Memory-Efficient Fine-Tuning Diffusion Transformers via Dynamic Patch Sampling and Block Skipping

Supplementary Material

6. Implementation Details

6.1. Details of DiT-BlockSkip

Block Skipping with Residual Precomputing. The block skipping with residual precomputing is trained in two stages: precomputing and fine-tuning. As it operates orthogonally to dynamic patch sampling, we apply dynamic patch sampling during the precomputing stage to extract and store residual features from images resized to a fixed lower resolution ($s_{\text{min}} \times s_{\text{min}}$). During fine-tuning, we apply the same patch sampling (*e.g.*, location and timesteps) and diffusion timesteps during precomputing, to ensure correct alignment with the precomputed residual features.

To further validate the generalization capability of our skip indices selection algorithm, we selected skip indices for the DreamBooth experiments (Section 4) using the CustomConcept101 dataset. Specifically, we computed the indices of the skipped block for the randomly sampled 30 classes from CustomConcept101. To compute semantic distances between samples, we employed DINO [2]. The values of n and m corresponding to the dataset and model configurations are summarized in Table 4. In our experiments, the FLUX and SANA models consist of 57 and 20 blocks, respectively. For FLUX, the number of skipped blocks k was set to 17, 23, and 29 for skip ratios of 30%, 40%, and 50%, respectively. For SANA, the corresponding values of k were 6, 8, and 10.

Hyperparameters. To ensure fair comparison with all baselines, our method was trained with a batch size of 1, a gradient accumulation step of 4, and 500 training iterations. The learning rate was set to 1e-4 with AdamW [26] optimizer, and the LoRA rank was set to 128 for FLUX and 512 for SANA, consistent with baseline settings. For inference, FLUX-based models used the Flow Match Euler Discrete scheduler [14] with a guidance scale of 3.5. SANA-based models followed the original implementation, employing Flow-DPM-Solver [42] with 20 inference steps and a guidance scale of 4.5. Moreover, the models generated 1024×1024 resolution images for evaluation. All experiments were conducted using an NVIDIA A100 GPU.

Table 4. Skip indices selected in FLUX and SANA for the CustomConcept101 datasets. **Encoder** denotes the pre-trained encoder to extract image embedding for semantic distance.

Model-Encoder	Skip ratio	$(\mathbf{n}^*, \mathbf{m}^*)$
FLUX-DINO	30%	(7, 10)
	40%	(13, 10)
	50%	(19, 10)
SANA-DINO	30%	(4, 2)
	40%	(4, 4)
	50%	(5, 5)
FLUX-CLIP	30%	(7, 10)
	40%	(11, 12)
	50%	(18, 11)
SANA-CLIP	30%	(2, 4)
	40%	(4, 4)
	50%	(5, 5)

6.2. Details of Baselines

All baseline implementations were based on the Diffusers library. For a fair comparison, all models were trained under the same settings: a batch size of 1, a gradient accumulation step of 4, and 500 training iterations using AdamW [26] optimizer.

Textual Inversion [13]. We used the official textual inversion implementation based on FLUX from the Diffusers library. We set the number of trainable tokens to four and the learning rate to 1e-3. Although various hyperparameter settings were explored, we report the results using the configuration that yielded the most reasonable performance. Nevertheless, we observed that textual inversion alone was insufficient for achieving satisfactory personalization with DiT-based T2I models. Since SANA does not provide a dedicated implementation for textual inversion, we implemented it manually by referencing the FLUX-based codebase. Specifically, in the PyTorch implementation, the embedding layer was included in the optimizer, and at each training iteration, all tokens except the trainable ones were overwritten with their original values. Although including only the trainable tokens in the optimizer could reduce the optimizer state memory footprint, we adhered to the official FLUX-based textual inversion implementation provided by the Diffusers library for consistency in our experiments. Consequently, as shown in Table 7, SANA, which employs a large language model as its text encoder, requires a substantial optimizer state memory due to the embedding layer containing approximately 260,000 tokens. Nevertheless, a significant drawback of this approach is that the text encoder must remain loaded on the GPU throughout training, and the backpropagation process must extend all the way to the embedding layer of the text encoder. As a result, this method incurs substantially higher memory consumption compared to approaches such as LoRA or our proposed

Table 5. Evaluation prompt list we used.

