Parallel Sequence Modeling via Generalized Spatial Propagation Network –Supplementary Material–

Contents

A. Stable-Context Condition

In this section, we present comprehensive mathematical proofs for the Stable-Context Condition introduced in Section 3.2 in the main paper. We revisit 2D linear propagation:

$$h_i^c = w_i^c h_{i-1}^c + \lambda_i^c \odot x_i^c, \quad i \in [1, n-1],\ c \in [0, C-1] \tag{1}$$

where i denotes the i_{th} row and $w_i \in \mathbb{R}^{n \times n}$ denotes the tri-diagonal matrix associating it to the $(i-1)_{tj}$ row. While λ_i denotes a vector to weigh x_i with element-wise product, it can also be formulated as a diagonal matrix with λ_i's original values being on the main diagonal. To expand Eq. (1), we denote $H \in \{h_i\}, i \in [0, N-1], N = n^2$ as the latent space where the propagation is carried out:

$$H_v = \begin{bmatrix} I & 0 & \cdots & \cdots & 0 \\ w_2 & \lambda_2 & 0 & \cdots & \cdots \\ w_3 w_2 & w_3 \lambda_2 & \lambda_3 & 0 & \cdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \vdots & \vdots & \cdots & \cdots & \lambda_{N-1} \end{bmatrix} X_v = G X_v, \tag{2}$$

Here, G is a lower triangular, $N \times N$ transformation matrix relating X and H, and X_v and H_v are vectorized sequences concatenating all the rows, i.e., $[x_0, x_1, ..., x_{n-1}]$ and $[h_0, h_1, ..., h_{n-1}]$, with length of N. We denote each $n \times n$ block as one sub-matrix, on setting $\lambda_0 = I$, the i^{th} constituent $n \times n$ sub-matrix of G_{ij} is:

$$G_{ij} = \begin{cases} \prod_{\tau=j+1}^{i} w_\tau \lambda_j, & j \in [0, i-1] \\ \lambda_j, & i = j \end{cases} \tag{3}$$

Thus, $W_{ij} = \prod_{\tau=j+1}^{i} w_\tau$ is crucial for maintaining dense connections between the elements in X and H. As noted in the main paper, $W_{ij} = \prod_{\tau=j+1}^{i} w_\tau$ must avoid collapsing to a matrix with a small norm to enable effective long-context propagation. This ensures that h_i can substantially contribute to h_j even over extended intervals, preserving the influence of distant elements. To achieve this, we design W_{ij} to satisfy $\sum_{j=0}^{n-1} W_{ij} = 1$, ensuring that each element in h_i is a weighted average of all the elements of $x'_j = \lambda_j x_j$. This design guarantees that the information in the j^{th} column is not diminished when propagated to the i^{th} column.

A.1. Long-context Condition

Theorem 1. *If all the tridiagonal matrices w_τ are row stochastic, then $\sum_{j=0}^{n-1} W_{ij} = 1$ is satisfied.*

Definition of Row Stochastic Matrix. A matrix T is row stochastic if:
- All elements are non-negative: $T_{ij} \geq 0$ for all i, j.
- The sum of the elements in each row is 1: $\sum_j T_{ij} = 1$ for all i.

Proof. Let A and B be two $n \times n$ row stochastic matrices. We need to show that their product $C = AB$ is also row stochastic.
Non-negativity: Since A and B are both nonnegative, all elements of their product C are also nonnegative:

$$C_{ik} = \sum_{j=1}^{n} A_{ij} B_{jk} \geq 0$$

because each $A_{ij} \geq 0$ and each $B_{jk} \geq 0$.
Row Sum: We need to show that the sum of the elements in each row of C is 1. Consider the i-th row sum of C:

$$\sum_{k=1}^{n} C_{ik} = \sum_{k=1}^{n} \sum_{j=1}^{n} A_{ij} B_{jk}$$

By changing the order of summation:

$$\sum_{k=1}^{n} C_{ik} = \sum_{j=1}^{n} A_{ij} \left(\sum_{k=1}^{n} B_{jk} \right)$$

Since B is row stochastic:

$$\sum_{k=1}^{n} B_{jk} = 1$$

Therefore:

$$\sum_{k=1}^{n} C_{ik} = \sum_{j=1}^{n} A_{ij} \cdot 1 = \sum_{j=1}^{n} A_{ij}$$

And since A is row stochastic:

$$\sum_{j=1}^{n} A_{ij} = 1$$

Hence:

$$\sum_{k=1}^{n} C_{ik} = 1$$

This shows that $C = AB$ is row stochastic.

Induction for Multiple Matrices. To extend this result to the product of several row stochastic matrices, we proceed by induction.
Base Case: The product of two row stochastic matrices is row stochastic, as shown above.
Inductive Step: Assume that the product of m row stochastic matrices $w_1 w_2 \cdots w_m$ is row stochastic. We need to show that $w_1 w_2 \cdots w_m w_{m+1}$ is also row stochastic.

By the induction hypothesis, $W = w_1 w_2 \cdots w_m$ is row stochastic. Since w_{m+1} is also row stochastic, the product $W w_{m+1}$ follows the properties proven above for the product of two row stochastic matrices.
Conclusion: By induction, the product of any finite number of row stochastic matrices is row stochastic. Since tridiagonal matrices form a subset of such matrices, the proof applies to them as well. This ensures that $\sum_{j=0}^{n-1} W_{ij} = 1$ holds for all i.

