, M., and Sutskever, I. Consistency models, 2023. URL https://proceedings.mlr.press/v202/song23a.html.

Spector, B. F., Arora, S., Singhal, A., Fu, D. Y., and Ré, C. Thunderkittens: Simple, fast, and adorable ai kernels, 2024. URL https://iclr.cc/virtual/2025/poster/31243.

Sun, X., Fang, J., Li, A., and Pan, J. Unveiling redundancy in diffusion transformers (dits): A systematic study, 2024. URL https://arxiv.org/abs/2411.13588.

Tillet, P., Kung, H. T., and Cox, D. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973. URL https://doi.org/10.1145/3315508.3329973.

Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. Linformer: Self-attention with linear complexity, 2020. URL https://arxiv.org/abs/2006.04768.

Wimbauer, F., Wu, B., Schoenfeld, E., Dai, X., Hou, J., He, Z., Sanakoyeu, A., Zhang, P., Tsai, S., Kohler, J., Rupprecht, C., Cremers, D., Vajda, P., and Wang, J. Cache me if you can: Accelerating diffusion models through block caching, 2024. URL https://arxiv.org/abs/2312.03209.

Xu, J., Liu, X., Wu, Y., Tong, Y., Li, Q., Ding, M., Tang, J., and Dong, Y. Imagereward: Learning and evaluating human preferences for text-to-image generation. In Oh, A., Naumann, T., Globerson, A., Saenko, K., Hardt, M., and Levine, S. (eds.), *Advances in Neural Information Processing Systems*, volume 36, pp. 15903–15935. Curran Associates, Inc., 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/33646ef0ed554145eab65f6250fab0c9-Paper-Conference.pdf.

Yao, J., Cheng, W., Liu, W., and Wang, X. Fasterdit: Towards faster diffusion transformers training without architecture modification, 2024. URL https://arxiv.org/abs/2410.10356.

Ye, Z., Chen, L., Lai, R., Lin, W., Zhang, Y., Wang, S., Chen, T., Kasikci, B., Grover, V., Krishnamurthy, A., et al. Flashinfer: Efficient and customizable attention engine for llm inference serving. *arXiv preprint arXiv:2501.01005*, 2025.

Yuan, Z., Zhang, H., Lu, P., Ning, X., Zhang, L., Zhao, T., Yan, S., Dai, G., and Wang, Y. Ditfastattn: Attention compression for diffusion transformer models, 2024. URL https://proceedings.neurips.cc/paper_files/paper/2024/hash/0267925e3c276e79189251585b4100bf-Abstract-Conference.html.

Zaheer, M., Guruganesh, G., Dubey, A., Ainslie, J., Alberti, C., Ontanon, S., Pham, P., Ravula, A., Wang, Q., Yang, L., and Ahmed, A. Big bird: Transformers for longer sequences, 2021. URL https://arxiv.org/abs/2007.14062.

Zhang, J., wei, J., Huang, H., Zhang, P., Zhu, J., and Chen, J. Sageattention: Accurate 8-bit attention for plug-and-play inference acceleration, 2025a. URL https://arxiv.org/abs/2410.02367.

Zhang, P., Chen, Y., Su, R., Ding, H., Stoica, I., Liu, Z., and Zhang, H. Fast video generation with sliding tile attention, 2025b. URL https://arxiv.org/abs/2502.04507.

Zheng, Z., Peng, X., Yang, T., Shen, C., Li, S., Liu, H., Zhou, Y., Li, T., and You, Y. Open-sora: Democratizing efficient video production for all, 2024. URL https://arxiv.org/abs/2412.20404.

Zou, C., Liu, X., Liu, T., Huang, S., and Zhang, L. Accelerating diffusion transformers with token-wise feature caching, 2025. URL https://iclr.cc/virtual/2025/poster/27718.

A. Extended Discussion of Latent Space Path Decompositions

In this appendix, we formally demonstrate that the latent space paths of rectified flow Diffusion Transformers (DiTs) can be decomposed into a sum of individually scaled vectors output by the attention and MLP layers (Fig. 2). This decomposition does not claim that each individual vector evolves along an independent path (since the vectors interact via non-linearities) – rather, we claim that the final latent space path comprises only scaled and shifted vectors output by attention and MLP layers. We assume that in each step of the diffusion process, the DiT output is scaled according to the noise schedule and added to the latent vector representation, as in the architectures of HunyuanVideo and FLUX.1-dev (Esser et al., 2024).

Attention and MLP Outputs as Scaled Vector Additions. We begin by explicitly defining attention and MLP layers within a DiT residual block.

Attention: Given query $q \in \mathbb{R}^{n\times d_c}$, key $k \in \mathbb{R}^{n\times d_c}$, and value $v \in \mathbb{R}^{n\times d_c}$ matrices projected from the residual stream $x \in \mathbb{R}^{n\times d}$, attention computes:

$$\text{Attention}(q,k,v) = \text{softmax}\left(\frac{qk^\top}{\sqrt{d_c}}\right)v \tag{2}$$

To explicitly show the decomposition into individual vectors, we rewrite attention per-head and per-token. Consider a single attention head h with projection matrices $W_q^h, W_k^h, W_v^h \in \mathbb{R}^{d\times d_c}$, and output projection $W_o^h \in \mathbb{R}^{d_c\times d}$. For each token i, attention can be expressed as a linear combination of dynamically computed vectors from the combined value-output projection:

$$\text{Attn}_h(x_i) = \sum_j \alpha_{ij}^h \cdot (x_j W_v^h W_o^h), \tag{3}$$

$$\text{where} \quad \alpha_{ij}^h = \text{softmax}_j\left(\frac{(x_i W_q^h)(xW_k^h)^\top}{\sqrt{d_c}}\right) \tag{4}$$

This shows that attention produces a weighted sum of vectors $(x_j W_v^h W_o^h)$, with scalars α_{ij}^h computed via softmax (Fig. 3, left). Each individual vector is thus scaled dynamically based on token interactions.

MLP: Similarly, given weight matrices $W_1 \in \mathbb{R}^{d\times d_f}$, $W_2 \in \mathbb{R}^{d_f\times d}$ and biases $b_1 \in \mathbb{R}^{d_f}$, $b_2 \in \mathbb{R}^d$, the MLP operation computes:

$$\text{MLP}(x) = \text{GELU}(xW_1 + b_1)W_2 + b_2 \tag{5}$$

The MLP can be viewed as outputting scaled rows of W_2 (the bias is a single extra static vector). Specifically, for token x_i:

$$\text{MLP}(x_i) = \sum_{m=1}^{d_f} \text{GELU}(x_i W_1^{:,m} + b_1)\cdot W_2^{m,:} + b_2 \tag{6}$$

Thus, MLP outputs scaled rows from W_2, with scalar coefficients computed as GELU applied to the dot product of x_i with each column of W_1 (plus a static shift from the bias).

Distributive Operations. Having established attention and MLP as explicitly adding scaled vectors, we now formally discuss operations distributing over vector addition within DiTs.

Modulations: Element-wise multiplication by modulation factors $m \in \mathbb{R}^d$ distributes trivially:

$$m \odot \sum_i v_i = \sum_i (m \odot v_i) \tag{7}$$

Layernorm: For vectors v_i, layer normalization is defined element-wise:

$$\text{LN}\left(\sum_i v_i\right) = \sum_i \frac{v_i - \mathbb{E}[\sum_i v_i]}{\sqrt{\text{Var}[\sum_i v_i]}}\gamma + \beta \tag{8}$$

