PARTITION GENERATIVE MODELING: MASKED MODELING WITHOUT MASKS

Anonymous authors
Paper under double-blind review

ABSTRACT

Masked generative models (MGMs) are widely used to capture complex data and enable faster generation than autoregressive models (AR) through parallel decoding. However, MGMs typically operate on fixed-length inputs, which can be inefficient: early in sampling, most tokens are masked and carry little information, leading to wasted computation. In contrast, AR models process only tokens generated previously, making early iterations faster. In this work, we introduce the "Partition Generative Model" (PGM), a novel approach that combines the strengths of AR and MGMs. Rather than masking, PGM partitions tokens into two groups and employs group-wise attention to block information flow between them. Since there is no information flow between partitions, the model can process the previously-generated tokens only during sampling, while retaining the ability to generate tokens in parallel and in any order. On OpenWebText, PGMs offer at least $5\times$ improvements in sampling latency and throughput, while producing samples with superior generative perplexity, compared to Masked Diffusion Language Models. In the ImageNet dataset, PGMs achieve up to $7\times$ better throughput compared to MaskGIT with only a small change in FID. Finally, we show that PGMs are compatible with distillation methods for MGMs, enabling further inference speedups.

1 INTRODUCTION

Masked generative modeling (MGM) excels at sampling from complex data distributions by iteratively denoising masked inputs. In fact, the MGM paradigm has proven successful in various modalities, such as images (Chang et al., 2022), video (Yu et al., 2023; Villegas et al., 2022), and audio spectrograms (Comunità et al., 2024). Furthermore, recent advances leveraging discrete diffusion (Campbell et al., 2022; Zhao et al., 2024; Lou et al., 2024; Sahoo et al., 2024; Shi et al., 2025; Ou et al., 2025) and discrete flow matching (Campbell et al., 2024; Gat et al., 2024) have shown that MGM can also be applied to text generation, challenging the traditional dominance of autoregressive modeling in this domain.

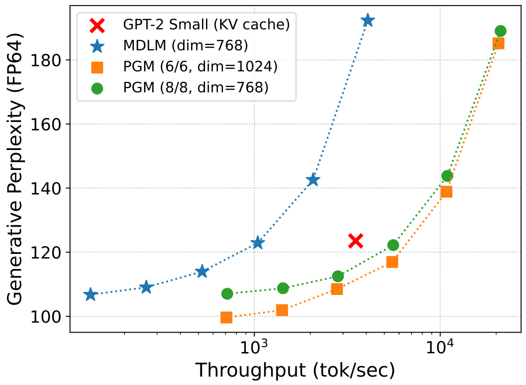


Figure 1: **Throughput:** PGM (ours) achieves better *generative perplexity* with a $\sim 5.3\times$ higher sampling throughput compared to MDLM, at a context length of 1024. The improvements come from our proposed neural network architecture.

Modern MGMs use the transformer architecture (Vaswani et al., 2023) with bidirectional attention to reconstruct masked tokens. This simple approach, which can be viewed as a form of generalized BERT (Devlin et al., 2019), can generate new samples by iteratively denoising a sequence of masked inputs.

Despite their ability to generate high-quality samples, MGMs face two challenges compared to autoregressive models (ARM). First, during sampling, MGMs process many masked tokens that carry minimal information, particularly in the early iterations, when most tokens are masked. In contrast, ARMs only process tokens they have previously generated. Secondly, during training, MGMs learn only at masked input positions, as unmasked tokens are trivial to predict with their unrestricted architectures, whereas AR models can learn at all but the first position due to their causal design.

In this work, we introduce “Partition Generative Models” (PGM), a novel approach that combines strengths from both ARMs and MGMs. In particular, PGMs do not need to process masked tokens during training and inference and can be implemented using Transformers with group-wise attention. The key insight behind PGM is simple: during training, instead of masking tokens, we partition them into two disjoint groups and train the model to predict one group from the other. As shown in Figure 2 (right), this crucial choice allows PGMs to only process unmasked tokens during sampling. In contrast, MGMs always handle full-length sequences. This leads to significant throughput and latency improvements for PGMs. Furthermore, like ARMs, PGMs can learn at every position in a single forward pass during training due to their constrained attention.

Our main contributions can be summarized as follows.

- We introduce “Partition Generative Models” (PGM), a simple alternative to MGM that combine strengths of ARMs and MGMs. We propose an encoder-decoder architecture, based on the Diffusion Transformer of Peebles & Xie (2023) that does not need to process any masked token during training and inference.
- PGM achieves a reduction of 1.95 in validation perplexity in LM1B (Chelba et al., 2014), compared to Masked Diffusion Language Models (MDLM; Sahoo et al. (2024)). In Open-WebText (Gokaslan & Cohen, 2019), PGMs can generate samples of better quality than MDLM with a $5 - 5.5\times$ improvement in sampling throughput and latency, when using the same number of steps as MDLM.
- PGMs can achieve up to $7.5\times$ higher throughput than MaskGIT, with only a marginal increase in FID.
- PGMs are compatible with distillation algorithms designed for MDLM, and preserve their performance on downstream tasks after distillation.

2 Background

2.1 Generative Language Modeling

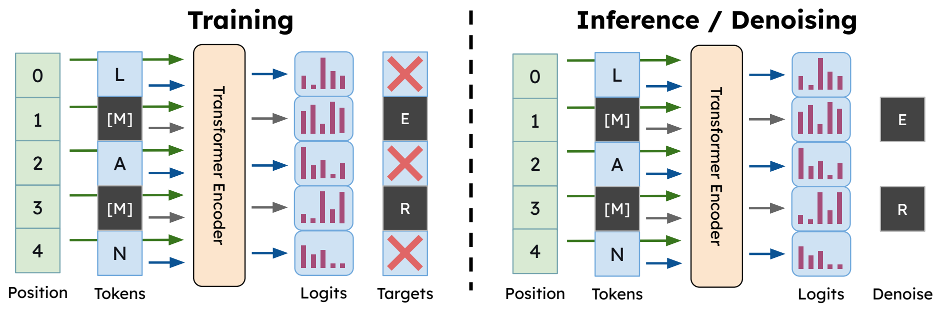
Language modeling addresses the task of generating sequences of discrete tokens (x_i) from a vocabulary $\mathcal{X} = \mathbb{Z}^{<N} = \{0, \ldots, N-1\}$. A language model generates sequences of length L, defined as elements of $\mathcal{X}^L = \left\{\mathbf{x}^{(i)} = (x_0^{(i)}, \ldots, x_{L-1}^{(i)}) : x_j^{(i)} \in \mathcal{X}\right\}_{i=0}^{N^L}$. The training data set $\mathcal{D} := \left\{\mathbf{x}^{(0)}, \ldots, \mathbf{x}^{(K-1)} : \mathbf{x}^{(i)} \in \mathcal{X}^L\right\}$ contains K such sequences. One fundamental objective of language modeling is to generate samples similar to those of the unknown distribution $p_0 : \mathcal{X}^L \to [0,1]$ that induced the training data set $\mathcal{D}$.

2.2 Masked Generative Modeling

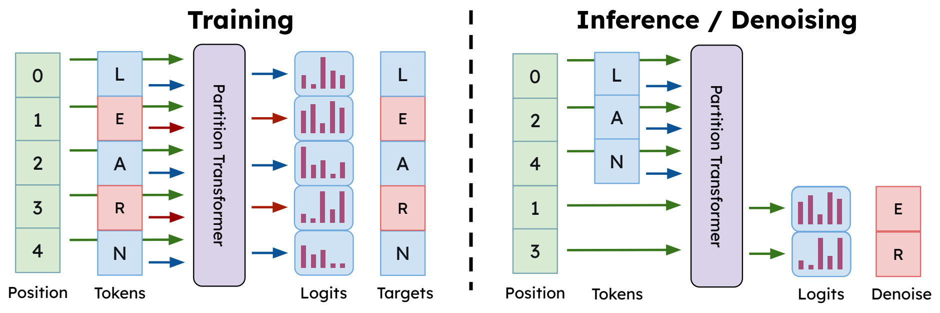
In MGM, the vocabulary $\mathcal{X}$ includes a special `MASK` token absent from the training set $\mathcal{D}$. During training, the `MASK` token is used to replace a fraction of the original tokens in the input sequences $\mathbf{x} \in \mathcal{D}$. Formally, we train a denoiser $\mathbf{x}_\theta : \mathcal{X}^L \to \mathbb{R}^{L \times N}$ with learnable parameters θ. To generate new samples, we initialize the sampling procedure with sequences composed entirely of `MASK` tokens. The model then iteratively replaces a subset of these masked tokens based on the predictions of $\mathbf{x}_\theta$. The training objective of the denoiser $\mathbf{x}_\theta$ can generally be written as follows:

$$\mathcal{L}_{\text{MGM}} := \mathbb{E}_{\mathbf{x}\sim\mathcal{D}, t\sim\mathcal{U}[0,1]}\left[w(t)\text{CE}(\mathbf{x}_\theta(\mathbf{z}_t; t), \mathbf{x})\right], \tag{1}$$

where t determines the proportion of tokens to mask. The corrupted sequence $\mathbf{z}_t$ is generated by independently masking each token in the sequence with time-dependent probability p_t. For simplicity, we could set $p_t = t$. The function $w : [0,1] \to \mathbb{R}_{\geq 0}$ can be used to emphasize certain noise levels more than others. Finally, CE$(\hat{\mathbf{x}}, y)$ denotes the cross-entropy loss between the vector $\hat{\mathbf{x}}$ and integer target y. Oftentimes, the cross-entropy loss is applied exclusively at the masked positions. In such cases, the denoiser model $\mathbf{x}_\theta$ is implemented to assign all probability mass to the input token at positions where the input tokens are *not* masked.



(a) Masked Generative Modeling (MGM)



(b) Partition Generative Modeling (PGM)

Figure 2: **Masked Generative Modeling (MGM) vs. Partition Generative Modeling (PGM). Training**: PGMs receive feedback at every position, while MGMs usually only apply loss to masked tokens. **Inference**: PGMs process only unmasked tokens, working with shorter sequences and predicting logits only for tokens to denoise. MGMs must process full-length sequences and compute logits at all positions. **Important note**: PGMs use a specialized architecture that ensures predictions for position i never depend on the token at position i.

