SIMPLE LINEAR ATTENTION LANGUAGE MODELS BALANCE THE RECALL-THROUGHPUT TRADEOFF

Simran Arora*, **Sabri Eyuboglu***, **Michael Zhang***, **Aman Timalsina, Silas Alberti, Dylan Zinsley, James Zou, Atri Rudra, Christopher Ré**
Department of Computer Science, Stanford University
{simran,eyuboglu,mzhang,alberti,jamesz,chrismre}@cs.stanford.edu
{dylanzin,atri}@buffalo.edu
{atimalsi}@purdue.edu

ABSTRACT

Recent work has shown that attention-based language models excel at *recall*, the ability to ground generations in tokens previously seen in context. However, the efficiency of attention-based models is bottle-necked during inference by the KV-cache's aggressive memory consumption. In this work, we explore whether we can improve language model efficiency (*e.g.* by reducing memory consumption) without compromising on recall. By applying experiments and theory to a broad set of architectures, we identify a key tradeoff between a model's *state size* and recall ability. We show that efficient alternatives to attention (*e.g.* H3, Mamba, RWKV) maintain a fixed-size recurrent state, but struggle at recall. We propose BASED a simple architecture combining linear and sliding window attention. By varying BASED window size and linear attention feature dimension, we can dial the state size and traverse the Pareto frontier of the recall-memory tradeoff curve, recovering the full quality of attention on one end and the small state size of attention-alternatives on the other. We train language models up to 1.3b parameters and show that BASED matches the strongest sub-quadratic models (*e.g.* Mamba) in perplexity and outperforms them on real-world recall-intensive tasks by 9.03 accuracy points. Implementations of linear attention are often less efficient than optimized standard attention implementations. To make BASED competitive, we develop IO-aware algorithms that enable 24× higher throughput on language generation than FlashAttention-2, when generating 1024 tokens using 1.3b parameter models. Code: https://github.com/HazyResearch/based

1 INTRODUCTION

The choice of sequence mixer in a language model affects both its quality and efficiency (Vaswani et al., 2017). Prior work shows that attention excels at *recall*, the ability to ground generations in previously seen tokens in context (Arora et al., 2023a; Olsson et al., 2022). However, attention throughput is bottlenecked by quadratic complexity in training and large memory consumption in inference. *Can we improve the real-world speed of language models without comprising quality?*

We start by exploring the pareto frontier of the tradeoff between high-recall and high-throughput models. We evaluate a diverse set of architectures on a difficult synthetic *associative recall* (AR) task and demonstrate a fundamental recall-memory (throughput) tradeoff that holds across architecture classes (Figure 2). Attention models perform AR perfectly, but a their recurrent state (*i.e.* the KV-cache) grows linearly with the sequence length. Sliding window attention offers a way to cap the size of the recurrent state at the cost of poorer long-range recall (Jiang et al., 2023). However, Mamba, a recently proposed SSM architecture expands the Pareto frontier beyond sliding window. This begs the question, what other models?

We study two simple techniques for improving attention efficiency: sliding window attention and softmax-approximating linear attention. Our results (Table 1, Figure 1, center) suggest that neither approach alone suffices to navigate the Pareto frontier. We find *linear attention* alone struggles to solve AR (Figure 1, center). We hypothesize that linear attention lacks the precision to perform local

*Equal contribution.

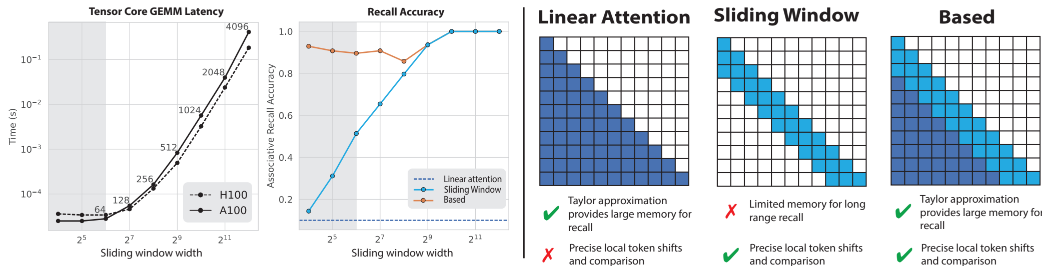

Figure 1: **BASED overview.** Combining linear attention with *tiny* sliding window softmax attention (up to 64 tokens in width) enables improved recall accuracy with limited efficiency overhead. (*Left*) Time to execute Cutlass GEMMs (y) vs. sliding window attention size (x) on tensor cores. (*Center)* Model recall accuracy (y) vs. sliding window attention size (x). We compare linear attention alone (dark blue), sliding window attention alone (light blue), and their combination (BASED, orange). (*Right*) Schematic diagram of BASED illustrating how the two components complement each other.

token shifts and comparisons (Fu et al., 2023a). In *sliding window attention*, AR range is limited by the width of the windows (Figure 1, center). As we increase the window size, the recurrent state grows linearly and has a non-linear affect on speed during training and inference (Figure 1, left).

We combine these two techniques into a single architecture, which we call BASED (Figure 1, right). We find that sliding window attention and linear attention complement each other, enabling BASED to expand the pareto frontier of the recall-memory tradeoff (Figure 2). We suspect that (1) the large recurrent memory of linear attention could make up for the limited range of narrow sliding window attention and (2) sliding window attention handles the precise local shifts needed to perform AR.

To make BASED competitive with SoTA attention (Dao, 2023) and recurrent (Gu & Dao, 2023) models under wall-clock and throughput metrics, we introduce several IO-aware optimizations.

1. Matrix multiplications up to 64×64, fit neatly into tensor core units of modern GPUs (Figure 1, left). We thus compute local softmax attention using windows sized at dimension 64.
2. In *linear attention*, for quality we use the 2nd-order Taylor approximation of softmax. However, with sequence length N and head dimension d, this naïvely requires $\mathcal{O}(Nd^3)$ time and space complexity (Zhang et al., 2024; Keles et al., 2023). Linear attention is also often slower than modern attentions in practice (Dao, 2023). Thus we develop custom CUDA kernels to make our attention competitive in real-world wall-clock time and memory usage. We reduce HBM IO cost relative to the naïve algorithm by $\mathcal{O}(Nd^2 - Nd)$ bytes, and avoid $O(Nd^3)$ bytes in SRAM to register data movement (Section 4).

We show that BASED competes in quality with strong Transformer++ (Touvron et al., 2023) and SoTA sub-quadratic (*e.g.* Mamba (Gu & Dao, 2023)) baselines at 1.3Bn parameters across language modeling on the Pile, DNA modeling, and the LM Eval Harness (Gao et al., 2023). BASED also outperforms prior sub-quadratic architectures on the AR slice of the Pile and in downstream recall-intensive tasks by 0.14 perplexity and 6.22 accuracy, respectively. In efficiency, BASED enables up to $24\times$ higher throughput than the strong FlashAttention-2 implementation on generation. Our code is available at: `https://github.com/HazyResearch/based`.

2 PRELIMINARIES

We briefly describe the preliminaries for BASED. The extended related works is in Appendix A.

Attention Softmax attention incurs low-throughput for long sequences. Attention has quadratic training complexity and for every new output $\boldsymbol{y}_n$ requires nd operations over a growing *KV-cache* of prior $\{\boldsymbol{k}_i, \boldsymbol{v}_i\}_{i=1}^{n-1}$ during generation. Meanwhile, *linear attentions* replace the softmax $\exp(\boldsymbol{q}^\top \boldsymbol{k})$ in standard attention with feature map dot-products $\phi(\boldsymbol{q})^\top \phi(\boldsymbol{k})$ (Katharopoulos et al., 2020b). These methods use matrix product associativity to compute attention in $\mathcal{O}(Nd^2)$ time and space in training, and constant memory and $\mathcal{O}(1)$ time per-token in generation (Kasai et al., 2021; Schlag et al., 2021). Present linear attentions struggle to perform recall and achieve fast wall-clock time vs. standard attention (Dao et al., 2022; Zhang et al., 2024).

Architecture	Params	Efficiency		Language Modeling (Pile)			Info. Extraction		QA	Common
		Prefill Tok./ms ↑	Generate Tok./ms ↑	All Ppl. ↓	AR Ppl. ↓	Other Ppl. ↓	SWDE Acc ↑	FDA Acc ↑	SQUAD F1 ↑	LM-Evals Avg. Acc. ↑
Transformer++	1.33b	103.50	0.99	**7.26**	**1.74**	**8.10**	**71.92**	**73.23**	**36.19**	**47.64**
BASED	1.35b	**161.71**	24.28	7.43	1.87	8.26	48.06	24.41	30.46	46.68
Mamba	1.32b	112.22	**25.69**	7.48	1.96	8.29	34.74	12.89	28.20	46.84
Transformer++	360m	207.77	23.82	**8.39**	**1.87**	**9.42**	**57.97**	**58.00**	**27.18**	**44.08**
BASED	363m	**514.57**	**47.23**	8.65	2.07	9.64	29.16	11.71	25.07	43.03
Mamba	358m	267.09	39.95	8.64	2.21	9.59	23.67	6.53	24.06	43.51
GLA	362m	—	—	9.12	2.36	10.68	—	—	—	—
RWKV v5	362m	—	—	9.79	2.40	10.90	—	—	—	—
H3	362m	—	—	10.60	4.88	11.23	6.75	0.64	7.87	39.35

Table 1: **Evaluation of pre-trained language models.** All models were trained on the same 10 billion tokens from the Pile (Gao et al., 2020). We report perplexity on the Pile test set and on two slices of the test set: AR tokens and other tokens (see Section 5, Arora et al. (2023a)). We report zero-shot performance on three *recall-intensive* tasks: information retrieval (SWDE, FDA) and QA (SQUAD). We also report average performance on the set of *LM Eval Harness* (Gao et al., 2023) reasoning tasks used in Gu & Dao (2023) (details in Appendix B.3). These tasks do not require significant recall capacity because the input text is typically very short. See Section 5. Architectures (RWKV-v5 and GLA) that do not implement recurrent generation views are omitted (—).

Attention-free Various models use attention-free sequence mixers such as state-space models (Gu et al., 2021; Sun et al., 2023), gated convolutions (Fu et al., 2023a; Poli et al., 2023) and input-dependent recurrences (Peng et al., 2023; Gu & Dao, 2023). We compare to these architectures.

3 NO FREE LUNCH FOR EFFICIENT AND HIGH-QUALITY RECALL

We use a synthetic recall task called MQAR (Arora et al., 2023a) to empirically demonstrate the tradeoffs between a model's hidden state (*i.e.,* the size of its recurrent state or KV-cache) and its recall ability. In Appendix E.1, we provide theoretical analysis to complement these experiments. Appendix D.1 contains details on how the state size is calculated for each evaluated architecture — attention, Mamba, H3, Hyena, and BASED. In MQAR, models see key-value mappings and at inference time, need to recall the mappings to predict the next token (e.g., 4, 6, 1, 2, 3 below):

$$\text{A 4 B 3 C 6 } \underbrace{\text{F 1}}_{\textbf{Key-Value}} \text{ E 2} \rightarrow \text{A ? C ? } \underbrace{\text{F ?}}_{\textbf{Query}} \text{ E ? B ?}$$

In Figure 2, we find that we can smoothly interpolate between small hidden-states that permit high-throughput generation at the cost of poor recall (H3, Sliding window), and large hidden-states with high recall performance yet poor generation throughput (Attention). Mamba (Gu & Dao, 2023) reduces the state size for efficiency, but sacrifices in recall score. Our proposed architecture, BASED (Section 4), will explore a different set of points on the Pareto-frontier as we vary the hyperparameters that determine its state size (*e.g.* feature dimension and model dimension).

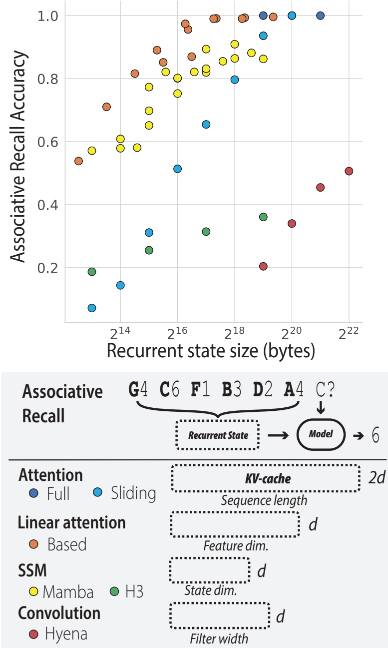

Figure 2: **Memory - recall tradeoff.** x-axis shows the state size (bytes) during generation; y-axis shows accuracy on the MQAR task (Arora et al., 2023a). For each architecture, we vary the hyperparameters that affect the state size. State sizes are computed using the equations in Appendix D.2.

We then find increased hidden state size alone may not be sufficient for recall. Several subquadratic models, such as gated convolutions (H3, Hyena, RWKV) fall below the Pareto frontier (Fig. 3). We also observe differences in the recall quality across linear attention feature maps; Taylor series approximations to the exponential function (De Brebisson & Vincent, 2015; Keles et al., 2023; Zhang et al., 2024) maintain Pareto-optimality. Figures 2 and 3 can be reproduced or extended to new architectures using the scripts provided at https://github.com/HazyResearch/zoology.

Our theoretical analysis provides further insight into these empirical observations. First, using results from communication complexity theory, we show that the recall capacity of *any* causal model (*e.g.* Mamba, Attention) is bounded by the size of its recurrent state (Theorem E.15 in Appendix E).

Theorem 3.1. *Any recurrent model*[1] *depending causally on input* $\boldsymbol{u} \in \{0,1\}^{N\times d}$ *requires* $\Omega(N)$*-bits*[2] *in state size to solve* MQAR.

This result suggests that the tradeoff observed in Figure 2 is fundamental, not an artifact of architectural quirks. We provide additional theoretical results in Appendix E.1.

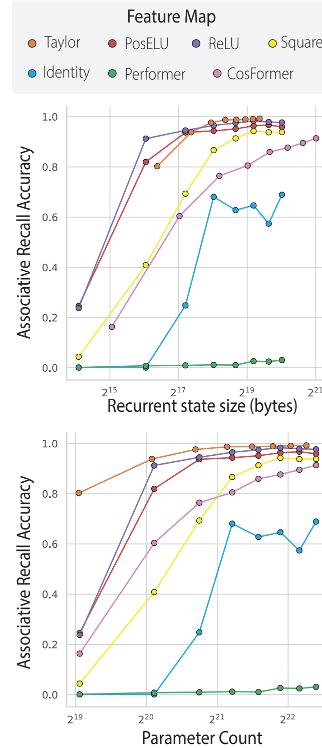

Figure 3: **Linear attention feature maps** x: state size (bytes) during generation or param. count; y: accuracy. State sizes are computed using the equations in Appendix D.2.

4 The Based Architecture

We now present the BASED architecture. BASED combines (1) exact softmax attention applied locally in small sliding windows, and (2) softmax-approximating linear attention applied globally (Fig. 1, right). BASED recovers 90.8% of full softmax attention's recall accuracy at 1e-5× its latency.

4.1 Tensor Core Window Attention

To efficiently model fine-grained local interactions, BASED uses sliding window attention with window sizes set to 64 tokens. Similar to past (causal) implementations (Child et al., 2019; Beltagy et al., 2020), for window size w each query $\boldsymbol{q}_i$ only attends to past keys $\{\boldsymbol{k}_{i-w+1}, \ldots, \boldsymbol{k}_i\}$. This enables $\mathcal{O}(Nw)$ time and space complexity for linear scaling in sequence length N, with a w-sized KV-cache for constant-memory generation.

Efficiency Unlike past sliding window attentions that keep w at sizes 256 (Parmar et al., 2018) to 4096 (Jiang et al., 2023). Large GEMMs are compute bound (e.g., for long-context attention). But using small 16×16 GEMMs also underutilizes the tensor cores since the launch code consumes clock cycles. To saturate the hardware we would like to pipeline calls so BASED sets w to use 64×64 tiles (Figure 1). To distinguish from prior sliding windows, we term this approach TCWINDOW.

4.2 Taylor Approximation Linear Attention

To model long range interactions — unsupported by pure small sliding window attention — BASED combines exact sliding window attention with a softmax-approximating linear attention. With $\phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j) \approx \exp(\boldsymbol{q}_i^\top \boldsymbol{k}_j/\sqrt{d})$, we compute attention for output $\boldsymbol{y}_i$ as $\sum_{j=1}^{i} \frac{\phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j)\boldsymbol{v}_j}{\phi(\boldsymbol{q}_i)\sum_{j=1}^{i}\phi(\boldsymbol{k}_j)} = \frac{\phi(\boldsymbol{q}_i)\sum_{j=1}^{i}\left(\phi(\boldsymbol{k}_j)^\top \boldsymbol{v}_j\right)}{\phi(\boldsymbol{q}_i)\sum_{j=1}^{i}\phi(\boldsymbol{k}_j)}$, which requires $\mathcal{O}(Nd^2)$ time and space (Katharopoulos et al., 2020a). Furthermore, we compute linear attention recurrently for efficient generation. Letting $\boldsymbol{s}_i = \sum_{j=1}^{i} \phi(\boldsymbol{k}_j)^\top \boldsymbol{v}_j$ and $\boldsymbol{z}_i = \sum_{j=1}^{i} \phi(\boldsymbol{k}_j)^\top$ be a "KV-state" and "K-state" respectively, during generation we compute $\boldsymbol{s}_i = \boldsymbol{s}_{i-1} + \phi(\boldsymbol{k}_i)^\top \boldsymbol{v}_i$, $\boldsymbol{z}_i = \boldsymbol{z}_{i-1} + \phi(\boldsymbol{k}_i)^\top$, $\boldsymbol{y}_i = \phi(\boldsymbol{q}_i)\boldsymbol{s}_i/\phi(\boldsymbol{q}_i)\boldsymbol{z}_i$. For ϕ, we use the 2nd-order Taylor series feature map approximating $\exp(\cdot)$ for its strong performance (Zhang et al., 2024), *i.e.*, $\phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j) = 1 + \boldsymbol{q}_i^\top \boldsymbol{k}_j + (\boldsymbol{q}_i^\top \boldsymbol{k}_j)^2/2$. While this naïvely results in $\mathcal{O}(Nd^3)$ complexity, we find smaller heads work without sacrificing quality, *i.e.*, $\boldsymbol{W}_q, \boldsymbol{W}_k \in \mathbb{R}^{d\times d'}$, $d' = 16$. For positional dependence, we use short gated convolutions (filter size 3) and Rotary embeddings (Su et al., 2023) in TCWINDOW.

How does the choice of feature map affect the memory-recall tradeoff? We evaluate a broad set of feature maps ($\phi_{\text{ReLU}}(x) = \max(x, 0)$, $\phi_{\text{PosELU}}(x) = \text{ELU}(x) + 1$, $\phi_{\text{Square}}(x) = x^2$, $\phi_{\text{Identity}}(x) = x$,

[1] In particular, for Mamba Gu & Dao (2023), see Corollary E.16.
[2] Here, we need the entries of the state to be bounded.

$\phi_{\text{CosFormer}}$ as defined in (Qin et al., 2022b), and $\phi_{\text{Performer}}$ as defined in (Choromanski et al., 2020)) using the experimental setup described in Section 3. In Figure 3 (top), we plot the memory-recall tradeoff curves for these feature maps. The Taylor series feature map, along with the simple ϕ_{PosELU} and ϕ_{ReLU} feature maps, sits at the pareto frontier. One advantage of the Taylor feature map over these alternatives is that it expands the recurrent state size (improving recall capacity) without changing the number of parameters. As shown in Figure 3 (bottom), the Taylor series feature map requires fewer parameters than alternatives to achieve high recall capacity. This analysis and the ablations in $Table$ 3 informed our decision to use the Taylor approximation, though other simple feature maps may be effective as well.

