Compress & Attend: One Language Model Across the Quality–Throughput Frontier

Anonymous

Abstract

The substantial inference costs of attention in transformers motivated the development of efficient sequence mixers: namely sparse and sliding window attention, convolutions and linear attention. Although these approaches result in impressive reductions in per-token inference costs, they often trade off with quality, specifically in-context recall. Apriori fixing this quality-cost trade-off at training time means being suboptimal from the get-go: some downstream applications might fundamentally require more memory for in-context recall, while other tasks may require lower latency and memory. This trade-off suggests the need for adjustable per-token inference costs without retraining. We propose a conceptually simple *meta*-sequence mixer: the Compress & Attend Transformer (CAT). CAT decodes chunks of tokens by *attending* to compressed chunks of the sequence so far. Both compression and decoding can use any existing sequence mixer. Decoding from the compressed sequence yields compute and memory savings, with chunk size setting the operating point on the quality-cost trade-off. Importantly, training CAT across multiple chunk sizes at once unlocks test-time control of this trade-off without any retraining, all in a single model. Instantiated with the most basic choice, dense attention as the mixer, CAT *surprisingly* suffices to match 10 popular and diverse efficient models (linear, hybrids, sparse) on real-world long-context recall at comparable inference costs, all from a single trained model. CAT further performs competitively on long-context understanding benchmarks while providing $1.4 - 3.2\times$ higher generation throughput than a dense transformer.

1 Introduction

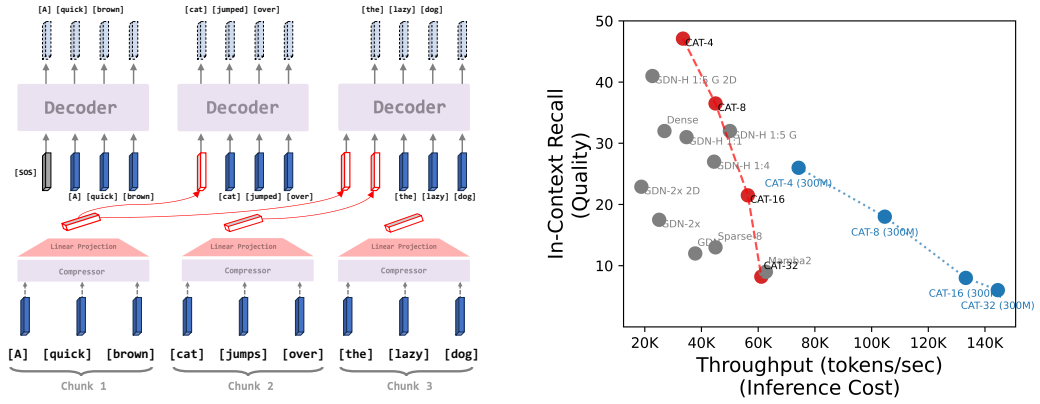
The cost of serving language models, especially in the era of reasoning and agents, outstrips their training cost. In fact, training costs can break even with inference cost in a matter of *weeks*[1] (Timbers, 2023). In turn, what matters now for long-term deployment is the quality per unit of inference cost that a language model provides. While a model using dense attention (Bahdanau et al., 2014; Vaswani et al., 2017) as a sequence mixer provides good quality, its per-token inference cost grows quickly with sequence length, making it expensive to deploy at long contexts with the current hardware.

This large cost of dense attention at long-contexts motivated efficient alternatives in the community. Approaches such as sparse and sliding window attention (Child et al., 2019; Zaheer et al., 2020; Jiang et al., 2023) heuristically restrict the tokens being attended to, and those such as linear attention and state-space models (Katharopoulos et al., 2020; Arora et al., 2024a; Dao and Gu, 2024; Yang et al., 2025) use a fixed-size recurrent state to reduce inference costs.

In all of these approaches, however, the reduced inference cost comes at the cost of quality: specifically, it trades off with in-context recall (Arora et al., 2024a; Jelassi et al., 2024; Wen et al., 2024). Importantly, these approaches expose this trade-off only through training-time choices. However,

[1] `together.ai` serves 400T tokens/month for open-source models (source). One of the popular open-source series `Qwen-3` was pretrained only on a total 36T tokens in total

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.



(a) **The Compress and Attend Transformer (CAT) architecture.** CAT chunks up a sequence of length N into N/C chunks of C tokens (illustrated for $C = 3$). Each chunk is parallelly compressed into a chunk representation. CAT then decodes each chunk by attending to past chunk representations. Observe the **reduced** sequence length in the decoder due to compression. The number of compressed chunks **grow** with sequence length resulting in *gracefully* growing memory. Chunk size in CAT acts as a ***knob***, offering test-time control of quality-efficiency trade-offs, where higher chunk sizes result in more efficiency.

(b) **CAT unlocks test-time control of quality-inference cost trade-offs**: a **single** adaptive CAT model (**red** dots) outperforms different families of efficient approaches at comparable throughputs (inference costs) on real-world in-context recall tasks. We compare **across 10 models** (**gray**), all having diverse model configurations, parameter counts (∼250M to ∼820M) and **varying inference costs** for fair and broad evaluation. **Blue** is a second CAT, *parameter-matched* (∼300M) to Dense; it achieves dramatically larger throughput than the efficient alternatives while outperforming several at recall.

Figure 1: Overview of CAT.

different downstream tasks have different quality requirements at test time, which an apriori training-time choice cannot accommodate. A general chat assistant writing short email replies has weak recall demands, and linear attention suffices; whereas a coding agent, by contrast, must recall function names across a repository-scale context, where the stronger recall of dense attention is worth its higher cost. A single high-recall model wastes compute on the email task, while a single low-recall model fails the coding agent's quality bar. One could pretrain different models for different tasks; this gets prohibitive for all possible downstream tasks. Hence, a single model with a knob to control the trade-off at test time becomes desirable, which no other existing sequence mixer *natively* provides.

Comparing quality based on different architectural traits, such as parameters, FLOPs, and memory footprint[2], *may* be misleading: two models with identical FLOPs or parameters can differ substantially in the actual monetary cost to serve them due to poor implementations and hardware *unfriendly* design. What matters is the hardware cost you actually pay for. Hardware gets priced in dollars per hour, which means the quantity to measure a model by so that it translates to cost is: tokens per hour (division gives the cost per token) or alternatively tokens per second, both of which measure throughput.[3] Thus, the standard for comparison is performance at various inference costs measured by throughput levels.

This paper provides a simple recipe to make inference cost controllable at test time. The recipe's key ingredient is a *meta*-sequence mixer: a conceptually simple arrangement of existing off-the-shelf sequence mixers where one mixer compresses the sequence, and another attends to this compressed sequence while decoding. We term this approach the **C**ompress & **A**ttend **T**ransformer (CAT). Concretely, CAT compresses chunks of tokens in parallel into a shorter sequence using a compressor, which a decoder then attends to while autoregressively modeling the tokens in the latest chunk. The compression and decoding are parallel over tokens during training, meaning there is no recurrence along the sequence (unlike Rae et al. (2020)), enabling *end-to-end scalable training*. When the cost of the sequence mixer used in CAT depends on the length of the sequence, decoding happens at a reduced sequence length due to compression (see Figure 1a) enabling *compute and total memory*

[2]An extreme example: imagine an architecture that has constant memory but keeps doing repeated reads and writes to this constant memory. While this has constant memory, the latency is poor due to high memory bandwidth in current hardware.

[3]we measure throughput as the maximum tokens per second a model can achieve under fixed hardware across batch sizes.

savings. Importantly, training CAT across multiple chunk sizes at once **unlocks controllability of quality-inference cost trade-offs directly at test time**.

Controllability should not come at the cost of performance. Most models improve in performance as the number of parameters increases. The downside of adding more parameters is higher inference costs, which depend on the architecture itself. Controllability, however, provides the ability to change the inference cost for the same number of parameters. In CAT, compression enables this efficiency, *so a feature of* CAT *is that it can use more parameters without driving up inference costs.* This same ability to increase the number of training parameters without increasing inference costs serves as a motivation for existing architectural choices such as mixture-of-experts (MoE), which can improve performance beyond that of a dense-attention transformer (Shazeer et al., 2017).

While one can instantiate the CAT meta-sequence mixer with any mixer, we implement CAT with the most vanilla off-the-shelf components: compression and decoding both use dense transformer black-boxes (see Figure 1a). Since dense attention is a staple and well-developed, training and inference can be done efficiently with existing infrastructure. We find CAT with simple dense attention is *surprisingly* effective: across benchmarks, CAT matches or surpasses several popular alternatives without custom kernels or requiring careful choices (say choosing attention-linear ratios in hybrids Wang et al. (2025)) across inference budgets.

The computational structure of CAT enables substantial reductions in inference cost, allowing CAT to remain efficient even with many more parameters. Figure 1b reports the performance-efficiency trade-off curve achieved by a **single** CAT model with more than twice as many parameters as its dense-attention counterpart (**red** dots). CAT achieves performance comparable to or better than a variety of existing models across a range of inference costs. Parameter-matched CAT (**blue** dots) also performs well, but operates in a different efficiency regime, so its throughput is substantially higher than that of other approaches, making direct performance comparisons less meaningful.

To summarize, this paper makes the following contributions:

- **A controllably efficient meta-sequence mixer.** We introduce CAT, in which a single test-time knob (chunk size) traverses the quality–cost trade-off without retraining. A feature of CAT is that it enables more parameters at varying levels of throughput.
- **A simple instantiation that matches different families of efficient alternatives at various throughput levels.** Instantiating CAT with dense attention yields a single adaptive model that:
  - achieves strong quality–throughput trade-offs compared to 10 efficient models on real-world long-context recall, without careful choices or coding-up custom kernels (Figure 1b).
  - is competitive in long-context modeling and understanding benchmarks (Tables 2 and 3)
  - matches the dense transformer on language modeling and is upto $1.4 - 3.2\times$ faster on throughput depending on the chosen chunk size (Table 4)
  - surpasses dense transformer on real-world recall tasks at the most accurate setting (CAT-4) while achieving better throughputs ($1.45\times$) and is $1.5\times$ faster and $2\times$ memory efficient; Figure 1b.
- **Off-the-shelf implementation.** We provide a parallel, scalable training implementation (scaling from 90M to 1B parameters) and a pure-PyTorch generation implementation requiring no custom CUDA or Triton kernels – unlike most efficient alternatives.
- **Extensibility.** CAT as a meta-sequence mixer wraps around existing sequence mixers (e.g., hybrids) and can serve as a drop-in replacement layer in other architectures (Table 5).

2 Compress and Attend Transformers (CATs)

Compression and decoding. CAT is a meta–sequence mixer that uses one sequence mixer to compress chunks of a sequence and another to decode each chunk given compressed representations.

Concretely, given a sequence $\mathbf{x} = (x_1, x_2, \ldots, x_N)$ of N tokens, we split the sequence into chunks $(\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_{N_C})$ containing C tokens each, such that $\mathbf{c}_i = (x_{C\cdot i+1}, \ldots, x_{C\cdot i+C}) = (\mathbf{x}_{i,1}, \ldots \mathbf{x}_{i,C}) = \mathbf{x}_{i,:}$, where $\mathbf{x}_{i,:}$ indexes the i-th chunk of C consecutive tokens (`numpy` array slicing). CAT compresses each chunk $\mathbf{c}_i$ using the compressor f_θ into chunk representations. The compressor f_θ is any sequence mixer with hidden size D_f, followed by a linear projection to D_g.

This leads to a compressed chunk representation $f_\theta(\mathbf{c}_i) \in \mathcal{R}^{D_g}$. That is:

$$\{x_1, \cdots x_N\} \xrightarrow{\text{chunk}} \{\mathbf{c}_i\}_{i=1}^{N_c} \xrightarrow{\text{compress}} \{f_\theta(\mathbf{c}_i)\}_{i=1}^{N_c}$$

After compression, CAT decodes the original sequence $\mathbf{x}$ from the compressed chunk representations $\{f_\theta(\mathbf{c}_i)\}_{i=1}^{N_C}$ using a decoder g_θ, which is a causal sequence mixer, having hidden size D_g, matching the linear projection from the compressor. CAT decodes chunks autoregressively, where to decode each token $\mathbf{x}_{i,j}$ in a chunk $\mathbf{c}_i$, the decoder takes as input the previous tokens $\{\mathbf{x}_{i,<j}\}$ in chunk $\mathbf{c}_i$ and the past chunk representations $\{f_\theta(\mathbf{c}_1), \ldots, f_\theta(\mathbf{c}_{i-1})\}$. Formally, the predictive distribution p_θ is:

$$p_\theta(\mathbf{c}_i \mid \mathbf{c}_{i-1} \cdots \mathbf{c}_1) = \prod_{j=1}^{C} g_\theta \left(\underbrace{\mathbf{x}_{i,j}}_{j^{\text{th}}\text{token in chunk } \mathbf{c}_i} \middle| \underbrace{\mathbf{x}_{i,j-1}, \ldots, \mathbf{x}_{i,1}}_{\text{previous tokens in chunk } \mathbf{c}_i}, \underbrace{f_\theta(\mathbf{c}_{i-1}) \cdots f_\theta(\mathbf{c}_1)}_{\text{past chunk representations}} \right) \tag{1}$$

When decoding cost depends on sequence length, CAT improves throughput by reducing the compute and memory required; the larger the chunk size, the larger the reduction.