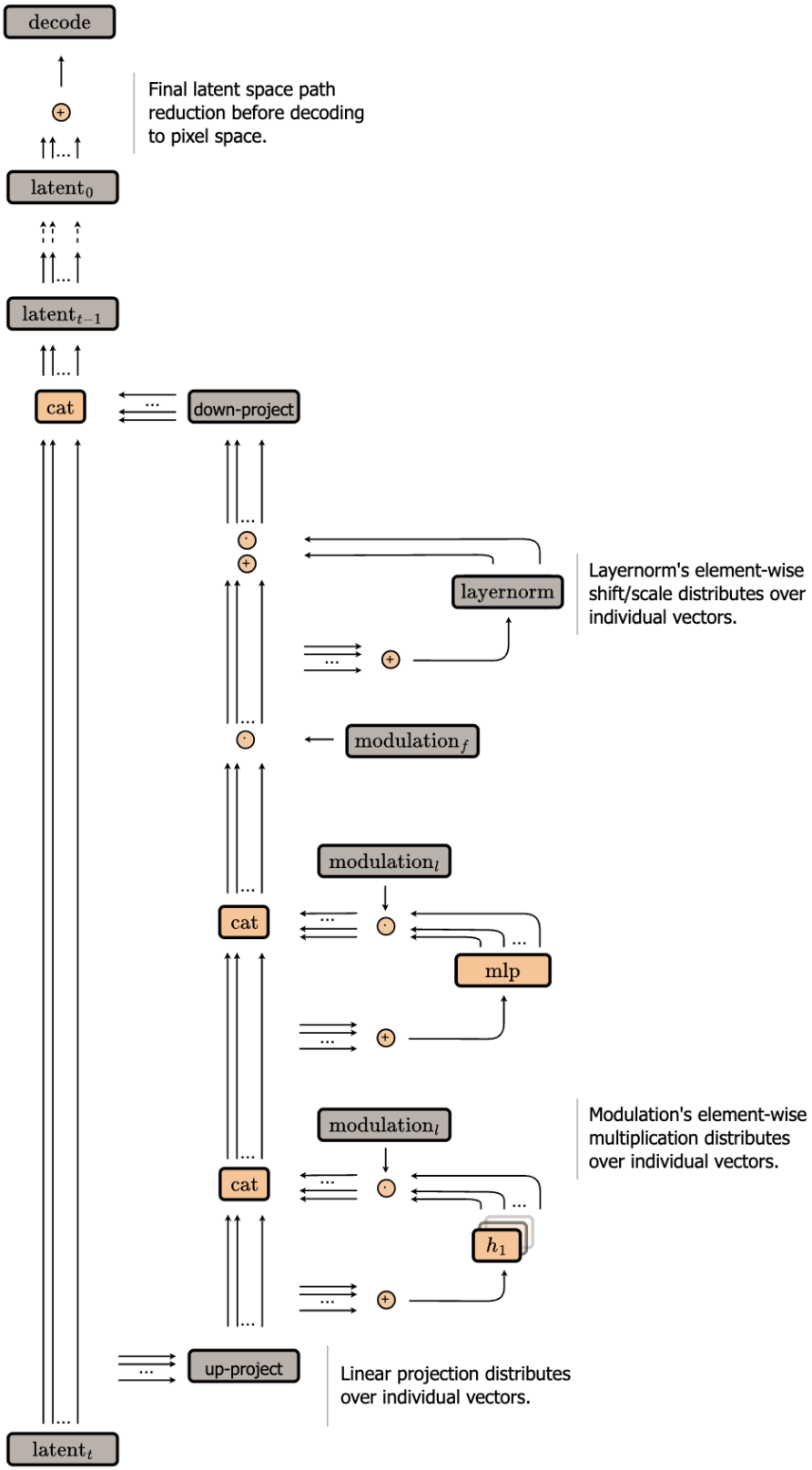

Figure 2. An individual-vector interpetation of the full HunyuanVideo DiT architecture. Attention and MLPs add new vectors to the residual stream, with downstream modulation, layernorm, and linear projection operations distributing over the vector sums.

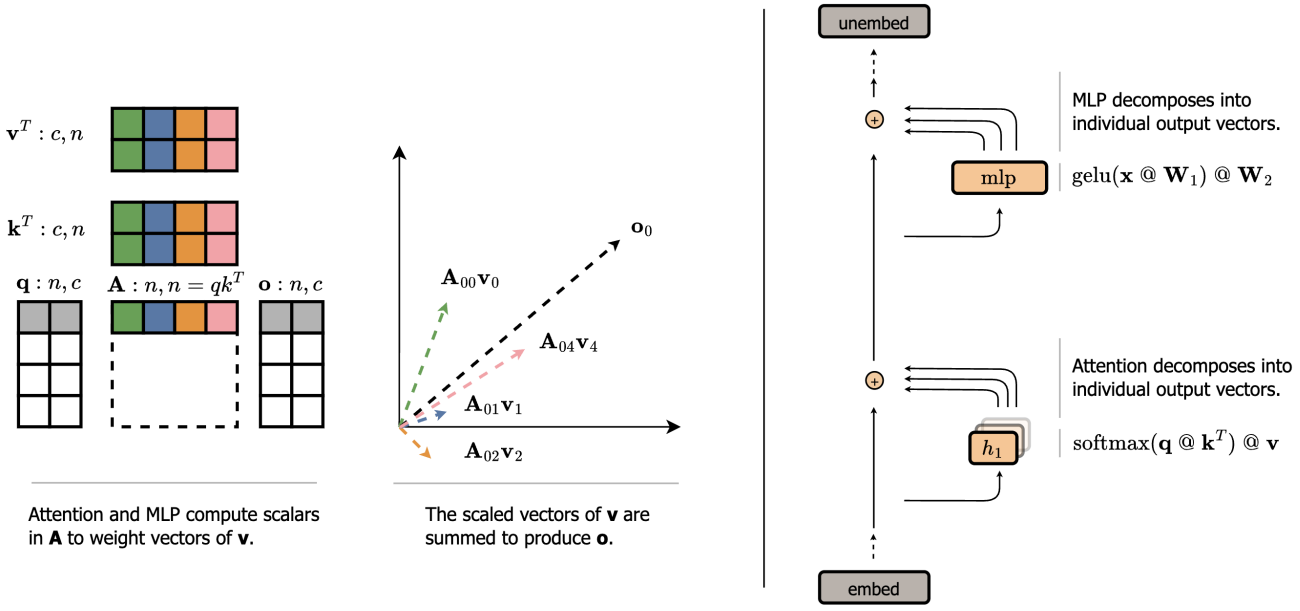

Figure 3. Left: Both MLP and attention operations use a non-linearity to compute the scalar coefficients for a linear combination of value vectors. In attention, the value vectors are dynamic (V is projected from the current token representation). In MLP, the value vectors are static (rows of the weights W2). Right: DiTs accumulate sums of these scaled and shifted value vectors (MLP and attention outputs).

The mean and variance computations do not distribute over vector addition – however, the resulting modification to the residual stream, as visualized in Fig. 2, only uses the mean and variance scalars as scale and shift factors, which do distribute over the sum of individual vectors. Thus, in the residual stream framework (Elhage et al., 2021), layernorm can be interpreted as performing a two phase computation: (1) a non-distributive "read" of the residual stream to compute scale and shift factors, (2) a distributive "write" over all vectors in the residual stream using the now fixed scale and shift factors computed in phase (1). This means that after layernorm, the residual stream is still a sum of individually identifiable vectors.

Linear Projections: Linear projections defined by matrices W distribute linearly:

$$W\left(\sum_i v_i\right) = \sum_i W v_i \tag{9}$$

Non-Distributive Operations. Operations such as softmax and GELU do not distribute over vector addition, but similar to layernorm, they do not directly modify the vectors in the residual stream in a non-distributive manner. Instead, they compute scalar coefficients that scale the vectors newly added to the residual stream (Fig. 2).

Final Latent Space Path Decomposition. Considering these properties collectively, we assert the following:

1. Attention and MLP layers output sums of scaled vectors, incrementally updating the residual stream.
2. Modulations, scale and shift factors computed by layer normalization, and linear projections distribute over these vector additions.
3. Non-distributive operations (softmax, GELU) are solely used to compute scalar coefficients of newly added vectors without directly modifying the residual stream.

Thus, the latent space path of a DiT output at any inference step t, denoted z_t, can be formally expressed as a sum of individually scaled vectors from attention and MLP layers across all previous steps $t' > t$ (where z_0 is the fully denoised

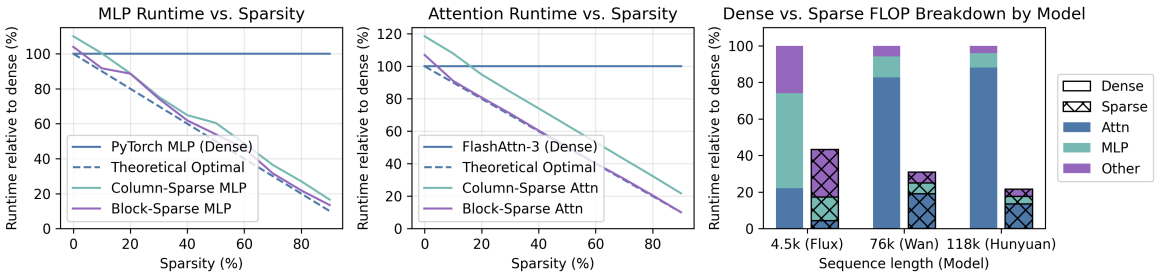

Figure 4. Left, Middle: Sparse kernel runtime scales linearly with sparsity. Column sparsity is competitive with block sparsity. *Right*: FLOP breakdown by model at Chipmunk hyperparameters.

