Partition Generative Modeling: Masked Modeling Without Masks

Justin Deschenaux [1] **Lan Tran** [1] **Caglar Gulcehre** [1]

Abstract

We introduce "Partition Generative Models" (PGMs), a novel approach to masked generative modeling (MGMs), particularly effective for masked diffusion language modeling (MDLMs). PGM divides tokens into two distinct groups and employs sparse attention patterns to prevent cross-group information exchange. Hence, the model is trained to predict tokens in one group based solely on information from the other group. This partitioning strategy eliminates the need for `MASK` tokens entirely. While traditional MGMs inefficiently process `MASK` tokens during generation, PGMs achieve greater computational efficiency by operating exclusively on unmasked tokens. Our experiments on OpenWebText with a context length of 1024 tokens demonstrate that PGMs deliver at least 5x improvements in both latency and throughput compared to MDLM when using the same number of sampling steps, while generating samples with better generative perplexity than MDLM. Finally, we show that PGMs can be distilled with Self-Distillation Through Time (SDTT), a method originally devised for MDLM, in order to achieve further inference gains.

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). In particular, 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.

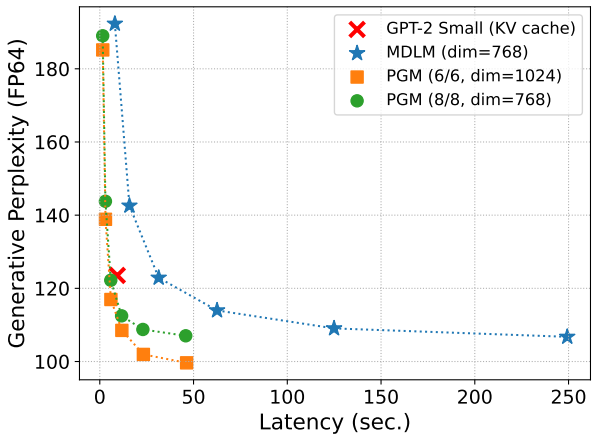


Figure 1. **Latency results:** PGM (ours) achieves better *generative perplexity* and at least 5x improvements in latency compared to MDLM on OpenWebText (Gokaslan & Cohen, 2019). All sampling uses power-of-2 step counts. Standard models are evaluated with 32 to 1024 steps, while the model distilled with SDTT uses from 8 to 1024 steps. The improvements come from our specialized neural network architecture.

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

Despite their ability to generate high-quality samples, MGMs face two key inference efficiency challenges compared to autoregressive models (ARMs). Firstly, unlike ARMs, which benefit from KV caching (Pope et al., 2022), MGM transformer encoders must recompute all activations in each decoding step. Secondly, MGMs process many masked tokens that carry minimal information, particularly in the early sampling stages, when most tokens are masked. In contrast, ARMs only need to process tokens they have already generated.

In this work, we introduce "Partition Generative Models"

[*]Equal contribution [1]Department of Computer Science, EPFL, Lausanne, Switzerland. Correspondence to: Justin Deschenaux <justin.deschenaux@epfl.ch>.

Proceedings of the Workshop on Generative AI for Biology at the 42nd International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025. Copyright 2025 by the author(s).

(PGM). This novel approach does not require processing masked tokens during training and inference and can be implemented using simple attention masking. 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), PGMs only process unmasked tokens during sampling. In contrast, MGMs must always handle full-length sequences. This leads to a significant performance gain for PGMs. Furthermore, while most MGMs compute predictions for all missing positions, PGMs can produce logits solely for the positions that will be effectively denoised in the current sampling step.

Our main contributions can be summarized as follows.

- We introduce "Partition Generative Models" (PGM), a simple and effective alternative to Masked Generative Modeling (MGM). We propose an encoder-decoder variant of the diffusion transformer (Peebles & Xie, 2023) that does not need to process any masked token at inference.
- PGM achieves a reduction of 1.95 in perplexity in LM1B (Chelba et al., 2014), compared to masked diffusion language models (MDLM) (Sahoo et al., 2024). In OpenWebText (Gokaslan & Cohen, 2019), PGMs can generate samples of better quality than MDLM with a 5- 5.5x improvement in throughput and latency, when using the same number of sampling steps as MDLM.
- Additionally, we show that PGMs can be distilled (Deschenaux & Gulcehre, 2025) and achieve 10.73x better latency and superior generative perplexity than GPT-2 with KV caching.

2. Background

2.1. Generative Language Modeling

Language modeling addresses the task of generating sequences of discrete tokens (x_i) from a vocabulary $\mathcal{X} = \mathbb{Z}^{<N} = \{0, ..., N-1\}$. 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 according to a predefined algorithm, for example, denoising a random subset of the `MASK` tokens. The training objective of the denoiser $\mathbf{x}_\theta$ can generally be written as follows:

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

where t determines the proportion of tokens to mask. The corrupted sequence $\mathbf{z}_t$ is generated by independently masking each token in $\mathbf{x}$ with time-dependent probability p_t. For simplicity, we could set $p_t = t$. The weighting function $w : [0,1] \to \mathbb{R}_{\geq 0}$ allows us to assign a different importance to each noise level. Finally, $\text{CE}(x, y)$ denotes the cross-entropy loss between the predicted logits x and the target integer labels y. Oftentimes, the cross-entropy loss is applied exclusively to masked tokens. In such cases, the denoiser model $\mathbf{x}_\theta$ is implemented in such a way that it assigns all probability mass to the input token at positions where the input tokens are *not* masked.

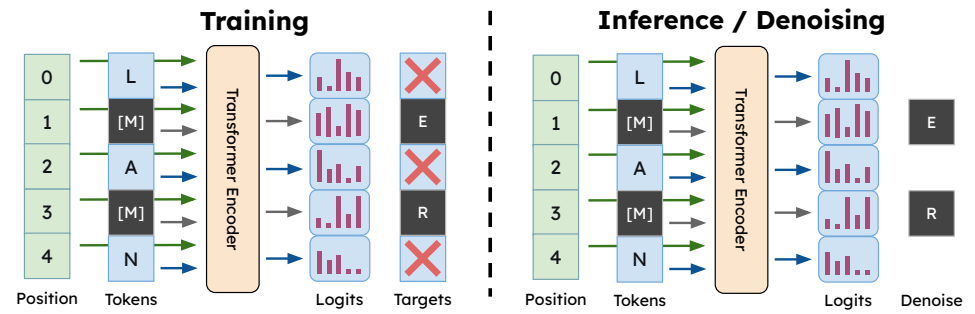
2.3. Masked Diffusion Language Modeling

"Masked diffusion language models" (MDLM) are sequence generative models that operate in discrete space. MDLMs have demonstrated performance comparable to Autoregressive models in validation perplexity and text generation quality. In this work, we use MDLM as our baseline MGM implementation and focus experiments on text generation.