2.3 Masked Diffusion Language Modeling

Masked diffusion language models (MDLM; Sahoo et al. (2024)) are sequence generative models that operate in discrete space. Sahoo et al. (2024) showed that MDLMs approach ARMs in validation perplexity and generation quality. We use MDLM as our primary baseline for the text experiments. For images, we compare against MaskGIT (Chang et al., 2022), which resembles MDLM and whose objective is also of the form in Equation (1).

Discrete Absorbing Diffusion Process MDLMs define a forward process that corrupts the data and a backward process that recovers it. For each token x in the sequence, the forward process linearly interpolates between its one-hot encoding $\mathbf{x}$ and $\boldsymbol{\pi}$, the one-hot vector for the `MASK` token. Formally:

$$q(\mathbf{z}_t|\mathbf{x}) := \text{Cat}(\mathbf{z}_t; \alpha_t \mathbf{x} + (1-\alpha_t)\boldsymbol{\pi}), \tag{2}$$

where $\alpha_t : t \to [0,1]$, the noise schedule, is a strictly decreasing function of t, and represents the noise level at time t. Furthermore, the following boundary conditions apply: $\alpha_0 = 1, \alpha_1 = 0$. The process is termed "absorbing" because the corruption is irreversible. Once a token is masked, it remains so throughout the forward process. The generative distribution $p_\theta(\mathbf{z}_s|\mathbf{z}_t)$ uses the same analytical form as the true posterior $p(\mathbf{z}_s|\mathbf{z}_t, \mathbf{x}) = \frac{p(\mathbf{z}_s|\mathbf{x})p(\mathbf{z}_t|\mathbf{z}_s)}{p(\mathbf{z}_t|\mathbf{x})}$, where $\mathbf{x}$ comes from the data distribution. Since $\mathbf{x}$ is not available during sampling, the output of the denoiser $\mathbf{x}_\theta$ is used in place of $\mathbf{x}$. Formally, $p_\theta(\mathbf{z}_s|\mathbf{z}_t) := q(\mathbf{z}_s|\mathbf{z}_t, \mathbf{x} = \mathbf{x}_\theta(\mathbf{z}_t; t))$. To derive a simple expression for $p_\theta(\mathbf{z}_s|\mathbf{z}_t)$, MDLM enforces that unmasked tokens are carried over during reverse diffusion, which induces the following expression:

$$p_\theta(\mathbf{z}_s|\mathbf{z}_t) = \begin{cases} \text{Cat}(\mathbf{z}_s; \mathbf{z}_t), & \mathbf{z}_t \neq \mathbf{m}, \\ \text{Cat}\left(\mathbf{z}_s; \frac{(1-\alpha_s)\mathbf{m} + (\alpha_s - \alpha_t)\mathbf{x}_\theta(\mathbf{z}_t, t)}{(1-\alpha_t)}\right), & \mathbf{z}_t = \mathbf{m} \end{cases} \tag{3}$$

Training Objective MDLM trains the denoiser $\mathbf{x}_\theta$ using a continuous-time limit of the typical negative evidence lower bound (NELBO) of diffusion models (**?**), which provides a tighter bound to the log-likelihood (Kingma et al., 2023). The denoiser defines a learned posterior distribution $p_\theta(\mathbf{z}_s|\mathbf{z}_t) := q(\mathbf{z}_s|\mathbf{z}_t, \mathbf{x}_\theta(\mathbf{z}_t, t))$, and the NELBO simplifies to a weighted cross-entropy loss between ground-truth samples $\mathbf{x}$ and the predictions of the denoiser $\mathbf{x}_\theta$:

$$\mathcal{L}^{\infty}_{\text{NELBO}} = \mathbb{E}_q \int_{t=0}^{t=1} \frac{\alpha'_t}{1-\alpha_t} \log\langle \mathbf{x}_\theta(\mathbf{z}_t, t), \mathbf{x}\rangle \mathrm{dt}. \tag{4}$$

2.4 Self-Distillation Through Time

Self-Distillation Through Time (SDTT) (Deschenaux & Gulcehre, 2025) speeds up the sampling of MDLMs through a similar approach as Progressive Distillation (Salimans & Ho, 2022). SDTT creates student and teacher copies of a pre-trained MDLM. The student learns to match the teacher's predictions over two steps of size dt. Once converged, the student can serve as the teacher for a new distillation round with step size $2dt$, halving the number of sampling steps.

3 Partition Generative Modeling

3.1 Motivations

"Partition Generative Modeling" (PGM) is similar to MGM but introduces key modifications to the training and sampling procedures. Most notably, PGMs eliminate the need for `MASK` tokens.

Training As seen in Figure 2a (left), in a single forward pass of an MGM, a loss can be computed for the masked positions only. In contrast, autoregressive language (AR) models receive a training signal at every position in a single forward pass. Intuitively, this difference could make MGMs less sample efficient than ARMs. We design PGMs such that we can compute the loss at every position in the sequence in a single forward pass, as shown in Figure 2b (left).

Sampling MGMs typically employ bidirectional architectures trained on fixed-length inputs. Consequently, during sampling, these models *have to* process arrays with the same dimensions as those used during training. Hence, during the initial sampling steps, the neural network processes primarily `MASK` tokens. These numerous `MASK` tokens provide minimal information, only indicating the current noise level. On the other hand, autoregressive models only process previously generated tokens. Additionally, MGMs compute predictions at all masked positions, whereas autoregressive models only generate predictions for the one position to denoise. PGMs only process previously generated tokens and compute predictions solely for tokens that will be denoised (Figure 2b, right). Nonetheless, PGMs maintain the parallel decoding capabilities of MGMs while offering substantial inference speedups.

3.2 Approach

Partitioning Tokens Instead of Masking For a training sequence $\mathbf{x} \in \mathcal{D}$, we partition tokens into two distinct groups labeled 0 and 1, rather than using `MASK` tokens. From the perspective of each group, tokens in the other group will not be visible due to constraints on the neural network architecture, even though no explicit `MASK` token is used. Since each training sequence is partitioned into two groups that predict each other, PGMs effectively create two sub-training examples per sequence. This is conceptually similar to training on two complementary masked sequences within the same batch. We isolate and study the effect of complementary masking from the neural network architecture in Section 5.3.

Training Objective Let $\mathbf{g} \in \{0,1\}^L$ be the binary sequence that denotes the group index of each token in $\mathbf{x}$. We train a denoiser network $\mathbf{x}_\theta$ that takes as input $\mathbf{x}$ and $\mathbf{g}$, and we ensure that only tokens in the same group are involved with each other to avoid information leakage. From the objective, $\mathbf{x}_\theta$ is trained to predict its input, which is only useful because of the constraints on the attention:

$$\mathcal{L}_{\text{PGM}} := \mathbb{E}_{\mathbf{x}\sim\mathcal{D}, t\sim\mathcal{U}[0,1]} \left[w^{\text{PGM}}(\mathbf{g}, t) \text{CE}(\mathbf{x}_\theta(\mathbf{x}; \mathbf{g}; t), \mathbf{x}) \right]. \tag{5}$$

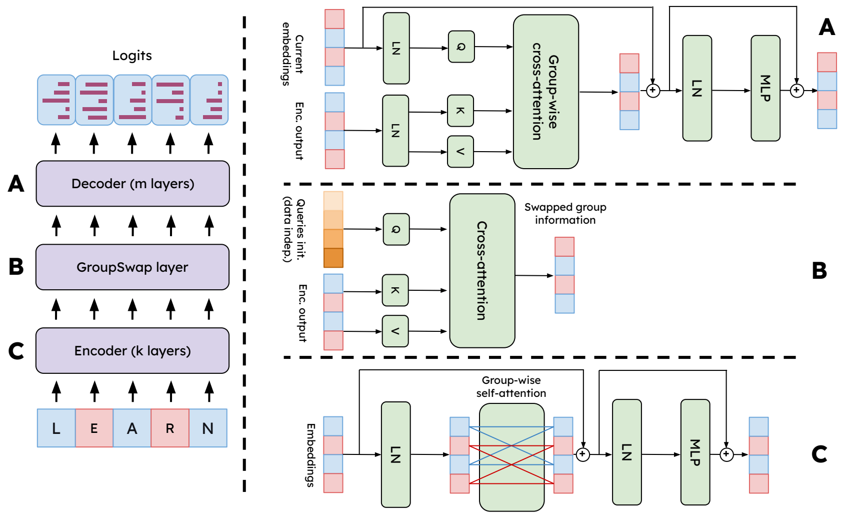


Figure 3: **PGM-compatible transformer architecture.** RoPE (Su et al., 2023) is applied before every attention layer but not shown for clarity. **(A)** Decoder layer with cross-attention to the encoder output and no self-attention between tokens. **(B)** GroupSwap layer that exchanges information between positions in group 0 and group 1, enabling each group to make predictions based on tokens from the other group. **(C)** Encoder layer with group-wise self-attention.

The key distinction from Equation 1 lies in the weighting function w^{PGM}. Let $t \in [0,1]$ be the probability of assigning a token to group 1, and assume for simplicity that exactly a fraction t of tokens belong to this group. From the perspective of group 0, the available information is equivalent to that of an MGM with noise level t, since it cannot access tokens in group 1. Conversely, group 1 experiences a noise level of $1-t$. Therefore, for the PGM loss in Equation 5 to respect the original MGM objective in Equation 1, we must scale the loss for tokens in group 0 according to the weight at the noise level t, and at $1-t$ for tokens in group 1. For example, if MDLM masked 30% of tokens, the PGM groups would contain 30% and 70% of the tokens. Let w represent the weighing function used to train an MGM. Then, the corresponding weight function w^{PGM} to train PGMs is defined as

$$w^{\text{PGM}}(\mathbf{g}, t)_i = \begin{cases} w(t) & \text{if } \mathbf{g}_i = 0 \\ w(1-t) & \text{if } \mathbf{g}_i = 1. \end{cases} \tag{6}$$

We adopt the weighting function of MDLM, namely $w(t) = \frac{\alpha_t'}{1-\alpha_t}$ (Equation 4). A visual comparison of the training processes for MGM and PGM is provided in Figure 2 (left).