output):

$$z_t = \text{LN}^t(\text{Mod}^{t,\text{final}} \odot z_{t+1}W_e)W_u * s_t \tag{10}$$

$$+ \sum_{l\in\text{layers}} \sum_h \sum_j \text{LN}^t(\text{Mod}^{l,t,\text{attn}} \odot \alpha_j^{h,l,t}(x_j^{l,t}W_v^{h,l}W_o^{h,l}))W_u * s_t \tag{11}$$

$$+ \sum_{l\in\text{layers}} \sum_m \text{LN}^t(\text{Mod}^{l,t,\text{mlp}} \odot \beta_m^{l,t}W_2^{m,:,l})W_u * s_t \tag{12}$$

Here, vectors $x_j^{l,t}W_v^{h,l}W_o^{h,l}$ and $W_2^{m,:,l}$ represent individual vectors produced by attention and MLP respectively, and scalars $\alpha_j^{h,l,t}$, $\beta_f^{l,t}$ represent their corresponding coefficients computed through non-linear activations. LN^t represents layernorm with scale and shift factors computed according to the state of the residual stream at timestep t. W_e, W_u represent the embed/unembed linear projections, respectively, and s_t represents the scale factor applied to the DiT output at timestep t according to the diffusion noise schedule.

B. Kernel Optimizations

Method	Speedup
Attention & unfused column-sum	1.0x
→ Fused Kernel	3.26x
MLP & unfused cache operations	1.0x
→ Fused Kernel	1.41x
Attention mask to indices conversion (unfused)	1.0x
→ Fused Kernel	1.39x
PyTorch top-k	1.0x
→ Approximate top-k	3.42x

Table 3. Kernel optimizations applied to Chipmunk's end-to-end algorithm measured on FLUX.1-dev shapes on H100-SXM5 GPUs with CUDA 12.8 and PyTorch 2.5.0.

In this appendix, we provide additional details on Chipmunk's kernel optimizations. We structure this discussion into three primary categories: (1) efficiently identifying dynamic sparsity patterns in attention kernels, (2) optimizing column-sparse computations in MLP layers, and (3) reducing GPU memory overhead through effective caching strategies.

B.1. Architecture Agnostic

Chipmunk works on all modern GPUs with tensor cores. Although our CUDA kernels are optimized specifically for NVIDIA H100 GPUs, Chipmunk can be efficiently implemented on earlier or later GPU architectures by using architecture-agnostic

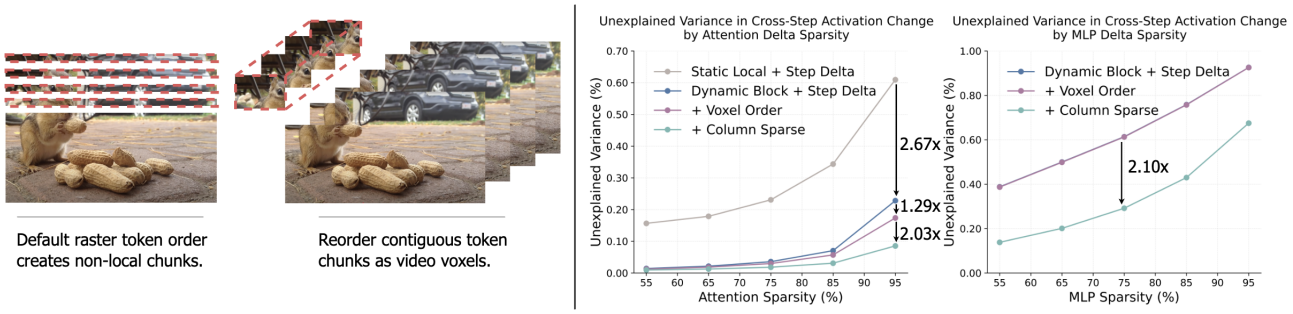

Figure 5. Left: Chipmunk reorders tokens such that column sparse patterns route voxels (3D cubes of pixels) to the same set of activated keys/values (attention) or model weights (columns/rows of W_1/W_2 in MLPs). *Right:* We plot the unexplained variance in cross-step activation changes (1 - R^2). Relative to dynamic block sparsity, dynamic column sparsity gives a 2x reduction in unexplained variance across both attention and MLPs. Voxel order reduces unexplained variance in attention but has nearly zero impact in MLP (overlapping with the blue line).

kernels written in Triton. These kernels can also be tuned for optimal performance on other hardware. We include several example Triton kernels in our codebase for reference.

B.2. Tile Packing: Mapping Sparse Computation to a Compacted Dense Computation

Chipmunk uses column sparse patterns on intermediate activations (Chen et al., 2021b; Li et al., 2022a; Ye et al., 2025), where a group of contiguous tokens only activates a certain set of individual keys/values (attention) or neurons (MLPs), to achieve low approximation error while remaining hardware-efficient.

Table 4. Comparing hardware efficiency and approximation error of block sparsity vs. column sparsity. Approximation error is measured as unexplained variance as described in 3.1. Runtime is measured as a % relative to dense computation at HunyuanVideo sequence length (118k).

	Attention			MLP		
	Sparsity	Error	Runtime	Sparsity	Error	Runtime
[192, 128] Block	95%	17.4%	5.1%	75%	61.3%	27.1%
[192, 1] Column	95%	8.5%	8.5%	75%	29.1%	27.7%

Block sparsity is efficient but has a higher approximation error than column sparsity. As described in Section 2, GPU kernels achieve peak efficiency by computing large block matrix multiplications. Block sparse kernels maintain hardware-efficiency because the outer loop simply skips certain tiles while leaving inner logic unchanged (Tillet et al., 2019). Block sparsity achieves near-optimal performance (Fig. 4, purple lines), but suffers from 2x higher approximation error than finer-grained column sparsity patterns (Table 4, Fig. 5).

Column sparsity can be made efficient with sparse gathers from HBM to SRAM. We implement column-sparse attention and MLP kernels to achieve 2x less approximation error than block sparsity (Fig. 5, purple & cyan lines) while maintaining competitive speedups (Table 4, Fig. 4). Traditional high-performance dense attention and MLP kernels, such as FlashAttention-3 and cuBLAS GEMMs, use the following structure: (1) load dense 2D tensors from HBM to SRAM, (2) compute the large matrix multiplications with tensor core instructions, and (3) store results back to HBM (Shah et al., 2024; Spector et al., 2024). To maintain tensor core utilization with column sparsity, we modify step (1) to pack sparse keys/values from non-contiguous rows in global memory into a dense tile in shared memory (Fig. 1) (Chen et al., 2021b; Li et al., 2022a; Ye et al., 2025). Tangentially, Chipmunk's sparse kernels can take on any static sparsity pattern (e.g. Sliding Tile Attention (Zhang et al., 2025b), DiTFastAttn (Yuan et al., 2024)) by simply passing in a particular set of indices.

Column sparsity routes a chunk of tokens to a set of individual key/values (attention) or neurons (MLP). Given an arbitrary attention or MLP computation, column sparse patterns mask intermediate activations with granularity $[C, 1]$ (Chen

et al., 2021b; Li et al., 2022a; Ye et al., 2025). This means that, given a chunk of contiguous tokens of size C, the chunk only activates a certain set of individual keys/values (attention) or neurons (MLPs). This is in contrast to MoE models, which route individual tokens to chunks of contiguous MLP neurons (experts) (Jiang et al., 2024). Column sparsity is parameterized by the chunk size C, and on H100 GPUs, we find $C = 192$ is efficient.

Reordering tokens into video voxel/image patch chunks improves attention quality. By default, contiguous tokens follow raster order (left-to-right, top-down, sequential frames). However, small video voxels (3D cubes of pixels) or image patches have similar color and brightness, and we expect them to exhibit similar interactions with other tokens. Thus, we reorder tokens at the beginning of the diffusion process such that each contiguous token chunk, which is routed to the same set of sparse indices, corresponds to a voxel/patch. We find this to improve attention approximation quality by 1.2x (Fig. 5).

B.3. Efficient Dynamic Sparsity Pattern Identification in Attention

We now address the challenge of efficiently identifying sparsity patterns in attention computations. As discussed previously (Section 3.2.2 of the main paper), Chipmunk employs fused kernels that simultaneously compute attention outputs and sparsity patterns via column sums (Alg. 4). A direct summation of the unnormalized logits (qk^T) across rows is infeasible because row magnitudes can vary significantly without softmax normalization. Naively computing column sums directly on the post-softmax matrix is also impractical due to FlashAttention's (Dao et al., 2022) incremental softmax, which never fully materializes intermediate softmax results within the kernel.

