Autoregressive Language Modeling using Compressed Sequence Mixing

Jatin Prakash [1] **Aahlad Puli** [1] **Rajesh Ranganath** [1]

Abstract

The transformer architecture is the default choice for large language models (LLMs) but the attention layers incur computational costs that scale quadratically with context length, which is prohibitive. To reduce these costs, many works propose alternative low-cost sequence mixers that approximate attention; for example, sparse or sliding window attention limits the inputs to attention and linear attention or convolutions limit the state size by removing or approximating the softmax transformation. These alternatives have limitations; e.g to solve tasks like multi-query associate recall, sparse-attention transformers need to be deeper than vanilla transformers and linear attention needs to be composed with self-attention. To build efficient LLMs without replacing the attention mechanism itself, we develop the Compress and Attend Transformer (CAT). CAT is a simple transformer-based architecture that decodes each token while only attending to compressed chunks of the sequence so far. The chunk-size limits the compressor cost and the compression reduces the costs for the decoder by a factor of the chunk size. It follows that CATs enjoy fast and memory-efficient generation, with upto $3\times$ generation throughput and $7\times$ less memory usage compared to a dense transformer. We show that CATs match dense transformer on perplexity and common language modeling evaluations. At the same time, CATs outperform existing efficient attention-alternatives on real-world recall benchmarks, showcasing similar generation throughput and memory usage.

1. Introduction

Transformers (Vaswani et al., 2017), due to their scalability and handling of long-range dependencies, are now the default architectures for large language models (LLMs). To handle long-range dependencies, transformers rely on the attention mechanism (Vaswani et al., 2017; Bahdanau et al., 2014) to aggregate information over the entire sequence. When inferring a token given a context consisting of the model input and some generated tokens, each layer of self-attention computes the inner products between the embeddings of the last token against every other token in the context. Thus, Transformers excel in recalling information from the context. However, the cost of computing attention, both in terms of compute and memory, increases with the size of the context.

Mitigating the inference cost requires avoiding computation of interactions between every pair of tokens. Broadly, three popular approaches tackle the generation cost of transformers by obtaining a summary of the context, called the recurrent state, in different ways. One can think of standard transformer having a recurrent state of the entire context. The first class of approaches improve efficiency by restricting attention to specific *strided* positions or local *sliding* window sizes (Child et al., 2019; Jiang et al., 2023). As the set of important tokens is not known in advance, these modifications strike a coarse trade-off between cost and performance (Gu & Dao, 2023; Arora et al., 2024a).

The second line of work replaces attention with its linear approximations or *linear attention* (Arora et al., 2024a; Katharopoulos et al., 2020). But such layers, by themselves, cannot solve simple compare-and-copy tasks (also called retrieval or recall tasks), and must be composed with other sequence mixers, such as local sliding window attention (Arora et al., 2024a).

The third line of work focuses on state-space models (SSMs), which keep a fixed length recurrent state that is read from when generating and updated after; recurrent neural networks and the recent MAMBA models Gu & Dao (2023); Dao & Gu (2024); Yang et al. (2025) fall in this class. Due to fixed recurrent state limiting the information that can be in memory, these models can solve recall tasks only upto fixed sequence lengths (Jelassi et al., 2024). Building competitive state-space models typically involves *hybrids*, which composes a state-space update layer with sliding window or dense attention layers. Choosing such a composition trades-off computation costs for performance because the attention layers that improve recall performance also come

[1]New York University. Correspondence to: Jatin Prakash <jp7467@nyu.edu>.

Proceedings of the 3^{rd} Efficient Systems for Foundation Models Workshop at the International Conference on Machine Learning, Vancouver, Canada. PMLR 267, 2025.

with larger time and memory costs.

Overall, the existing efficient sequence mixers restrict the recurrent state in a way that can harm recall ability when used without carefully composing them with computationally expensive attention layers (Arora et al., 2024a; Yang et al., 2025). To move beyond such choices, we ask the following question: *Can we build simple and efficient architectures without replacing or restricting the attention mechanism?*

To answer this, we start with the fact that natural language has redundancies (Shannon, 1951; Zipf, 2016; Mahowald et al., 2013), and therefore can be compressed. A sequence in natural language compressed into a shorter sequence forms a smaller recurrent state. Building on this fact, we propose a simple new architecture called the CAT. CAT models *chunks of tokens* in the sequence given compressed representations of past chunks. The two stages, compression of the chunks in the context and the decoding of the next chunk, are both parameterized with transformers, meaning the self-attention mechanism is retained.

Choosing a chunk size then trades-off between quality and efficiency: with a chunk size of C, the cost of inferring a sequence of size N in the compressor and decoder become $O(NC)$ and $O((N/C + C)^2)$ respectively. Choosing C appropriately makes CAT cost less than the $O(N^2)$ cost of dense attention. The compressor itself is learned end-to-end to avoid artificial restrictions on what information is kept in the recurrent state. Powered by the learnable compression of the context into a growing recurrent state, CAT outperforms existing efficient architectures on recall tasks while showing similar memory usage and generation speeds. Overall, this paper makes the following contributions:

1. Introduces the Compress and Attend Transformer (CAT) architecture, a simple and efficient alternative to dense transformers that can be trained end-to-end via the standard auto-regressive log-likelihood objective.
2. Elucidates how *chunk size* and *compression size* enable interpolating between dense transformers and efficient architectures, providing an alternative way to trade-off accuracy for efficiency.
3. Shows that CAT is up to 3× faster with 7× smaller memory footprint compared to dense transformers, matching it on perplexity on FineWeb-Edu (Penedo et al., 2024) and common language modeling and common-sense reasoning evaluations.
4. Demonstrates that CATs achieve better recall on real-world tasks[1] compared to existing efficient alternatives while showing similar memory usage and generation speeds.

[1]Huggingface link to EVAPORATE.

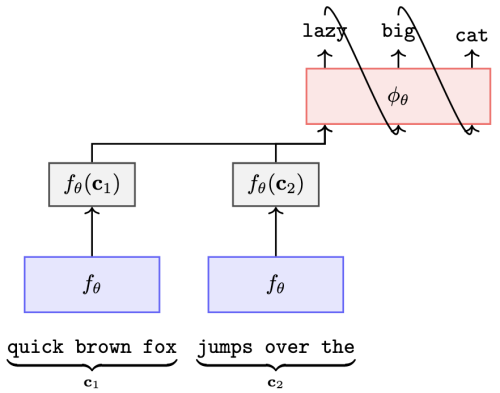

Figure 1: The Compress and Attend Transformer (CAT) architecture. The ith chunk (size 3 chunks shown) is decoded auto-regressively given compressed past chunks until $i-1$. The length of the sequence to the decoder reduces from N to N/C due to the compression, thus reducing the time and memory costs in the decoder due to self-attention.

2. Compress and Attend Transformers

Consider the task of generating a sequence of N tokens with a dense-attention transformer. In generating the token at the ith position, a transformer *attends* to all tokens in the history before the position i. However, human language has redundancies (Shannon, 1951; Zipf, 2016; Mahowald et al., 2013), meaning that one can compress the history without destroying any information. However, the subset of tokens to attend to is apriori unknown, and one needs to be learn the *compressor* to extract sufficient information from the history about the future. We instantiate this idea in the Compress and Attend Transformer (CAT). A schematic is given in fig. 1.

