GRAPH MACHINE: PRETRAINING WITH SPARSE DYNAMIC POINTERS

Anonymous authors
Paper under double-blind review

ABSTRACT

We introduce the Graph Machine (GM), an architecture that maintains an $O(n)$-sized state and accesses it through sparse, dynamic routing. Unlike methods with fixed-size states or sparse but static routing, GM preserves $O(n)$ complexity in its sparse layers without restricting the potentially accessible state size to $O(1)$. Instead, GM uses pointer-like edges with discrete indices and differentiable weights, updated through a referral mechanism resembling pointer chasing. We replace 75% of Qwen3-0.6B's dense Transformer layers with GM sparse layers and train the resulting models end to end from scratch on 15.7B tokens using a standard pretraining recipe. With only 2 of 4,096 tokens retrieved per KV head in each sparse layer, loss degrades only slightly; with 4, the best model marginally improves loss.

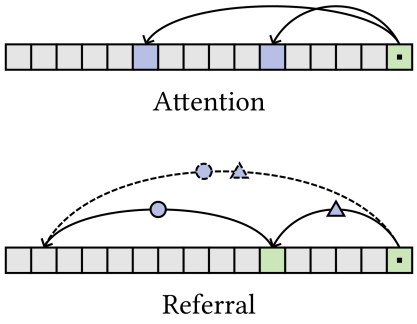


Figure 1: Attention and referral viewed as pointer operations.

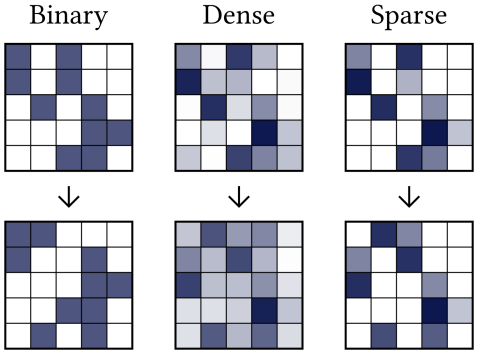


Figure 2: Different referral approaches viewed as adjacency matrix transformations ($\ell = 2$).

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

1 INTRODUCTION

We classify sequence modeling approaches by the sizes of their state, access, and dynamic addresses.

Recurrent models such as RNNs (Elman, 1990) and SSMs (Gu et al., 2022) maintain a $\Theta(1)$-sized state and must therefore compress their history.

Transformers (Vaswani et al., 2017) avoid this compression by retaining a $\Theta(n)$-sized state in their keys and values. However, vanilla attention allows each step (token) to access the entire state, leading to $\Theta(n^2)$ complexity.

Methods such as sliding-window attention (Beltagy et al., 2020) reduce this access to $\Theta(1)$. Yet the accessible positions are fixed independently of the new step's content: all but $\Theta(1)$ positions can be excluded in advance. Thus, their access is sparse but static.

A simple information-theoretic argument shows that dynamic, $\Theta(1)$-sized access to a $\Theta(n)$-sized state requires each new token to provide $\Theta(\log n)$ bits that can be used directly to address—identify and retrieve—a constant-sized portion of the state. Together, these properties define a fourth category.

Table 1: A sequence-model taxonomy by state size, per-step access size, and per-step dynamic address bits

Type	Examples	State	Access	Dynamic address bits
1	RNN, SSM	$\Theta(1)$	$\Theta(1)$	$\setminus$
2	Vanilla transformer	$\Theta(n)$	$\Theta(n)$	$\setminus$
3	Sliding-window transformer	$\Theta(n)$	$\Theta(1)$	$O(1)$
4	GM	$\Theta(n)$	$\Theta(1)$	$\Theta(\log n)$

Although each dynamic address theoretically contains $\Theta(\log n)$ bits, at practical sequence lengths it fits in a fixed-width integer such as `int64`, making its storage cost effectively constant. The Graph Machine therefore instantiates this fourth class: it maintains $\Theta(n)$ state entries, accesses $\Theta(1)$ entries per step, and uses $\Theta(\log n)$ bits of dynamic addressing per step. Thus, GM stores both floating-point hidden states, which we call *node features*, and integer edge targets which we call *edge indices*. At each new step, stored edge indices are used to retrieve $\Theta(1)$ positions for information aggregation by attention (Figure 1).