To overcome these limitations without resorting to slower, unfused kernels, we employ an approximation leveraging the slow-changing nature of activations across inference steps. Specifically, we reuse softmax normalization constants from the previous inference step to approximate column sums. Although slightly stale, these constants remain effective due to the incremental changes between steps. Thus, our fused kernel outputs both the correctly normalized attention result and an approximate column sum (normalized using previous constants) suitable for subsequent top-k sparsity selection.

B.4. Optimizing Column-Sparse Delta MLP Computations

Next, we detail kernel optimizations specific to column-sparse delta computations within the MLP layers. Here, the static nature of MLP value vectors (rows of weight matrix W_2) allows for additional optimization compared to attention, which has dynamic vectors projected from token representations.

B.4.1. Computing MLP Step-Deltas Efficiently in One Pass

Unlike attention, MLP step-deltas can be efficiently computed in a single step rather than requiring a subtraction and subsequent addition of vectors (Alg. 6). Given cached activations and MLP outputs, we directly compute sparse cross-step deltas as follows:

1. Compute sparse intermediate activations.
2. Compute the difference against the cached activation.
3. Multiply this sparse delta by the static value vectors (rows of W_2).
4. Directly accumulate this result into the cached output.

This reduces computational overhead compared to the two-step subtraction-addition method required for dynamic attention vectors, but introduces additional challenges in kernel optimization.

B.4.2. Persistent Grid and Warp-Specialization for Complex Epilogues

The first GEMM kernel in Chipmunk's MLP delta computation involves a complex epilogue due to the combined computational steps of delta computation and memory operations (Alg. 6). To optimize these, we find the combination of persistent grids and warp-specialization to be particularly effective:

1. **Persistent Grid Kernels:** One threadblock is launched per GPU Streaming Multiprocessor (SM), allowing each threadblock to iterate over multiple work tiles.

2. **Warp-Specialization:** Within each threadblock, separate warp groups are assigned to compute/data loading operations, allowing better overlap between computation and memory operations.

This combination allows the overlap of the producer warpgroup's memory loading prologue with the consumer warpgroups' high latency epilogue operations.

B.4.3. Custom Kernel for Efficient Cache Writeback

We also found the most time-consuming step in the first MLP GEMM epilogue to be the scattering of activation cache updates into global memory. To address this, we fuse this memory-bound cache writeback operation into the second GEMM operation, which is compute-bound. Specifically, we utilize the CUDA driver API to allocate streaming multiprocessors (SMs) to a custom kernel implementing the cache writeback operation, while using the rest of the SMs for the GEMM. We compute the number of SMs to allocate based on the degree of wave quantization so that this does not impact the runtime of the GEMM—it just repurposes any leftover SMs. Our custom cache writeback kernel uses the TMA-based reduction PTX instructions (`cp.reduce.async.bulk`) to perform large atomic updates into global tensors.

B.5. Minimizing GPU Memory Overhead from Activation Caches

Finally, Chipmunk stores activation caches MLP and attention layers, making memory efficiency critical—particularly in single-GPU workloads with large sequence lengths (e.g., HunyuanVideo with 118k tokens per video). Each attention layer requires caching (1) boolean masks indicating active [192, 1] columns, and (2) activation outputs from the previous inference step.

We implement two optimizations to reduce GPU memory footprint:

1. **Bitpacked Sparsity Masks:** Standard boolean masks (torch.bool) consume one byte per entry. With a torch-compiled bitpacking function, we reduce memory usage by 8x, while incurring negligible computational overhead.

2. **CPU Offloading with Double-Buffered Communication:** We preallocate pinned (page-locked) CPU tensors and implement double-buffering on the GPU. This approach reduces GPU memory and communication overhead by overlapping GPU computations of the current layer with simultaneous transfers of the next layer's sparsity masks and activations from CPU to GPU memory.

Table 5. Memory usage comparison between naive and optimized implementations.

	Naive	Optimized	Memory Reduction
Sparsity Mask Cache	104 GB	13 GB	8x
Activation Cache	43 GB	2.8 GB	15x
Column-Sum Intermediate State	668 GB	3.5 GB	192x

C. Chipmunk Algorithm

In this appendix, we describe the Chipmunk algorithm in detail. Chipmunk accelerates Diffusion Transformer (DiT) inference by interleaving dense and sparse-delta steps to exploit the slow changing activations across steps. Dense steps refresh cached activations and identify dynamic sparsity patterns, while sparse steps efficiently compute sparse activation deltas against cached activations. Structured column-chunk sparsity patterns are applied to the intermediate activations of both attention and MLP to enable hardware-efficient sparse algorithms. To improve this column-sparse approximation quality, Chipmunk applies a reordering to tokens at the beginning of the diffusion process such that each contiguous chunk of c tokens corresponds to a 3D video voxel (Alg. 1). All tokens in one voxel will therefore share the same sparsity pattern in later steps. The inverse permutation is applied before the model's final projection. We will now discuss the details of Chipmunk's dense and sparse computations for attention and MLP layers[1].

[1]We note that the algorithms described here are for reference correct implementations, rather than optimized for speed. In practice, operations in these algorithms implemented in optimized CUDA kernels as described in Section 3 of the main paper and Appendix B.

Algorithm 1. **Voxel Reordering.** Reorders tokens so that contiguous chunks correspond to coherent 3D voxels, improving the quality of subsequent column-sparse approximations.

```
# x            : (b, t, h, w, d)      3D tokens
# vt, vh, vw   : int, int, int        Voxel shape

def voxel_order(x, vt, vh, vw):
    return rearrange(
        x,
       'b (tc vt) (hc vh) (wc vw) d -> b tc hc wc vt vh vw d',
        vt=vt, vh=vh, vw=vw
    )

def reverse_voxel_order(x, vt, vh, vw):
    return rearrange(
        x,
       'b tc hc wc vt vh vw d -> b (tc vt) (hc vh) (wc vw) d',
        vt=vt, vh=vh, vw=vw
    )
```

C.1. Chipmunk Attention

Dense Steps. We run full scaled-dot-product attention on the reordered sequence

$$\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{B\times H\times N\times E}, \qquad \mathbf{P} = \mathrm{softmax}\big(\tfrac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{E}}\big),\ \mathbf{O}_{\text{dense}} = \mathbf{P}\mathbf{V}.$$

We then partition the query axis of attention matrix $\mathbf{P}$ into chunks of c contiguous tokens (the queries of one voxel),

$$\mathbf{P} \to \mathrm{reshape}\big(B, H, \tfrac{N}{c}, c, N\big)$$

and sum over the c dimension. The result is a "column-sum" tensor

$$\mathbf{D} \in \mathbb{R}^{B\times H\times \frac{N}{c}\times N}, \qquad \mathbf{D}_{b,h,i,j} = \sum_{q=ic}^{(i+1)c-1} P_{b,h,q,j}$$

which tells us, for each voxel i, how much total attention probability it assigns to key j (Alg. 4). A top-k over the last dimension then selects the most-attended keys/values for each query voxel:

$$\mathrm{idx}_{b,h,i,:k} = \mathrm{TopK}_k\big(\mathbf{D}_{b,h,i,:}\big)$$

These indices are cached for the upcoming sparse steps. Finally, Chipmunk defines the attention activation cache as

$$\mathbf{O}_{\text{cache}} = \mathbf{O}_{\text{dense}} - \mathrm{softmax}\big(\mathbf{Q}\,\mathbf{K}_{\text{idx}}^\top\big)\,\mathbf{V}_{\text{idx}}$$

such that subsequent steps can directly add to this cache to compute a sparse replacement of the top attention interactions identified by the cached indices (Alg. 2).

Sparse Delta Steps. In subsequent sparse steps Chipmunk only recomputes the top attention interactions defined by the indices cache (Alg. 3):

$$\Delta\mathbf{O} = \mathrm{softmax}\big(\mathbf{Q}\,\mathbf{K}_{\text{idx}}^\top\big)\,\mathbf{V}_{\text{idx}}, \qquad \mathbf{O} = \mathbf{O}_{\text{cache}} + \Delta\mathbf{O}.$$

The computational complexity of attention thus drops from $\mathcal{O}(N^2)$ to $\mathcal{O}(Nk)$, reproducing the dense result when $k = N$. Conceptually, this computation corresponds exactly to Fig. 1 in the main paper, where we visualize the sparse recomputation

of the fastest changing vectors in attention's output linear combination (scaled rows of $\mathbf{V}$), while reusing the remaining vectors from the cache.