Given a sequence of N tokens $\{x_i\}_{i\leq N} \in \mathbb{R}^D$, we split the sequence into chunks of size C represented by $\{\mathbf{c}_i\}_{i\leq N_c}$, where $N_c = \frac{N}{C}$ is an integer,

$$\mathbf{x} = \{x_1, x_2, \cdots x_N\} \rightarrow \{\mathbf{c}_1, \mathbf{c}_2, \cdots \mathbf{c}_{N_c}\}.$$

We produce a new sequence by compressing each chunk $\mathbf{c}_i$ into elements that live in a space of ambient dimension D_d: $f_\theta(\mathbf{c}_i) \in \mathbb{R}^{D_d}$.

$$\{\mathbf{c}_1, \mathbf{c}_2, \cdots \mathbf{c}_{N_c}\} \rightarrow^{f_\theta} \{f_\theta(\mathbf{c}_1), f_\theta(\mathbf{c}_2), \cdots f_\theta(\mathbf{c}_{N_c})\}.$$

We propose to stack sequence mixers that aggregate information across sequence elements, such as self-attention, on top of the compressed sequence and then decode the original sequence from the compressed one. Formally, the decoder can be any model that takes in the compressed sequence as the input and outputs a distribution over the tokens in the sequence of tokens. For a sequence-to-sequence model, ϕ_θ,

the predictive distribution for the ith chunk is

$$p_\theta(\mathbf{c}_i = \{\mathbf{x}_{Ci} \cdots \mathbf{x}_{Ci+C-1}\} \mid \mathbf{c}_1 \cdots \mathbf{c}_{i-1}) = \phi_\theta(\{\mathbf{x}_{Ci} \cdots \mathbf{x}_{Ci+C-1}\} \mid f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1}))\}$$

Mechanically, we parameterize f_θ as a bidirectional transformer whose outputs are projected down with a linear layer and ϕ as a causal transformer that decodes tokens within each chunk auto-regressively while only attending to the previous tokens via the compressed sequence. We term this architecture the **Compress and Attend Transformer (CAT).** Unlike encoder-decoder architectures that attend directly to processed token-level embeddings of $\mathbf{c}_i$ (Raffel et al., 2020; Vaswani et al., 2017), ϕ_θ attends to the compressed $f_\theta(\mathbf{c}_i)$.

2.1. Training CATs

To model language from a compressed context as well as modeling it from the entire context, the compressed sequence $\{f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1})\}$ needs to retain the information in the chunks $\mathbf{c}_1 \cdots \mathbf{c}_{i-1}$ about the following chunks $f(\mathbf{c}_i) \cdots f(\mathbf{c}_{N_c})$. Retaining all the necessary information implies that the future chunks are all statistically independent of the past chunks given the compressed history:

$$\mathbf{c}_i \cdots \mathbf{c}_{N_c} \perp\!\!\!\perp \mathbf{c}_1, \cdots \mathbf{c}_{i-1} \mid f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1}), \quad (1)$$

In other words, sufficiency means that all the dependencies between chunks are captured by the compressed representation. This allows one to model the entire sequence from just the sequence of compressed representations. How can we build such compressors? We show that the default auto-regressive objective applied to the chunks guarantees the sufficiency:

Theorem 2.1. *Training* CATs *with auto-regressive log-likelihood yields sufficient compressors.*

The proof can is in appendix F. Theorem 2.1 shows that one can learn the two components of CAT, ϕ_θ and f_θ, together in an end-to-end fashion with the established auto-regressive cross-entropy loss. Next, we discuss how CATs is faster and memory efficient during training and generation than dense-attention transformer.

2.2. The cost of computing CATs

Here, we analyze CAT's attention costs in the compressor and the decoder with the chunk size is C and the projected dimension is D_d.

Compressor: The compressor costs are less than that of a dense transformer because each chunk is at most size C and there are $N_c = \frac{N}{C}$ chunks: thus, the time costs scale as $O\left(\frac{N}{C}C^2D\right) = O(NCD)$ and memory costs scale as $O(\frac{N}{C}C^2) = O(NC)$ (here, D is the embedding size for the compressor, which is different from D_d). Moreover, the compressor for each chunk is fully independent of the rest meaning that the compression stage is fully parallelizeable across chunks during training. During generation, it takes one transformer call to process the tokens within chunk in parallel to compute the compressed sequence token. Thus, the compressor is very efficient during training and generation compared to a dense transformer.

Decoder: The compressed sequence has at most $\frac{N}{C}$ tokens, and the decoding within each chunk looks at fewer than N_c (or $\frac{N}{C}$) compressed tokens ($f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1})$) and fewer than C tokens within the chunk. Thus, operating on the compressed sequences leads to attention costs of at most $O\left(\left(\frac{N}{C} + C\right)^2 D_d\right)$ in time across all chunks $\mathbf{c}_i$ and $O\left(\left(\frac{N}{C} + C\right)^2\right)$ in attention memory during training. During generation, the attention costs remain the same for the decoder, however, the memory required for the recurrent state (or the KV-cache) scales as $O(\frac{N}{C}D_d)$, which is a factor $O(C)$ times lower than a dense transformer. For even a moderate chunk size of 4, this can result in considerable reductions in memory during generation. Section 3.3 empirically compares CAT against different architectures.

Choosing hyperparameters for CAT. Arbitrary choices of chunk size C and projected dimension (decoder embedding size) D_d may violate the assumption in theorem 2.1 that the CAT model achieves sufficiency. The flexibility, or capacity, required to satisfy this assumption is in tension with lower latency and memory requirements. To instantiate CAT that matches the performance of a dense-attention transformer while requiring fewer computational resources, we need to set the *right* chunk-size C and projected dimension D_d. If C is too big (such as N), the cost of compressing may be comparable to that of dense attention; C being too small (such as 1) makes the decoder cost prohibitive. On the other hand, reducing D_d improves efficiency but can hurt language modeling and recall performance. Refer to section 3 and appendix C.3 for a discussion on this trade-off.

We experiment with different combinations of depth, embedding sizes for the encoder and decoder sizes in appendix D on the WikiText-103 dataset (Merity et al., 2016), which informed our architecture choices. We find that the decoder embedding size D_d and chunk size have the largest effect on performance. When compared against a Transformer of depth L and embedding size D, CAT models have an encoder of depth $L/2$ and the same embedding size and a decoder of depth L and an embedding size of $2D$. We experiment with different chunk sizes $\{4, 8, 16, 32\}$ which offer speedups and memory reductions that outpace the cost increase due to the doubling of the dimension. Despite the larger dimension, CAT models provide faster inference with a smaller memory footprint compared to dense transformers, at every chunk size we work with.

3. Experiments

This section compares CAT with the dense transformer and existing efficient architectures, along with their *hybrid* variants, on both language modeling and recall tasks such as question-answering and retrieval. Having pre-trained on the FineWeb-Edu dataset (Penedo et al., 2024), we zero-shot evaluate CAT and the other efficient architectures on a suite of real-world recall tasks (Arora et al., 2024b). Finally, the section demonstrates the gains in speed and reduction in the memory footprint that CAT obtains compared to the other architectures while modeling language similarly well.

Setting, baselines, and experimental details. We compare CAT with the dense transformer (or Transformer++ (Touvron et al., 2023)), and three recently proposed competitive efficient architectures: BASED (Arora et al., 2024a) which composes a linearized self-attention layer and sliding window attention layers, MAMBA2 (Dao & Gu, 2024) a state-space model, the Gated Delta Net (GDN) (Yang et al., 2025) which improves MAMBA2 with a new state-update rule. All models use hidden size $D = 1024$, 12 layers with the recommended hyper-parameters and layer-split in their respective papers. All models except CAT have ≈ 270 million parameters each. All models are trained for the same number of steps on 5 billion tokens, with a max sequence length of 1024. We train all models on the FineWeb-Edu dataset (Penedo et al., 2024). More details about the setup can be found in appendix E. We report results for different chunk sizes $C \in \{8, 16\}$ to show different trade-offs between performance and efficiency. CAT models, due to the $2D$ embedding size in the decoder, go up to $\sim 800M$ parameters and yet are faster and use lesser memory during inference. Ablations on the chunk size, compressor and decoder depth can be found in appendix B along with implementation details for CAT regarding training and generation, and the pseudo-code.

3.1. Language modeling evaluations

Table 1 reports perplexity on the held-out FineWeb-Edu data on WikiText-103 and LAMBADA datasets. We additionally evaluate pretrained models zero-shot on key language modeling benchmarks (HellaSwag, ARC-C, ARC-E, PIQA, WinoGrande, OpenbookQA). CAT models perform similar to the dense transformer on the FineWeb data while being among the best on all the other evaluations.

3.2. Language Recall and Question-Answering

We zero-shot evaluate all the models pretrained FineWeb-Edu on a suite of real-world recall tasks (the EVAPORATE suite which contains SWDE, DROP, SQUAD and Trivia QA datasets). The average length of the query in SWDE is $\approx 1K$ while all the other datasets have query-length under 300. Due to relatively small scale of our models and pretraining, we only evaluate on queries that are upto 1024 tokens. Appendix E gives additional details.