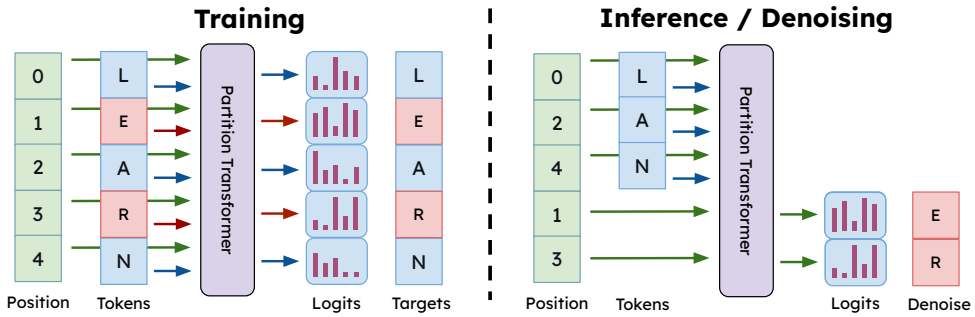
Discrete absorbing diffusion process MDLMs define a forward process that corrupts the data and a backward process that recovers it. For each token in the sequence, the forward process linearly interpolates between one-hot encoding of each token in $\mathbf{x}$ and $\boldsymbol{\pi}$, the absorbing distribution that assigns all mass to the `MASK` token. Formally:

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

where α_t defines the noise injection schedule for $t \in [0,1]$. This schedule must be such that $\alpha_t \in [0,1]$, α_t is strictly decreasing as t increases, and should satisfy the boundary conditions $\alpha_0 \approx 1$ and $\alpha_1 \approx 0$. The process is termed "absorbing" because the corruption is irreversible. Once a token becomes a `MASK` token, it remains masked throughout



(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.

the forward process. The posterior sampling distribution $p_\theta(\mathbf{z}_s|\mathbf{z}_t)$ uses the same analytical form as the true posterior $p(\mathbf{z}_s|\mathbf{z}_t, \mathbf{x})$, where $\mathbf{x}$ denotes a sample 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:

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

Training objective MDLM trains the denoiser $\mathbf{x}_\theta$ using a continuous-time negative evidence lower bound (NELBO) (Sohl-Dickstein et al., 2015), which provides a tighter bound to the log-likelihood of the data than the discrete-time NELBO (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 $\mathbf{x}_\theta$:

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

2.4. Self-Distillation Through Time

"Self-Distillation Through Time" (SDTT) (Deschenaux & Gulcehre, 2025) speeds up the sampling of MDLMs through a process similar to "Progressive Distillation" (Salimans & Ho, 2022). SDTT creates student and teacher copies from a pre-trained MDLM. The student learns to match the prediction that the teacher makes in two steps of size dt. After convergence, the student can be used for a new round of distillation with a step size of $2dt$, further decreasing the number of sampling steps to match the original teacher by a factor of two.

3. Partition Generative Modeling

3.1. Motivations

"Partition Generative Modeling" (PGM) is based on MGM but introduces key modifications to the training and sampling procedures. Most notably, PGMs eliminate the need

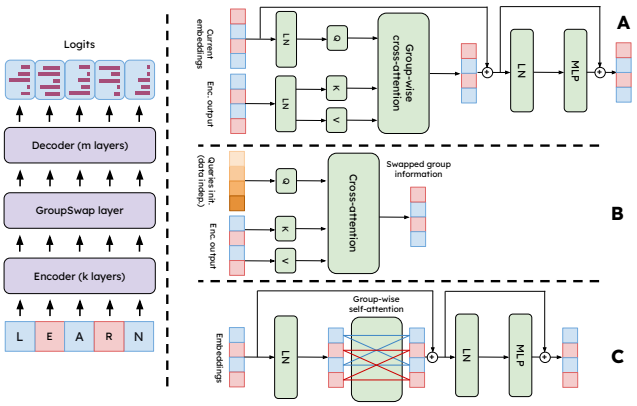


Figure 3. **PGM-compatible transformer architecture.** RoPE (Su et al., 2023) is applied before every attention layer but not shown for clarity. **(A)** Decoder layer with cross-attention to the encoder output and no self-attention between tokens. **(B)** GroupSwap layer that exchanges information between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

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 discrepancy could make MGMs less sample-efficient than AR models. 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 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 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 for 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). Hence, 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. Because each training sequence is partitioned into two groups that predict each other, PGMs implement a mechanism that creates two subtraining examples per "physical" sequence in the batch. Hence, a similar performance improvement could be expected *as if we were training an MGM with a twice larger batch size, with complementary masking between copies*. We isolate and study the effect of complementary masking from the neural 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:

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

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 the group 1. Hence, the tokens in group 0 perceive a noise level of t, while those in group 1 perceive a noise level of $1-t$. Therefore, we must scale the loss for tokens in group 0 according to the noise level t, and by $1-t$ for tokens in group 1. Let w represent the weighing function used to train an MGM. Then, the corresponding PGM weight function w^{PGM} 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 $w(t) = \frac{\alpha_t'}{1-\alpha_t}$, defined in Equation 4. A visual comparison of the training processes for MGM and PGM is provided in Figure 2 (left).

Sampling Since PGMs are trained without communication between tokens in different partitions, PGMs can process only unmasked tokens (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 MASK tokens 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 compute predictions exclusively for those positions, which drastically improves the inference efficiency while retaining the parallel generation abilities. 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 4.

4. PGM Compatible Transformer Architecture

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

Encoder The encoder consists of a series of partition-wise transformer blocks. These blocks operate like 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 for positions to decode.

4.1. The GroupSwap Layer

In the encoder, the information remains separated by group: tokens in group 0 only access information about tokens in group 0, and likewise for group 1. However, we want the neural network to make predictions for positions in group 1 based on information in group 0. Hence, we need to exchange information between groups, and we implement this exchange via a cross-attention layer that we call the GroupSap layer, visualized in Figure 3 (B). Let $\mathbf{g} \in \{0,1\}^L$ be a binary vector that identifies the membership of the group at each position. After the GroupSwap layer, $\tilde{\mathbf{g}}$, the logical NOT of $\mathbf{g}$, tracks which group the information at each position originated from. The main design choice of the GroupSwap layer is how to initialize cross-attention queries. To avoid leakage, if $\tilde{\mathbf{g}}_i = 0$, then the query at position i should not depend on the tokens from partition 1.