A.2. Stability Condition

Theorem 2. *The stability of Eq.* (1) *is ensured when all matrices w_τ are row stochastic.*

By ignoring c for simplicity, we re-write Eq. (1), where each $h_{k,i}$ is computed as the following. We will use $p_{k,i}$ in the proof of Theorem 2.

$$h_{k,i} = \lambda_{k,i} x_{k,i} + \sum_{k \in \mathbb{N}} p_{k,i} h_{k,i-1} \tag{4}$$

Proof. The stability of linear propagation refers to preventing responses or errors from growing unbounded and ensuring that gradients do not vanish during backpropagation, as described in [1]. Specifically, for a stable model, the norm of the temporal Jacobian $\frac{\partial h_i}{\partial h_{i-1}}$ should be less than or equal to 1. In our case, this requirement can be met by ensuring that the norm of each transformation matrix w_i satisfies:

$$\left\| \frac{\partial h_i}{\partial h_{i-1}} \right\| = \|w_i\| \leq \sigma_{\max} \leq 1,$$

where $\sigma_{\max}$ denotes the largest singular value of w_i. This condition ensures stability.

By Gershgorin's Circle Theorem, every eigenvalue σ of a square matrix w_i satisfies:

$$|\sigma - p_{i,i}| \leq \sum_{k=1, k \neq i}^{n} |p_{k,i}|, \quad i \in [1, n],$$

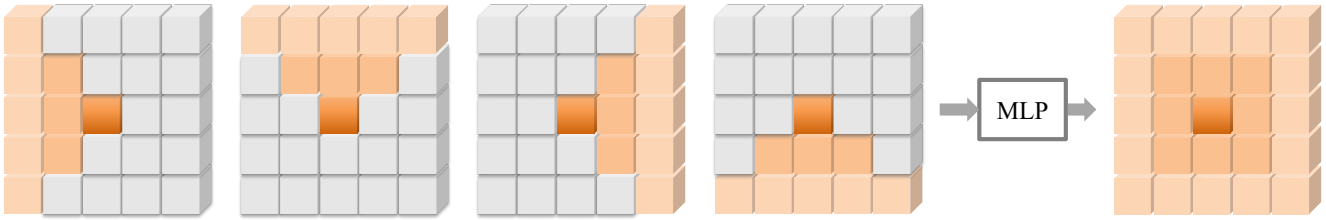

Figure 1. GSPN guarantees **Dense Pairwise Connections** via 3-way connection and 4-directional scanning, as introduced in Sec. 3.2 and detailed in Sec. A.3. The scanning of each direction corresponds to a lower triangular affinity matrix. The finally full matrix is obtained through a learnable linear aggregation, denoted as "MLP" in the figure.

which implies:

$$\sigma_{\max} \leq |\sigma - p_{i,i}| + |p_{i,i}| \leq \sum_{k=1}^{n} |p_{k,i}|.$$

If w_i is row stochastic, then:

$$\sum_{k=1}^{n} |p_{k,i}| = 1 \quad \text{for each row.}$$

Thus:

$$\sigma_{\max} \leq 1,$$

satisfying the stability condition. Making w_τ row stochastic ensures that the norm constraint $\|w_i\| \leq 1$ holds, which provides a sufficient condition for model stability, as presented in [1].

A.3. Guaranteeing Dense Pairwise Connections

In this section, we provide an intuitive rationale for selecting the 3-way connection, represented by a tri-diagonal matrix, as the minimal structure needed for dense matrix production through multiplication. To propagate non-zero entries throughout a matrix, sufficient connectivity is essential. Diagonal matrices, which have non-zero elements only on the main diagonal, fail to propagate non-zero values to off-diagonal positions, resulting in products that remain sparse. Similarly, matrices with non-zero elements limited to the main diagonal and one adjacent diagonal (e.g., upper or lower bi-diagonal matrices) restrict the spread of non-zero entries, maintaining a banded structure even after multiplication.

In contrast, tri-diagonal matrices, with non-zero elements on the main diagonal as well as the adjacent upper and lower diagonals, enable significant propagation of non-zero values during multiplication. This 3-way connection facilitates the spread of non-zero entries beyond the initial three bands. When two tri-diagonal matrices are multiplied, their non-zero entries extend further, and repeated multiplications lead to more comprehensive filling of the resulting matrix. Consequently, the tri-diagonal structure represents the minimal configuration with sufficient off-diagonal connections to eventually produce a dense matrix. This makes tri-diagonal matrices the simplest form capable of ensuring dense propagation in matrix products.

In Figure 1, we show the 3-way connection in 4 different directions. The scanning of each direction corresponds to a lower triangular affinity matrix, i.e., G in Eq. (2). A full dense matrix is obtained through learnable aggregation via a linear layer, as described in Sec. 4.2, guaranteeing dense pairwise connections with sub-linear complexity.

B. Attention Heatmaps of GSPN

The heatmap in Figure 2 presents a comprehensive analysis of our GSPN across four distinct directional scans, revealing a pronounced anisotropic behavior. Each directional heatmap represents a unique lower triangular affinity matrix. An additional aggregated heatmap provides a holistic view, synthesizing the directional insights into a unified representation that captures long-context and dense pairwise connections through a 3-way connection mechanism.

C. CUDA Implementation

To help readers better understand the algorithm before diving into the CUDA implementation details, we first present a PyTorch version of GSPN's forward and backward passes in Algorithm 1. This high-level implementation illustrates the core logic that will be parallelized in our CUDA kernels.