During training, compression and decoding happen in parallel for all tokens in the sequence because the compression of a chunk does not depend on earlier chunks. This choice allows the entire CAT model to be efficiently trained end-to-end with the standard next-token prediction loss. The end-to-end training lets CAT *learn what to retain* in its compressed chunk representations, rather than relying on fixed attention patterns or complex state update rules.

Training for test-time control of inference cost. Changing the chunk size in CAT trades off quality for compute and memory efficiency. Training CAT with multiple chunk sizes yields a single adaptive model whose compute-memory budget can be adjusted at test time without retraining. We uniformly sample a chunk size C at each training iteration and pass a *learnable* indicator token to CAT to indicate the current chunk size. The compressed tokens are separated from the uncompressed ones in the decoder using a marker token shared across chunk sizes. After training, one can use the same CAT model at different compute/memory budgets by changing the indicator token at test time. Appendix C.4 provides further details.

CAT as a layer. The principles discussed above are architecture-agnostic: CAT can be instantiated as a modular layer that can be inserted in any architecture, unlocking controllable cost there and enabling new hybrid designs. In layer form, a simple linear projection can serve as the compressor, and a sequence mixer (e.g., dense attention) as the decoder.

2.1 How to implement fast and scalable CATs

For simplicity, we instantiate the sequence mixers for both compression and decoding as *vanilla dense attention*, demonstrating just how **far** such simple design choices can go with CAT, yielding a competitive and test-time controllable architecture. With dense transformers serving as both the compressor and the decoder, CAT admits a pure PyTorch implementation for scalable training and fast generation requiring no custom CUDA or Triton kernels. We outline this approach here.

Fast and Parallel Compression. Compression of chunks of tokens is efficient and can be executed in parallel, for instance by using `torch.vmap`, to produce $\{f_\theta(\mathbf{c}_i)\}$ for all chunks $\mathbf{c}_i$. This costs a total of $O(\frac{N}{C} \cdot C^2) = O(NC)$ in self-attention compute, rather than $O(N^2)$.

Naive and Slow Training. For training the decoder, a naive implementation can lead to slower training. To compute logits for tokens in chunk $\mathbf{c}_i$, that is computing $g_\theta(\mathbf{c}_i \mid f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1}))$ in parallel can be non-trivial. Since, for chunk $\mathbf{c}_i$, the number of past chunks varies, making shapes variable and as a result, harder to parallelize the computation of logits. One could employ a python loop and compute logits for every chunk sequentially, but that would be slow and would not scale. Padding to make shapes constant to allow parallelism would make things worse by increasing wasteful computations. In fact, even if one bypasses varying shapes problem and manages to compute logits for every chunk in parallel, the total self-attention operations in the decoder would scale as $O(\sum_{i=1}^{N_c}(i + C)^2) = O((\frac{N}{C})^3)$, that is cubic in sequence length. Thus, even the ideal parallel approach for training will not scale, despite the simplicity of CAT.

Parallel and Scalable Training. To overcome above **training challenges** in CATs, we observe that in computing logits for every chunk $\mathbf{c}_i$, one calculates exactly the same key-value vectors for the representation $f_\theta(\mathbf{c}_j)$ in the decoder transformer, where $j < i$. This means that computation is duplicated. We exploit this observation in training CATs. We implement training by interleaving compressed representations into the sequence: $\{\mathbf{c}_1, f_\theta(\mathbf{c}_1), \mathbf{c}_2, f_\theta(\mathbf{c}_2), \dots\}$. A custom attention mask (App. Figure 11) lets a token in chunk $\mathbf{c}_i$ attend to earlier tokens in the same chunk and to prior chunk representations $f_\theta(\mathbf{c}_{<i})$, but not to raw tokens in other chunks. This lets the decoder reuse the keys and values of $f_\theta(\mathbf{c}_i)$ when computing logits for any later chunk. The resulting complexity is $O(N^2/C)$ – a constant-factor improvement over the dense transformer's $O(N^2)$, enabling potentially faster pre-training (see Section 5).

Fast and Efficient Generation. Due to compression, CATs can throw away past chunks of tokens, and only keep their compressed chunk representations in memory. This straightaway results in a big reduction of memory; the KV cache is slashed by a factor of C, even for a modest chunk size of 4 (see Table 4). **Notably these memory savings are *independent* of sequence length**; in other words, CAT always results in memory reduction ***relative*** to a dense transformer at any sequence length (be it 4K or 128K). This compressed sequences means both reduced HBM accesses and that the decoder attends to atmost $\frac{N}{C} + C$ tokens during generation, resulting in an increased throughput.

Implementing generation is similar to how it occurs for a dense transformer. A pure PyTorch implementation (inspired by gpt-fast) for CATs is on par with efficient architectures that utilize custom kernels. Given a prompt, CAT first computes chunk representations in parallel and prefills them into the decoder's KV cache. Generation then proceeds chunk by chunk: tokens within a chunk are decoded sequentially, and on chunk completion the chunk is compressed and its representation is appended to the KV cache before decoding continues. See Sections C and E.3 for further details and PyTorch style pseudo-code.

Scaling parameters while remaining efficient. As CAT's decoder operates on a compressed sequence rather than the full one, its compute and memory requirements are lower than a dense transformer, at the same parameter count as chunk size increases. This computational structure allows for the usage of a larger decoder with more parameters in CAT while keeping inference cost below that (or comparable) to the smaller parameter dense transformer and many parameter matched models.

Method	Unrestricted access to memory?	Flexible memory?	Scalable training?	Compute & memory efficient?	Controllable inference costs?	Mixable with CAT?	Usable in CAT's decoder?
Dense (Vaswani et al., 2017)	✓	✓	✓	✗	✗	✓	✓
Sparse attention (Child et al., 2019)	✗	✓	✓	✓	✗	✓	✓
NSA (Yuan et al., 2025)	✓	✓	✓	✗	✗	✓	✓
Sliding window attn. (Jiang et al., 2023)	✗	✗	✓	✓	✗	✓	✓
Linear attention (Dao and Gu, 2024)	✓	✗	✓	✓	✗	✓	✗
Recursive compression (Chevalier et al., 2023)	✓	✓	✗	✓	✗	✓	✓
MegaByte/Block Tf. (Ho et al., 2024; Yu et al., 2023)	✓	✗	✓	✓	✗	✓	✓
CATs	✓	✓	✓	✓	✓	✓	✓

Table 1: We categorize existing related work by key properties desirable for an efficient architecture, and indicate whether CAT can complement these approaches as a **meta-sequence mixer**. *"Unrestricted access to memory?"* signifies whether an architecture can freely access any part of the memory in the past, without any artificial restrictions, *"Mixable with CAT?"* indicates whether CAT as a layer be used in these approaches, *"Usable in CAT's decoder?"* indicates whether the method can serve as a compressor or decoder within CAT.

3 Related work.

Efficient sequence mixers reduce attention's cost in different ways: sparse and sliding window attention restrict which tokens are attended to (Child et al., 2019; Zaheer et al., 2020; Jiang et al., 2023); linear attention and state-space models replace softmax with a fixed-size recurrent state (Katharopoulos et al., 2020; Dao and Gu, 2024; Yang et al., 2025); recurrent compression accumulates state sequentially (Rae et al., 2020; Chevalier et al., 2023); and hierarchical or chunk-based architectures compress the sequence into coarser units (Nawrot et al., 2021; Yu et al., 2023; Ho et al., 2024; Pagnoni et al., 2025). Each family trades off something — recall (Arora et al., 2024a; Jelassi et al., 2024;

Wen et al., 2024), training scalability (Geiping et al., 2025), or careful hybrid tuning (Waleffe et al., 2024; Wang et al., 2025) — and crucially, **none expose test-time control of inference cost**. CAT is complementary rather than competing: any length-dependent mixer can serve as its compressor or decoder and inherit controllability, and orthogonal techniques (e.g., MoEs (Shazeer et al., 2017), speculative decoding, training-free sparsification (Nawrot et al., 2025)) compose on top of CAT.

CAT complements most existing (or future) approaches, can extend them, or be mixed with them to unlock test-time control of inference cost. Table 1 highlights the relationships and conceptual differences between CAT and the most relevant related work, and App. A gives the full related work.

4 Experiments

4.1 Models in Comparison and Training Setup

CAT as a *meta*-sequence mixer complements any sequence mixer whose decoding cost depends on sequence length – instantiating CAT's decoder with such a mixer reduces costs due to the compressed sequence. This includes sequence mixers like dense attention including hybrid models using attention in few layers. Consequently, the only *strict competitors* are sequence mixers with length-independent costs, such as pure linear attention models (which cannot be composed with CAT; see Table 1). The main results reflect this categorization: into (a) strict baselines and (b) *complementary to* CAT, for clarity. That being said, we compare broadly against recent state-of-the-art architectures (across hyper-parameters) of both kinds with different quality-throughput trade-offs, and show that a single CAT model with simple dense attention is **highly competitive across the throughput levels**.

Models in comparison: Our evaluations include: (i) attention-based: standard dense transformer (Touvron et al., 2023) and sparse transformer (Child et al., 2019), (ii) Linear Transformers such as Mamba2 (Dao and Gu, 2024) and GatedDeltaNet (GDN) (Yang et al., 2025), as well as (iii) Hybrid architectures with GDN and attention layers interleaved in some pre-specified ratio. By default, all models below are configured with $L = 12$ layers and $D = 1024$ hidden dimension; any deviations are explicitly stated below. Further, going beyond the default hyperparameter settings, we scale up each model type by adjusting their model configurations to yield models at varying throughput levels: (i) linear attention with a $2\times$ recurrent state size (i.e. GDN-$2\times$), (ii) multiple dense-linear attention ratios in hybrid architectures (i.e. GDN-H 1:1, GDN-H 1:4), (iii) hybrids with global attention layers (i.e. GDN-H 1:5 G), (iv) and $2\times$ the model dimension (for instance GDN $2\times$ 2D, GDN-H 1:5 G 2D, Sparse-4/8). Most model types has at least 2 configurations to ensure fair and broader comparison. This process results in a total of 10 models with parameter counts ranging from $\sim$250M **to upto** $\sim$**820M**, and **varying** throughput levels.

We compare these 10 models against a *single* CAT model. The only strict baselines to CAT are sequence mixers whose inference costs are length independent – this includes Mamba2, GDN, GDN-$2\times$ and the larger parameter GDN-$2\times$ 2D. The rest in our broad comparison are *complementary to* CAT. See section 5 for further discussion on the complementary nature of CATs (Table 5).

CAT configuration: We study performance at various throughput levels. The decoder holds the majority of parameters in CAT, and dominates overall inference cost (see Section 4), and thus determines throughput. Parameter-matched CAT achieves substantially higher throughputs than any baseline we compare against, and therefore is incomparable to them. So, we report results here with a CAT model that has twice the hidden dimension in the decoder as that of the dense transformer (Section B.6 provides the parameter-matched results). We use $L = 12$ layers (same as baselines), but a wider hidden size of $D_g = 2D = 2048$ for the decoder. The compressor is small and uses $L = 3$ layers and hidden size of $D_f = D = 1024$. Depth of compressor does not have major effect (App. D). This makes the parameter count for CATs nearly $\sim (820 + 150)$ M parameters (similar to the larger models in our comparison). We train CAT simultaneously on chunk sizes $C = \{4, 8, 16, 32\}$ to yield a single model that works with different chunk sizes at once, offering test-time control of inference costs.

Setup: All models were trained on 15B tokens of FineWeb-Edu Penedo et al. (2024) with a context length of 4K following Behrouz et al. (2024); Yang et al. (2025). We use the AdamW optimizer Loshchilov and Hutter (2017) with a peak learning rate of 8e-4, weight decay of 0.1, gradient clipping of 1.0, batch-size of 0.5M tokens, employing the GPT2 tokenizer. App. E provides details for each model and training. To obtain throughput, refer to details at Section E.3.

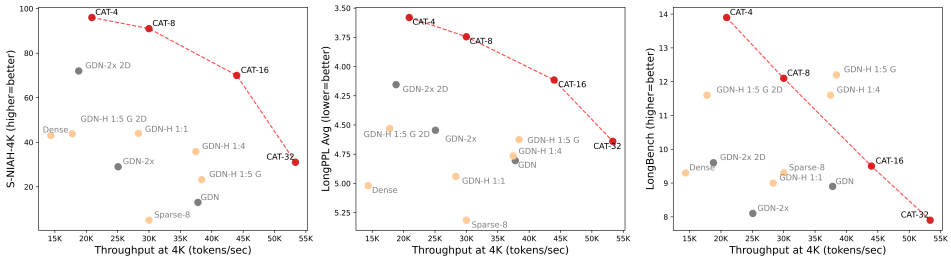


Figure 2: We report quality-throughput trade-offs for different models across diverse long-context tasks. CAT provides a strong trade-off, all in a single model. Refer to Table 2 for needle-in-haystack task (NIAH-N); Table 7 in appendix reports full evaluations on the LongPPL and LongBench.