C.2. Chipmunk MLP

For MLP layers, Chipmunk similarly alternates dense and sparse-delta computations. Dense steps fully compute activations and initialize caches, while sparse steps efficiently identify and recompute only activations that significantly change across steps (Alg. 5).

Dense steps. A standard two-layer MLP layer produces

$$\mathbf{A} = \sigma\big(\mathbf{X}\mathbf{W}_1^\top + \mathbf{b}_1\big),\ \mathbf{O}_{\text{dense}} = \mathbf{A}\mathbf{W}_2,$$

with shapes $\mathbf{X} \in \mathbb{R}^{B\times N\times D},\ \mathbf{W}_{1,2} \in \mathbb{R}^{F\times D}$. Chipmunk caches three tensors on dense steps:

1. Tile-mean pre-activation $\mathbf{T}_m = \text{mean}_c(\text{reshape}_c(\mathbf{X})\mathbf{W}_1^\top) \in \mathbb{R}^{B\times N/c\times F}$.
2. Full activation tensor $\mathbf{A}_{\text{cache}}$.
3. MLP output $\mathbf{M}_{\text{cache}} = \mathbf{O}_{\text{dense}}$.

Sparse Delta Steps. Chipmunk uses a token-merged approximation to identify the k neurons that have changed the most since the last step (for each size-c chunk of contiguous tokens) (Alg. 7):

$$\Delta_{\text{tm}} = |\mathbf{T} - \mathbf{T}_m|, \quad \text{idx} = \text{TopK}(\Delta_{\text{tm}}) \in \mathbb{R}^{B\times N/c\times k},$$

We then recompute only those token-neuron interactions and reuse the rest from the cache (Alg. 6). As an optimization, we compute MLP in a single step—without the separate subtraction and addition of Eq. 1 in the main paper—because of the static nature of the second operand (W_2).

$$\Delta\mathbf{M} = \sigma\big(\mathbf{X}\,\mathbf{W}_{1,\text{idx}}^\top - \mathbf{A}_{\text{cache, idx}}\big)\mathbf{W}_{2,\text{idx}}, \qquad \mathbf{O} = \mathbf{M}_{\text{cache}} + \Delta\mathbf{M}.$$

Analogous to attention, this computation corresponds exactly to Fig. 1 in the main paper. We compute a sparse replacement of the fastest changing vectors in MLP's output linear combination (scaled rows of $\mathbf{W}_2$), while reusing the remaining vectors from the cache.

Algorithm 2. **Chipmunk Attention.** Computes attention outputs by interleaving dense and sparse-delta steps. Dense steps initialize sparsity patterns and caches, while sparse steps selectively recompute attention interactions based on cached indices and activations.

```
def chipmunk_attn(q, k, v, is_dense_step):
    # q, k, v       : (b, h, n, e)
    # is_dense_step : bool
    # returns o     : (b, h, n, e)

    if is_dense_step:
      o, cs, m, l  = colsum_attn(
                       q, k, v,
                       prev_m, prev_l
                     )
      inds, counts = topk(cs, dim=-1)
      prev_m       = m
      prev_l       = l
      o_cache      = colsparse_delta_attn(
                         q, k, v,
                         inds, counts,
                         o, o_scale = -1
                       )

    else:
      o            = colsparse_delta_attn(
                         q, k, v,
                         inds, counts,
                         o_cache, o_scale = 1
                       )

    return o
```

Algorithm 3. **Column Sparse Delta Attention.** Efficiently recomputes sparse attention interactions defined by previously cached indices.

```
def colsparse_delta_attn(q, k, v, inds, counts, o_cache, o_scale):
    # q, k, v       : (b, h, n, e)
    # inds          : (b, h, n/c, topk)   sparse indices per Q-tile
    # counts        : (b, h, n/c)         number active per Q-tile
    # o_cache       : (b, h, n, e)

    # c             : q  tile size
    # ck            : kv tile size

    o = o_cache.clone()
    for i in range(0, n, c):
        q_tile        = q[:, :, i : i+c]           # (b, h, c, e)
        o_tile        = o[:, :, i : i+c]           # (b, h, c, e)
        idx_q         = inds[:, :, i//c, :]        # (b, h, topk)
        counts_amt    = counts[:, :, i//c]         # (b, h, 1)

        for j in range(0, counts_amt, ck):
            idx_tile  = idx_q[..., j : j+ck]       # (b, h, ck)
            k_tile    = k.gather(idx_tile, dim=-2) # (b, h, ck, e)
            v_tile    = v.gather(idx_tile, dim=-2) # (b, h, ck, e)
            qk        = q_tile @ k_tile.T(-2, -1)  # (b, h, c, ck)
            p         = online_softmax(qk)         # (b, h, c, qk)
            o_tile   += o_scale * (p @ v_tile)     # (b, h, c, e)

    return o
```

Algorithm 4. **Column Sum Attention.** Computes standard attention along with column-wise sums of attention probabilities, which are used for dynamically identifying sparsity patterns.

```
def colsum_attn(q, k, v, prev_m, prev_l):
    # q, k, v   : (b, h, n, e)
    # prev_m    : (b, h, n, 1)
    # prev_l    : (b, h, n, 1)
    # returns o : (b, h, n, e)
    #        cs : (b, h, n/c, n)  column-chunk sums

    qk       = q @ k.T(-2, -1) / sqrt(e)          # (b, h, n, n)
    p        = softmax(qk, dim=-1)                # (b, h, n, n)
    o        = p @ v                              # (b, h, n, e)

    p_approx = exp(qk - prev_m) / prev_l          # (b, h, n, n)
    cs       = rearrange(                         # (b, h, n/c, n)
                 p_approx,
               'b h (nc c) m -> b h nc c m',
                 c=c
               ).sum(dim=-2)

    return o, cs
```

Algorithm 5. **Chipmunk MLP.** Performs MLP layer computations by alternating dense and sparse-delta steps. Dense steps compute full activations and initialize caches, and sparse steps efficiently update only the activations that exhibit significant changes across inference steps.

```
def chipmunk_mlp(x, w1, w2, is_dense_step):
    # x              : (b, n, d)
    # w1, w2         : (f, d)
    # is_dense_step : bool
    # returns o      : (b, n, d)

    if is_dense_step:
      o, preact, act = mlp(x, w1, w2)
      tm_cache       = rearrange(
                         preact,
                        'b (nc c) d -> b nc c d',
                         c=192
                       ).mean(dim=-2)
      act_cache      = act.clone()
      mlp_cache      = o.clone()

    else:
      inds, counts   = compute_mlp_indices(
                         x,
                         w1,
                         tm_cache
                       )
      o              = colsparse_delta_mlp(
                         x,
                         w1, w2,
                         self.inds, self.counts,
                         act_cache, o_cache
                       )

    return o
```

Algorithm 6. **Column Sparse Delta MLP.** Computes sparse deltas for MLP outputs by selectively recomputing only significantly changed neuron activations.

```
def colsparse_mlp_delta(x, w1, w2, inds, counts, a_cache, o_cache):
    # x, w1, w2    : (b, n, d), (f, d), (f, d)
    # inds         : (b, n/c, topk)    active neurons per token-tile
    # counts       : (b, n/c)          number active per token-tile
    # a_cache      : (b, f, n)         cached activations
    # o_cache      : (b, n, d)         cached MLP output
    #
    # c            : M tile size
    # cn           : N tile size
    # ck           : K tile size

    # ---- GEMM1 : delta = x @ W1[idx, :].T - a_cache  ----
    tasks = [(i, j) for i in range(0,         n, c)
                    for j in range(0, counts[i], cn)]
    for (i, j) tasks:
        idx_tile     = inds[:, i//c, j : j+cn]         # (b, cn)
        a_tile       = a_cache[idx_tile].T             # (b, c, cn)
        pre          = zeros_like(a_tile)              # (b, c, cn)

        for k in range(0, d, ck):
            x_tile   = x[:, i: i + c, k: k + ck]       # (b, c, ck)
            w1_tile  = w1[idx_tile, k : k + ck]        # (b, cn, ck)
            pre     += x_tile @ w1_tile.T              # (b, c, cn)

        a            = act_fn(pre + w1.bias[idx_tile]) # (b, c, cn)
        delta        = a - a_cache                     # (b, c, cn)
        a_cache.scatter_(-2, idx, a.T)                 # (b, cn, c)

    # ---- GEMM2 : o_cache += delta @ W2[idx, :]  ----
    tasks = [(i, j) for i in range(0, n, c)
                    for j in range(0, d, cn)]
    for (i, j) tasks:
        o_tile       = o_cache[:, i: i + c]            # (b, c, cn)

        for k in range(0, counts[i], ck):
            idx_tile = inds[:, i//c, k: k + ck]        # (b, ck)
            d_tile   = delta[:, i: i + c, k: k + ck]   # (b, c, ck)
            w2_tile  = w2[idx_tile, j : j + cn]        # (b, ck, cn)
            o_tile  += delta @ w2_tile                 # (b, c, cn)

    return o_cache                                     # (b, n, d)
```