To update the stored edges across layers, we use a *referral* (Hou, 2026) mechanism that constructs the next neighborhoods from the current ℓ-hop neighborhoods. Referral can be understood as neighbors recursively referring to their neighbors for ℓ rounds, as composing ℓ-hop edges into one-hop edges, or as taking an adjacency matrix A and producing A^ℓ (Figure 1). If each node retains s neighbors, referral can produce up to s^ℓ candidate paths. Keeping the neighborhood size fixed at s therefore requires a hard selection that reduces these candidates to s neighbors. To learn the selection, a probabilistic policy is needed in the form of

$$A'_{\text{binary}} \sim \pi_\theta\big(\cdot \mid A^\ell_{\text{binary}}\big)\,,$$

which would introduce credit assignment difficulties.

Instead, we soften the edges rather than the policy, reformulating the top-s selection into a top-s approximation over soft weights. We start with a soft and sparse adjacency matrix, take its ℓ-th power, and then sparsify the result to form the new adjacency matrix (Figure 2). I.e., we approximate

$$A'_{\text{dense}} = A^\ell_{\text{dense}}$$

with

$$A'_{\text{sparse}} = \text{Sparsify}_s\big(A^\ell_{\text{sparse}}\big)\,.$$

The resulting edge weights then contribute to the attention weights alongside the usual query–key factors in a product-of-experts (Hinton, 2002) manner.

Intuitively, referral seeks to provide useful relational representations to attention through sparsity-constrained intermediaries. Rather than learning to shift the support directly, it learns from redistributing weight mass within the support. From another perspective, each edge is now jointly represented

by integer edge indices and floating-point *edge weights*, the latter carrying gradients that discrete indices cannot.

Viewed globally, the Graph Machine maintains and updates a graph with soft directed edges. Sparse attention passes content along edges to update nodes, while referral passes addresses along edges to form new ones. In this paper, we hybridize GM and Transformer layers and train the resulting Graph Language Machines (GLMs) under a standard language-model pretraining setup. Throughout the paper, we move between the vocabularies of adjacency matrices, edges, neighbors, addresses, pointers, and indices as appropriate.

2 Architecture

We first provide an overview of the GM architecture, then introduce two common operations, and finally describe its two new sparse edge submodules.

2.1 Overview

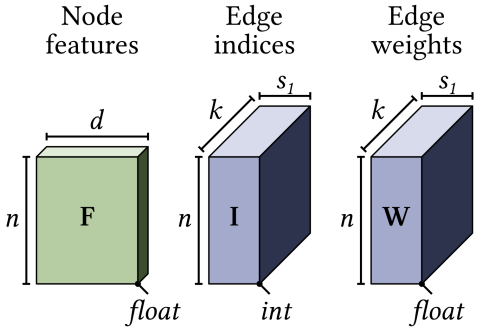


Figure 3: GM states.

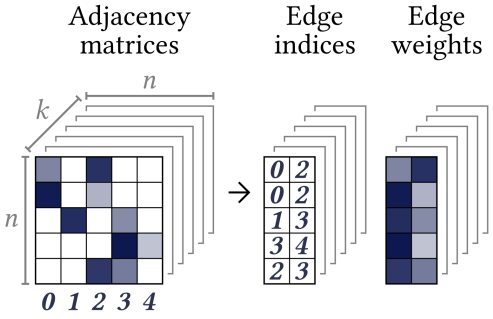


Figure 4: Adjacency matrices → edge states.