Sampling Since the two groups never interact during training, PGMs can process clean tokens only (Figure 2b, right) during inference. Assuming the same posterior distribution $p_\theta(\mathbf{z}_s|\mathbf{z}_t)$ (Equation 3) as MDLM, an MGM denoises each `MASK` token randomly and independently with probability $\frac{\alpha_s-\alpha_t}{1-\alpha_t}$. When implemented as a PGM, it means that one can equivalently select a subset of tokens and denoise exclusively those positions. To simplify the implementation of batched sampling, PGM can denoise a fixed number of tokens at each sampling step, unlike MDLM, which denoises a random number of tokens. The pseudocode is presented in Algorithm 1. PGMs can also sample a random number of tokens at each step, though this requires padding batched sequences. We provide the pseudocode for this approach in Algorithm 2 and compare the perplexity, latency, and throughput of both approaches in Table 6. Empirically, sampling a deterministic number of tokens at every step improves the generative perplexity.

4 The Partition Transformer

Figure 3 illustrates our proposed PGM-compatible Transformer model. The architecture consists of three components: an encoder, the novel GroupSwap layer, and a decoder.

Model ↓	#Params	Val. PPL	Latency (sec) ↓	TP (tok/sec) ↑
LM1B (ctx len. 128)				
MDLM	170M	27.67	3.78	1'081.57
MDLM[†] (Compl. masking)	170M	**25.72**	3.78	1'081.57
PGM 6 / 6	171M	<u>26.80</u>	**2.12**	**1'930.93**
OpenWebText (ctx len. 1024)				
MDLM	170M	23.07	31.41	1'043.22
MDLM[†] (Compl. masking)	170M	22.98	31.41	1'043.22
PGM 8 / 8	203M	<u>22.61</u>	**5.86**	**5'585.57**
PGM 6 / 6 (dim. 1024)	268M	**21.43**	5.93	5'518.09

Table 1: **Validation Perplexity:** On LM1B, PGM with matching number of layers outperform MDLM. *PGM k / m* denotes our model with k encoder and m decoder layers. We highlight the best PGM in gray. The sampling latency and throughput (TP) are measured with a batch size of 32. **On OWT, our PGM outperforms MDLM while delivering at least $5\times$ higher throughput.** See Table 5 for ablations on the architecture. [†] Models trained with a $2\times$ larger batch size (subsection 5.3).

Encoder The encoder consists of a series of partition-wise self-attention transformer blocks. These blocks operate similarly to standard transformer blocks with bidirectional attention, with the key difference that we prevent information from flowing between different groups by masking entries in the attention matrix that correspond to pairs of tokens in different groups.

Decoder The decoder consists of cross-attention layers, where the keys and values are computed based on the output of the encoder. In contrast, the queries are computed using either the output of the GroupSwap layer (first block of the decoder) or the output of the previous decoder block. Importantly, there is no self-attention layer in the decoder, which allows efficient generation, as we can compute predictions solely at the positions that we will decode.

4.1 The GroupSwap Layer

In the encoder, information remains localized: if a token belongs to group 0, its hidden representation only depend on tokens in group 0. For prediction, however, we require the opposite: representations at positions in group 0 must depend exclusively on group 1, and vice versa. To enforce this, we introduce the *GroupSwap* layer (Figure 3B), which exchanges information between groups. The GroupSwap layer is implemented using cross-attention. If a token at position ℓ belongs to group 0, the predictions at position ℓ must rely only on information from group 1. Hence, to prevent information leakage, the queries used in cross-attention cannot depend on tokens in group 0. We describe two ways of initializing these queries below.

Data-Independent Initialization Let $\mathbf{u} \in \mathbb{R}^H$ be a learnable vector. To initialize the queries, we replicate $\mathbf{u}$ across the sequence length, add fixed positional encodings, and apply layer normalization followed by a linear projection. Formally, let $V \in \mathbb{R}^{L\times H}$ denote the query initialization such that $V_{i;\cdot}$ denotes the i-th row of V. Then,

$$V_{i;\cdot} = W\left[\text{LN}\left(u + \text{pos}_{i;\cdot}\right) + b\right], \tag{7}$$

where $W \in \mathbb{R}^{H\times H}$, $b \in \mathbb{R}^H$ are learnable parameters and LN denotes layer normalization (Ba et al., 2016). The positional encoding is computed as

$$\text{pos}_{i,j} = \begin{cases} \cos\left(\frac{i}{10000^{2j/H}}\right) & \text{if } j < {}^{H}\!/_{2} \\ \sin\left(\frac{i}{10000^{2j/H-1}}\right) & \text{otherwise} \end{cases} \tag{8}$$

Data-Dependent Initialization Let $X \in \mathbb{R}^{L\times H}$ be the encoder output. We first perform a group-wise aggregation over the sequence length (e.g., `logsumexp` or `mean`) to obtain vectors

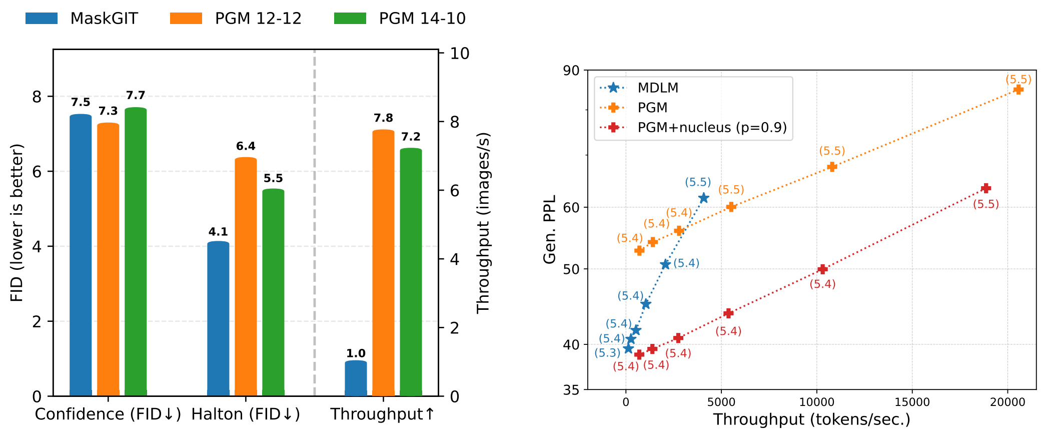


Figure 4: **Left**: FID on Imagenet256. PGM achieves at least 7× higher throughput than MaskGIT with competitive quality. **Right**: After distillation, PGM (6 / 6, dim. 1024) with nucleus sampling remains significantly faster than MDLM. The text next to each point is the unigram entropy, as a proxy for diversity (Dieleman et al., 2022). Importantly, PGM is significantly faster for matched Gen. PPL and entropy.

$Y_0, Y_1 \in \mathbb{R}^H$, representing the aggregated information for groups 0 and 1, respectively. The data-dependent query initialization V' is then computed as

$$V'_{i;\cdot} = V_{i;\cdot} + \begin{cases} Y_1, & \text{if } g_i = 0 \\ Y_0 & \text{otherwise.} \end{cases} \tag{9}$$

5 Experiments

We investigate the performance of PGMs on language and image modeling tasks. Find complete details of the hyperparameters of the experiments in Section B. We report language modeling experiments in Section 5.1, and on OWT in Section 5.2. Finally, we study the effect of complementary masking on the final performance in Section 5.3.

5.1 Language Modeling on LM1B

Experimental setting. We closely follow the settings of Sahoo et al. (2024) and train models with a context length of 128 tokens. The shorter documents are padded to 128 tokens. We train a *Diffusion Transformer* (Peebles & Xie, 2023) with 12 layers without time conditioning and compare it with our Partition Transformer (Section 4). All models are trained with a batch size of 512. We evaluate different variants of the Partition Transformer after 200k training steps, and the version with the best validation perplexity is further trained until reaching 1M training steps.

Results Table 1 shows that after 1M steps, PGM reaches a validation perplexity of 1.95 lower than MDLM. In Table 5 (left), we observe that using as many encoder and decoder layers performs best. Surprisingly, data-independent queries perform similarly to data-dependent queries. Therefore, we use the simpler, data-independent queries in the rest of the experiments. While PGM outperforms MDLM on LM1B, it does not reach its theoretical limit (subsection 5.3) yet.

5.2 Language Modeling on OpenWebText

Experimental Settings We closely follow Sahoo et al. (2024), and train models with a context length of 1024 tokens with sentence packing (Raffel et al., 2023). To ablate the architecture, we train models for 200k steps and compare them based on the validation perplexity. The two models with the best performance are further trained until 1M steps.

Results Table 1 shows that after 1M steps, slightly larger PGM variants outperform MDLM in validation perplexity, while delivering up at least $5\times$ higher throughput. As shown in Table 5 (right), PGMs with the same number of layers as MDLM underperform slighly in terms of validation perplexity. Figure 1, Table 6 and 8 provide more detailed latency and throughput evaluation. We hypothesize that the speedups in inference could make PGMs particularly relevant for the scaling of test-time computation (Madaan et al., 2023; Yao et al., 2023; Snell et al., 2024; Wu et al., 2024; Chen et al., 2024; Brown et al., 2024; Goyal et al., 2024).

Downstream Performance Following Deschenaux & Gulcehre (2024); Nie et al. (2025), we compare PGM and MDLM on downstream tasks from the `lm-eval-harness` suite (Gao et al., 2024). As shown in Table 2, PGM outperforms MDLM on six of eight tasks, while overall performance remains similar. These results suggest that PGM achieves faster inference without sacrificing downstream accuracy. Note that `lm-eval-harness` is originally designed for ARMs, and needs to be adapted for MGMs. We explain how in Section C.4 and compare MDLM and PGMs on additional tasks.