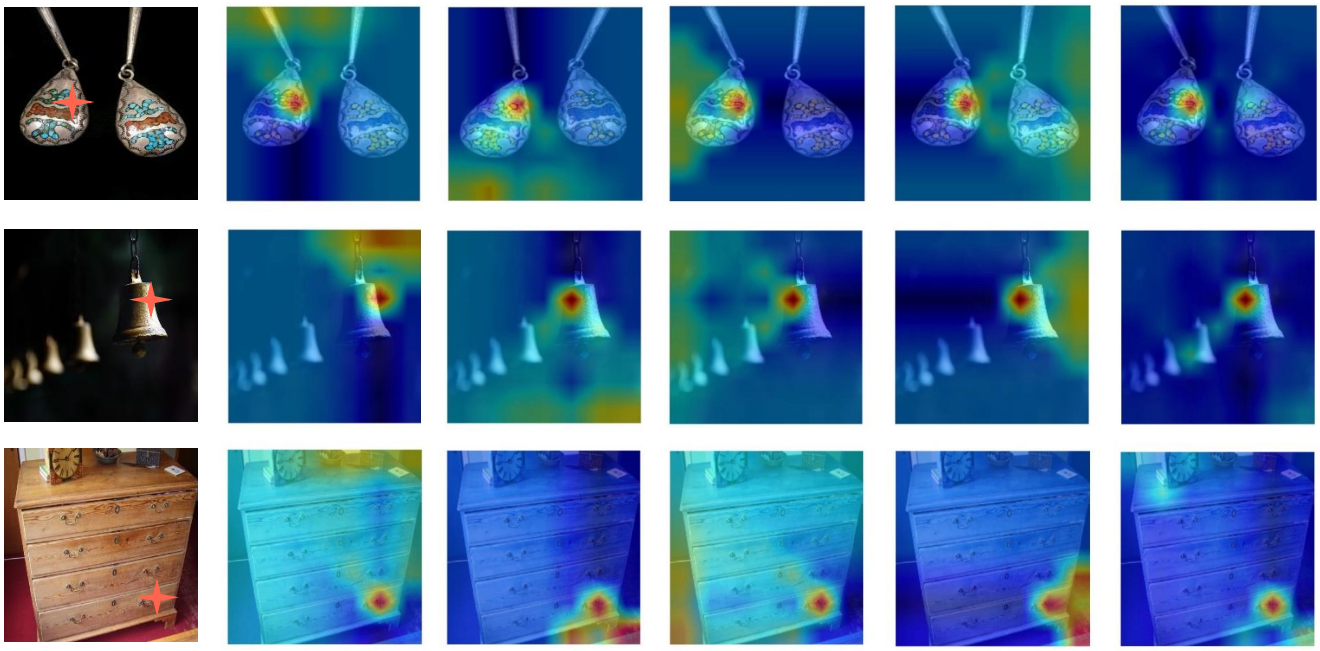

Figure 2. Illustration of heatmaps for the query patch (marked with an orange star) along different directions and the averaged results.

This pseudo-code shows a straightforward two-loop implementation in PyTorch style. While the outer loop over width must be sequential due to dependencies, the inner loop over height can actually be parallelized since each h-index computation within a column only depends on values from the previous column. This observation motivates our CUDA implementation, which parallelizes the height dimension computation while maintaining the necessary sequential processing along the width dimension.

We implement a highly-parallel computation for GSPN on CUDA-enabled GPUs, consisting of forward and backward passes. The detailed process is shown in Algorithm 2 and Algorithm 3.

Forward Pass. For each position (n, c, h, w) in the 4D tensor, we compute three directional connections (diagonal-up, horizontal, diagonal-down) using gates $G1$, $G2$, and $G3$. These operations are equivalent to matrix multiplication between h and w dimensions (Figure 2). The computation is divided into g groups, where $g = 1$ indicates global GSPN, and $g > 1$ indicates local GSPN. The final hidden state H combines input transformation $x_{hype} = BX$ with directional connections h_{hype}.

Backward Pass. Gradients are computed for all inputs $(X, B, G1, G2, G3)$ by reverse propagation. For each position, gradients flow from future timesteps into h_{diff}. Input gradients X_{diff} are computed via B values, while gate gradients $(G1_{diff}, G2_{diff}, G3_{diff})$ use error terms and previous hidden states.

Both passes leverage CUDA's parallel processing by distributing computations across threads. As the inner loop operates in parallel, GSPN's complexity is determined by the outer loop, resulting in $\mathcal{O}(\sqrt{N})$ complexity.

D. Limitation

The main limitation of the current GSPN framework lies in the optimization of memory access in our customized CUDA kernel implementation. The hidden vector H, frequently accessed by multiple threads, is stored in global memory without utilizing shared memory, leading to inefficient memory access patterns, increased latency, and higher contention for global memory bandwidth, particularly at high resolutions. Additionally, the lack of coalesced memory access and reliance on redundant index computations further degrade performance, especially as channel and batch sizes increase.

While efficiency analysis in Figure 1 highlights GSPN's scalability, demonstrating significant advantages over transformer-based and linear attention approaches for image resolutions beyond 2K, it reveals limited efficiency gains for low-resolution inputs. This discrepancy stems from implementation inefficiencies in our CUDA kernel, such as suboptimal value access and the absence of shared memory optimization. Addressing these bottlenecks is critical for fully leveraging GSPN's theoretical efficiency across a wider range of input sizes.