IO-Aware Implementation for Real-world Effiency Despite theoretically improved complexity, linear attention is often slower in practice than standard attention (Dao et al., 2022). For number of heads H and effective feature dimension $D = 1 + d' + d'^2/2$, the baseline implementation (Appendix C) materializes large hidden state $KV \in \mathbb{R}^{H \times d \times D}$ in HBM. It further requires $2HND$ bytes for writing featurized $\boldsymbol{Q}, \boldsymbol{K}$ to HBM, $2HND + HNd$ bytes to read $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ tiles for the causal dot product, and HNd bytes to write the result. Our Algorithm 1 drastically reduces the HBM to SRAM data movement, by $O(2HND - 2HNd')$ bytes, and also performs computation in-register, avoiding $\mathcal{O}(Nd^3)$ bytes in SRAM to register data movement. We provide an extended section on our IO-aware algorithms in Appendix C.

5 Results

We evaluate the efficiency and quality of BASED versus strong baselines: Transformer++ (Touvron et al., 2023), Mamba (Gu & Dao, 2023), GLA (Yang et al., 2023), Hyena (Poli et al., 2023), RWKV (Peng et al., 2023), and H3 (Fu et al., 2023a). Extended results are in Appendix B. We find:

1. **Pretraining and downstream performance.** On the Pile (10Bn tokens, same data order across architectures), BASED matches or outperforms the strongest sub-quadratic models at 355M and 1.3Bn parameters. For downstream evaluation on LM Eval Harness (Gao et al., 2023) tasks in Gu & Dao (2023); Yang et al. (2023), BASED likewise competes with strongest baselines (Table 1).
2. **Language modeling recall.** BASED closes the gap to Attention on the challenging associative recall slice of the Pile (see Table 1), outperforming prior sub-quadratic architectures. We apply our pretrained models zero-shot to a suite of recall-intensive tasks (*e.g.* information extraction, QA), showing that BASED outperforms other efficient architectures with comparable throughput.
3. **Efficiency.** For 1.3Bn (360M) parameter models, our BASED IO-aware algorithm achieves 56% (28%) faster prefill than FlashAttention-2 (Dao, 2023) and 44% (76%) faster than Mamba at $4k$ sequence length and 1.3Bn (360M) parameters. It achieves $24\times$ (98%) higher generation throughput (tokens/second) over FlashAttention-2 implementation and achieves 95% and (118%) the throughput of the recurrent Mamba architecture at batch size 128 (Fig. 4, NVIDIA H100).
4. **Architecture ablations** In Table 3, we ablate the key design choices for BASED: (1) feature map, (2) feature dimension, (3) use of sliding windows, and (4) use of short gated-convolutions.

Together, our results point to BASED's promise as a language modeling architecture.

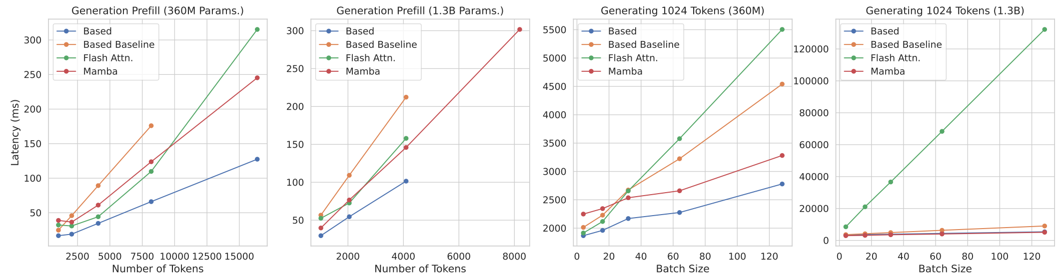

Figure 4: (**Left**) Throughput numbers for the varied prefill sequence lengths at a fixed batch size of 2. (**Right**) Throughput at varied batch sizes for generating 1024 tokens. The y-axis shows the in latency (ms). Lines are cutoff when the model runs out of memory. We show results for both 360M and 1.3Bn params., and all numbers are computed on an NVIDIA H100 GPU.

ACKNOWLEDGMENTS

We thank Daniel Fu, Songlin Yang, Jessica Grogan, Eric Nguyen, Michael Wornow, Alyssa Unell, and Gautam Machiraju for their helpful feedback and discussion during this work. We thank the Hazy Research lab and Together AI for supporting this work. We gratefully acknowledge the support of NIH under No. U54EB020405 (Mobilize), NSF under Nos. CCF2247015 (Hardware-Aware), CCF1763315 (Beyond Sparsity), CCF1563078 (Volume to Velocity), and 1937301 (RTML); US DEVCOM ARL under Nos. W911NF-23-2-0184 (Long-context) and W911NF-21-2-0251 (Interactive Human-AI Teaming); ONR under Nos. N000142312633 (Deep Signal Processing), N000141712266 (Unifying Weak Supervision), N000142012480 (Non-Euclidean Geometry), and N000142012275 (NEPTUNE); Stanford HAI under No. 247183; NXP, Xilinx, LETI-CEA, Intel, IBM, Microsoft, NEC, Toshiba, TSMC, ARM, Hitachi, BASF, Accenture, Ericsson, Qualcomm, Analog Devices, Google Cloud, Salesforce, Total, the HAI-GCP Cloud Credits for Research program, the Stanford Data Science Initiative (SDSI), and members of the Stanford DAWN project: Facebook, Google, and VMWare. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views, policies, or endorsements, either expressed or implied, of NIH, ONR, or the U.S. Government. AR's research is supported by NSF grant CCF#2247014.

REFERENCES

Silas Alberti, Niclas Dern, Laura Thesing, and Gitta Kutyniok. Sumformer: Universal approximation for efficient transformers. *arXiv preprint arXiv:2307.02301*, 2023. A.1.1

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. *International Conference on Learning Representations*, 2023a. 1, 1, 2, 3, A.1.2, D.3, E.1, E.1, E.1, E.2, E.5, E.4, E.9, E.4, E.4, E.5

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:2304.09433*, 2023b. D.3, D.3

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016. A.1.1

Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020. 4.1, A.1.1, A.1.1, D.2

Yonatan Bisk, Rowan Zellers, Ronan Le Bras, Jianfeng Gao, and Yejin Choi. Piqa: Reasoning about physical commonsense in natural language, 2019. B.3

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020. B.1

Peter Bürgisser, Michael Clausen, and Mohammad A Shokrollahi. *Algebraic complexity theory*, volume 315. Springer Science & Business Media, 2013. E.2

Emmanuel Candes, Xiaodong Li, Yi Ma, and John Wright. Robust principal component analysis? *arXiv:0912.3599*, 2009. A.1.1

Beidi Chen, Tri Dao, Eric Winsor, Zhao Song, Atri Rudra, and Christopher Ré. Scatterbrain: Unifying sparse and low-rank attention approximation. *arXiv preprint arXiv:2110.15343*, 2021a. A.1.1

Yifan Chen, Qi Zeng, Heng Ji, and Yun Yang. Skyformer: Remodel self-attention with gaussian kernel and nystr\"om method. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems*, 2021b. URL `https://openreview.net/forum?id=pZCYG7gjkKz`. A.1.1

Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019. 4.1, A.1.1

Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, et al. Rethinking attention with performers. *arXiv preprint arXiv:2009.14794*, 2020. 4.2, A.1.1

Krzysztof Choromanski, Haoxian Chen, Han Lin, Yuanzhe Ma, Arijit Sehanobish, Deepali Jain, Michael S Ryoo, Jake Varley, Andy Zeng, Valerii Likhosherstov, et al. Hybrid random features. *arXiv preprint arXiv:2110.04367*, 2021. A.1.1

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, 2018. B.3

James W Cooley and John W Tukey. An algorithm for the machine calculation of complex fourier series. *Mathematics of computation*, 19(90):297–301, 1965. A.1.2

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. 2023. 1, 2, 3, A.2, C.2.2, C.2.2, D.1

Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems*, 2022. 2, 4.2, A.2.1

Yann N Dauphin, Angela Fan, Michael Auli, and David Grangier. Language modeling with gated convolutional networks. In *International conference on machine learning*, pp. 933–941. PMLR, 2017. A.1.2

Alexandre De Brebisson and Pascal Vincent. An exploration of softmax alternatives belonging to the spherical loss family. *arXiv preprint arXiv:1511.05042*, 2015. 3, A.1.1

Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, et al. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 1, 2021. A.1.2

Daniel Y. Fu, Tri Dao, Khaled K. Saab, Armin W. Thomas, Atri Rudra, and Christopher Ré. Hungry Hungry Hippos: Towards language modeling with state space models. In *International Conference on Learning Representations*, 2023a. 1, 2, 5, A.1.2, B.1, D.1, D.2

Daniel Y. Fu, Elliot L. Epstein, Eric Nguyen, Armin W. Thomas, Michael Zhang, Tri Dao, Atri Rudra, and Christopher Ré. Simple hardware-efficient long convolutions for sequence modeling. *arXiv preprint arXiv:2302.06646*, 2023b. A.1.2, A.2

Daniel Y Fu, Hermann Kumbong, Eric Nguyen, and Christopher Ré. Flashfftconv: Efficient convolutions for long sequences with tensor cores. *arXiv preprint arXiv:2311.05908*, 2023c. A.2, A.2.2

Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, Shawn Presser, and Connor Leahy. The Pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020. 1, 4, D.3

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. A framework for few-shot language model evaluation, 12 2023. URL `https://zenodo.org/records/10256836`. 1, 1, 1, B.3, 4

Albert Gu and Tri Dao. Mamba: Linear-time sequence modeling with selective state spaces. *arXiv preprint arXiv:2312.00752*, 2023. 1, 1, 1, 2, 3, 1, 5, 1, A.1.2, A.2.2, B.1, B.1, B.3, D.1, D.2, E.1, E.2, 5, E.3.2, E.7

Albert Gu, Karan Goel, and Christopher Ré. Efficiently modeling long sequences with structured state spaces. *arXiv preprint arXiv:2111.00396*, 2021. 2, A.1.2, A.2.2

Albert Gu, Ankit Gupta, Karan Goel, and Christopher Ré. On the parameterization and initialization of diagonal state space models, 2022. A.1.2

Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal state spaces are as effective as structured state spaces, 2022. A.1.2

Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc Le. Transformer quality in linear time. In *International Conference on Machine Learning*, pp. 9099–9117. PMLR, 2022. A.2.2

Hanhwi Jang, Joonsung Kim, Jae-Eon Jo, Jaewon Lee, and Jangwoo Kim. Mnnfast: A fast and scalable system architecture for memory-augmented neural networks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 250–263, 2019. A.2.1

Thathachar S Jayram, Ravi Kumar, and Dandapani Sivakumar. The one-way communication complexity of hamming distance. *Theory of Computing*, 4(1):129–135, 2008. E.12

Albert Q Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023. 1, 4.1, A.1.1

Jungo Kasai, Hao Peng, Yizhe Zhang, Dani Yogatama, Gabriel Ilharco, Nikolaos Pappas, Yi Mao, Weizhu Chen, and Noah A. Smith. Finetuning pretrained transformers into RNNs. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 10630–10643, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.830. URL https://aclanthology.org/2021.emnlp-main.830. 2, A.1.1

A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *Proceedings of the International Conference on Machine Learning (ICML)*, 2020a. URL https://arxiv.org/abs/2006.16236. 4.2, A.1.1, A.1.2, 5, D.1, E.6

Angelos Katharopoulos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. Transformers are rnns: Fast autoregressive transformers with linear attention. In *International conference on machine learning*, pp. 5156–5165. PMLR, 2020b. 2, A.1.1, A.2, A.2.2, C.1, E.1

Feyza Duman Keles, Pruthuvi Mahesakya Wijewardena, and Chinmay Hegde. On the computational complexity of self-attention. In *34th International Conference on Algorithmic Learning Theory*, volume 201, pp. 1–23, 2023. 2, 3, A.1.1

Nikita Kitaev, Łukasz Kaiser, and Anselm Levskaya. Reformer: The efficient transformer. *arXiv preprint arXiv:2001.04451*, 2020. A.1.1

Swastik Kopparty. Topics in algorithms and complexity theory: Spring 2020. 2020. E.6.1, E.7.2

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pp. 611–626, 2023. A.2, A.2.1

Hao Liu and Pieter Abbeel. Blockwise parallel transformer for long context large models. *arXiv preprint arXiv:2305.19370*, 2023. A.2.1

Hao Liu, Matei Zaharia, and Pieter Abbeel. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023. A.2, A.2.1

Colin Lockard, Prashant Shiralkar, and Xin Luna Dong. OpenCeres: When open information extraction meets the semi-structured web. In Jill Burstein, Christy Doran, and Thamar Solorio (eds.), *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, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1309. URL https://aclanthology.org/N19-1309. D.3

Xuezhe Ma, Chunting Zhou, Xiang Kong, Junxian He, Liangke Gui, Graham Neubig, Jonathan May, and Zettlemoyer Luke. Mega: Moving average equipped gated attention. *arXiv preprint arXiv:2209.10655*, 2022. A.1.2

Harsh Mehta, Ankit Gupta, Ashok Cutkosky, and Behnam Neyshabur. Long range language modeling via gated state spaces, 2022. A.1.2

Eric Nguyen, Michael Poli, Marjan Faizi, Armin Thomas, Callum Birch-Sykes, Michael Wornow, Aman Patel, Clayton Rabideau, Stefano Massaroli, Yoshua Bengio, Stefano Ermon, Stephen A. Baccus, and Chris Ré. Hyenadna: Long-range genomic sequence modeling at single nucleotide resolution, 2023. B.1

Catherine Olsson, Nelson Elhage, Neel Nanda, Nicholas Joseph, Nova DasSarma, Tom Henighan, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, et al. In-context learning and induction heads. *arXiv preprint arXiv:2209.11895*, 2022. 1, A.1.2

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, 2016. B.3

Niki Parmar, Ashish Vaswani, Jakob Uszkoreit, Lukasz Kaiser, Noam Shazeer, Alexander Ku, and Dustin Tran. Image transformer. In *International conference on machine learning*, pp. 4055–4064. PMLR, 2018. 4.1, A.1.1

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, Kranthi Kiran GV, Xuzheng He, Haowen Hou, Przemyslaw Kazienko, Jan Kocon, and Jiaming et al. Kong. Rwkv: Reinventing rnns for the transformer era. *arXiv:2305.13048*, 2023. 2, 5, A.1.2, B.1, D.1

Michael Poli, Stefano Massaroli, Eric Nguyen, Daniel Y Fu, Tri Dao, Stephen Baccus, Yoshua Bengio, Stefano Ermon, and Christopher Ré. Hyena hierarchy: Towards larger convolutional language models. *arXiv preprint arXiv:2302.10866*, 2023. 2, 5, A.1.2, B.1, D.1, D.2

Zhen Qin, Xiaodong Han, Weixuan Sun, Dongxu Li, Lingpeng Kong, Nick Barnes, and Yiran Zhong. The devil in linear transformer. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 7025–7041, Abu Dhabi, United Arab Emirates, December 2022a. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.473. URL https://aclanthology.org/2022.emnlp-main.473. A.1.1

Zhen Qin, Weixuan Sun, Hui Deng, Dongxu Li, Yunshen Wei, Baohong Lv, Junjie Yan, Lingpeng Kong, and Yiran Zhong. cosformer: Rethinking softmax in attention. *arXiv preprint arXiv:2202.08791*, 2022b. 4.2, A.1.1

Markus N Rabe and Charles Staats. Self-attention does not need $o(n^2)$ memory. *arXiv preprint arXiv:2112.05682*, 2021. A.2.1

Liliang Ren, Yang Liu, Shuohang Wang, Yichong Xu, Chenguang Zhu, and ChengXiang Zhai. Sparse modular activation for efficient sequence modeling. *arXiv preprint arXiv:2306.11197*, 2023. A.1.2

David W. Romero, Anna Kuzina, Erik J. Bekkers, Jakub M. Tomczak, and Mark Hoogendoorn. Ckconv: Continuous kernel convolution for sequential data. 2022. A.1.2

Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale, 2019. B.3

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In *International Conference on Machine Learning*, pp. 9355–9366. PMLR, 2021. 2, A.1.1

Jianlin Su, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding, 2023. 4.2, B.1

Yutao Sun, Li Dong, Shaohan Huang, Shuming Ma, Yuqing Xia, Jilong Xue, Jianyong Wang, and Furu Wei. Retentive network: A successor to transformer for large language models, 2023. 2

Yi Tay, Mostafa Dehghani, Dara Bahri, and Donald Metzler. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6):1–28, 2022. A.1.1

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, and Shruti Bhosale. Llama 2: Open foundation and fine-tuned chat models. *arXiv:2307.09288*, 2023. 1, 5, B.1, D.1

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. volume 30, 2017. 1, D.2

A. Vyas, A. Katharopoulos, and F. Fleuret. Fast transformers with clustered attention. 2020. C.1, 5, D.1

Junxiong Wang, Jing Nathan Yan, Albert Gu, and Alexander M Rush. Pretraining without attention. *arXiv preprint arXiv:2212.10544*, 2022. A.1.2

Sinong Wang, Belinda Z Li, Madian Khabsa, Han Fang, and Hao Ma. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020. A.1.1

Yunyang Xiong, Zhanpeng Zeng, Rudrasis Chakraborty, Mingxing Tan, Glenn Fung, Yin Li, and Vikas Singh. Nyströmformer: A nyström-based algorithm for approximating self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pp. 14138–14148, 2021. A.1.1

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. *arXiv preprint arXiv:2312.06635*, 2023. 5, 1, A.1.2, A.2, A.2.2, B.1, D.1, E.1

Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, and et al. Big bird: Transformers for longer sequences. *Proceedings of NeurIPS*, 2020. A.1.1, A.1.1

Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence?, 2019. B.3

Zhanpeng Zeng, Sourav Pak, Jeffrey Kline, Glenn Fung, and Vikas Sing. Multi resolution analysis (mra) for approximate self-attention. *Proceedings of the 39 th International Conference on Machine Learning*, 2022. A.1.1

Biao Zhang and Rico Sennrich. Root mean square layer normalization. *Advances in Neural Information Processing Systems*, 32, 2019. A.1.1

Michael Zhang, Kush Bhatia, Hermann Kumbong, and Christopher Re. The hedgehog & the porcupine: Expressive linear attentions with softmax mimicry. In *The Twelfth International Conference on Learning Representations*, 2024. URL https://openreview.net/forum?id=4g02l2N2Nx. 2, 2, 3, 4.2, A.1.1, C.1, 5, 6

Chen Zhu, Wei Ping, Chaowei Xiao, Mohammad Shoeybi, Tom Goldstein, Anima Anandkumar, and Bryan Catanzaro. Long-short transformer: Efficient transformers for language and vision. *Advances in neural information processing systems*, 34:17723–17736, 2021. A.1.1

Zhenhai Zhu and Rau Soricut. H-transformer-1d: Fast onedimensional hierarchical attention for sequences. *n Annual Meeting of the Association for Computational Linguistics*, 2021. A.1.1

The appendix is organized as follows:

1. Appendix A includes an extended related works discussion.
2. Appendix E.1 includes a theoretical discussion to complement Section 3 in the main paper.
3. Appendix B includes extended results and ablations to complement Section 5.
4. Appendix C includes details on the IO-aware implementation and benchmarking for BASED.
5. Appendix D provides experimental details.
6. Appendix E includes theoretical results and proofs.

A RELATED WORKS

Our work relates broadly to various developments in efficient sequence modeling. In this section, we organize these related works into (1) model-based or algorithmic contributions (Appendix A.1) and (2) implementation or systems-based contributions (Appendix A.2).

A.1 EFFICIENT LANGUAGE MODELING ARCHITECTURES

While Transformers often achieve state-of-the-art language modeling quality, their design motivates various efficiency improvements when both processing input sequences and generating outputs. In particular, various works try to retain their modeling quality, while improving on their quadratic scaling ($\mathcal{O}(N^2)$ in input sequence length N) when processing inputs and $\mathcal{O}(NM)$ time and space when decoding outputs for outputs of length M (when caching prior keys and values in the attention mechanism).