Method	FineWeb ↓	Wiki ↓	LMBD ↓	LM Eval ↑
Dense	21.2	26.1	48.8	42.0
BASED	21.4	27.1	47.2	41.8
MAMBA2	**19.9**	<u>24.9</u>	46.3	42.5
GDN	21.6	26.6	48.6	41.6
CAT-8	<u>20.7</u>	**24.8**	<u>46.1</u>	**42.9**
CAT-16	21.1	25.2	**46.0**	<u>42.8</u>

Table 1: We measure perplexity on a held-out set of FineWeb-Edu, and measure zero-shot perplexity on WikiText, LAMBADA (LMBD), and also zero-shot average accuracies across benchmarks common language modeling and common-sense reasoning. The tag ↓ means lower is better and ↑ means higher is better. CAT models perform as well or better than the dense transformer on held-out FineWeb data and are among the best on all the others.

Table 2 shows the results; CATs outperform existing efficient architectures. Notably, CAT-8 outperforms MAMBA2 or GatedDeltaNet while having faster generation speeds and similar memory usage at large batch sizes(fig. 2). When compared to BASED, CAT-16 outperforms it while having similar generation speeds and memory usage(fig. 2).

As attention improves recall abilities to long contexts, we also compare CAT-8 against architectures that use restricted self-attention layers. The first is sparse attention (Child et al., 2019; Jiang et al., 2023), with the stride set to 8 and with an embedding size $2D = 2048$. The second is a hybrid of GDN that uses sliding window attention of half the context size at every other layer, called Gated Delta Net-H1 (GDN-H1) (Yang et al., 2025). This comparison focuses on the SWDE dataset which has the largest average query length in the EVAPORATE datasets we consider ($\approx$ 1K).

Table 3 reports the results along with the theoretical cost reductions offered by the baselines, while we report the empirical cost reductions achieved by CAT-8. CAT-8 outperforms GDN-H1 while being $1.4\times$ faster using $3.4\times$ lesser memory than the dense transformer; GDN-H1 is actually slower and only offers a $2.5\times$ reduction in memory usage. Further, CAT dramatically outperforms sparse attention, closing $> 50\%$ of the gap to the dense transformer, while being just as fast and memory efficient.

3.3. Generation throughput and memory

In this section, we benchmark the generation throughput and memory consumption of various architectures. We compare dense transformer, BASED, MAMBA2 and CAT with chunk size of 8 and 16. All architectures use the same configuration as the language modeling experiments in section 3.1. Appendix B.1 gives further details about the setup.

Model	SWDE	DROP	SQuAD	Triv	Avg.
Dense	39.2	15.3	26.9	13.7	23.8
BASED	10.6	13.4	18.0	10.6	13.2
MAMBA2	10.6	15.0	20.5	**12.7**	14.7
GDN	12.0	13.9	18.3	12.1	14.1
CAT-8	**30.5**	**15.7**	**21.3**	11.8	**19.8**
CAT-16	13.5	15.2	14.6	12.6	14.0

Table 2: Zero-shot performance on real-world recall tasks. CAT-8 outperforms MAMBA2 and Gated-DeltaNet (GDN), while being similar or better in generation throughput and memory consumption (see fig. 2). CAT-16 outperforms BASED and is as efficient. Notably, CAT-8 closes 70% of the gap between the performance of best baseline and that of the dense transformer on SWDE, which that has the longest average query length ($\sim$ 1K) among the datasets (others have an average query length $\leq$ 300 tokens (Arora et al., 2024b)).

Model	Speed Up	Mem. Reduction	SWDE
Dense	1.0×	1.0×	39.2
GDN-H1	0.7×	2.5×	28.0
Sparse*	1.4×	3.4×	19.7
CAT-8	**1.4**×	**3.4**×	**30.5**

Table 3: SWDE recall performance of different models along with latency and memory usage at batch size 512 and sequence length 1024. * denote theoretical calculations, not actual wall-clock time or hardware memory utilization, which could be worse. CAT-8 outperforms both baselines while being as efficient or better.

Scaling with batch size. Figure 2 compares different architectures as one scales batch-size, given a fixed sequence length of 1024, the same setting used in section 3.1. Here, CAT outperforms both dense transformer (CAT is 1.4× faster, 3.5× cheaper) and MAMBA2 (CAT is 50% faster, similar in memory usage) in generation speeds and memory usage. Compared to BASED, CAT has similar generation speeds and memory usage as one scales the batch size to 512 when chunk size is to $C = 16$. Interestingly, we note that MAMBA2 scales poorly with batch-size, consuming even more time than a dense transformer on 1K sequence length. Refer to appendix C.6 on a ablation.

Scaling with sequence length. Figure 3 compares architectures as one scales the sequence length of generation, given a fixed batch-size of 256. We observe that CAT generates sequences upto 3× faster than the dense transformer while using upto 7× lesser memory. Compared to

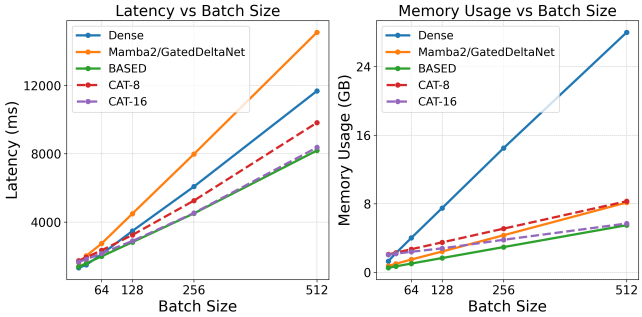

Figure 2: Latency (in seconds) and memory requirements (GB) of CATs with different chunk-size as the batch size increases, at fixed sequence length 1024. At highest batch size 512 and chunk size 8, CAT (or CAT-8) achieves upto 1.4× faster generation than the dense transformer and 1.5× faster than MAMBA2. On the other hand, CATs reduce memory requirements by a factor of $\approx$ 3.4× when compared to the dense transformer and are similar in memory usage to MAMBA2. On the other hand, CAT-16 achieves generation speeds and memory similar to BASED. Refer to appendix C.6 for poor scaling of latency of MAMBA2 w.r.t batch size at 1K sequence length.

MAMBA2, CAT-8 is similarly fast in generation while consuming slightly more memory at higher sequence lengths due to still a linear dependence on the sequence length, albeit it's lower by a factor of $O(C)$ (chunk size) compared to dense transformer). Compared to BASED, CAT-16 is slower and consumes more memory for the same reasons above. That being said, modern language models that are trained on longer sequence lengths usually use huge embedding dimension (D), for e.g. Llama3-70B uses $D \sim 8K$, which can be favorable to how memory scales in CAT with sequence length. Interestingly, we observed that the official implementation for BASED (Arora et al., 2024a) uses a slightly wrong implementation for sliding window KV-cache (i.e. memory increases with the sequence length). The benchmarks in fig. 3 are after we patched with an appropriate fix. Appendix C.7 benchmarks the official implementation.

3.4. Additional results.

Appendix C provides further evaluations of the CAT models. In appendix C.2, we evaluate CAT on the challenging multi-query associative recall (MQAR) task (Arora et al., 2023a; 2024a), where it solves the task with similar or better memory requirements compared to efficient architectures. Sparse attention does not solve the task without being deeper than the Transformer, making it slower than CATs.

CATs provides a different way to trade-off recall for more efficiency in terms of generation speeds and memory usage compared to existing architectures. To observe this trade-off, we pretrained CAT models at different chunk sizes

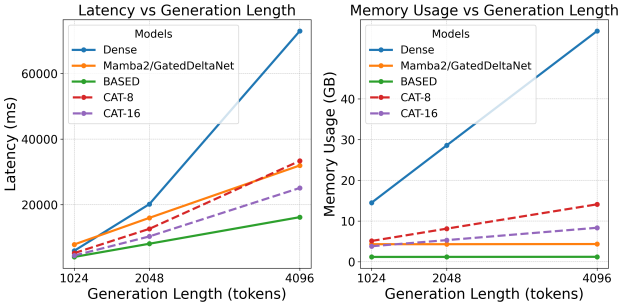

Figure 3: Latency and memory requirements of CATs with different chunk-size v/s the generation length. As generation length increases to 4096, CATs achieves upto $\approx 3\times$ faster generation than the dense transformer, and reduce memory requirements by a factor of $\approx 7\times$.