Algorithm 1 Pseudocode of GSPN in a PyTorch-like Style.

```
# Input parameters:
# X: Input feature tensor of shape (batch_size, channels, height, width)
# B: Input transformation matrix of shape (batch_size, channels, height, width)
# G1: Top-left gate weights controlling diagonal-up connections
# G2: Left gate weights controlling horizontal connections
# G3: Bottom-left gate weights controlling diagonal-down connections
# All gates G1,G2,G3 have shape (batch_size, channels, height, width)

batch_size, channels, height, width = X.size()
H = torch.zeros_like(X)

for w in range(width):
   for h in range(height):
      # Get current inputs
      x_t = X[..., h, w] # Current input features
      b_t = B[..., h, w] # Current transformation weight

      if w > 0:
         # Top-left connection (h-1, w-1)
         g1 = G1[..., h, w] if h > 0 else 0
         h1_prev = H[..., h-1, w-1].clone() if h > 0 else 0
         h1_gated = g1 * h1_prev if h > 0 else 0

         # Left connection (h, w-1)
         g2 = G2[..., h, w]
         h2_prev = H[..., h, w-1].clone()
         h2_gated = g2 * h2_prev

         # Bottom-left connection (h+1, w-1)
         g3 = G3[..., h, w] if h < height-1 else 0
         h3_prev = H[..., h+1, w-1].clone() if h < height-1 else 0
         h3_gated = g3 * h3_prev if h < height-1 else 0

         h_sum = h1_gated + h2_gated + h3_gated
      else:
         h_sum = 0

      H[..., h, w] = b_t * x_t + h_sum

return H
```

E. Parallel Time and Memory Complexity Analysis

In this section, we provide a detailed analysis of the computational complexity for various attention mechanisms and propagation methods used in handling 2D data, focusing on their time and memory efficiency.

Softmax Attention. Softmax attention is the most classical attention model. It computes full pairwise interactions between all N pixels in a 2D grid, resulting in a total sequential work of $O(N^2 \cdot d)$, where d is the feature dimension. While its ideal parallel time complexity is $O(1)$ due to independent pairwise computations, practical parallel time complexity is $O\left(\frac{N^2 \cdot d}{P}\right)$, where P represents the number of available GPU cores, and its memory complexity is $O(N^2)$.

Linear Attention. Linear attention reduces complexity by computing $Q(K^T V)$, which avoids the need for full pairwise interactions seen in softmax attention. This factorization allows for a linear computational cost of $O(N \cdot d)$, where N is the total number of pixels or tokens, and d is the feature dimension. The total sequential work of $O(N \cdot d)$ arises because each element in Q interacts with a transformed version of V (i.e., $K^T V$), which can be computed in parallel across N. The ideal parallel time complexity is $O(1)$ because each computation for the final result can be theoretically performed independently if there are enough processing cores. The practical parallel time, $O\left(\frac{N \cdot d}{P}\right)$, depends on the available GPU cores P, which distribute the computation workload. Memory complexity is $O(N \cdot d)$, as only intermediate vectors for Q, K, and V need to be stored during computation, making it significantly more efficient than softmax attention for large-scale data processing.

Algorithm 2 Forward Pass CUDA Implementation

Require: Input tensors X, B, G1, G2, G3
Require: width, kNItems
Ensure: Output tensor H

```
g = ⌈width / kNItems⌉
count = g × height × channels × num
for t = 0 to kNItems - 1 do
    parfor index = 0 to count - 1 do
        Calculate n, c, h, k from index
        w = k × nitems + t
        x_data = X[n,c,h,w]
        b_data = B[n,c,h,w]
        h1 = G1[n,c,h,w,h-1,w-1] × H[n,c,h-1,w-1]
        h2 = G2[n,c,h,w,h,w-1] × H[n,c,h,w-1]
        h3 = G3[n,c,h,w,h+1,w-1] × H[n,c,h+1,w-1]
        h_hype = h1 + h2 + h3
        x_hype = b_data × x_data
        H[n,c,h,w] = x_hype + h_hype
    end parfor
end for
```

Algorithm 3 Backward Pass CUDA Implementation

Require: Input tensors X, B, G1, G2, G3, H, H_diff
Require: width, kNItems
Ensure: Output tensors X_diff, B_diff, G1_diff, G2_diff, G3_diff, H_diff

```
g = ⌈width / kNItems⌉
count = g × height × channels × num
for t = 0 to kNItems - 1 do
    parfor index = 0 to count - 1 do
        Calculate n, c, h, k from index
        w = width - 1 - k × nitems - t
        h_diff = H_diff[n,c,h,w]
        Update h_diff with future timestep contributions
        H_diff[n,c,h,w] = h_diff
        X_diff[n,c,h,w] = B[n,c,h,w] × h_diff
        B_diff[n,c,h,w] = X[n,c,h,w] × h_diff
        G1_diff[n,c,h,w,h-1,w-1] = h_diff × H[n,c,h-1,w-1]
        G2_diff[n,c,h,w,h,w-1] = h_diff × H[n,c,h,w-1]
        G3_diff[n,c,h,w,h+1,w-1] = h_diff × H[n,c,h+1,w-1]
    end parfor
end for
```

