State-Space Architectures for Scalable Diffusion-based 3D Molecule Generation

Adrita Das[*]
Independent Researcher
adrita.riman@gmail.com

Peiran Jiang
Carnegie Mellon University
peiranj@andrew.cmu.edu

Dantong Zhu
Columbia University
dantongzhu1103@gmail.com

Barnabas Poczos
Carnegie Mellon University
bapoczos@cs.cmu.edu

Jose Lugo-Martinez
Carnegie Mellon University
jlugomar@andrew.cmu.edu

Abstract

Generative models, particularly diffusion models, have shown significant promise in accelerating molecular therapeutic and material discovery. However, efficiently generating high-quality, large, and complex molecules remains a challenge. Previous approaches often struggle with handling intricate molecular structures, suffer from slow and memory-intensive diffusion processes, and lack effective mechanisms for capturing long-range dependencies in molecular graphs. In this study, we aim to leverage the long-range dependency modeling capability of the State-Space Models (SSMs) to extend its applicability to 3D molecule generation. Additionally,we introduce a framework leveraging a few-step iterative diffusion process based on a Euclidean State-Space Model for efficient molecule generation. We further show that the combination of structured state transitions and adaptive node selection reduces memory footprint during both training and sampling, enabling efficient generation of molecules with hundreds of atoms. We evaluate the framework along both efficiency and quality dimensions, presenting results on inference speed, training FLOPs, and standard generation quality metrics. Code, models, and datasets are publicly available in this Repository

1 Introduction

Generation of novel, valid molecules is a computationally intensive task, as navigating the vast chemical space often requires significant resources to process and model complex molecular interactions. Recent advances in deep learning have accelerated this process, but the computational demands of these models remain a challenge, particularly in resource-constrained environments. In the rapidly evolving field of molecular therapeutics and materials discovery, generative models hold immense promise for streamlining key stages of the design process. Diffusion models [9, 19, 26], in particular, have emerged as a leading tool due to their ability to generate diverse high quality samples from learned distributions. These models excel in unconditional generation [13] and with additional data-driven guidance, they are also capable of conditional generation [6, 32]. Current molecular generation methods, while advancing the field of drug discovery and materials science, face significant inefficiencies due to inherent limitations in their design and computational requirements. These challenges stem from the complexity of molecular structures, the high-dimensional search space, and the computational cost of existing models. As the size of the molecule increases, the search space grows exponentially, leading to longer generation times and higher resource consumption. The scalability of diffusion models, especially those employing transformer architectures, has been long

[*]Corresponding Author.

constrained by their intrinsic quadratic computational cost. GeoDiff [30] achieves state-of the-art performance in molecular conformer generation but faces significant scalability challenges. The iterative denoising process increases computational complexity as molecule size grows, while enforcing roto-translational invariance adds further memory overhead. Graph Diffusion Transformer (Graph DiT) [18] enables multi-conditional molecular generation by integrating a graph-dependent noise model and a Transformer-based denoiser, achieving superior performance across various molecular properties. However, its scalability is limited due to the quadratic complexity of the Transformer architecture and the inefficiency of iterative denoising, especially for large and complex molecules. Existing molecule generation methods are typically optimized for small molecules and fail to scale efficiently to larger systems, such as polymers, due to iterative bottlenecks, gradient instability, and computational overhead [11, 31]. We propose a scalable SE(3)-equivariant diffusion framework built on the Directly Denoising Diffusion Model (DDDM) by Zhang et al. [34], combining few-step generation with architectural optimizations to significantly reduce runtime and memory usage while maintaining high fidelity. [2] Our key contributions are as follows:

1. **A Euclidean State-Space Diffusion Framework for 3D Molecules.** We introduce an SE(3)-equivariant diffusion model built on Euclidean State-Space Models, enabling efficient modeling of long-range molecular dependencies while preserving geometric equivariance.
2. **Few-step Direct Denoising with Structured State Transitions.** Building on the DDDM formulation, we design a small-step iterative generation process that reduces the number of denoising steps while maintaining sample quality, substantially improving runtime and training efficiency.
3. **Adaptive Node Selection for Scalable Molecular Generation.** We integrate an input-driven node selection mechanism that dynamically sparsifies message passing, reducing memory overhead and enabling scaling to molecules with hundreds of atoms.
4. **A GPU-efficient Equivariant Message Passing Layer.** We incorporate a CG-optimized SE(3)-equivariant kernel that accelerates tensor products and reduces memory traffic, enabling efficient equivariant reasoning at scale.
5. **Compute–quality tradeoffs on large-molecule generation.** We demonstrate substantial improvements in inference speed, memory efficiency, and training compute while achieving competitive or superior performance compared to existing diffusion-based molecular generators.

2 Notations and Preliminaries

2.1 Diffusion Models

Diffusion models (DDPMs) [9, 19, 24] are generative models that learn data distributions via a forward process (adding noise) and a reverse process (removing noise). Let $\{\beta_i\}_{i=1}^T$ denote a sequence of positive noise scales such that $0 < \beta_1, \beta_2, \ldots, \beta_T < 1$. The forward process gradually adds Gaussian noise to the data x_0 over time steps t in a Markov chain: $p_{\alpha_t}(\mathbf{x}_t \mid \mathbf{x}_0) = \mathcal{N}\big(\mathbf{x}_t;\, \sqrt{\alpha_t}\,\mathbf{x}_0,\, (1-\alpha_t)\mathbf{I}\big)$, where $\bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s)$ is the cumulative noise schedule and $x_0 \sim p^* \propto \exp(-f^*)$ is a sample from the true data distribution and $f^*(x)$ is the corresponding energy function. We consider a discrete Markov chain $x_0, x_1, \ldots, x_T$. The perturbed data distribution is denoted as $p_\alpha(\hat{\mathbf{x}}) := \int p^*(\mathbf{x})\, p_\alpha(\hat{\mathbf{x}} \mid \mathbf{x})\, d\mathbf{x}$. A variational Markov chain in the reverse direction is parameterized as $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}\Big(\mathbf{x}_{t-1}; \frac{1}{\sqrt{1-\beta_t}}\big(\mathbf{x}_t + \beta_t\, s_\theta(\mathbf{x}_t, t)\big),\, \beta_t\mathbf{I}\Big)$. Samples are produced via an ancestral sampling procedure, following $\mathbf{x}_{t-1} = \frac{1}{\sqrt{1-\beta_i}}\,\big(\mathbf{x}_t + \beta_t\, s_{\theta^*}(\mathbf{x}_t,)\big) + \sqrt{\beta_t}\,\mathbf{z}_t$.

where, $s_\theta(x_t, t)$ is the score functions parameterized by θ, which predicts the noise component in x_t. The optimal parameters θ^* are learned by minimizing the expected denoising error between the predicted and true noise over all timesteps.

Directly Denoising Diffusion. From [25] reformulation of DDPMs as Stochastic Differential Equations (SDEs) to generalize the forward process: $d\mathbf{X}_t = -\frac{1}{2}\beta(t)\mathbf{X}_t\, dt + \sqrt{\beta(t)}\, d\mathbf{B}_t$, where $\mathbf{B}_t$ represents Brownian motion. The reverse-time VP SDE is: $d\mathbf{X}_t = \big[-\frac{1}{2}\beta(t)(\mathbf{X}_t + \nabla \log p_t(\mathbf{X}_t))\big]\, dt +$

[2]We will release all trained models on Hugging Face.

$\sqrt{\beta(t)}\,d\mathbf{B}_t$, where $\nabla \log p_t$ is estimated by a time-dependent score network $s_\theta(\mathbf{x}, t)$. Directly Denoising Diffusion Models (DDDMs) [34] integrate DDPMs with the Probability Flow (PF) ODE framework, enabling faster denoising without complex solvers. The solution of the PF ODE is obtained by evaluating the integral expression $\mathbf{x}_0 = \mathbf{x}_T + \int_0^T -\frac{1}{2}\beta(t)[\mathbf{x}_t + \nabla_{\mathbf{x}_t} \log p_t(\mathbf{x}_t)]\,dt$, where $\mathbf{x}_T \sim \mathcal{N}(0, I)$. Directly Denoising Diffusion Models (DDDM) refine the estimate of the clean state $\mathbf{x}_0$ by leveraging the probability flow ODE. Specifically, the mapping $f(\mathbf{x}_0, \mathbf{x}_t, t) = \mathbf{x}_t - F(\mathbf{x}_0, \mathbf{x}_t, t)$ is defined, where F involves an integral of the drift term parameterized by the noise schedule $\beta(t)$. A neural approximation $f_\theta(\mathbf{x}_0, \mathbf{x}_t, t) = \mathbf{x}_t - F_\theta(\mathbf{x}_0, \mathbf{x}_t, t)$ is trained such that $f_\theta \approx f$.

$$\Theta := \arg\min_\theta \; \mathbb{E}_{t\sim\mathcal{U}[1,T]}\left[\mathbb{E}_{\mathbf{x}_0\sim p_{\text{data}}(\mathbf{x}_0)}\left[\mathbb{E}_{\mathbf{x}_t\sim\mathcal{N}\left(\sqrt{\bar\alpha_t}\mathbf{x}_0,(1-\bar\alpha_t)\mathbf{I}\right)}\left[d\big(f_\theta(\mathbf{x}_0^{(n)}, \mathbf{x}_t, t), \mathbf{x}_0\big)\right]\right]\right] \tag{1}$$

where, $d(\cdot,\cdot)$ is a suitable distance metric. We define Θ as the set of optimal neural parameters for the denoising network. The metric function $d(\cdot,\cdot)$ satisfies standard properties: for all vectors x and y, $d(x, y) \geq 0$, and $d(x, y) = 0$ if and only if $x = y$.