(following the setup in section 3.1 and section 3.2) and evaluated their downstream zero-shot recall abilities. Thus, CAT allows one to *interpolate* between dense attention and compressed sequence attention, trading off accuracy for efficiency according to the task requirements. These results can be found appendix C.3.

4. Related Work

Reducing the cost of self-attention enables scaling transformers to large contexts and has been the focus of much work Child et al. (2019); Parmar et al. (2018); Beltagy et al. (2020); Jiang et al. (2023). Common techniques include shrinking the set of tokens over which attention is computed; e.g. striding patterns and sliding window attention make the shrunken set include every kth token or the k tokens adjacent to the position being decoded (Child et al., 2019). More recently, concurrent works like (Yuan et al., 2025) falls into the same spirit as ours, trying to compress past tokens, however, a notable difference is the compression operation is performed at every layer, unlike in our case, which happens once. Works like (Arora et al., 2024a; Katharopoulos et al., 2020) linearize attention to make a fixed-size recurrent state that can be updated via simple averaging; the technique is to approximate self-attention with linear operations of query, key, and value vectors transformed through a feature map. Alternatively, one can replace attention with linear or pseudo-linear sequence mixers such as state-space models (SSMs) (Gu et al., 2021; Sun et al., 2023), gated convolutions (Fu et al., 2022; Poli et al., 2023) and input-dependent recurrunces (Peng et al., 2023; Gu & Dao, 2023) and more recently (Yang et al., 2025). Appendix A gives an extended discussion.

5. Discussion

We propose a simple modification to the Transformer architecture that first compresses chunks of tokens before modeling them sequentially, called the Compress and Attend Transformer (CAT). CAT is able to model language as well as dense attention from compressed sequences that are upto 16x smaller than the original sequence; at this compression, the CAT model is 3x faster with a 7x smaller memory footprint than the dense transformer, demonstrating the advantages of modeling language by mixing compressed sequences. CATs outperform existing efficient architectures like BASED (Arora et al., 2024a) MAMBA2 (Dao & Gu, 2024), and Gated Delta Net (Yang et al., 2025) on recall tasks while offering a different trade-off, taking up to 70% more time to achieve better recall while using similar memory. The CAT architectures demonstrates a performance-efficiency trade-off with simple choices, demonstrating a different approach to building efficient LLMs while approaching the recall abilities of dense transformers.

Future work Our current study is constrained by limited computational resources, which has prevented scaling to larger models and datasets. CAT models still do scale quadratically in time and memory meaning that, despite the compression, efficient models outpace CATs in both time and memory at large enough sequence lengths. Reducing the compressor and decoder depths offer further speedups and still outperforming existing efficient alternatives in Recall performance; appendix D shows these results. Due to CAT being larger in parameter count compared to the dense transformer, the training costs increase ($\leq 2\times$); this is due to the cost of the multi-layer perceptron (MLP) layer in the transformer block. Appendix B.5 provides more detail. Techniques such as the Mixture-of-Experts (MoEs) (Shazeer et al., 2017) focus on reducing the MLP cost, which is complementary to reducing the cost of self-attention. Techniques like MoEs can be readily used in CATs because both the compressor and the decoder components use the same structure of the transformer block.

Our fast inference implementation for CAT was inspired by `https://github.com/pytorch-labs/gpt-fast`; appendix B provides the pseudo-code. Our simple PyTorch implementation, which is possible because CAT retains the attention mechanism as is, performs competitively with implementations that use custom specialized kernels. New machinery developed to speed up any causal transformer, such as (Kwon et al., 2023) can be directly used for CATs, highlighting their simplicity and modularity.

This discussion also motivates the study of combining compression with other efficient architectures. For example, how well would a BASED model do at recall when built jointly with a compressor that see 16-token chunks? A separate direction of inquiry would involve benchmarks beyond recall-based tasks, especially on those where dense transformers typically excel. Finally, other domains such as vision, speech, and robotics that work with compressible signals can benefit from CAT architectures.

6. Acknowledgments

This work was partly supported by the NIH/NHLBI Award R01HL148248, NSF Award 1922658 NRT-HDR: FUTURE Foundations, Translation, and Responsibility for Data Science, NSF CAREER Award 2145542, ONR N00014-23-1-2634, Optum, and Apple.

References

Arora, S., Eyuboglu, S., Timalsina, A., Johnson, I., Poli, M., Zou, J., Rudra, A., and Ré, C. Zoology: Measuring and improving recall in efficient language models. *arXiv preprint arXiv:2312.04927*, 2023a.

Arora, S., Yang, B., Eyuboglu, S., Narayan, A., Hojel, A., Trummer, I., and Ré, C. Language models enable simple systems for generating structured views of heterogeneous data lakes. *arXiv preprint arXiv:2304.09433*, 2023b.

Arora, S., Eyuboglu, S., Zhang, M., Timalsina, A., Alberti, S., Zinsley, D., Zou, J., Rudra, A., and Ré, C. Simple linear attention language models balance the recall-throughput tradeoff. *arXiv preprint arXiv:2402.18668*, 2024a.

Arora, S., Timalsina, A., Singhal, A., Spector, B., Eyuboglu, S., Zhao, X., Rao, A., Rudra, A., and Ré, C. Just read twice: closing the recall gap for recurrent language models. *arXiv preprint arXiv:2407.05483*, 2024b.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

Barrault, L., Duquenne, P.-A., Elbayad, M., Kozhevnikov, A., Alastruey, B., Andrews, P., Coria, M., Couairon, G., Costa-jussà, M. R., Dale, D., et al. Large concept models: Language modeling in a sentence representation space. *arXiv preprint arXiv:2412.08821*, 2024.

Beltagy, I., Peters, M. E., and Cohan, A. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

Bisk, Y., Zellers, R., Gao, J., Choi, Y., et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pp. 7432–7439, 2020.

Child, R., Gray, S., Radford, A., and Sutskever, I. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019.

Clark, P., Cowhey, I., Etzioni, O., Khot, T., Sabharwal, A., Schoenick, C., and Tafjord, O. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

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

Dao, T., Fu, D., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness. *Advances in neural information processing systems*, 35:16344–16359, 2022.

Dong, J., Feng, B., Guessous, D., Liang, Y., and He, H. Flex attention: A programming model for generating optimized attention kernels. *arXiv preprint arXiv:2412.05496*, 2024.

Dua, D., Wang, Y., Dasigi, P., Stanovsky, G., Singh, S., and Gardner, M. Drop: A reading comprehension benchmark requiring discrete reasoning over paragraphs. *arXiv preprint arXiv:1903.00161*, 2019.

Fu, D. Y., Dao, T., Saab, K. K., Thomas, A. W., Rudra, A., and Ré, C. Hungry hungry hippos: Towards language modeling with state space models. *arXiv preprint arXiv:2212.14052*, 2022.

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

Gu, A., Goel, K., and Ré, C. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021.

Jelassi, S., Brandfonbrener, D., Kakade, S. M., and Malach, E. Repeat after me: Transformers are better than state space models at copying. *arXiv preprint arXiv:2402.01032*, 2024.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. Mistral 7b, 2023. URL https://arxiv.org/abs/2310.06825.

Joshi, M., Choi, E., Weld, D. S., and Zettlemoyer, L. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension. *arXiv preprint arXiv:1705.03551*, 2017.

Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J., Zhang, H., and Stoica, I. Efficient memory management for large language model serving

with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023.

Lockard, C., Shiralkar, P., and Dong, X. L. 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)*, pp. 3047–3056, 2019.

Mahowald, K., Fedorenko, E., Piantadosi, S. T., and Gibson, E. Info/information theory: Speakers choose shorter words in predictive contexts. *Cognition*, 126(2):313–318, 2013.