5.3 Disentangling the Effect of the Architecture and Complementary Masking

To disentangle the contributions of PGM, we isolate the effect of complementary masking (subsection 3.2) by training a standard bidirectional transformer encoder with double batch size, using two complementary masked versions of each input sequence. This approach establishes an upper bound on potential performance gains, as it directly measures the impact of having complementary masks during gradient updates. We evaluated standard MDLM against MDLM with complementary masking on LM1B (Chelba et al., 2014) and OpenWebText Gokaslan & Cohen (2019).

Table 1 shows that complementary masking reduces validation perplexity in LM1B but provides smaller gains on OpenWebText. This may explain why PGMs with the same number of parameters outperform MDLMs on LM1B but not on OpenWebText. In both datasets, a gap remains between MDLM with complementary masking and PGM, likely due to the current neural network architecture. Because complementary masking does not improve models on OpenWebText, we increase model size to surpass the validation perplexity of MDLM. Nonetheless, PGMs with more parameters generate higher-quality text and achieve significantly faster inference (Figure 1). In Section C.1, we present preliminary experiments exploring why complementary masking improves performance on LM1B but not on OpenWebText.

5.4 Further Speedups via Distillation

PGM already delivers improvements over MDLM in both throughput and latency, but we can push these gains further using “Self Distillation Through Time” (SDTT; Deschenaux & Gulcehre (2025)). We apply the distillation loss to the token in one of the partitions only, as if they were `MASK`ed, and leave the design of new distillation methods for PGMs to future work. Hence, our setting naturally favors the MDLM baseline.

Figure 4 (right) and Table 7 compare the generative perplexity, entropy, and generation speed of PGMs and MDLM. We find that after 5 rounds of distillation with SDTT, PGMs reach higher Generative Perplexity and unigram Entropy than MDLM (see Table 7 for more precise numbers). After introducing nucleus sampling ($p = 0.9$) (Holtzman et al., 2020), PGMs produce samples with comparable Generative Perplexity and entropy as MDLM. Because nucleus sampling introduces some overhead, PGMs go from being at least $5\times$ faster than MDLM to about $4.6\times$ faster for the same number of steps.

Since generative perplexity alone does not fully capture language modeling performance, we also evaluate the distilled models on downstream tasks. As shown in Table 2, distillation slightly improves the accuracy on some tasks and reduces it on others, but the overall performance remains similar. Notably, PGMs still achieve slightly higher accuracy than MDLM on most tasks.

5.5 PGM on ImageNet

In Figure 4 (left), we compare the Fréchet Inception Distance (FID; (Heusel et al., 2018)) of samples from MaskGIT (Chang et al., 2022) and PGM. All models are trained for 500k steps on ImageNet256

Table 2: **Accuracy on downstream tasks** (Gao et al., 2024). HS: HellaSwag, OQA: OpenBook QA. Arc: Arc-easy. We select the tasks following Nie et al. (2025). We see that distillation slightly changes the downstream tasks performance, but that PGMs continue to outperform MDLM on most tasks.

	LAMBADA	**Arc**	**BoolQ**	**HS**	**OQA**	**PIQA**	**RACE**	**SIQA**
Before Distillation								
MDLM	38.52	37.88	49.42	31.36	**28.60**	58.27	**28.04**	38.84
PGM 8 / 8	**46.98**	**40.40**	**53.49**	<u>33.20</u>	<u>26.60</u>	<u>58.92</u>	26.89	<u>39.97</u>
PGM 6 / 6 (1024)	<u>41.39</u>	<u>39.98</u>	<u>49.82</u>	**34.27**	25.40	**59.19**	<u>27.37</u>	**40.28**
After Distillation (SDTT)								
MDLM	41.34	33.80	48.59	30.75	**28.80**	57.73	<u>27.94</u>	38.79
PGM 8 / 8	**47.22**	**37.42**	**51.50**	<u>31.62</u>	<u>25.80</u>	<u>59.03</u>	**30.62**	**39.61**
PGM 6 / 6 (1024)	<u>44.48</u>	<u>36.70</u>	<u>49.36</u>	**32.55**	25.00	**59.85**	27.37	<u>39.25</u>

(Deng et al., 2009), with the same pre-trained VQGAN tokenizer (Esser et al., 2021) as Besnier et al. (2025). We evaluate the FID of samples generated with MaskGIT's original confidence-based sampler and the Halton sampler of Besnier et al. (2025). We sample with classifier-free guidance weight $\gamma \in \{0, 1, 4\}$, and report the result with the best γ for each model. Using the confidence-based sampler, PGM slightly outperforms MaskGIT, whereas with the Halton sampler it performs slightly worse. In terms of throughput, PGM is at least $7\times$ faster than MaskGIT. Find more experimental details in Section B.3.

6 Related Work

Discrete diffusion Although autoregressive models currently dominate text generation, recent advances in discrete diffusion (Austin et al., 2023; Lou et al., 2024; Shi et al., 2025; Sahoo et al., 2024; von Rütte et al., 2025; Schiff et al., 2025; Haxholli et al., 2025; Sahoo et al., 2025) and discrete flow matching (Campbell et al., 2024; Gat et al., 2024) have demonstrated can MGMs can approach AR models in generation quality. We propose an efficient inference approach that, unlike previous methods, does not require processing `MASK` tokens, yet remains able to generate tokens in any order.

Block Diffusion "Block Diffusion" (Arriola et al., 2025) (BD) proposes a hybrid architecture that interpolates between an autoregressive and a discrete diffusion model. Although BDs can generate tokens in parallel and allow KV caching (Pope et al., 2022), BDs still require generating tokens in a (block-) autoregressive fashion. In contrast, MDLM and PGMs can generate tokens in completely arbitrary orders.

Non-Autoregressive Language Models Any-order and any-subset autoregressive models (Yang et al., 2020; Pannatier et al., 2024; Shih et al., 2022; Guo & Ermon, 2025) learn an autoregressive distribution of tokens given arbitrary token subsets. In contrast, in this work, we accelerate MDLMs (Sahoo et al., 2024), which do not enforce causal attention on the tokens.

7 Conclusion

We introduce "Partition Generative Modeling" (PGM), a novel approach to masked generative modeling that eliminates `MASK` tokens entirely. PGM achieves significant improvements in inference speed on both text and images, with minimal effect on quality. The significant improvements suggest that PGM might be suited for domains that benefit from test-time scaling, such as coding and reasoning. We show that PGMs can be distilled with SDTT (Deschenaux & Gulcehre, 2025) for further acceleration. Future work could explore optimizations to the PGM architecture, investigating distillation techniques specifically designed for PGMs, and extending the approach to multimodal settings. Additionally, exploring how PGMs can be scaled to larger sizes and longer context lengths is an interesting direction. In summary, PGM offers an alternative to masked generative models, with particular advantages for applications where inference speed is critical.

References

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

Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. Structured denoising diffusion models in discrete state-spaces, 2023. URL `https://arxiv.org/abs/2107.03006`.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E. Hinton. Layer normalization, 2016. URL `https://arxiv.org/abs/1607.06450`.

Victor Besnier, Mickael Chen, David Hurych, Eduardo Valle, and Matthieu Cord. Halton scheduler for masked generative image transformer, 2025. URL `https://arxiv.org/abs/2503.17076`.

Bradley Brown, Jordan Juravsky, Ryan Ehrlich, Ronald Clark, Quoc V. Le, Christopher Ré, and Azalia Mirhoseini. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL `https://arxiv.org/abs/2407.21787`.

Andrew Campbell, Joe Benton, Valentin De Bortoli, Tom Rainforth, George Deligiannidis, and Arnaud Doucet. A continuous time framework for discrete denoising models, 2022. URL `https://arxiv.org/abs/2205.14987`.

Andrew Campbell, Jason Yim, Regina Barzilay, Tom Rainforth, and Tommi Jaakkola. Generative flows on discrete state-spaces: Enabling multimodal flows with applications to protein co-design, 2024. URL `https://arxiv.org/abs/2402.04997`.

Huiwen Chang, Han Zhang, Lu Jiang, Ce Liu, and William T. Freeman. Maskgit: Masked generative image transformer, 2022. URL `https://arxiv.org/abs/2202.04200`.

Ciprian Chelba, Tomas Mikolov, Mike Schuster, Qi Ge, Thorsten Brants, Phillipp Koehn, and Tony Robinson. One billion word benchmark for measuring progress in statistical language modeling, 2014. URL `https://arxiv.org/abs/1312.3005`.

Lingjiao Chen, Jared Quincy Davis, Boris Hanin, Peter Bailis, Ion Stoica, Matei Zaharia, and James Zou. Are more llm calls all you need? towards scaling laws of compound inference systems, 2024. URL `https://arxiv.org/abs/2403.02419`.

Marco Comunità, Zhi Zhong, Akira Takahashi, Shiqi Yang, Mengjie Zhao, Koichi Saito, Yukara Ikemiya, Takashi Shibuya, Shusuke Takahashi, and Yuki Mitsufuji. Specmaskgit: Masked generative modeling of audio spectrograms for efficient audio synthesis and beyond, 2024. URL `https://arxiv.org/abs/2406.17672`.

Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pp. 248–255. Ieee, 2009.

Justin Deschenaux and Caglar Gulcehre. Promises, outlooks and challenges of diffusion language modeling, 2024. URL `https://arxiv.org/abs/2406.11473`.

Justin Deschenaux and Caglar Gulcehre. Beyond autoregression: Fast llms via self-distillation through time, 2025. URL `https://arxiv.org/abs/2410.21035`.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL `https://arxiv.org/abs/1810.04805`.

Sander Dieleman, Laurent Sartran, Arman Roshannai, Nikolay Savinov, Yaroslav Ganin, Pierre H. Richemond, Arnaud Doucet, Robin Strudel, Chris Dyer, Conor Durkan, Curtis Hawthorne, Rémi Leblond, Will Grathwohl, and Jonas Adler. Continuous diffusion for categorical data, 2022. URL `https://arxiv.org/abs/2211.15089`.

Patrick Esser, Robin Rombach, and Björn Ommer. Taming transformers for high-resolution image synthesis, 2021. URL https://arxiv.org/abs/2012.09841.

Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. The language model evaluation harness, 07 2024. URL https://zenodo.org/records/12608602.