We note that most related lines of work build on one of two primitives: *attention approximations* (*e.g.,* linear attentions, sparse attentions, sparse and low-rank attentions), or *state-space models* (SSMs) (which have alternative parameterizations as either "long" convolutional models or recurrent neueral networks). Both model classes achieve subquadratic time and space complexity when processing inputs, while linear attentions and SSMs also enable better than $\mathcal{O}(NM)$ decoding via their ability to process inputs recurrently like a recurrent neural network (RNN).

We describe each of these model classes next.

A.1.1 EFFICIENT ATTENTIONS

We focus on two of the most related paradigms for efficiently computing attention here, *structured sparse attentions* and *linear attentions*. We acknowledge a great deal of prior work to compute attention more efficiently, such as via locality-sensitive hashing (Kitaev et al., 2020), random sparse attentions (Zaheer et al., 2020), and sequence compression (Wang et al., 2020; Zhu et al., 2021; Alberti et al., 2023). Please see Tay et al. (2022) for a comprehensive survey.

Structured sparse attentions Structured sparse attentions reduce attention's time and memory requirements by only attending over specific strided patterns or local *sliding windows* (Parmar et al., 2018; Child et al., 2019; Beltagy et al., 2020). For example, Parmar et al. (2018) propose computing attention only over a local window of the past w tokens, such that processing sequences N tokens long only takes $\mathcal{O}Nw$ time and space. Child et al. (2019) note that this window alone may not all capture all desired dependencies (such as long-term interactions), and propose two strided patterns to compute dot products between queries and keys further away. Beltagy et al. (2020) further propose allowing specific tokens to attend to all other tokens in a dense manner.

While further popularized in recent large language models (Mistral, Jiang et al. (2023)), we note that these implementations use large window sizes that still leave room for improving efficiency. In Based, we introduce a hardware-guided design (using small windows) and sliding window implementation that allows us to capitalize on sparse attention's efficiency.

Linear attentions Linear attentions preserve the same "sequence-mixing" operations as standard attention, computing dot products between queries and keys to weight corresponding values. However, their key insight is to replace the softmax in standard attention with alternative kernel functions Katharopoulos et al. (2020b). Mechanically, by removing the $\exp(\boldsymbol{q}^\top \boldsymbol{k})$ in favor of feature map dot-products $\phi(\boldsymbol{q})^\top \phi(\boldsymbol{k})$, these methods use matrix product associativity to compute attention

in $\mathcal{O}(Nd^2)$ time and space (Katharopoulos et al., 2020a). Furthermore, they permit a *recurrent view* for constant memory and $\mathcal{O}(1)$ time per-token generation Kasai et al. (2021); Schlag et al. (2021).

Prior works propose different feature maps ϕ to improve linear attention modeling quality. Katharopoulos et al. (2020b) originally use the *positive elu* function $1+\text{elu}$ such that $\phi(\boldsymbol{q})^\top\phi(\boldsymbol{k})$ remains positive and attention weights remain affine. Qin et al. (2022b) instead use the ReLU function combined with a cosine-based reweighting function to add a locality bias. Other approaches propose feature maps that aim to approximate the Softmax, such as Random Fourier Features (Choromanski et al., 2020; 2021) the Nystrom method (Xiong et al., 2021; Chen et al., 2021b), or deterministic low-degree polynomial approximations (Zhang et al., 2024; De Brebisson & Vincent, 2015; Keles et al., 2023). Finally, recent works treat the feature map as a learnable function (Kasai et al., 2021), and optionally train the feature map explicitly to recover the softmax kernel (Zhang et al., 2024).

Combining sparse and linear attentions Finally, our work is closely related to a long line of work on combining sparse and linear attention. Scatterbrain (Chen et al., 2021a), building on works such as BigBird (Zaheer et al., 2020) and Longformer (Beltagy et al., 2020), shows how a sparse and low-rank approximations can be combined into a single unbiased approximation. This approximation is inspired by robust PCA Candes et al. (2009). As motivation, they show that any low rank approximation of attention's $\exp(QK^T)$ will have a much larger approximation error than a sparse plus low rank approximation. Note that the Scatterbrain method is largely agnostic to the details of any specific architecture or choice of hyperparameters used in the sparse and low-rank approximations. The focus is on how to combine them so as to maintain an unbiased estimate. In contrast, our work studies how the choice of architecture and hyperparameters affect the model's efficiency and quality (we're agnostic to the specific approach for combining the attention). For example, Scatterbrain uses a fixed low-rank approximation (i.e. $\tilde{d} << d$) in experiments. In contrast, we focus on the recall-memory tradeoff and study what happens when we increase the size of d. A major takeaway from our study of this tradeoff is that we actually need $d_{\text{query}} > d_{\text{model}}$ to match attention's recall capacity. Our IO-aware implementation shows how to achieve large speedups even when $d_{\text{query}} > d_{\text{model}}$.

There are a number of other works which can also be viewed as combinations of sparse and linear attention. Multi-resolution analysis attention (MRA-2) uses wavelets to approximate the attention matrix (Zeng et al., 2022). A special form of MRA-2 can be viewed as a combination of sparse and low rank attention for a specific wavelet decomposition. H-transformer-1D uses a hierarchy of matrices including full dense attention on the diagonal and low-rank approximations elsewhere (Zhu & Soricut, 2021). TransNormer (Qin et al., 2022a) apply normalizations such as LayerNorm (Ba et al., 2016) or RMSNorm (Zhang & Sennrich, 2019) to linear attention outputs in certain layers, and apply softmax attention in local chunks in other layers.

A.1.2 Attention Alternatives

We now review other attention alternatives, which focus on improving upon the quadratic scaling of attention. Initial work in this vein uses linear time invariant state space models (SSMs) or long convolutions, which can efficiently process sequences of length N in $O(N \log N)$ time invoking the FFT-convolution theorem (Cooley & Tukey, 1965), as the sequence mixer (Gu et al., 2021; Romero et al., 2022; Gupta et al., 2022; Gu et al., 2022; Mehta et al., 2022; Ma et al., 2022; Wang et al., 2022; Fu et al., 2023b). SSMs can also be rewritten as recurrences to permit fast $O(1)$ inference.

Subsequent work identified that the long convolution alone is not expressive enough to perform particular sub-tasks in language modeling. Prior work shows pure linear SSMs cannot perform associative recall, a skill that is correlated with a model's in-context learning capability Elhage et al. (2021); Olsson et al. (2022), and introduces multiplicative interactions (via gating or Hadamard product (Dauphin et al., 2017)) between tokens to allow the model to compare tokens in the sequence (Fu et al., 2023a; Poli et al., 2023; Peng et al., 2023). However, Arora et al. (2023a) show empirically and theoretically the class of gated convolution architectures, any architectures built from the two gating and convolution primitives, struggles to learn associative recall (on synthetic and real language data) as efficiently as attention. They show that while attention solves AR in constant many layers / with model dimension that is independent of sequence length, any gated convolution architecture uses dimensionality that scales with the sequence length — we build upon their upper bound theoretical results with a lower bound argument in Appendix E.1. We also study a broader set of architectures in this work beyond gated convolutions.

Gu & Dao (2023); Arora et al. (2023a); Yang et al. (2023) identify that the use of *input-dependent* sequence mixers is important for an architecture to perform AR as efficiently as attention. AR requires shifting information that appears prior in a sequence to interact with the current (last) tokens in the sequence, in order to predict the next token (Fu et al., 2023a). While gating is one way to introduce data-dependence (Poli et al., 2023), allowing comparing tokens in two (e.g. a shifted and unshifted) sequences, it is difficult to *select which information* from the prefix of the sequence to shift forwards in the first place, using gating alone. Intuitively, the information to shift *depends on the input's properties*. Thus, several subquadratic architectures consider alternate strategies to introduce input-dependence (Katharopoulos et al., 2020a; Gu & Dao, 2023; Ren et al., 2023; Ma et al., 2022; Yang et al., 2023). We present another strategy for efficient input-dependent sequence mixing in our work.

A.2 Efficient Implementations

Beyond designing new model architectures, various works introduce systems-level innovations to improve training and inference efficiency. These include alternative implementations of architecture primitives such as attention (Dao, 2023; Liu et al., 2023; Kwon et al., 2023), long convolutions (Fu et al., 2023c;b), and linear attention (Katharopoulos et al., 2020b; Yang et al., 2023). They frequently achieve both reduced memory and increased computational speed on modern GPUs by "fusing" operations such as matrix multiplications into a single CUDA kernel, and designing "IO-aware" ways to distribute and compute the results of various read and write operations between different levels of GPU memory.

A.2.1 Efficient Attention Implementations

Dao et al. (2022) introduce FlashAttention, an alternative yet exact implementation of softmax attention that improves memory and speed by both fusing attention operations into a single CUDA kernel and distributing the attention operations to better exploit High Bandwidth Memory (HBM) and Static Random Access Memory (SRAM). They first compute attention's query-key-value dot-products, masking, and softmax, together as a single kernel. By doing so after a single load to SRAM before moving the output back to HRAM, they exploit SRAM's fast compute and reduce the total number of read-write operations. To get around SRAM's small memory size and avoid attention's quadratic memory size over input sequence length, they use *tiling* to split up the query, key, and value inputs into smaller "blocks", compute the attention operations for each block, and adjust the outputs after computing all blocks to properly normalize the softmax (Rabe & Staats, 2021; Jang et al., 2019). To perform backpropagation fast on SRAM, they get around SRAM's limited storage by *recomputing* the gradients rather than storing them. Despite the extra operations, this IO-aware implementation still significantly improves wall-clock time during training.

Similarly making use of block-wise computation, Liu et al. (2023) instead compute attention blocks across different *devices* in RingAttention, enabling training and inference over much larger context lengths that scale with device count. They distribute and compute the attention operations in each block across multiple hosts in parallel, likewise keeping track of summary statistics to gather results correctly into exact attention. However, they introduce an "overlapping" mechanism to coordinate communication of blocks to reduce overhead. They further make use of Blockwise Parallel Transformers (Liu & Abbeel, 2023) to reduce memory, which similar to FlashAttention removes the quadratic in memory scaling of attention by dividing the attention operation into separate blocks before gathering back the adjusted softmax output with block-wise normalization statistics.

As a complement to attention training and inference, Kwon et al. (2023) improve attention generation with PagedAttention. PagedAttention similarly uses block-wise computation to address memory utilization issues during generation, where the KV cache can grow an undetermined amount. Existing systems may naïvely handle this by pre-allocating large amounts of contiguous memory. However, this can result in low utilization and computational bottlenecks. Accordingly, PagedAttention divides attention's growing KV cache into *KV blocks* that can be stored separately on physical memory. This enables more flexible memory management, where smaller chunks can be allocated in different locations when needed to reduce memory-based bottlenecks.

In Based, we use similar blocking strategies to more efficiently compute both the second-order Taylor series linear attention and the sliding window softmax attention, and for both training and inference.

A.2.2 Efficient Attention-Alternative Implementations

Beyond optimizations for attention, various works also introduce similar "IO-aware" implementations to improve memory usage and speed for convolutional and recurrent operations. We overview the most relevant works to Based, which make use of similar techniques such as fusing operations and blocking (tiling) to compute results in SRAM.

Long convolutions Fu et al. (2023c) improve the efficiency of long convolutions on modern GPUs. They build on using the Fast Fourier Transform (FFT), which enables computing convolutions with filter sizes equal to input sequence length from $\mathcal{O}(N^2)$ (if N is filter size and sequence length) to $\mathcal{O}(N \log N)$. However, to compute this algorithm efficiently on GPUs, they break down the convolution into separate matrix multiply operations via a *Monarch* decomposition of the FFT, which allows both (1) fusing multiple steps of the FFT together (for reduced read-write operations) and (2) scheduling these operations for fast computation in SRAM while remaining under the smaller SRAM memory constraints.

Recurrence Gu & Dao (2023) improve the efficiency of recent neural state-space models (SSMs) (Gu et al., 2021) using several similar techniques to FlashAttention, specifically with regard the recurrent view. They load the SSM parameters into SRAM for computation before saving results back in HBM, and also use *recomputation* where during backpropagation the intermediate states are not saved but rather recomputed when inputs are loaded from HBM to SRAM. They finally improve wall-clock time by parallelizing the recurrent view of the SSM as a parallel scan.

Linear Attention Finally, several works propose techniques to improve the real-world wall-clock time and memory-usage of linear attention. Katharopoulos et al. (2020b) fuse several operations in the causal dot product of linear attention. (Yang et al., 2023) use blocking to divide the linear attention matrices into SRAM-computable chunks in FlashLinearAttention. As a trade-off between the slow yet memory-efficient RNN view of linear attention and faster but memory-intensive parallel "standard attention" view, they further optimize a "chunk-wise" implementation of linear attention (Hua et al., 2022). When processing input sequences, the input is first divided into several non-overlapping chunks, where we save memory by computing "kv states" at the end of each chunk, and save time by computing the tokens in a given chunk in parallel.

Model	Params	HG38 PPL ↓ N =1024	N =4096	N =8192
Transformer++	46.2	2.52	2.50	2.51
Mamba	46.1	**2.51**	**2.49**	**2.49**
Based	48.8	**2.51**	2.50	**2.49**

Table 2: **DNA modeling performance on the HG38 dataset.** All models are pretrained from scratch for 10Bn tokens at $N = 1\text{k}$, 4k, and 8k sequence lengths respectively. We report results after hyperparameter sweeping the learning rate for each architecture.

B Extended Results

Here we provide additional discussion of the quality results to complement Section 5.

B.1 Extended Quality Results

In Section 5, we compared to the following baselines: Transformer++ (Llama architecture (Touvron et al., 2023)), which adds rotary encodings (Su et al., 2023) and gated linear units to the standard Transformer architecture (Brown et al., 2020). We compare an early class of efficient architectures built from gating and long-convolution primitives including Hyena (Poli et al., 2023), RWKV (Peng et al., 2023), and H3 (Fu et al., 2023a). We finally compare to recent state-of-the-art architectures, that use input-dependent sequence aggregation to improve in quality upon the long-convolution models, including Mamba (Gu & Dao, 2023) and Gated Linear Attention (Yang et al., 2023).

DNA modeling We evaluate each architecture on its ability to perform DNA next token prediction (Table 2), as in prior work (Nguyen et al., 2023; Gu & Dao, 2023). The DNA tasks uses a byte-level tokenizer wherin the vocabulary consists of characters corresponding to the nucleotide bases. Intuitively, models need to aggregate information across multiple tokens to predict accurately. We evaluate the competitive architectures from Table 2 on the HG38 (human genome) benchmark at $1k$, $4k$, and $8k$ sequence lengths (Nguyen et al., 2023; Gu & Dao, 2023). We find BASED is competitive with SoTA architectures across sequence lengths, generalizing beyond natural language. For all architectures, we hyperparameter search the learning rate $\{2e-4, 4e-4, 8e-4\}$ and select the best. Higher learning rates performed better, so additional hyperparameter tuning could lead to further improvements.

B.2 Ablations of Based Choices

In Table 3, we ablate the key design choices in BASED. We find each design decision leads to meaningful improvements in performance.

B.3 LM Eval Harness Downstream Evaluations

In order to ensure that our models perform comparable on downstream evaluations, we evaluate on the same protocol as Gu & Dao (2023), utilizing the LM evaluation harness by EleutherAI Gao et al. (2023). In particular, we use the following set of metrics and tasks:

- LAMBADA (perplexity and accuracy) Paperno et al. (2016)
- HellaSwag (normalized accuracy) Zellers et al. (2019)
- PIQA (accuracy) Bisk et al. (2019)
- ARC-challenge (normalized accuracy) and, separately, the easy subset ARC-easy (accuracy) Clark et al. (2018)
- WinoGrande (accuracy) Sakaguchi et al. (2019)

Normalized accuracy refers to accuracy normalized by sequence length and is used to maintain the equivalent setting to Gu & Dao (2023). The results can be found in Table 4.

Hyperparameters				Language Modeling (Pile)			Info. Extraction	QA
Feat. Map	Feat. Dim.	Sliding	Convs.	All Ppl. ↓	AR Ppl. ↓	Other Ppl. ↓	FDA Acc. ↑	SQUAD Acc. ↑
Taylor Exp. (2)	16 (153)	✓	✓	8.65	2.07	9.64	11.71	25.07
Performer	16 (16)	✓	✓	9.08	8.53	11.62	0.36	7.47
CosFormer	16 (32)	✓	✓	9.03	2.42	9.98	7.71	24.63
CosFormer	64 (128)	✓	✓	8.82	2.18	9.80	9.07	27.85
Taylor Exp. (2)	8 (45)	✓	✓	8.77	2.18	9.75	12.79	22.35
Taylor Exp. (2)	16 (153)	✓	✓	8.65	2.07	9.64	11.71	25.07
Taylor Exp. (2)	24 (325)	✓	✓	8.58	2.02	9.58	20.87	24.77
Taylor Exp. (2)	32 (561)	✓	✓	8.56	2.00	9.57	12.89	26.74
Taylor Exp. (2)	16 (153)	✓	✓	8.65	2.07	9.64	11.71	25.07
Taylor Exp. (2)	16 (153)	✗	✓	8.91	2.11	9.94	10.16	24.5
Taylor Exp. (2)	16 (153)	✓	✗	8.74	2.09	9.74	2.36	18.87
Taylor Exp. (2)	24 (325)	✗	✗	9.49	2.29	10.58	8.71	11.33

Table 3: **Ablations of design choices and hyperparameters in BASED.** All models are 362M param variants of the BASED architecture described in Section 4, trained to 10 billion tokens on the Pile. We ablate four hyperparameters central to the design of BASED: (1) the choice of feature map ϕ (see Section 4.2), (2) the size of the feature dim d' (we show the effective size of the feature after applying the feature map in parantheses, see Section 4.2), (3) the use of local sequence mixers (sliding window attention and short convolutions).

Architecture	Params.	LAMBADA Ppl. ↓	LAMBADA Acc. ↑	HellaSwag Acc. Norm. ↑	PIQA Acc ↑	Arc-E Acc ↑	Arc-C Acc. Norm. ↑	WinoGrande Acc. ↑	Average Acc. ↑
Transformer (LLaMa)	360m	18.39	42.52	33.48	63.98	46.04	24.49	53.99	44.08
Transformer (Pythia)	356m	25.17	37.16	31.32	63.76	44.82	23.8	51.54	42.08
BASED	363m	21.80	38.66	33.43	64.42	45.79	24.66	51.22	43.03
Mamba	358m	20.23	39.65	33.63	65.02	47.01	25.00	50.75	43.51
H3	362m	57.59	23.58	30.62	63.11	45.20	23.29	50.28	39.35
Transformer (LLaMa)	1.33b	11.12	49.10	39.29	66.16	51.68	26.19	53.43	47.64
BASED	1.35b	12.35	46.96	39.11	66.32	50.72	26.54	50.43	46.68
Mamba	1.32b	13.11	46.13	39.41	66.38	52.36	25.94	50.83	46.84

Table 4: **Downstream evaluation of pre-trained language models.** The same set of models as in Table 1, all were trained on the same 10 billion tokens drawn from the Pile (Gao et al., 2020), evaluated using the LM eval harness by EleutherAI Gao et al. (2023)

.

C IO Aware Implementations

In this section, we provide additional details pertaining to the benchmarking experiments and we provide micro-benchmarking results for the individual kernels we contribute, to complement the end-to-end benchmarking results in Section 5.

C.1 Forward / Generation Prefill

Baselines In Figure 4, we implement BASED using our IO-aware Taylor linear attention Algorithm 1. The baseline approach presented in Zhang et al. (2024), prior to our kernel, uses the popular linear attention CUDA kernel from Fast Transformers for computing the causal dot product Katharopoulos et al. (2020b); Vyas et al. (2020). [3]. The listing below shows the baseline implementation for reference (where line 76-77 can be computed using pure PyTorch or the Fast Transformers kernel).

Micro Benchmark To complement the end-to-end architecture benchmarks in Section 5, we provide micro benchmark results for only the linear attention forward pass in Figure 5.