Merity, S., Xiong, C., Bradbury, J., and Socher, R. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

Milakov, M. and Gimelshein, N. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

Paperno, D., Kruszewski, G., Lazaridou, A., Pham, Q. N., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The lambada dataset: Word prediction requiring a broad discourse context. *arXiv preprint arXiv:1606.06031*, 2016.

Parmar, N., Vaswani, A., Uszkoreit, J., Kaiser, L., Shazeer, N., Ku, A., and Tran, D. Image transformer. In *International conference on machine learning*, pp. 4055–4064. PMLR, 2018.

Penedo, G., Kydlíček, H., Lozhkov, A., Mitchell, M., Raffel, C. A., Von Werra, L., Wolf, T., 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.

Peng, B., Alcaide, E., Anthony, Q., Albalak, A., Arcadinho, S., Biderman, S., Cao, H., Cheng, X., Chung, M., Grella, M., et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

Poli, M., Massaroli, S., Nguyen, E., Fu, D. Y., Dao, T., Baccus, S., Bengio, Y., Ermon, S., and Ré, C. Hyena hierarchy: Towards larger convolutional language models. In *International Conference on Machine Learning*, pp. 28043–28078. PMLR, 2023.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21 (140):1–67, 2020.

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

Sakaguchi, K., Bras, R. L., Bhagavatula, C., and Choi, Y. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

Shannon, C. E. Prediction and entropy of printed english. *Bell system technical journal*, 30(1):50–64, 1951.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

Sun, Y., Dong, L., Huang, S., Ma, S., Xia, Y., Xue, J., Wang, J., and Wei, F. Retentive network: A successor to transformer for large language models. *arXiv preprint arXiv:2307.08621*, 2023.

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

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

Yuan, J., Gao, H., Dai, D., Luo, J., Zhao, L., Zhang, Z., Xie, Z., Wei, Y., Wang, L., Xiao, Z., et al. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*, 2025.

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., and Choi, Y. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.

Zipf, G. K. *Human behavior and the principle of least effort: An introduction to human ecology*. Ravenio books, 2016.

A. Extended Related work

Reducing the cost of the self-attention helps scale Transformers to large contexts and much work pushes along this direction Child et al. (2019); Parmar et al. (2018); Beltagy et al. (2020); Jiang et al. (2023). Common techniques include shrinking the set of tokens over which attention is computed; e.g. striding patterns and sliding window attention make the shrunken set include every kth token or the k tokens adjacent to the position being decoded (Child et al., 2019). The time and memory costs of attention go down with the shrunken set along with the expressivity of the model built with these modifications. In turn, to achieve performance similar to that of a dense-attention Transformers, the efficient models either require big window sizes (making their KV cache large again) or more layers or larger dimension (in the case of sparse transformers) (Arora et al., 2024a).

A different line of work reduces the generation cost of transformers by limiting the recurrent state, which is the vector required to decode each token. Self-attention keeps track of the entire context meaning that the recurrent state increases in size with each decoded token. Work like (Arora et al., 2024a; Katharopoulos et al., 2020) linearize attention to make a fixed-size recurrent state that can be updated via simple averaging; the technique is to approximate self-attention with linear operations of query, key, and value vectors transformed through a feature map. The choice of the feature map falls to the user and approximating attention well requires the feature map to be large in size, which can counteract the gains in computational costs achieved by the linearization. Alternatively, one can replace attention with linear or pseudo-linear sequence mixers such as state-space models (SSMs) (Gu et al., 2021; Sun et al., 2023), gated convolutions (Fu et al., 2022; Poli et al., 2023) and input-dependent recurrunces (Peng et al., 2023; Gu & Dao, 2023). Typical implementations of linear attention do achieve impressive reductions in generation cost but restrict the expressivity of linearized attention to the extent that these models do not solve simple recall tasks without large state sizes (Arora et al., 2024a; 2023a). Arora et al. (2024a) show that many such state-space models are subsumed by Linearized attention and propose the BASED architecture that composes a kernel-approximation of attention with convolutional sequence mixers and sliding-window attention, which helps improve recall performance to an extent.

Unlike the work discussed above, the compressed sequence mixer requires no changes to the attention mechanism itself. Instead, we rely on the fact that natural language is redundant and can be compressed, and attention on compressed sequences is faster and requires lesser memory compared to attention on the full sequence.

Instead of approximating or replacing attention with low latency/memory layers, one can optimize the computation of attention to reduce wall-clock time and memory by leveraging hardware advancements. For example, Dao et al. (2022) compute attention in blockwise manner and exploit the nature of online softmax (Milakov & Gimelshein, 2018) which removes the need to instantiate the entire QK matrix and reduce calls to slow-read part of the GPU memory. As we utilize the attention mechanism as is, any reductions in cost due to hardware optimization that apply to the attention mechanism also proportionally reduce the cost of CAT models.

Finally, (Barrault et al., 2024) suggest learning "concepts" instead of tokens by modeling the latent representation of language produced by pushing the token sequence through a large sentence embedder. The focus of this work is to decouple the modeling of the low-level details in each language, like tense and grammar, from the larger concept space that is shared across languages. In contrast, the goal with the compressed sequence mixer is to reduce the cost of modeling sequences and can be used as a plug-and-play replacement to the latent concept model.

B. Implementation details

B.1. Generation benchmark details

Both dense transformer and CAT use FlexAttention with a causal mask. We use the kernel provided in (Dao & Gu, 2024) for MAMBA2, and BASED uses the causal dot product Fast Transformers CUDA kernel provided in (Arora et al., 2024a). We directly use the scripts provided by (Arora et al., 2024a) and (Dao & Gu, 2024) for benchmarking BASED and MAMBA2. 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.

B.2. Implementing CAT

Training implementation: To implement parallel chunk training efficiently on PyTorch, we make use of `torch.vmap` and the FlexAttention API (Dong et al., 2024). We compute $f_\theta(\mathbf{c}_i)$ using `torch.vmap` due to fixed shapes of each $\mathbf{c}_i$. To efficiently compute $\phi_\theta(\{\mathbf{x}_{Ci} \cdots \mathbf{x}_{Ci+C-1}\} \mid f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1}))$ is not straight-forward due to varying number past $f_\theta(\mathbf{c}_i)$. Thus, we can't apply `torch.vmap` directly like we did in computing $f_\theta(\mathbf{c}_i)$. In order to get around this, we directly pass all tokens $\{\mathbf{x}\}$ and $f_\theta(\mathbf{c}_i)$ to ϕ_θ and mask at appropriate positions to *emulate* parallel training for $\phi_\theta(\{\mathbf{x}_{Ci} \cdots \mathbf{x}_{Ci+C-1}\} \mid f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1}))$. We make use of FlexAttention API to obtain a custom self-attention kernel specifically for this masking scheme. This custom fused kernel gives us a significant boost in training throughput in self-attention costs compared to using a naive PyTorch masked implementation.

That being said, an efficient training kernel can be developed using similar principles as described in (Yuan et al., 2025). In our experiments, using FlexAttention did not give significant boosts compared to training speeds using Flash Attention on a dense transformer. This could be due to the fact that speeding up the attention maps (that we use, described in the below section) requires different principles than Flash Attention like optimization that Flex Attention might be using under the hood; similarly discussed in (Yuan et al., 2025).

We provide a naive training step implementation in PyTorch style pseudo-code at appendix B.3.

Presently, to distinguish between different $f_\theta(\mathbf{c}_i)$, current implementation of CAT passes the chunk `input_ids` $\mathbf{c}_i$ along with a learnable position embedding p_i (say) directly to f_θ. One could use sinosuidal embeddings too for this purpose, which can render CAT length extrapolation capability. This is left as a future work.

Generation implementation: We modify the implementation provided in `gpt-fast`[2] repository that makes use of CUDA graphs to reduce CPU overheads during generation using the `torch.compile(mode="reduce-overhead")` feature. We declare a static KV-cache memory of $O((C+\frac{N}{C})\cdot 2D)$. Whenever CAT finishes generating a chunk c_i, we compute $f_\theta(\mathbf{c}_i)$ representation, and prefill $f_\theta(\mathbf{c}_i)$ at position i in ϕ_θ, and start generating the next chunk tokens $\{\mathbf{x}_{Ci+1} \cdots \mathbf{x}_{Ci+1+C-1}\}$ from that position autoregressively, and this process continues. Note that in this process, one can use a simple causal mask.