LIVE Prompt List	Non-LIVE Prompt List
a <*> <subject> in the jungle	a <*> <subject> in the jungle
a <*> <subject> in the snow	a <*> <subject> in the snow
a <*> <subject> on the beach	a <*> <subject> on the beach
a <*> <subject> on a cobblestone street	a <*> <subject> on a cobblestone street
a <*> <subject> on top of pink fabric	a <*> <subject> on top of pink fabric
a <*> <subject> on top of a wooden floor	a <*> <subject> on top of a wooden floor
a <*> <subject> with a city in the background	a <*> <subject> with a city in the background
a <*> <subject> with a mountain in the background	a <*> <subject> with a mountain in the background
a <*> <subject> with a blue house in the background	a <*> <subject> with a blue house in the background
a <*> <subject> on top of a purple rug in a forest	a <*> <subject> on top of a purple rug in a forest
a <*> <subject> wearing a red hat	a <*> <subject> with a wheat field in the background
a <*> <subject> wearing a santa hat	a <*> <subject> with a tree and autumn leaves in the background
a <*> <subject> wearing a rainbow scarf	a <*> <subject> with the Eiffel Tower in the background
a <*> <subject> wearing a black top hat and a monocle	a <*> <subject> floating on top of water
a <*> <subject> in a chef outfit	a <*> <subject> floating in an ocean of milk
a <*> <subject> in a firefighter outfit	a <*> <subject> on top of green grass with sunflowers around it
a <*> <subject> in a police outfit	a <*> <subject> on top of a mirror
a <*> <subject> wearing pink glasses	a <*> <subject> on top of the sidewalk in a crowded street
a <*> <subject> wearing a yellow shirt	a <*> <subject> on top of a dirt road
a <*> <subject> in a purple wizard outfit	a <*> <subject> on top of a white rug
a red <*> <subject>	a red <*> <subject>
a purple <*> <subject>	a purple <*> <subject>
a shiny <*> <subject>	a shiny <*> <subject>
a wet <*> <subject>	a wet <*> <subject>
a cube shaped <*> <subject>	a cube shaped <*> <subject>

method.

DreamBooth [36]. We utilized the official DreamBooth implementation from the Diffusers library. A learning rate of 5e-5 was used for fine-tuning FLUX, and 1e-4 for SANA. Prior preservation loss was also employed during training.

LoRA [15]. We used the official LoRA implementation from the Diffusers library. We used a learning rate of 1e-4 and set the LoRA rank to 128 for FLUX and 512 for SANA.

LISA [29]. We implemented LISA by referencing the original paper and modifying the Diffusers LoRA codebase accordingly. All experiments were conducted using the same LoRA configuration and hyperparameters for fair comparison.

LoRA-FA [51]. LoRA-FA was implemented based on its original paper using the Diffusers LoRA code as a foundation. We ensured consistent LoRA settings and hyperparameters to maintain a fair experimental setup.

HollowedNet [6]. As the official implementation was not publicly available, we reimplemented HollowedNet based on the paper and guidance from the authors. The original method was demonstrated only on U-Net-based Stable Diffusion [35]; thus, we modified it to support transformer block skipping by incorporating our residual feature precomputing method. Since HollowedNet fine-tunes only shallow layers in U-Net, we applied a similar strategy to DiT-based T2I models. For the fair comparison, we used the same hyperparameter settings including the learning rate, batch size, and rank as those used for LoRA.

6.3. Prompt List for Evaluation.

Table 5 shows the prompt list we utilized to generate images for evaluation. The text prompts are identical to the text prompts provided in the DreamBooth dataset [36].

6.4. Details of User Study

We assessed user preferences between LoRA [15], which shows the best performance across the baselines, and our proposed model based on FLUX. Participants were instructed to select the image with better subject and text fidelity, or choose 'undecided' if the two results were comparable. Specifically, subject fidelity was evaluated based on similarity to the training image, while text fidelity was judged by how well the image reflected the input prompt. The user study was conducted over 60 sets. The final results were calculated by averaging the preferences across all participants.

7. Training Memory

Table 6 and 7 report detailed training memory usage for FLUX- and SANA-based baselines as well as our proposed method. To isolate memory usage relevant to trainable components, we precomputed text features when the text encoder was not updated and encoded training images into latent space using a VAE in advance; both were excluded from memory calculations (except for textual inversion). We conducted training using `bfloat16` mixed precision, which is commonly adopted for FLUX and SANA for memory-efficient training.