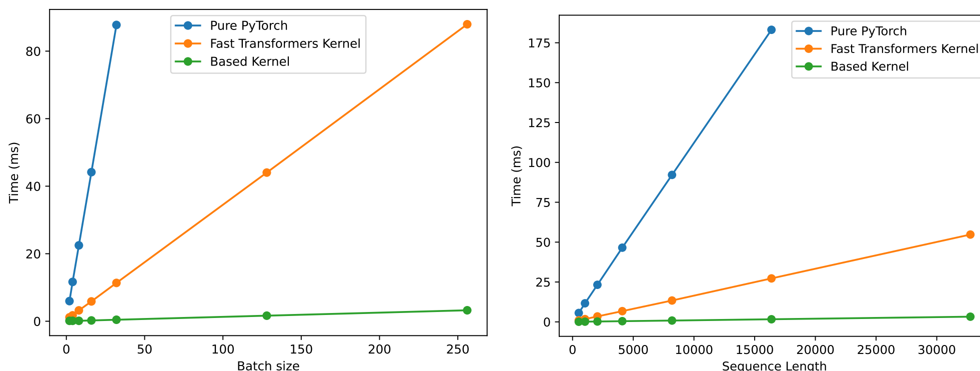

Figure 5: Time (ms) for different ways of computing the Taylor linear attention forward pass — using Pure PyTorch (shown in the Listing and introduced in Zhang et al. (2024)), Fast Transformers kernel (as indicated in the listing) Vyas et al. (2020); Katharopoulos et al. (2020a), or our BASED kernel (Algorithm 1). **(Left)** Varying the batch size at fixed sequence length 1024. **(Right)** Varying the sequence length at fixed batch size 4. **(All)** Benchmarking uses 16 feature dimension, 16 heads, 64 head dimension, and focuses on the *numerator* of the linear attention. Each point represents the median across 10 iterations is measured on a single NVIDIA H100 GPU. Lines terminate on out-of-memory errors.

```
1  from einops import rearrange
2  import torch
3  from torch import nn
4
5  class TaylorExp(nn.Module):
6      """
7      Feature map to compute 2nd-order Taylor approx. of exp(q^T k / sqrt(d
       ))
8      """
9
10     def __init__(self, input_dim, head_dim_idx, temp=None, eps=1e-12):
11         super().__init__()
12
13         self.input_dim = input_dim
14         self.head_dim_idx = head_dim_idx
15         self.temp = 1.0 if temp is None else temp
16         self.eps = eps
17
18         self.r2 = math.sqrt(2)
19         self.rd = math.sqrt(self.input_dim)
```

[3] https://github.com/idiap/fast-transformers/blob/master/fast_transformers/attention/causal_linear_attention.py

```
20         self.rrd = math.sqrt(self.rd)
21 
22     def forward(self, x: torch.Tensor):
23         # Get 2nd-order terms (rearrange(x * x), '... m n -> ... (m n)')
24         x2 = (x.unsqueeze(-1) * x.unsqueeze(-2)).flatten(start_dim=-2) /
    self.r2
25         term1 = torch.ones(x[..., :1].shape).to(x.device)
26         term2 = x / self.rrd
27         term3 = x2 / self.rd
28         terms = [term1, term2, term3]
29         return torch.cat(t for t in terms), dim=self.head_dim_idx)
30 
31 
32 class TaylorLinAttn(nn.Module):
33     def __init__(self):
34         super().__init__()
35         self.d_model = d_model
36         self.feature_dim = 16
37         self.num_heads = 16
38         self.num_key_value_heads = 16
39         self.head_dim = self.d_model // self.num_key_value_heads
40         self.eps = 1e-12
41 
42         feature_map_kwargs = {
43             "input_dim": self.feature_dim,
44             "head_dim_idx": -1,
45             "eps": 1e-12,
46         }
47         self.feature_map = TaylorExp(**feature_map_kwargs)
48         self.proj_q = nn.Linear(
49             self.d_model, self.feature_dim * self.num_heads, bias=False
50         )
51         self.proj_k = nn.Linear(
52             self.d_model, self.feature_dim * self.num_heads, bias=False
53         )
54         self.proj_v = nn.Linear(
55             self.d_model, self.num_key_value_heads * self.head_dim, bias=
    False
56         )
57         self.proj_o = nn.Linear(
58             self.num_heads * self.head_dim, self.d_model, bias=False
59         )
60 
61     def forward(self, hidden_states: torch.Tensor, *args, **kwargs):
62         b, l, _ = hidden_states.size()
63         q = self.proj_q(hidden_states)
64         k = self.proj_k(hidden_states)
65         v = self.proj_v(hidden_states)
66         q = q.view(b, l, self.num_heads, self.feature_dim).transpose(1,
    2)
67         k = k.view(b, l, self.num_key_value_heads, self.feature_dim).
    transpose(1, 2)
68         v = v.view(b, l, self.num_key_value_heads, self.head_dim).
    transpose(1, 2)
69 
70         # Linear attention
71         q, k = self.feature_map(q), self.feature_map(k)
72         q, k, v = q.unsqueeze(-2), k.unsqueeze(-2), v.unsqueeze(-1)
73 
74         # Compute attention causal (alternatively use the Fast
    Transformers kernel)
75         num = (q * (k * v).cumsum(dim=2)).sum(dim=-1)
76         denom = (q * k.cumsum(dim=2)).sum(dim=-1) + self.eps
77         y = (num / denom)
78 
```

```
79        y = rearrange(y, "b h l d -> b l (h d)")
80        y = self.proj_o(y)
81        return y
```

Listing 1: PyTorch implementation of Taylor linear attention.

Algorithm Here we revisit the key equations we aim to compute and then describe Algorithm 1 in detail.

Objective First recall from Section 4:

$$\boldsymbol{o}_i = \sum_{j=1}^{i} \frac{\phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j)\boldsymbol{v}_j}{\phi(\boldsymbol{q}_i)\sum_{j=1}^{i}\phi(\boldsymbol{k}_j)} = \frac{\phi(\boldsymbol{q}_i)\sum_{j=1}^{i}\left(\phi(\boldsymbol{k}_j)^\top \boldsymbol{v}_j\right)}{\phi(\boldsymbol{q}_i)\sum_{j=1}^{i}\phi(\boldsymbol{k}_j)} \tag{1}$$

where q_i reflects the ith of N total tokens in the sequence and every query attends to every past key in $\mathcal{O}(Nd^2)$ time and space complexity for embedding dimension d.

To approximate $\exp(\boldsymbol{q}_i^\top \boldsymbol{k}_j/\sqrt{d})$, we use the 2nd-order Taylor series feature map, picking $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^{d^2}$ such that

$$\phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j) = 1 + \boldsymbol{q}_i^\top \boldsymbol{k}_j + \frac{(\boldsymbol{q}_i^\top \boldsymbol{k}_j)^2}{2} \tag{2}$$

In this section, we will refer to q_i as a **tile** of data (e.g. of 16 tokens) instead of as a single token since the hardware operates on chunks of data in parallel.

Algorithm description In Algorithm 1, we allow each thread block to compute the result for a particular $(\text{batch}, \text{head})$ input. Within the thread block, we use 8 warps / workers to produce the result. We initialize data structures B_q, B_k, B_v in SRAM and $\text{q}_\text{a}, \text{q}_\text{b}, \text{k}_\text{a}, \text{k}_\text{b}, \text{q}_\text{frag}, \text{k}_\text{frag}, \text{v}_\text{frag}$ in register to hold chunks or *tiles* of the q, k, v inputs. We initialize data structures $A0, A1, A2$ in SRAM and a0_frag, a1_accum, qA2_accum in register to hold computation for the running KV state for the $0^{th}, 1^{st}, 2^{nd}$ order Taylor polynomial terms.

We partition the computation along the sequence dimension into n_blocks, where in each loop from 1 to n_blocks, the warps load the next 8 chunks into fast memory. Note that for 2048 sequence length and 8 warps, 16 tile size, we end up with $\text{n}_\text{tiles} = 128$ and $\text{n}_\text{blocks} = 16$. In each iteration, each warp loads in 16×16 tiles of q, k and 16×64 tiles of v, where 16 indicates a chunk of 16 tokens along the sequence dimension and $16, 64$ are the feature and head dimensions respectively. Once tiles are streamed in, we do not need to reuse them, which is key to the efficiency of linear attention.

Zeroeth order Taylor terms: During the computation, for the 0th term in the Taylor polynomial, q, k are 1 after we apply the feature map (Equation (2)). Therefore, computing a cumulative sum over $q(k^T v)$ reduces to maintaining a cumulative sum of v as we iterate across the sequence.

First order Taylor terms: Next we consider the 1st order terms. **On-diagonal:** First consider the on-diagonal blocks, e.g. with respect to tiles q_i, k_i, v_i. For these, we simply multiply $q^T k$, masking (making it causal), and then multiplying with v, following the order of operations in standard attention. This makes it easy to apply the masking (0 out non-causal elements). Now each warp contains a local result for its set of on-diagonal tiles of q_i, k_i, v_i. **Off-diagonal:** However, we need to obtain a *global* cumulative sum where $(q_i^T k_j)v_j$ depends on all $j \in [1..i]$ (Equation (1)). Each warp is therefore missing values for tiles $j \in [1..i-1]$. To incorporate this computation, we will now compute the cumulative KV hidden state for the warp up until $i-1$ and multiply this with the local tile of q (i.e. q_frag). To accomplish this, note in Algorithm 1, we multiply ${\text{k}_\text{frag}}^T$ and v_frag to compute local tiles of the hidden state, local to each warp, in thread register. To perform the global cumulative sum across the 8 warps' local results, we write from registers (thread specific) to $A1$ in SRAM (shared across warp threads). After computing the global cumulative sum in shared memory, each warp loads back the KV state (in $A1$) into its registers such that it contains all the preceeding KV (history) for tiles $[1..i-1]$. We then multiply the local q_frag in register with this KV state to update the final output for the 1st up until the current n_blocks. Note that we maintain the running KV state corresponding to the 1st order term in $A1$ shared memory for the next iteration along n_blocks.

Algorithm 1 Computing the 0th ($T0$), 1st ($T1$), 2nd ($T2$) Order Taylor Linear Attention Terms

Input: Input projected hidden states $q, k, v \in \mathbb{R}^{N\times d}$.
Output: Output $y = T0 + T1 + T2 \in \mathbb{R}^{N\times d}$
Parallelize into batch $\times$ heads parallel computations, with $\mathrm{n_{warps}} = 8$ warps per block.
Within a block:
Define tile size T ▷ $T = 16$ in BASED
Define $\mathrm{n_{tiles}} = \frac{N}{T}$ ▷ Block along the sequence dimension
Define $\mathrm{n_{blocks}} = \mathrm{n_{tiles}}/\mathrm{n_{warps}}$ ▷ Block along the number of warps
Define $\mathrm{tic} = 0, \mathrm{toc} = 1$ ▷ Flags for asynchronous data loading

Create SRAM buffers B_q, B_k (Size $2 \times \mathrm{n_{warps}} \times T \times T$) and B_v (Size $2 \times \mathrm{n_{warps}} \times T \times 4T$)
Create SRAM buffers $\mathrm{A0}, \mathrm{A1}, \mathrm{A2}$ (Size $\mathrm{n_{warps}} \times T \times 4T$) for storing interim. results for $T0, T1, T2$ as warps process the sequence
Create SRAM buffers $total_{A0}$ and $total_{A1}$ to hold cumulative ("KV") state corresponding to $T0, T1$
Create SRAM buffers y of (Size $\mathrm{n_{warps}} \times T \times 4T$) for storing the final output

Create register fragments $\mathrm{q_a}, \mathrm{q_b}, \mathrm{k_a}, \mathrm{k_b}, \mathrm{q_{frag}}, \mathrm{k_{frag}}, \mathrm{qk_{accum}}$ of size 16×16. We create register fragments $\mathrm{v_{frag}}, \mathrm{a0_{frag}}, \mathrm{a1_{accum}}, A2_0, A2_1, \mathrm{qA2_{accum}}, \mathrm{o_{accum}}$ of size 16×64. These fragments are for holding data during in-register computation. Initialize the fragments to 0.
Each warp loads initial tiles $B_q[\mathrm{tic}][\mathrm{warpid}] \leftarrow Q_t$, $B_k[\mathrm{tic}][\mathrm{warpid}] \leftarrow K_t$ and $B_v[\mathrm{tic}][\mathrm{warpid}] \leftarrow V_t$ ▷ HBM into SRAM

for $\mathrm{cur_{block}} \in [0..\mathrm{n_{blocks}} - 1]$; $\mathrm{tic} = 0 \oplus = 1, \mathrm{toc} \oplus 1$ **do** ▷ XORs tic and toc to toggle.
  Warp loads $B_q[\mathrm{toc}][\mathrm{warpid}] \leftarrow Q_t$ for $\mathrm{cur_{block}} + 1$ ▷ HBM to SRAM
  Warp loads $B_k[\mathrm{toc}][\mathrm{warpid}] \leftarrow K_t$ for $\mathrm{cur_{block}} + 1$
  Warp loads $B_v[\mathrm{toc}][\mathrm{warpid}] \leftarrow V_t$ for $\mathrm{cur_{block}} + 1$

  Warp loads $\mathrm{q_{frag}} \leftarrow q[\mathrm{tic}][\mathrm{warpid}]$ ▷ SRAM into register
  Warp loads $\mathrm{k_{frag}} \leftarrow k[\mathrm{tic}][\mathrm{warpid}]$
  Warp loads $\mathrm{v_{frag}} \leftarrow v[\mathrm{tic}][\mathrm{warpid}]$

  Compute the warp-local cumulative sum on $\mathrm{v_{frag}} \rightarrow \mathrm{a0_{frag}}$. ▷ T0 computation
  Add the running $A0$ to the current $\mathrm{a0_{frag}}$

  Compute $\mathrm{q_{frag}}{\mathrm{k_{frag}}}^T$ (attention) and make it causal and store in a $\mathrm{qk_{accum}}$ ▷ T1 computation
  Compute $\mathrm{qk_{accum}}\mathrm{v_{frag}} \rightarrow \mathrm{o_{accum}}$ ▷ Store causal $qk^T v$
  Warps store $\mathrm{k_{frag}^T}\mathrm{v_{frag}} \rightarrow \mathrm{a1_{accum}}$ and write $\mathrm{a1_{accum}} \rightarrow A1[\mathrm{warpid}]$ ▷ Register to SRAM
  Compute cumulative sum over $A1$ in SRAM, updating $A1$ entries
  Warps read $A1$ tiles back to registers ▷ Each warp now contains its preceeding $A1$
  Warps multiply the values in register with $\mathrm{q_{frag}}$ to update $\rightarrow \mathrm{o_{accum}}$ ▷ Add in T1 to the running result
  Update $\mathrm{a0_{frag}} \rightarrow \mathrm{o_{accum}}$ ▷ Add in T0 to the running result
  Square $\mathrm{qk_{accum}}$, multiply with $\mathrm{v_{frag}}$ and add $\rightarrow \mathrm{o_{accum}}$ ▷ Add in diagonal T2 to the running result
  Sum the values of $\mathrm{o_{accum}}$ into $y[\mathrm{warpid}]$

  for block in $\mathrm{n_{warps}}$ iterations **do** ▷ Remaining T2 computation; Assumes feature dimension 16
    Each of 8 warps copies the *same* slice of $q[\mathrm{tic}][\mathrm{warpid}]$ to 2 registers q_a, q_b
    Each thread j in the warp computes $q_a[:, 2j]q_a$ for dimension $2j$, and for $2j+1$ (and for q_b). Together the threads compute the 256 elements resulting from the second order outer product in the feature map.
    Each warp stores two slices of $A2$: $A2_0$ and $A2_1$ ▷ Partitioning the large $A2$ across warp registers
    Accumulate both $q_a A2_0$ and $q_b A2_1 \rightarrow \mathrm{qA2_{accum}}$
    Warp writes $\mathrm{qA2_{accum}} \rightarrow A2[\mathrm{warpid}]$ ▷ Register to SRAM
    Sum results across all in $A2[\mathrm{warpid}]$ and store the sum in $y[\mathrm{block}]$ ▷ Add in T2

    Each of 8 warps copies the *same* slice of $k[\mathrm{tic}][\mathrm{block}]$ to 2 registers k_a, k_b ▷ KV state update
    Square k_a and k_b
    Each of the 8 warps loads $v[\mathrm{tic}][\mathrm{block}]$ to $\mathrm{v_{frag}}$ in register
    Multiply k_a and $\mathrm{v_{frag}}$, k_b and $\mathrm{v_{frag}}$ and accumulate the results into $A2_0$ and $A2_1$, the two in-register slices of $A2$ for the warp, respectively

End. Store y. Optionally store $A0$, $A1$, $A2$ (comprising the "KV state") for generation. ▷ SRAM to HBM

Second order Taylor terms: We finally need to compute the 2nd order term. Similar to the 1st order term, we'll consider **On-diagonal:** We can leverage the computation from above. We'll square the causal $(qk^T)^2$ from above and multiply with $\mathrm{v}_{\mathrm{frag}}$ to obtain the *portion* of the 2nd order term corresponding to the on-diagonal tiles q_i, k_i, v_i. **Off-diagonal:** Again, we also need to compute the result with respect to tiles $[1..i-1]$.

- **Partitioning KV hidden state for 2nd order** Because the hidden state for the second order term is large ($\mathcal{O}(d^2 D)$ in feature dimension d and head dimension D) and warps have a limited number of registers, we slice its storage across the registers of the 8 warps. Considering the the $16^2 \times 64$ ($d^2 \times D$) hidden state (stored in $A2$ SRAM in Algorithm 1), we divide this into 16 slices along the sequence dimension and let each of the 8 warps handle 2 of the 16×64 slices (stored in $A2_0, A2_1$ fragments in thread registers in Algorithm 1). Warp i will maintain slices $2i$ and $2i+1$ in two registers per thread.
- **Computing output for 2nd order** Each warp i loads in one tile of q_i into 2 registers. We will use the 32 threads in the warp to compute the 256 outer product terms for each token computed by the Taylor 2nd order term (for feature dimension 16).

  Next, the threads multiply these 256 terms with the running $A2_0$ and $A2_1$ slices. The results for the two slices are summed in register and then stored in SRAM ($A2[\mathrm{warpid}]$). Since o_i is ultimately the sum of q_i terms multiplied with *all* slices of $A2$ (Equation (1)), we then sum the results from all the warps together (which hold the remaining slices of $A2$) and store the result in $y[\mathrm{block}]$. We can think of $y[\mathrm{block}]$ as holding the result up until the $(8 \times \mathrm{cur}_{\mathrm{block}} + \mathrm{block})$ tile of tokens (note 8 is because in each increment of $\mathrm{cur}_{\mathrm{block}}$, the 8 warps handle 8 different tiles of the sequence).
- **Updating the KV state:** For $\mathrm{block} = i$, we load in $k[i], v[i]$ tiles of size 16×16 and 16×64 respectively to registers $k_a, k_b, \mathrm{v}_{\mathrm{frag}}$. We compute the 256 outer product terms on $k[i]$ using the 32 threads, multiply with $\mathrm{v}_{\mathrm{frag}}$, and store the result in the $A2_0, A2_1$ running state.

The final result in y is summed into the output to complete the 2nd order computation.

C.2 Next Token Prediction

During next token prediction in generation, we use IO-aware algorithms for the expensive KV state update in Taylor linear attention and for the sliding window attention computation.

C.2.1 Taylor linear attention

The KV update in PyTorch is provided in the following listing. In Figure 6 we benchmark the speed of the PyTorch implementation against our kernel.