Data-Independent Initialization Let $\mathbf{u} \in \mathbb{R}^H$ denote a learnable vector of dimension H. We initialize the queries by replicating $\mathbf{u}$ along the sequence length and adding a fixed positional encoding, followed by a layer norm and a linear layer. 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,

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

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

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

Data-Dependent Initialization Let $X \in \mathbb{R}^{L \times H}$ denote the output of the encoder. We first compute a group-wise aggregation operation over the sequence length, such as `logsumexp` or `mean`, to obtain vectors $Y_0, Y_1 \in \mathbb{R}^H$, which represent the aggregation over groups 0 and 1, respectively. Then, the data-dependent query initialization V' is 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

Empirically, we investigated the performance of PGMs on language modeling and image generation tasks. We present complete details of the hyperparameters of the experiments in Appendix A.

5.1. Language Modeling Performance on LM1B

Experimental setting. We closely follow the MDLM setup (Sahoo et al., 2024) closely and train models with a context length of 128 tokens. The shorter documents are padded to 128 tokens. We train a twelve-layer *diffusion transformer* (Peebles & Xie, 2023) variant without time conditioning and compare it with our Partition Transformer (Section 4). We trained our models with a batch size of 512. We ablate on the architecture choices after training for 200k steps. The model with the best validation perplexity is further trained up to 1M steps and compared with MDLM.

Results Table 1 (left) shows that the use of as many layers in the encoder and decoder achieves the lowest validation perplexity. Additionally, data-independent queries slightly outperform data-dependent queries. After 1M steps, PGM reaches a validation perplexity of 1.95 lower than MDLM.

Algorithm 1 Simplified Sampling for PGMs

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

5.2. Language Modeling on OpenWebText

Experimental setting We closely follow MDLM (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 and compared with MDLM.

Architecture ablations Table 1 (right) shows that, as for LM1B, allocating as many layers to the encoder and decoder achieves the best validation perplexity. However, we find that PGMs with the same number of layers as MDLMs underperform in terms of validation perplexity. Therefore, we experiment with growing the model by using either more layers or growing the embedding dimension. Nonetheless, at inference, our larger PGMs achieve at least a 5x latency and throughput improvement against MDLM, as shown in Figure 1 and Table 4. Empirically, increasing the embedding dimension is more effective in terms of generation quality and speed. We hypothesize that significant 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), for example, in reasoning 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 regular MDLM against MDLM with complementary masking in LM1B (Chelba et al., 2014) and OpenWebText (Gokaslan & Cohen, 2019).

Table 1 shows that while complementary masking reduces the validation perplexity by 1.95 on LM1B, it offers negligible benefits on OWT. This difference might explain why PGMs with the same number of layers outperform MDLMs on LM1B, but not on OWT. In fact, on both data sets, there remains a gap between MDLM with complementary masking and PGM, which can be attributed to the neural network architecture being used. Since complementary masking does not lead to stronger models in OWT, we must increase the size of the model to match the validation perplexity of MDLM. However, recall that PGMs with more parameters generate higher-quality text and are significantly faster than MDLMs in inference (Figure 1). We investigate why complementary masking helps on LM1B but not on OWT in Section 5.7.

5.4. Further speedups via distillation

PGM already delivers improvements over MDLM in both throughput and latency but we can push these gains further by applying "Self Distillation Through Time" (SDTT) (Deschenaux & Gulcehre, 2025). For simplicity, we apply the distillation loss on partition 1, treating it as if it contained MASK tokens. This simple strategy allows us to use the original implementation of SDTT with few modifications. We leave the design of new distillation methods for PGMs to future work.

As shown in Figure 1, our best model, distilled with SDTT, can, in 16 steps, generate samples with better generative

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 (lsm)	32.70
PGM 6 / 6 (mean)	33.89
1M steps	
MDLM	27.67
MDLM (Compl. masking)	**25.72**

Model (OWT)	Val. PPL ↓
200k steps	
MDLM	25.35
MDLM (Compl. masking)	25.32
PGM 6 / 6	26.96
PGM 8 / 8	25.10
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	22.70
PGM 6 / 6 (dim. 1024)	**21.43**

Table 1. 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.

Model	Acc. (%) ↑
MDLM	42.26
PGM 8 / 8	**43.22**
PGM 6 / 6 (dim. 1024)	41.22

Table 2. **Accuracy on LAMBADA.**

perplexity than MDLM would in 1024. *This induces a latency improvement of more than 280x over the original MDLM*. We compare PGM+SDTT with MDLM+SDTT in Appendix B. Additionally, PGM+SDTT achieves a 10.73x improvement over GPT-2 that uses KV-caching, while achieving a lower generative perplexity. See Table 4 and Table 5 for the exact numbers. [t]

5.5. PGM on ImageNet

We evaluated PGM against MDLM on VQGAN tokenized images (Esser et al., 2021). Our preliminary results show that while PGM still offers speedups on image data, the gains are less pronounced than for text generation. This difference stems primarily from the need for a larger encoder relative to the decoder in image tasks. Our experiments are carried out on a smaller scale than a full MaskGIT implementation (Chang et al., 2022), with a comprehensive exploration of PGM architectures for images reserved for future work. Additional details are provided in Appendix B.

5.6. Downstream Performance

We compare the downstream performance of our models on the LAMBADA benchmark (Paperno et al., 2016). Table 2 shows that PGMs achieve a similar accuracy to MDLM, with a slight advantage to the PGM variant with more layers.

5.7. 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, which 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 evidence suggests that complementary masking is particularly beneficial for applications with short context lengths.

6. Related Work

Diffusion on discrete space 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) and discrete flow matching (Campbell et al., 2024; Gat et al., 2024) have demonstrated that MGMs can also generate text whose quality approaches samples from autoregressive models. Our work focuses primarily on the text domain, where our approach appears to be particularly promising. Although recent discrete diffusion and flow matching works focus on the modeling choices in defining the diffusion process, we focus on the inference efficiency. In principle, it should be straightforward to use PGMs with a score parameterization (Meng et al., 2023; Lou et al., 2024; Zhang et al., 2025) instead of the mean parameterization (Sahoo et al., 2024; Shi et al., 2025) that we use.

Accelerating diffusion models via distillation Relatively few works on distillation have been explored for discrete diffusion models. In particular, "Self-Distillation Through Time" (SDTT) (Deschenaux & Gulcehre, 2025), inspired by "Progressive Distillation" (Salimans & Ho, 2022) teaches the student MGM to match multiple sampling steps of the teacher MGM, given corrupted training examples. "Di4C" (Hayakawa et al., 2025) teaches the MGM denoiser $\mathbf{x}_\theta$ the correlation between different positions. Leveraging the uncovered connection between discrete and continuous diffusion, "DUO" (Sahoo et al., 2025) proposes "discrete consistency distillation', a distillation method for uniform discrete diffusion, inspired by consistency models (Song et al., 2023). "Di[M]O" (Zhu et al., 2025) distills models with hybrid absorbing/uniform noise into one-step generators. In contrast, PGMs offer latency and throughput acceleration through a novel architecture, while remaining compatible with distillation, as discussed in Section 5.4.