3 Methodology

3.1 Denoising Framework

Our denoising framework is designed on the GraphGPS [21]: an MPNN+Global Attention Hybrid. The GPS framework is built on three key components: (i) positional or structural encoding (SE/PE), (ii) a local message-passing mechanism, and (iii) a global attention mechanism. The processing modules of scalable GPS construct a computational graph that integrates message-passing graph neural networks (MPNNs) with Transformer-based global attention, using attention mechanisms with linear complexity $O(V)$ in the number of nodes. Our denoising framework replaces linear attention mechanisms in the GraphGPS framework with selective state space layers, enabling input-driven graph sparsification. Leveraging the inherent modularity of the GraphGPS framework, we replace the standard global attention layers with state-space model–based layers, including Mamba [8], Mamba-2 [5], Hydra, and Jamba. This modular design allows for seamless integration of alternative architectures, enabling systematic comparison across different state-space formulations within the same overall denoising pipeline. Unlike dense attention, where all nodes can attend to one another, most SSMs update each node only with information from preceding nodes, creating positional asymmetry in the available context. For SSMs with unidirectional scans, we employ two node-ordering schemes. In the importance-based scheme, nodes are ordered by heuristic scores (e.g., node degree or eigenvector centrality), with high-importance nodes placed later in the sequence to exploit richer contextual information. Alternatively, we follow [29] and shuffle node ordering during training to preserve permutation invariance. In our framework, the node embeddings are augmented with structural or positional encodings (SE/PE). In addition, we add sinusoidal timestep embeddings into the inputs of our model to provide explicit temporal conditioning.

3.1.1 Model Implementations

In this section, we discuss about the SSM blocks used for our experiments and the model architecture of the denoising model. For the state-space modeling components, we experiment with recent structured state-space layers, specifically the Jamba [17] architecture and the Hydra [10] employing the generalized matrix mixer framework. Jamba is based on a hybrid Transformer-Mamba mixture-of-experts (MoE) architecture. Jamba interleaves Transformer and Mamba layers, with optional MoE blocks for scalable capacity. This flexible architecture supports configurations tailored to different computational resource constraints and task objectives. By varying the ratio of Transformer to Mamba layers, the architecture can flexibly trade off between memory efficiency, training time, and long-context modeling capacity. Hydra is introduced as a bidirectional extension of Mamba within the structured state-space model (SSM) framework. The design is motivated by recent insights that SSMs can be understood as semiseparable matrix mixers, where the transition dynamics are parameterized by semiseparable matrices that enable efficient sequence mixing. While such matrices underpin the computational efficiency of models like Mamba, their inherent causality constraint limits them to unidirectional processing, restricting applicability in tasks where bidirectional context is essential. To overcome this limitation, Hydra leverages the matrix mixer perspective discussed in [10] and adopts quasiseparable matrices as its core building block.For the message passing module in our denoising network, we employ the recently introduced GPU-optimized Clebsch–Gordan (CG) sparse kernel generator of Bharadwaj et al. [2]. This kernel provides an efficient implementation of the $O(3)$-equivariant tensor product by decomposing the CG contraction into register-level subkernels, minimizing global memory traffic, and fusing the contraction with subsequent graph convolution operations. Using this kernel allows our equivariant message passing layers to maintain full rotational, translational, reflectional, and permutation equivariance while significantly reducing the computational bottlenecks typically associated with CG-based equivariant GNNs. We trained the models with architectures listed in the Table 1, which we describe in the next section. [3]

Table 1: Architectural configurations of the proposed methods

Model	L	$\mathbf{d}_{\text{state}}$	$\mathbf{d}_{\text{conv}}$	$\mathbf{d}_{\text{model}}$	**# Heads**	**Params**
EGNN + Transformer	8	–	–	512	8	63.0M
EGNN + Mamba	6	512	256	512	8	63.4M
EGNN + Mamba-2	6	512	256	512	8	63.2M
EGNN + Hydra	6	256	128	256	4	55.7M
EGNN + Jamba	6	256	128	256	4	286.9M

3.2 Model Architectures

In this section, we discuss the architectures of our message passing and SSM module in detail. Modern architectures, like the Transformer, consist of two key elements or components that is: a sequence mixer (Multi-Head Attention) and a channel mixer (Feed-Forward Network). Most sequence mixers can be expressed as matrix multiplications of the form $Y = MX$. This is often defined as the Matrix Mixer framework which includes sequence based models such as attention, state-space models etc. As discussed in [15], the key insight lies in the structure of the mixer matrix M. Building on this foundation, [10] proposed a systematic methodology for designing new mixers or architectures. The major bottleneck lies the matrix multiplication in M which might incur quadratic costs. Self-Attention offers strong flexibility and effectiveness for sequence mixing, but comes at a high computational cost. Most modern sequence-based architectures rely on two core principles: data-dependent matrix mixing and scalability, allowing sequence mixers to generalize beyond their training lengths.

3.2.1 Implementations for a Single 80GB GPU

Transformer. Our Transformer block follows the encoder–decoder architecture of Vaswani et al. [27]. The model is composed of a stack of layers, where each layer contains a multi-head self-attention mechanism followed by a position-wise feed-forward network. Residual connections are employed around each of these sub-layers, and layer normalization is applied to the outputs. We set the channel dimension for our transformer block to 512.

Mamba. Our configuration employed a Mamba block with a state size of 256, model dimension of 256, and 4 attention heads. Our architecture follows Gu and Dao [8] in omitting explicit positional encodings, while using RMSNorm [33] for normalization. Mamba extends structured state space models by allowing parameters to vary as functions of the input, enabling content-based reasoning over discrete modalities. This selective mechanism allows the model to adaptively propagate or forget information depending on the token context. Mamba incorporates a hardware-optimized recurrent implementation that achieves linear scaling in sequence length and up to 5× faster inference than Transformers. The model dimension is expanded by an expansion factor, which we set to 2 for all our Mamba blocks.

[3]For comparison, we also benchmark this implementation against the equivariant graph neural network from Satorras et al. [22] (EGNN), which provides an $E(3)$-equivariant alternative that does not rely on Clebsch–Gordan tensor products. This comparison highlights both the efficiency gains and the expressivity differences offered by CG-based approach.

Mamba-2. For Mamba-2, we adopt the same overall architecture as in Mamba, with each layer substituted by the revised Mamba-2 block from Dao and Gu [5]. We set the internal Mamba-2 state dimension to 128 and expansion factor 2. We set d_{conv} as the dimensionality of the internal convolutional layer in each block to 64.

Hydra. Hydra builds on the conceptual foundation of the Matrix Mixer Sequence Models. Hwang et al. [10] introduced Hydra which uses a quasiseparable matrix mixer framework. Quasiseparable matrices generalize both the low-rank matrix mixers of linear attention and the semiseparable matrices of state space models, used as bidirectional extension of the semiseparable matrices. Hydra restricts M to being a structured matrix which is essential for subquadratic matrix multiplication algorithms. The state dimension of the hydra module is 256 and d_{conv} is set to 128 and expansion factor 2.

Jamba. Jamba is a novel Attention-SSM hybrid architecture, which combines Transformer layers with the Mamba layers at a certain ratio along with the mixture-of-experts (MoE) [4] module. To improve model capacity without proportionally increasing computational cost, Jamba incorporates Mixture-of-Experts (MoE) layers in place of MLP blocks. Specifically, MoE is applied in alternating layers, with 8 experts per layer and the top-2 experts selected for each token, following [17]. MoE block is used every 2 layers instead of MLP. Also, following [17] each Jamba block used in our architecture is a combination of mamba and attention layers with the ratio of attention-to-Mamba layers set to 1:6. Each layer in the block is either a Mamba or an Attention layer, followed by a multi-layer perceptron (MLP). Jamba stabilizes large-scale training with RMSNorm in Mamba layers, removing the need for positional embeddings such as RoPE [4]. The architecture uses standard components, including (Group Query Attention) GQA [3], SwiGLU activation function [23], and MoE load balancing.