Method	Total Seq.	Ideal Parallel TC	Practical Parallel TC T_p	MC
Softmax Attention	$O(N^2 \cdot d)$	$O(1)$	$O\left(\frac{N^2 \cdot d}{P}\right)$	$O(N^2)$
Linear Attention	$O(N \cdot d)$	$O(1)$	$O\left(\frac{N \cdot d}{P}\right)$	$O(N \cdot d)$
Mamba	$O(N \cdot d)$	$O(N)$	$O(N \cdot d)$	$O(d)$
GSPN-global	$O(4N \cdot d)$	$O(4\sqrt{N})$	$O\left(\frac{4N \cdot d}{P}\right)$	$O(4\sqrt{N} \cdot d)$
GSPN-local	$O(\frac{4N \cdot d}{g})$	$O(\frac{4\sqrt{N}}{g})$	$O\left(\frac{4N \cdot d}{g \cdot P}\right)$	$O(\frac{4\sqrt{N} \cdot d}{g})$

Table 1. Complexity analysis for different attention mechanisms and propagation methods. "TC" denotes time complexity, and "MC" denotes memory complexity.

Mamba. Mamba performs sequential pixel-to-pixel propagation, where each pixel depends on the result of its predecessor, leading to a total sequential work complexity of $O(N \cdot d)$, where N is the number of pixels and d is the feature dimension. While the strict sequential dependency limits parallelism across a single sequence, practical parallel time complexity can be expressed as $O\left(\frac{N \cdot d}{P}\right)$ when P processing units are utilized, particularly in scenarios where multiple sequences or batches are processed concurrently. Memory complexity remains minimal at $O(d)$, as only the current and previous pixel states need to be stored. While Mamba is memory-efficient, its sequential nature within a single sequence makes it less scalable for high-resolution 2D data compared to more parallel-friendly methods.

GSPN. GSPN processes data using line scan, where each pixel in a row/column depends only on its adjacent pixels from the previous row/column through a tridiagonal matrix structure. This design leads to a total sequential work complexity of $O(4N \cdot d)$, where N is the total number of pixels and d is the feature dimension. The ideal parallel time complexity is $O(4\sqrt{N})$, as each row can be processed in parallel, but rows are processed sequentially. The practical parallel time is $O\left(\frac{4N \cdot d}{P}\right)$. Memory complexity is $O(4\sqrt{N} \cdot d)$, as only the current and previous rows are stored during computation. GSPN's line-wise parallelism and reduced sequence length provide a balance between computational efficiency and scalability, making it suitable for handling high-resolution 2D data more effectively than fully sequential methods like Mamba.

F. Details of the Overall Architecture

Image Classification. We implement a four-level hierarchical architecture for image classification. The initial stem module processes the $H \times W \times 3$ input through two consecutive 3×3 convolutions (stride 2, padding 1). The first convolution outputs half the channels of the second, with each convolution followed by LN and GELU activation. The subsequent levels implement a progressive feature hierarchy, where levels 1-2 employ local GSPN blocks for efficient processing at higher resolutions ($H/4 \times W/4$, $H/8 \times W/8$), while levels 3-4 utilize global GSPN blocks for contextual integration at lower resolutions ($H/16 \times W/16$, $H/32 \times W/32$). Between levels, dedicated downsampling layers perform spatial reduction through 3×3 convolutions (stride 2, padding 1) followed by LN, where each downsampling operation halves the spatial dimensions while maintaining the hierarchical feature representation.

Table 2. **Details of GSPN models for image classification. G** = global GSPN, while **L** = local GSPN.

Model	#Layers	Hidden size	MLP ratio	Mixing type
GSPN-T	[2, 2, 7, 2]	[96, 192, 384, 768]	[4.0, 4.0, 4.0, 4.0]	[L, L, G, G]
GSPN-S	[3, 3, 9, 3]	[108, 216, 432, 864]	[4.0, 4.0, 4.0, 4.0]	[L, L, G, G]
GSPN-B	[4, 4, 15, 4]	[120, 240, 480, 960]	[4.0, 4.0, 4.0, 4.0]	[L, L, G, G]

Class-conditional Generation. We present an elegant and versatile architecture for class-conditional generation models in Figure 3 (b). Specifically, GSPN parameterizes the noise prediction network $\epsilon_\theta(x_t, t, y)$, which estimates the noise introduced into the partially denoised image, considering the timestep t, condition y, and noised image x_t as inputs. The initial stage transforms the input image into flattened 2-D patches, subsequently converting them into a sequence of tokens with dimension D through linear embedding. Note that there is not learnable positional embeddings in the sequence. We explore patch sizes of $p = 2$ in the design space but would hold for any patch size. Beyond noised image inputs, the model incorporates additional conditional information such as noise timesteps t and conditions y like class labels or natural language. To integrate these, we append the vector embeddings of timestep t and class condition as supplementary tokens in the input sequence. These tokens are added to image tokens. The hidden states from the main branch and the skip branch are concatenated and linearly projected before input to the subsequent GSPN module. The final GSPN module decodes the hidden state sequence into noise prediction and diagonal covariance prediction, preserving the original spatial input dimensions. A standard linear decoder applies the final layer normalization and linearly transforms each token, subsequently rearranging the decoded tokens to the original spatial layout.

G. More Samples Generated from GSPN

We present additional qualitative results to demonstrate GSPN's generation capabilities. Figure 3 showcases diverse high-quality images generated by GSPN-XL for class-conditional generation. Furthermore, Figure 4 - Figure 6 display generation results from our distilled GSPN models, trained on both SD-v1.5 and SD-XL. The samples exhibit comparable visual quality to their transformer-based counterparts while benefiting from GSPN's efficient architecture.