Despite being a simple, pure PyTorch implementation, it is competitive with implementations using custom CUDA kernels (see section 3.3).

Refer to PyTorch style psuedo-code for generation at appendix B.3.

B.3. PyTorch style psuedo-code

B.3.1. TRAINING

```
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
    attention mask presented in below section
        # use the previous i+1 fx to predict the current chunk
        # shape of cur_chunk_logits: (b, 1, l, V)
```

[2] https://github.com/pytorch-labs/gpt-fast

```
        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

B.3.2. GENERATION

```

# 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

B.4. FlexAttention Mask used during training

To efficiently compute $\phi_\theta(\{\mathbf{x}_{Ci} \cdots \mathbf{x}_{Ci+C-1}\} \mid f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1}))$ during training in parallel, we make use of the FlexAttention API with a custom attention mask as show below.

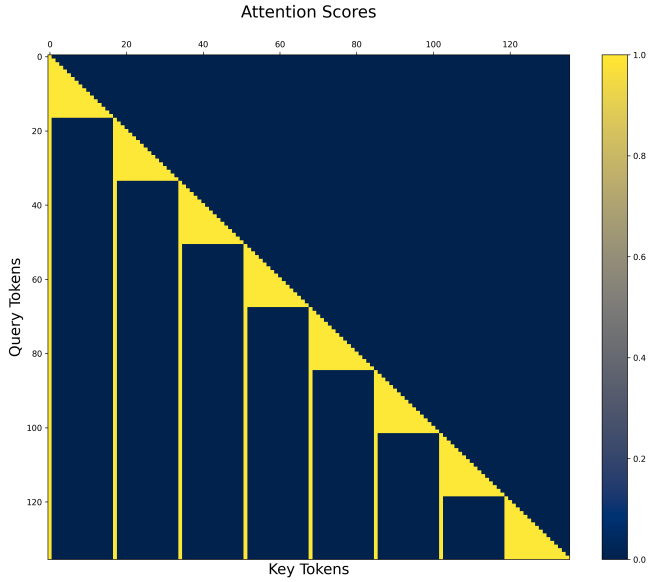

Figure 4: Sequence length is 128, and the chunk size that we use in this particular attention mask is $C = 16$.

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

B.5. Training throughput and memory discussion and comparison

As mentioned above in the implementation details, due to the unavailability of an efficient training kernel, theoritical speed ups that due to reduction in attention FLOPs in the CAT architecture don't appear in training wall-clock times. Additionally, MLPs in a transformer drive the majority of the FLOPs budget during training.

At a sequence length of 1024, CAT takes $\leq 2\times$ to train compared to a dense transformer and takes $\leq 2\times$ memory. However, at higher sequence lengths, such as 4096, this reduces to $\leq 1.5\times$ more time and memory, meaning even with an inefficient attention kernel, we start to see the asymptotic behaviour of CAT's attention FLOPs kick in. Note that this time takes into account that CAT uses $D_d = 2D$.

Developing an efficient attention kernel for training CATs is left as future work.

C. More results

C.1. Results on WikiText-103

Setup and baselines: We compare CAT with dense transformer, local sliding window attention (Jiang et al., 2023), sparse attention (Child et al., 2019), BASED (Arora et al., 2024a) and MAMBA2 (Dao & Gu, 2024). All models were trained for the same steps, having max context length upto 512, with the default hyper-parameters. Sparse attention uses a chunk size of 4. For more details regarding the setup, refer to appendix E.

Observations: We find that CAT models (across chunk sizes) performs competitively in terms of perplexity when compared with efficient architectures like BASED and MAMBA2. Notably, we tried two variants for BASED: (i) following the paper's recommendation to use 20% sliding window and 20% linear attention layers (reported as BASED), and (ii) increasing sliding window and linear attention layers by 2× (reported as BASED-2×). We find that BASED underperforms significantly, while BASED-2× performs competitively, highlighting the *complicated design process for modern efficient architectures*.

Note that CAT is able to compress upto 64 tokens in WikiText-103 dataset, without losing significant perplexity. This might point to a lot of redundancy in language in this dataset.

Architecture	Perplexity
Dense Attention	16.7
Sliding Window	17.8
Sparse Attention	19.1
BASED	20.8
BASED-2×	17.2
MAMBA2	17.0
CAT-8	17.4
CAT-16	17.7
CAT-32	17.6
CAT-64	17.3

Table 4: Perplexity results on WikiText-103 for various models.

C.2. Synthetic multi-associate query recall

Setup: We additionally evaluate CAT 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 256 having varying number of key-value pairs. 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. Note that the state size for CAT is $\frac{N}{C} \cdot D = 4096$ for this particular sequence length and model dimension. More details about the task can be found in appendix E. Sparse attention uses a chunk size of 4; Sliding window uses a window size of 64.

Method	Solves?	State Size
Dense	✓	16384
Sparse	✗	4096
Sliding Window	✗	4096
BASED	✓	4096
CAT	✓	4096

Table 5: For each method, we report the state size at which the particular method was trained for the MQAR task. Each method was grid searched for best possible hyper-parameters. We use the state size calculations provided in (Arora et al., 2024a; 2023a).

Observations: We find that CAT is able to solve the MQAR task using similar memory requirements as other efficient architectures such as BASED. Notably, we find the sparse attention as well as sliding window attention fail to solve the task at 2 layers, highlighting their dependence on depth.

C.3. Trading off language recall accuracy for efficiency in CAT

Here, we show how the language recall changes when one increases chunk size C. We clearly see a drop in performance when we go to higher chunk sizes like $C = 32$.

We additionally show plots for latency/memory usage vs batch size/sequence length across different chunk sizes C for CAT in the main text along with other efficient architectures (refer to section 3.3).

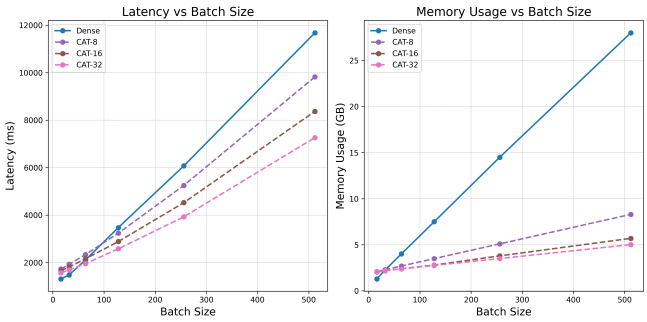

Figure 5: Latency/Memory usage across batch sizes for different CATs; fixed batch size of 512, and generation length is 1024.

More details about these recall evaluations can be found in appendix E.

Model	SWDE	DROP	SQuAD	TriviaQA	Avg.
Dense	**39.2**	**15.3**	**26.9**	**13.7**	**23.8**
CAT-8	30.5	15.7	21.3	11.8	19.8
CAT-16	13.5	15.2	14.6	12.6	14.0
CAT-32	7.9	11.8	11.5	10.6	10.5

Table 6: Zero-shot evaluation of various CAT models on language recall tasks.

C.4. Additional results on language recall

More details about these experiments can be found in appendix E. Sparse attention (Child et al., 2019) uses a chunk size of 8, with model dimension of $2D$, similar to CAT.

Model	SWDE	DROP	SQuAD	TriviaQA	Avg.
Dense	**39.2**	**15.3**	**26.9**	**13.7**	**23.8**
GatedDeltaNet-H1	28.0	18.1	26.6	13.3	21.5
Sparse $2D$	19.7	14.5	18.7	13.7	16.7
CAT-8	30.5	15.7	21.3	11.8	19.8

Table 7: Zero-shot evaluation of models on language recall tasks.

C.5. Results on common language modeling benchmarks

We evaluate all models on common LM evaluation benchmarks. More details about these evaluation experiments can be found appendix E.