States. GM maintains three state tensors (Figure 3). Alongside the $n \times d$ node features, it maintains edge indices and edge weights, both of shape $n \times k \times s_1$. We obtain this representation by generalizing the adjacency matrix to k copies and representing them in an s_1-sparse coordinate format, giving each node k edges and each edge s_1 member positions (Figure 4). Across the s_1 member positions, the edge-index tensor specifies target nodes, while the edge-weight tensor provides corresponding nonnegative weights that sum to one. We call the edge states passed between layers stored edges, distinguishing them from the mixed edges used directly by referral and attention within the submodules.

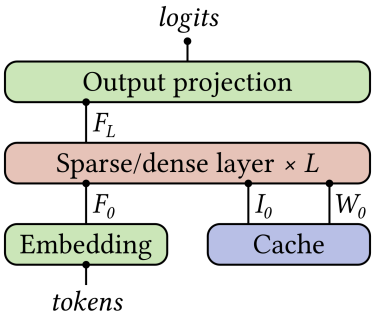


Figure 5: GM layers.

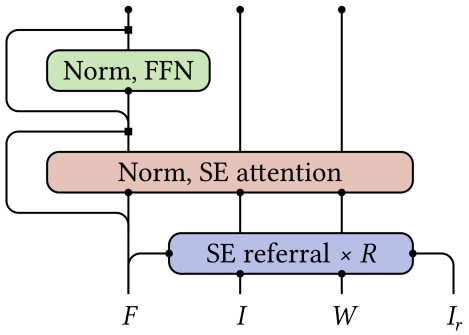


Figure 6: A sparse GM layer.

Layers. Token embeddings initialize the node features in our GLMs. Each token's initial input edges point to itself and its $k-1$ nearest preceding tokens, clamped to the first token and cached for reuse. Each input edge places all its weight on its single target and fills its remaining member positions with zero indices and weights. The dense layers are standard Transformer layers. Each sparse layer except the first applies r edge-referral submodules, each performing two-hop referral, followed by an edge-attention submodule and an MLP. Referral updates the edges, while attention and the MLP update the node features. Edges remain causal because they are derived from previous causal edges or masked attention. As usual, both dense and sparse layers use pre-normalization and residual connections for node features. A final output projection produces prediction logits from node features.

2.2 Operations

Sparsify. A conceptual weighted average of adjacency-matrix rows requires several additional steps in a sparse-coordinate implementation. Duplicate indices must be coalesced, the highest-weight entries selected, and their weights renormalized to achieve the target sparsity. For example, equally averaging $([0,3],[0.4,0.6])$ and $([2,3],[0.8,0.2])$ first produces

$$([0,3,2,3],[0.2,0.3,0.4,0.1]).$$

Coalescing the two entries for index 3, dropping the lowest-weight entry, and renormalizing gives

$$([2,3],[0.5,0.5]).$$

Since we use hard top-s selection, only the retained weights receive gradients. Sparsification occurs when stored edges are mixed before referral and attention, and when the resulting edges are reconciled after referral.

Mix. We mix the k stored edges of each node before using them for referral or attention, much as a pointwise convolution mixes channels in a CNN. For each output edge, mixing logits are projected from the node features and normalized over the input edges with a softmax, producing a mixing matrix M. The resulting coefficients are used to compute a weighted average of the input edges. Weighted averaging $\widetilde{A} = MA$ alone would keep new edges approximately within the convex hull of earlier edges. We therefore introduce nonlinearity through temperature scaling both before and after the average. Separate input- and output-edge temperatures are parameterized as the softplus or exponential of projections from the node features; each acts as an exponent on the corresponding edge weights, which are then renormalized. I.e., for an edge-weight vector a, the temperature-scaling is

$$\mathrm{T}_\tau(a) = \frac{a^\tau}{\sum_j a_j^\tau}, \quad \tau > 0,$$

and mixing conceptually follows

$$\widetilde{A} = \mathrm{T}_{\boldsymbol{\tau}_{\mathrm{out}}}\left(M\ \mathrm{T}_{\boldsymbol{\tau}_{\mathrm{in}}}(A)\right).$$

In our sparse-coordinate implementation of referral, output-temperature scaling is applied to the uncoalesced contributions; sparsification then coalesces duplicate indices and reduces the resulting edge to the desired sparsity.