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. "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 allows 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 arbitrary orders.

Other modalities In images, "MaskGIT" (Chang et al., 2022) demonstrates the power of MGMs. Trained on discrete tokens from a VQGAN (Esser et al., 2021), MaskGIT generates high-quality images in as few as 8 steps, leveraging the parallel token prediction abilities of the denoising neural network. This makes MaskGIT substantially faster than the autoregressive baseline (Esser et al., 2021). MaskGIT has been successfully applied to other modalities, including videos (Villegas et al., 2022; Yu et al., 2023) and audio (Comunità et al., 2024). UniDisc (Swerdlow et al., 2025) trains a multimodal MGM that can process both text and image tokens, while we focus on a single modality in this work.

7. Conclusion

We introduced Partition Generative Modeling (PGM), a novel approach to masked generative modeling that eliminates `MASK` tokens. PGM achieves significant improvements in inference speed while maintaining or improving generation quality. We show that PGMs are compatible with distillation methods devised for masked diffusion models, and present preliminary results on images. 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 are interesting directions. Overall, PGM offers an interesting alternative to masked generative models, with particular advantages for applications where inference speed is critical.

8. Limitations

We need to increase the parameter count of our models to match the validation perplexity of the MDLM baseline when using a context length of 1024. Although PGMs are faster at inference, training is more costly (Appendix C). When applying SDTT to PGMs, we observe smaller improvements in Generative Perplexity than on MDLM. This suggests that the architecture can be further improved in future work. While Partition Generative Modeling is a general principle, the experiments on images are only preliminary, and whether the proposed transformer architecture can be applied in multimodal settings is a question to be studied in future work.

Impact Statement

Language models are dual-use technologies, and thus, they can have unethical uses, such as fake content generation, and they can suffer from bias if applied to data sets that are not carefully curated. This paper focuses specifically on speeding up discrete diffusion language models at test time to reduce their computational demands; while the performance of our PGMs improves over MDLMs, it remains unclear whether diffusion language models can overtake large autoregressive models at scale, hence we do not have specific concerns with regard to this contribution.

References

Arriola, M., Gokaslan, A., Chiu, J. T., Yang, Z., Qi, Z., Han, J., Sahoo, S. S., and Kuleshov, V. Block diffusion: Interpolating between autoregressive and diffusion language models, 2025. URL https://arxiv.org/abs/2503.09573.

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

Ba, J. L., Kiros, J. R., and Hinton, G. E. Layer normalization, 2016. URL https://arxiv.org/abs/1607.06450.

Besnier, V., Chen, M., Hurych, D., Valle, E., and Cord, M. Halton scheduler for masked generative image transformer, 2025. URL https://arxiv.org/abs/2503.17076.

Brown, B., Juravsky, J., Ehrlich, R., Clark, R., Le, Q. V., Ré, C., and Mirhoseini, A. Large language monkeys: Scaling inference compute with repeated sampling, 2024. URL https://arxiv.org/abs/2407.21787.

Campbell, A., Benton, J., Bortoli, V. D., Rainforth, T., Deligiannidis, G., and Doucet, A. A continuous time framework for discrete denoising models, 2022. URL https://arxiv.org/abs/2205.14987.

Campbell, A., Yim, J., Barzilay, R., Rainforth, T., and Jaakkola, T. Generative flows on discrete state-spaces: Enabling multimodal flows with applications to protein co-design, 2024. URL https://arxiv.org/abs/2402.04997.

Chang, H., Zhang, H., Jiang, L., Liu, C., and Freeman, W. T. Maskgit: Masked generative image transformer, 2022. URL https://arxiv.org/abs/2202.04200.

Chelba, C., Mikolov, T., Schuster, M., Ge, Q., Brants, T., Koehn, P., and Robinson, T. One billion word benchmark for measuring progress in statistical language modeling, 2014. URL https://arxiv.org/abs/1312.3005.

Chen, L., Davis, J. Q., Hanin, B., Bailis, P., Stoica, I., Zaharia, M., and Zou, J. Are more llm calls all you need? towards scaling laws of compound inference systems, 2024. URL https://arxiv.org/abs/2403.02419.

Comunità, M., Zhong, Z., Takahashi, A., Yang, S., Zhao, M., Saito, K., Ikemiya, Y., Shibuya, T., Takahashi, S., and Mitsufuji, Y. Specmaskgit: Masked generative modeling of audio spectrograms for efficient audio synthesis and beyond, 2024. URL https://arxiv.org/abs/2406.17672.

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

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019. URL https://arxiv.org/abs/1810.04805.

Dieleman, S., Sartran, L., Roshannai, A., Savinov, N., Ganin, Y., Richemond, P. H., Doucet, A., Strudel, R., Dyer, C., Durkan, C., Hawthorne, C., Leblond, R., Grathwohl, W., and Adler, J. Continuous diffusion for categorical data, 2022. URL https://arxiv.org/abs/2211.15089.

Esser, P., Rombach, R., and Ommer, B. Taming transformers for high-resolution image synthesis, 2021. URL https://arxiv.org/abs/2012.09841.

Gat, I., Remez, T., Shaul, N., Kreuk, F., Chen, R. T. Q., Synnaeve, G., Adi, Y., and Lipman, Y. Discrete flow matching, 2024. URL https://arxiv.org/abs/2407.15595.

Gokaslan, A. and Cohen, V. Openwebtext corpus. http://Skylion007.github.io/OpenWebTextCorpus, 2019.

Goyal, S., Ji, Z., Rawat, A. S., Menon, A. K., Kumar, S., and Nagarajan, V. Think before you speak: Training language models with pause tokens, 2024. URL https://arxiv.org/abs/2310.02226.

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

Haxholli, E., Gurbuz, Y. Z., Can, O., and Waxman, E. 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.

Hayakawa, S., Takida, Y., Imaizumi, M., Wakaki, H., and Mitsufuji, Y. Distillation of discrete diffusion through dimensional correlations, 2025. URL https://arxiv.org/abs/2410.08709.

Heusel, M., Ramsauer, H., Unterthiner, T., Nessler, B., and Hochreiter, S. Gans trained by a two time-scale update rule converge to a local nash equilibrium, 2018. URL https://arxiv.org/abs/1706.08500.

Hu, V. T. and Ommer, B. [mask] is all you need, 2024. URL https://arxiv.org/abs/2412.06787.