Itai Gat, Tal Remez, Neta Shaul, Felix Kreuk, Ricky T. Q. Chen, Gabriel Synnaeve, Yossi Adi, and Yaron Lipman. Discrete flow matching, 2024. URL https://arxiv.org/abs/2407.15595.

Aaron Gokaslan and Vanya Cohen. Openwebtext corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

Sachin Goyal, Ziwei Ji, Ankit Singh Rawat, Aditya Krishna Menon, Sanjiv Kumar, and Vaishnavh Nagarajan. Think before you speak: Training language models with pause tokens, 2024. URL https://arxiv.org/abs/2310.02226.

Gabe Guo and Stefano Ermon. Reviving any-subset autoregressive models with principled parallel sampling and speculative decoding, 2025. URL https://arxiv.org/abs/2504.20456.

Etrit Haxholli, Yeti Z. Gurbuz, Oğul Can, and Eli Waxman. Efficient perplexity bound and ratio matching in discrete diffusion language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=Mri9WIfxSm.

Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2018. URL https://arxiv.org/abs/1706.08500.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration, 2020. URL https://arxiv.org/abs/1904.09751.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017. URL https://arxiv.org/abs/1412.6980.

Diederik P. Kingma, Tim Salimans, Ben Poole, and Jonathan Ho. Variational diffusion models, 2023. URL https://arxiv.org/abs/2107.00630.

Aaron Lou, Chenlin Meng, and Stefano Ermon. Discrete diffusion modeling by estimating the ratios of the data distribution, 2024. URL https://arxiv.org/abs/2310.16834.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Shashank Gupta, Bodhisattwa Prasad Majumder, Katherine Hermann, Sean Welleck, Amir Yazdanbakhsh, and Peter Clark. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Shen Nie, Fengqi Zhu, Chao Du, Tianyu Pang, Qian Liu, Guangtao Zeng, Min Lin, and Chongxuan Li. Scaling up masked diffusion models on text, 2025. URL https://arxiv.org/abs/2410.18514.

Jingyang Ou, Shen Nie, Kaiwen Xue, Fengqi Zhu, Jiacheng Sun, Zhenguo Li, and Chongxuan Li. Your absorbing discrete diffusion secretly models the conditional distributions of clean data, 2025. URL https://arxiv.org/abs/2406.03736.

Arnaud Pannatier, Evann Courdier, and François Fleuret. Sigma-gpts: A new approach to autoregressive models, 2024. URL https://arxiv.org/abs/2404.09562.

William Peebles and Saining Xie. Scalable diffusion models with transformers, 2023. URL https://arxiv.org/abs/2212.09748.

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

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023. URL `https://arxiv.org/abs/1910.10683`.

Subham Sekhar Sahoo, Marianne Arriola, Yair Schiff, Aaron Gokaslan, Edgar Marroquin, Justin T Chiu, Alexander Rush, and Volodymyr Kuleshov. Simple and effective masked diffusion language models, 2024. URL `https://arxiv.org/abs/2406.07524`.

Subham Sekhar Sahoo, Justin Deschenaux, Aaron Gokaslan, Guanghan Wang, Justin Chiu, and Volodymyr Kuleshov. The diffusion duality, 2025. URL `https://arxiv.org/abs/2506.10892`.

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

Tim Salimans, Ian Goodfellow, Wojciech Zaremba, Vicki Cheung, Alec Radford, and Xi Chen. Improved techniques for training gans, 2016. URL `https://arxiv.org/abs/1606.03498`.

Yair Schiff, Subham Sekhar Sahoo, Hao Phung, Guanghan Wang, Sam Boshar, Hugo Dalla-torre, Bernardo P. de Almeida, Alexander Rush, Thomas Pierrot, and Volodymyr Kuleshov. Simple guidance mechanisms for discrete diffusion models, 2025. URL `https://arxiv.org/abs/2412.10193`.

Jiaxin Shi, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis K. Titsias. Simplified and generalized masked diffusion for discrete data, 2025. URL `https://arxiv.org/abs/2406.04329`.

Andy Shih, Dorsa Sadigh, and Stefano Ermon. Training and inference on any-order autoregressive models the right way, 2022. URL `https://arxiv.org/abs/2205.13554`.

Charlie Snell, Jaehoon Lee, Kelvin Xu, and Aviral Kumar. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL `https://arxiv.org/abs/2408.03314`.

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. URL `https://arxiv.org/abs/2104.09864`.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023. URL `https://arxiv.org/abs/1706.03762`.

Ruben Villegas, Mohammad Babaeizadeh, Pieter-Jan Kindermans, Hernan Moraldo, Han Zhang, Mohammad Taghi Saffar, Santiago Castro, Julius Kunze, and Dumitru Erhan. Phenaki: Variable length video generation from open domain textual description, 2022. URL `https://arxiv.org/abs/2210.02399`.

Dimitri von Rütte, Janis Fluri, Yuhui Ding, Antonio Orvieto, Bernhard Schölkopf, and Thomas Hofmann. Generalized interpolating discrete diffusion, 2025. URL `https://arxiv.org/abs/2503.04482`.

Yangzhen Wu, Zhiqing Sun, Shanda Li, Sean Welleck, and Yiming Yang. An empirical analysis of compute-optimal inference for problem-solving with language models, 2024. URL `https://arxiv.org/abs/2408.00724`.

Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V. Le. Xlnet: Generalized autoregressive pretraining for language understanding, 2020. URL `https://arxiv.org/abs/1906.08237`.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik Narasimhan. Tree of thoughts: Deliberate problem solving with large language models, 2023. URL `https://arxiv.org/abs/2305.10601`.

Lijun Yu, Yong Cheng, Kihyuk Sohn, José Lezama, Han Zhang, Huiwen Chang, Alexander G. Hauptmann, Ming-Hsuan Yang, Yuan Hao, Irfan Essa, and Lu Jiang. Magvit: Masked generative video transformer, 2023. URL `https://arxiv.org/abs/2212.05199`.

Lingxiao Zhao, Xueying Ding, Lijun Yu, and Leman Akoglu. Unified discrete diffusion for categorical data, 2024. URL `https://arxiv.org/abs/2402.03701`.

Kaiwen Zheng, Yongxin Chen, Hanzi Mao, Ming-Yu Liu, Jun Zhu, and Qinsheng Zhang. Masked diffusion models are secretly time-agnostic masked models and exploit inaccurate categorical sampling, 2025. URL `https://arxiv.org/abs/2409.02908`.

A Limitations

To match the validation perplexity of the MDLM baseline at a context length of 1024, our models require an increased parameter count. We attribute this to the GroupSwap layer, and future work will explore more efficient mechanisms for information exchange between groups in PGMs. While PGMs offer faster inference, their training is slightly more computationally expensive (Section D), as use `torch`'s default attention implementation ("sdpa") for simplicity. By reordering tokens according to their group assignment, the self-attention matrix becomes block-diagonal. Future work will explore efficient kernel implementations that exploit this block-diagonal sparsity. Partition Generative Modeling is a general framework, and its application to multimodal settings remains an open direction for future research.

B Experimental Details

We trained all models from scratch rather than using the pre-trained models released by the MDLM authors. Our models achieve comparable performance to the original work. On LM1B, we obtain a validation perplexity of 27.67 after 1M steps (compared to MDLM's reported 27.04), while on OWT, we reach 23.07 (versus MDLM's 23.21).

Minor differences can be expected since estimating the perplexity of diffusion language models involves a Monte-Carlo approximation of the NELBO (Equation 4) with finitely many samples. Although we used libraries (e.g PyTorch) with the same version as MDLM, differences in compute environments and underlying software stacks may also contribute to these variations. Since the performance gap is small, we are confident that we used the code of MDLM correctly.

B.1 LM1B

For the LM1B dataset, we employed the `bert-base-uncased` tokenizer with a context length of 128 tokens, padding shorter sequences. Our architecture consisted of a Diffusion Transformer (DiT) with 12 transformer blocks, 12 attention heads, a hidden dimension of 768, and a dropout rate of 0.1. We optimized the model using Adam (Kingma & Ba, 2017) (learning rate 3e-4, betas of (0.9, 0.999), epsilon 1e-8) without weight decay. We based our implementation on the official MDLM codebase. We trained with a global batch size of 512 across 8 GPUs (2 nodes with 4 GPUs), gradient clipping at 1.0, and a constant learning rate with 2,500 steps of linear warmup. We trained for 1 million steps with an EMA rate of 0.9999. Besides the neural network hyperparameters, the other parameters were unchanged when training the PGM.

B.2 OWT

For the OpenWebText (OWT) dataset, we used the GPT-2 tokenizer with a context length of 1024 tokens. Our architecture consisted of a Diffusion Transformer (DiT) with 12 transformer blocks, 12 attention heads, a hidden dimension of 768, and a dropout rate of 0.1. We optimized the model using Adam (Kingma & Ba, 2017) with a learning rate of 3e-4, betas of (0.9, 0.999), and epsilon of 1e-8, without weight decay. We trained with a global batch size of 512 across 16 GPUs (4 nodes with 4 GPUs). We applied gradient clipping at 1.0 and used a constant learning rate schedule with 2,500 steps of linear warmup. The model was trained for 1 million steps with an EMA rate of 0.9999.

Table 3: Latency and throughput for a single training step of the MDLMs and PGMs, computed on a single A100-SXM4-80GB GPU. On LM1B, PGM introduces a negligible overhead over MDLM. On OWT, our PGM with 6 encoder and decoder layers and an embedding dimension of 1024 achieves around 75% of the training throughput of MDLM. Recall that at inference, the same PGM is around $5\times$ faster than MDLM.

Model	**Forward Pass**		**Forward + Backward**	
	Latency (ms)	**Seq/sec**	**Latency (ms)**	**Seq/Sec**
LM1B (context length 128, batch size 64, trained on 8 GPUs)				
MDLM	0.03 ± 0.00	$1'978.87 \pm 44.21$	0.08 ± 0.00	714.80 ± 15.47
PGM 6 / 6	0.03 ± 0.00	$1'966.60 \pm 102.14$	0.08 ± 0.00	794.42 ± 18.81
OpenWebText (context length 1024, batch size 32, trained on 16 GPUs)				
MDLM	0.13 ± 0.00	233.28 ± 2.58	0.39 ± 0.00	80.86 ± 0.15
PGM 8 / 8	0.17 ± 0.00	188.07 ± 0.75	0.47 ± 0.00	68.04 ± 0.08
PGM 6 / 6 (dim. 1024)	0.18 ± 0.00	176.47 ± 0.65	0.50 ± 0.00	62.85 ± 0.19

Besides the neural network hyperparameters, the other parameters were unchanged when training the PGM.