Model	HS	PIQA	ARC-E	ARC-C	WG	OpenbookQA	Avg.
Dense	0.358	0.662	0.537	0.224	0.530	0.212	0.420
BASED	0.354	0.646	0.553	0.224	0.520	0.212	0.418
MAMBA2	0.372	0.645	0.560	0.237	0.520	0.216	0.425
DELTANET	0.370	0.635	0.541	0.230	0.500	0.220	0.416
DELTANET-H1	0.360	0.649	0.546	0.230	0.516	0.192	0.416
CAT-8	0.356	0.643	0.577	0.269	0.504	0.228	0.429
CAT-16	0.354	0.641	0.585	0.254	0.514	0.232	0.428

Table 8: Common langauge modeling and common-sense reasoning accuracy (HS: HellaSwag, WG: WinoGrande, OBQA: OpenbookQA).

Model	WikiText (PPL)	LAMBADA (PPL)	Avg.
Dense	26.13	48.87	37.50
BASED	27.19	47.26	37.23
MAMBA2	24.92	46.34	35.63
DELTANET	26.50	48.60	37.55
DELTANET-H1	25.27	47.40	36.34
CAT-8	24.79	46.15	35.47
CAT-16	25.23	45.96	35.60

Table 9: Language modeling perplexity (lower is better).

C.6. MAMBA2 benchmarks on different state sizes

We provide benchmarks for MAMBA2 using different d_{state} settings. We observe that as one increases d_{state}, MAMBA2 starts taking more time to perform generation than dense attention on larger batch sizes. This might not be true for all sequence lengths, however, it is observed at relatively smaller sequence length of 1K that we experiment with. This could be due to higher values of state sizes that increases the overall FLOPs per token for generation. Therefore a higher constant, and thus, a higher slope when one scales the batch size. We use the official code provided here that uses efficient CUDA graphs to benchmark generation throughput: https://github.com/state-spaces/mamba/blob/main/benchmarks/benchmark_generation_mamba_simple.py

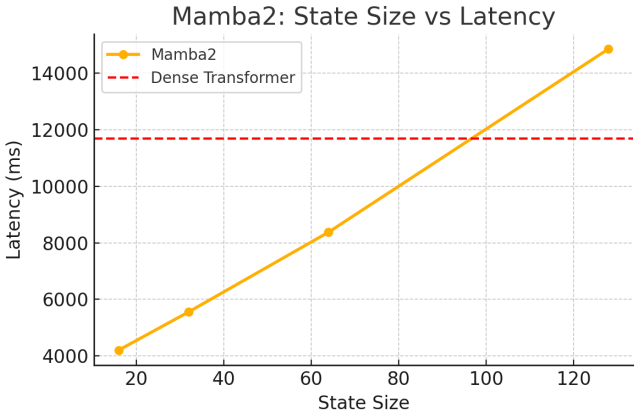

Figure 6: Comparison of state-size d_{state} vs latency of generation in MAMBA2. Benchmarks were conducted using 1K sequence length, batch size of 512. MAMBA2 and dense transformer use the same configuration as defined in section 3.1.

C.7. BASED benchmarks using official code and our patched code

We use the official code provided in BASED: https://github.com/HazyResearch/based/blob/main/train/benchmark/configs/01-29-forward-360m.py.

We observe that the official code uses more memory as sequence length increases. However, after patching the KV-cache fix (which is supposed to happen here: https://github.com/HazyResearch/based/blob/931f27a1c7bca842f4a703cf91ca8fc038dceba6/based/models/mixers/slide_attention.py#L377), memory usage of BASED remains constant at all sequence lengths.

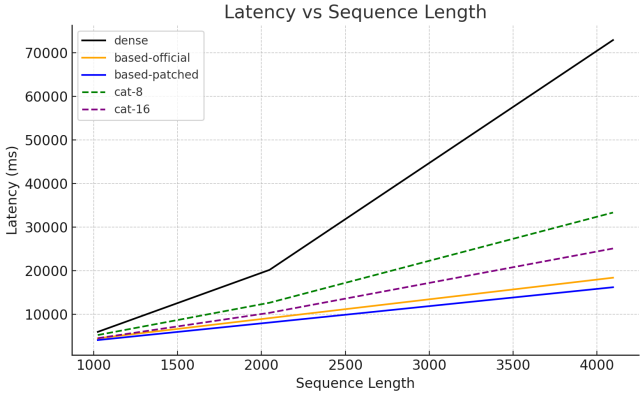

Figure 7: Latency of generation measured across different sequence length

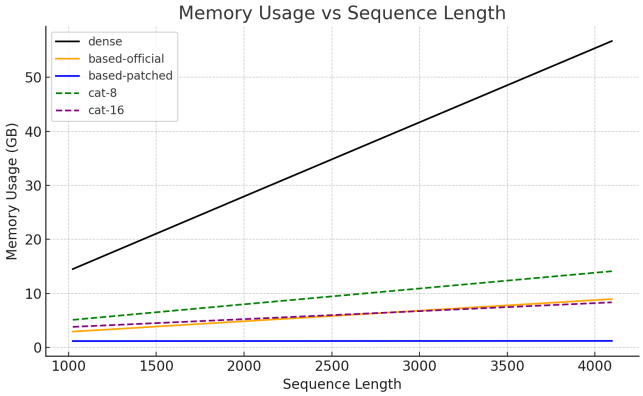

Figure 8: Memory usage of generation measured across different sequence length

D. Ablation results on choices for the architecture for CAT

D.1. Ablation on using D different from D_d

With this ablation, we show that the compressor f_θ can use half the embedding size as compared to the decoder. We fix $D_d = 1536$ for these experiments. For this ablation, we use WikiText-103. Both compressor and decoder use the depth $L = 6$.

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

Table 10: Comparison of choices on D compared to D_d on WikiText-103 perplexity.

We observe that one can get away with using smaller value of D as compared to D_d. What matters is the size of D_d (decoder embedding size, or the compressed embedding size), which we show in the next ablation.

D.2. Ablation on D_d

We ablate on different choices of D_d along with different chunk sizes in CAT . In this setup, we fix D in the compressor, and only vary D_d 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_d	**Perplexity**
4	D	19.8
	$2D$	17.4
8	D	20.4
	$2D$	17.7
16	D	20.2
	$2D$	17.6

Table 11: Comparison on choices of chunk sizes and sizes of D_d on WikiText-103 perplexity.

We observe that we obtain the best perplexities when we $D_d = 2D$ for the particular chunk size we are using. This goes back to our discussion on compression as well as compute required to decode in section 2.2. Using this observation, we used this as our *default* configuration for the FineWeb-Edu experiments. Note that, as pointed out in section 2.2, chunking renders CAT substantial gains in speed and memory that one can increases D_d by $2\times$ while still being significantly faster than dense transformer and on par with other efficient architectures (see section 3.3).

D.3. Ablation on depth of the compressor L_{f_θ}

We ablate on the depth L_{f_θ} of the compressor f_θ. For a fixed chunk-size, $D = 768$ (compressor embedding size), $D_d = 2D$ (compression size or the decoder embedding size), and a fixed depth of the decoder ($L_{\phi_\theta} = 6$), we vary the compressor depth L_{f_θ}.

Chunk Size C	**Depth of Compressor** L_{f_θ}	**Perplexity**
8	6	17.4
	3	17.4
16	6	17.8
	3	17.7

Table 12: 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. This pings back to our discussion on different capacity requirements for the compressor and decoder in section 2.2. 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. One might require some lower threshold of compressor depth to start compressing chunks of tokens, but we leave this to future work.

D.4. Ablation on depth of the decoder

One more interesting avenue that we explore is: can we reduce the depth of the decoder too? since we operate on compressed sequence of chunks, one could get away with lower depth of the decoder. Moreover, we decided to use $D_d = 2D$ as the decoder embedding size. This was initially done to increase compute for decoding from compressed chunks, but can one still decode effectively from reduced decoder depth. For this experiment, we use the same setup in section 3.1 with $D_d = 2D = 2048$. We use compressor depth of $L_{f_\theta} = 4$ and $L_{\phi_\theta} = 8$ for this experiment. We use chunk size of $C = 8$ for this experiment.

Decoder Depth	Perplexity	Avg. Recall
12	20.7	19.8
8	21.8	18.6

Table 13: Performance across different decoder depths for FineWeb-Edu.