2.3 Submodules

Sparse edge referral (SER). During sparse edge referral, rather than using a single adjacency matrix A, we take two matrices A_1 and A_2 and perform two-hop referral via

$$A' = \mathrm{Sparsify}_{s_1}(A_1 A_2).$$

At a path level, referral composes a first edge $e_1 = (n_1 \xrightarrow{w_1} n_2)$ with a second edge $e_2 = (n_2 \xrightarrow{w_2} n_3)$, omitting sparsification:

$$e_1 \circ e_2 = \left(n_1 \xrightarrow{w_1 w_2} n_3\right).$$

Concretely, to construct k new edges, we first mix the k stored edges into $2k$ channels, forming k pairs that represent the two legs of the new edges. Within each pair, e_1 and e_2 are sparsified to s_2 and

s_3, respectively. The e_1 indices are then used to retrieve the corresponding e_2 indices and weights. This produces up to $s_2 s_3$ members per edge before sparsification to s_1. Optionally, refresh edges augment the stored edges before mixing. These refresh edges can come from the initial input edges or from the top-k indices and normalized weights of the most recent dense-attention layer.

Sparse edge attention (SEA). During sparse edge attention, we mix the k stored edges into g channels, one per KV head, and sparsify each to s_4. The resulting edge indices specify which positions provide the keys and values. We compute the usual scaled query–key scores over these positions and combine them with the mixed edge weights as a product of experts. Specifically, we call the query–key scores *node factors* and scale them using per-head temperatures parameterized analogously to the output-edge temperatures in mixing. We call the logarithms of the mixed edge weights *edge factors*. Adding the node and edge factors and applying softmax produces the final attention weights. Formally, for one query head and its associated KV head, let $\mathcal{N}(i)$ denote the positions retrieved for node i, and let w_{ij} denote their mixed edge weights. The final attention weights are

$$a_{ij} = \mathrm{softmax}_{j \in \mathcal{N}(i)} \left(\tau \frac{q_i^\top k_j}{\sqrt{d_h}} + \log w_{ij} \right) \propto w_{ij} \exp \left(\tau \frac{q_i^\top k_j}{\sqrt{d_h}} \right).$$

Here, addition in logit space corresponds to a product of experts in probability space. Value aggregation and output projection then follow as in standard attention. Optionally, we use the retrieved indices and final attention weights to replace a subset of the stored edges. This realigns the pointer distribution after approximate referral using the feature-based evidence obtained during attention.

3 Results

We use Qwen3-0.6B (Yang et al., 2025a) as our baseline and as a representative modern dense LLM. For a controlled comparison, all models are trained from scratch using the same backbone and training hyperparameters, codebase, and random seed. We use a conventional LLM training recipe based on established practice, without tuning it to our specific conditions.

For GM, we use a sparse-to-dense layer ratio of 3:1, scheduled as $[\mathrm{S}, \mathrm{S}, \mathrm{D}, \mathrm{S}] \times 7$. We use 16–32 edges, 0–6 referral steps, sparsity budgets of $(2, 4, 4, 2)$ or $(4, 4, 4, 4)$, 8 input-refresh edges, and 0–8 dense-refresh edges, with realignment enabled. Temperatures are parameterized by the horizontally shifted softplus $f_\tau(x) = \mathrm{softplus}(x + \log(e - 1))$, such that $f_\tau(0) = 1$. RoPE (Su et al., 2024) is omitted from SEA.