Table 3. **Details of GSPN models for class-conditional generation.** We follow DiT model configurations for the Base (B), Large (L) and XLarge (XL) variants. Steps/sec is measured on ImageNet 256×256 generation with patch size equal to 2 with an A100.

Model	#Layers	Hidden size	#Params (M)	Steps/s
GSPN-B	30	900	137	2.78
GSPN-L	56	1200	443	1.15
GSPN-XL	56	1500	690	0.88

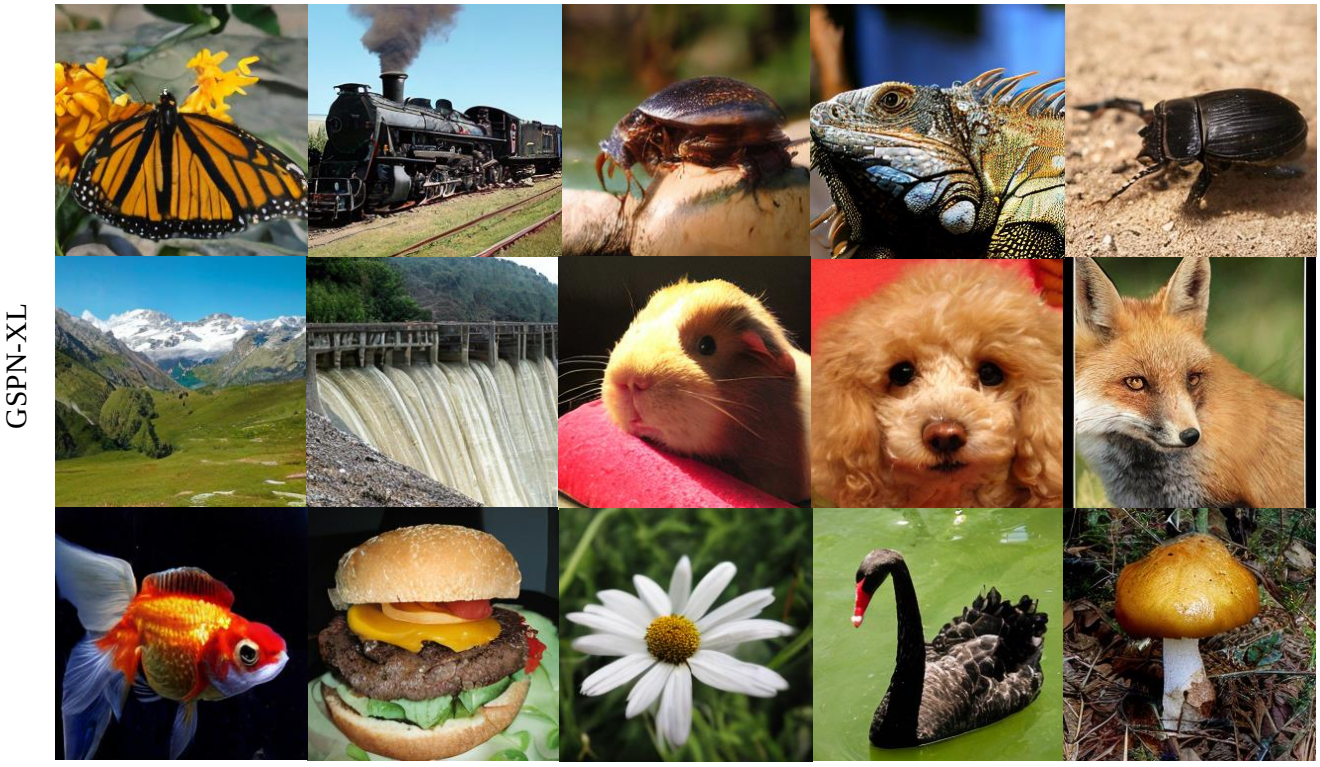

Figure 3. Qualitative results of class-conditional generation from our 256×256 resolution GSPN-XL/2 models. To ensure consistency with the quantitative results reported in the paper, classifier-free guidance was NOT applied to any of the outputs.