B.3 ImageNet

For the ImageNet experiments, we used a pre-trained VQGAN tokenizer (Esser et al., 2021; Besnier et al., 2025), following exactly the setup of HaltonMaskGIT (Besnier et al., 2025). The images are tokenized into sequences of 1024 tokens. This allowed for a direct comparison between PGM and MaskGIT, both trained in the codebase of Besnier et al. (2025) and the FID is evaluated using the Halton sampler and the original sampler.

All models use 24 transformer blocks. For PGM, we add a GroupSwap layer to enable information exchange between partition groups. We use the same hyperparameters as HaltonMaskGIT for all models, except we reduce the training duration to 500k steps (from 2M) due to computational constraints. All models are trained to be class-conditional, which enables the use of classifier-free guidance to significantly improve performance.

As shown in Table 9, PGMs slightly outperform MaskGIT in FID when using the original confidence-based sampler. With the Halton sequence-based sampler, PGMs achieve a marginally higher FID, as reported in Table 10. In terms of efficiency, PGMs deliver up to $7.5\times$ higher throughput than MaskGIT, as shown in Table 11.

B.4 Impact of Numerical Precision on Sampling

Zheng et al. (2025) identified that Masked Diffusion Models often achieve lower generative perplexity results because of underflow in the logits when sampling using low precision. The resulting decrease in token diversity can make evaluations based solely on generative perplexity misleading. Hence, we always cast the logits to FP64 before sampling.

B.5 Sample-Based Evaluation

Generative Perplexity We use the generative perplexity to evaluate the quality of samples, following prior work (Lou et al., 2024; Sahoo et al., 2024; Deschenaux & Gulcehre, 2025). The generative perplexity measures how well a reference model (in our case, GPT-2 Large) can predict the next token in generated sequences. Specifically, we generate $1'024$ samples from each model being evaluated. For each generated sample, we compute the generative perplexity using GPT-2 Large as follows:

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log p_{\text{GPT-2 Large}}(x_i|x_{<i})\right), \tag{10}$$

where L is the length of the sequence, x_i is the i-th token, and $p_{\text{GPT-2 Large}}(x_i|x_{<i})$ is the probability assigned by GPT-2 Large to token x_i given the preceding tokens $x_{<i}$.

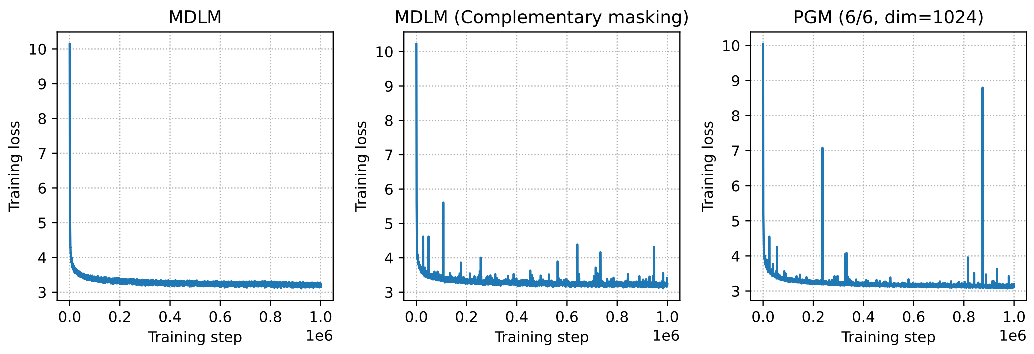


Figure 5: Training loss of MDLM, MDLM with Complementary Masking (Section 5.3) and PGM. Complementary masking seems to introduce spikes in the loss, even though it did not cause the models to diverge.

Unigram Entropy Unfortunately, a low generative perplexity can be achieved by generating repetitive text. To catch such cases, we compute the average unigram entropy of the generated samples:

$$\text{Unigram Entropy} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{v\in\mathcal{X}}\frac{c(v,\mathbf{x}^{(i)})}{L}\log\frac{c(v,\mathbf{x}^{(i)})}{L}, \tag{11}$$

where $\mathcal{X}$ is the vocabulary, v is a token of the vocabulary, and $c(v,\mathbf{x})$ is the empirical appearance count of the token v in the sequence $\mathbf{x}$. Low unigram entropy helps us to catch degenerate generation, as shown in prior work (Dieleman et al., 2022).

Fréchet Inception Distance and Inception Score On image generation tasks, we evaluate the quality of samples using the Fréchet Inception Distance (FID) (Heusel et al., 2018) and Inception Score (IS) (Salimans et al., 2016). Both metrics are computed using $50'000$ images, following the standard practice.

C Additional Results

C.1 Impact of Context Length on the Effectiveness of Complementary Masking

There are three key differences between our experiments on LM1B and OWT. First, we used different tokenizers: `bert-base-uncased` for LM1B and `GPT2`'s tokenizer for OWT, following the setup of MDLM (Sahoo et al., 2024). Second, the context lengths differ significantly: 128 tokens for LM1B versus 1024 for OWT. Third, we train on different datasets that might have different characteristics.

We observed that complementary masking helps when training on OWT using a shorter context length of 128 tokens with the GPT-2 tokenizer. Indeed, after the 200k training step, the MDLM with complementary masking achieved a validation PPL of 37.92, outperforming the standard MDLM, which reached 39.90. This suggests that PGMs may not need extra parameters when the sequence length is short. Exploring the use of PGMs in domains where the sequence length is short, such as modeling chemical sequences, is a promising direction for future work.

C.2 MDLM+SDTT vs PGM+SDTT

The precision of logits during sampling can have a significant effect on sample quality, as noted in Section B.4. Hence, we cast all logits to FP64 prior to sampling, unlike the original MDLM and SDTT implementations.

Model	LAMBADA	ARC-e	ARC-c	HSwag	MathQA	PIQA	WinoG
MDLM	38.52	34.26	**24.66**	31.54	20.70	57.89	51.93
PGM 8 / 8	**46.98**	37.37	24.06	33.10	21.24	59.09	51.30
PGM 6 / 6 (1024)	41.39	**38.80**	22.95	**33.92**	**21.71**	**61.43**	**54.30**

Table 4: Accuracy on downstream tasks. We evaluate MDLM and PGM on LAMBADA, ARC Easy and Challenge, HellaSwag, MathQA, PIQA, and WinoGrande. Both models show comparable performance across tasks. PGM outperforms MDLM on all but one benchmark, where the difference between MDLM and PGM 8 / 8 is small.

Using higher precision also affects distillation, which compresses two sampling steps into one. As shown in Table 7, models distilled with float32 achieve lower generative perplexity than those trained with mixed precision (bfloat16). We therefore report float32 results in the main body.

C.3 Training Stability

Complementary masking introduces occasional spikes in the training loss in both MDLMs and PGMs, as shown in Figure 5. This phenomenon should be kept in mind when scaling PGMs to larger sizes. Despite these spikes, all runs converged on the first attempt. We observed different precision requirements between models. For loss computations, MDLMs performed best with BF16 precision, while PGMs achieved better results with FP32 precision. Both models use mixed precision within the neural network; the precision difference only affects computations performed outside the model, such as the loss calculation.

C.4 Additional Downstream Tasks

Table 4 shows more downstream evaluation results following SDTT (Deschenaux & Gulcehre, 2025), where PGM outperforms MDLM on all but one benchmark, where the difference is small. We compare the models using the `lm-eval-harness` library (Gao et al., 2024). The `lm-eval-harness` library was designed for autoregressive language models and needs to be adapted for MDLM. For multiple choice questions, `lm-eval-harness` relies on a function that computes the log-likelihood of each answer $\mathbf{y}_i$ given a prefix $\mathbf{x}$. The model computes $p(\mathbf{y}_i|\mathbf{x})$ for each possible answer i and choosing the one with the highest log-likelihood.

Table 4 reports additional downstream results as in Deschenaux & Gulcehre (2025), where PGM outperforms MDLM on all but one benchmark, with only a small gap on the latter. We evaluate models with the `lm-eval-harness` library (Gao et al., 2024), originally designed for autoregressive LMs and adapted here for MDLM. For multiple-choice tasks, `lm-eval-harness` computes the log-likelihood of each candidate answer $\mathbf{y}_i$ given a prefix $\mathbf{x}$, i.e., $p(\mathbf{y}_i|\mathbf{x})$, and selects the answer with the highest score.

While `lm-eval-harness` uses the log-likelihood of the continuation, the NELBO objective (Equation 4) bounds the log-likelihood of the *complete* sequence $(\mathbf{x}, \mathbf{y}_i)$. However, we only need to know which continuation achieves the highest log-likelihood, not to compute the exact log-likelihood. Using Bayes' theorem, we note that

$$\log p(\mathbf{y}_i|\mathbf{x}) = \log p(\mathbf{x}, \mathbf{y}_i) - \log p(\mathbf{x}) \propto \log p(\mathbf{x}, \mathbf{y}_i), \tag{12}$$

since $\log p(\mathbf{x})$ is constant with respect to $\mathbf{y}_i$. Therefore, we can simply evaluate the variational bound on $\log p(\mathbf{x}, \mathbf{y}_i)$ to select the most likely continuation y_i.

C.5 Performance on longer context length

Due to the high computational cost, we were unable to train models with context lengths greater than 1024. Nevertheless, we report the latency and throughput of both MDLM and PGM at a context length of 4096. As shown in Table 8, PGM remains substantially faster than MDLM in this setting.

D COMPUTATIONAL COSTS

This section presents the computational costs associated with the models reported in this paper. We exclude costs associated with exploratory experiments that yielded inferior results and were not included in this manuscript.