As shown in the tables, we categorize training memory

Table 6. Detailed training memory comparison and TFLOPs with baselines based on **FLUX**. Inference resolution is 1024×1024

Method	Skip Ratio	Training Resolution	Training Memory	Parameter Memory	Optimizer State Memory	Other Memory	Training TFLOPs	Precompute TFLOPs
Textual Inversion	–	512×512	42.15 GiB	31.27 GiB	**0.14** GiB	10.74 GiB	44.57	–
DreamBooth	–	512×512	68.96 GiB	22.17 GiB	22.52 GiB	24.27 GiB	158.77	–
LoRA	–	512×512	35.99 GiB	22.84 GiB	1.34 GiB	11.82 GiB	41.67	–
LISA	–	512×512	35.98 GiB	22.84 GiB	1.34 GiB	9.33 GiB	41.67	–
LoRA-FA	–	512×512	32.48 GiB	22.84 GiB	0.67 GiB	8.97 GiB	41.32	–
HollowedNet	30 %	512×512	24.11 GiB	15.62 GiB	1.04 GiB	7.46 GiB	24.35	19.85
HollowedNet	40 %	512×512	18.40 GiB	11.91 GiB	0.79 GiB	5.70 GiB	18.44	19.85
HollowedNet	50 %	512×512	12.23 GiB	7.93 GiB	0.54 GiB	3.76 GiB	11.80	19.85
Ours	30 %	256×256	20.78 GiB	15.54 GiB	0.88 GiB	4.37 GiB	14.59	9.94
Ours	40 %	256×256	15.61 GiB	11.58 GiB	0.56 GiB	3.46 GiB	12.30	9.94
Ours	50 %	256×256	**10.42** GiB	**7.63** GiB	0.25 GiB	**2.55** GiB	**9.96**	9.94

Table 7. Detailed training memory comparison and TFLOPs with baselines based on **SANA**. Inference resolution is 1024×1024

Method	Skip Ratio	Training Resolution	Training Memory	Parameter Memory	Optimizer State Memory	Other Memory	Training TFLOPs	Precompute TFLOPs
Textual Inversion	–	1024×1024	15.31 GiB	7.86 GiB	2.20 GiB	5.25 GiB	8.91	–
DreamBooth	–	1024×1024	24.69 GiB	2.99 GiB	5.98 GiB	15.72 GiB	13.34	–
LoRA	–	1024×1024	8.35 GiB	3.50 GiB	1.03 GiB	3.83 GiB	6.41	–
LISA	–	1024×1024	6.67 GiB	3.50 GiB	**0.10** GiB	3.07 GiB	6.00	–
LoRA-FA	–	1024×1024	7.40 GiB	3.50 GiB	0.51 GiB	3.39 GiB	6.16	–
HollowedNet	30 %	1024×1024	5.90 GiB	2.48 GiB	0.72 GiB	2.70 GiB	4.48	2.91
HollowedNet	40 %	1024×1024	5.08 GiB	2.14 GiB	0.62 GiB	2.33 GiB	3.84	2.91
HollowedNet	50 %	1024×1024	4.26 GiB	**1.79** GiB	0.51 GiB	1.95 GiB	3.19	2.91
Ours	30 %	512×512	4.29 GiB	2.48 GiB	0.72 GiB	1.09 GiB	1.26	0.82
Ours	40 %	512×512	3.69 GiB	2.14 GiB	0.62 GiB	0.94 GiB	1.08	0.82
Ours	50 %	512×512	**3.10** GiB	**1.79** GiB	0.51 GiB	**0.79** GiB	**0.90**	0.82

into three components: (1) **Parameter Memory**, (2) **Optimizer State Memory**, and (3) **Other Memory**, which includes the combined memory from forward (activations) and backward (gradients), excluding parameters and optimizer states. Temporary memory was negligible and thus not considered in our measurements.

As shown in Tables 6 and 7, the proposed method, demonstrates the lowest consumption of training memory and TFLOPs among the compared approaches. In the case of FLUX, the presence of two types of transformer blocks, such as single attention block and dual attention block, results in differing block indices for skipping, which leads to discrepancies in parameter memory usage compared to other HollowedNet variants. Notably, the technique of block skipping with residual feature precomputing significantly reduces parameter memory, optimizer state memory, and other memory. Furthermore, dynamic patch sampling, while not affecting parameter or optimizer state memory, substantially decreases both forward and backward memory usage as well as TFLOPs.