Algorithm 7. **Computing MLP Sparse Indices.** Identifies neurons in MLP layers with the largest cross-step activation changes.

```
def compute_mlp_indices(x, w1, tm_cache):
    # x                : (b, n, d)
    # w1               : (f, d)
    # tm_cache         : (b, n/c, f)
    #
    # returns inds   : (b, n/c, topk)  top neurons per token-group
    #         counts : (b, n/c, 1)     active neurons per token-group

    tm           = rearrange(                          # (b, n/c, d)
                     x,
                    'b (nc c) d -> b nc c d',
                     c=c
                   ).mean(dim=-2)
    tm           = tm @ w1.T                           # (b, n/c, f)
    delta        = (tm - tm_cache).abs()               # (b, n/c, f)
    inds, counts = delta.topk(dim=-1).indices          # (b, n/c, topk)

    return inds, counts
```

D. Experiments

We evaluate Chipmunk against state-of-the-art DiT acceleration methods on text-to-video and text-to-image tasks. First, we outline our experimental setup (D.1). Quantitative comparisons in Tables 2 and 6 demonstrate Chipmunk's efficiency and quality improvements (D.2). Qualitative examples (Fig. 9) illustrate Chipmunk's preservation of visual quality under significant acceleration (D.3).

Method	Efficiency			Visual Quality	
	FLOPs ↓	Speedup ↑	Latency (s) ↓	ImRe ↑	CLIP ↑
FLUX.1-dev, T=50 (768 × 1280)					
Flux	100%	1x	6.60s	0.76	31.07
STA	84%	1.15x	5.73s	0.75	31.13
DiTFastAttn	83%	1.09x	6.05s	**0.80**	31.29
Chipmunk	**58%**	**1.41x**	**4.90s**	**0.80**	**31.31**
Step+Token Caching (ToCa)	66%	1.51x	4.37s	0.76	31.21
Step Caching (TeaCache)	39%	2.51x	2.64s	0.68	31.37
Chipmunk+Step Cache	**31%**	**2.56x**	**2.57s**	**0.77**	**31.44**

Table 6. Performance comparison of various methods across different datasets for image generation.

D.1. Setup

Models. We evaluate Chipmunk across three state-of-art DiT models: HunyuanVideo and WAN2.1 for text-to-video generation and FLUX.1-dev for text-to-image generation, all using their default number of generation steps (50). As shown in Fig. 4 (right), these models vary in sequence and FLOP breakdown, creating a strong evaluation of acceleration ability in different inference regimes. Of the three models, FLUX.1-dev has the smallest sequence length at 4.5k and allocates a majority of FLOPs to MLP layers. WAN2.1 and HunyuanVideo are extremely bound by attention at sequence lengths of 76k and 118k, respectively. All benchmarks are performed on H100-SXM5 GPUs with CUDA 12.8 and PyTorch 2.5.0, comparing against a 650 TFLOP FlashAttention-3 baseline (Shah et al., 2024).

Baselines. We compare Chipmunk against a number of recent DiT acceleration techniques including TeaCache (Liu et al., 2024a), ToCa (Zou et al., 2025), Sliding Tile Attention (STA) (Zhang et al., 2025b), and DiTFastAttn (Yuan et al., 2024). TeaCache is a method for dynamically reusing DiT step outputs during slow-changing portions of the generation process,

and ToCa dynamically computes token importance scores to recompute activations for only the most important tokens while reusing the rest. STA extends sliding window attention to 3D video to exploit the natural locality in attention computations, and DiTFastAttn reuses non-local attention interactions in a subset of steps to avoid a full recomputation. Here, we also stack Chipmunk with a simple static step cache schedule, untuned for any particular model/prompt. Full baseline descriptions are available in Appendix D.

Evaluations and Metrics. We evaluate text-to-video generation with the standard VBench evaluation, comprised of 16 dimensions that are weighted to produce a final score shown to align with human judgement (Huang et al., 2023). For text-to-image generation, we compute two standard metrics: ImageReward (ImRe) (Xu et al., 2023), a common human-preference trained reward model, and CLIP, a widely used metric assessing semantic alignment between text prompts and images (evaluated on the ImRe dataset) (Radford et al., 2021).

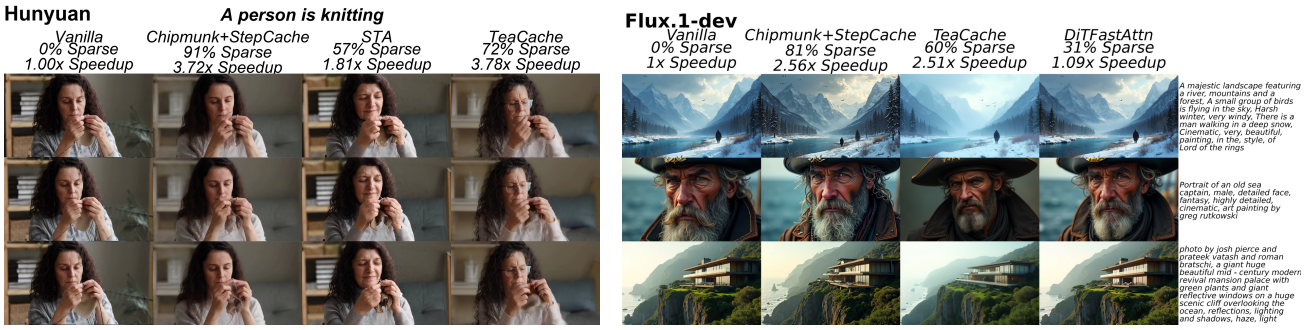

Figure 6. Qualitative comparisons across videos (left) and images (right). For videos, frames are stacked vertically (down is later). See Appendix D for more.

Hyperparameters. In a warm-up phase for each model, we test 100 E2E generations in order to select values of *MLP sparsity level* and *attention sparsity level* that will achieve 95% of the change in activations across steps. For the *step schedule* hyperparameter, we choose a simple schedule of interleaving 1 dense step every 10 sparse steps; further optimization could yield additional efficiency improvements.

D.2. Quantitative Results

Across image and video models, Chipmunk achieves the largest acceleration levels while maintaining near-lossless quality across ImageReward, CLIP, and VBench (Tables 2 and 6). We evaluate results in the context of three regimes: **(1) Acceleration Held Constant.** Holding acceleration level constant, Chipmunk's fine-grained caching of individual attention and MLP vectors enable it to maintain higher quality than TeaCache, which caches full step outputs (HunyuanVideo, rows 4-5; FLUX.1-dev, rows 6-7). **(2) Quality Held Constant.** When quality is held nearly constant, Chipmunk achieves higher acceleration than STA, ToCa, and DiTFastAttn, due to its dynamic, granular identification of sparsity patterns (HunyuanVideo, rows 2-3; FLUX.1-dev, rows 2-5). **(3) Chipmunk Speed-Quality Tradeoff.** Chipmunk's sparsity hyperparameter can modulate the speed-quality tradeoff (WAN2.1, rows 3-4).

D.3. Qualitative Results

In videos, we observe that Chipmunk preserves small, high-moving parts such as hands knitting (Fig. 9). In images, Chipmunk maintains strong visual quality, including details from long prompts. In both text-to-image and text-to-video tasks, Chipmunk enhances details on the subject of the generation, such as the buttons on the captain's jacket. We hypothesize this is due to the $\sim$90% sparse attention matrix focusing on the most relevant parts of the prompt (e.g., the subject). We speculate Chipmunk's improved ImageReward, VBench, and CLIP scores may stem from this focus on subject prominence, as these metrics evaluate how closely outputs align with prompts that largely describe a subject (Radford et al., 2021; Xu et al., 2023).