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

Kingma, D. P., Salimans, T., Poole, B., and Ho, J. Variational diffusion models, 2023. URL https://arxiv.org/abs/2107.00630.

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

Madaan, A., Tandon, N., Gupta, P., Hallinan, S., Gao, L., Wiegreffe, S., Alon, U., Dziri, N., Prabhumoye, S., Yang, Y., Gupta, S., Majumder, B. P., Hermann, K., Welleck, S., Yazdanbakhsh, A., and Clark, P. Self-refine: Iterative refinement with self-feedback, 2023. URL https://arxiv.org/abs/2303.17651.

Meng, C., Choi, K., Song, J., and Ermon, S. Concrete score matching: Generalized score matching for discrete data, 2023. URL https://arxiv.org/abs/2211.00802.

Ou, J., Nie, S., Xue, K., Zhu, F., Sun, J., Li, Z., and Li, C. Your absorbing discrete diffusion secretly models the conditional distributions of clean data, 2025. URL https://arxiv.org/abs/2406.03736.

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

Paperno, D., Kruszewski, G., Lazaridou, A., Pham, Q. N., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The lambada dataset: Word prediction requiring a broad discourse context, 2016. URL https://arxiv.org/abs/1606.06031.

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

Pope, R., Douglas, S., Chowdhery, A., Devlin, J., Bradbury, J., Levskaya, A., Heek, J., Xiao, K., Agrawal, S., and Dean, J. Efficiently scaling transformer inference, 2022. URL https://arxiv.org/abs/2211.05102.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer, 2023. URL https://arxiv.org/abs/1910.10683.

Sahoo, S. S., Arriola, M., Schiff, Y., Gokaslan, A., Marroquin, E., Chiu, J. T., Rush, A., and Kuleshov, V. Simple and effective masked diffusion language models, 2024. URL https://arxiv.org/abs/2406.07524.

Sahoo, S. S., Deschenaux, J., Gokaslan, A., Wang, G., Chiu, J. T., and Kuleshov, V. The diffusion duality. In *ICLR 2025 Workshop on Deep Generative Model in Machine Learning: Theory, Principle and Efficacy*, 2025. URL https://openreview.net/forum?id=CB0Ub2yXjC.

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

Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., and Chen, X. Improved techniques for training gans, 2016. URL https://arxiv.org/abs/1606.03498.

Schiff, Y., Sahoo, S. S., Phung, H., Wang, G., Boshar, S., Dalla-torre, H., de Almeida, B. P., Rush, A., Pierrot, T., and Kuleshov, V. Simple guidance mechanisms for discrete diffusion models, 2025. URL https://arxiv.org/abs/2412.10193.

Shi, J., Han, K., Wang, Z., Doucet, A., and Titsias, M. K. Simplified and generalized masked diffusion for discrete data, 2025. URL https://arxiv.org/abs/2406.04329.

Shih, A., Sadigh, D., and Ermon, S. Training and inference on any-order autoregressive models the right way, 2022. URL https://arxiv.org/abs/2205.13554.

Snell, C., Lee, J., Xu, K., and Kumar, A. Scaling llm test-time compute optimally can be more effective than scaling model parameters, 2024. URL https://arxiv.org/abs/2408.03314.

Sohl-Dickstein, J., Weiss, E., Maheswaranathan, N., and Ganguli, S. Deep unsupervised learning using nonequilibrium thermodynamics. In Bach, F. and Blei, D. (eds.), *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pp. 2256–2265, Lille, France, 07–09 Jul 2015. PMLR. URL https://proceedings.mlr.press/v37/sohl-dickstein15.html.

Song, Y., Dhariwal, P., Chen, M., and Sutskever, I. Consistency models, 2023. URL https://arxiv.org/abs/2303.01469.

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., and Liu, Y. Roformer: Enhanced transformer with rotary position embedding, 2023. URL https://arxiv.org/abs/2104.09864.

Swerdlow, A., Prabhudesai, M., Gandhi, S., Pathak, D., and Fragkiadaki, K. Unified multimodal discrete diffusion, 2025. URL https://arxiv.org/abs/2503.20853.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need, 2023. URL https://arxiv.org/abs/1706.03762.

Villegas, R., Babaeizadeh, M., Kindermans, P.-J., Moraldo, H., Zhang, H., Saffar, M. T., Castro, S., Kunze, J., and Erhan, D. Phenaki: Variable length video generation from open domain textual description, 2022. URL https://arxiv.org/abs/2210.02399.

von Rütte, D., Fluri, J., Ding, Y., Orvieto, A., Schölkopf, B., and Hofmann, T. Generalized interpolating discrete diffusion, 2025. URL https://arxiv.org/abs/2503.04482.

Wu, Y., Sun, Z., Li, S., Welleck, S., and Yang, Y. An empirical analysis of compute-optimal inference for problem-solving with language models, 2024. URL https://arxiv.org/abs/2408.00724.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding, 2020. URL https://arxiv.org/abs/1906.08237.

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models, 2023. URL https://arxiv.org/abs/2305.10601.

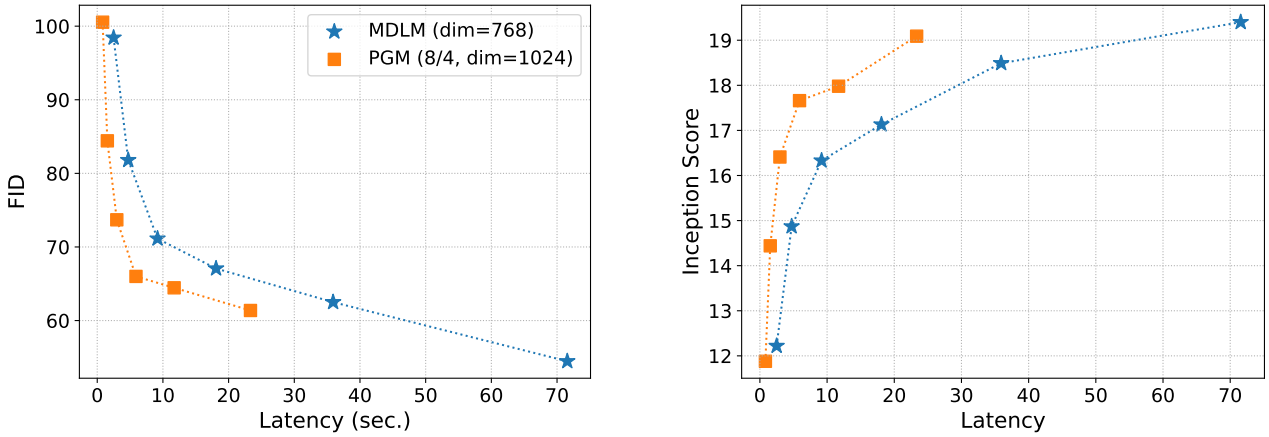
Yu, L., Cheng, Y., Sohn, K., Lezama, J., Zhang, H., Chang, H., Hauptmann, A. G., Yang, M.-H., Hao, Y., Essa, I., and Jiang, L. Magvit: Masked generative video transformer, 2023. URL https://arxiv.org/abs/2212.05199.