We name each GLM by its sparsity budget, number of stored edges, number of referral steps, and whether it uses eight dense-refresh edges. Theia (IO Interactive, 2026) models use a sparsity budget of $(2, 4, 4, 2)$, whereas Hyperion models use $(4, 4, 4, 4)$. Because s_4 determines attention sparsity, Theia and Hyperion retrieve 2 and 4 positions per KV head during SEA, respectively. For example, Hyperion-K16-R3-S uses a $(4, 4, 4, 4)$ sparsity budget, 16 edges, three referral steps, and eight dense-refresh edges. In the tables, we group conditions by sparsity and then order them by the number of referral steps.

We train on a randomly sampled, overprovisioned subset of FineWeb-Edu (Penedo et al., 2024), which is processed for less than one epoch. Documents are packed into 4,096-token sequences and padded as needed. A padding token is placed at the start of each sequence to absorb underflowing edge indices. Across a random sample of 100K documents, document length has a mean of 1,035 and a standard deviation of 1,909 Qwen3 tokens. A sequence length of 4,096 therefore does not imply a comparable effective context length within each document.

Each training run uses a single H100 SXM and a reference implementation written primarily in generic PyTorch (Paszke et al., 2019), with a custom Triton (Tillet et al., 2019) kernel for sparsification. Under this reference implementation, GLMs are generally several times slower than Qwen3 on this hardware. In preliminary experiments under the same experimental settings, new flash kernels help GM models achieve around 80% of Qwen3's training throughput, with a throughput advantage appearing with longer sequence lengths.

With the backbone dimensions held fixed, the referral and temperature parameters increase the parameter counts of referral-equipped GLMs by 9%–23%. Nevertheless, these models reduce total estimated training compute by 5%–15% and referral-plus-attention compute by 10%–30% relative to

Table 2: Parameter counts and estimated training FLOPs. Only multiplications and accumulations are counted, each as one FLOP; normalization, softmax, activations, RoPE, and sparsification are excluded. SEA KV access is measured over a 4,096-token sequence relative to dense causal attention.

Model	Parameters	Total compute	Ref + att compute	75% att KV access
Qwen3	596M	78.4 EFLOPs	38.8 EFLOPs	100.000%
Theia-K24	601M	62.3 EFLOPs	22.7 EFLOPs	0.098%
Theia-K32-R2	711M	72.7 EFLOPs	33.0 EFLOPs	0.098%
Theia-K24-R3	700M	71.6 EFLOPs	32.0 EFLOPs	0.098%
Theia-K24-R4	733M	74.7 EFLOPs	35.1 EFLOPs	0.098%
Theia-K16-R4-S	686M	73.1 EFLOPs	33.5 EFLOPs	0.098%
Theia-K16-R6	700M	71.6 EFLOPs	32.0 EFLOPs	0.098%
Hyperion-K32-R1	657M	67.5 EFLOPs	27.9 EFLOPs	0.195%
Hyperion-K24-R3	700M	71.6 EFLOPs	32.0 EFLOPs	0.195%
Hyperion-K16-R3	650M	66.9 EFLOPs	27.3 EFLOPs	0.195%
Hyperion-K16-R3-S	664M	71.1 EFLOPs	31.4 EFLOPs	0.195%
Hyperion-K16-R4	666M	68.5 EFLOPs	28.8 EFLOPs	0.195%

Qwen3. Most of the additional parameters belong to referral projections, which support relatively inexpensive operations. Over a length-n sequence, dense causal attention accesses an average of $(n+1)/2$ KV positions per query. At $n = 4096$, retrieving at most two or four positions therefore corresponds to 0.098% or 0.195% of dense causal KV access, respectively.

Table 3: Test loss throughout training. Qwen3 entries give absolute loss, whereas GLM entries give the difference from Qwen3. Highlighting marks the model with the lowest final loss in each sparsity category.