In contrast, textual inversion requires continuous forward and backward passes through the text encoder during training, preventing precomputation of text prompts (*e.g.*, "a photo of sks dog") and thereby increasing parameter memory requirements. PEFT methods such as LISA and LoRA-FA reduce optimizer state memory but do not significantly impact overall training memory. Although HollowedNet achieves considerable memory savings, it suffers from a substantial performance drop, as described in the main paper. As it is based on a U-Net architecture, we integrated our residual feature precomputing technique with HollowedNet. However, due to the empirical nature of U-Net-based methods, HollowedNet fails to identify crucial layers for personalization.

8. Additional Results

8.1. CustomConcept101 Results

Table 8 and Table 9 present the quantitative results on the CustomConcept101 dataset. We include only the models

Table 8. Comparison with baselines based on **FLUX** in CustomConcept101 dataset. Inference resolution is 1024×1024

Method	Skip Ratio	Training Res.	Training Mem.	DINO	CLIP-I	CLIP-T
LoRA	-	512×512	35.99 GiB	**0.6726**	**0.7961**	0.2937
LISA	-	512×512	35.98 GiB	0.5985	0.7491	**0.3085**
HollowedNet	30 %	512×512	24.11 GiB	0.4378	0.6570	0.3021
HollowedNet	40 %	512×512	18.40 GiB	0.4568	0.6662	0.3073
HollowedNet	50 %	512×512	12.23 GiB	0.4242	0.6553	0.3061
Ours	30 %	256×256	20.78 GiB	0.6455	0.7737	0.3004
Ours	40 %	256×256	15.61 GiB	0.6377	0.7696	0.3007
Ours	50 %	256×256	**10.42** GiB	0.6137	0.7513	0.3056

Table 9. Comparison with baselines based on **SANA** in CustomConcept101 dataset. Inference resolution is 1024×1024

Method	Skip Ratio	Training Res.	Training Mem.	DINO	CLIP-I	CLIP-T
LoRA	-	1024×1024	8.35 GiB	**0.6545**	0.7792	0.3096
LISA	-	1024×1024	6.67 GiB	0.4944	0.6970	**0.3213**
HollowedNet	30 %	1024×1024	5.90 GiB	0.6459	0.7708	0.2983
HollowedNet	40 %	1024×1024	5.08 GiB	0.6257	0.7537	0.2932
HollowedNet	50 %	1024×1024	4.26 GiB	0.5828	0.7191	0.2955
Ours	30 %	512×512	4.29 GiB	0.6522	**0.7826**	0.3040
Ours	40 %	512×512	3.69 GiB	0.6489	0.7810	0.3025
Ours	50 %	512×512	**3.10** GiB	0.6441	0.7765	0.3033

Table 10. Comparison with partial LoRA and gradient checkpointing using SANA.

Method	Training Mem.	DINO$^{\uparrow}$	CLIP-I$^{\uparrow}$	CLIP-T$^{\uparrow}$
LoRA	8.35 GiB	0.7374	0.8108	0.3254
Grad. Checkpointing	5.66 GiB	0.7374	0.8122	0.3241
Partial LoRA (50%)	7.20 GiB	0.6870	0.7874	0.3261
Ours (50%)	**3.10 GiB**	0.7277	0.8034	0.3177

that performed competitively in the DreamBooth setting, excluding Textual Inversion, DreamBooth, and LoRA-FA due to their relatively low performance. Similar to the results on the DreamBooth dataset, our method achieves performance comparable to LoRA. Notably, SANA (CLIP-I) achieves the highest performance even while skipping 30% of transformer blocks. In contrast, LISA and HollowedNet exhibit a significant performance drop when 50% of the blocks are skipped.

Fig. 11 and Fig. 12 provide qualitative comparisons between our method and FLUX-based baseline models on the DreamBooth and CustomConcept101 datasets, respectively. In particular, as shown in Fig. 12, while LISA fails to preserve the identity of the target subject, our method demonstrates both superior memory efficiency and competitive identity preservation and text fidelity compared to LoRA.

8.2. Comparison with Additional Baselines

We further evaluate our approach against two widely used memory-efficient baselines: gradient checkpointing and partial LoRA. Gradient checkpointing reduces peak activation memory by discarding intermediate activations during the forward pass and recomputing them only when needed during the backward pass. Partial LoRA mitigates the memory overhead of optimizer states by selectively fine-tuning only a subset of layers. For partial LoRA (50%), we selectively fine-tune only the 50% of blocks selected by our block selection algorithm. Table 10 shows the results of partial LoRA and gradient checkpointing using SANA model. While these techniques are straightforward to implement for memory reduction, they both necessitate that the entire base model parameters remain resident in GPU VRAM. In contrast, our method utilizes block skipping to actively offload inactive weights from the GPU, significantly reducing the weight memory footprint of SANA while maintaining comparable performance.