4.2 Results

Long-context recall: Figure 1b reports results on real-world in-context recall tasks from Arora et al. (2024a) for a given throughput requirement. We report results on SWDE and FDA, which have longer sequences among the datasets in the suite (others have an average length of ≤ 300 tokens (Arora et al., 2024b)). Table 9 provides numbers, and B.10 shows evaluations on all datasets. **CAT performs as good or better than all models across inference cost budgets (throughputs), using a single model only.** Linear models (Mamba2, GDN) lag far behind dense attention, while hybrids reduce the gap. **CAT provides strong recall-cost trade-off, benefiting from the gracefully growing memory**, i.e. growing linearly but by a constant factor less compared to dense attention. CAT interestingly outperforms even the dense transformer at these tasks (at moderate chunk sizes $= 4, 8$), while having a higher throughput. Section E.2 provides more details about the task.

Table 2: Accuracy on RULER Hsieh et al. (2024) S-NIAH-N benchmark and average LM evals Gao et al. (2024). All CAT results come from a single model, evaluated at different chunk sizes. Figure 2 plots these results against throughput. Tables 10 and 11 in the appendix provide more results.

	S-NIAH-N (↑)			LM Evals[3] (↑)
Model	**1K**	**2K**	**4K**	**Avg.**
Strict baselines				
GDN	84.7	69.1	13.6	43.5
GDN-2×	78.0	61.4	29.0	43.8
GDN-2× 2D	99.3	97.0	72.0	46.4
CAT-4	96.0	97.0	**96.0**	43.9
CAT-8	90.0	93.0	91.0	44.1
CAT-16	76.0	72.0	70.0	44.3
CAT-32	60.0	37.0	31.0	45.1
Complementary				
Dense	96.0	92.0	43.0	42.1
GDN-H 1:1	99.0	97.0	44.0	43.0
GDN-H 1:5 G 2D	**99.5**	**99.3**	43.8	45.8

Figure 2 reports results on the needle-in-haystack task (NIAH-N) from the RULER benchmark Hsieh et al. (2024), that is, retrieve a seven token number from long-context.

CATs outperform the existing efficient models as context length increases, and **interestingly show *slower* degradation with length**. This slow degradation can possibly be attributed to reduced sequence length in CAT that leads to fewer *distractions* for the saturating dense attention on long-contexts (Barbero et al., 2024; Vasylenko et al., 2025; Chiang and Cholak, 2022; Golovneva et al., 2025). More discussion can be found in App. B.9, where we extend these results to the harder task from RULER (NIAH-U: retrieve 32 token `uuids`).

Long-context language modeling and understanding: We test language modeling and understanding on long contexts (upto 4K contexts). As standard perplexity (or test log-loss) averaged over all tokens **does not** indicate downstream long-context ability (Fang et al., 2025; Liu et al., 2023; Hu et al., 2024), fig. 2 conducts evaluations on the LongPPL Fang et al. (2025) metric, that calculates loss on a few key tokens which are essential for long-context understanding. We employ Llama-3.1-8B as evaluator. Additionally, fig. 2 reports evaluation on LongBench Bai et al. (2023) that tests for long-context understanding. All CATs perform competitively on longer contexts.

Short-context language modeling and understanding benchmarks: The last column in Table 2 (and Table 11 in appendix) reports zero-shot accuracies on key common-sense reasoning benchmarks for completeness where CATs show similar performance to the rest of the models.

[3]common LM eval measures performance on **short sequences** ($\leq$ 30 tokens on average), which is *not* indicative of long-context performance, as reported multiple times in the literature Bertsch et al. (2026); Fang et al. (2025). We perform the eval for completeness and transparency.

Table 3: Longer-context evaluation on RULER NIAH-N Hsieh et al. (2024). CAT degrades *gracefully* as context grows; all CATs are a single model with test-time adjustment. †16K uses continued pre-training of the 8K model on 16K sequences.

Model	2K	4K	8K	16K†	LM evals ↑
CAT-4	99.0	90.4	**98.0**	**90.1**	44.6
CAT-8	94.4	**92.0**	87.0	82.1	44.5
CAT-16	82.7	82.7	57.8	51.8	45.3
CAT-32	71.8	42.9	26.2	11.6	45.8
Dense	99.3	59.5	37.9	5.0	44.0
GDN-H 1:1	**99.7**	53.2	24.9	13.2	44.4

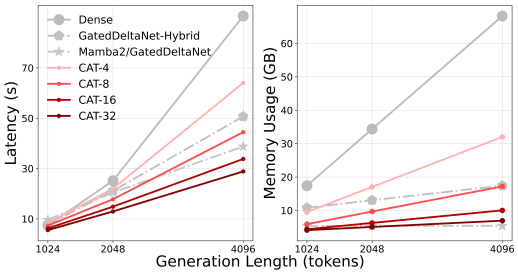


Table 4: A single CAT model generates $1.4-3.2\times$ faster than the dense transformer with $2.2-9.5\times$ lower memory usage.

Benchmarking generation: Figure 4 compares architectures as one scales the sequence length, with a fixed batch-size of 320 to maximize throughput. CAT generates sequences $1.4-3.2\times$ **faster** than the dense transformer while showcasing **up to** $2.2-9.5\times$ **lower total memory usage** as one increases chunk sizes, despite using significantly more parameters than the baselines due to wider decoder and the additional compressor. This is not surprising since the major bottlenecks during generation are: (a) KV cache size that drives the main memory requirement during generation and not the parameter count (Sec. 5), (b) memory accesses required for a token, and (c) FLOPs used per token determined by the past tokens being attended to. CATs reduce these factors despite carrying more parameters overall. App. E.3 provides implementation details.

Appendix: We provide additional results demonstrating CAT's flexibility and scalability. First, we show CAT can be used as a drop-in layer (App. B.11) and that its compressor and decoder can make use of any sequence mixer (App. B.12). We then present scaling experiments (App. 8), ablations on design choices (App. D), evaluations on the synthetic MQAR task (App. B.7 – tested on $4\times$ longer sequences than usual), and analysis of perplexity across chunk boundaries (App. B.14).

4.2.1 Long(er) Context Evaluation

We provide scaled up results on 8K sequence length for a few models in our comparison and CAT. All models were trained from scratch on 8K sequence length for 30B tokens on FineWeb-Edu Penedo et al. (2024). We additionally increased the number of layers to 18 for all models. Table 3 reports these. We restrict this study to a subset of models due to compute constraints. To stress test at longer contexts, we perform continued pre-training for all models for additional 1B tokens at 16K sequence length, and then evaluate them on NIAH-N task again. At both context lengths, CAT again provides a better or similar trade-off across different throughput levels all in a single model.

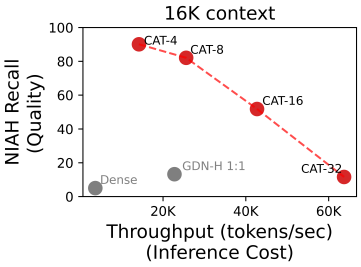


Figure 3: Table 3 plotted against throughput.

4.3 CAT is a *meta*-sequence mixer

CAT is a *meta*-architecture that wraps around any sequence mixer. Table 5 reports exploratory results, and full exploration is left to future work, showing the compute and memory savings achieved by CAT, when composed with GDN-H instead of the dense Transformer. The gains provided by CAT amplifies that provided by GDN-H. Any mixer with a length-dependent decoding cost — dense, sparse, or hybrid — wrapped by CAT achieves compute and memory savings and enables controllable efficiency. So, any length-dependent mixers, and techniques that accelerate them (e.g., speculative decoding (Leviathan et al., 2023)), are complementary to CAT.

Table 5: CAT composes with existing efficient sequence mixers: GDN-H 1:1 as the CAT decoder saves more memory than either component alone, and even improves recall. Trained at 1K context for 5B tokens, chunk size 16. See Section B.12.

Model	Mem. (↑)	SWDE (↑)
Dense	$1.0\times$	39.2
GDN-H 1:1	$2.5\times$	28.0
CAT-16 (Dense)	$7\times$	13.5
CAT-16 (GDN-H)	$\mathbf{14}\times$	16.5

5 Discussion

Gracefully growing memory. Sequence mixers like dense attention or linear recurrences take an *extreme* route for long-context modeling: either the memory grows *too quickly*, or the memory is fixed. CATs take a **middle-ground**, where the memory increases *gracefully* by constant factor

less than dense. This flexible memory results in better recall-cost trade-off. Interestingly, growing memory may result in *better memory management* as compared to fixed memory models Wen et al. (2024) despite using the same memory at a particular sequence length. We stress test this memory management on the MQAR task Arora et al. (2023a) in Figure 4 (more details in Section B.7). CAT provides strong performance-cost trade-off of in MQAR despite taking the same memory as fixed memory models. This better trade-off also echoes in our main evaluations (Figure 1b).

However, note that CAT fundamentally does have length-dependent costs, and a pure linear model (say, GDN-2$\times$ 2D) may overtake CAT in raw throughput at very long sequences. However, the question is whether throughput comes with *usable* quality. Constant memory trades off severely with long-context performance, making it ineffective at the very task it was designed for. Hybrids recover part of this trade-off by reintroducing a growing memory of dense attention, and CAT does the same through its growing set of compressed chunk representations. Together, these point to a fundamental property for modeling long contexts well: a memory that grows with the context.

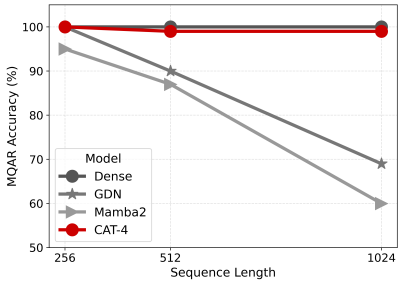


Figure 4: Comparison of sequence mixers on MQAR Arora et al. (2023a) task (up to 4$\times$ the usual length). All models are **memory-matched** in bytes at **every sequence length** (except dense transformer). CAT outperforms linear models especially at longer sequences, while still consuming same memory (or state size), providing a better trade-off, possibly due to better learning in a gracefully growing memory.

What is the training cost of CAT? CAT reduces attention FLOPs significantly through compressed sequences, but training cost depends on both attention and feedforward layers. At 4K sequence length, feedforward FLOPs dominate over attention (Austin et al., 2025), so CAT's training cost is $\sim 2.5\times$ that of a smaller but throughput-matched dense transformer. This overhead shrinks as sequences lengthen and attention becomes the bottleneck: at 16K it falls to $\sim 1.25\times$, and at 64K it drops below the throughput-matched dense baseline despite CAT's larger parameter count. More importantly, **at any sequence length**, CAT **amortizes** training cost by exposing multiple operating points from a single model. Training a separate model for each operating point would cost more in total: matching the throughputs of GDN, GDN-2$\times$, GDN-2$\times$ 2D, and GDN-H 1:1 with independent runs takes at least 2$\times$ the training FLOPs of a single adaptive CAT, and the gap widens as pretraining moves to longer sequences. Since inference cost dominates the total cost of building and deploying models, the ability to serve at multiple throughput levels from a single model is more valuable than the training overhead. See Section C.5 for details.

6 Conclusion and Future Work

This paper set out to provide a simple recipe for making per-token inference cost controllable at test time, while enabling the use of more parameters at varying levels of throughput. CAT delivers on this: a single CAT model exposes a test-time knob (chunk size) that traverses the quality–throughput frontier without retraining. Instantiated with plain dense attention — an off-the-shelf component requiring no hyperparameter tuning or custom kernels — CAT matches or surpasses popular efficient baselines at various throughput levels.

> The value of CAT lies not in any single benchmark comparison, but in providing a **simple** recipe (using off-the-shelf mixers and existing infrastructure) to train models whose inference cost can be controlled at test time. As it stands, ***no* other sequence mixer provides such controllability natively**, and CAT *wraps* around them as a *meta*-sequence mixer to provide this ability.

Limitations and Future Work. Future work should explore how to provide a better trade-off despite a growing memory. Another interesting direction is data-dependent adaptivity. CAT, as it stands, requires users to choose a chunk size appropriate for their compute and memory budgets. Instead, one could post-train CAT to learn to allocate budget itself based on the context and task. Such post-training would enable truly adaptive efficiency. Finally, scaling CAT to larger model sizes and trillions of tokens, and evaluating on reasoning-heavy benchmarks (`GSM8K`, `MATH`, `GPQA`, `SWE-bench`), is beyond our limited compute budget and is left to future work ; we expect the benefits shown here to transfer at scale, as observed in prior work (Yang et al., 2025; Dao and Gu, 2024).

References

Sandhini Agarwal, Lama Ahmad, Jason Ai, Sam Altman, Andy Applebaum, Edwin Arbus, Rahul K Arora, Yu Bai, Bowen Baker, Haiming Bao, et al. gpt-oss-120b & gpt-oss-20b model card. *arXiv preprint arXiv:2508.10925*, 2025.

Simran Arora, Sabri Eyuboglu, Aman Timalsina, Isys Johnson, Michael Poli, James Zou, Atri Rudra, and Christopher Ré. Zoology: Measuring and improving recall in efficient language models. *arXiv preprint arXiv:2312.04927*, 2023a.

Simran Arora, Brandon Yang, Sabri Eyuboglu, Avanika Narayan, Andrew Hojel, Immanuel Trummer, and Christopher Ré. Language models enable simple systems for generating structured views of heterogeneous data lakes. *arXiv preprint arXiv:2304.09433*, 2023b.

Simran Arora, Sabri Eyuboglu, Michael Zhang, Aman Timalsina, Silas Alberti, Dylan Zinsley, James Zou, Atri Rudra, and Christopher Ré. Simple linear attention language models balance the recall-throughput tradeoff. *arXiv preprint arXiv:2402.18668*, 2024a.