Model	Test loss / Δ test loss			
	@ 25%	@ 50%	@ 75%	@ 100%
Qwen3	2.869	2.744	2.660	2.587
Theia-K24	+0.041	+0.039	+0.040	+0.040
Theia-K32-R2	+0.016	+0.017	+0.018	+0.020
Theia-K24-R3	+0.012	+0.012	+0.013	+0.014
Theia-K24-R4	+0.013	+0.013	+0.014	+0.015
Theia-K16-R4-S	+0.010	+0.013	+0.014	+0.016
Theia-K16-R6	+0.013	+0.014	+0.014	+0.015
Hyperion-K32-R1	+0.008	+0.008	+0.009	+0.009
Hyperion-K24-R3	-0.002	-0.003	-0.003	-0.002
Hyperion-K16-R3	0.000	0.000	-0.001	0.000
Hyperion-K16-R3-S	-0.005	-0.005	-0.005	-0.003
Hyperion-K16-R4	-0.004	-0.003	-0.003	-0.002

All models are evaluated on the same fixed held-out random subset of FineWeb-Edu. Final test losses are around 2.60, consistent with broader LLM pretraining experience at this scale.

Increasing the number of edges beyond the 16 attention heads provides a moderate improvement in the matched Hyperion comparison: K24-R3 outperforms K16-R3 by 0.003 at the end of training. This improvement comes at additional parameter and compute cost because full cross-edge mixing scales as k^2.

Referral clearly improves performance from zero to a few steps: Theia-K24-R3 improves over the non-referral Theia-K24 baseline by 0.026 at 100%. Additional referral steps can also help under some conditions, with Hyperion-K16-R4 improving over Hyperion-K16-R3 by 0.003. However, this trend does not hold in every setting. Under the sparser Theia budget with more edges than heads, Theia-K24-R4 performs 0.001 worse than Theia-K24-R3.

Preliminary experiments suggest benefits from realignment and input refresh. We observe a similar benefit from dense refresh: Hyperion-K16-R3-S improves over Hyperion-K16-R3 by 0.004 at 100%.

Within the tested configurations, neither parameter count nor estimated compute is a strong predictor of performance. Hyperion-K16-R3-S achieves the best loss despite being among the less expensive GLMs, with 11% more parameters and 19% less referral-plus-attention compute than Qwen3.

Overall, the best Theia model increases final loss by approximately 0.014, while the best Hyperion model reduces it by approximately 0.003.

Table 4: Downstream evaluation accuracy (%; higher is better) on ten tasks. Scores are rounded to two decimal places. Bold marks the highest score in each column, including ties.

Model	PIQA	HellaSwag	COPA	SocialIQA	ARC-Easy
Qwen3	68.93	36.83	**72.00**	39.25	**64.60**
Theia-K24	68.72	35.76	69.00	38.79	63.47
Theia-K32-R2	68.99	36.16	69.00	38.79	62.08
Theia-K24-R3	69.48	36.76	70.00	38.28	62.37
Theia-K24-R4	68.28	36.38	**72.00**	38.49	62.37
Theia-K16-R4-S	69.37	36.48	69.00	**40.84**	63.43
Theia-K16-R6	69.42	36.34	67.00	38.33	63.85
Hyperion-K32-R1	**69.70**	36.69	71.00	39.05	63.09
Hyperion-K24-R3	69.64	36.89	69.00	39.82	63.34
Hyperion-K16-R3	69.42	36.71	**72.00**	39.87	63.05
Hyperion-K16-R3-S	68.39	**37.01**	**72.00**	40.02	62.58
Hyperion-K16-R4	68.82	36.82	68.00	39.05	63.01

Model	ARC-Challenge	OpenBookQA	SciQ	RACE-high	LAMBADA
Qwen3	27.65	24.20	84.90	**34.26**	**38.19**
Theia-K24	28.16	24.20	84.50	31.29	34.64
Theia-K32-R2	28.16	25.60	83.00	30.81	36.62
Theia-K24-R3	27.82	24.40	81.90	32.54	36.02
Theia-K24-R4	27.39	24.60	84.00	32.15	35.03
Theia-K16-R4-S	28.24	25.00	85.10	31.77	36.91
Theia-K16-R6	29.01	24.60	84.60	32.06	37.28
Hyperion-K32-R1	27.47	23.20	83.20	32.15	36.44
Hyperion-K24-R3	29.18	**25.80**	82.70	32.73	37.57
Hyperion-K16-R3	28.67	24.40	84.10	31.48	37.20
Hyperion-K16-R3-S	28.33	24.40	**85.80**	31.48	37.47
Hyperion-K16-R4	**29.69**	23.20	84.70	31.77	37.49