Zhang, R., Zhai, S., Zhang, Y., Thornton, J., Ou, Z., Susskind, J., and Jaitly, N. Target concrete score matching: A holistic framework for discrete diffusion, 2025. URL https://arxiv.org/abs/2504.16431.

Zhao, L., Ding, X., Yu, L., and Akoglu, L. Unified discrete diffusion for categorical data, 2024. URL https://arxiv.org/abs/2402.03701.

Zheng, K., Chen, Y., Mao, H., Liu, M.-Y., Zhu, J., and Zhang, Q. Masked diffusion models are secretly time-agnostic masked models and exploit inaccurate categorical sampling, 2025. URL https://arxiv.org/abs/2409.02908.

Zhu, Y., Wang, X., Lathuilière, S., and Kalogeiton, V. Di[M]o: Distilling masked diffusion models into one-step generator, 2025. URL https://arxiv.org/abs/2503.15457.



(a) FID vs. Latency.

(b) Inception Score (IS) vs. Latency.

Figure 4. Comparison of the performance of PGMs and MDLMs when modeling discrete image tokens. For a given compute budget, PGMs outperform MDLMs. Note that we do *not* target state-of-the art result in this experiment, rather we compare the regular MDLM and PGM that were used in experiments on text.

A. 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.

A.1. LM1B

For the LM1B dataset, we employed the `bert-base-uncased` tokenizer with a context length of 128 tokens, padding shorter sequences to 128 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) (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.

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

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 introduce a negligible overhead over MDLM. On OWT, our PGM with 6 encoder and decoder layer and an embedding dimension of 1024 achieves around 75% of the training throughput of MDLM. Recall that at inference, the same PGM is around 5x faster than MDLM. On ImageNet, the PGM achieves 68% of the training throughput of 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
ImageNet (context length 257, batch size 128, trained on 4 GPUs)				
MDLM	0.11 ± 0.01	$1'066.98 \pm 11.13$	0.33 ± 0.00	385.77 ± 0.67
PGM 8 / 4 (dim=1024)	0.17 ± 0.01	727.43 ± 40.58	0.45 ± 0.02	279.15 ± 8.74

A.3. ImageNet

For the ImageNet dataset, we used a pre-trained VQGAN tokenizer (Esser et al., 2021; Besnier et al., 2025) with a downsampling factor of 16, resulting in 256 tokens per image (16x16 grid). 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 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 4 GPUs on a single node. 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. Instead of the whole ImageNet, we used the Imagenet256 datast, downloaded from Kaggle, as used in prior work (Hu & Ommer, 2024). Besides the neural network hyperparameters, the other parameters were unchanged when training the PGM. Since the dataset version does not have a predefined validation set, we use 50'000 randomly selected ones as validation. We train our models to be class unconditional. We use a dictionary with $16'386$ values, representing the $16'384$ values from the VQGAN codebook (Esser et al., 2021; Besnier et al., 2025), a `MASK` token and a special "BOS" token.

A.4. Impact of Numerical Precision on Sampling

Prior work (Zheng et al., 2025) identified that Masked Diffusion Models often achieve lower generative perplexity results because of underflow in the logits when sampling using 16-bit precision. The resulting decrease in token diversity can make evaluations based solely on generative perplexity misleading. To ensure fair comparison across all models, we cast all logits to FP64 before sampling for every model in our experiments, including GPT-2.

A.5. Sample-based evaluation

Generative perplexity We evaluate the quality of the generated text using the generative perplexity as our primary metric, following previous 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 sequences generated by our models. 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 $x_{<i}$.

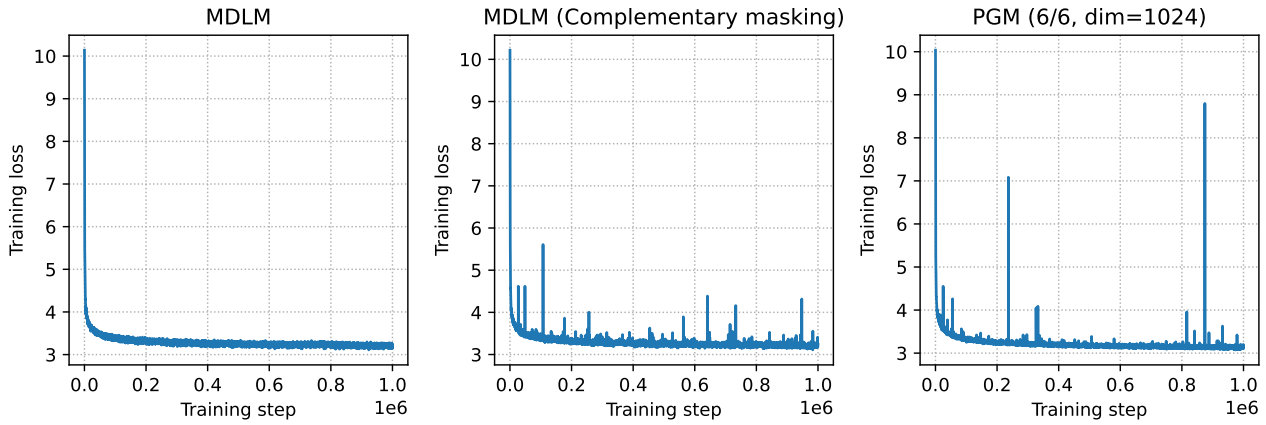


Figure 5. Training loss of MDLM, MDLM with Complementary Masking (Section 5.3) and PGM. Complementary masking seems to introduce spikes in the loss, even though it did not cause the models to diverge.

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 allows us to catch repetitive generation, as seen in prior work (Dieleman et al., 2022).

Fréchet Inception Distance and Inception Score For image generation tasks, we evaluate the quality of generated images using the Fréchet Inception Distance (FID) (Heusel et al., 2018) and Inception Score (IS) (Salimans et al., 2016). For efficiency, we use $10'000$ generated images to compute those metrics.

B. Additional results

B.1. MDLM+SDTT vs PGM+SDTT

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

Using a higher precision also impacts the distillation process, which works by compressing two sampling steps into one. Therefore, generating the teacher targets using higher precision increases the final generative perplexity compared to the results reported by SDTT, as shown in Table 6 and Figure 6. However, since we made only this minimal change while otherwise using the official SDTT implementation, we are confident in the validity of our results.