Simran Arora, Aman Timalsina, Aaryan Singhal, Benjamin Spector, Sabri Eyuboglu, Xinyi Zhao, Ashish Rao, Atri Rudra, and Christopher Ré. Just read twice: closing the recall gap for recurrent language models. *arXiv preprint arXiv:2407.05483*, 2024b.

Jacob Austin, Sholto Douglas, Roy Frostig, Anselm Levskaya, Charlie Chen, Sharad Vikram, Federico Lebron, Peter Choy, Vinay Ramasesh, Albert Webson, and Reiner Pope. How to scale your model. 2025. Retrieved from https://jax-ml.github.io/scaling-book/.

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Yushi Bai, Xin Lv, Jiajie Zhang, Hongchang Lyu, Jiankai Tang, Zhidian Huang, Zhengxiao Du, Xiao Liu, Aohan Zeng, Lei Hou, et al. Longbench: A bilingual, multitask benchmark for long context understanding. *arXiv preprint arXiv:2308.14508*, 2023.

Federico Barbero, Andrea Banino, Steven Kapturowski, Dharshan Kumaran, João Madeira Araújo, Oleksandr Vitvitskyi, Razvan Pascanu, and Petar Veličković. Transformers need glasses! information over-squashing in language tasks. *Advances in Neural Information Processing Systems*, 37: 98111–98142, 2024.

Ali Behrouz, Peilin Zhong, and Vahab Mirrokni. Titans: Learning to memorize at test time. *arXiv preprint arXiv:2501.00663*, 2024.

Amanda Bertsch, Luca Soldaini, Matthew R. Gormley, Graham Neubig, Hannaneh Hajishirzi, Kyle Lo, and Dirk Groeneveld. Cracks in the foundation: Seemingly minor architectural choices impact long context extension. Technical report, Allen Institute for AI, 2026. URL `https://allenai.org/papers/olmpool`.

Lucas Beyer, Pavel Izmailov, Alexander Kolesnikov, Mathilde Caron, Simon Kornblith, Xiaohua Zhai, Matthias Minderer, Michael Tschannen, Ibrahim Alabdulmohsin, and Filip Pavetic. Flexivit: One model for all patch sizes. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 14496–14506, 2023.

Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.

Alexis Chevalier, Alexander Wettig, Anirudh Ajith, and Danqi Chen. Adapting language models to compress contexts. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023. URL `https://openreview.net/forum?id=kp1U6wBPXq`.

David Chiang and Peter Cholak. Overcoming a theoretical limitation of self-attention. *arXiv preprint arXiv:2202.12172*, 2022.

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Damai Dai, Chengqi Deng, Chenggang Zhao, RX Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Yu Wu, et al. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. *arXiv preprint arXiv:2401.06066*, 2024.

Tri Dao and Albert Gu. Transformers are ssms: Generalized models and efficient algorithms through structured state space duality. *arXiv preprint arXiv:2405.21060*, 2024.

Fnu Devvrit, Sneha Kudugunta, Aditya Kusupati, Tim Dettmers, Kaifeng Chen, Inderjit Dhillon, Yulia Tsvetkov, Hannaneh Hajishirzi, Sham Kakade, Ali Farhadi, and Prateek Jain. Matformer: Nested transformer for elastic inference. In *Workshop on Advancing Neural Network Training: Computational Efficiency, Scalability, and Resource Optimization (WANT@NeurIPS 2023)*, 2023. URL https://openreview.net/forum?id=93BaEweoRg.

Juechu Dong, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. Flex attention: A programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*, 2024.

Dheeru Dua, Yizhong Wang, Pradeep Dasigi, Gabriel Stanovsky, Sameer Singh, and Matt Gardner. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. *arXiv preprint arXiv:1903.00161*, 2019.

Lizhe Fang, Yifei Wang, Zhaoyang Liu, Chenheng Zhang, Stefanie Jegelka, Jinyang Gao, Bolin Ding, and Yisen Wang. What is wrong with perplexity for long-context language modeling? In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=fL4qWkSmtM.

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

Jonas Geiping, Sean McLeish, Neel Jain, John Kirchenbauer, Siddharth Singh, Brian R Bartoldson, Bhavya Kailkhura, Abhinav Bhatele, and Tom Goldstein. Scaling up test-time compute with latent reasoning: A recurrent depth approach. *arXiv preprint arXiv:2502.05171*, 2025.

Olga Golovneva, Tianlu Wang, Jason Weston, and Sainbayar Sukhbaatar. Multi-token attention. *arXiv preprint arXiv:2504.00927*, 2025.

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023.

Namgyu Ho, Sangmin Bae, Taehyeon Kim, Hyunjik Jo, Yireun Kim, Tal Schuster, Adam Fisch, James Thorne, and Se-Young Yun. Block transformer: Global-to-local language modeling for fast inference. *Advances in Neural Information Processing Systems*, 37:48740–48783, 2024.

Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

Yutong Hu, Quzhe Huang, Mingxu Tao, Chen Zhang, and Yansong Feng. Can perplexity reflect large language model's ability in long text understanding? *arXiv preprint arXiv:2405.06105*, 2024.

Samy Jelassi, David Brandfonbrener, Sham M Kakade, and Eran Malach. Repeat after me: Transformers are better than state space models at copying. *arXiv preprint arXiv:2402.01032*, 2024.

Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

Huiqiang Jiang, Yucheng Li, Chengruidong Zhang, Qianhui Wu, Xufang Luo, Surin Ahn, Zhenhua Han, Amir H Abdi, Dongsheng Li, Chin-Yew Lin, et al. Minference 1.0: Accelerating pre-filling for long-context llms via dynamic sparse attention. *Advances in Neural Information Processing Systems*, 37:52481–52515, 2024.

Mandar Joshi, Eunsol Choi, Daniel S Weld, and Luke Zettlemoyer. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pages 5156–5165. PMLR, 2020.

Aditya Kusupati, Gantavya Bhatt, Aniket Rege, Matthew Wallingford, Aditya Sinha, Vivek Ramanujan, William Howard-Snyder, Kaifeng Chen, Sham Kakade, Prateek Jain, et al. Matryoshka representation learning. *Advances in Neural Information Processing Systems*, 35:30233–30249, 2022.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning*, pages 19274–19286. PMLR, 2023.

Yuhong Li, Yingbing Huang, Bowen Yang, Bharat Venkitesh, Acyr Locatelli, Hanchen Ye, Tianle Cai, Patrick Lewis, and Deming Chen. Snapkv: Llm knows what you are looking for before generation. *Advances in Neural Information Processing Systems*, 37:22947–22970, 2024.

Hong Liu, Sang Michael Xie, Zhiyuan Li, and Tengyu Ma. Same pre-training loss, better downstream: Implicit bias matters for language models. In *International Conference on Machine Learning*, pages 22188–22214. PMLR, 2023.

Colin Lockard, Prashant Shiralkar, and Xin Luna Dong. Openceres: When open information extraction meets the semi-structured web. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3047–3056, 2019.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Piotr Nawrot and Edoardo Maria Ponti. nanoSparseAttention: The simplest implementation of recent sparse attention patterns for efficient LLM inference. `https://github.com/PiotrNawrot/nano-sparse-attention`, 2024. Accessed: 2026-05-01.

Piotr Nawrot, Szymon Tworkowski, Michał Tyrolski, Łukasz Kaiser, Yuhuai Wu, Christian Szegedy, and Henryk Michalewski. Hierarchical transformers are more efficient language models. *arXiv preprint arXiv:2110.13711*, 2021.

Piotr Nawrot, Jan Chorowski, Adrian Łańcucki, and Edoardo M Ponti. Efficient transformers with dynamic token pooling. *arXiv preprint arXiv:2211.09761*, 2022.

Piotr Nawrot, Robert Li, Renjie Huang, Sebastian Ruder, Kelly Marchisio, and Edoardo M Ponti. The sparse frontier: Sparse attention trade-offs in transformer llms. *arXiv preprint arXiv:2504.17768*, 2025.

Artidoro Pagnoni, Ramakanth Pasunuru, Pedro Rodriguez, John Nguyen, Benjamin Muller, Margaret Li, Chunting Zhou, Lili Yu, Jason E Weston, Luke Zettlemoyer, et al. Byte latent transformer: Patches scale better than tokens. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 9238–9258, 2025.

Denis Paperno, Germán Kruszewski, Angeliki Lazaridou, Quan Ngoc Pham, Raffaella Bernardi, Sandro Pezzelle, Marco Baroni, Gemma Boleda, and Raquel Fernández. The lambada dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.

Guilherme Penedo, Hynek Kydlíček, Anton Lozhkov, Margaret Mitchell, Colin A Raffel, Leandro Von Werra, Thomas Wolf, et al. The fineweb datasets: Decanting the web for the finest text data at scale. *Advances in Neural Information Processing Systems*, 37:30811–30849, 2024.

Qwen. Qwen3-next: Towards ultimate training & inference efficiency, 2025. URL https://qwen.ai/blog?id=4074cca80393150c248e508aa62983f9cb7d27cd&from=research.latest-advancements-list. Accessed: 2025-09-18.

Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, Chloe Hillier, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SylKikSYDH.

Pranav Rajpurkar, Robin Jia, and Percy Liang. Know what you don't know: Unanswerable questions for squad. *arXiv preprint arXiv:1806.03822*, 2018.

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Kevin Slagle. Spacebyte: Towards deleting tokenization from large language modeling. *Advances in Neural Information Processing Systems*, 37:124925–124950, 2024.

Jiaming Tang, Yilong Zhao, Kan Zhu, Guangxuan Xiao, Baris Kasikci, and Song Han. Quest: Query-aware sparsity for efficient long-context llm inference. *arXiv preprint arXiv:2406.10774*, 2024.

Finbarr Timbers. Large language models aren't trained enough. https://finbarr.ca/llms-not-trained-enough/, February 2023. Accessed: 2025-11-25.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Pavlo Vasylenko, Hugo Pitorro, André FT Martins, and Marcos Treviso. Long-context generalization with sparse attention. *arXiv preprint arXiv:2506.16640*, 2025.

Roger Waleffe, Wonmin Byeon, Duncan Riach, Brandon Norick, Vijay Korthikanti, Tri Dao, Albert Gu, Ali Hatamizadeh, Sudhakar Singh, Deepak Narayanan, et al. An empirical study of mamba-based language models. *arXiv preprint arXiv:2406.07887*, 2024.

Dustin Wang, Rui-Jie Zhu, Steven Abreu, Yong Shan, Taylor Kergan, Yuqi Pan, Yuhong Chou, Zheng Li, Ge Zhang, Wenhao Huang, et al. A systematic analysis of hybrid linear attention. *arXiv preprint arXiv:2507.06457*, 2025.

Kaiyue Wen, Xingyu Dang, and Kaifeng Lyu. Rnns are not transformers (yet): The key bottleneck on in-context retrieval, 2024. URL https://arxiv.org/abs/2402.18510.

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated delta networks: Improving mamba2 with delta rule. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=r8H7xhYPwz.

Lili Yu, Dániel Simig, Colin Flaherty, Armen Aghajanyan, Luke Zettlemoyer, and Mike Lewis. Megabyte: Predicting million-byte sequences with multiscale transformers. *Advances in Neural Information Processing Systems*, 36:78808–78823, 2023.

Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, YX Wei, Lean Wang, Zhiping Xiao, et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*, 2025.

Manzil Zaheer, Guru Guruganesh, Kumar Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big bird: Transformers for longer sequences. *Advances in neural information processing systems*, 33:17283–17297, 2020.

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

A Related Work

Method	Unrestricted Access to Memory?	Flexible memory?	Scalable training?	Both compute & memory efficient?	Controllable inference costs?	Mixable with CAT?	Usable in CAT's decoder?
Dense: Vaswani et al. (2017)	✓	✓	✓	✗	✗	✓	✓
Sparse Attention: Child et al. (2019)	✗	✓	✓	✓	✗	✓	✓
NSA: Yuan et al. (2025)	✓	✓	✓	✗	✗	✓	✓
Sliding window Attn.: Jiang et al. (2023)	✗	✗	✓	✓	✗	✓	✓
Linear Attention: Dao and Gu (2024)	✓	✗	✓	✓	✗	✓	✗
Recursive compression: Chevalier et al. (2023)	✓	✓	✗	✓	✗	✓	✓
MegaByte/Block Transformer: Ho et al. (2024); Yu et al. (2023)	✓	✗	✓	✓	✗	✓	✓
CATs	✓	✓	✓	✓	✓	✓	✓

Table 6: We categorize existing related work by key properties desirable for an efficient architecture, and indicate whether CAT can complement these approaches as a **meta-sequence mixer**. *"Both compute and memory efficient?"* signifies savings during inference; *"Unrestricted Access to Memory"* signifies whether an architecture can freely access any part of the memory in the past, without any artificial restrictions; *"Mixable with CAT?"* indicates whether CAT as a layer be used in these approaches; *"Usable CAT's compressor/decoder?"* indicates whether the method can serve as a compressor or decoder within CAT. Note that CAT itself can be recursively used as compressor/decoder.