8.3. Ablation Results on Dynamic Patch Sampling

Our dynamic patch sampling mechanism adopts a 'Low-to-High' strategy, where the patch size is increased in proportion to the diffusion timestep t (i.e., patch size $\uparrow$ as t $\uparrow$), utilizing larger patches at higher noise levels to align with the structural-to-textural transition of the generative process. This principled approach captures coarse global struc-

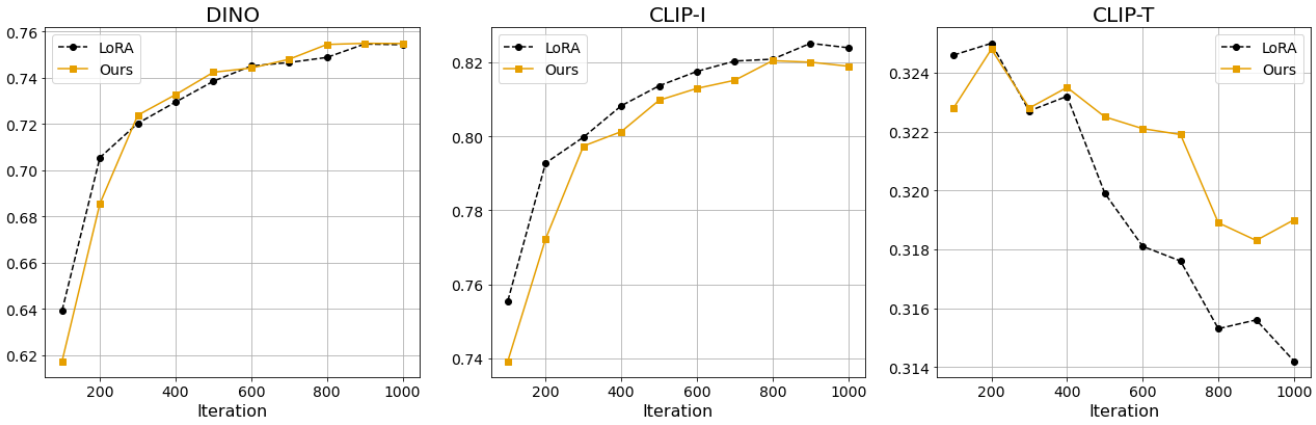

Figure 8. Performance comparison across training iterations for LoRA and our method.

tures via larger patches and refines fine details via smaller patches. To validate its effectiveness, we conducted additional ablation study using the FLUX model, comparing our method against 'High-to-Low' (patch size $\downarrow$ as $t \uparrow$) and 'Random' sampling baselines. As shown in Table 11, our strategy consistently achieves superior performance across all evaluation metrics, highlighting the advantage of our proposed dynamic patch sampling.

Table 11. Comparison of different patch sampling using FLUX.

Method (FLUX)	Training Res.	DINO$^\uparrow$	CLIP-I$^\uparrow$	CLIP-T$^\uparrow$
High-to-Low	256×256	0.7164	0.8093	0.3182
Random	256×256	0.7187	0.8068	0.3192
Ours	256×256	**0.7253**	**0.8099**	**0.3196**

8.4. Ablation Results on Memory Consumption

We provide a detailed ablation study on the memory consumption of our method, comparing the effects of Dynamic Patch Sampling and Block Skipping. As shown in Table 12, the baseline LoRA model requires 35.99 GiB of memory. Applying only Dynamic Patch Sampling reduces the memory usage to 30.48 GiB. In contrast, applying Block Skipping alone at levels of 30%, 40%, and 50% progressively reduces memory usage to 24.24 GiB, 18.84 GiB, and 13.40 GiB, respectively. Both methods individually reduce memory usage while maintaining performance. When used together, they offer further improvements in training memory efficiency.

8.5. Training Dynamics and Convergence Behavior

All methods were trained for the same number of iterations (500 iterations as reported in the main paper). To assess convergence behavior in detail, we additionally trained SANA using both LoRA and our method for 1,000 iterations, and report performance measured every 100 iterations in Fig. 8.