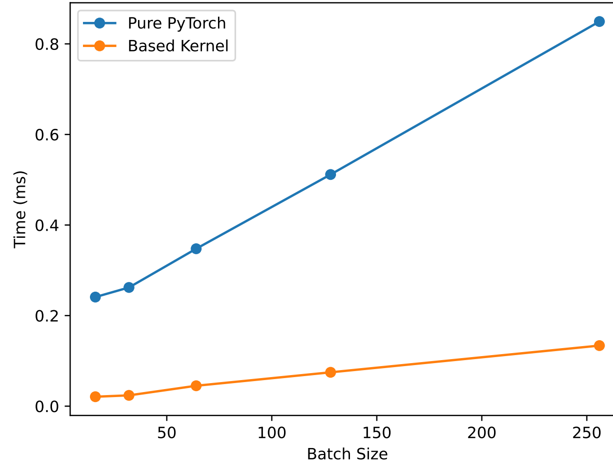

Figure 6: Time (ms) for computing the Taylor linear attention recurrent update using Pure PyTorch (shown in the Listing and introduced in Zhang et al. (2024)) vs. our Based kernel. Benchmarking uses 16 feature dimension, 16 heads, 64 head dimension, and focuses on the *numerator* of the linear attention, the more expensive term. Each point represents the median across 10 iterations is measured on a single NVIDIA H100 GPU.

```
1  from einops import rearrange
2  import torch
3  from torch import nn
4
5  def step(self, kv_state: torch.Tensor, k_state: torch.Tensor, q: torch.
       Tensor, k: torch.Tensor, v: torch.Tensor):
6          """
7          Compute linear attention with recurrent view
8          -> Assume q.shape is (b, h, 1, D); k and v.shape are (b, h, l, d)
       , where D is the dimension after applying the feature map and d is
       the head dimension.
9          """
10         b, h, l, d = q.shape
11         assert l == 1, f'q.shape is {q.shape} but should be ({b}, {h}, 1,
        {d})'
12         # Expand dims for broadcasting to compute linear attention
13         q, k, v = q.unsqueeze(-2), k.unsqueeze(-2), v.unsqueeze(-1)
14
15         kv_state += k[:, :, -1:] * v[:, :, -1:]
16         k_state  += k[:, :, -1:]
17
18         # Compute linear attention
19         num = (q * kv_state).sum(dim=-1)
20         y = num / ((q * k_state).sum(dim=-1) + self.eps)
21
22         y = rearrange(y, 'b h l d -> b l (h d)').to(q.dtype)
23         return self.dropout(self.out_proj(y))
```

Listing 2: PyTorch implementation of Taylor linear attention KV update

Algorithm 2 Computing KV State Updates

Input: KV_{t-1} state $\in \mathbb{R}^{Hd'^2d}$, at time t. Featurized $q, k \in \mathbb{R}^{B\times H\times 1\times D}$ and $V \in \mathbb{R}^{B\times H\times 1\times d}$, for d as the head dimension (e.g. 64) and D as the expanded feature map dimension (e.g. $273 = 1+16+16^2$ for feature dim 16). To be hardware-friendly, we let $D = 320$ (s.t. $320 \mod 64 = 0$) via padding.
Output: Updated KV_t state.

Parallelize into batch $\times$ heads parallel computations, with $\mathrm{n_{warps}} = 8$ warps per block.
Within a block:
Define $\mathrm{n_{threads}} = \mathrm{n_{warps}} \times 32$ ▷ Assuming 32 threads per warp
Define $\mathrm{buffer_{size}} = \mathrm{n_{warps}} \times 8 \times d$
Define $\mathrm{total_{batches}} = \frac{D}{\mathrm{n_{warps}} \times 8}$ ▷ E.g. $\mathrm{total_{batches}} = 5$ if $D = 320$; For k, $\frac{320}{5} = 64$ values per batch
Define $\mathrm{tic} = 0, \mathrm{toc} = 1$

Create SRAM buffer B_q (Size D) for q
Create SRAM buffer B_k (Size D) for k
Create SRAM buffer B_v (Size d) for V
Create SRAM buffer B_{kvs} (Size $2 \times \mathrm{buffer_{size}}$) for storing blocks of $\mathrm{kv_{state}}$
Create SRAM buffer o (Size d) for output.
Create SRAM buffer A (Size $\mathrm{n_{warps}} \times d$) for intermediate computation

Create register buffer $\mathrm{v_{reg}}$ (Size 2) to store V data
Create register $\mathrm{A_{reg}}$ (Size 2) for intermediate computation

Warps load $B_q \leftarrow q$ ▷ HBM to SRAM; Load all $D = 320$ elements of q
Warps load $B_k \leftarrow k$
Warps load $B_v \leftarrow V$
Warps load chunk $B_{kvs}[\mathrm{tic}] \leftarrow \mathrm{kv_{state}}$ ▷ Load $(1 \times 64) \times 64$ of the $(\mathrm{total_{batches}} \times 64) \times 64$ elements in KV_{t-1}

Initialize $m = 0$
for Threads $j \in [0..31]$; $j < d$; $j+ = 32, m+ = 1$ **do** ▷ Each thread holds 2 values ($d = 64$; 32 threads)
  Load $\mathrm{v_{reg}}[m] \leftarrow v[j]$ ▷ SRAM to Register; Now $v[j]$ is stored in thread $j \mod 32$

for $i \in [0..\mathrm{total_{batches}}]$; $i = i + 1$, $\mathrm{tic} \oplus 1$, $\mathrm{toc} \oplus 1$ **do**
  Loads $B_{kvs}[\mathrm{toc}] \leftarrow$ next batch of $\mathrm{kv_{state}}$ ▷ Asynchronous loads of next batch

  for $j = \mathrm{warpid}$; $j < d$; $j+ = \mathrm{n_{warps}}$ **do** ▷ Each of the 8 warps loads 8 of the 64 *rows* of k, q in the batch
    $\mathrm{k_{val}} \leftarrow B_k[i * d + j]$ ▷ Grab single rows $q[i]$ and $k[i]$, Broadcast to all threads
    $\mathrm{q_{val}} \leftarrow B_q[i * d + j]$

    $p = B_{kvs}[\mathrm{tic}] + j * d$ ▷ Point to output rows of KV_t; We write $d \times \frac{D}{\mathrm{total_{batches}}}$ sub-matrix for this batch
    Initialize $m = 0$
    for Thread $k \in [0..31]$; $k < d$; $k+ = 32, m+ = 1$ **do**
      $p[k]+ = \mathrm{k_{val}} * \mathrm{v_{reg}}[m]$ ▷ Update running state by multiplying broadcasted $\mathrm{k_{val}}$ with the full $\mathrm{v_{reg}}$
      ▷ This updates a $1 \times d$ strip of the $d \times D$ full KV_t outer product
      $\mathrm{A_{reg}}[m]+ = \mathrm{q_{val}} * p[k]$ ▷ Multiply $\mathrm{q_{val}}$ with the running state, updating all values in the $1 \times d$ output

  Write out new KV_t state for this batch: $B_{kvs}[\mathrm{tic}][k]$ ▷ SRAM to HBM

Initialize $m = 0$
for Threads $j \in [0..31]$; $j < d$; $j+ = 32, m+ = 1$ **do** ▷ Each thread holds info for 2 of the 64 output values
  Store $A[\mathrm{warpid}][j] \leftarrow \mathrm{A_{reg}}[m]$ ▷ Register to SRAM

for Thread j; $j < d$; $j+ = \mathrm{n_{threads}}$ **do** ▷ $d = 64$ threads put values from first warp in n_j
  $n_j = A[0][j]$ ▷ Each warp had only computed output values for a subset of (e.g. 8) rows of k and q
  for $w \in [0..\mathrm{n_{warps}}]$ **do**
    Sum the $n_j + = A[w][j]$ across ▷ Need to combine results across warps
  Store $o[j] \leftarrow n_j$
Write output o ▷ SRAM to HBM

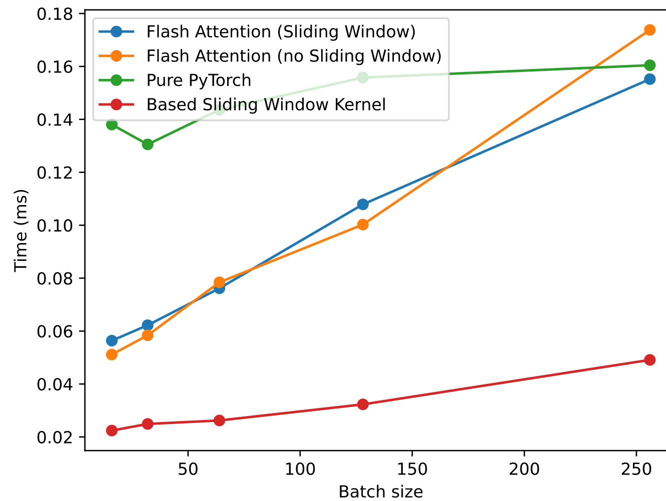

Figure 7: Time (ms) for different ways of computing sliding window attention next token prediction — using PyTorch, Flash Attention (which supports a sliding window function), or our inference kernel. Each point represents the median across query tokens at different token positions in the generation $\in \{100, 250, 500, 750\}$.

C.2.2 Sliding window attention

Baselines During training / prefill, we use the Flash Attention sliding window implementation (Dao, 2023).

Our IO-aware implementation focuses on next token prediction. In the listing below, we include a Torch reference. Our IO-aware sliding window attention algorithm is provided in 3. The key insight is to fuse operations *in thread registers* to minimize slower SRAM to register data movement.

Micro Benchmark We benchmark key baselines (Torch, Flash Attention-2 Dao (2023), and the Based kernel on an NVIDIA H100 GPU in Figure 7. The benchmark uses window size 64, head dimension 64, and number of heads 16. We vary the batch size on the x axis and repeat the median timing across iterations on the y axis. Note that these timings include only the attention computation and not the time for updating the KV-cache state. These timings also do not include any processing for Rotary encodings (as shown below).

```
import torch
from torch import nn

"""
b: batch size
h: number of heads
n: sequence length
d: head dimension

w: window size

qw: b x h x 1 x d
kw: b x h x w x d
vw: b x h x w x d
"""

w = torch.einsum("bhod, bhnd-> bhn",qw, kw)
a = torch.nn.functional.softmax(w, dim=-1)
result = torch.einsum("bhn,bhnd->bhd", a, vw)
```

Listing 3: PyTorch implementation of Sliding Window

Algorithm 3 Sliding window generation

Input: KV_{t-1} state $\in \mathbb{R}^{Hwd}$, at time t and projected hidden states $q, k, v \in \mathbb{R}^{B\times H\times 1\times d}$, for H heads, head dimension d, sliding window size w, and batch size B.
Output: Updated KV_t state.
Parallelize into batch $\times$ heads parallel computations, with $\mathrm{n_{warps}} = 4$ warps per block.
Within a block:
Define tile size T ▷ $T = 16$ in BASED
Define $\mathrm{n_{threads}} = \mathrm{n_{warps}} \times 32$ ▷ Assuming 32 threads per warp

Create SRAM buffers B_k and B_v (Each of size $4T \times 4T$) to hold k, v. ▷ Assumes $4T = 64$ is the w, d
Create SRAM vector B_q (Size $1 \times 4T$) to hold q during the kernel execution. ▷ Single query, assume $d = 64$
Create SRAM vector B_w (Size $1 \times 4T$) of type float for intermediate attention computation.
Create SRAM vector B_o (Size $1 \times 4T$) to hold the output. ▷ Single output, assume $d = 64$
Create SRAM buffers max and sum (Each of workers by float size).

Create register fragments $\mathrm{q_{reg}}$, $\mathrm{k_{reg}}$, $\mathrm{v_{reg}}$ to hold data during fused computation in-register.
Create register fragments $\mathrm{w_{reg}}$ (size $1 \times 4T$) and $\mathrm{wv_{reg}}$ (size $4T \times 1$) to store intermediate computation in-register.
Create register fragment $\mathrm{o_{reg}}$ (size $4T \times 1$) to store output in-register.

Loads $B_k \leftarrow k$ using $\mathrm{n_{threads}}$; $B_v \leftarrow v$ using $\mathrm{n_{threads}}$; $B_q \leftarrow q$ using one warp. ▷ HBM to SRAM
Loads $\mathrm{q_{reg}} \leftarrow B_q$. q gets broadcasted to all warps. ▷ SRAM to Register
Loads $\mathrm{k_{reg}} \leftarrow B_k[\mathrm{warpid}]$. Each warp gets $T \times 4T$ of the $4T \times 4T$ in B_k (*i.e.* a column).
Loads $\mathrm{v_{reg}} \leftarrow B_v[\mathrm{warpid}]$. Each warp gets $T \times 4T$ of the $4T \times 4T$ in B_v (*i.e.* a column).

Initialize w_{reg} to zero
$\mathrm{w_{reg}} \leftarrow \mathrm{q_{reg}}\mathrm{k_{reg}}$ ▷ Matrix-vector (GEMV) multiplication

Initialize float $m = -\infty$ for the max ▷ Obtain the max across tiles for Softmax
Update $m \leftarrow \max(\mathrm{w_{reg}})$ with the max from the local data
max[warpid] $\leftarrow m$ for all warps to access
Iterate over $\mathrm{n_{warps}}$ entries in max buffer to compute the global max of $\mathrm{w_{reg}}$
Put global max back into each warp's m float

Initialize float $s = 0$ for the sum ▷ Obtain the sum across tiles for Softmax
Update $s \leftarrow \mathrm{sum}(\mathrm{w_{reg}})$ with the sum from the local data
$\mathrm{sum}[\mathrm{warpid}] \leftarrow s$ for all warps to access
Iterate over $\mathrm{n_{warps}}$ entries in sum buffer to compute the global sum of $\mathrm{w_{reg}}$
Put global sum back into each warp's s float

$\mathrm{w_{reg}} \leftarrow \mathrm{w_{reg}} - m$ ▷ Start attention computation in register
$\mathrm{w_{reg}} \leftarrow \exp(\mathrm{w_{reg}})$
$\mathrm{w_{reg}} \leftarrow \frac{\mathrm{w_{reg}}}{s}$
$B_w[\mathrm{warpid}] \leftarrow \mathrm{w_{reg}}$ ▷ Register to SRAM; storing for the slice of k
$\mathrm{wv_{reg}} \leftarrow B_w$ ▷ SRAM to Register. Warp loads entirety of B_w; all slices
Initialize $\mathrm{o_{reg}}$ to zero.
$\mathrm{o_{reg}} \leftarrow \mathrm{wv_{reg}}\mathrm{v_{reg}}$ ▷ Matrix-vector (GEMV) multiplication

Write $\mathrm{o_{reg}}$ to global memory ▷ Register to SRAM, SRAM to HBM

D Experimental Details

D.1 Language Model Pretraining

We use A100 80GB Nvidia GPUs to run all experiments. We use training infrastructure closely adapted from the FlashAttention code base: https://github.com/Dao-AILab/flash-attention/tree/main for all pretraining runs Dao (2023). The Pile data is tokenized using the GPT2BPETokenizer and all models see the data in the same order. Here we provide details on the hyperaparamters and configurations used for training each architecture. We also provide details on the FLOPs computation.

- **Based** We train using the specifications in Table 5. Our implementation is provided here: https://github.com/HazyResearch/based. The initial models were trained and evaluated using the Fast Transformer CUDA kernels discussed in Appendix C (Vyas et al., 2020; Katharopoulos et al., 2020a).
- **Transformer++ (Touvron et al., 2023)** We refer to the modern Llama architecture with Rotary encodings, RMSNorm and SwiGLU as Transformer++, following prior work Gu & Dao (2023); Yang et al. (2023). We train using the the specifications in Table 6 using the Flash Attention training code provided here: https://github.com/Dao-AILab/flash-attention/tree/main Dao (2023).
- **Mamba (Gu & Dao, 2023)** We train using the specifications in Table 7, where the parameters are sourced from the Appendix of Gu & Dao (2023). The implementation is sourced from the provided reference at https://github.com/state-spaces/mamba.
- **Hyena (Poli et al., 2023)** We train using the specifications in Table 8, where the parameters are sourced from the Appendix of Poli et al. (2023). The implementation is sourced from the provided reference at https://github.com/HazyResearch/safari.
- **H3 (Fu et al., 2023a)** We train using the specifications in Table 9. The implementation is sourced from the provided reference at https://github.com/HazyResearch/safari.
- **RWKV (Peng et al., 2023)** We train using the specifications in Table 10 and use the reference implementation at https://github.com/BlinkDL/RWKV-LM. We specifically evaluate RWKV-V5.
- **Gated Linear Attention (GLA)** We train using the specifications in Table 11. We train following the reference implementation at https://github.com/berlino/gated_linear_attention.

We give all models the improved Transformer++ recipe (e.g., SwiGLU) as relevant.

D.2 Computing Recurrent State Size

In this section, we provide details on how we compute the size of the recurrent hidden state for the results described in Section 3. We train and evaluate six sequence mixers on a synthetic associative recall task: attention Vaswani et al. (2017), sliding window attention Beltagy et al. (2020), Mamba Gu & Dao (2023), H3 Fu et al. (2023a), Hyena Poli et al. (2023), and Based. For each, we vary hyperparameters that affect the memory consumption during inference. We compare how MQAR accuracy varies with the size of the recurrent hidden state.

Based. The recurrent state size in Based is determined by the model dimension d and the size of the hidden dimension after applying the feature map $\tilde{d}$. The $+1$ accounts for the K-state required for computing the denominator. For more details on the recurrent view of Based, see 4.

$$\text{sizeof}(\boldsymbol{s}_i) = (d+1) \times \tilde{d} \tag{3}$$

In Based, we use the Taylor Exponential feature map after projecting d down to a smaller dimension d'. With this approach, recurrent state size is given by:

$$\text{sizeof}(\boldsymbol{s}_i) = (d+1) \times (1 + \frac{3d'}{2} + \frac{d'^2}{2}) \tag{4}$$

In our synthetic experiments, we run Based with $d \in \{48, 64, 128\}$ and $d' \in \{8, 16, 24\}$.

Attention. The recurrent state size (*i.e.* KV-cache size) in attention depends on two parameters: the model dimension d and the sequence length N. The 2 in the expression below accounts for the separate storage for keys and values in the KV-cache.

$$\text{sizeof}(\boldsymbol{s}_i) = 2 \times d \times N \tag{5}$$

In our synthetic experiments we run attention with $d \in \{64, 128\}$. The sequence length N is determined by the task, not the model architecture.

Sliding window attention. The recurrent state size in sliding window attention is given by the model dimension d and the width of the sliding window k_{sliding}. The 2 in the expression below accounts for the separate storage for keys and values in the KV-cache.

$$\text{sizeof}(\boldsymbol{s}_i) = 2 \times d \times \min(N, k_{\text{sliding}}) \tag{6}$$

In our synthetic experiment we run sliding window attention with $d \in \{128\}$ and $k_{\text{sliding}} \in \{8, 16, 32, 64, 128, 256, 512, 1024\}$.

Mamba. The recurrent state size in Mamba is determined by the model dimension d and the number of heads h. The 2 in the expression below accounts for the expansion in the Mamba block.

$$\text{sizeof}(\boldsymbol{s}_i) = 2 \times d \times d_{\text{state}} \tag{7}$$

In our synthetic experiments, we run Mamba with $d \in \{64, 128, 256\}$ and $d_{\text{state}} \in \{8, 16, 24\}$.

H3. The recurrent state size in H3 is determined by the model dimension d and the number of heads d_{state}.

$$\text{sizeof}(\boldsymbol{s}_i) = d \times d_{\text{state}} \tag{8}$$

In our synthetic experiments, we run H3 with $d \in \{64, 128, 256\}$ and $d_{\text{state}} = \frac{d}{4}$.

Hyena. The recurrent state size in Hyena is determined by the model dimension d and the number of heads h. The 2 in the expression below accounts for the separate storage for keys and values in the KV-cache.

$$\text{sizeof}(\boldsymbol{s}_i) = d \times N \tag{9}$$

In our synthetic experiments, we run Hyena with $d \in \{64, 128, 256\}$.

D.3 Language Model Evaluation

In this section, we provide details on each of the evaluations (columns) reported in Tables 1 and 3.

Pile *(Language Modeling).* First, we report overall perplexity on the Pile test set Gao et al. (2020). Then , to understand how much of the perplexity gap is due to recall capacity, we also evaluate perplexity on two slices (*i.e.* subsets) of the test set:

1. *Associative recall(AR) tokens.* Tokens in the final position of a bigram which previously occured in context, but ≤ 1250 times in the training data.
2. *Other tokens.* All other tokens.