Efficient sequence mixers: Sparse or sliding window attention (Child et al., 2019; Zaheer et al., 2020; Jiang et al., 2023) heuristically restrict which tokens are attended to. This reduces compute (and sometimes memory), but if the wrong mask is chosen, these methods underperform or require more depth (Arora et al., 2024a). Matching dense transformer quality often requires large windows or composition with dense attention at specific layers (Arora et al., 2024a; Agarwal et al., 2025). Linear attention (Katharopoulos et al., 2020; Arora et al., 2024a; Dao and Gu, 2024; Yang et al., 2025) replaces softmax with kernelized attention, admitting a recurrent form with constant memory. Recent variants add data-dependent gating (Dao and Gu, 2024; Yang et al., 2025), but all require handcrafted state update rules. The fixed-size recurrent state struggles with long-range recall (Arora et al., 2024a; Jelassi et al., 2024; Wen et al., 2024), and making these mixers competitive requires careful composition with attention – a process that involves significant trial-and-error (Waleffe et al., 2024; Qwen, 2025; Wang et al., 2025). CAT complements any sequence mixer whose decoding cost depends on sequence length, since instantiating CAT's decoder with such a mixer reduces costs in that mixer due to the compressed sequence. Consequently, the only strict competitors are sequence mixers with length-independent costs, such as pure linear attention. Otherwise, any existing efficient sequence mixer can serve in CAT, or be used alongside CAT for test-time flexibility, including hybrid designs (App. B.12).

Training-free efficiency: Plethora of works have tackled reducing compute requirements of a transformer in a *post-hoc* manner i.e. after it has been trained using full-attention (also called *training-free* sparse attention) Nawrot et al. (2025); Li et al. (2024); Tang et al. (2024). However, because models are trained dense but run sparse, train-test mismatch can hurt quality. Moreover, many target only compute savings and *not* memory savings, which is the main bottleneck for long-contexts. That being said, these are complementary and can be applied directly to the decoder in CAT, providing additional savings besides what CAT already provides. We deliberately instantiated CAT with the most-basic and ubiquitous sequence mixer: dense attention for this purpose. For completeness, we provide a comparison with a training-free method Jiang et al. (2024) in App. B.16.

Mixture-of-Experts (MoEs) (Shazeer et al., 2017; Dai et al., 2024) take a different approach: they increase parameters in feed-forward layers via sparse computation, without increasing inference costs – making them complementary to any sequence mixer, including those that CAT utilizes in compressor and decoder.

Compressing past context: Recurrent compression (Rae et al., 2020; Chevalier et al., 2023) enables generation of longer sequences on limited compute and memory. However, sequential training is slow and memory-intensive, scaling poorly on modern hardware that favors parallelism. Training recurrent models also poses optimization challenges; Geiping et al. (2025) required careful recipes to prevent collapse when scaling up. Native Sparse Attention (NSA) (Yuan et al., 2025) attends to compressed chunks as well as few raw tokens, with compression happening in parallel at every layer. This is similar in spirit to CAT, but NSA retains the full KV cache for the entire context—yielding compute savings but no memory savings during inference. **Further, no method here provides test-time control of inference costs, which is the primary goal of CAT.**

Hierarchical transformers: Hourglass architectures (Nawrot et al., 2021; 2022; Slagle, 2024) downsample the sequence into coarse tokens, then upsample before decoding. This saves compute during training, but generation still requires memory accesses for all past tokens, especially *fine-grained* ones, which is the main bottleneck. MegaByte and Block Transformer (Ho et al., 2024; Yu et al., 2023) model sequences as independent chunks conditioned on a single compressed representation of the past. While this improves efficiency, the fixed-size bottleneck hurts recall even on simple tasks (see App. B.8).

Byte Latent Transformer (BLT) Pagnoni et al. (2025) utilizes an *hourglass* model to compress in an *data-dependent* way according to entropy, judged by a separately trained model. However, there is no mechanism to change inference costs in BLT if the same sequence is fed into the model. CAT is actually complementary, where the goal is to provide controllable inference-costs. In fact, one may instantiate the main network of BLT as CAT, and get both data-dependent compression and controllable efficiency in a single model.

In summary, CAT complements most existing (or future) approaches, can extend them, or be mixed with them to unlock test-time control of inference cost.

Adaptive architectures: Kusupati et al. (2022); Devvrit et al. (2023) learns representations during training time that can work at different granularity during test-time, yielding adaptivity to the learned architecture. However, coarser granularity of *Matryoshka* representations result in loss of language modeling performance (in terms of perplexity) Devvrit et al. (2023). That being said, one could apply similar approaches to CATs making them complimentary. CATs use the same high-level approach described in Beyer et al. (2023): learn a single model that can work for various patch sizes at once depending on the downstream use-case at test-time. However, Beyer et al. (2023) worked with image classification tasks; CATs deal with language modeling and generation.

B More experiments

B.1 Long-context modeling and understanding

Table 7: LongPPL Fang et al. (2025) (we report `log_loss`) and zero-shot evaluation on a suite of tasks from LongBench Bai et al. (2023) (upto 4K tokens) and common LM evals Gao et al. (2024). Refer to App. Table 8 and Table 11 for task-wise results, and to Section E.2 for details. All CATs are a single model, and perform competitively.

	LongPPL ↓		**LongBench ↑**	**LM Evals**[3] **↑**
Model	GovReport	PG19	Avg.	Avg.
Strict baselines				
Mamba2	4.71	5.21	8.0	43.7
GDN	4.59	5.02	8.9	43.5
GDN-2×	4.23	4.86	8.1	43.8
GDN-2× 2D	3.90	4.41	9.6	46.4
CAT-4	**2.96**	**4.20**	**13.9**	43.9
CAT-8	3.19	4.30	12.1	44.1
CAT-16	3.66	4.57	9.5	44.3
CAT-32	4.36	4.92	7.9	45.1
Complementary				
Dense	4.50	5.54	9.3	42.1
Sparse-8	5.09	5.54	9.3	43.2
GDN-H 1:1	4.55	5.33	9.0	43.0
GDN-H 1:4	4.49	5.04	11.6	42.9
GDN-H 1:5 G	4.31	4.94	12.2	43.7
GDN-H 1:5 G 2D	4.02	5.04	11.6	45.8

B.2 LongBench

	Single-doc QA		**Multi-doc QA**		**Few Shot**		**Avg.**
Model	QAS	MQA	HQA	2WMQ	TQA	TREC	
Dense	3.9	12.2	6.9	**10.8**	11.2	10.6	9.3
Sparse-8	5.1	11.0	7.0	10.6	10.5	5.6	9.3
Mamba2	4.1	11.9	**7.6**	7.6	9.0	7.6	8.0
GDN	**8.3**	**15.5**	6.0	7.9	7.4	8.3	8.9
GDN-2×	4.1	11.8	6.7	9.6	9.8	6.8	8.1
GDN-2× 2D	4.6	14.0	6.9	8.9	11.3	12.1	9.6
GDN-H 1:1	4.2	13.3	6.6	11.6	11.8	6.5	9.0
GDN-H 1:4	4.6	13.0	7.0	10.4	10.6	24.2	11.6
GDN-H 1:5 G	4.7	12.5	6.4	12.2	9.2	28.3	12.2
GDN-H 1:5 G 2D	3.9	12.9	8.3	9.1	11.9	23.7	11.6
CAT-4	5.6	12.7	7.4	9.9	12.1	**35.6**	**13.9**
CAT-8	5.5	11.0	6.1	8.0	**12.4**	29.5	12.1
CAT-16	4.3	14.1	6.1	5.6	10.5	16.6	9.5
CAT-32	4.7	11.0	7.0	6.6	10.0	8.3	7.9

Table 8: Zero-shot evaluation of baselines on suite of tasks from LongBench Bai et al. (2023) up to 4K tokens. Refer to Section E.2. CAT-4/8/16/32 are a single model.

B.3 Recall

B.4 RULER

[3]This evaluation only considers **short sequences** ($\leq$ 30 tokens on average), which is *not* indicative of long-context performance, as reported multiple times in the literature Fang et al. (2025); Bertsch et al. (2026). Our claims concern long-context recall and modeling.

Table 9: Zero-shot performance on real-world in-context recall tasks measured upto 4K sequence lengths. H and G stand for Hybrid and Global respectively. **Fig. 1b gives an inference costs matched comparison using throughput.** All CATs here are a single model, whose costs can be changed at *test-time* depending on downstream use-case.

Model	SWDE	FDA	Avg. ↑
Strict baselines			
Mamba2	13.5	4.5	9.0
GDN	18.0	6.8	12.0
GDN-2×	24.0	11.0	17.5
GDN-2× 2D	29.7	16.2	22.9
CAT-4	**49.1**	**45.1**	**47.1**
CAT-8	38.2	34.8	36.5
CAT-16	27.5	15.4	21.5
CAT-32	13.2	3.2	8.2
Complementary			
Dense	43.4	19.7	32.0
Dense D/2	32.0	22.0	27.0
Sparse-4	36.0	16.0	26.0
Sparse-8	20.9	6.0	13.0
GDN-H 1:1	44.0	17.8	31.0
GDN-H 1:1 D/2	17.0	3.2	10.0
GDN-H 1:4	33.0	20.0	27.0
GDN-H 1:5 G	38.0	25.0	32.0
GDN-H 1:5 G 2D	<u>46.0</u>	<u>36.0</u>	<u>41.0</u>

Table 10: Accuracy on RULER Hsieh et al. (2024) S-NIAH-N benchmark. All CATs are a single model.

	S-NIAH-N (↑)		
Model	**1K**	**2K**	**4K**
Strict baselines			
Mamba2	97.7	81.1	18.6
GDN	84.7	69.1	13.6
GDN-2×	78.0	61.4	29.0
GDN-2× 2D	99.3	97.0	72.0
CAT-4	96.0	<u>97.0</u>	**96.0**
CAT-8	90.0	93.0	<u>91.0</u>
CAT-16	76.0	72.0	70.0
CAT-32	60.0	37.0	31.0
Complementary			
Dense	96.0	92.0	43.0
Sparse-8	51.2	46.2	5.0
GDN-H 1:1	<u>99.0</u>	97.0	44.0
GDN-H 1:4	98.0	96.0	35.8
GDN-H 1:5 G	98.3	93.0	23.2
GDN-H 1:5 G 2D	**99.5**	**99.3**	43.8

B.5 Short-context language understanding evaluations

Model	HS↑	PQ↑	AE↑	AC↑	WG↑	OQA↑	Avg.↑
Dense	34.8	65.6	56.7	24.4	51.1	20.0	42.1
Sparse-8	35.6	66.8	57.3	25.4	51.1	22.8	43.2
Mamba2	36.1	67.0	59.2	26.5	51.9	21.6	43.7
GDN	36.1	66.8	58.7	25.2	51.6	22.8	43.5
GDN-2×	35.9	67.4	58.6	27.2	51.8	21.8	43.8
GDN-2× 2D	37.5	69.9	63.6	29.4	52.8	25.6	46.5
GDN-H 1:1	36.8	66.3	56.4	25.8	52.1	20.4	43.0
GDN-H 1:4	34.8	67.0	57.0	26.5	50.3	22.0	42.9
GDN-H 1:5 G	36.0	67.6	57.4	26.6	51.5	23.4	43.7
GDN-H 1:5 G 2D	38.6	70.1	62.3	27.8	52.7	23.6	45.8
CAT-4	35.6	66.4	59.5	27.1	51.5	23.4	43.9
CAT-8	35.4	66.8	60.1	27.4	51.3	23.6	44.1
CAT-16	35.5	67.3	60.2	27.0	52.0	23.8	44.3
CAT-32	35.9	68.2	61.0	27.0	53.6	25.0	45.1

Table 11: Zero-shot accuracy on common-sense reasoning benchmarks. However, note that these evaluations considers short sequences only ($\leq$ 30 tokens on average). Hence, we test language understanding on longer contexts in Table 7 on LongBench (Bai et al., 2023), LongPPL Fang et al. (2025) and test in-context recall on real world tasks (Arora et al., 2023b) in the main text. Section E.2 expands the acronyms in table 11.

B.6 Parameter-Matched Results

The main paper reports the parameter-matched CAT (**blue** dots) alongside the larger model in Figure 1b (in-context recall); Figure 5 does the same for the remaining long-context evaluations.

We acknowledge that both parameter count and inference cost are axes for model comparison. That being said, quality *at a given inference-cost* is the correct comparison criterion in the current age, where models spend more time doing inference as chat-assistants, coding agents and more – and what the end-user eventually cares about is quality for cost (parameter count matters in so far as they affect the inference cost). Read that way, the parameter-matched CAT is not a weaker version of the model in the main text: it is the *cheapest* model in the entire comparison at every chunk size, operating at throughputs that none of the 10 baselines reach. Comparing it *directly* against the baselines at matched parameters is therefore an apples-to-oranges comparison in CAT's disfavour on quality and in its favour on cost; the honest reading is that increasing CAT's decoder parameters (**blue** → **red**) improves quality without a proportional hit to throughput.

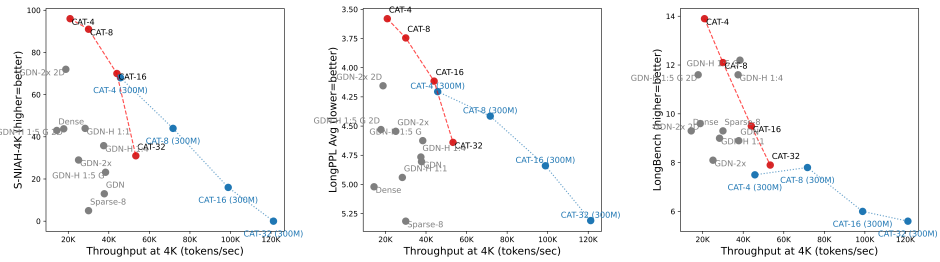


Figure 5: Quality–throughput trade-offs with the parameter-matched CAT (**blue**, ∼300M) plotted alongside the model of Figure 2 (**red**, ∼970M) and the baselines (gray). Each curve is a single model read out at chunk sizes $C = 4, 8, 16, 32$.