This means one could gain some efficiency in terms of generation speeds and memory usage by shaving of some layers off of the decoder. Notably, this configuration of CAT outperforms on all other efficient architectures in generation throughput (8 layer model takes $\sim$ 6600ms on 1024 sequence length, batch-size 128), memory usage (taking $\frac{3}{4}^{th}$ memory of CAT-8 12 layers) and closes 50% of the gap in average recall performance between the efficient alternatives and the dense transformer, although with a 5% drop in language modeling performance.

E. Dataset details:

E.1. Language recall experiments

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) and DROP (Dua et al., 2019). We could not use FDA (Arora et al., 2023b) since we only trained models upto 1K sequence length due to limited computation budget. Since our pretrained models are small, we use the Cloze Completion Formatting prompts provided by (Arora et al., 2024b).

Due to relatively smaller scale of our models and pre-training, we only evaluate queries that are upto $\leq 1K$ tokens. We use greedy decoding across all models to generate samples. We use the same evaluation metric as suggested in (Arora et al., 2024b), where the model generates upto 48 tokens.

E.2. Language common-sense and reasoning experiments

Following common practices done in (Gu & Dao, 2023; Dao & 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). We source our datasets from: `https://huggingface.co/collections/DatologyAI/standard-llm-evals-67f58694230e7e2a3cad4e34`.

E.3. FineWeb-Edu experiments

We use the dataset provided here: `https://huggingface.co/datasets/HuggingFaceFW/fineweb-edu`. We use the first 5B tokens from their 10B token processed split.

For all models, we use the LR as 8e-4 and cosine decay it to 8e-5 with a linear warm up for 250 gradient steps, with a global batch-size of 256, and train it for 18K steps that amounts to around 5B training tokens. We use the GPT2 tokenizer. All models were trained using `bfloat16` mixed-precision training. We use the Adam optimizer with weight decay as 0.1, and $\beta_1 = 0.9, \beta_2 = 0.95$ with a gradient clipping of 1.0.

1. Dense transformer (or Transformer++) (Vaswani et al., 2017; Touvron et al., 2023): This is a 12 layer model, that use $D = 1024$. We use rotary position embeddings along with the FlashAttention kernel to perform self-attention. The MLP is a SwiGLU MLP (Touvron et al., 2023). The model size comes around to be 257M parameters.

2. BASED (Arora et al., 2024a): The model is again 12 layers, where 20% layers are linear attention, 20% layers are local sliding window with size of 128, and the rest are `BaseConv` layers according to (Arora et al., 2024a) recommendation . The MLPs used in a block is a SwiGLU MLP. We use $D = 1024$, and the Taylor feature dimension $d = 16$ for linear attention. For `BaseConv`, we use $k = 3$ and the `expand_proj` to be 4. For sliding window attention, we utilize the FlexAttention API. The model size comes around to be 267M parameters. We use the official codebase to implement BASED for our experiments: `https://github.com/HazyResearch/based` and generation throughput and memory benchmarking.

3. MAMBA2 (Dao & Gu, 2024): The model uses 24 layers with $D = 1024$. 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`. The model size comes around 261M parameters. 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. Gated Delta Net (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 (same as MAMBA2 above). For the hybrid version, we use sliding window layers at every other layer with a sliding window size of 512.

5. CAT: The compressor uses $D = 1024$ with 6 layers of Transformer++ blocks. We project C tokens to a D_d sized embedding using a simple linear layer. The decoder uses 12 layers of Transformer++, with $D_d = 2048$ i.e. $D_d = 2D$. The total parameter count, due to compressor and decoder, comes around to be $\sim 800M$. However, note that, even with more parameters, CAT is still efficient in memory and generation.

E.4. WikiText-103 experiments

We use the dataset provided here: `https://huggingface.co/datasets/Salesforce/wikitext`

For all models, we use the LR as 6e-4 and cosine decay it to 1e-5 with a linear warm up for 500 gradient steps, with a global batch-size of 128, and train it for 8K steps that amounts to around 0.53B training tokens. We use the Llama2 tokenizer for these experiments. All models were trained using `bfloat16` mixed-precision training. We use the Adam optimizer with weight decay as 0.1, and $\beta_1 = 0.9, \beta_2 = 0.95$ with a gradient clipping of 1.0.

1. Dense transformer (or Transformer++): This is a 6 layer model, that use $D = 768$. Rest of the configuration is same as FineWeb-Edu experiments.
2. Sparse transformer++ (Child et al., 2019): This is a 6 layer model, that use $D = 768$ that uses a sparse mask with a chunk size of 4. We used FlexAttention API to create optimized Flash Attention like kernel for this.
3. Sliding Window transformer++ (Jiang et al., 2023): This is a 6 layer model, that use $D = 768$ that uses a sliding window size of 64.
4. BASED: We use $D = 768$, and the Taylor feature dimension $d = 16$ for linear attention, with a local sliding window size of 64. Rest of the configuration is same as FineWeb-Edu experiments. Note that, we used two configuration for BASED in WikiText experiments. BASED-2$\times$ uses 2$\times$ the sliding window and linear attention layers.
5. MAMBA2: The model uses 12 layers with $D = 768$. Rest of the configuration is same as FineWeb-Edu experiments.
6. CAT : The compressor uses $D = 768$ with 3 layers of Transformer++ blocks. We project C tokens to a D_d sized embedding using a simple linear layer. The decoder uses 6 layers of Transformer++, with $D_d = 1536$ i.e. $D_d = 2D$.

E.5. MQAR experiments

We use the scripts provided here: `https://github.com/HazyResearch/zoology` to create our datasets on sequences upto 256 in length with varying key-value pairs.

All models use a batch-size of 128, with a embedding size $D = 64$.

F. Proofs

Proof. Assume that the CAT model achieved the optimum:

$$\theta^* = \arg\max_{\theta} \mathop{\mathbb{E}}_{\mathbf{c}_1,\cdots\mathbf{c}_{N_c}} \sum_{i \le N_c} \log \phi_\theta(\{\mathbf{x}_{Ci} \cdots \mathbf{x}_{Ci+C-1}\} \mid f_\theta(\mathbf{c}_1) \cdots f_\theta(\mathbf{c}_{i-1})).$$

At optimality, you have that the model's auto-regressive conditional distribution matches that the of the true data-generating process.

$$\begin{aligned} \forall i \quad p(\mathbf{c}_i \cdots \mathbf{c}_{N_c} \mid f_{\theta^*}(\mathbf{c}_1) \cdots f_{\theta^*}(\mathbf{c}_{i-1})) &= p(\mathbf{c}_i \cdots \mathbf{c}_{N_c} \mid \mathbf{c}_1, \cdots \mathbf{c}_{i-1}) \\ &= p(\mathbf{c}_i \cdots \mathbf{c}_{N_c} \mid \mathbf{c}_1, \cdots \mathbf{c}_{i-1}, f_{\theta^*}(\mathbf{c}_1) \cdots f_{\theta^*}(\mathbf{c}_{i-1})), \end{aligned}$$

where we use the fact that conditioned on $\mathbf{c}_1, \cdots \mathbf{c}_{i-1}$, any deterministic functions of the chunks are independent of all other random variables. The above equality implies eq. (1):

$$\begin{aligned} & \forall i, \quad p(\mathbf{c}_i \cdots \mathbf{c}_{N_c} \mid f_{\theta^*}(\mathbf{c}_1) \cdots f_{\theta^*}(\mathbf{c}_{i-1})) = p(\mathbf{c}_i \cdots \mathbf{c}_{N_c} \mid \mathbf{c}_1, \cdots \mathbf{c}_{i-1}, f_{\theta^*}(\mathbf{c}_1) \cdots f_{\theta^*}(\mathbf{c}_{i-1})) \\ \implies & \forall i, \quad \mathbf{c}_1, \cdots \mathbf{c}_{i-1} \perp\!\!\!\perp \mathbf{c}_i \cdots \mathbf{c}_{N_c} \mid f_{\theta^*}(\mathbf{c}_1) \cdots f_{\theta^*}(\mathbf{c}_{i-1}). \end{aligned}$$

□