Table 12. Memory usage with dynamic patch sampling and block skipping applied.

Method (FLUX)	Training Res.	Memory (GiB)
LoRA	512x512	35.99
Dynamic Patch Sampling	256x256	30.48
Block Skipping (30%)	512x512	24.63
Block Skipping (40%)	512x512	18.76
Block Skipping (50%)	512x512	12.85
Ours (30%)	256x256	20.78
Ours (40%)	256x256	15.61
Ours (50%)	256x256	10.42

While LoRA exhibits slightly better performance in the early stages (up to 200 iterations), it fails to fully converge. Beyond 300 iterations, both methods continue to improve in terms of image fidelity, as reflected in the DINO and CLIP-I scores, and ultimately achieve comparable performance. These results demonstrate that our method converges at a similar rate to LoRA while maintaining competitive performance throughout the training process.

8.6. Training Time

We analyze the training time of our method with a 50% block skipping ratio applied to the SANA model over 100 iterations. Fig. 10 depicts the training time of our method relative to LoRA, highlighting the impact of block skipping and dynamic patch sampling. As shown in Fig. 10, we break down the total time spent across four stages: precomputing, saving features, loading features, and training. Even when accounting for precomputation and I/O overhead, our final method (41.75 s) achieves approximately 25% faster training time compared to LoRA (56.00 s). Furthermore, for FLUX, our method reduces training memory usage by