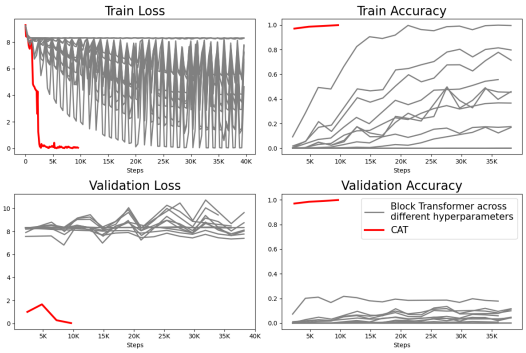


Figure 6: Block Transformer Ho et al. (2024); Yu et al. (2023) (across different configurations and hyperparameters) fails to solve a simple MQAR task with only 4 key-value pairs tested on modest sequence length of 256 tokens. Note that training of CAT stops when it solves the task perfectly.

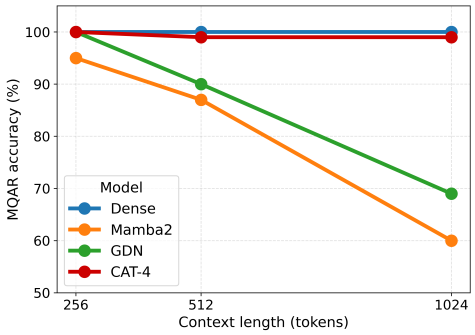


Figure 7: Comparison of different architectures across sequence lengths on MQAR task. We measure test-accuracy on the hardest subset. All architectures are memory matched in bytes at every point (except dense transformer).

B.7 Synthetic tasks

We begin by evaluating models on the multi-query associative recall (MQAR) task Arora et al. (2023a), a standard diagnostic benchmark for testing in-context recall. Our setup mostly follows Arora et al. (2024a): except we train and evaluate models on sequences $4\times$ the standard sequence length, upto 1024 tokens containing maximum amount of key-value pairs possible. We do not test length generalization here. We compare CAT with dense attention and linear methods, namely Mamba2 Dao and Gu (2024) and Gated DeltaNet Yang et al. (2025). Crucially, we ensure all models get equal memory (or state size) down to the level of bytes at each sequence length (except dense) to test recall-memory trade-offs. For linear models, this required scaling up state sizes to match CAT at every sequence length. All models were grid-searched for best hyper-parameters. We specifically only compare with linear sequence mixers having length independent costs since they are the only strict baselines to CAT (we discuss this in next section). CAT maintains strong recall performance across all sequence lengths. We especially note the following:

> Sequence mixers like dense attention or linear recurrences take an *extreme* route for long-context modeling: either the memory keeps growing *aggressively*, or the memory is fixed. CATs take a **middle-ground**, where the memory increases *gracefully* by constant factor less than dense. This flexible memory learns long-context correlations better, resulting in better recall despite taking the same memory (better trade-off) compared to existing fixed-memory sequence mixers.

To rule out any memory discrepancy, (Fig. 4) evaluates on MQAR task (Arora et al., 2023a), matching memory budgets down to the level of bytes, and stress-tests models up to 1K sequence length ($5\times$ standard); Figure 7 in reports results. Baselines are grid-searched over learning rates. Linear models collapse at longer contexts, while CATs remain near-perfect, thanks to the flexible yet efficient memory scaling. We use the same setup in App. E.4.

B.8 Comparison with MegaByte/Block Transformer

The MegaByte/Block Transformer (Ho et al., 2024; Yu et al., 2023) has elements similar to CAT but fail to solve a simple in-context recall task in fig. 6 across different hyperparameters and architecture configurations due to the fixed memory bottleneck. In fact, the block transformer overfits on the task. CATs alleviate the memory bottleneck with a gracefully growing memory, allowing it to solve the task, with even lower memory requirements.

In figure 6, we evaluate in-context recall ability for Block Transformer architectures Ho et al. (2024); Yu et al. (2023), that model chunks of tokens similar to CATs but with a subtle but salient difference in the architecture circuit (that we explain below). For this experiment, we test on the MQAR task (a synthetic needle-in-haystack task Arora et al. (2023a)) on a modest sequence length of 256. We test the accuracy of retrieving just 4 needles. We parametrize components of Block Transformer that is: global model and local model using a transformer, the embedder is a look-up table or

a transformer. We keep the patch size/chunk size as 4 – same as CAT. We keep the identical training setup for both architectures. We grid search for hyper-parameters (`lr`, `hidden_size`, and embedder parameterization), even **using more memory** than the CAT baseline, in its global decoder. Even in these simple settings and added advantage, Block Transformer Ho et al. (2024); Yu et al. (2023) fails to solve the task (fig. 6) – instead the model starts to memorize the train points, as seen from train loss and train accuracy – train metrics keep getting better, however, test metrics suffer.

CATs directly pass all the "local" patch/chunk representations directly to the decoder, unlike the block transformer that forces the history to be compressed into fixed dimensional representation. This design choice helps CAT *alleviate the memory bottleneck* that Ho et al. (2024) suffers from where the architecture must compress everything from the past into a single "global" representation to generate the next chunk. Note that this different design choice in CATs does not introduce any memory/compute overhead compared to Block Transformer Ho et al. (2024), it just changes the circuit of the architecture. In fact, CATs don't utilize three different components (embedder, global decoder, local decoder) – it only uses a compressor and a decoder, reducing the design space and (significant) parameter requirements further.

B.9 RULER benchmark

Table 10 (in the main text) reported results on RULER Hsieh et al. (2024) single-needle tasks: S-NIAH-N (recall number from the context). We observed linear recurrent models (Mamba2, GDN) struggle at longer contexts, and while GDN-Hybrid narrows the gap with dense transformers, performance still drops at longer contexts. CATs-4/8/16 outperform the efficient baselines as context length increases, showing slower degradation with length, even compared to the dense transformer. This slow degradation can possibly be attributed to reduced sequence length in CAT that leads to fewer *distractions* for attention (Barbero et al., 2024; Vasylenko et al., 2025; Chiang and Cholak, 2022; Golovneva et al., 2025). Further, large-chunk CAT underperforms at short contexts but interestingly surpass baselines at long ones. One reason why large-chunk CAT underperforms could be due to ineffective compression – due to larger chunks, the compressor in CAT is not always able to surface the *right* information in the chunk representation for accurate retrieval. More pre-training or finetuning on specific task data alleviates this problem for large chunk CAT (see App. B.15). That being said, there is an upper limit to how much information fixed sized chunk representations can practically learn to hold for large token chunks.

Table 12 further reports results on the harder S-NIAH-U (recall a long alpha-numeric string or UUID).

Table 12: Accuracy on RULER Hsieh et al. (2024) S-NIAH-U benchmark.

	S-NIAH-U		
Model	**1K**	**2K**	**4K**
Dense	**93.6**	55.7	19.8
Sparse	12.8	1.4	0.8
Mamba2	46.7	4.6	1.0
GDN	38.9	2.6	2.0
GDN-H 1:1	50.9	5.6	2.6
CAT-4	<u>79.6</u>	**59.3**	<u>46.5</u>
CAT-8	68.1	<u>57.5</u>	**47.3**
CAT-16	10.0	6.6	3.8
CAT-32	0.0	0.0	0.0

B.10 Recall evaluation

Here, we evaluate all baselines on all datasets from the EVAPORATE suite of tasks that tests for real-world in-context recall.

Model	SWDE	FDA	Squad	TriviaQA	Drop	Avg.
Dense	43.4	19.7	<u>31.0</u>	15.0	<u>19.4</u>	<u>26.7</u>
Sparse	20.9	6.0	20.7	15.2	19.3	16.4
Mamba2	13.5	4.5	24.9	13.9	17.8	14.9
GDN	18.0	6.8	25.5	**15.5**	17.2	16.6
GDN-H 1:1	44.0	17.8	**32.9**	<u>15.4</u>	**19.8**	26.0
CAT-4	**49.1**	**45.1**	28.3	15.0	17.9	**31.1**
CAT-8	<u>38.2</u>	<u>34.8</u>	25.9	14.0	18.3	26.2
CAT-16	27.5	15.4	20.4	14.8	16.9	18.9
CAT-32	13.2	3.2	15.8	13.0	14.3	11.9

Table 13: Zero-shot performance on real-world in-context recall tasks from EVAPORATE suite, measured upto 4K sequence lengths. Note that only SWDE and FDA have long token sequences among the datasets in the suite (others have an average length of ≤ 300 tokens Arora et al. (2024b)).

B.11 CAT as a layer

While CAT presented in the paper is a separate meta-sequence mixer, one can take the core concepts and instantiate CAT as a layer that can be swapped in any sequence model as a drop-in replacement. This can unlock lots of interesting possibilities starting with creating hybrid as well as adaptive architectures that mixes CAT layers alongside dense attention, or perhaps even linear attention. We leave this open for future work.

To instantiate CAT as a seperate layer in itself, we parameterize the *compressor* as a simple linear projection. We use the dense attention mechanism itself as the *decoder*. Before applying the compression and decoding from compressed chunk representations, we artificially up-project the input embeddings in the layer – this is done following the observation in the main paper that decoding from compressed representations requires higher dimensionality. We will release the implementation in our code. Table 15 reports MQAR accuracy when CAT is used as a layer. We use a fixed chunk size of 4 in this experiment. We use 2 layers of CAT. Rest of the setup follows Arora et al. (2024a).

Method	Solves?	State Size
Dense	✓	16384
CAT	✓	4096
CAT (layer)	✓	4096

Table 14: CAT instantiated as a seperate layer solves the MQAR task.

B.12 CAT is a meta sequence mixer

CAT has two components: a compressor and a decoder – each of these could make use of any sequence mixers, such as linear attention. Here we provide preliminary result on the MQAR task where the decoder in CAT is a GDN-HYBRID architecture (Yang et al., 2025) (having a 1:1 dense-to-linear ratio). This new architecture solves this task, empirically demonstrating the use of a hybrid sequence mixer inside of CAT: meaning rather than CAT being a strict competitor to hybrids, **CAT is complementary**. Further, the use of a different sequence mixers inside of CAT can unlock the test-time control of efficiency with those sequence mixers (e.g., GDN-HYBRID in this case).

CAT solves the MQAR task when the decoder is instantiated as a GDN-Hybrid architecture (1:1 dense-linear ratio) with 2 layers. We use the same setup described in Arora et al. (2024a): using sequences upto 256 with maximum key-values in the sequence. The chunk size used is set to 4.

Method	Solves?
Dense	✓
CAT	✓
CAT (GDN-HYBRID decoder)	✓

Table 15: The decoder in CAT is replaced with a GDN-HYBRID architecture. The resulting CAT architecture solves the MQAR task.

Further, we train a CAT variant with GDN-1:1 as the decoder on 5B tokens of FineWeb at 1K context. CAT reduces memory cost when applied on top of GDN-H 1:1, and the resulting model even outperforms CAT with a dense attention decoder – further evidence that CAT and efficient sequence mixers are complementary.

B.13 CATs scale as well as their dense counterparts

Figure 8 demonstrates that CATs scale similar to their dense transformer equivalents. We evaluate against three dense transformer scales $\{31M, 92M, 260M\}$, with their CAT equivalents containing parameters $\{95M, 326M, 1B\}$. All models were trained for 15B tokens, under the setup in section 4.

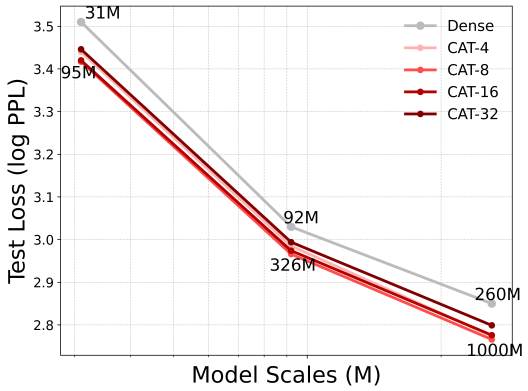


Figure 8: CATs scale like their dense transformer counterparts while being up to $3\times$ faster and $9\times$ more memory-efficient. All CAT curves come from a single model, evaluated at different chunk sizes. **Note that while CAT occupies more parameters, it is still both compute and memory efficient compared to the densee transformer at every scale.**

B.14 Across chunk analysis

We provide how the validation loss changes within a chunk in Figure 9. We provide averaged results across all chunks. We provide different curves for each chunk size.

Interestingly, across all chunk sizes, the loss is highest when decoding the first token from the compressed representations only. After that token is decoded, the loss decreases steadily as CAT keeps decoding tokens from both compressed representation and raw tokens that appear before inside the chunk.