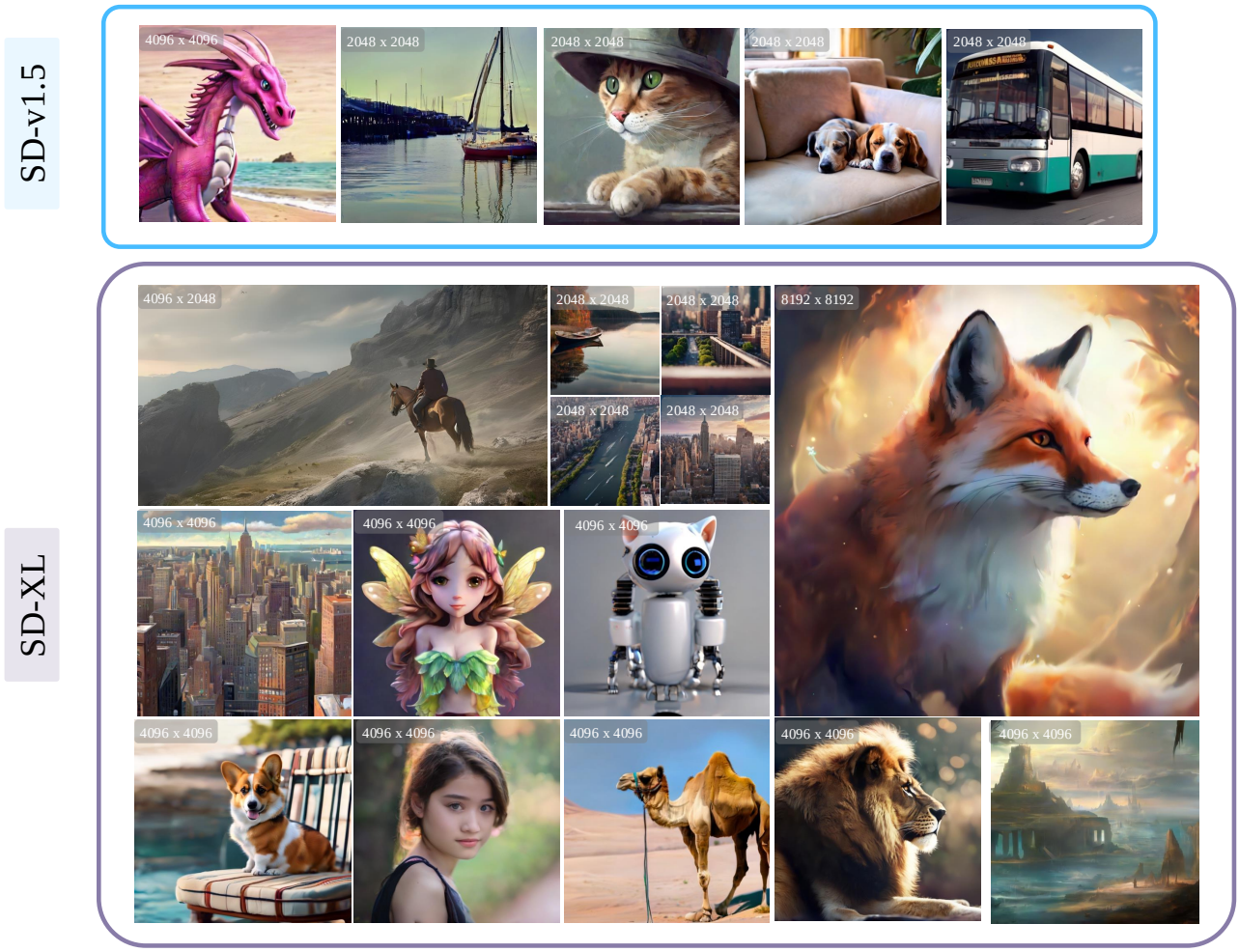

Figure 4. Examples of GSPN at various higher resolutions based upon SD-v1.5 and SDXL. GSPN enable to synthesize images up to a resolution of 16384×8192 using a single A100. Best viewed in PDF with zoom.