In Fig. 10, we test the effect of reusing non-recomputed attention interactions from a previous step and using dynamically identified sparsity patterns. On the left, we observe that only using a static local mask at high sparsity levels introduces quality degradation and object warping. On the right, we find that reusing attention interactions and adding only 1% of

dynamically selected top attention interactions restores significant quality. We also study the speed-quality tradeoff at high levels of sparsity in Fig. 11. At high levels of sparsity, such as 92% on the left, we observe warping on detailed objects such as a hand making fine motor movements to draw on paper. We find that slightly increasing the number of dynamically selected attention interactions to be recomputed, from 1% to 5%, significantly improves quality. For a sample of playable VBench video generations from HunyuanVideo VBench, please see this anonymized YouTube link: `https://www.youtube.com/watch?v=rr0Pg4LHqVI`. For a sample of playable VBench video generations from WAN2.1 VBench, please see this anonymized YouTube link: `https://www.youtube.com/watch?v=etquKck_wtc`

Failure Modes. At times, the background of videos appears slightly out of focus (bookshelf behind the woman knitting), which we similarly speculate can be attributed to the sparsity in the attention matrix concentrating on subject-based parts of the prompt. In text-to-image tasks, even though Chipmunk maintains prompt adherence and high visual quality, we find minor differences in details of the generations when compared to the reference images (e.g., the number of background birds in row 1), likely due to the number of FLOPs removed.

D.4. Hyperparameters

In this section, we expand upon the hyperparameters of all methods used to create the tables.

- **TeaCache:** The threshold parameter is set to 0.78 for FLUX.1-dev, 0.2 for WAN2.1, and 0.65 for HunyuanVideo.
- **Sliding Tile Attention:** We approximated the tile size to approximately cover 58% sparsity, as described in their paper (Zhang et al., 2025b).
- **DiTFastAttn:** The default hyperparameters available at the GitHub implementation repository `xdit-project/xDiT` were used: window size=512, number of calibrations=4, and with caching enabled.
- **ToCa:** The default hyperparameters suggested in their paper were used (N=2, R=90%) (Zou et al., 2025).
- **Chipmunk:** We apply 84% attention sparsity and 70% MLP sparsity for FLUX.1-dev. For text-to-video generation we use 95% attention sparsity on HunyuanVideo and 82% attention sparsity on WAN2.1. Since MLP runtime accounts for a very small percentage of wall clock time in both HunyuanVideo and WAN2.1, we only apply sparse attention deltas to achieve the best speed-quality tradeoff.
- **Step Caching:** For entries marked Chipmunk + Step Caching, we use a simple uniform schedule we found to approximate the behavior of a number of adaptive scheduling methods. In the middle W steps of the diffusion process, we only compute every nth step, skipping all others by reusing the last computed model output. We use $W = 30$ and $n = 4$, which corresponds to a roughly 1.8x speedup with 50 total steps. Thus, each step is either (i) fully skipped, (ii) partially sparse (Chipmunk), or (iii) fully dense.

D.4.1. Qualitative Evaluations

Evaluation across prompts. Using the methods of the main results table in the paper body, we generate images across a variety of prompts and methods (Fig. 7). We observe that naively skipping steps may impair visual quality due to blur and loss of detail. We find Chipmunk preserves aesthetic quality but may change some details of the image while maintaining strong adherence to prompts. Although DiTFastAttn has strong quality, it only achieves a minor speedup.

Evaluation across speed-quality tradeoff. We evaluate multiple values of MLP and attention sparsity, ranging from 0% to 90%, to better understand how the MLP & attention sparsity parameters modulate the speed-quality tradeoff (Fig. 8). The images maintain strong quality levels of sparsity reaching 70-80%. Beyond 80%, significant artifacts are introduced.

E. Extended Related Work

Towards Few-Step Diffusion Models. Early diffusion models required hundreds to thousands of denoising steps per generated sample, originating from the foundational work of Sohl-Dickstein et al. (Sohl-Dickstein et al., 2015) and Ho et al. (Ho et al., 2020). Subsequent methods significantly reduced inference steps by enhancing sampling efficiency: DDIM introduced a non-Markovian forward diffusion process that decouples training and sampling steps, while the DPM-Solver family achieved substantial speedups by utilizing dedicated diffusion ODE solvers without retraining (Song et al., 2022;

Figure 7. Images on 1280x768 FLUX.1-dev evaluated on different captions and prompts randomly sampled from the ImageReward dataset. On the left, we have vanilla reference images. Naively skipping steps (third column) introduces significant artifacts, such as a blurry images and a loss of detail. Chipmunk preserves aesthetic quality but may change some details of the image (despite maintaining prompt adherence). DiTFastAttn also achieves high quality but only a minor speedup.

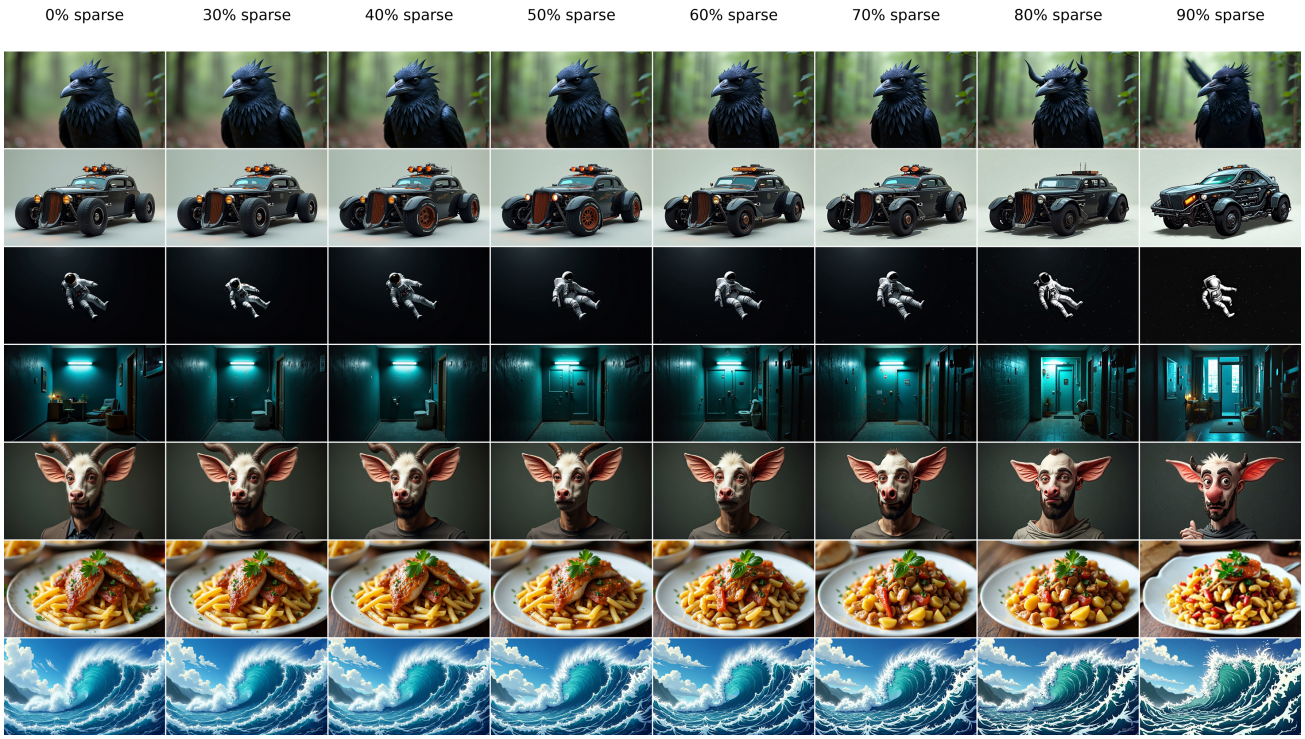

Figure 8. Speed-quality tradeoff on 1280x768 images generated on FLUX.1-dev with 50 steps at varying levels of Chipmunk sparsity. The images maintain strong quality at high levels of sparsity reaching 70-80%. Beyond 80%, significant artifacts are introduced and visual quality noticeably degrades resulting in blurry images or loss of detail. A single value was used for both attention and MLP sparsity. Prompts are listed below:

- ”anthropomorphic crow werecreature, photograph captured in a forest”
- ”a concept art of a vehicle, cyberpunk”
- ”astronaut drifting afloat in space, in the darkness away from anyone else, alone, black background dotted with stars, realistic”
- ”photo of a interior taken with a cheap digital camera at night flash lighting”
- ”A realistic photo of a man with big ears”
- ”delicious plate of food”
- ”tumultuous plunging waves, anime, artwork, studio ghibli, stylized, in an anime format”
- ”an alien planet viewed from space, extremely, beautiful, dynamic, creative, cinematic”