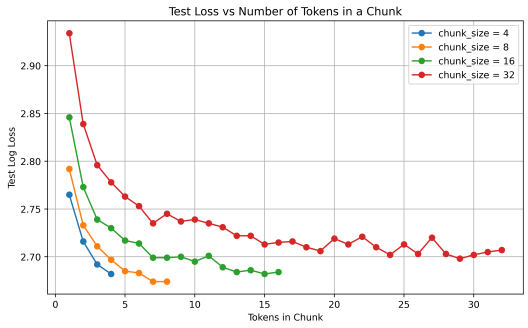


Figure 9: Chunk Analysis

B.15 Finetuning CATs on S-NIAH-U

S-NIAH-U is a task where model needs to recall 32 token long UUID strings from the long context. This section reports performance of CATs after task specific finetuning on samples from S-NIAH-U. We only apply the loss on tokens that appear in the answer span. Table 16 reports these results. This is accompanied by loss curves for different CATs depending on chunk size in Figure 10 on this task.

We observe two things: (i) after finetuning, performance goes up significantly for all chunk sizes. This signifies as chunk size increased, compressor in CATs, before finetuning, was not surfacing the *right* information in the chunk representation. (ii) the loss curves during finetuning indicate the same as well, however it still does not go completely to zero, especially for CAT-32. This indicates that there are limits to what information a fixed sized chunk representation can practically learn to surface, justifying its sub-par accuracy on the task.

This problem of not surfacing the *right* information in the chunk representation could be alleviated by more and longer pre-training, or choosing smaller chunk sizes for tasks that require accurate recall.

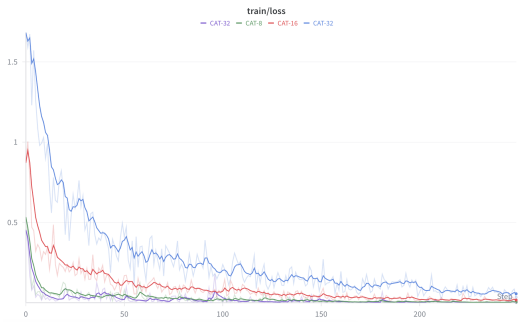


Figure 10: Loss curves when finetuning different CATs on samples from S-NIAH-U task.

Model	Before	After
CAT-4	46.5	97.1
CAT-8	47.3	97.0
CAT-16	3.8	94.2
CAT-32	0.0	64.3

Table 16: Performance on 4K sequence length before and after finetuning for different CAT variants.

B.16 Comparison with Training-Free Block-Sparse Attention

An interesting comparison is with training-free block-sparse attention, where the past KV cache is chunked into blocks and top-K block selection is applied at inference time Jiang et al. (2024). We emphasize that this approach is *complementary* to CAT rather than a direct competitor: training-free block-sparse attention yields only *compute* savings, while CAT is designed to deliver both *compute and memory* savings during inference (see Table 1 for a detailed comparison with closely related works). In long-context inference regimes on modern hardware, memory savings are typically the more critical bottleneck, since available FLOPs substantially outpace available GPU memory.

Nevertheless, for completeness, we evaluate training-free block-sparse attention on the SWDE recall task. We chunk the past KV cache into blocks and perform top-K block selection Jiang et al. (2024) with the reference implementation from the `nano-sparse-attention` repository Nawrot and Ponti (2024). We fix `block_size` $= 32$ and sweep the top-K parameter. We also include the trainable strided sparse attention models Child et al. (2019) from Table 9 for reference.

For block-sparse attention, the per-token FLOPs reduction is $\mathcal{O}(N/(\texttt{block_size} \cdot \texttt{top_K}))$ with sequence length $N = 4096$ and `block_size` $= 32$. For CAT, both the per-token FLOPs reduction and the memory reduction are $\mathcal{O}(N/\texttt{chunk_size})$.

Method	Theoretical Memory Red.	Theoretical Per-Token FLOPs Red.	SWDE
Dense	1×	1×	43.4
Block-Sparse Attention (Training-Free)			
top-32	1×	4×	43.0
top-16	1×	8×	34.4
top-8	1×	16×	18.9
top-4	1×	32×	8.0
Trained Strided Sparse Attention			
Sparse-4	4×	4×	36.0
Sparse-8	8×	8×	20.9
CAT (Ours)			
CAT-4	4×	4×	**47.1**
CAT-8	8×	8×	**36.5**
CAT-16	16×	16×	**21.5**
CAT-32	32×	32×	**8.2**

Table 17: Comparison of CAT against training-free block-sparse attention and trainable strided sparse attention on the SWDE recall task ($N = 4096$). Training-free block-sparse attention provides only FLOPs savings ($1\times$ memory reduction), whereas CAT achieves both compute and memory savings at a better recall–cost trade-off.

Takeaways. Two observations follow from Table 17:

- Training-free block-sparse attention provides FLOPs savings but *no memory savings*, since the full KV cache must still be retained for top-K block selection.
- CAT achieves *both* compute and memory savings, and attains a strictly better recall–cost trade-off across all matched compression ratios, a benefit we attribute to end-to-end training of the compression and decoding, and thus not resulting in a train-test mismatch that happens with *training-free* approaches.

This further supports our view that block-sparse attention is complementary to CAT rather than a competing baseline: assuming CAT's decoder uses a sequence mixer with length-dependent cost (as in our default dense attention decoder), block-sparse selection could in principle be layered on top of CAT to obtain additional FLOPs reductions on the already-compressed KV cache.

C Implementation details and PyTorch style pseudo-code

In this section, we discuss some implementation details regarding CATs. We repeat some text presented in the main paper to be self-contained below.

C.1 Training

Training: While CATs are simple and build on dense transformer abstractions, their naive PyTorch training implementation is very inefficient.

Note that compression of chunks of tokens is efficient since it can be done in parallel, specifically using `torch.vmap`$(f_\theta(\mathbf{c}_i))$ for all chunks $\mathbf{c}_i$. This costs a total of $O(\frac{N}{C} \cdot C^2) = O(NC)$ in self-attention compute, which is much better than $O(N^2)$.

But, computing logits for tokens in chunk $\mathbf{c}_i$, that is computing $g_\theta(\mathbf{c}_i \mid f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1}))$ can be non-trivial since for chunk $\mathbf{c}_i$, we have $i-1$ past chunk representations $\{f_\theta(\mathbf{c}_1), f_\theta(\mathbf{c}_2) \dots f_\theta(\mathbf{c}_{i-1})\}$. In other words, there are different number of past chunk representations for every chunk, making shapes variable and as a result, harder to parallelize computation of logits. One could employ a python loop and compute logits for every chunk sequentially, but that would be slow and won't scale. In fact, even if one manages to compute logits for every chunk in parallel, the total self-attention operations in the decoder would be $O(\sum_{i=1}^{\frac{N}{C}}(i+C)^2) = O((\frac{N}{C})^3)$, that is cubic in sequence length. Padding to make shapes constant would make things worse. Thus, naive techniques will not scale.

With such difficulties in making the training scalable, it may not be surprising that despite the simplicity of CATs*, it was not attempted in the community.* Note that unlike CATs, similar architectures Ho et al. (2024); Yu et al. (2023) do not have this problem: computing logits can be naively parallelized due to fixed shapes and self-attention operations scale quadratically due to a single compressed representation for the past.

In CATs, observe that in computing logits chunks $\mathbf{c}_i, \mathbf{c}_{i+1} \dots \mathbf{c}_{\frac{N}{C}}$, one calculates the same key-values for chunk representations $f_\theta(\mathbf{c}_j)$ in the decoder, where $j < i$. This points to repeated and identical computations. To exploit this observation, we take advantage of a custom attention mask in decoder to calculate logits for all chunks in parallel, and reuse computations done for a past chunk representation to be used for a computations for logits for a future chunk. To be concrete, once we calculate all chunk representations $f_\theta(\mathbf{c}_i)$ in parallel using `torch.vmap`, we insert $f_\theta(\mathbf{c}_i)$s at particular positions in the original sequence: after every chunk $\mathbf{c}_i$, we attach its chunk representation. That is, sequence would look like: $\{\mathbf{c}_1, f_\theta(\mathbf{c}_1), \mathbf{c}_2, f_\theta(\mathbf{c}_2), \dots \mathbf{c}_i, f_\theta(\mathbf{c}_i) \dots\}$. Now, we pass this sequence into the decoder during training, with a custom attention mask (see Figure 11) that allows a token in chunk $\mathbf{c}_i$ to attend to previous tokens within that chunk only as well as only to previous chunk representations, which would be $f_\theta(\mathbf{c}_{i-1}), f_\theta(\mathbf{c}_{i-2}) \dots f_\theta(\mathbf{c}_1)$ only. Any token in chunk $\mathbf{c}_i$ does not attend to raw tokens outside this chunk. This implementation allows re-use of key-values for chunk representations $f_\theta(\mathbf{c}_i)$ for calculation of logits of future chunks, in parallel, making the training of CATs efficient and scalable. We utilize the FlexAttention API Dong et al. (2024) to automatically create a custom kernel for the custom mask (Figure 11). Note that this way of computing logits is quadratic in sequence length but with a constant times better: concretely it is $O(\frac{N}{C} \cdot N + \frac{N}{C} \cdot C^2) = O(\frac{N^2}{C})$, **which is $C\times$ better than $O(N^2)$** (yellow dots in figure 11 provides a visual proof for this cost; number of yellow dots are significantly lower than $\frac{N^2}{2}$). Mathematically the cost of attention in CATs decoder is: $\sum_{i=1}^{N}[\frac{i}{C}] + (i \bmod C) + 1 = O(\frac{N^2}{C})$, where [.] is the floor function, and mod is modulo operator.

For a discussion in training throughput, refer to a discussion in Appendix C.5.

```
def forward(input_ids, targets):

    input_ids = einops.rearrange("b (k c) -> b k c", k=num_chunks, c=chunk_size)

    # calculate f(x)
    # shape of fx: (b, k, D_d)
    fx = torch.vmap(f)(input_ids)

    output_logits = list()
```

```
    for i in range(num_chunks): # note that this loop is done in parallel with the
         custom attention mask presented in the appendix
        # use the previous i+1 fx to predict the current chunk
        # shape of cur_chunk_logits: (b, 1, l, V)
        cur_chunk_logits = phi(input_ids[:, i, :], fx[:, :i+1, :])
        output_logits.append(cur_chunk_logits)
    output_logits = torch.cat(output_logits, dim=1) # shape: (b, k, c, V)
    output_logits = einops.rearrange(output_logits, "b k c v -> b (k c) v") #
        arrange all chunks logits together (or flatten)
    return torch.nn.functional.cross_entropy(output_logits, targets) # return the
        loss
```

Listing 1: Pseudocode for training step

C.2 CAT's training attention mask

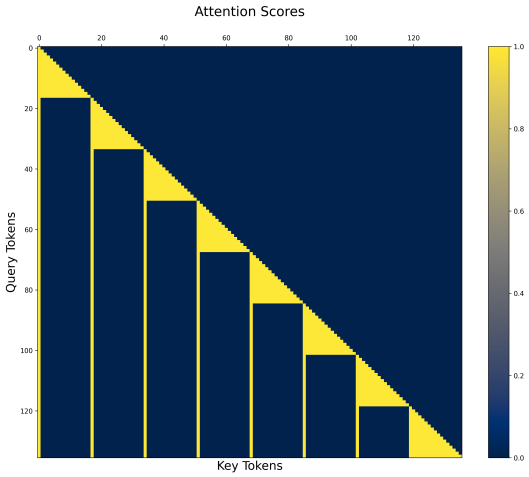


Figure 11: Sequence length is 128, and the chunk size that we use in this particular attention mask is $C = 16$.

Note that attention mask in figure 11 looks very similar to the attention mask as defined in Child et al. (2019), however, in CAT's case: (a) it is not heuristic choice, and (b), tokens in a particular chunk attend to the past $f_\theta(\mathbf{c}_i)$ representations obtained by the compressor, rather than the past token embeddings at that position as done in Child et al. (2019).

C.3 Generation

The decoder during generation attends to atmost $\frac{N}{C} + C$ tokens. Due to compression, CATs can throwaway past chunks of tokens, and only keep their compressed chunk representations in memory. This straightaway results in a big reduction of memory; the KV cache is slashed by a factor of C. For even a moderate chunk size of 4, this results in big reductions in memory during generation (Figure 4) compared to a dense transformer. This slash in memory is accompanied by reduced memory accesses a decoder makes in CATs, which is the major bottleneck during generation. Costs for self-attention in CATs decoder scale as $O(\frac{N^2}{C})$, which is again, $C\times$ better than $O(N^2)$ for a dense transformer.

Implementing generation is simpler than training and very similar to how it occurs for a dense transformer. In fact, a pure PyTorch implementation for CATs is on-par with efficient architectures that utilize custom kernels. We inspire our implementation from: `https://github.com/meta-pytorch/gpt-fast`. Given i chunks of tokens: firstly, `torch.vmap` over chunks independently to calculate $f_\theta(\mathbf{c}_i)$ in parallel. Then prefill the decoder's KV cache in parallel with the obtained $f_\theta(\mathbf{c}_i)$s. Now generate the next chunk $\mathbf{c}_{i+1}$ autoregressively one token at a time. Note that this uses a simple causal mask since the previous positions are already prefilled with $f_\theta(\mathbf{c}_i)$s, which is required to decode chunk $\mathbf{c}_{i+1}$. Once all the tokens of the chunk $\mathbf{c}_{i+1}$ are generated, calculate $f_\theta(\mathbf{c}_{i+1})$ and prefill the decoder's KV cache just after the position where $f_\theta(\mathbf{c}_i)$ was cached. Now the KV cache is ready for generation of the next chunk $\mathbf{c}_{i+2}$ and this process will continue.