3.3 Fused Clebsch-Gordon Module

The tensor product of two irreducible representations $x^{(l_1)}$ and $y^{(l_2)}$ lives in a $(2l_1+1)(2l_2+1)$–dimensional space and remains equivariant under the action of $O(3)$. This tensor product decomposes into a direct sum of irreps according to

$$D^{(l_1)}(g) \otimes D^{(l_2)}(g) \;=\; \bigoplus_{l=|l_1-l_2|}^{l_1+l_2} D^{(l)}(g). \tag{2}$$

which allows the resulting features to be reorganized across type-l spaces. The Clebsch–Gordan (CG) tensor product provides an explicit rule for constructing each type-l component using CG coefficients $C^{(l,m)}_{(l_1,m_1)(l_2,m_2)}$.

For vectors $x^{(l_1)} \in \mathbb{R}^{2l_1+1}$ and $y^{(l_2)} \in \mathbb{R}^{2l_2+1}$, the m-th component of the type-l output is given by

$$\big(x^{(l_1)} \otimes_{\mathrm{cg}} y^{(l_2)}\big)^{(l)}_m \;=\; \sum_{m_1=-l_1}^{l_1} \sum_{m_2=-l_2}^{l_2} C^{(l,m)}_{(l_1,m_1)(l_2,m_2)}\, x^{(l_1)}_{m_1}\, y^{(l_2)}_{m_2}. \tag{3}$$

In the model under consideration, this Clebsch–Gordan (CG) interaction appears inside the message-passing update. Substituting the corresponding instantiation, the type-(l_o,p_o) output for node a at layer k takes the form

$$V^{(k,l_o,p_o)}_{ac_0m_o} = \sum_{l_f,l_i,p_i} \sum_{m_f,m_i} C^{(l_o,m_o)}_{(l_i,m_i)(l_f,m_f)} \frac{1}{|\mathcal{N}_a|} \sum_{b\in\mathcal{N}_a} \sum_{c} \psi^{(k,l_o,l_f,l_i,p_i)}_{abc}\, Y^{(l_f)}_{m_f}(\hat{r}_{ab})\, V^{(k-1,l_i,p_i)}_{bcm_i}. \tag{4}$$

The core computational bottleneck lies in efficiently evaluating the Clebsch–Gordan (CG) tensor product that arises when interacting equivariant feature vectors within an equivariant neural network layer. Given two feature vectors $x \in \mathbb{R}^n$ and $y \in \mathbb{R}^m$, each transforming under representations (D_{in}, D_x) and (D_{in}, D_y) respectively, a new output z must be produced that remains equivariant to a target representation D_z. While the Kronecker product $x \otimes y$ provides a general equivariant interaction, its dimensionality nm is often prohibitive. The standard remedy applies a structured,

[4]All experiments are conducted using the Jamba variant without explicit positional encodings, as the Mamba layers implicitly capture positional information.

block-sparse transform P containing Clebsch–Gordan coefficients, which decomposes $x \otimes y$ into Wigner blocks. The redundant blocks can then be removed, and the resulting components are recombined through a structured weight matrix W, yielding

$$z = W\,P(x \otimes y), \tag{5}$$

with the desired equivariant representation. This operation, known as the CG tensor product, can be expressed equivalently through matrix multiplication, tensor contraction, or Einstein notation. Efficient evaluation of this operator, denoted $\mathrm{TP}(P, x, y, W)$, is essential for practical deployment of rotation-equivariant models. We use the efficient CG product computation from Bharadwaj et al. [2] for our module.

Table 2: Configuration of the Jamba-based architectures

Model	L	**d**$_{\text{state}}$	**d**$_{\text{conv}}$	**d**$_{\text{model}}$	**Params**	**# Experts**	**Top-**K	**Active Params**
EGNN + Jamba	8	512	256	512	2.15B	16	4	537.5M
EGNN + Jamba	6	256	128	256	286.9M	8	2	71.7M
EGNN + Jamba	10	512	256	512	1.68B	8	2	420M
EGNN + Jamba	10	256	128	256	478M	8	2	119.5M
EGNN + Jamba	8	128	64	128	144.5M	16	4	36.1M

3.4 Experimental Setup and Baselines

For the training and evaluation of our models, we utilize the GEOM dataset [1], a collection of high-quality molecular conformations generated using metadynamics within the CREST software [20]. The node and edge chemical features, f_a and f_{ab}, follow the construction described in [12]. The node features include atom identity, atomic number, aromaticity, degree, hybridization, implicit valence, formal charge, ring membership, and ring size, yielding a 74-dimensional feature vector for the GEOM-DRUGS subset. Edge features are represented as a 4-dimensional one-hot encoding of the bond type. We adopt an 84%/10%/6% split for the training, validation, and test sets, respectively, and all models are trained on approximately $55{,}000$ molecules from the training split. We also construct a dataset from GEOM-MoleculeNet and call it as GEOM-Long range, containing molecules with more than 100 atoms to assess the ability of our models to generalize to larger molecules.

3.5 Training Infrastructure and Hyperparameters

We train all models using a batch size of 128 with a learning rate that peaks at 3×10^{-4} and decays to a minimum of 3×10^{-5}. A warmup phase is applied, followed by a cosine learning rate schedule. Optimization is performed using Adam with a weight decay of 0.1 and momentum parameters $\beta_1 = 0.9$ and $\beta_2 = 0.95$. All experiments are conducted using the BF16 precision format. For fairness, we use identical hyperparameter settings across all models without individual tuning, ensuring that any performance differences arise from architectural variations rather than optimization choices. We trained the models using Distributed Data Parallel (DDP) with microbatch gradient accumulation. Each minibatch is divided into several microbatches, which are processed sequentially; gradients are accumulated locally using PyTorch's `no_sync` mechanism and synchronized only on the final microbatch.

We employ a variance-preserving (VP) diffusion process with the standard cosine noise schedule proposed by Nichol & Dhariwal [19]. The cumulative product of signal coefficients is defined as

$$\bar{\alpha}_t = \frac{f(t)}{f(0)}, \qquad f(t) = \cos^2\left(\frac{t/T + s}{1 + s} \cdot \frac{\pi}{2}\right), \tag{6}$$

where $s = 0.008$ is a small offset introduced for numerical stability. The discrete noise variance at timestep t is then given by

$$\beta_t = 1 - \frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}, \tag{7}$$

which ensures a smooth signal-to-noise ratio decay throughout the diffusion process.

3.6 Evaluation

We evaluate our model on both standard GEOM metrics benchmarks.

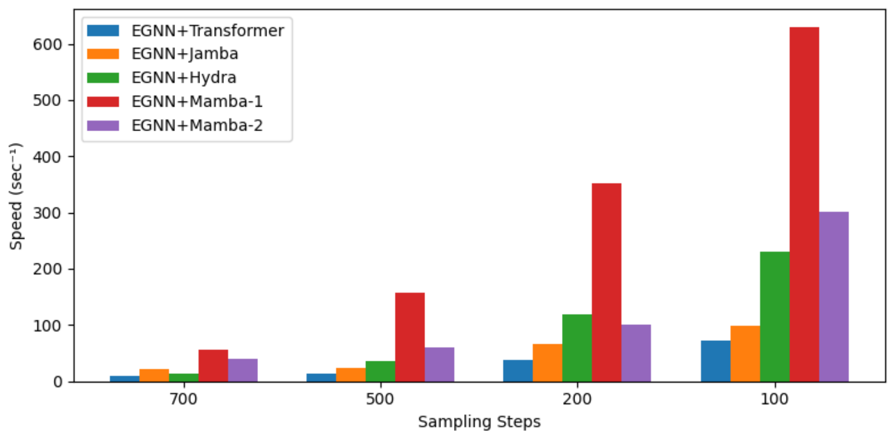

Figure 1: Comparison of sampling speed across different methods using equivariant message passing.