To construct these slices, we exactly follow the protocol in Arora et al. (2023a) and refer the reader to that work for more details. We compute these slices on the first 16 million tokens in the test set.

SWDE *(Information Extraction).* The task in the SWDE benchmark is to extract semi-structured relations from raw HTML websites. For example, given an IMBD page for a movie (*e.g. Harry Potter and the Sorcerer's Stone*) and a relation key (*e.g.* release date), the model must extract the correct relation value (*e.g.* 2001). The SWDE benchmark was originally curated by Lockard et al. (2019) for the task of open information extraction from the semi-structured web. Because we are evaluating the zero-shot capabilities of relatively small language models, we adapt the task to make it slightly easier. Our task setup is similar after to that used in Arora et al. (2023b).

FDA *(Information Extraction).* The task is to extract key-value pairs from a set of PDFs scraped from the FDA website. We use the dataset and labels collected in Arora et al. (2023b). We break apart the documents into chunks of 1,920 tokens. For every key-value pair that appears in the chunk, we create a zero-shot prompt using the simple prompt template:

```
{chunk} \n {key}:
```

We allow the model to generate a fixed number of tokens after the prompt and check (with case insensitivity) if the value is contained within the generation. We report **accuracy**, the fraction of prompts for which the generation contains the value.

Below we include one example of a zero-shot prompt for the key-value pair "*Type of Test: Quantitative, colorometric, pyranose oxidase (PROD)*". The actual chunk is substantially longer in the dataset (note the ellipsis).

> 510(k) SUBSTANTIAL EQUIVALENCE DETERMINATION DECISION SUMMARY ASSAY ONLY TEMPLATE A. 510(k) Number: k180209 B. Purpose for Submission: New Device C. Measurand: 1,5-Anhydroglucitol (1,5-AG) D. **Type of Test: Quantitative, colorometric, pyranose oxidase (PROD)** E. Applicant: Diazyme Laboratories Inc. F. Proprietary and Established Names: Diazyme 1,5-AG Assay G. Regulatory Information: 1. Regulation section: 21 CFR 864.7470; Glycosylated hemoglobin assay 2. Classification: Class II ... *[1,920 tokens of context from the PDF]* ... Diazyme's 1,5-AG assay uses the enzyme pyranose oxidase (PROD) to oxidize the 2nd position hydroxyl group of 1,5-AG and to detect the generated hydrogen peroxide by colorimetry using peroxidase (POD). **Type of Test:**

SQUAD *(Question Answering).* The Stanford Question Answering Dataset (SQUAD) can be used to evaluate the reading comprehension of language models. The model is given a passage of text and a question whose answer is contained in the passage.

Because the models trained in this work are relatively small-scale (up to 1.3 billion parameters trained on 10 billion tokens) and not instruction fine-tuned, they struggle to answer questions when asked directly. To make the task more amenable to these raw language models, we first use GPT-4 to reformat the questions to more closely resemble the next-token-prediction task the models were trained on:

```
 Can you rewrite this question and answer as a statement.
Ensure that the answer is the last part of the statement.  \n \n
Question:  {question} \n\n Answer:  {answer} \n\n Rewrite:
```

For example, the question and answer "*Question: Which NFL team represented the AFC at Super Bowl 50? Answer: Denver Broncos*" was rewritten by GPT-4 as "*The NFL team that represented the AFC at Super Bowl 50 was the Denver Broncos.*" We verify that the rewritten sentence does indeed end with the answer, discarding any sentences where it does not (40% of questions).

We run the reformatting on 5,000 squad questions from the validation set, yielding a final dataset of **2,984 questions** formatted as next token predictions.

Below we include one example of a zero-shot prompt. The reformatted question is in bold.

> For the third straight season, the number one seeds from both conferences met in the Super Bowl. The Carolina Panthers became one of only ten teams to have completed a regular season with only one loss, and one of only six teams to have acquired a 15–1 record, while the Denver Broncos became one of four teams to have made eight appearances in the Super Bowl. The Broncos made their second Super Bowl appearance in three years, having reached Super Bowl XLVIII, while the Panthers made their second Super Bowl appearance in franchise history, their other appearance being Super Bowl XXXVIII. Coincidentally, both teams were coached by John Fox in their last Super Bowl appearance prior to Super Bowl 50. **The team in Super Bowl 50 that had a 15-1 record was the**

E Theoretical Results

E.1 Key Results

To complement the empirical results on the throughput-recall tradeoffs in Section 3, here we theoretically show lower bounds on attention-free alternativess' abilities to perform recall. We prove that existing gated convolutions and recurrent models require scaling in size to solve recall over increasing vocabularies. This analysis reinforces our empirical observations on the throughput-recall tradeoff.

Prior work formally shows how different architecture classes solve the MQAR problem Arora et al. (2023a). They show that different sequence mixers need asymptotically different parameter scaling to solve MQAR. The broader implication is that even though a sequence mixer might *process* long-context in sub-quadratic time, we must also understand how efficiently the model can learn important language modeling skills. We extend the prior analysis in (Arora et al., 2023a) with two lower-bound arguments. First, we show that a minimal gated-convolutional model — i.e. a model that can provably simulate *any* architecture built from gating and standard *input-independent* convolutions — identified as in (Arora et al., 2023a) cannot solve MQAR in constant many layers. Input-independence refers to the convolution being parameterized by model weights, and thus does not change depending on the particular input being processed. (Theorem E.20 and Theorem E.30 in Appendix E).

Theorem E.1. *Given an input sequence* $\boldsymbol{u} \in \{0,1\}^{3N\times}$*, where* N *and denote the sequence length and head dimension, respectively, a data-independent model needs* $\log(2d)$*-layers to solve* MQAR *for* $d = \log_2(c)$*, where* c *denotes the vocabulary size*[4]*.*

Remark E.2. Further, for a class of input encodings that generalizes one-hot encodings termed as p-hot encodings (Definition E.23), input-dependent needs at least $\lfloor \log(2p) \rfloor$-layers to solve MQAR where $d = p \cdot \sqrt[p]{c}$.

In contrast, recall that attention solves MQAR in constant-many layers Arora et al. (2023a). While recent recurrent models Gu & Dao (2023); Yang et al. (2023) use asymptotically smaller state sizes than attention, we next show a lower bound on the state size of any such model that depends causally on the input (Theorem E.15 in Appendix E).

Theorem E.3. *Any recurrent model*[5] *depending causally on input* $\boldsymbol{u} \in \{0,1\}^{N\times d}$ *requires* $\Omega(N)$*-bits*[6] *in state size to solve AR.*

Finally, we show that we can simulate linear attention (Katharopoulos et al., 2020b), the foundation of BASED, using (Arora et al., 2023a) with a poly-log blowup in the number of layers (Proposition E.11 in Appendix E), pointing to the relative efficiency of linear attention over gated-convolution architectures.

E.2 Proofs Setup

Our focus in this section will be on the theoretical results of the paper. Specifically, we will show the equivalence of models and (Gu & Dao, 2023) with , a minimal gated-convolution operator (Arora et al., 2023a, Definition 4.1), and prove lower bounds for the MQAR problem (Arora et al., 2023a, Section H.7.1) in various settings. We begin by setting notation and introducing the theoretical formulations of the models.

Notation. We will be denoting the all 1 row vector of size k, given by $\begin{bmatrix}1 & 1 & \dots & 1 & 1\end{bmatrix}$, and the all 0 row vector of size k, given by $\begin{bmatrix}0 & 0 & \dots & 0 & 0\end{bmatrix}$, as $\mathbf{1}^k$ and $\mathbf{0}^k$, respectively. We will also construe the standard basis vector $\mathbf{e}_i$ as a column vector in these notes, and adhere to the following matrix indexing convention: $\mathbf{M}[i,j]$ is the entry in the ith row and the jth column, $\mathbf{M}[i,:] \in \mathbb{F}^{1\times n}$ denotes the ith row, and $\mathbf{M}[:,j] \in \mathbb{F}^{m\times 1}$ denotes the jth column of $\mathbf{M} \in \mathbb{F}^{m\times n}$, where $\mathbb{F}$ is a field and the reader can substitute $\mathbb{F}$ for $\mathbb{R}$ for convenience. For a matrix $\mathbf{M} \in \mathbb{R}^{n\times m}$, we define the

[4]That is, each token from the vocabulary has the natural binary encoding in $\{0,1\}^{\log_2(c)}$

[5]In particular, for Mamba Gu & Dao (2023), see Corollary E.16.

[6]Here, we need the entries of the state to be bounded.

pair-wise Hadamard product of columns of $\mathbf{M}$ as $\boldsymbol{M} \in \mathbb{R}^{n\times m^2}$, where

$$(\boldsymbol{M})[:,i] := \mathbf{M}[:,j] \odot \mathbf{M}[:,k] \quad \text{for} \quad i \in [m^2],$$
$$j = \left\lfloor \frac{i-1}{m} \right\rfloor + 1, \quad k = (i-1) \mod m + 1. \tag{10}$$

Moreover, we define the element-wise exponentiation of a matrix $\mathbf{M}$ as $\exp[\mathbf{M}]$ where $\exp[\mathbf{M}]_{ij} = \exp(\mathbf{M}_{ij})$. Next, we denote the *Hadamard product* of vectors $\mathbf{u}, \mathbf{v} \in \mathbb{F}^n$ as $\mathbf{u} \odot \mathbf{v}$; the operation can be extended to matrices accordingly, and for vectors $\mathbf{u}, \mathbf{v} \in \mathbb{F}^n$, we denote their *linear (or acyclic) convolution* as $\mathbf{u} * \mathbf{v}$.

Arithmetic Circuit Notation. We briefly introduce the notation of arithmetic circuits (Bürgisser et al., 2013). An *arithmetic circuit* $\mathcal{C}$ with variables $X \triangleq \{x_1, x_2, \ldots, x_n\}$ over a field $\mathbb{F}$ is interpreted as a directed acyclic graph, where the input nodes are labelled by either the variables from X or constants from $\mathbb{F}$ and the internal nodes are labelled by $+$ or $\times$ with the output being the polynomial computed at the output node.

We shall also refer to the *size* of the circuit as the number of nodes, the *depth* of the circuit as the length of the longest path between an input node and the output node, and the *width* of the circuit as the number of parallel operations in the circuit, or 'wires' which will be intersected by a horizontal 'cut' through the circuit. Moreover, the *degree* of a circuit is defined as the degree of the polynomial computed by the circuit. We summarize this with the following definition:

Definition E.4. An arithmetic circuit $\mathcal{C}$ is an (n, s, Δ, w)*-circuit* if $\mathcal{C}$ is an n-variate arithmetic circuit of size s and of depth at most Δ, and width w.

E.3 The Models

We now introduce the definitions of the models and for the reader's convenience. Note that we have redefined these models to ensure consistency with the notation presented above.

E.3.1 Based

The model combines two layer types: and defined below.

Definition E.5 ((Arora et al., 2023a)**.** Given an input sequence $\boldsymbol{u} \in \mathbb{R}^{N\times}$, where N is the sequence length and is the model dimension, a learned weight matrix $\boldsymbol{W}^B \in \mathbb{R}^{\times}$ and biases $\boldsymbol{B}^B, \boldsymbol{B}^K \in \mathbb{R}^{N\times}$ and a matrix of convolution filters $\boldsymbol{K} \in \mathbb{R}^{N\times}$, a layer computes the following:

$$\boldsymbol{z} := (\boldsymbol{u}\boldsymbol{W}^B + \boldsymbol{B}^B) \odot \left(\boldsymbol{K} * \boldsymbol{u} + \boldsymbol{B}^K\right) \in \mathbb{R}^{N\times}, \tag{11}$$

where the convolutions are applied across the input length N.

Definition E.6 ((Katharopoulos et al., 2020a)**.** Given an input sequence $\boldsymbol{u} \in \mathbb{R}^{N\times}$, where N is the sequence length and is the model dimension, a set of linear projections[7] $\texttt{Projection}_q, \texttt{Projection}_k \in \mathbb{R}^{\times}, \texttt{Projection}_v \in \mathbb{R}^{\times}$, where is the feature dimension, the layer computes the following:

$$\boldsymbol{z} := \left(\boldsymbol{Q}\,\boldsymbol{K}^\top\right)\boldsymbol{V} \in \mathbb{R}^{N\times}, \tag{12}$$

where $\boldsymbol{Q} := \texttt{Projection}_q(\boldsymbol{u}), \boldsymbol{K} := \texttt{Projection}_k(\boldsymbol{u}), \boldsymbol{V} := \texttt{Projection}_v(\boldsymbol{u})$, and we have

$$\boldsymbol{Q} = [\mathbf{1}, \boldsymbol{Q}, \boldsymbol{Q}] \in \mathbb{R}^{N\times(1++^2)},$$
$$\boldsymbol{K} = [\mathbf{1}, \boldsymbol{Q}, \boldsymbol{K}] \in \mathbb{R}^{N\times(1++^2)}.$$

E.3.2 Mamba

We now introduce the model from (Gu & Dao, 2023).

Definition E.7 ((Gu & Dao, 2023)**.** Given an input sequence $\boldsymbol{u} \in \mathbb{R}^{N\times}$, where N is the sequence length and is the model dimension, the layer computes the following:

$$\boldsymbol{z} := (\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C})(\boldsymbol{u}) \in \mathbb{R}^{N\times}, \tag{13}$$

[7] By linear projections of a matrix $\boldsymbol{u} \in \mathbb{R}^{m\times n}$, we mean $\boldsymbol{u}\boldsymbol{W} + \boldsymbol{B}$ for some weight matrix $\boldsymbol{W} \in \mathbb{R}^{n\times n}$ and bias $\boldsymbol{B} \in \mathbb{R}^{m\times n}$.

with the parameters, $\overline{\boldsymbol{A}} \in \mathbb{R}^{\times}, \overline{\boldsymbol{B}} \in \mathbb{R}$, defined as

$$\begin{aligned} \overline{\boldsymbol{A}} &:= \exp\left(\Delta \boldsymbol{A}\right), \\ \overline{\boldsymbol{B}} &:= \left(\Delta \boldsymbol{A}\right)^{-1} \left(\exp\left(\Delta \boldsymbol{A}\right) - \boldsymbol{I}\right) \cdot \Delta \boldsymbol{B}, \\ &= \boldsymbol{A}^{-1} (\exp\left(\Delta \boldsymbol{A}\right) - \boldsymbol{I}) \cdot \boldsymbol{B}, \end{aligned} \tag{14}$$

where , the state dimension, and $\boldsymbol{A} \in \mathbb{R}^{\times}$ are parameters of the model and do not depend on the input $\boldsymbol{u}$, along with the following *input-dependent* parameters $\boldsymbol{B}, \boldsymbol{C} \in \mathbb{R}^{N\times}, \Delta \in \mathbb{R}^{N\times}$ defined as

$$\begin{aligned} \boldsymbol{B} &:= \texttt{Linear}_{d,d'}\left(N\times\right)(\boldsymbol{u}) \in \mathbb{R}, \\ \boldsymbol{C} &:= \texttt{Linear}_{d,d'}\left(N\times\right)(\boldsymbol{u}) \in \mathbb{R}, \\ \Delta &:= \texttt{Linear}_{d,d'}\left(N\times\right)(\boldsymbol{u}) \in \mathbb{R} \end{aligned} \tag{15}$$

for $i \in [N]$. It is important to note here that the parameters $\overline{\boldsymbol{B}}, \boldsymbol{C}, \Delta$ are causal[8] and we denote the dependence on upto the ith row of the input $\boldsymbol{u}$ for $i \in [N]$ by adding a subscript i where the dependence for $\overline{\boldsymbol{A}}_i \in \mathbb{R}^{\times}$ is inherited from Δ_i in equation 14 and we denote $\overline{\boldsymbol{B}}[i,:] =: \boldsymbol{B}_i, \overline{\boldsymbol{C}}[i,:] =: \boldsymbol{C}_i$.

Finally, the in equation 13 is realized as a linear recurrence. That is, for every $(i,j) \in [N] \times []$, we have

$$\begin{aligned} \boldsymbol{h}[i,j] &= \overline{\boldsymbol{A}}_i \boldsymbol{h}[i-1,j] + \overline{\boldsymbol{B}}_i \boldsymbol{u}[i,j] \\ \boldsymbol{z}[i,j] &= \boldsymbol{C}_i^{\top} \boldsymbol{h}[i,j] \end{aligned} \tag{16}$$

where $\boldsymbol{h}[i,j] \in \mathbb{R}, \boldsymbol{z}[i,j] \in \mathbb{R}$ denote the latent state and the output of the in Eq. (13), respectively.

E.4 Equivalency to

For a polynomial with variables X over a field $\mathbb{F}$, there exists a corresponding arithmetic circuit $\mathcal{C}$ over X that computes the output of the polynomial at its terminating node when interpreted as a directed acyclic graph. For any such arithmetic circuit $\mathcal{C}$ of size s and depth Δ, (Arora et al., 2023a, Theorem 4.2) showed the existence of an equivalent operator that uses $\tilde{\mathcal{O}}(s\Delta)$ parameters and $\tilde{\mathcal{O}}(\Delta)$ layers. In the sequel, we use this result by expressing the model outputs computed in equation 12 and equation 13 as polynomials in $\boldsymbol{u}$ and $\exp(\boldsymbol{u})$ to show the equivalency between these disparate models. We would now like to recall (Arora et al., 2023a, Theorem 4.2). Before doing so, we first establish the following definitions from (Arora et al., 2023a).

Definition E.8. An $(N, , d, N', d')$ − Gated Convolution Model is a stacked sequence to sequence model with L layers such that:

1. input and output are $N \times d$ matrices,
2. each layer's operations consist of element-wise gating, convolution, linear projection, and
3. all the individual gated convolution layers take in $N' \times d'$ matrices and output $N' \times d'$ matrices. We refer to the tuple (N', d') as the *inner dimension* of the model.

We also assume that the input $\in \mathbb{R}^{N\times d}$ is embedded into $' \in \mathbb{R}^{N'\times d'}$ such that

$$'[n,t] = \begin{cases} [n,t] & \text{if } n < N,\ t < d \\ 0 & \text{otherwise.} \end{cases}$$

The output from the last layer $\boldsymbol{z} \in \mathbb{R}^{N'\times d'}$ is transformed into output $\boldsymbol{y} \in R^{N\times d}$ by extracting the top left $N \times d$ entries in $\boldsymbol{z}$.

Theorem E.9 (Arora et al. (2023a), Theorem 4.2)**.** *For any (nd, s, Δ, w)-arithmetic circuit $\mathcal{C}$, there exists an equivalent (N, Δ', d, N', d') − BaseConv with $N = n, \Delta' = \mathcal{O}(\Delta \log w)$, $N' = \mathcal{O}(w), d' = d$ that simulates $\mathcal{C}$.*

Remark E.10. For notational simplicity, we will use $\boldsymbol{u}_{i,j}$ as the symbol for the variable in the polynomial in $\boldsymbol{u}$ representing the entry $\boldsymbol{u}[i,j]$.

[8]That is, $\boldsymbol{B}[i,:], C[i,:]$ and $\Delta[i,:]$ depend only on $\boldsymbol{u}[0 \cdots i-1]$.

We now present the results showing equivalency between the models in Appendix E.3 and the layer in equation 11 using Theorem E.9.

Proposition E.11. *Given an input* $\boldsymbol{u} \in \mathbb{R}^{N\times}$, *there exists an equivalent* $\left(N, O(\log^2(Nd)), , O(N(+^2), O(\max(d,^2)))\right) - \textsc{BaseConv}$ *that computes the output of the layer with feature dimension , cf. Eq.* (12).

Proof. For the matrices $\boldsymbol{Q}, \boldsymbol{K} \in \mathbb{R}^{N\times}, \boldsymbol{V} \in \mathbb{R}^{N\times}$ with the corresponding projection matrices $\boldsymbol{W}^Q, \boldsymbol{W}^k \in \mathbb{R}^{\times}, \boldsymbol{W}^V \in \mathbb{R}^{\times}$, a single layer that computes each of these matrices by simply taking identical projection and $\boldsymbol{h}^s, \boldsymbol{h}^l, \boldsymbol{B}^s \equiv 0$ and $\boldsymbol{B}^\ell \equiv^{N\times}$, the all 1 matrix. Using the remembering primitive (Arora et al., 2023a, Proposition H.10), we can compute each of these in turn while remembering others using $O(1)$ layers and Nd parameters.