This simple implementation enables CATs to be $1.4 - 3.2\times$ **faster** than the dense transformer while showcasing **upto** $2.2 - 9.5\times$ **lower total memory usage** as one increases chunk sizes.

```

# https://github.com/pytorch-labs/gpt-fast/blob/7dd5661e2adf2edd6a1042a2732dcd3a94064ad8/generate.py#L154
def generate_chunk_by_chunk(
    input_ids
):
    # assume input_ids.shape == (batch_size, 1, chunk_size)

    # declare/reset static KV cache, shape: [batch_size, num_chunks + chunk_size, 2, D_d]

    input_pos = 0

    # compress the first chunk (batch_size, 1, chunk_size) -> (batch_size, 1, D_d)
    # get fx for the very first chunk
    fx = f(input_ids) # shape of fx: (batch_size, 1, D_d)
    next_token = prefill(fx, input_pos) # prefill at idx 0 with fx in phi

    new_chunks = list()

    for i in range(num_chunks - 1):

        # generate entire chunk using fx that was prefilled earlier in phi
        next_chunk = generate_chunk(next_token)
        new_chunks.append(next_chunk.clone())

        # get new fx
        # compress the new obtained chunk
        fx = f(next_chunk) # (batch_size, 1, chunk_size) -> (batch_size, 1, D_d)

        # prefill again at input_pos
        input_pos += 1
        next_token = prefill(fx, input_pos) # prefill fx at idx `input_pos` in phi

    new_chunks = torch.cat(new_chunks)
    return new_chunks
```

Listing 2: Pseudocode for generation

C.4 Adaptive CATs training details

To enable training of adaptive CATs, we made some choices that we now describe. In every training iteration, we sample a chunk size uniformly at random and perform loss computation. Further, due to variable size of a chunk in every training iteration, one cannot keep a single projection matrix that projects processed token embeddings in the compressor to a single chunk representation (since shapes for projection matrix would be different for different chunk size). One could tackle this by keeping an independent projection matrix for every chunk size, but we found this didn't work well empirically, possibly due to reduced updates for every chunk size's projection weights (only one chunk size's projection weights are updated per iteration; this is not the case with compressor or the decoder, they are updated every iteration). Instead, we took inspiration from Beyer et al. (2023) where the authors declared a single projection matrix for all chunk sizes, and then linearly interpolated the matrix to the desired shape depending on the current chunk size. This means the linear interpolation is also under `torch.autograd` and is optimized so that the final linearly interpolated projection matrix gives a *good* chunk representation for every chunk size.

C.5 CAT's training throughput analysis

We make use of FlexAttention API to obtain a custom self-attention kernel specifically for the masking scheme section 11. This fused kernel gives a significant boost in training throughput in self-attention costs compared to using a naive PyTorch masked implementation.

MLPs in a transformer drive the majority of the FLOPs budget during training at smaller sequence lengths Austin et al. (2025). At a sequence length of 4096, CATs take $\leq 2.35\times$ to train compared to a dense transformer (measured on batch size of 8 with compressor depth of 3, decoder depth of 6, hidden size for compressor $D = 1024$ and hidden size for decoder $D_g = 2D = 2048$ for CAT, compared against dense transformer having depth of 6 and $D = 1024$, on a A100 80 GB PCIe.) At 16K sequence length, this gap reduces significantly and CAT only costs $\leq 1.25\times$ more.

Despite this, CAT amortizes this overhead in two ways: (i) through cheaper inference, which dominates lifetime cost, and (ii) by replacing multiple models with one — training independent models to cover the same range of inference budgets would require separate pretraining runs for each operating point, costing more than a single CAT.

C.6 Time taken by each CAT component during generation

Here we measure time taken by each CAT component during generation, specifically time taken by: decoder attention, decoder FFNs, and time taken by parallel compression. Section C.6 provides these results. We use the same setup in benchmarking as described in Section 4. We use a chunk size of 8 for this ablation.

Component	Time (ms)	Percentage (%)
Attention in Decoder	30,817	70.1
FFN in Decoder	11,555	26.3
Compression in Compressor	1,551	3.5
Total	43,932	100.0

D Some ablations on the CAT architecture

D.1 Ablation on hidden size of compressor

With this ablation, we show that increasing hidden size of the compressor does not help in improving perplexity. We fix $D_g = 1536$ for these experiments. For this ablation, we use a smaller WikiText-103 dataset. Both compressor and decoder use the same depth $L = 6$.

Chunk Size C	**Size of** D_f	**Perplexity**
16	768	17.6
	1536	17.6

Table 18: Comparison of choices of hidden size of compressor on WikiText-103 perplexity.

There is no effect of increasing the hidden size of the compressor. The performance before and after remains the same.

D.2 Ablation on hidden size of decoder

We ablate on different choices of D_g along with different chunk sizes in CAT . In this setup, we fix D_f in the compressor, and only vary D_g or C (chunk size). We use WikiText-103 for these experiments. In this setup, $D = 768$. Both compressor and decoder use the same depth of $L = 6$.

Chunk Size C	**Size of** D_g	**Perplexity**
4	D	19.8
	$2D$	17.4
8	D	20.4
	$2D$	17.7
16	D	20.2
	$2D$	17.6

Table 19: Comparison on choices of chunk sizes and sizes of D_g on WikiText-103 perplexity.

We observe that we obtain the best perplexities when we $D_g = 2D$ for the particular chunk size we are using. Using this observation, we used this as our *default* configuration for the FineWeb-Edu experiments.

Model	D_f	D_g	**Perplexity**	**Avg. recall**
Dense	$--$	D	21.2	23.8
CAT	D	D	23.8	13.7
CAT	D	$2D$	**20.7**	**19.8**

Table 20: Impact on perplexity and average recall performance of CAT when varying D_g. For dense, D_g implies hidden size for itself. Here, $D = 1024$. $D_g = 2D$ gives better perplexity and average recall. We train CAT only at chunk size $C = 8$ for these experiments. All models were trained for 5B tokens with 1K sequence length. Rest of the setup follows Sec. 4.

D.3 Ablation on depth of the compressor

We ablate on the depth of the compressor. For a fixed chunk-size, $D_f = 768$ (compressor embedding size), $D_g = 1536$ (decoder hidden size), and a fixed depth of the decoder, we vary the compressor depth.

Chunk Size C	Depth of Compressor	Perplexity
8	6	17.4
	3	17.4
16	6	17.8
	3	17.7

Table 21: Comparison on choices of depth of the compressor across different chunk sizes C on WikiText-103.

We have an interesting observation that one can reduce the depth of the compressor without sacrificing on the downstream perplexity. This could mean one can compress small chunks of tokens without a requiring high capacity. In our generation benchmarks, we observed that compressor depth play less of a role in latency as compared to the decoder depth (since we compress tokens in parallel using one transformer call). That being said, compressor depth does play a significant role in training costs (due to the MLP training costs in the compressor). Therefore, reducing compressor depth goes into overall advantage for the CAT architecture.

However, what is the limit, and can one go to even a 1 layer of compressor is an interesting question to ask. There might be some lower bound on the compressor depth to start compressing chunks of tokens, but we leave this to future work.

E More experiment details

Here we provide more details about the experiments done in the main text.

E.1 Baselines

In this section, we provide details about the models used in our experiments.

Model	Total (M)	Embedding (M)	Non-Embedding (M)
Dense	260	50	210
Dense-D/2	92	25	70
Mamba2	260	50	210
GDN	310	50	260
GDN-2x	310	50	260
GDN-Hybrid 1:1	280	50	230
GDN-Hybrid 1:1 D/2	111	25	86
GDN-Hybrid 1:4	280	50	230
GDN-Hybrid Global 1:5	300	50	250
GDN-Hybrid Global 1:5 2D	820	100	720
GDN-2x 2D	820	100	720
Sparse-4	820	100	720
Sparse-8	820	100	720
CAT-4/8/16/32	150 + 820	50 + 100	100 + 720

Table 22: Model parameter sizes in millions, separated into embedding and non-embedding parameters. Parameters for CATs consists of parameters in compressor + parameters in decoder.

By default, all models below are configured with $L = 12$ layers and $D = 1024$ hidden dimension; any deviations are explicitly stated.

1. Dense transformer (or Transformer++) Vaswani et al. (2017); Touvron et al. (2023): We use rotary position embeddings along with the FlashAttention kernel to perform self-attention. The MLP is a SwiGLU MLP Touvron et al. (2023). Dense-D/2 uses $2\times$ lower model dimension of $D = 512$.
2. Sparse transformer Child et al. (2019): Follows the Dense transformer configuration, except the attention mask used. Moreover, we used $D = 2 \cdot 1024 = 2048$ for this baseline for a fair comparison with CATs. We used FlexAttention API to create optimized Flash Attention like kernel for this. We use a stride length of 4 (Sparse-4) and 8 (Sparse-8) that tries to compete with CAT-4 and CAT-8 respectively.
3. MAMBA2 Dao and Gu (2024): The model uses 2 Mamba mixer per layer. All layers use the MAMBA2 block without any mixing any attention. The `expand` is set to 2, $d_{state} = 128$, and convolution $k = 4$. Activations used are `SiLU`. We use the official codebase for MAMBA2 generation throughput and memory benchmarking: `https://github.com/state-spaces/mamba` and code from: `https://github.com/fla-org/flash-linear-attention` for training.
4. GatedDeltaNet (GDN) Yang et al. (2025): We use the implementation provided at `https://github.com/fla-org/flash-linear-attention` for training. We use `head_dim` as 128 and `num_heads` as 8 (same as MAMBA2 above). GDN-2X stands for recurrent state size increased by $2\times$.
5. GatedDeltaNet Hybrids (GDN-H) Yang et al. (2025) We insatiate multiple GatedDeltaNet Hybrids models in our comparison. GDN-H 1:1 uses use sliding window layers at every other layer with a sliding window size of 2048. GDN-H 1:4 uses the same sliding window layers, but in the ratio of $1:4$ with linear attention. GDN-H 1:1 D/2 uses the same sliding window layers at every other layer, but the model dimension is scaled down by $2\times$ to $D = 512$.

   Finally, GDN-H 1:5 G uses linear-dense attention ratio as 1:5 but with a global attention. GDN-H 1:5 2D G uses linear-dense attention ratio as 1:5 with global attention and uses $2\times$ the hidden size.

E.2 Datasets

Following common practices done in Gu and Dao (2023); Dao and Gu (2024); Arora et al. (2024a); Yang et al. (2025), we evaluate all models on multiple common sense reasoning benchmarks: PIQA Bisk et al. (2020), HellaSwag Zellers et al. (2019), ARC-challenge Clark et al. (2018), WinoGrande Sakaguchi et al. (2021) and measure perplexity on WikiText-103 Merity et al. (2016)and LAMBADA Paperno et al. (2016). In Table 11, HS denotes HellaSwag, PQ denotes PIQA, AE denotes ARC-Easy, AC denotes ARC-Challenge, WG denotes Winogrande, OQA denotes OpenBookQA, LMB denotes LAMBADA, Wiki denotes WikiText, and FW denotes FineWeb-Edu.

We evaluate on tasks from LongBench Bai et al. (2023) where each abbrevation in table 8 stands for: QAS: `qasper`, MQA: `multifieldqa_en`, HQA: `hotpotqa`, 2WMQ: `2wikimqa`, TQA: `triviaqa`, TREC: `trec` split of LongBench.

To measure real-world recall accuracy, we use datasets used in Arora et al. (2024a;b). Namely these consists of SWDE Lockard et al. (2019) for structured HTML relation extraction and several question answering datasets including SQuAD Rajpurkar et al. (2018), TriviQA Joshi et al. (2017), DROP Dua et al. (2019) and FDA Arora et al. (2023b). Since our pretrained models are small, we use the Cloze Completion Formatting prompts provided by Arora et al. (2024b).

We evaluate on tasks from the needle-in-haystack benchmark RULER Hsieh et al. (2024).

Additionally, we evaluate on datasets from the LongBench benchmark Bai et al. (2023) to evaluate long-context understanding.

E.3 Generation

Both dense transformer and CAT use out-of-the-box FlexAttention API causal dot product kernels in PyTorch. We use the script provided in Dao and Gu (2024) to benchmark[4] Mamba2 and `flash-linear-attention` repo to benchmark GDN. All benchmarks used a prefill of 8 tokens. All benchmarks were run using a single NVIDIA A100 80GB PCIe, and use CUDA cache graphs for the next-token prediction.

To measure throughput, we measure the latency at increasing batch sizes (in powers of 2) and compute the maximum tokens per second a model can obtain on a fixed hardware. For our experiments, fixed hardware for throughput is a single A100 PCIe 80GB GPU.

E.4 MQAR setup

We evaluate models on the synthetic multi-associate query recall (MQAR) task, proposed in Arora et al. (2023a) and further popularized in Arora et al. (2024a). All models use depth of 2 layers, and are trained and tested on sequence lengths upto 1024 having the maximum number of key-value pairs possible. CAT models use a 1 layer compressor, followed by a 2 layer decoder, with a chunk size of 4, both using model dimension of $D = D_d = 64$ in this case.

We use the state size calculations provided in Arora et al. (2024a; 2023a).

E.5 Main figure details

Figure 1b reports results at 2K sequence length since both SWDE and FDA datasets have queries with context length upto 2K.

[4] github.com/state-spaces/mamba