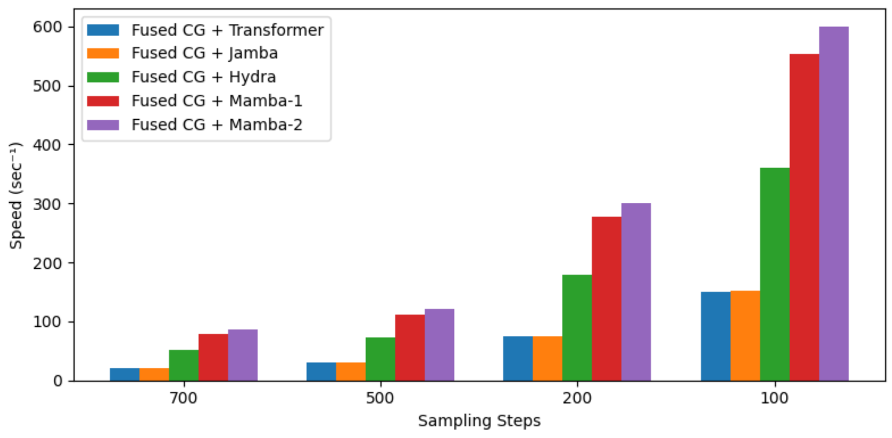

Figure 2: Comparison of sampling speed using Fused CG across different methods.

4 Results

4.1 Throughput Analysis

Following Lacombe et al.[16], we evaluate sampling speed using both the number of diffusion steps and the average number of samples generated per second over 10,000 samples. We compare the speed-up in sample time or reduction in per-step latency across different methods and how it influences the sample quality. The throughput and sampling speed were being evaluated on a single H100 80GB GPU.

Figures 1 and 2 report the sampling throughput across different equivariant message passing backbones, while Tables 3–5 quantify the corresponding samples-per-second rates under varying sampling steps and architectural configurations. Across all methods, throughput scales approximately inversely with the number of sampling steps, reflecting the dominant contribution of per-step latency to overall runtime. At a fixed number of diffusion steps, structured state-space and matrix-mixer architectures consistently achieve substantially higher throughput than attention-based models. In particular, at 100 sampling steps, Transformer-based models achieve approximately 150 samples/s, whereas Hydra, Mamba-1, and Mamba-2 reach approximately 360, 554, and 600 samples/s, respectively. This corresponds to a $2.4\times$–$4.0\times$ speed-up over Transformer-based models under identical sampling conditions. As the number of sampling steps increases, this performance gap becomes more pronounced. At 700 steps, Transformer-based models operate at roughly 21 samples/s, while Hydra and Mamba-based models achieve between 51 and 86 samples/s, yielding a $2.4\times$–$4.1\times$ improvement in throughput. This trend indicates that architectures with linear-time recurrence and structured state

Table 4: Performance with fixed MoE parameters at different sampling steps

Methods	**Sampling Steps**	**Speed (sec^{-1})**	**Total Experts**	**Top Experts/Token**	**Novelty (%)**	**QED**
Fused CG + Jamba	700	20.23	16	2	99.33	0.66
Fused CG + Jamba	500	28.32	16	2	98.67	0.63
Fused CG + Jamba	200	70.8	16	2	95.72	0.557
Fused CG + Jamba	100	153.6	16	2	94.36	0.55

transitions amortize computational cost more effectively over long diffusion trajectories. We further observe that Hydra consistently occupies an intermediate regime between attention-based models and pure state-space models, benefiting from bidirectional matrix mixing while maintaining favorable computational scaling. Mamba-1 and Mamba-2 exhibit the highest throughput across all sampling regimes, with Mamba-2 achieving the best overall sampling speed, particularly in the low-step regime where per-step overhead dominates. Overall, these results demonstrate that sampling throughput is primarily governed by per-step computational complexity, and that replacing attention with structured state-space or matrix-mixer layers yields substantial reductions in sampling latency. This enables faster generation without modifying the diffusion process itself, making such architectures particularly attractive for large-scale or latency-sensitive molecular generation settings.

Table 3: Performance of various methods across different sampling steps using the Fused CG MP.

Methods	Sampling Steps	Speed (sec^{-1})	Novelty(%)	Diversity(%)	Validity(%)	QED
Fused CG + Transformer	700	21.37	92.3	89.2	88.26	0.42
Fused CG + Transformer	500	29.92	92.3	89.2	88.26	0.42
Fused CG + Transformer	200	74.8	91.14	88.7	88.67	0.45
Fused CG + Transformer	100	149.6	89.56	84.5	82.33	0.32
Fused CG + Jamba	700	21.6	97.52	99.5	99.32	0.65
Fused CG + Jamba	500	30.24	97.82	99.51	98.35	0.65
Fused CG + Jamba	200	75.6	96.54	96.86	98.29	0.49
Fused CG + Jamba	100	151.2	96.12	96.62	97.15	0.42
Fused CG + Hydra	700	51.36	99.1	98.56	99.66	0.65
Fused CG + Hydra	500	71.90	97.1	97.25	98.5	0.55
Fused CG + Hydra	200	179.75	96.58	96.35	96.56	0.56
Fused CG + Hydra	100	359.5	95.43	95.25	95.77	0.49
Fused CG + Mamba-1	700	79.17	97.65	98.91	96.65	0.63
Fused CG + Mamba-1	500	110.84	97.65	98.91	96.65	0.63
Fused CG + Mamba-1	200	277.1	96.54	97.67	95.52	0.59
Fused CG + Mamba-1	100	554.2	94.45	95.65	95.56	0.55
Fused CG + Mamba-2	700	85.78	96.64	97.54	95.57	0.62
Fused CG + Mamba-2	500	120.09	96.64	97.54	95.57	0.62
Fused CG + Mamba-2	200	300.225	95.52	95.61	94.45	0.57
Fused CG + Mamba-2	100	600.45	94.57	94.45	93.67	0.52

4.2 Role of MoE in Model Efficiency

We further analyze the effect of Mixture-of-Experts (MoE) routing on sampling throughput. In MoE models, each token or node is routed to a small subset of the total experts, allowing the model to increase capacity while keeping per-step computational cost low. Table 5 summarizes the performance of *Fused CG + Jamba* with different MoE configurations at 100 sampling steps. Increasing the total number of experts generally increases capacity but has a minimal impact on per-step speed, because only the top 1 or 2 experts are activated for each token. For example, moving from 2 to 16 total experts changes the speed only marginally from 116 samples/s to 142 samples/s, demonstrating the efficiency of sparse expert activation in reducing latency while preserving model expressivity.

Table 5: Performance with MoE Parameters at 100 sampling steps

Methods	Sampling Steps	Speed (sec^{-1})	#Total Experts	#Top Experts at each token	Novelty (%)	QED
Fused CG + Jamba	100	115.97	2	1	99.01	0.516
Fused CG + Jamba	100	116.01	4	1	98.79	0.524
Fused CG + Jamba	100	120.61	8	1	99.59	0.575
Fused CG + Jamba	100	141.6	16	1	99.06	0.586

4.3 Training Efficiency and Inference Speed

We measure the Model FLOP Utilisation (MFU) and compare it against Transformer-based baseline models. Following prior works [14, 28], we define the model FLOPs per second as the number of floating point operations (FLOPs) required for a single forward and backward pass, divided by the iteration time. The MFU is computed as the ratio between the model FLOPs per second and the peak theoretical FLOPs per second of the hardware (GPU) used during training. Formally,

$$\text{MFU} = \frac{\text{FLOPs per forward \& backward pass}/t_{\text{iter}}}{\text{Peak FLOPs of GPU}}, \tag{8}$$

where t_{iter} denotes the iteration time.

All MFU measurements were collected during training on four H100 GPUs with a data-parallel size of 128. The graph denoising model based on the attention–SSM hybrid (Jamba-style) achieved an MFU of approximately 25%. The Transformer-based baseline achieved an MFU of nearly 37%. The Hydra-based baseline recorded an MFU of 20.68%, while Mamba-1 and Mamba-2 achieved MFUs of 22.34% and 23.39%, respectively. The pure SSM model benefits from inference-time speedups, and the hybrid attention–SSM model additionally benefits from its Mamba blocks during inference compared to the pure Transformer baseline.

5 Conclusion and Future Work

Across our experiments, we observe clear trade-offs between throughput, sample quality, and architectural choices. The hybrid attention–SSM architecture (Jamba) strikes a favorable balance: while not as fast as the pure Mamba variants, it consistently produces samples with strong validity and novelty, outperforming both the transformer and Hydra baselines in terms of sample quality. Importantly, when scaling the mixture-of-experts configuration, we find that increasing the total number of experts per layer while keeping the number of active experts per token fixed provides additional capacity without substantially affecting inference speed. This suggests that sparsely activated MoE designs allow the model to grow in representational power without incurring proportional computational overhead. Hydra-based models similarly improve inference-time throughput but show somewhat lower training efficiency, as reflected in their lower MFU. Our results suggest that SSM-based models hold considerable potential for accelerating inference. One of the key limitations of our method is that, as DDDM maintains $x_0^{(n)}$ for every sample in the dataset, it incurs additional memory overhead during training. In our extended work, we aim to establish theoretical guarantees, particularly convergence results for diffusion inference, to complement and strengthen the empirical findings. We also aim to explore the use of more efficient or expressive equivariant graph neural networks or kernel-based methods to further accelerate the inference.