For a fair comparison, we re-distilled our MDLM checkpoints with SDTT using higher precision. Specifically, we used FP32 precision when generating teacher targets, allowing us to compare MDLM+SDTT against our PGM+SDTT with the lowest generative perplexity. We observe that after distillation, the entropy of samples generated with MDLM+SDTT decreases much more than the entropy of samples generated by PGM+SDTT. Figure 6 shows that MDLM+SDTT achieves lower generative perplexity than PGM+SDTT, which indicates clear opportunities for architectural improvements in future work.

C. 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.

C.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. Despite this training overhead, recall that we focus on the inference efficiency. 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 varied by dataset: approximately 22 hours for LM1B, 4.5 days for OWT, and 3.8 days for ImageNet.

C.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- 5.5x improvements in latency and throughput over MDLM while reaching superior generative perplexity. For inference measurements, we use a single NVIDIA A100-SXM4-80GB GPU and report both latency and throughput metrics. The efficiency gain stems from the ability of PGMs to process only unmasked tokens during inference, as illustrated in Figure 2. Table 4 compares MDLM and PGMs on the generative perplexity, unigram entropy, latency, and throughput. We compute mean and standard deviation of the latency and throughput over 20 batches after two warmup batches.

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. MDLMs performed slightly better with BF16 precision, while PGMs showed improved results when using FP32 precision for the PyTorch Lightning trainer. Note that in both cases, the neural network backbone computations were consistently performed in BF16. The precision difference only affected operations outside the model, such as loss calculations, where models trained in FP32 achieved slightly lower validation loss.

C.4. Licensing

Our code and model artifacts will be released under the MIT license. Regarding datasets, 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 2 MDLM-equivalent sampling for PGMs.

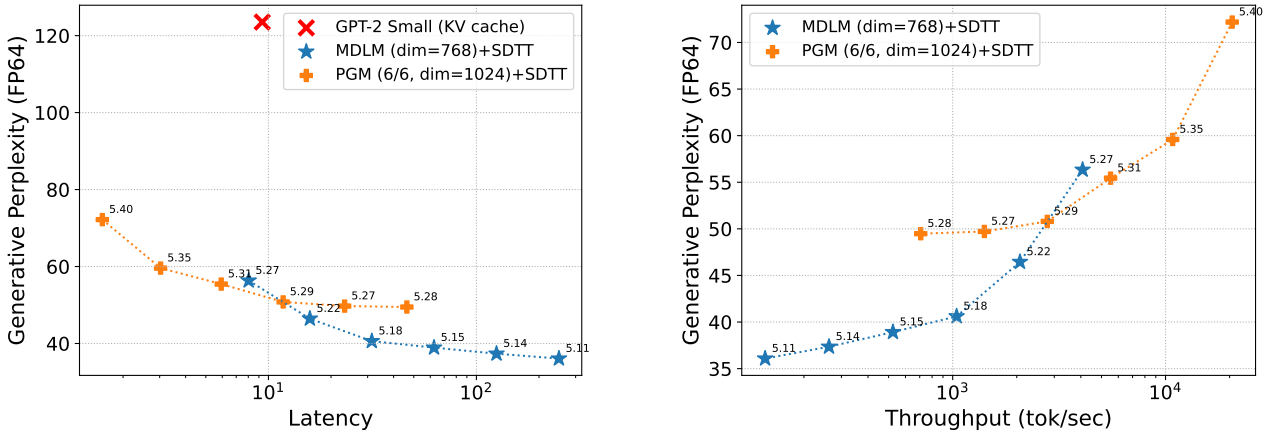
```
Input: Batch size BS, number of steps K, model length L, special BOS index
Output: Generated samples x
x ← empty_tensor(BS, 1)                                   ▷ Initialize
x[:, 0] ← BOS                                             ▷ Set BOS as first token
k ← L/K                                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

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

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

Algorithm 4 Extract the correct number of predictions per sequence