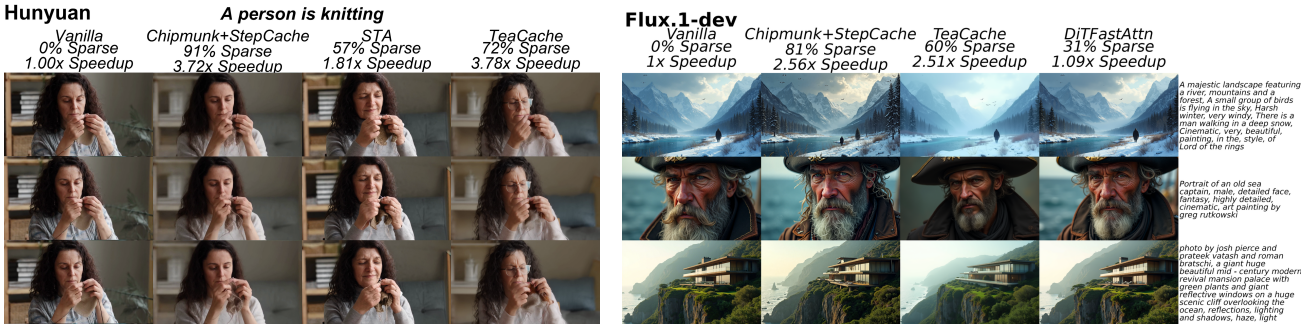

Figure 9. Qualitative comparisons across videos (left) and images (right). For videos, frames are stacked vertically (down is later). See Appendix D for more.

Lu et al., 2022). Another line of research has specifically targeted training processes optimized for single-step inference. For instance, Rectified Flow (Liu et al., 2022) learns straight-line mappings from noise to data, and Consistency Models (Song et al., 2023) directly model single-step noise-to-data transformations from arbitrary points along the noising trajectory. Additionally, step-distillation techniques, such as Progressive Distillation (Salimans & Ho, 2022), efficiently train few-step student models to approximate the longer sampling trajectories of multi-step teacher models. Complementing these approaches, which allocate computational resources uniformly once the number of steps is determined, Chipmunk studies an orthogonal dimension of efficiency: dynamically allocating computation within individual inference steps by selectively recomputing only the most important activations.

Efficient Attention Approximations. The quadratic complexity of self-attention mechanisms has driven extensive research into efficient approximations. Prior works have explored many strategies, including low-rank approximations (Wang et al., 2020), random-feature projections (Choromanski et al., 2022), locality-sensitive hashing or local attention masks (Kitaev et al., 2020; Zaheer et al., 2021), and combined sparse-plus-low-rank decompositions (Chen et al., 2021a; Arora et al., 2025). Recent studies on video and diffusion models have adopted static sliding-window attention masks (Zhang et al., 2025b; Yuan et al., 2024) and quantized attention computations (Zhang et al., 2025a). Chipmunk focuses on a dynamic sparse attention approximation to speedup DiTs and can be complemented with other techniques such as low-rank approximations or static mask patterns.

Efficient MLP Approximations. Conditional computation strategies such as Mixture-of-Experts (Shazeer et al., 2017; Fedus et al., 2022) selectively activate a subset of expert MLP modules per token, reducing FLOPs proportionally to top-k gating decisions. More fine-grained neuron-level sparsity methods, such as contextual sparsity (Liu et al., 2023), dynamically select only the most relevant neurons during autoregressive decoding, guided by a lightweight prediction model. Chipmunk complements these existing conditional computation techniques with the addition of activation reuse across inference steps. Instead of purely gating experts or neurons, Chipmunk leverages multi-step inference to selectively recompute only the neurons exhibiting significant changes, while reusing the activations of non-activated neurons from a cache. While initially demonstrated for diffusion models, the general concept of caching and sparse recomputation has potential applicability in other multi-step inference contexts, such as autoregressive decoding.

Figure 10. Ablation of adding deltas (to reuse attention interactions that are not recomputed) and 1% dynamic TopK attention mask on top of static local voxel attention. We find that at high sparsity levels, using only a static local mask results in artifacts and object warping. Adding deltas and just 1% of the dynamically selected top attention interactions significantly improves quality. *Note: Because the 3D dimensions do not divide evenly into 3D voxels, both configurations shown above also use full attention to and from the remainder slice in each dimension.*

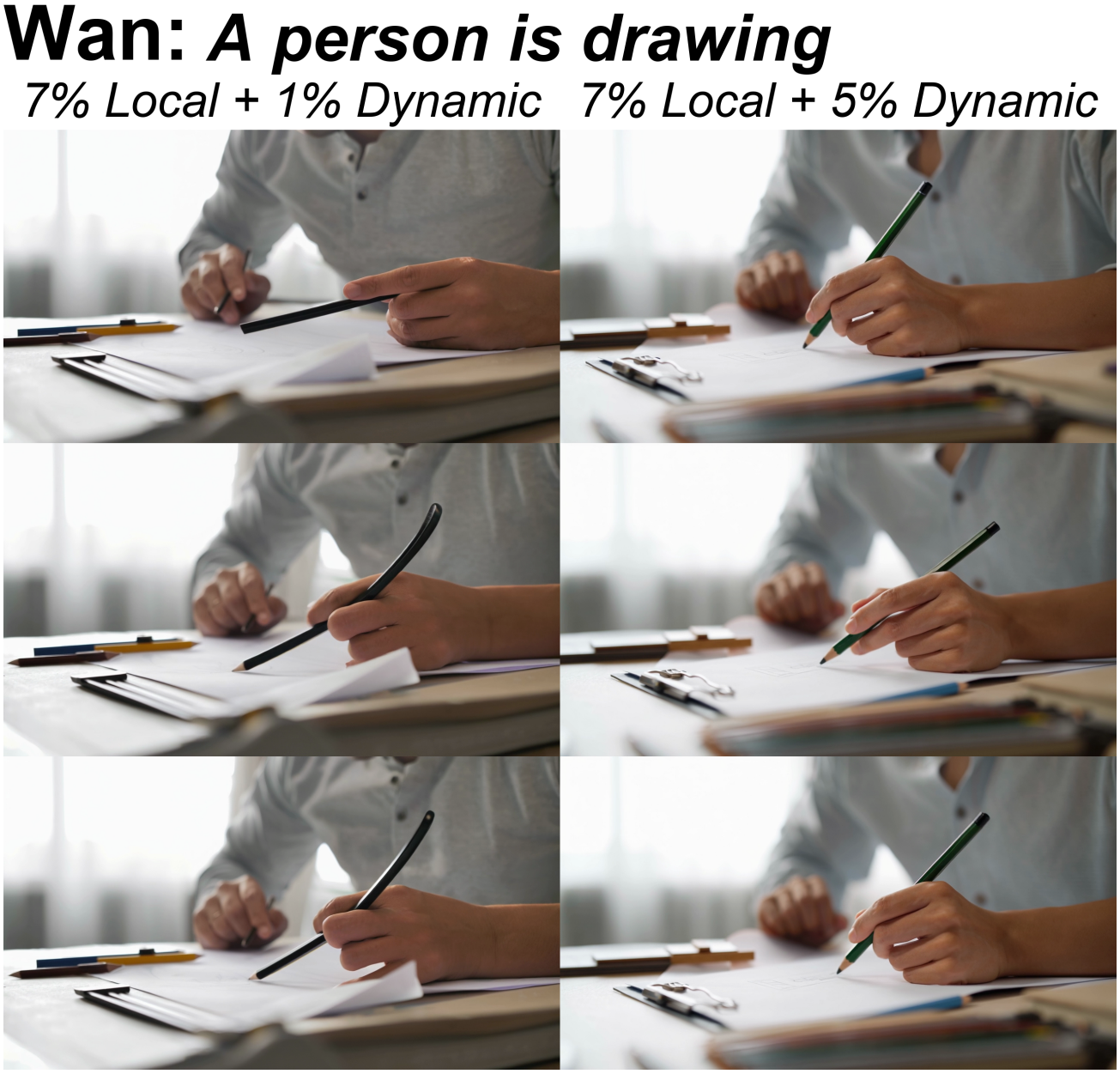

Figure 11. Comparing the quality of WAN2.1 generation at 1% dynamic top-k attention interactions and 5% dynamic top-k attention interactions. On the left, we see that at high sparsity levels, detailed objects such as the pencil and hands begin to experience warping. We then see on the right that increasing the number of dynamically selected top-k attention interactions from 1% to 5% restores high quality generation of the pencil and hands.