References

[1] Simon Axelrod and Rafael Gómez-Bombarelli. Geom: Energy-annotated molecular conformations for property prediction and molecular generation. *ArXiv*, abs/2006.05531, 2020. URL https://api.semanticscholar.org/CorpusID:219558923.

[2] Vivek Bharadwaj, Austin Glover, Aydin Buluc, and James Demmel. An efficient sparse kernel generator for o(3)-equivariant deep networks, 2025. URL https://arxiv.org/abs/2501.13986.

[3] Yuang Chen, Cheng Zhang, Xitong Gao, Robert D. Mullins, George A. Constantinides, and Yiren Zhao. Optimised grouped-query attention mechanism for transformers, 2024. URL https://arxiv.org/abs/2406.14963.

[4] Zixiang Chen, Yihe Deng, Yue Wu, Quanquan Gu, and Yuanzhi Li. Towards understanding mixture of experts in deep learning, 2022. URL https://arxiv.org/abs/2208.02813.

[5] Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality, 2024. URL https://arxiv.org/abs/2405.21060.

[6] Xin Ding, Yongwei Wang, Kao Zhang, and Z. Jane Wang. Ccdm: Continuous conditional diffusion models for image generation, 2025. URL https://arxiv.org/abs/2405.03546.

[7] Florian Grötschla, Jiaqing Xie, and Roger Wattenhofer. Benchmarking positional encodings for gnns and graph transformers, 2024. URL https://arxiv.org/abs/2411.12732.

[8] Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces, 2024. URL https://arxiv.org/abs/2312.00752.

[9] Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models, 2020. URL https://arxiv.org/abs/2006.11239.

[10] Sukjun Hwang, Aakash Lahoti, Tri Dao, and Albert Gu. Hydra: Bidirectional state space models through generalized matrix mixers, 2024. URL https://arxiv.org/abs/2407.09941.

[11] Wengong Jin, Regina Barzilay, and Tommi Jaakkola. Multi-objective molecule generation using interpretable substructures, 2020. URL https://arxiv.org/abs/2002.03244.

[12] Bowen Jing, Gabriele Corso, Jeffrey Chang, Regina Barzilay, and Tommi Jaakkola. Torsional diffusion for molecular conformer generation, 2023. URL https://arxiv.org/abs/2206.01729.

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

[14] Vijay Korthikanti, Jared Casper, Sangkug Lym, Lawrence McAfee, Michael Andersch, Mohammad Shoeybi, and Bryan Catanzaro. Reducing activation recomputation in large transformer models, 2022. URL https://arxiv.org/abs/2205.05198.

[15] Goomba Lab. Hydra part 1: Matrix mixer. https://goombalab.github.io/blog/2024/hydra-part1-matrix-mixer/, 2024. Accessed: 2025-08-24.

[16] Romain Lacombe and Neal Vaidya. Accelerating the generation of molecular conformations with progressive distillation of equivariant latent diffusion models, 2024. URL https://arxiv.org/abs/2404.13491.

[17] Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. Jamba: A hybrid transformer-mamba language model, 2024. URL https://arxiv.org/abs/2403.19887.

[18] Gang Liu, Jiaxin Xu, Tengfei Luo, and Meng Jiang. Graph diffusion transformers for multi-conditional molecular generation, 2024. URL https://arxiv.org/abs/2401.13858.

[19] Alex Nichol and Prafulla Dhariwal. Improved denoising diffusion probabilistic models, 2021. URL https://arxiv.org/abs/2102.09672.

[20] Philipp Pracht, Fabian Bohle, and Stefan Grimme. Automated exploration of the low-energy chemical space with fast quantum chemical methods. *Phys. Chem. Chem. Phys.*, 22:7169–7192, 2020. doi: 10.1039/C9CP06869D. URL http://dx.doi.org/10.1039/C9CP06869D.

[21] Ladislav Rampášek, Mikhail Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer, 2023. URL https://arxiv.org/abs/2205.12454.

[22] Victor Garcia Satorras, Emiel Hoogeboom, and Max Welling. E(n) equivariant graph neural networks, 2022. URL https://arxiv.org/abs/2102.09844.

[23] Noam Shazeer. Glu variants improve transformer, 2020. URL https://arxiv.org/abs/2002.05202.

[24] Jascha Sohl-Dickstein, Eric A. Weiss, Niru Maheswaranathan, and Surya Ganguli. Deep unsupervised learning using nonequilibrium thermodynamics, 2015. URL https://arxiv.org/abs/1503.03585.

[25] Yang Song and Stefano Ermon. Improved techniques for training score-based generative models, 2020. URL https://arxiv.org/abs/2006.09011.

[26] Yang Song, Jascha Sohl-Dickstein, Diederik P. Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. Score-based generative modeling through stochastic differential equations, 2021. URL https://arxiv.org/abs/2011.13456.

[27] 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.

[28] Roger Waleffe, Wonmin Byeon, Duncan Riach, Brandon Norick, Vijay Korthikanti, Tri Dao, Albert Gu, Ali Hatamizadeh, Sudhakar Singh, Deepak Narayanan, Garvit Kulshreshtha, Vartika Singh, Jared Casper, Jan Kautz, Mohammad Shoeybi, and Bryan Catanzaro. An empirical study of mamba-based language models, 2024. URL https://arxiv.org/abs/2406.07887.

[29] Chloe Wang, Oleksii Tsepa, Jun Ma, and Bo Wang. Graph-mamba: Towards long-range graph sequence modeling with selective state spaces, 2024. URL https://arxiv.org/abs/2402.00789.

[30] Minkai Xu, Lantao Yu, Yang Song, Chence Shi, Stefano Ermon, and Jian Tang. Geodiff: a geometric diffusion model for molecular conformation generation, 2022. URL https://arxiv.org/abs/2203.02923.

[31] Nianzu Yang, Huaijin Wu, Kaipeng Zeng, Yang Li, and Junchi Yan. Molecule generation for drug design: A graph learning perspective. *arXiv preprint arXiv:2301.XXXXX*, 2023. URL https://arxiv.org/abs/2301.XXXXX. Department of Computer Science and Engineering, MoE Key Lab of Artificial Intelligence, Shanghai Jiao Tong University.

[32] Zheyuan Zhan, Defang Chen, Jian-Ping Mei, Zhenghe Zhao, Jiawei Chen, Chun Chen, Siwei Lyu, and Can Wang. Conditional image synthesis with diffusion models: A survey, 2025. URL https://arxiv.org/abs/2409.19365.

[33] Biao Zhang and Rico Sennrich. Root mean square layer normalization, 2019. URL https://arxiv.org/abs/1910.07467.

[34] Dan Zhang, Jingjing Wang, and Feng Luo. Directly denoising diffusion models. In *Forty-first International Conference on Machine Learning*, 2024. URL https://openreview.net/forum?id=k5ncz7TIPX.

A Appendix

A.1 Efficient Subkernel Decomposition for CG Operations

In practice, the Clebsch–Gordan (CG) tensor product admits substantial structure that can be exploited for efficient computation. Each nonzero block of the CG coefficient tensor P corresponds to a Wigner triple (ℓ_x, ℓ_y, ℓ_z) and is small, highly sparse, and repeated many times across the full tensor. Rather than repeatedly invoking a naïve tensor contraction for every block, modern equivariant models decompose the CG tensor product into a sequence of specialized subkernels that align with the sparsity and block structure of P and the associated weight matrix W. Two patterns dominate in existing architectures. *Kernel B* contracts a sparse CG block with multiple segments of x and a shared segment of y, followed by multiplication with a diagonally arranged submatrix of W, and is employed in models such as NequIP and MACE. *Kernel C* performs an analogous contraction but uses a dense weight submatrix $W \in \mathbb{R}^{b' \times b}$, as seen in DiffDock and earlier 3D-equivariant classifiers. Although additional kernel patterns are theoretically possible, empirical model designs rely almost exclusively on these two due to their alignment with common CG block structure and their computational efficiency.