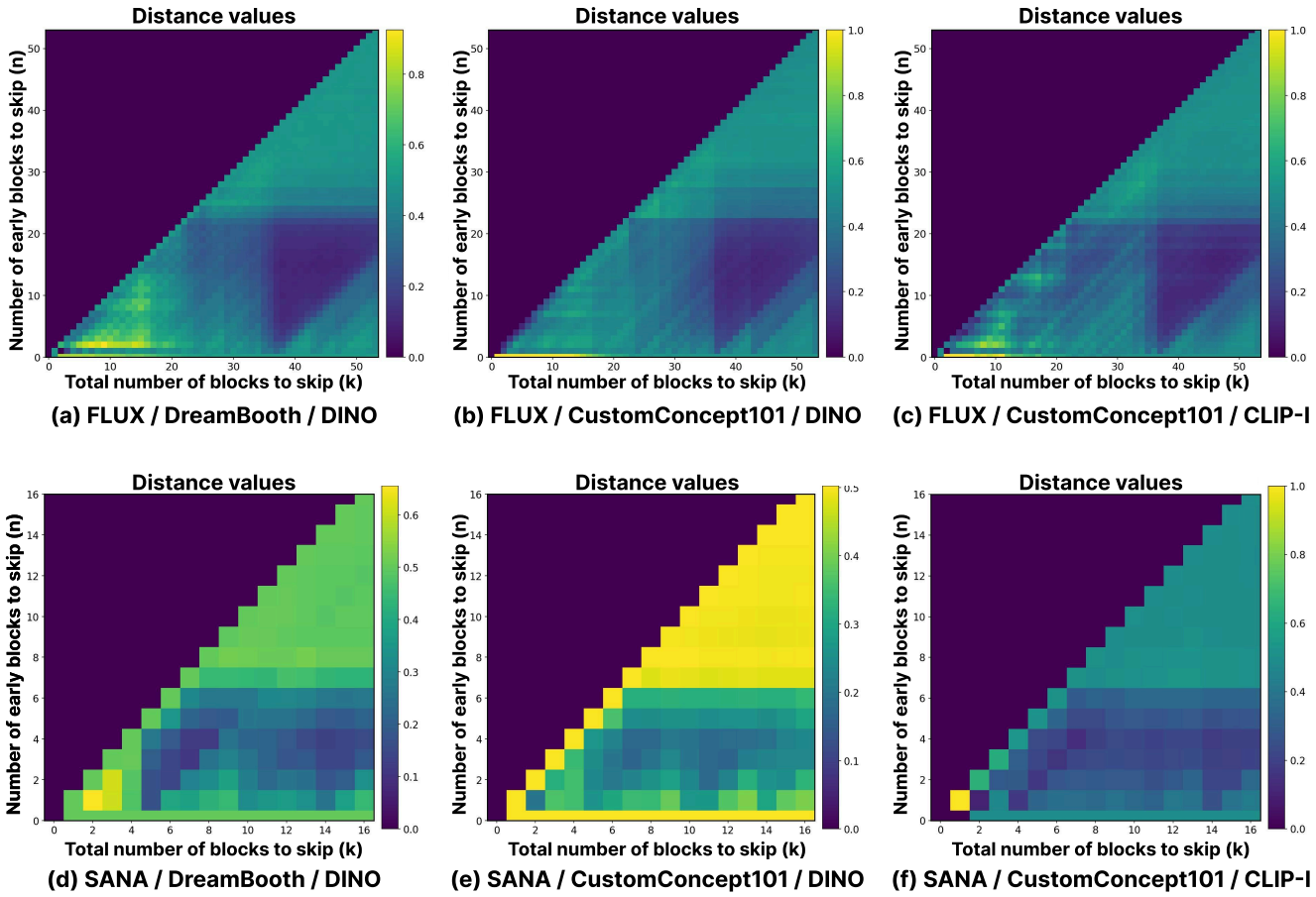

Figure 9. Visualization of average semantic distances for all combinations of n and m. Each subplot is labeled with the corresponding model, dataset, and pre-trained encoder. Since $m = k - n$, each column shares the same total number of skipped blocks k. Each value shows the average distance when cross-attention is masked on the first n and the last m blocks.

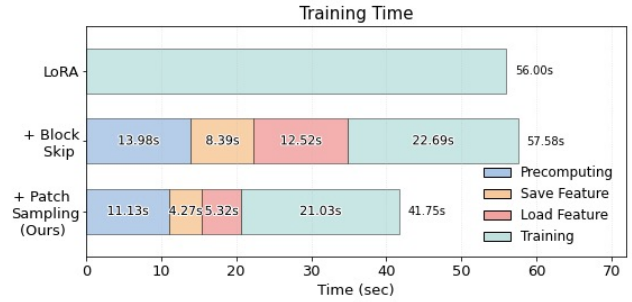

Figure 10. Comparison of training time between LoRA and our method including block skipping and dynamic patch sampling.

about 71% compared to the baseline, which is the primary contribution of this work.

8.7. Semantic Distances

Figure 9 visualizes the semantic distance for all combinations of n and m on the DreamBooth and CustomConcept101 datasets using the FLUX and SANA models. Each value represents the average semantic distance when the image-to-text cross-attention maps in the first n and last m blocks are masked. Fig. 9 shows that semantic distance increases when n exceeds approximately 23 in FLUX and 8 in SANA. This suggests that the middle blocks, when skipped, contribute more to the degradation of output quality, underscoring their importance in preserving subject identity. Furthermore, we conducted an ablation study on the encoder used for computing the proposed semantic distance. Table 4 shows that the index differences are minimal when using DINO similarity (DINO) versus CLIP image similarity (CLIP-I), showing that both measures yield similar semantic distance trends. CLIP text similarity (CLIP-T) was excluded from the analysis because it lacks the ability to accurately capture subject identity. This observation is further supported by Fig. 9 (b) and (c), as well as (e) and (f), where the patterns of semantic distance are consistent across DINO and CLIP-I scores.

9. Discussion and Limitations

Reducing Memory for Residual Feature precomputing. For both our method and HollowedNet [6], we excluded the memory used during the precomputing stage from the reported training memory. This is because, in DiT-based models, it is feasible to partially forward only specific layers. In particular, FLUX alone requires approximately 22.17 GiB of parameter memory, making it impractical to load the entire model onto the GPU simultaneously in on-device personalization scenarios.

To address this, we can perform partial forward passes for residual feature precomputing: only the necessary transformer blocks are loaded into the GPU to compute and store intermediate feature maps up to that point. The remaining blocks are then loaded sequentially to complete the precomputing of residual features. This staged strategy significantly reduces parameter memory usage during forward computation, which otherwise constitutes the largest portion of memory consumption.

Compatibility with Other Methods. To reduce training memory, prior works have leveraged techniques such as quantization, parameter-efficient fine-tuning (PEFT) methods like LoRA [15], and gradient checkpointing. Our method is compatible with these approaches and can be integrated to further reduce memory overhead. For example, as demonstrated by our integration with LoRA, our method can be extended to other PEFT techniques, such as DoRA [25], without loss of generality.

On-Device Deployment. Although actual demonstration on mobile or IoT device falls outside our current scope, which focuses on the fundamental algorithmic efficiency of DiT fine-tuning, our substantial reductions in VRAM and FLOPs significantly enhance edge feasibility. A practical consideration for on-device deployment is the non-trivial ROM required for storing precomputed features. While this overhead can be mitigated by adopting a periodic feature-saving strategy (*e.g.*, caching per iteration) rather than full-dataset pre-storage, such an approach introduces additional implementation complexity. We consider the co-optimization of ROM and VRAM for seamless on-device fine-tuning as a promising direction for future work.