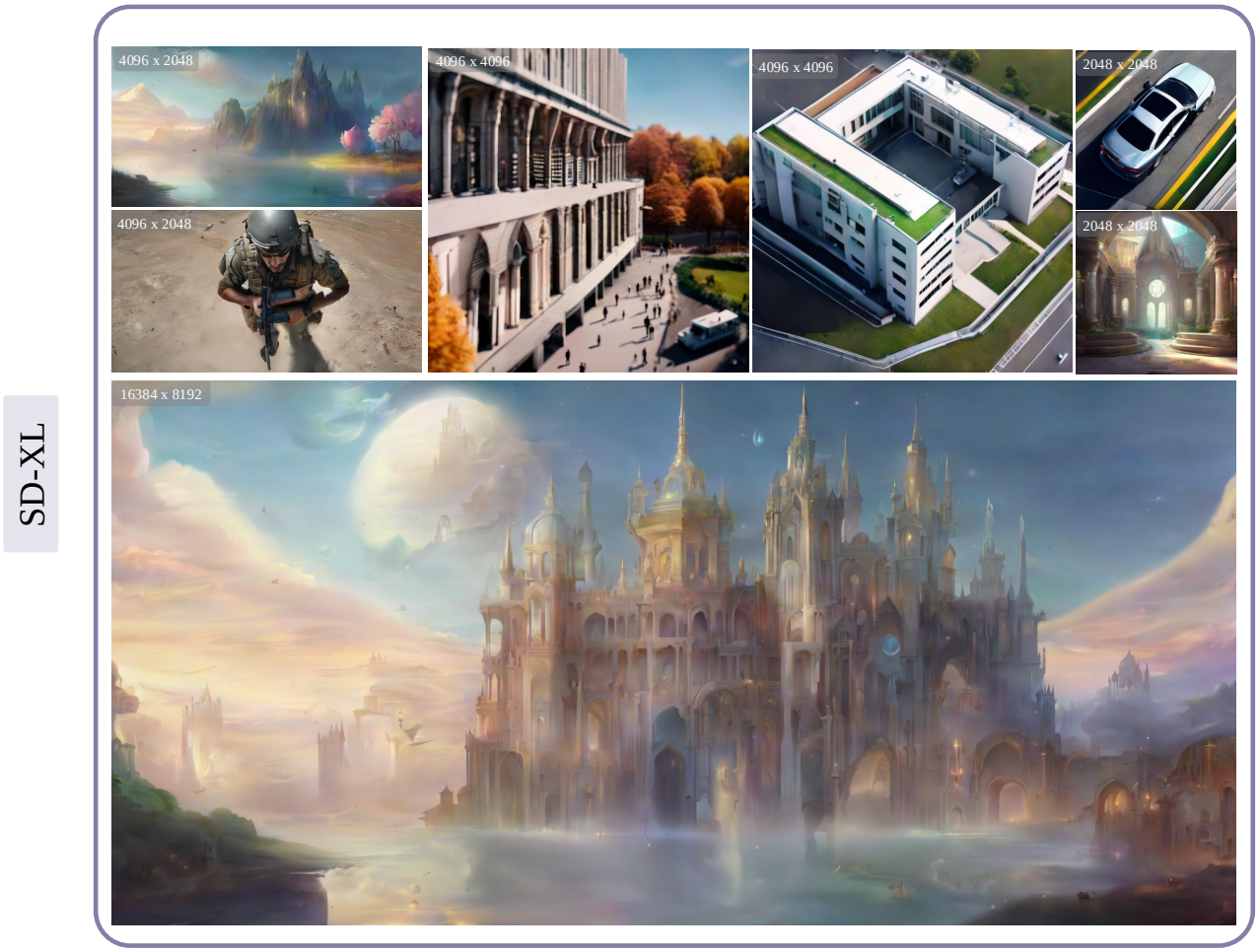

Figure 5. Examples of GSPN at various higher resolutions based upon SDXL. GSPN enable to synthesize images up to a resolution of 16384×8192 using a single A100. Best viewed in PDF with zoom.

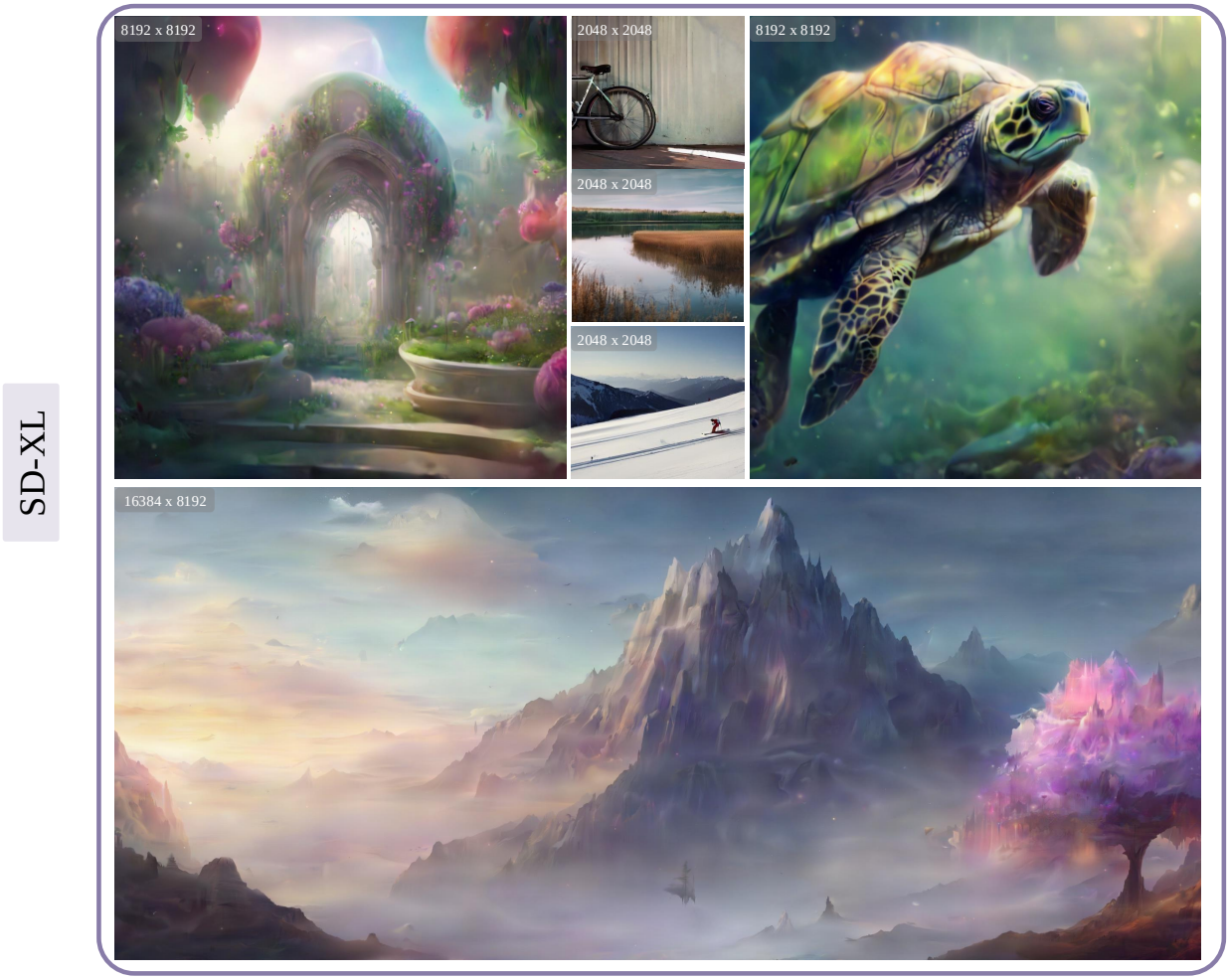

Figure 6. Examples of GSPN at various higher resolutions based upon SDXL. GSPN enable to synthesize images up to a resolution of 16384×8192 using a single A100. Best viewed in PDF with zoom.

References

[1] Sifei Liu, Shalini De Mello, Jinwei Gu, Guangyu Zhong, Ming-Hsuan Yang, and Jan Kautz. Learning affinity via spatial propagation networks. *NeurIPS*, 2017. 3, 4