To efficiently execute Clebsch–Gordan (CG) tensor products at scale, the kernel assigns each triple (x, y, W) to an independent GPU warp, enabling fully coalesced memory access and removing the need for CTA-level synchronization. The inputs are staged in shared memory and processed through a sequence of subkernels that correspond to the structured operations of the CG tensor product. Since CG feature vectors may exceed shared memory capacity, the computation is split into multiple phases determined at compile time, with each phase loading only the required slices of x, y, W, and z. The scheduler minimizes global memory traffic by reusing cached segments whenever possible, using simple heuristics that cover most configurations in equivariant GNNs. Each subkernel is JIT-generated, unrolling sparse CG tensor contractions into fused arithmetic instructions to eliminate runtime indirection and maximize parallelism. Depending on the structure of the weight matrix W, the kernel performs either diagonal scaling (Kernel B) or a warp-level dense matrix multiplication (Kernel C). This design enables efficient execution even for large batches and high-cost configurations, where each CG tensor product may require tens of thousands of FLOPs and multiple staged computation phases. We express the Clebsch–Gordan tensor product as

$$z = P\,(x \otimes y)\,W, \tag{9}$$

where P is the block-sparse CG coefficient tensor, $x \in \mathbb{R}^{b_x}$ and $y \in \mathbb{R}^{b_y}$ are feature vectors grouped by irreps, and W is the learned weight matrix.

The CG operator decomposes into blocks indexed by Wigner triples:

$$P = \bigoplus_{(\ell_x, \ell_y, \ell_z)} P_{\ell_x \ell_y \ell_z}. \tag{10}$$

The full contraction can be written as:

$$z_{\ell_z} = \sum_{\ell_x, \ell_y} P_{\ell_x \ell_y \ell_z}\,\left(x_{\ell_x} \otimes y_{\ell_y}\right) W_{\ell_x \ell_y \ell_z}. \tag{11}$$

where each $P_{\ell_x \ell_y \ell_z}$ is small and sparse.

A.2 Kernel B: Diagonal Weight Submatrix

If W is block-diagonal:

$$W_{\ell_x \ell_y \ell_z} = \operatorname{diag}(w_1, \ldots, w_b), \tag{12}$$

$$z_{\ell_z} = \sum_{\ell_x, \ell_y} \left(P_{\ell_x \ell_y \ell_z}(x_{\ell_x} \otimes y_{\ell_y})\right) \odot w. \tag{13}$$

This corresponds to diagonal scaling after sparse CG contraction.

A.3 Kernel C: Dense Weight Submatrix

When W is fully dense:

$$W_{\ell_x \ell_y \ell_z} \in \mathbb{R}^{b' \times b}, \tag{14}$$

the contraction becomes:

$$z_{\ell_z} = \sum_{\ell_x, \ell_y} W_{\ell_x \ell_y \ell_z} \left[P_{\ell_x \ell_y \ell_z} (x_{\ell_x} \otimes y_{\ell_y}) \right]. \tag{15}$$

This requires a warp-level dense GEMM per block.

A.4 Warp-Level Parallel Scheduling

Each triple $(x_{\ell_x}, y_{\ell_y}, W_{\ell_x \ell_y \ell_z})$ is assigned to a GPU warp:

$$\text{warp}_i \leftarrow (\ell_x, \ell_y, \ell_z). \tag{16}$$

Shared memory staging splits CG feature vectors:

$$x_{\ell_x} = \bigcup_{p=1}^{P} x_{\ell_x}^{(p)}, \tag{17}$$

$$y_{\ell_y} = \bigcup_{p=1}^{P} y_{\ell_y}^{(p)}, \tag{18}$$

and the final output accumulates partials:

$$z_{\ell_z} = \sum_{p=1}^{P} z_{\ell_z}^{(p)}. \tag{19}$$

The fused structured operation for each block is

$$z_{\ell_z}^{(p)} = \begin{cases} \left(P_{\ell_x \ell_y \ell_z}^{(p)} (x^{(p)} \otimes y^{(p)}) \right) \odot w, & \text{(Kernel B)}, \\ W_{\ell_x \ell_y \ell_z} \left(P_{\ell_x \ell_y \ell_z}^{(p)} (x^{(p)} \otimes y^{(p)}) \right), & \text{(Kernel C)}. \end{cases} \tag{20}$$

$$\boxed{x, y \xrightarrow{\text{warp}} \begin{cases} P(\cdot) \odot w & \text{Kernel B,} \\ W\, P(\cdot) & \text{Kernel C} \end{cases} \xrightarrow{\text{accumulate}} z}$$

A.5 Backward Pass

The kernel system implements the full derivative pipeline for the Clebsch–Gordan (CG) tensor product. Given two irreducible representations $x \in \mathbb{R}^{2\ell_x+1}$ and $y \in \mathbb{R}^{2\ell_y+1}$, the CG tensor product produces an output $z \in \mathbb{R}^{2\ell_z+1}$ through a sparse coefficient tensor P. The forward computation evaluates

$$z[k] = \sum_{(i,j) \in \mathcal{N}_k} P[i,j,k]\, x[i]\, y[j], \tag{21}$$

where $\mathcal{N}_k$ denotes the nonzero index pairs for output channel k. This sparse traversal is fused with the dense multiplication by a weight matrix $W \in \mathbb{R}^{b \times b'}$, enabling

$$z' = Wz \tag{22}$$

to be computed in a single pass without additional global-memory traffic.

The backward kernel jointly computes gradients with respect to x, y, and W. Let $g_z = \partial E / \partial z$ denote the gradient of a scalar energy E. Linearity of the tensor product yields

$$\frac{\partial E}{\partial x[i]} = \sum_{(i,j,k) \in \mathcal{N}} P[i,j,k]\, y[j]\, (W^\top g_z)[k], \tag{23}$$

$$\frac{\partial E}{\partial y[j]} = \sum_{(i,j,k)\in\mathcal{N}} P[i,j,k]\, x[i]\, (W^\top g_z)[k], \tag{24}$$

$$\frac{\partial E}{\partial W} = g_z\, z^\top. \tag{25}$$

All three gradients are produced within a single fused kernel using warp-level reductions to combine contributions across threads.

Higher-order derivatives required for force-based training are computed without introducing new kernels. Mixed partials of the form $\partial^2 E/(\partial R\, \partial W)$ are obtained by expressing them as linear combinations of outputs from existing forward and backward routines, enabling the double-backward pass to reuse the same computational primitives.

The kernel framework also supports a fused graph-convolution mode. For a graph $G = (V, E)$, node-level outputs are computed as

$$z_j = \sum_{(j,k,e)\in\mathcal{N}(j)} \mathrm{TP}(P,\, x_k,\, y_e,\, W_e)\,, \tag{26}$$

where $\mathcal{N}(j)$ denotes the neighbors of node j. The kernel iterates over edges using a sparse-matrix–multiplication-style schedule and accumulates partial results in a local buffer, reducing global memory writes from $O(|E|)$ to $O(|V|)$ and eliminating the need to duplicate node features.

A.6 Mixture of Experts and Sparse Mixture of Experts

The Mixture-of-Experts (MoE) layer replaces the standard feed-forward network by routing each token to a small subset of expert networks. Given n experts $\{E_0, \ldots, E_{n-1}\}$, the MoE output for a token x is

$$y = \sum_{i=0}^{n-1} G(x)_i\, E_i(x). \tag{27}$$

Here, $G(x) \in \mathbb{R}^n$ is the gating distribution.

To maintain efficiency, only the top-K experts are activated. The gate is computed using a linear projection followed by a Top-K softmax:

$$G(x) = \mathrm{Softmax}\big(\mathrm{TopK}(xW_g)\big)\,, \tag{28}$$

where the Top-K operator is defined componentwise as

$$\big(\mathrm{TopK}(\ell)\big)_i = \begin{cases} \ell_i, & \text{if } \ell_i \text{ is among the top-}K \text{ logits,} \\ -\infty, & \text{otherwise.} \end{cases} \tag{29}$$

This sparsity ensures that each token only uses K experts, keeping compute cost fixed as the total number of experts n grows. In Transformer blocks, MoE replaces the FFN sublayer. Models such as Mixtral use $K = 2$ with SwiGLU experts:

$$y = \sum_{i=0}^{n-1} \mathrm{Softmax}\big(\mathrm{Top2}(xW_g)\big)_i\, \mathrm{SwiGLU}_i(x). \tag{30}$$

To prevent the gate from collapsing onto a small subset of experts, MoE architectures add a load balancing loss that encourages tokens to be distributed more evenly across experts. One widely used formulation is

$$\mathcal{L}_{\text{load}} = n \sum_{i=0}^{n-1} \bar{G}_i\, \bar{F}_i, \tag{10}$$

where $\bar{G}_i$ denotes the mean gating probability for expert i and $\bar{F}_i$ is the empirical fraction of tokens routed to that expert.