```
Input: x, concrete_lengths, n_denoise_per_seq, denoised_token_values, clean_positions, noisy_positions, noisy_pos_input
Output: Updated x, clean_positions, noisy_positions, concrete_lengths
new_concrete_lengths ← concrete_lengths + n_denoise_per_seq                ▷ Update sequence lengths
n_tok_to_add ← max(new_concrete_lengths) - shape(x, 1)                     ▷ Calculate padding needed
if n_tok_to_add ¿ 0 then
    pad ← zeros(BS, n_tok_to_add)                                          ▷ Create padding tensor
    x ← concat(x, pad, dim=1)                                              ▷ Pad the sequences
    clean_positions ← concat(clean_positions, pad, dim=1)                  ▷ Pad the positions
end if
for i in range(BS) do
    if n_denoise_per_seq[i] = 0 then
        continue                                                           ▷ Skip if no tokens to denoise
    end if
    x[i, concrete_lengths[i]:new_concrete_lengths[i]] ←
        denoised_token_values[i, :n_denoise_per_seq[i]]
    clean_positions[i, concrete_lengths[i]:new_concrete_lengths[i]] ←
        noisy_pos_input[i, :n_denoise_per_seq[i]]
    noisy_positions[i, :shape(noisy_positions, 1) - n_denoise_per_seq[i]] ←
        noisy_positions[i, n_denoise_per_seq[i]:]
end for
return x, clean_positions, noisy_positions, new_concrete_lengths
```



(a) Generative PPL vs. Latency.

(b) Throughput vs. Latency.

Figure 6. Comparison of PGMs and MDLMs *after distillation with SDTT* on OWT. We see that PGMs achieve higher generative perplexity than MDLM. This result suggests that our proposed PGM transformer architecture, while effective before distillation, can be improved in future work.

Table 4. 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.

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 5. Sample quality and efficiency of PGM 6 / 6 (dim. 1024) with uniform sampling on OpenWebText with different numbers of sampling steps *after distillation with SDTT*. We generate sequences of 1024 tokens with a batch size of 32 to measure the latency and throughput. See Table 4 for un-distilled models.

Model	**Gen. PPL** ↓	**Entropy** ↑	**Latency** ↓ **(ms)**	**Throughput** ↑ **(tok/s)**
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
+ *SDTT (10k steps per round, loss in FP32, 7 rounds)*				
8 steps	176.62	5.48	0.51 ± 0.00	$63'342.23 \pm 206.49$
16 steps	100.50	5.44	0.87 ± 0.00	$37'607.96 \pm 49.61$
32 steps	72.19	5.40	1.59 ± 0.01	$20'569.99 \pm 95.63$
64 steps	59.59	5.35	3.03 ± 0.01	$10'805.31 \pm 14.11$
128 steps	55.44	5.31	5.93 ± 0.01	$5'518.09 \pm 13.46$
256 steps	50.80	5.29	11.77 ± 0.01	$2'782.78 \pm 3.46$
512 steps	49.72	5.27	23.25 ± 0.01	$1'408.88 \pm 1.05$
1024 steps	49.48	5.28	46.31 ± 0.02	707.52 ± 0.34
+ *SDTT (10k steps per round, loss in FP32, 5 rounds)*				
8 steps	268.81	5.62	0.51 ± 0.00	$63'342.23 \pm 206.49$
16 steps	140.89	5.59	0.87 ± 0.00	$37'607.96 \pm 49.61$
32 steps	92.06	5.54	1.59 ± 0.01	$20'569.99 \pm 95.63$
64 steps	73.94	5.50	3.03 ± 0.01	$10'805.31 \pm 14.11$
128 steps	64.45	5.47	5.93 ± 0.01	$5'518.09 \pm 13.46$
256 steps	60.70	5.46	11.77 ± 0.01	$2'782.78 \pm 3.46$
512 steps	57.94	5.44	23.25 ± 0.01	$1'408.88 \pm 1.05$
1024 steps	57.26	5.43	46.31 ± 0.02	707.52 ± 0.34
+ *SDTT (10k steps per round, loss in FP64, 7 rounds)*				
8 steps	223.52	5.57	0.51 ± 0.00	$63'342.23 \pm 206.49$
16 steps	124.70	5.54	0.87 ± 0.00	$37'607.96 \pm 49.61$
32 steps	86.64	5.49	1.59 ± 0.01	$20'569.99 \pm 95.63$
64 steps	71.93	5.46	3.03 ± 0.01	$10'805.31 \pm 14.11$
128 steps	63.35	5.43	5.93 ± 0.01	$5'518.09 \pm 13.46$
256 steps	60.44	5.42	11.77 ± 0.01	$2'782.78 \pm 3.46$
512 steps	58.29	5.40	23.25 ± 0.01	$1'408.88 \pm 1.05$
1024 steps	58.33	5.41	46.31 ± 0.02	707.52 ± 0.34
+ *SDTT (10k steps per round, loss in FP64, 5 rounds)*				
8 steps	289.35	5.66	0.51 ± 0.00	$63'342.23 \pm 206.49$
16 steps	152.81	5.63	0.87 ± 0.00	$37'607.96 \pm 49.61$
32 steps	101.61	5.59	1.59 ± 0.01	$20'569.99 \pm 95.63$
64 steps	82.73	5.56	3.03 ± 0.01	$10'805.31 \pm 14.11$
128 steps	70.36	5.53	5.93 ± 0.01	$5'518.09 \pm 13.46$
256 steps	68.23	5.53	11.77 ± 0.01	$2'782.78 \pm 3.46$
512 steps	65.69	5.51	23.25 ± 0.01	$1'408.88 \pm 1.05$
1024 steps	64.22	5.50	46.31 ± 0.02	707.52 ± 0.34

Table 6. Sample quality and efficiency *after distillation* of PGM 6 / 6 (dim. 1024) and MDLM. We generate sequences of 1024 tokens with a batch size of 32 to measure the latency and throughput. See Table 4 for un-distilled models.

Model	**Gen. PPL** ↓	**Entropy** ↑	**Latency** ↓ **(ms)**	**Throughput** ↑ **(tok/s)**
PGM+SDTT (10k steps per round, loss in FP32, 7 rounds)				
32 steps	72.19	5.40	1.59 ± 0.01	$20'569.99 \pm 95.63$
64 steps	59.59	5.35	3.03 ± 0.01	$10'805.31 \pm 14.11$
128 steps	55.44	5.31	5.93 ± 0.01	$5'518.09 \pm 13.46$
256 steps	50.80	5.29	11.77 ± 0.01	$2'782.78 \pm 3.46$
512 steps	49.72	5.27	23.25 ± 0.01	$1'408.88 \pm 1.05$
1024 steps	49.48	5.28	46.31 ± 0.02	707.52 ± 0.34
MDLM+SDTT (10k steps per round, loss in FP32, 7 rounds)				
32 steps	56.34	5.27	8.037 ± 0.01	$4'077.08 \pm 3.06$
64 steps	46.45	5.22	15.82 ± 0.01	$2'070.67 \pm 0.69$
128 steps	40.61	5.18	31.41 ± 0.01	$1'043.22 \pm 0.16$
256 steps	–	–	62.54 ± 0.01	523.90 ± 0.06
512 steps	–	–	124.94 ± 0.16	262.26 ± 0.33
1024 steps	–	–	249.31 ± 0.11	131.42 ± 0.05

Table 7. Sample quality and efficiency on ImageNet with different numbers of sampling steps.

Model	**FID** ↓	**IS** ↑	**Latency** ↓ **(ms)**	**Throughput** ↑ **(tok/s)**
MDLM				
8 steps	98.43	12.22	2.49 ± 0.01	$26'331.16 \pm 23.77$
16 steps	81.80	14.87	4.72 ± 0.01	$13'918.07 \pm 4.57$
32 steps	71.13	16.33	9.18 ± 0.01	$7'164.60 \pm 2.93$
64 steps	67.07	17.13	18.09 ± 0.01	$3'635.35 \pm 1.09$
128 steps	62.50	18.49	35.91 ± 0.01	$1'831.69 \pm 0.44$
256 steps	54.47	19.40	71.56 ± 0.02	919.31 ± 0.26
PGM 8 / 4 (dim. 1024, non-uniform)				
8 steps	102.34	11.56	1.70 ± 0.01	$3'635.70 \pm 421.41$
16 steps	85.22	13.92	3.02 ± 0.02	$21'725.97 \pm 188.56$
32 steps	74.69	16.06	5.65 ± 0.03	$11'641.26 \pm 67.16$
64 steps	68.48	17.54	10.83 ± 0.10	$6'070.08 \pm 59.86$
128 steps	65.83	18.71	20.46 ± 0.23	$3'215.10 \pm 35.89$
256 steps	63.44	18.52	37.71 ± 0.52	$1'744.82 \pm 24.00$
PGM 8 / 4 (dim. 1024, uniform)				
8 steps	100.54	11.88	0.82 ± 0.00	$79'838.18 \pm 65.09$
16 steps	84.42	14.44	1.54 ± 0.01	$42'483.88 \pm 49.88$
32 steps	73.69	16.41	2.98 ± 0.01	$22'006.84 \pm 26.18$
64 steps	66.00	17.66	5.90 ± 0.01	$11'148.64 \pm 8.70$
128 steps	64.45	17.98	11.71 ± 0.01	$5'615.94 \pm 2.75$
256 steps	61.37	19.09	23.33 ± 0.01	$2'819.38 \pm 0.58$