Next, we derive an expression for each entry $(i, j) \in [N] \times [^2]$ of $\boldsymbol{Q}, \boldsymbol{K} \in \mathbb{R}^{N\times^2}$. From equation 10, observe that each entry of $\boldsymbol{M}$ can be written as the product of entries from $\boldsymbol{M}$. Hence we have

$$\begin{aligned}(\boldsymbol{Q})[i,j] &\equiv \boldsymbol{Q}[i,k] \cdot \boldsymbol{Q}[i,\ell] \\ (\boldsymbol{K})[i,j] &\equiv \boldsymbol{K}[i,k] \cdot \boldsymbol{K}[i,\ell]\end{aligned} \tag{17}$$

for $k = \left\lfloor \frac{j-1}{} \right\rfloor + 1, \quad \ell = (j-1) \mod + 1$. Note, however, that we can simulate the above by first increasing the inner dimension and copying over columns of $\boldsymbol{Q}$ to get $\boldsymbol{Q}_1, \boldsymbol{Q}_2 \in \mathbb{R}^{N\times d}$ defined as $\boldsymbol{Q}_1[i,j] := \boldsymbol{Q}[i,k]$ and $\boldsymbol{Q}_2[i,j] := \boldsymbol{Q}[i,\ell]$ for $k = \left\lfloor \frac{j-1}{} \right\rfloor + 1, \quad \ell = (j-1) \mod + 1$ so that $(\boldsymbol{Q}) = \boldsymbol{Q}_1 \odot \boldsymbol{Q}_2$, which, *mutatis mutandis*, also applies to $(\boldsymbol{K})$ We can achieve the copying of the columns by simply using the projection matrix $\boldsymbol{W}^B$ and another permutation matrix $\boldsymbol{P}$. Apart from the multiplication by $\boldsymbol{P}$, we only need to use $O(1)$ layers, and moreover, since the circuit that computes $\boldsymbol{P}\boldsymbol{u}$ simply rearranges the input, there exists a single layer that computes $\boldsymbol{P}\boldsymbol{u}$ (Arora et al., 2023a, Corollary H.20). By the stacking lemma (Arora et al., 2023a, Lemma H.11), we can stack these layers to get a composition of the outputs so far to get a $\left(N, O(1), , O(N(+^2), O(\max(d,^2)))\right) - \textsc{BaseConv}$ model. Moreover, the concatenated matrices $\overline{\boldsymbol{Q}}, \overline{\boldsymbol{K}} \in \mathbb{R}^{N\times(1++^2)}$ then take the addition of the computed components so far which again takes $O(1)$ layers of .

Finally, we can express each entry $(i, j) \in [N] \times []$ of the output of as a polynomial as follows:

$$\boldsymbol{z}_{i,j}(\boldsymbol{u}) \equiv \sum_{m\in[1++^2], n\in[N]} \overline{\boldsymbol{Q}}[i,m] \cdot \overline{\boldsymbol{K}}[n,m] \cdot \boldsymbol{V}[n,j]. \tag{18}$$

Thus, we can derive the arithmetic circuit that computes $\boldsymbol{z}_{i,j}(\boldsymbol{u})$ by taking in the outputs of the layers so far as input and compute each of the terms inside the sum by multiplying the outputs from all three and compute the sum using additional $\log\lceil N\rceil$ depth. Each term inside the sum requires two multiplication gates with depth 2, each of which serve as inputs to the circuit with size N computing the sum. Moreover, there are $N\cdot$ such output gates each of which is computed in parallel resulting in a circuit of size $O(N\cdot)$, depth $O(\log(N))$ and width $O(N)$. O Overall, applying Theorem E.9 then results in an equivalent $\left(N, O(\log^2(Nd)), , O(N(+^2), O(\max(d,^2)))\right) - \textsc{BaseConv}$ model that computes $\boldsymbol{z}$. ☐

E.5 The Lower Bounds

In the sequel, we consider the *multiple-query associative recall* problem (MQAR) as defined in (Arora et al., 2023a, Section H.7.1). We briefly recall the definition here.

> Suppose we are given an input sequence $\boldsymbol{u}[0 \cdots 3N-1] \triangleq \{(\boldsymbol{k}_0, \boldsymbol{v}_0, \boldsymbol{q}_0), \ldots, (\boldsymbol{k}_{-1}, \boldsymbol{v}_{-1}, \boldsymbol{q}_{-1})\}$ with each $\boldsymbol{k}_i, \boldsymbol{v}_i, \boldsymbol{q}_i \in C$ is a token drawn from a vocabulary of size $c = |C|$. Our goal is then to check, for each $1 \le i \le -1$, whether there exists $0 \le j < i$ such that $\boldsymbol{q}_i \equiv \boldsymbol{k}_j$, and if so, output $\boldsymbol{v}_j$.

E.6 The Space Complexity of AR

We will start by providing a lower bound on the space complexity of solving the standard associative recall (AR) problem. As AR is a subclass of MQAR, this naturally provides a lower bound on the space complexity of MQAR as well. Here, we formally recall the associative recall problem.

> The AR problem takes key-value pairs $\{\boldsymbol{k}_i, \boldsymbol{v}_i\}_{i=0}^{n-1}$ along with a query $\boldsymbol{q}$ appended at the end as input and the goal is to output $\boldsymbol{v}_i$ if $\boldsymbol{q} = \boldsymbol{k}_i$ for some $i \in [0, N-1]$.

We now require a randomized communication complexity lower bound result for the *index problem*:

> The index problem has two agents, Alice and Bob, where Alice has a string $\boldsymbol{x} \in \{0,1\}^n$ and Bob has an index $i \in [n]$, and the goal for the players is to output the i-th entry $\boldsymbol{x}_i$. Moreover, we also require the communication to be *one-way*: only Alice is allowed to send a single message to Bob and Bob needs to output the answer.

We will make use of the following lower-bound result.

Theorem E.12 (Jayram et al. (2008))**.** *The one-way randomized communication complexity*[9] *of the index problem for sending an n-length bit string is $\Omega(n)$.*

We now use Theorem E.12 to provide a lower bound on the number of bits required by the following class of models to solve AR.

Definition E.13 (Recurrent Models)**.** A model taking an input $\boldsymbol{u} \in \mathbb{R}^{N\times}$, where N is the input length and d is the model dimension, is termed a *recurrent model* if its i-th state, representing the output at location i, $\boldsymbol{Z}^i \in \mathbb{R}^{d'}$, with d' denoting the state size, is determined exclusively by the preceding elements of the input $\boldsymbol{u}[0 \ldots i-1]$. The state $\boldsymbol{Z}^i_{\mathcal{M}}$ represents the accumulated information of the model depending on the inputs up to the i-th element, and is distinct from learned parameters that are static with respect to the input sequence.

Specifically, $\boldsymbol{Z}^i(\boldsymbol{u}) = \phi(\boldsymbol{u}[0 \ldots i-1])$, indicating that the state is a function of the input history but not of the entire input sequence simultaneously. Moreover, we can express this as:

$$\boldsymbol{Z}^i(\boldsymbol{u}) = f^i(\boldsymbol{Z}^{i-1}, \boldsymbol{u}[i]), \tag{19}$$

for a sequence of functions $\{f^i\}_{i\in[N]}$, where each function is tailored to evolve the state based on the immediate past state and the current input.

Remark E.14. Note that Definition E.13 excludes models that inherently require the entire input sequence for computation at any state, such as those based on non-causal convolutional operations over the full input.

Theorem E.15. *Any recurrent model (Definition E.13) that solves AR requires $\max_i \left|\boldsymbol{Z}^i\right|$ to be at least$\Omega(N)$-bits.*

Proof. Consider an instance $(\boldsymbol{x}, i)$ of the index problem with $\boldsymbol{x} \in \{0,1\}^N$. We now describe the corresponding instance of the AR problem:

$$\{j, \boldsymbol{x}_j\}_{j=0}^{N-1}, i. \tag{20}$$

Next, consider the following one-way protocol for solving the index problem using the regressive model . Alice with their access of $\boldsymbol{x} \in \{0,1\}^N$ generate an input for AR (without the query) as in equation 20. Alice then runs the model on $\{i, \boldsymbol{x}_j\}_{j=0}^{N-1}$ and sends the memory content of running the model to Bob. This should include the state $\boldsymbol{Z}^{N-1}$ of size d' as we can reasonably assume that both have access to the set of functions $\{f^j\}_{j\in[N]}$. Since we assume that this model solves AR, the output $\texttt{Out}[N,:] = \boldsymbol{x}_i$ should contain the associated value of i. Here, Bob can compute $\texttt{Out}[N,:]$ by using the memory content sent by Alice and applying the function f^N as follows.

$$\boldsymbol{x}_i = \texttt{Out}[N,:] = f^N(\boldsymbol{Z}^{N-1}, \boldsymbol{u}[N]).$$

That is, the total number of bits that are communicated in this protocol is $\left|\boldsymbol{Z}^{N-1}\right|$. Now, if $\max_j \left|\boldsymbol{Z}^j\right|$ is $o(N)$ bits, we have shown that a one-way communication protocol exists for solving the index problem exists that uses $o(N)$ communication complexity. This contradicts Theorem E.12 and hence, we conclude that the model solving AR also needs $\Omega(N)$ bits. ☐

[9] The randomized communication complexity of function f is defined as $\min_\pi \|\pi\|$, where π ranges over all randomized protocols that can solve f with probability of success at least $2/3$.

Corollary E.16. *Given an input $\boldsymbol{u} \in \mathbb{R}^{N\times}$ to the AR problem, a causal model with all entries in its computation taking $O(1)$ bits needs $+ \geq \Omega(N)$ bits to solve AR.*

Proof. We will first show that causal is a recurrent model. To see this, first observe equation 16 and note the fact that the input-dependent parameters $\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C}, \Delta$ are causal as mentioned in Definition E.7.

Next, due to equation 16, in order to compute $\boldsymbol{z}_{N,:} \in \mathbb{R}$, we need $\boldsymbol{C}_N \in \mathbb{R}, \overline{\boldsymbol{B}}_N \in \mathbb{R}$ and $\Delta_N \in \mathbb{R}$ along with $\boldsymbol{h}[N-1,:] \in \mathbb{R}$. Here, we have the $(N-1)$-st state $\boldsymbol{Z}^{N-1} \in \mathbb{R}^{3+}$ given by

$$\boldsymbol{Z}^{N-1} := \{\boldsymbol{h}[i-1,:], \Delta_N^1, \overline{\boldsymbol{B}}_N^1, \boldsymbol{C}_N^1\},$$

where $\Delta_N^1, \overline{\boldsymbol{B}}_N^1, \boldsymbol{C}_N^1$ are all linear functions of $\boldsymbol{u}[0\cdots N-1]$ that we receive from the $(N-1)$-st state and we compute $\Delta_N^2, \overline{\boldsymbol{B}}_N^2, \boldsymbol{C}_N^2$ as linear functions of $\boldsymbol{u}[N]$ so that we have $\Delta_N = \Delta_N^1 + \Delta_N^1, \overline{\boldsymbol{B}}_N = \overline{\boldsymbol{B}}_N^1 + \overline{\boldsymbol{B}}_N^2, \boldsymbol{C}_N = \boldsymbol{C}_N^1 + \boldsymbol{C}_N^2$. We can then define the function f^N as follows:

$$\begin{aligned}
\boldsymbol{Z}^N[j] &= \exp(\Delta_N[j]\boldsymbol{A})\boldsymbol{h}[N-1,j] + \overline{\boldsymbol{B}}_N\boldsymbol{u}[N,j] \\
&= \overline{\boldsymbol{A}}_N\boldsymbol{h}[N-1,j] + \overline{\boldsymbol{B}}_N\boldsymbol{u}[N,j], \\
\texttt{Out}[N,j] &= f^N(\boldsymbol{Z}^{N-1})[j] = \boldsymbol{C}_N^\top\boldsymbol{Z}^N[j].
\end{aligned}$$

Thus, due to Theorem E.15, we can conclude that $\left|\boldsymbol{Z}^{N-1}\right|$ does require $\Omega(N)$-bits to solve AR. Finally, assuming each entry of $\boldsymbol{Z}^{N-1}$ needs $O(1)$ bits to represent, the overall state $\boldsymbol{Z}^{N-1}$ needs $O(+)$ to represent, which completes the proof of the claim. ☐

E.6.1 Lower Bound on the Number of Layers for MQAR with $d = \log_2 c$

Setup. We take $d = \log_2 c$ to encode all c possible tokens from C. That is, all the 2^d possible d-bit vectors can appear as a token in the input for MQAR. We will show that data-independent needs $\Omega(\log d) = \Omega(\log\log c)$-layers to solve this setting of MQAR, while Attention (+ReLU) can solve this in $O(1)$ layers.

We first provide the trivial solution using Attention (+ReLU).

Proposition E.17. *Attention (with linear biases and ReLU) followed by two layers of MLPs can solve MQAR for an input sequence $\boldsymbol{u} \in \{0,1\}^{3N\times d}$ such that $d = \log_2(c)$ in $O(1)$ layers.*

Proof. Given a row $\boldsymbol{u}[i,:] \in \{0,1\}^d$, we express each row as $\boldsymbol{w}[i,:] \in \{-1,1\}^d$ by applying the projection $\boldsymbol{u}\boldsymbol{W} + \boldsymbol{B}$, where $\boldsymbol{W} := \mathrm{diag}(2,\ldots,2) \in \mathbb{R}^{d\times d}$ and the bias matrix $\boldsymbol{B}$ is the matrix of all -1's so that $\boldsymbol{w}[i,j] = 2\boldsymbol{u}[i,j] - 1$. Then, we can specify the query and key projection matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{3N\times d}$ as follows:

$$\begin{aligned}
\mathbf{K}[i,:] &\equiv \begin{cases} \boldsymbol{w}[i,:] = \boldsymbol{k}_{\lfloor i/3\rfloor} & \text{if } i \equiv 0 \mod 3 \\ \mathbf{0} & \text{otherwise} \end{cases} \\
\mathbf{Q}[i,:] &\equiv \begin{cases} \boldsymbol{w}[i,:] = \boldsymbol{q}_{\lfloor i/3\rfloor} & \text{if } i \equiv 2 \mod 3 \\ \mathbf{0} & \text{otherwise} \end{cases} \quad, \\
\mathbf{V}[i,:] &\equiv \begin{cases} \boldsymbol{w}[i+1,:] = \boldsymbol{v}_{\lfloor i/3\rfloor} & \text{if } i \equiv 0 \mod 3 \\ \mathbf{0} & \text{otherwise} \end{cases}
\end{aligned}$$

where the values are shifted to the corresponding key index. Computing the pair-wise inner products then yields

$$\mathbf{Q}\mathbf{K}^\top[i,j] \equiv \begin{cases} \langle \boldsymbol{q}_{\lfloor i/3\rfloor}, \boldsymbol{k}_{\lfloor j/3\rfloor}\rangle & \text{if } i \equiv 2 \mod 3 \text{ and } j \equiv 0 \mod 3 \\ \mathbf{0} & \text{otherwise} \end{cases}$$

However, since both $\boldsymbol{q}_{\lfloor i/3\rfloor}, \boldsymbol{k}_{\lfloor j/3\rfloor} \in \{-1,1\}^d$, we have $\langle \boldsymbol{q}_{\lfloor i/3\rfloor}, \boldsymbol{k}_{\lfloor j/3\rfloor}\rangle \leq d$ with equality iff $\boldsymbol{q}_{\lfloor i/3\rfloor} \equiv \boldsymbol{k}_{\lfloor j/3\rfloor}$. We then subtract off $d-1$ from each of the $3N \times 3N$ entries by taking the bias $\mathbf{B} \in \mathbb{R}^{3N\times 3N}$ as the matrix with each entry $-d+1$. Let $\mathbf{Z} := \mathrm{ReLU}(\mathbf{Q}\mathbf{K}^\top + \mathbf{B})$ so that we have

$$\mathbf{Z}[i,j] = \{\boldsymbol{q}_{\lfloor i/3\rfloor} \equiv \boldsymbol{k}_{\lfloor j/3\rfloor}\}.$$

Next, as we may have multiple matches and we only need to return 1, we modify $\boldsymbol{Z}$ by multiplying with the matrices $\boldsymbol{W}_1, \boldsymbol{W}_2 \in \mathbb{R}^{d\times d}$ and adding the bias $\boldsymbol{B} \in \mathbb{R}^{d\times d}$ defined as follows:

$$\boldsymbol{W}_1[k,j] := \begin{cases} 1 & \text{if } k \geq j \\ 0 & \text{otherwise} \end{cases}, \quad \boldsymbol{W}_2[\ell,k] := \begin{cases} -1 & \text{if } k = 0 \\ 1 & \text{if } k = \ell, \ell \neq 0 \\ 0 & \text{otherwise} \end{cases}, \quad \boldsymbol{B}[i,j] = 1.$$

For $\boldsymbol{Z}_1 := \boldsymbol{Z}\boldsymbol{W}_1$ and $\boldsymbol{Z}_2 := \boldsymbol{Z}\boldsymbol{W}_1\boldsymbol{W}_2$, we have:

$$\boldsymbol{Z}_1[i,j] = \sum_k \boldsymbol{Z}[i,k]\boldsymbol{W}_1[k,j] = \sum_{k\geq j} \boldsymbol{Z}[i,k],$$
$$\boldsymbol{Z}_2[i,j] = \sum_k \boldsymbol{Z}_1[i,k]\boldsymbol{W}_2[k,j] = \boldsymbol{Z}_1[i,j] - \boldsymbol{Z}_1[i,0].$$

That is, each entry in $\boldsymbol{Z}_1$ sums the entries in the row that are at the same or higher column index while each column in $\boldsymbol{Z}_2$ subtracts the first entry—the sum of all entries in the row—from each entry in the row. Semantically, for each row in $\boldsymbol{Z}_1$, the entries from 0 to the index of the first match must have the same value, and thus, are the only non-negative entries in $\boldsymbol{Z}_2$. Next, we add the bias and activate under ReLU to get $\boldsymbol{Z}' \in \mathbb{R}^{3N\times d}$:

$$\boldsymbol{Z}'[i,k] := \mathrm{ReLU}(\boldsymbol{Z}_2 + \boldsymbol{B})[i,k] = \begin{cases} 1 & \text{if } k \leq \min\{j|\ \boldsymbol{q}_{\lfloor i/3\rfloor} \equiv \boldsymbol{k}_{\lfloor j/3\rfloor}\} \\ 0 & \text{otherwise.} \end{cases}$$

Now, we multiply by the weight matrix $\boldsymbol{W}_3 \in \mathbb{R}^{3N\times d}$ defined as

$$\boldsymbol{W}_3[k,j] := \begin{cases} -1 & \text{if } k = j+1 \\ 1 & \text{if } k = j \\ 0 & \text{otherwise} \end{cases}$$

This yields the retriever $\overline{\boldsymbol{Z}} = \boldsymbol{Z}'\boldsymbol{W}_3 \in \mathbb{R}^{3N\times d}$ given by

$$\overline{\boldsymbol{Z}}[i,k] := \sum_\ell \boldsymbol{Z}'[i,\ell]\boldsymbol{W}_3[\ell,k] = \boldsymbol{Z}'[i,k] - \boldsymbol{Z}'[i,k+1] = \{k = \min\{j|\ \boldsymbol{q}_{\lfloor i/3\rfloor} \equiv \boldsymbol{k}_{\lfloor j/3\rfloor}\}\}.$$

Finally, we multiply with the values $\mathbf{V}$ to get

$$(\overline{\boldsymbol{Z}}\mathbf{V})[i,:] \equiv \overline{\boldsymbol{Z}}[i,:]\mathbf{V} \equiv \overline{\boldsymbol{Z}}[i,j^*]\cdot\mathbf{V}[j^*,:] \equiv \begin{cases} \boldsymbol{v}_{j^*} & \text{if } \boldsymbol{q}_{\lfloor i/3\rfloor} \equiv \boldsymbol{k}_{\lfloor j^*/3\rfloor}, j^* = \min\{j|\ \boldsymbol{q}_{\lfloor i/3\rfloor} \equiv \boldsymbol{k}_{\lfloor j/3\rfloor}\}. \\ \mathbf{0} & \text{if no such } j^* \text{ exists.} \end{cases}$$

That is, the row corresponding to the query returns the value associated to the first matching key. Thus, the model with Attention (computing $\boldsymbol{Z}$) followed by two MLPs computing $\boldsymbol{Z}'$ and $\overline{\boldsymbol{Z}}$, respectively, solves the MQAR problem. ☐

Next, we relate the output of L layers of to the degree of the polynomial that it computes.