We further evaluate the models on ten downstream tasks (Table 4). Evaluations use the LM Evaluation Harness in the zero-shot setting without chat templates. We report accuracy, selecting multiple-choice answers by unnormalized continuation log-likelihood and using greedy exact match for LAMBADA. GM's effect on downstream performance is uneven. Interestingly, all GM models underperform Qwen3 on ARC-Easy, whereas most improve on ARC-Challenge. Several GM models also improve on PIQA, SocialIQA, OpenBookQA, and SciQ, while all underperform on context-heavy benchmarks such as RACE-high and LAMBADA.

These results show that most Transformer layers can be replaced by sparse layers retrieving only 2 or 4 of 4,096 positions per KV head without materially sacrificing overall quality, while meaningfully reducing estimated compute.

4 Related work

This work builds directly on the original GM work (Hou, 2026); we refer to the two papers as GM-1 and GM-2.

1. GM-1 uses a bespoke Sudoku benchmark, whereas GM-2 adopts a standard language-model pretraining setting, making its results easier to contextualize.
2. GM-1 primarily studies the inductive bias introduced by edges, leaving computational practicality to future work. Its dense edge representation incurs cubic time and quadratic space complexity, limiting experiments to a few hundred nodes. GM-2 instead uses sparse edge representations and operations. Its edge indices and weights form a sparse coordinate representation of GM-1's edge addresses, while SER and SEA are sparse counterparts of GM-1's referral and attention operations.
3. GM-2 simplifies the state by unifying node and edge features, removing the need for carefully engineered interactions among separate representations.
4. GM-2 introduces refresh and realignment, allowing referral to reuse initial or dense-attention-derived edges and sparse attention to update stored edges.

GM is related to efficient sequence models and their hybrids (Gu & Dao, 2024; Yang et al., 2025b; Lieber et al., 2024; Kimi Team, 2025). GM is also situated within the sparse-attention literature (Child et al., 2019; Beltagy et al., 2020; Roy et al., 2021; Yuan et al., 2025; DeepSeek-AI, 2025).

5 Limitations and conclusion

This work demonstrates GM using a simple architecture and preliminary kernels, with experiments limited to small model and training scales and evaluation restricted to pretraining loss and ten downstream tasks. Further architectural and implementation improvements, broader evaluation across scales and setups, and deeper investigation of GM's inductive biases in language modeling remain future work. Its explicit relational states and operations also offer opportunities for mechanistic interpretation.

In conclusion, although some amount of global dense attention likely remains valuable under current language-modeling settings, our results show—at least at our scale, under our setup, and by our measure—that it is nonetheless highly trimmable. If part of attention's role is fundamentally to pass and resolve addresses, then we can make the core machinery—logarithmic-sized identification bits and direct retrieval through them—primitives of the architecture. GM now carries and transmits addresses via indices rather than features, and resolves them through indexed gathering rather than dense scoring.

This creates new degrees of freedom for architectural design. On the efficiency side, dense computation can be traded off against sparse memory operations. On the inductive-bias side, architectures can trade off relational traversal against global search, and address passing against content passing. Finding the optimal balance will require further exploration.

AI use statement

In this work, LLMs mainly assisted with implementation, while all major ideas were contributed by the human authors. The reference implementation was written entirely by the authors, with limited LLM consultation and no direct code access during development. LLMs subsequently audited the code and prepared documentation for the reproduction-ready version. The flash kernels were developed semi-autonomously by LLM agents and tested against the reference implementation.