D.1 TRAINING COSTS

Training PGMs is currently slower than training MGMs since we use `torch.sdpa` with dense tensor masks. Future work should explore efficient kernels to address this limitation. We measure the latency and throughput using a single NVIDIA A100-SXM4-80GB GPU, with results reported in Table 3. We compute the mean and standard deviation over 100 batches after 2 warmup batches.

The total training duration approximately equals the per-step latency multiplied by the number of steps. Experiments with complementary masking required twice the computational resources due to larger batch sizes and gradient accumulation. Training times for 1M steps varied by dataset: approximately 22 hours for LM1B, 4.5 days for OWT, and 3.8 days for ImageNet.

D.2 INFERENCE COSTS

We evaluate the inference efficiency of PGMs compared to MDLMs and GPT-2 with KV caching. As shown in Figure 1, PGMs achieve around $5 - 5.5\times$ improvements in throughput over MDLM while reaching superior generative perplexity. For inference measurements, we use a single NVIDIA A100-SXM4-80GB GPU. The efficiency gain stems from the ability of PGMs to process only unmasked tokens during inference, as illustrated in Figure 2. Table 6 compares MDLM and PGMs on the generative perplexity, unigram entropy, latency, and throughput. We compute the mean and standard deviation of the latency and throughput over 20 batches after two warmup batches.

D.3 LICENSING

Our code and model artifacts will be released under the MIT license. The OWT dataset (Gokaslan & Cohen, 2019) is available under the Apache License 2.0. We were unable to identify a specific license for the LM1B dataset (Chelba et al., 2014). The images in ImageNet remain the property of their respective copyright holders.

Algorithm 1 Simplified Sampling for PGMs

```
1: **Input:** Batch size BS, number of steps K, model length L, special BOS index
2: **Output:** Generated samples x
3: x $\leftarrow$ empty_tensor(BS, 1)                                   $\triangleright$ *Initialize*
4: x[:, 0] $\leftarrow$ BOS                                              $\triangleright$ *Set BOS as first token*
5: k $\leftarrow$ L/K                                 $\triangleright$ *Number of tokens to denoise at each step*
6: decoded_positions $\leftarrow$ zeros(BS, 1)   $\triangleright$ *Keep track of already-decoded and positions to decode*
7: positions_to_decode $\leftarrow$ 1+ rand_row_perm(BS, L-1) $\triangleright$ *Each rows is a permutation of* $\{1, ..., L\}$
8: **for** _ in range(K) **do**
9:     pos_to_decode $\leftarrow$ positions_to_decode[:, :k]          $\triangleright$ *Random positions to be predicted*
10:    new_values $\leftarrow$ pgm_predict(x, decoded_positions, pos_to_decode)
11:    $x$ $\leftarrow$ concat([x, new_values], dim=1)   $\triangleright$ *Add new values to the sequence length dimension*
12:    decoded_positions $\leftarrow$ concat([decoded_positions, pos_to_decode], dim=1)
13:    positions_to_decode $\leftarrow$ positions_to_decode[:, k:]     $\triangleright$ *Remove the k decoded positions*
14: **end for**
15: out $\leftarrow$ reoder(x, decoded_positions)                    $\triangleright$ *Sort based on positions*
16: **return** out
```

Algorithm 2 MDLM-equivalent sampling for PGMs.

```
1: Input: Batch size BS, number of steps K, model length L, special BOS index
2: Output: Generated samples x
3: x ← empty_tensor(BS, 1)                                          ▷ Initialize
4: x[:, 0] ← BOS                                                    ▷ Set BOS as first token
5: k ← L/K                                                          ▷ Number of tokens to denoise at each step
6: clean_positions ← zeros(BS, 1)                                   ▷ Keep track of clean and noisy positions
7: concrete_lengths ← ones(BS, 1)  ▷ Keep track of the actual length of each sequence (some are
   padded).
8: noisy_positions ← 1+ rand_row_perm(BS, L-1)
9: for _ in range(K) do
10:    n_denoise_per_seq, noisy_pos_input ← sample_noisy(noisy_positions, k)   ▷ Algorithm 3
11:    new_values ← pgm_predict(x, clean_positions, noisy_pos_input)
12:    x, clean_positions, noisy_positions, concrete_lengths ← extract_predictions(
13:        x,                                                       ▷ Algorithm 4
14:        clean_positions,
15:        noisy_positions,
16:        noisy_pos_input,
17:        concrete_lengths,
18:        n_denoise_per_seq,
19:        new_values)
20: end for
21: out ← reoder(x, clean_positions)                                ▷ Sort based on clean_positions
22: return out
```

Algorithm 3 Sample the number of tokens to denoise from a binomial distribution and pad the input.

```
1: Input: Noisy positions tensor, probability of denoising prob_denoise, model length L, concrete
   lengths tensor
2: Output: Noisy positions to denoise
3: n_denoise_per_seq ← binomial(BS, L, prob_denoise)        ▷ Sample from binomial distribution
4: n_denoise_per_seq ← min(n_denoise_per_seq, L - concrete_lengths)   ▷ Don't denoise more
   than available
5: denoise_seq_len ← max(n_denoise_per_seq, 0)           ▷ Maximum number of tokens to denoise
6: if denoise_seq_len = 0 then
7:     return empty_tensor()                                ▷ Nothing to denoise
8: end if
9: noisy_pos_input ← noisy_positions[:, :denoise_seq_len]   ▷ Some predictions won't be used
10: return n_denoise_per_seq, noisy_pos_input
```

Algorithm 4 Extract the correct number of predictions per sequence

```
1: Input: x, concrete_lengths, n_denoise_per_seq, denoised_token_values, clean_positions,
   noisy_positions, noisy_pos_input
2: Output: Updated x, clean_positions, noisy_positions, concrete_lengths
3: new_concrete_lengths ← concrete_lengths + n_denoise_per_seq   ▷ Update sequence lengths
4: n_tok_to_add ← max(new_concrete_lengths) - shape(x, 1)       ▷ Calculate padding needed
5: if n_tok_to_add > 0 then
6:     pad ← zeros(BS, n_tok_to_add)                             ▷ Create padding tensor
7:     x ← concat(x, pad, dim=1)                                 ▷ Pad the sequences
8:     clean_positions ← concat(clean_positions, pad, dim=1)     ▷ Pad the positions
9: end if
10: for i in range(BS) do
11:     if n_denoise_per_seq[i] = 0 then
12:         continue                                             ▷ Skip if no tokens to denoise
13:     end if
14:     x[i, concrete_lengths[i]:new_concrete_lengths[i]] ←
15:         denoised_token_values[i, :n_denoise_per_seq[i]]
16:     clean_positions[i, concrete_lengths[i]:new_concrete_lengths[i]] ←
17:         noisy_pos_input[i, :n_denoise_per_seq[i]]
18:     noisy_positions[i, :shape(noisy_positions, 1) - n_denoise_per_seq[i]] ←
19:         noisy_positions[i, n_denoise_per_seq[i]:]
20: end for
21: return x, clean_positions, noisy_positions, new_concrete_lengths
```

Model (LM1B)	**Val. PPL ↓**
200k steps	
MDLM	34.29
MDLM (Compl. masking)	**30.87**
PGM 8 / 4	32.83
PGM 10 / 2	33.55
PGM 4 / 8	32.84
PGM 6 / 6	<u>32.69</u>
PGM 6 / 6 (lsm)	32.70
PGM 6 / 6 (mean)	33.89
1M steps	
MDLM	27.67
MDLM (Compl. masking)	**25.72**
PGM 6 / 6	<u>26.80</u>

Model (OWT)	**Val. PPL ↓**
200k steps	
MDLM	25.35
MDLM (Compl. masking)	25.32
PGM 6 / 6	26.96
PGM 8 / 8	<u>25.10</u>
PGM 10 / 6	25.19
PGM 6 / 6 (dim. 1024)	**23.75**
1M steps	
MDLM	23.07
MDLM (Compl. masking)	22.98
PGM 8 / 8	22.61
PGM 6 / 6 (dim. 1024)	**21.43**

Table 5: Perplexity evaluations. Validation perplexity of the Masked Diffusion Language Model (MDLM) and PGMs (ours) on LM1B and OpenWebText (OWT). The row *MDLM (Compl. masking)* denotes an MDLM trained with the complementary masking strategy discussed in Section 5.3. The row *PGM k / m* denotes a PGM with k encoder and m decoder layers, and we highlighted the best PGM results in gray. *lsm* and *mean* denote the *logsumexp* and *mean* queries initializations (Section 4). **Takeaway:** using the same number of layers in the encoder and decoder, and data-independent queries performed best. On LM1B, our PGM reaches 1.95 lower perplexity than MDLM after 1M steps. On OWT, we grow the embedding dimension or the number of layers to outperform OWT.

Table 6: Sample quality and efficiency on OpenWebText with different numbers of sampling steps. We generate sequences of 1024 tokens with a batch size of 32 to measure the latency and throughput. PGM 6 / 6 with a hidden dimension of 1024 and uniform sampling achieves at least a $5\times$ latency and throughput improvement over MDLM, with better generative perplexity and matching entropy.