A.7 Message Passing Module

At the core of the architecture lies the *Equivariant Graph Convolutional Layer* (EGCL) from [22], which operates on the node feature embeddings $\mathbf{h}^l = \{\mathbf{h}_0^l, \ldots, \mathbf{h}_{M-1}^l\}$, the coordinate embeddings $\mathbf{x}^l = \{\mathbf{x}_0^l, \ldots, \mathbf{x}_{M-1}^l\}$, and the edge attributes $E = (a_{ij})$. The EGCL updates both the node features and the coordinates using the following operations:

$$m_{ij} = \phi_e\left(\mathbf{h}_i^l,\, \mathbf{h}_j^l,\, \|\mathbf{x}_i^l - \mathbf{x}_j^l\|^2,\, a_{ij}\right), \tag{31}$$

$$\mathbf{x}_i^{l+1} = \mathbf{x}_i^l + C\sum_{j\neq i}(\mathbf{x}_i^l - \mathbf{x}_j^l)\,\phi_x(m_{ij}), \tag{32}$$

$$m_i = \sum_{j\neq i} m_{ij}, \tag{33}$$

$$\mathbf{h}_i^{l+1} = \phi_h\left(\mathbf{h}_i^l,\, m_i\right). \tag{34}$$

Here, the functions ϕ_e, ϕ_x, and ϕ_h denote learnable neural network modules, and C is a normalization constant (often learned or set to $1/M$). These update rules ensure that both feature and coordinate updates remain fully E(n)-equivariant.

A.8 Positional Encodings

We follow the modular framework of GRAPHGPS [21] to incorporate structural/positional encodings into the inputs of our denoising model. We use graph-based Laplacian positional encodings. If L represents the Laplacian matrix of a given graph $G = (V, E)$, as a symmetric and positive semidefinite (PSD) matrix, L can be decomposed using its eigenvalues and eigenvectors as:

$$L = \sum_i \lambda_i u_i u_i^\top \tag{35}$$

where λ_i denotes the eigenvalues and u_i corresponds to the associated eigenvectors. In a unified framework for graph neural network (GNN) positional encodings, the normalized graph Laplacian is defined as:

$$L = I - D^{-1/2}AD^{-1/2} = U^\top \Lambda U \tag{36}$$

Here, each row of U represents a corresponding eigenvector of the graph, while Λ is a diagonal matrix containing the eigenvalues. Based on this formulation, the positional encoding for a node k can be represented as [7]:

$$X_k^{\text{PE}} = f(U_{k,:}, \Lambda, \Theta, \{\cdot\}) \tag{37}$$

where $U_{k,:}$ denotes the k-th row of U, Λ holds the eigenvalues, Θ represents parameters controlling linear or non-linear transformations applied to these matrices, and $\{\cdot\}$ refers to any additional parameters specific to different encoding approaches.

For the transformer block, which does not use recurrence or convolution, we employ sinusoidal positional encodings to capture token order. These encodings are added to token embeddings and are defined as [27]:

$$\text{PE}(\text{pos}, 2i) = \sin\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right), \quad \text{PE}(\text{pos}, 2i+1) = \cos\left(\frac{\text{pos}}{10000^{2i/d_{\text{model}}}}\right) \tag{38}$$

where pos is the position index and i is the embedding dimension. In our implementation, we construct a positional vector of the same dimension as the input embedding, $|\text{PE}(i)| = |x_i| = d_e$, and add it element-wise:

$$x_i^p = x_i + \text{PE}(i) \tag{39}$$

A.9 State Space Operations

- **Mamba-1: Hardware-aware recurrent scanning.** Naive recurrent SSMs scale as $\mathcal{O}(BTDN)$, while convolutional variants achieve $\mathcal{O}(BTD\log T)$ complexity but suffer from high I/O overhead. In practice, recurrent formulations are often faster for moderate sequence length T and state dimension N due to smaller constant factors, though they are limited by sequential computation and memory usage. Mamba-1 [8] addresses these issues through a hardware-aware parallel scan that fuses discretization, recurrence, and output projection into a single GPU kernel. Intermediate states of shape (B, T, D, N) are kept in on-chip SRAM, reducing memory reads to $\mathcal{O}(BTD + DN)$ and writes to $\mathcal{O}(BTD)$, yielding an $\mathcal{O}(N)$ reduction in I/O cost. For long sequences, scanning is performed in chunks with state carryover, enabling linear $\mathcal{O}(T)$ time complexity and substantially improved throughput compared to convolutional SSMs and quadratic self-attention.
- **Mamba-2: Structured State Decomposition and shared dynamics.** Mamba-2 further improves efficiency via Structured State Decomposition (SSD), which exploits the block structure of semiseparable matrices to combine the linear recurrence of SSMs with their dual quadratic formulation. SSD achieves linear FLOP complexity in sequence length T while reformulating computations as matrix multiplications, significantly accelerating training. Compared to dense attention with $\mathcal{O}(T^2N)$ training cost and $\mathcal{O}(T^2)$ memory usage, SSD reduces training complexity to $\mathcal{O}(TN^2)$ and memory usage to $\mathcal{O}(TN)$. Additionally, Mamba-2 constrains the state transition matrix to a scalar-times-identity form, inducing shared recurrence dynamics across state dimensions and channels. This parameter sharing preserves expressivity while enabling efficient matrix-multiplication-based implementations, supporting larger state sizes and faster training and inference.
- **Hydra: The Double-Headed Mamba**

  Hydra achieves efficient bidirectional sequence modeling by parameterizing the sequence mixer with a *quasiseparable* matrix, which admits sub-quadratic matrix–vector multiplication. Given an input sequence $X \in \mathbb{R}^{L\times C}$, the output is computed as

  $$Y = MX, \tag{40}$$

  where the mixer matrix $M \in \mathbb{R}^{L\times L}$ satisfies the quasiseparable structure

  $$m_{ij} = \begin{cases} \vec{c}_i^{\top}\vec{A}_{i:j}\vec{b}_j, & i > j, \\ \delta_i, & i = j, \\ c_j^{\top} A_{j:i} b_i, & i < j. \end{cases} \tag{41}$$

  This structure constrains the rank of any off-diagonal submatrix to be at most N, enabling $\mathcal{O}(L)$ sequence mixing while supporting full bidirectional context. In contrast to dense attention mechanisms with $\mathcal{O}(L^2)$ complexity, Hydra preserves the linear-time scalability of selective state space models without sacrificing bidirectionality. Moreover, the decoupled diagonal parameterization avoids the expressivity constraints imposed by addition-based bidirectional SSMs, yielding an efficient and flexible architecture for long-context modeling.

[5]

A.10 Node Features Representation

Spherical EGNNs utilize tensors for node and occasionally edge features. While we focus on node features, the principles also apply to edge feature tensors. We represent the (hidden) features at node a as a list of geometric tensors with various l-values, ranging from 0 to l_{max}:

$$\vec{V}_a^{(0:l_{\max})} := \bigoplus_{l=0}^{l_{\max}} \vec{V}_a^{(l)} = \begin{bmatrix} \vec{V}_a^{(0)} \\ \vdots \\ \vec{V}_a^{(l_{\max})} \end{bmatrix} \tag{42}$$

[5]In graph modeling, this translates to data-dependent node selection: relevant nodes are filtered at each recurrence step, effectively “attending” only to selected context.

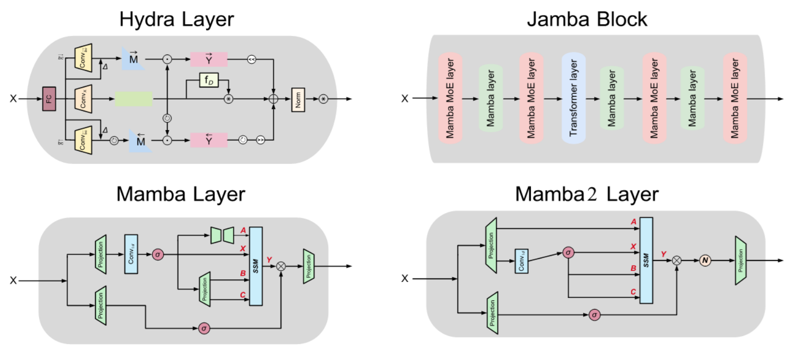

Figure 3: Illustration of the different neural network layers used in our framework: (top-left) Hydra Layer, (top-right) Jamba Block, (bottom-left) Mamba Layer, and (bottom-right) Mamba² Layer. Each diagram shows the flow of input features through projections, convolutions, and structured modules, highlighting the internal computations and skip connections..

The representation of features in equivariant networks involves three main axes. First, a *channel axis*, as multiple tensor types l may exist at each node; for simplicity, we assume an equal number of channels for all l-types. Second, a *tensor axis*, denoted by the $(0 : l_{\text{max}})$ superscript, which corresponds to a concatenated list of spherical tensors of different types. Finally, a *tensor-component axis*, which for each tensor of type l consists of $2l + 1$ components.

A.11 Generation Quality Metrics

The core metrics used to evaluate generative models are validity, uniqueness, and diversity.