Figure 11. Additional qualitative comparison with baselines on the DreamBooth dataset.

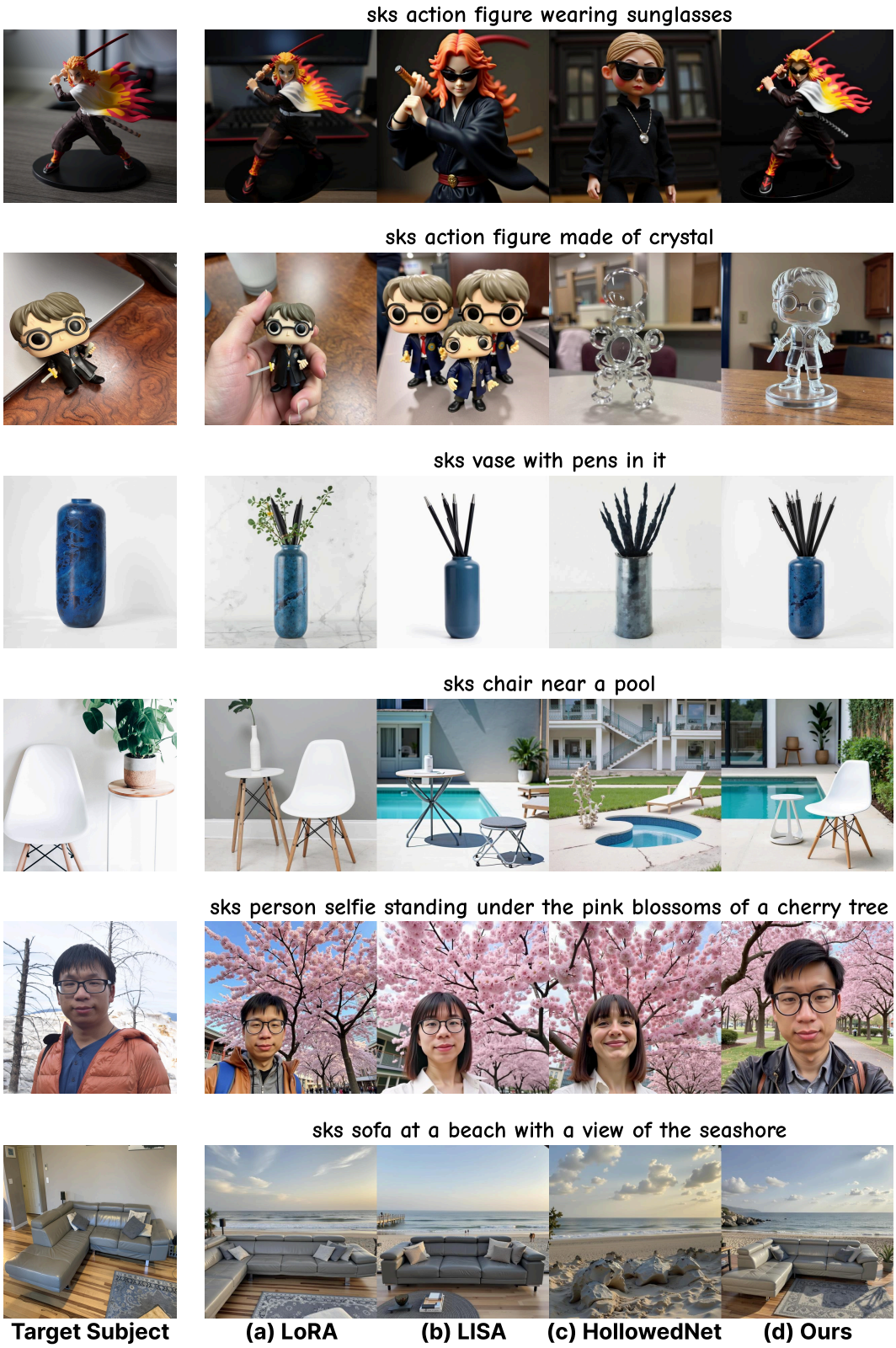

Figure 12. Qualitative comparison with baselines on the CustomConcept101 dataset.