Additionally, LLMs assisted in varying degrees with brainstorming, information searches, literature surveys, experimental design, methodological feedback, interpretation of results, and polishing the paper's title, language, mathematical presentation, and formatting. Other tasks in the ICLR 2027 AI use disclosure lists were either not applicable or performed without generative AI. The authors take responsibility for the final content, including text, claims, and artifacts produced with generative AI assistance.

REPRODUCIBILITY STATEMENT

We provide an anonymized copy of the code repository as supplementary material, including all code and files for model implementation, configurations, data preprocessing, training, evaluation, and flash kernels. Its README documents the detailed reproduction instructions.

REFERENCES

Iz Beltagy, Matthew E. Peters, and Arman Cohan. Longformer: The long-document transformer, 2020.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers, 2019.

DeepSeek-AI. DeepSeek-V3.2: Pushing the frontier of open large language models, 2025.

Jeffrey L. Elman. Finding structure in time. *Cognitive Science*, 14(2):179–211, 1990. doi: 10.1207/s15516709cog1402_1.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *Proceedings of the First Conference on Language Modeling*, 2024.

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. In *International Conference on Learning Representations*, 2022.

Geoffrey E. Hinton. Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800, 2002. doi: 10.1162/089976602760128018.

Lintai Hou. Graph machine: Exploring edge mechanisms as an inductive bias, 2026.

IO Interactive. 007 first light. Video game, May 2026.

Kimi Team. Kimi Linear: An expressive, efficient attention architecture, 2025.

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, Avshalom Manevich, Nir Ratner, Noam Rozen, Erez Schwartz, Mor Zusman, and Yoav Shoham. Jamba: A hybrid transformer-mamba language model, 2024.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pp. 8024–8035. Curran Associates, Inc., 2019.

Guilherme Penedo, Hynek Kydlíček, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro von Werra, and Thomas Wolf. The FineWeb datasets: Decanting the web for the finest text data at scale. In *Advances in Neural Information Processing Systems*, volume 37, pp. 30811–30849. Curran Associates, Inc., 2024. doi: 10.52202/079017-0970.

Aurko Roy, Mohammad Saffar, Ashish Vaswani, and David Grangier. Efficient content-based sparse attention with routing transformers. *Transactions of the Association for Computational Linguistics*, 9:53–68, 2021. doi: 10.1162/tacl_a_00353.

Jianlin Su, Murtadha H. M. Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. RoFormer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024. doi: 10.1016/j.neucom.2023.127063.

Philippe Tillet, H. T. Kung, and David Cox. Triton: An intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, Mapl '19, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. doi: 10.1145/3315508.3329973.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, volume 30, pp. 5998–6008. Curran Associates, Inc., 2017.

An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, Chujie Zheng, Dayiheng Liu, Fan Zhou, Fei Huang, Feng Hu, Hao Ge, Haoran Wei, Huan Lin, Jialong Tang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Yang, Jiaxi Yang, Jing Zhou, Jingren Zhou, Junyang Lin, Kai Dang, Keqin Bao, Kexin Yang, Le Yu, Lianghao Deng, Mei Li, Mingfeng Xue, Mingze Li, Pei Zhang, Peng Wang, Qin Zhu, Rui Men, Ruize Gao, Shixuan Liu, Shuang Luo, Tianhao Li, Tianyi Tang, Wenbiao Yin, Xingzhang Ren, Xinyu Wang, Xinyu Zhang, Xuancheng Ren, Yang Fan, Yang Su, Yichang Zhang, Yinger Zhang, Yu Wan, Yuqiong Liu, Zekun Wang, Zeyu Cui, Zhenru Zhang, Zhipeng Zhou, and Zihan Qiu. Qwen3 technical report, 2025a.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving Mamba2 with delta rule. In *The Thirteenth International Conference on Learning Representations*, 2025b.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Yuxing Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 23078–23097, Vienna, Austria, July 2025. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.1126.