Validity quantifies the percentage of chemically sound structures within a generated set of molecules G. This is typically determined using molecular parsing tools like RDKit. Its calculation is defined as:

$$\text{Validity}(G) = \frac{|\{m \in G : \text{is\_valid}(m)\}|}{|G|} \times 100 \tag{43}$$

where is_valid(m) is a function that verifies the chemical validity of a molecule m.

Novelty measures the fraction of generated molecules that are not present in the training dataset D_{train}, indicating the model's ability to generate new, unseen molecules. For a generated set G, novelty is defined as:

$$\text{Novelty}(G) = \frac{|\{m \in G : m \notin D_{\text{train}}\}|}{|G|} \times 100 \tag{44}$$

where m represents a generated molecule and D_{train} is the set of molecules seen during training. A higher novelty score indicates greater generation of previously unseen molecules.

Diversity assesses how well the generated set covers the chemical space. This is often quantified by the average pairwise Tanimoto distance between molecules' fingerprint representations. For a generated set G and a fingerprint function $f(\cdot)$, diversity is given by:

$$\text{Diversity}(G) = \frac{2}{|G|(|G|-1)} \sum_{i=1}^{|G|} \sum_{j=i+1}^{|G|} \text{Tanimoto}(f(m_i), f(m_j)) \tag{45}$$

where $m_i, m_j \in G$ are molecules from the generated set, and Tanimoto$(\cdot, \cdot)$ calculates the Tanimoto similarity between their fingerprints.

A.12 Quantitative Estimate of Druglikeness (QED)

The Quantitative Estimate of Druglikeness (QED) is a widely adopted metric designed to quantify a molecule's druglikeness by combining eight key physicochemical properties into a single desirability score.

The desirability function $d(p_i)$ for each individual property p_i is empirically modeled using an asymmetric double sigmoidal function:

$$d(p_i) = \frac{1}{1 + \exp[-a_i(p_i - b_i)]} \times \frac{1}{1 + \exp[-c_i(p_i - d_i)]} \tag{46}$$

where $a_i, b_i, c_i,$ and d_i are parameters fitted specifically for property p_i.

The overall QED value is then calculated by taking a weighted geometric mean of these individual desirability scores:

$$\text{QED} = \left(\prod_{i=1}^{n} d(p_i)^{w_i} \right)^{1/\sum_{i=1}^{n} w_i} \tag{47}$$

where w_i represents the weight assigned to property p_i, reflecting its contribution to druglikeness. QED scores range from 0 to 1, with higher values indicating greater druglikeness. This metric offers a continuous and nuanced assessment, proving more effective than binary, rule-based methods like Lipinski's Rule of Five, particularly in identifying drug-like compounds that may not conform to traditional filters. It also facilitates the prioritization of promising drug candidates within specific chemical spaces.

[t] E(3)-equivariant tensor product score network implemented using `e3nn`, with Clebsch–Gordan tensor product kernels accelerated via `OpenEquivariance` (OEQ)

```python
# ============================================================
# Equivariant Tensor Product Score Model
# ============================================================

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch_scatter import scatter
from torch_cluster import radius_graph

from e3nn import o3
from e3nn.nn import BatchNorm


class TensorProductConvLayer(nn.Module):
    """
    Equivariant message passing layer based on explicit tensor
    products between node features and spherical harmonics.
    """
    def __init__(
        self,
        in_irreps,
        sh_irreps,
        out_irreps,
        n_edge_features,
        instructions=None,
        residual=True,
        batch_norm=True,
    ):
        super().__init__()

        self.in_irreps = o3.Irreps(in_irreps)
        self.sh_irreps = o3.Irreps(sh_irreps)
        self.out_irreps = o3.Irreps(out_irreps)
        self.residual = residual

        if instructions is None:
            instructions = [
                (i, j, k, "uvu", True)
                for i, _ in enumerate(self.in_irreps)
                for j, _ in enumerate(self.sh_irreps)
                for k, _ in enumerate(self.out_irreps)
                if o3.selection_rule(
                    self.in_irreps[i].ir,
                    self.sh_irreps[j].ir,
                    self.out_irreps[k].ir,
                )
            ]

        self.tp = o3.TensorProduct(
            self.in_irreps,
            self.sh_irreps,
            self.out_irreps,
            instructions=instructions,
            shared_weights=False,
            internal_weights=False,
        )

        self.fc = nn.Sequential(
            nn.Linear(n_edge_features, n_edge_features),
            nn.ReLU(),
            nn.Linear(n_edge_features, self.tp.weight_numel),
        )

        self.batch_norm = BatchNorm(self.out_irreps) if batch_norm else None

    def forward(self, node_attr, edge_index, edge_attr, edge_sh, reduce="mean"):
        src, dst = edge_index

        W = self.fc(edge_attr)
        msg = self.tp(node_attr[dst], edge_sh, W)
        out = scatter(msg, src, dim=0, dim_size=node_attr.size(0), reduce=reduce)

        if self.residual:
            if out.size(-1) > node_attr.size(-1):
                node_attr = F.pad(node_attr, (0, out.size(-1) - node_attr.size(-1)))
            out = out + node_attr[:, : out.size(-1)]

        if self.batch_norm is not None:
            out = self.batch_norm(out)

        return out


class TensorProductScoreModel(nn.Module):
    """
    Multi-layer E(3)-equivariant score network with tensor
    product convolutions and radius-based graph construction.
    """
    def __init__(
        self,
        channels=64,
        in_node_features=64,
        in_edge_features=80,
        sh_lmax=2,
        ns=32,
        nv=8,
        num_conv_layers=4,
        max_radius=5.0,
        radius_embed_dim=50,
        batch_norm=True,
        residual=True,
    ):
        super().__init__()

        self.ns = ns
        self.max_radius = max_radius
        self.in_edge_features = in_edge_features

        self.sh_irreps = o3.Irreps.spherical_harmonics(lmax=sh_lmax)

        self.node_embedding = nn.Sequential(
            nn.Linear(in_node_features, ns),
            nn.ReLU(),
            nn.Linear(ns, ns),
        )

        self.distance_expansion = GaussianSmearing(
            start=0.0, stop=max_radius, num_gaussians=radius_embed_dim
        )

        self.edge_embedding = nn.Sequential(
            nn.Linear(in_edge_features + radius_embed_dim, ns),
            nn.ReLU(),
            nn.Linear(ns, ns),
        )

        irrep_seq = [
            f"{ns}x0e",
            f"{ns}x0e + {nv}x1o + {nv}x2e",
            f"{ns}x0e + {nv}x1o + {nv}x2e + {nv}x1e + {nv}x2o",
            f"{ns}x0e + {nv}x1o + {nv}x2e + {nv}x1e + {nv}x2o + {ns}
    x0o",
        ]

        self.layers = nn.ModuleList()
        for i in range(num_conv_layers):
            self.layers.append(
                TensorProductConvLayer(
                    irrep_seq[i],
                    self.sh_irreps,
                    irrep_seq[min(i + 1, len(irrep_seq) - 1)],
                    n_edge_features=3 * ns,
                    residual=residual,
                    batch_norm=batch_norm,
                )
            )

        self.final_irreps = o3.Irreps(irrep_seq[-1])
        self.output = nn.Sequential(
            nn.Linear(self.final_irreps.dim, channels),
            nn.SiLU(),
            nn.Linear(channels, channels),
        )

    def forward(self, data):
        node_attr, edge_index, edge_attr, edge_sh = self.build_graph(
    data)
        src, dst = edge_index

        node_attr = self.node_embedding(node_attr)

        dist_emb = self.distance_expansion(edge_attr[:, 0])
        edge_attr = self.edge_embedding(torch.cat([edge_attr, dist_emb], dim=-1))

        for layer in self.layers:
            edge_feat = torch.cat(
                [edge_attr, node_attr[src, : self.ns], node_attr[dst, : self.ns]], dim=-1
            )
            node_attr = layer(node_attr, edge_index, edge_feat, edge_sh)

        return self.output(node_attr)

    def build_graph(self, data):
        radius_edges = radius_graph(data.pos, r=self.max_radius, batch=data.batch)
        edge_index = torch.cat([data.edge_index, radius_edges], dim=1)

        edge_attr = data.edge_attr
        if edge_attr is None:
            edge_attr = torch.zeros(
                data.edge_index.size(1), self.in_edge_features, device=data.pos.device
            )

        pad = torch.zeros(
            radius_edges.size(1), self.in_edge_features, device=edge_attr.device
        )
        edge_attr = torch.cat([edge_attr, pad], dim=0)

        src, dst = edge_index
        edge_vec = data.pos[dst] - data.pos[src]

        edge_sh = o3.spherical_harmonics(
            self.sh_irreps, edge_vec, normalize=True, normalization="component"
        )

        return data.x, edge_index, edge_attr, edge_sh
```