Model	**Gen. PPL** $\downarrow$	**Entropy** $\uparrow$	**Latency** $\downarrow$ **(ms)**	**Throughput** $\uparrow$ **(tok/s)**
MDLM				
32 steps	192.31	5.73	8.037 ± 0.01	$4'077.08 \pm 3.06$
64 steps	142.58	5.69	15.82 ± 0.01	$2'070.67 \pm 0.69$
128 steps	122.89	5.67	31.41 ± 0.01	$1'043.22 \pm 0.16$
256 steps	113.96	5.66	62.54 ± 0.01	523.90 ± 0.06
512 steps	109.05	5.64	124.94 ± 0.16	262.26 ± 0.33
1024 steps	106.75	5.64	249.31 ± 0.11	131.42 ± 0.05
PGM 8 / 8 (uniform sampling)				
32 steps	189.02	5.73	1.55 ± 0.01	$21'120.99 \pm 83.59$
64 steps	143.79	5.69	3.00 ± 0.01	$10'914.91 \pm 41.69$
128 steps	122.21	5.66	5.86 ± 0.02	$5'585.57 \pm 24.49$
256 steps	112.48	5.65	11.64 ± 0.03	$2'814.99 \pm 9.33$
512 steps	108.76	5.64	22.98 ± 0.02	$1'425.89 \pm 1.61$
1024 steps	107.03	5.63	45.84 ± 0.03	714.71 ± 0.50
PGM 8 / 8 (non uniform sampling)				
32 steps	194.09	5.73	2.07 ± 0.02	$15'764.09 \pm 192.12$
64 steps	143.60	5.69	3.90 ± 0.07	$8'405.14 \pm 158.01$
128 steps	124.38	5.67	7.41 ± 0.08	$4'419.77 \pm 53.27$
256 steps	116.85	5.66	14.73 ± 0.19	$2'223.6372 \pm 28.47$
512 steps	111.11	5.64	28.15 ± 0.32	$1'163.79 \pm 13.25$
1024 steps	108.24	5.63	54.62 ± 0.66	599.97 ± 7.27
PGM 6 / 6 (dim. 1024, uniform sampling)				
32 steps	185.16	5.73	1.59 ± 0.01	$20'569.99 \pm 95.63$
64 steps	138.87	5.70	3.03 ± 0.01	$10'805.31 \pm 14.11$
128 steps	116.95	5.67	5.93 ± 0.01	$5'518.09 \pm 13.46$
256 steps	108.51	5.65	11.77 ± 0.01	$2'782.78 \pm 3.46$
512 steps	101.94	5.63	23.25 ± 0.01	$1'408.88 \pm 1.05$
1024 steps	99.64	5.62	46.31 ± 0.02	707.52 ± 0.34
PGM 6 / 6 (dim. 1024, non-uniform sampling)				
32 steps	191.30	5.74	2.12 ± 0.07	$15'415.56 \pm 467.20$
64 steps	138.67	5.69	3.940 ± 0.06	$8'318.72 \pm 135.47$
128 steps	118.17	5.67	7.60 ± 0.09	$4'311.80 \pm 54.92$
256 steps	108.93	5.65	14.84 ± 0.20	$2'207.71 \pm 29.71$
512 steps	105.41	5.64	28.56 ± 0.33	$1'147.17 \pm 13.47$
1024 steps	102.93	5.62	55.50 ± 0.36	590.37 ± 3.85

Table 7: Generative perpleixty of MDLM and PGM after distillation with varying precision.

Model	**Gen. PPL** $\downarrow$	**Entropy** $\uparrow$	**Latency** $\downarrow$ **(ms)**	**Throughput** $\uparrow$ **(tok/s)**
MDLM+SDTT (loss in BF16)				
32 steps	66.26	5.49	8.037 ± 0.01	$4'077.08 \pm 3.06$
64 steps	53.98	5.46	15.82 ± 0.01	$2'070.67 \pm 0.69$
128 steps	48.02	5.44	31.41 ± 0.01	$1'043.22 \pm 0.16$
256 steps	45.86	5.42	62.54 ± 0.01	523.90 ± 0.06
512 steps	44.21	5.40	124.94 ± 0.16	262.26 ± 0.33
1024 steps	43.19	5.38	249.31 ± 0.11	131.42 ± 0.05
MDLM+SDTT (loss in FP32)				
32 steps	61.65	5.46	8.037 ± 0.01	$4'077.08 \pm 3.06$
64 steps	50.65	5.43	15.82 ± 0.01	$2'070.67 \pm 0.69$
128 steps	45.06	5.40	31.41 ± 0.01	$1'043.22 \pm 0.16$
256 steps	41.70	5.37	62.54 ± 0.01	523.90 ± 0.06
512 steps	40.63	5.36	124.94 ± 0.16	262.26 ± 0.33
1024 steps	39.50	5.32	249.31 ± 0.11	131.42 ± 0.05
PGM 6 / 6 (dim. 1024)+SDTT (loss in BF16)				
32 steps	91.61	5.56	1.59 ± 0.01	$20'569.99 \pm 95.63$
64 steps	72.73	5.52	3.03 ± 0.01	$10'805.31 \pm 14.11$
128 steps	63.83	5.49	5.93 ± 0.01	$5'518.09 \pm 13.46$
256 steps	58.74	5.47	11.77 ± 0.01	$2'782.78 \pm 3.46$
512 steps	58.77	5.47	23.25 ± 0.01	$1'408.88 \pm 1.05$
1024 steps	56.47	5.46	46.31 ± 0.02	707.52 ± 0.34
PGM 6 / 6 (dim. 1024) nucleus (p=0.9)+SDTT (loss in BF16)				
32 steps	68.33	5.50	1.74 ± 0.01	$18'866.12 \pm 18.35$
64 steps	53.88	5.45	3.18 ± 0.01	$10'307.16 \pm 6.58$
128 steps	46.99	5.42	6.10 ± 0.01	$5'375.20 \pm 2.40$
256 steps	43.22	5.40	11.95 ± 0.01	$2'742.74 \pm 1.32$
512 steps	42.79	5.39	23.63 ± 0.01	$1'386.79 \pm 0.69$
1024 steps	40.99	5.38	46.83 ± 0.02	699.80 ± 0.24
PGM 6 / 6 (dim. 1024)+SDTT (loss in FP32)				
32 steps	84.97	5.52	1.74 ± 0.01	$20'569.99 \pm 95.63$
64 steps	67.60	5.49	3.18 ± 0.01	$10'805.31 \pm 14.11$
128 steps	60.06	5.47	6.10 ± 0.01	$5'518.09 \pm 13.46$
256 steps	55.97	5.45	11.95 ± 0.01	$2'782.78 \pm 3.46$
512 steps	54.13	5.44	$1'408.88 \pm 1.05$	$1'408.88 \pm 1.05$
1024 steps	52.77	5.44	46.83 ± 0.02	707.52 ± 0.34
PGM 6 / 6 (dim. 1024) nucleus (p=0.9)+SDTT (loss in FP32)				
32 steps	63.46	5.45	1.59 ± 0.01	$18'866.12 \pm 18.35$
64 steps	49.94	5.41	3.03 ± 0.01	$10'307.16 \pm 6.58$
128 steps	43.84	5.39	5.93 ± 0.01	$5'375.20 \pm 2.40$
256 steps	40.76	5.36	11.77 ± 0.01	$2'742.74 \pm 1.32$
512 steps	39.46	5.36	23.25 ± 0.01	$1'386.79 \pm 0.69$
1024 steps	38.81	5.35	46.31 ± 0.02	699.80 ± 0.24
PGM 8 / 8 +SDTT (loss in BF16)				
32 steps	102.64	5.54	1.55 ± 0.01	$21'120.99 \pm 83.59$
64 steps	82.93	5.50	3.00 ± 0.01	$10'914.91 \pm 41.69$
128 steps	73.19	5.48	5.86 ± 0.02	$5'585.57 \pm 24.49$
256 steps	70.30	5.47	11.64 ± 0.03	$2'814.99 \pm 9.33$
512 steps	68.07	5.46	22.98 ± 0.02	$1'425.89 \pm 1.61$
1024 steps	65.87	5.44	45.84 ± 0.03	714.71 ± 0.50
PGM 8 / 8 +SDTT (loss in FP32)				
32 steps	87.64	5.51	1.55 ± 0.01	$21'120.99 \pm 83.59$
64 steps	70.47	5.48	3.00 ± 0.01	$10'914.91 \pm 41.69$
128 steps	62.66	5.46	5.86 ± 0.02	$5'585.57 \pm 24.49$
256 steps	59.38	5.45	11.64 ± 0.03	$2'814.99 \pm 9.33$
512 steps	57.57	5.44	22.98 ± 0.02	$1'425.89 \pm 1.61$
1024 steps	56.12	5.44	45.84 ± 0.03	714.71 ± 0.50

Table 8: Throughput (TP) of MDLM and PGM with a context length of 4096, for varying number of inference steps. PGM is significantly faster than MDLM.

Model	TP (4096)	TP (1024)	TP (256)	TP (64)
MDLM	30.45 ± 0.06	121.25 ± 0.02	483.53 ± 0.25	$1'912.16 \pm 1.44$
PGM 8/8	128.99 ± 0.23	697.36 ± 32.83	**2'216.91** ± 3.06	**8'203.82** ± 6.60
PGM 6/6 (dim=1024)	**129.01** ± 0.67	**706.65** ± 36.23	$2'146.60 \pm 15.12$	$8'175.69 \pm 7.85$

Table 9: Comparison of the quality of samples from PGM and MaskGIT with the same number of layers, using the original confidence-based sampler.

Model	FID $\downarrow$			IS $\uparrow$		
	w = 0.0	w = 1.0	w = 4.0	w = 0.0	w = 1.0	w = 4.0
MaskGIT	**14.38**	**7.74**	7.53	**82.49**	**151.26**	**289.75**
PGM 12/12	18.68	8.91	**7.30**	67.39	136.51	289.43
PGM 14/10	21.81	10.18	7.71	59.98	121.06	265.72

Table 10: Comparison of the quality of samples from PGM and MaskGIT with the same number of layers, using the Halton sampler (Besnier et al., 2025).

Model	FID $\downarrow$			IS $\uparrow$		
	w = 0.0	w = 1.0	w = 4.0	w = 0.0	w = 1.0	w = 4.0
MaskGIT	28.84	**4.14**	12.51	59.18	**263.66**	**367.62**
PGM 12/12	**22.58**	**10.07**	**6.38**	**66.50**	134.44	311.06
PGM 14/10	25.09	11.43	**5.54**	**62.17**	120.02	302.36

Table 11: Latency and Throughput of PGM and Maskgit on ImageNet256, when sampling in 32 steps with a batch size of 32. PGM is significantly faster than MaskGIT, especially when using classifier-free guidance (cfg).

Model	Num. params.	TP (sec.)	Speedup	TP (sec.)	Speedup	Lat.
MaskGIT	458M	1.048	1x	1.033	1x	30.599
PGM 12/12	464M	**12.883**	>**6.54**x	**7.771**	>**7.52**x	**2.484**
PGM 14/10	464M	12.715	>6.15x	7.235	>6.97x	2.642