Lemma E.18. *For any input sequence $\boldsymbol{u}$, there exists a multilinear polynomial equivalent (over Boolean inputs) to the polynomial computed by L layers of with degree at most 2^L.*

Proof. Let $P(\boldsymbol{u})$ be the polynomial computed by L layers of . Since the output of a single layer of is equivalent to a polynomial over the input variables with degree at most 2, composing L such layers yields a polynomial of degree at most 2^L. However, $P(\boldsymbol{u})$ need not be multi linear, but the polynomial defined as

$$Q(\boldsymbol{u}) := (\cdots((P(\boldsymbol{u}) \mod (u_1^2 - u_1)) \mod (u_2^2 - u_2))\cdots) \mod (u_{3Nd}^2 - u_{3Nd})$$

is equivalent to $P(\boldsymbol{u})$ as $(u_i^2 - u_i)$ evaluates to 0 for each input var $u_i \in \{0,1\}$. However, $\deg(Q(\boldsymbol{u})) \leq \deg(P(\boldsymbol{u}))$, and thus, the claim holds. ☐

We now relate the MQAR (in the above setting) to the degree of the polynomial that it computes.

Lemma E.19. *The MQAR problem with $d = \log_2(c)$ is represented by a multi-linear polynomial of degree $2d+1$.*

Proof. We will start by specifying the obvious Boolean circuit that solves MQAR. First, we take the XNOR of keys and queries bitwise as follows.

$$\boldsymbol{x}^{ij} = \boldsymbol{q}_i \texttt{ xnor } \boldsymbol{k}_j := (\boldsymbol{q}_i \wedge \boldsymbol{k}_j) \vee (\neg \boldsymbol{q}_i \wedge \neg \boldsymbol{k}_j) \text{ for } i > j, \tag{21}$$

where, for $\boldsymbol{x}, \boldsymbol{y} \in \{0,1\}^d$, we have

$$[\boldsymbol{x} \texttt{ xnor } \boldsymbol{y}][k] := \begin{cases} 1 & \text{if } \boldsymbol{x}[k] = \boldsymbol{y}[k] \\ 0 & \text{othwerise} \end{cases}$$

That is, each bit from $\boldsymbol{x}^{ij}$ is set to 1 iff the corresponding bits from $\boldsymbol{q}_i$ and $\boldsymbol{k}_j$ match. Next, we take the AND of the d-bits to get

$$\boldsymbol{y}^{ij} := \bigwedge_{k \in [d]} \boldsymbol{x}_k^{ij}, i > j. \tag{22}$$

Thus, $\boldsymbol{y}^{ij}$ is set to 1 iff the query $\boldsymbol{q}_i$ matches with the key $\boldsymbol{k}_j$. Finally, we AND with each bit of the values to get the output $\boldsymbol{z}^{ij}$ with the kth bit for $k \in [d]$ given by

$$\boldsymbol{z}_k^{ij} := \boldsymbol{y}_{ij} \wedge [\boldsymbol{v}_j]_k. \tag{23}$$

Thus, the output of the circuit can be represented as

$$\boldsymbol{z}^{ij} = \begin{cases} \boldsymbol{v}_i & \text{if } \boldsymbol{q}_i \equiv \boldsymbol{k}_j, i > j \\ \boldsymbol{0} & \text{otherwise.} \end{cases}$$

We can now directly translate the above circuit into a multi-linear polynomial. With slight abuse of notation, we have the following correspondence for equation 22, where $\boldsymbol{u}_i \equiv \boldsymbol{q}_i, \boldsymbol{u}_j \equiv \boldsymbol{k}_j, i > j$ and we use $\boldsymbol{u}_{ij}$ to represent the variable corresponding to the entry $\boldsymbol{u}[i,j]$.

$$\boldsymbol{x}_k^{ij}(\boldsymbol{u}) := \boldsymbol{u}_{ik}\boldsymbol{u}_{jk} + (1 - \boldsymbol{u}_{ik})(1 - \boldsymbol{u}_{jk}) \quad \text{for each } k \in [d], i > j.$$

Next, we translate equation 22 as follows.

$$\boldsymbol{y}^{ij}(\boldsymbol{u}) := \prod_{k \in [d]} \left(\boldsymbol{u}_{ik}\boldsymbol{u}_{jk} + (1 - \boldsymbol{u}_{ik})(1 - \boldsymbol{u}_{jk})\right).$$

Finally, we can write the polynomial that computes MQAR as follows.

$$\boldsymbol{z}^{ij}(\boldsymbol{u}) := \left(\prod_{k \in [d]} \boldsymbol{u}_{ik}\boldsymbol{u}_{jk} + (1 - \boldsymbol{u}_{ik})(1 - \boldsymbol{u}_{jk})\right) \boldsymbol{u}_{(i+1)k} \quad \text{for each } k \in [d], i > j, \tag{24}$$

where $\boldsymbol{u}[i+1,:] \equiv \boldsymbol{v}_j$. It is then easy to observe that equation 24 is multi-linear and has degree $2d + 1$. □

We are now ready to provide the lower bound.

Theorem E.20. *A data-independent model needs* $\log(2d)$*-layers to solve* MQAR *for an input sequence* $\boldsymbol{u} \in \{0,1\}^{3N \times d}$ *with* $d = \log_2(c)$.

Proof. Due to Lemma E.19, we know there exists a multi-linear polynomial that solves MQAR, and due to (Kopparty, 2020, Lecture 3, Proposition 4), it is unique. Specifically we cannot solve MQAR with a multi-linear polynomial of degree $\leq 2d$. Now, assume that there is a model with L layers that exactly solves MQAR. Then, due to Lemma E.18, this yields a multilinear polynomial $P(\boldsymbol{u})$ of degree at most 2^L. Here, if $L \leq \log(2d)$, then the resulting with L layers results in a multilinear polynomial of degree $\leq 2d$. This contradicts the above claim that we cannot have a multi linear polynomial of degree $< 2d + 1$ that exactly represents MQAR. Consequently, a data-independent model needs $\geq \log(2d)$-layers to solve MQAR. □

E.7 LOWER BOUND ON THE NUMBER OF LAYERS FOR $d \geq \log_2 c$ WITH SPECIFIC ENCODINGS

E.7.1 THE EQUALITY PROBLEM

For an input pair $\boldsymbol{u}_1, \boldsymbol{u}_2$ where each $\boldsymbol{u}_i$ is a token drawn from a vocabulary of size $c = |C|$ and embedded in $\{0,1\}^d$, we define the *equality problem* (EQ) as checking whether the two encodings are equal: $\boldsymbol{u}_1 \equiv \boldsymbol{u}_2$.

We first note that any model that solves MQAR also solves EQ via the following proposition.

Proposition E.21. *Any model M_{MQAR} that solves MQAR also solves EQ using the same number of layers.*

Proof. If there exists a model M_{MQAR} that solves MQAR using L layers, then for an arbitrary input instance for EQ given by $\boldsymbol{u}_1, \boldsymbol{u}_2 \in \mathbb{R}^{2\times d}$, we can produce the following input instance for MQAR: $\boldsymbol{u} := \{(\boldsymbol{u}_1, , \boldsymbol{u}_1), (\boldsymbol{u}_2, , \boldsymbol{u}_2)\}$ and solve EQ using L layers with M_{MQAR} returning iff there is a match. ☐

Due to Proposition E.21, we obtain the following corollary.

Corollary E.22. *Any lower bound $\overline{L}$ on the number of layers L of to solving EQ is also a lower bound on the number of layers required for solving* MQAR.

We now try to prove a lower bound for the case of $d \geq \log_2 c$. First, note that there are embeddings here where the lower bound from E.20 holds: consider the embedding where the first $\log_2 c$ has the compact binary embedding as before but the last $d - \log_2 c$ bits are the same for all the tokens. We will instead prove a lower bound for a more interesting set of embeddings.

E.7.2 THE p-HOT ENCODING FOR $p \geq 1$

Definition E.23 ((Almost) p-Hot Encoding)**.** We define the *p-hot encoding* to be the collection of embeddings for a token $\boldsymbol{x}_t$ with $0 \leq t < c$ such that we express t in base $\sqrt[p]{c} : (t_0, .., t_{p-1}) \in [0, \sqrt[p]{c})^p$ and represent each t_i as one hot encoding in $\{0,1\}^{\sqrt[p]{c}}$. That is, we take $d = p \cdot \sqrt[p]{c}$.

Moreover, we define the *almost p-hot encoding* to be the collection of embeddings where each t_i is mapped in $\{0,1\}^{\sqrt[p]{c}-1}$ obtained by dropping the last bit of its one-hot encoding in $\{0,1\}^{\sqrt[p]{c}}$.

Note that both of the encodings have p-many blocks derived from each of the one-hot encodings.

Definition E.24 (Block-Exclusive)**.** We say that a polynomial P with variables in $\boldsymbol{u} := (\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{p-1})$ is *block-exclusive* if each non-zero monomial in P given by the product

$$\prod_{i\in[p],\ j\in[\sqrt[p]{c}]} \boldsymbol{u}_{i,j}$$

does not contain any product of the form $\boldsymbol{u}_{i,j}\boldsymbol{u}_{i,j'}$ for $i \in [p], j, j' \in [\sqrt[p]{c}]$.

Remark E.25. The condition specified in Definition E.24 ensures that a block-exclusive polynomial is necessarily multilinear, as it disallows the term $\boldsymbol{u}_{i,j}\boldsymbol{u}_{i,j'}$ for $j = j'$ in any non-zero monomial.

Lemma E.26. *For any Boolean function $f : \{0,1\} \to \{0,1\}$ with inputs from the almost p-hot encoding* or *the p-hot encoding setting, there exists a block-exclusive polynomial equivalent to f.*

Proof. Given an input $\boldsymbol{u}$ to f from the almost p-hot encoding or the p-hot encoding such that $\boldsymbol{u} := (\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{p-1})$, we first observe that the polynomial $P(\boldsymbol{u})$ representing $f(\boldsymbol{u})$ cannot have a non-zero monomial with variables from the same block. Specifically, for $0 \leq j < p$, any non-zero monomial in P cannot have a product of the form $\boldsymbol{u}_{j,k}\boldsymbol{u}_{j,k'}$ for $k \neq k'$. To see this, assume that there exists a non-zero monomial in P with at least two terms $\boldsymbol{u}_{j,k}\boldsymbol{u}_{j,k'}$ from the same jth block in $\boldsymbol{u}$, then monomial always evaluates to 0 as the jth block is derived from the one-hot encoding in $\{0,1\}^{\sqrt[p]{c}}$ or the almost one-hot encoding in $\{0,1\}^{\sqrt[p]{c}-1}$, and hence, cannot have more than one bit set to 1.

Next, if a non-zero monomial in P does contain a product of the form $\boldsymbol{u}_{j,k}\boldsymbol{u}_{j,k'}$ for $k, k' \in [\sqrt[p]{c}]$, we can define the polynomial

$$Q(\boldsymbol{u}) := (\cdots((P(\boldsymbol{u}) \mod (u_{0,0}^2 - u_{0,0})) \mod (u_{0,1}^2 - u_{0,1}))\cdots) \mod (u_{p-1,\sqrt[p]{c}-1}^2 - u_{p-1,\sqrt[p]{c}-1}).$$

Since each entry is Boolean, Q is equivalent to P over Boolean inputs, and thus, Q is the block-exclusive polynomial equivalent to f. $\square$

Proposition E.27. *Any Boolean function $f : \{0,1\} \to \{0,1\}$ with inputs from the almost p-hot encoding setting has a unique representation as a block-exclusive polynomial.*

Proof. Due to (Kopparty, 2020, Proposition 4), we know that every Boolean function f is represented by a multilinear polynomial. Moreover, from Lemma E.26, we know that the polynomial $P(\boldsymbol{u})$ representing $f(\boldsymbol{u})$ is block-exclusive for $\boldsymbol{u}$ with the almost p-hot encoding.

To show uniqueness, we replicate the argument from (Kopparty, 2020, Lecture 3, Proposition 4): Given two block-exclusive polynomials P and P' equivalent to f with inputs from the almost p-hot encoding, we have $(P - P')(\boldsymbol{u}) \equiv 0$. Now, assume, for the sake of contradiction, that $P - P' \not\equiv 0$. Here, note that as $P - P'$ is not identically zero and we have a non-zero monomial, and since the inputs are from the almost p-hot encoding, we know that this monomial cannot contain any product of the form $\boldsymbol{u}_{j,k}\boldsymbol{u}_{j,k'}$. Let $S \subseteq [p] \times [\sqrt[p]{c} - 1]$ be a minimal set of indices such that the monomial $\prod_{(j,k)\in S} \boldsymbol{u}_{j,k}$ appears in $P - P'$ with non-zero coefficient. Note that χ_S forms a valid input to f as each block in S can be assigned at most one non-zero entry. Then, since $(P - P')(\chi_S) \neq 0$ as every other monomial will get at least one variable that is assigned to 0 for χ_S , we achieve a contradiction, and thus, $P - P'$ must be identically zero on inputs from the almost p-hot encoding. $\square$

Lemma E.28. *The EQ problem in the almost p-hot encoding setting is represented by a block-exclusive polynomial of degree $2p$.*

Proof. Each input pair $\boldsymbol{u}^1, \boldsymbol{u}^2$ to the EQ problem can be represented as $\boldsymbol{u}^i := (\boldsymbol{u}_0^i, \ldots, \boldsymbol{u}_{p-1}^i)$ for $i \in \{1, 2\}$, where for each $0 < j < p$ such that we have

$$\boldsymbol{u}_j^i := (\boldsymbol{u}_{j,0}^i, \ldots, \boldsymbol{u}_{j,\sqrt[p]{c}-2}^i) \in \{0,1\}^{\sqrt[p]{c}-1}.$$

The following polynomial takes the inner product of each of these one-hot encodings:

$$P^j(\boldsymbol{u}) := \sum_{k=0}^{\sqrt[p]{c}-2} \boldsymbol{u}_{j,k}^1 \cdot \boldsymbol{u}_{j,k}^2 + (1 - \sum_{k=0}^{\sqrt[p]{c}-2} \boldsymbol{u}_{j,k}^1)(1 - \sum_{k=0}^{\sqrt[p]{c}-2} \boldsymbol{u}_{j,k}^2)$$

for $0 < j < p$. Here, note that there can be only be at most 1 in both $\boldsymbol{u}_j^1$ and $\boldsymbol{u}_j^2$, and thus, $P^j(\boldsymbol{u}) = 1$ iff the jth block agree.

Next, the following polynomial is equivalent to the Boolean function that solves the EQ problem:

$$P(\boldsymbol{u}) := \prod_{j=0}^{p-1} P^j(\boldsymbol{u}),$$

and we have $P(\boldsymbol{u}) = \{\boldsymbol{u}^1 \equiv \boldsymbol{u}^2\}$. Here, note that P is multi-linear and has degree $2p$ as each P^j is a degree-2 polynomial. Moreover, P is block-exclusive as each P^j is block-exclusive and we only multiply monomials from different blocks in P.

$\square$

Proposition E.29. *Let P be the block-exclusive polynomial that solves the EQ problem in the p-hot encoding. Then, $\deg(P) \geq 2p$.*

Proof. For the sake of contradiction, assume that there exists a block-exclusive polynomial P that solves EQ in the p-hot encoding setting with degree $\leq 2p - 1$. Then, given an input $\boldsymbol{u} := (\boldsymbol{u}_0, \ldots, \boldsymbol{u}_{p-1})$ from the almost p-hot encoding, where each block $\boldsymbol{u}_i$ corresponds to the

truncated bit string from the one-hot encoding in $\{0,1\}^{\sqrt[p]{c}-1}$, we can convert this input to the p-hot encoding $\boldsymbol{v} := (\boldsymbol{v}_0, \ldots, \boldsymbol{v}_{p-1})$ as follows:

$$\boldsymbol{v}_i := \left(\boldsymbol{u}_{i,0}, \ldots, \boldsymbol{u}_{i,\sqrt[p]{c}-2}, 1 - \sum_{j=0}^{\sqrt[p]{c}-2} \boldsymbol{u}_{i,j}\right)$$

Then, the block-wise multilinear polynomial $Q(\boldsymbol{u}) = P(\boldsymbol{v})$ solves the EQ problem in the almost one-hot encoding setting and has $\deg(Q) \leq \deg(P) \leq 2p - 1$ which contradicts the combination of Proposition E.27 and Lemma E.28. $\square$

Theorem E.30. *A data-independent model needs at least $\lfloor \log(2p) \rfloor$-layers to solve* MQAR *for an input sequence $\boldsymbol{u} \in \{0,1\}^{3N \times d}$ in the p-hot encoding setting, where $d = p \cdot \sqrt[p]{c}$.*

Proof. We know from Corollary E.22 that it suffices to show a lower bound for the EQ problem. Moreover, we know from Proposition E.29 that we cannot solve the EQ problem in the p-hot encoding setting with a block-exclusive polynomial of degree $\leq 2p - 1$. Now, assume that there is a model with L layers that exactly solves EQ in the p-hot encoding setting. Then, due to Lemma E.18 and Proposition E.27, this yields a block-exclusive polynomial $P(\boldsymbol{u})$ of degree at most 2^L. Here, if $L < \lfloor \log(2p) \rfloor$ which , then the resulting with L layers results in a block-exclusive polynomial of degree $\leq 2p - 1$. This contradicts the above claim that we cannot have a block-exclusive polynomial of degree $< 2p$ that exactly represents EQ. Consequently, a data-independent model needs $\geq \lfloor \log(2p) \rfloor$-layers to solve EQ. $\square$

Table 5: BASED Training Settings

	355M	1.4B
Optimizer	Adam	
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.95$	
Optimizer eps	$1e-8$	
Precision	BFloat16	
Warmup	1%	
Learning rate decay	Cosine	
Learning rate (min, base)	8e-5, 8e-4	
Global batch size	256	
Weight decay	0.1	
Num Layers	27	36
Hidden Size	1024	1792
MLP Activation	SwiGLU	
MLP Width	2	
Num. Linear Attn Layers	5	7
Num. Linear Attn Heads	16	
Taylor Feature Dimension	16	
Linear Attn Positional Encodings	None	
Num. Sliding Window Layers	5	7
Sliding Window Size	64	16
Sliding Window Heads	16	
Sliding Window Positional Encodings	Rotary	
Num. Layers	17	22
Projection Expansion Factor	4	
Filter Size	3	
Activation	SiLU	

Table 6: Attention Training Settings

	355M	1.4B
Optimizer	Adam	
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.95$	
Optimizer eps	$1e-8$	
Precision	BFloat16	
Warmup	1%	
Learning rate decay	Cosine	
Learning rate (min, base)	8e-5, 8e-4	
Global batch size	256	
Weight decay	0.1	
Num Layers	24	36
Hidden Size	1024	1680
Num Heads	16	24
RMSNorm	True	
MLP Bias	False	
Flash Attn	True	
Rotary Emb. Fraction	0.5	
MLP Activation	SwiGLU	
MLP Width	4	

Table 7: Mamba Training Settings

	355M	1.4B
Optimizer	Adam	
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.95$	
Optimizer eps	$1e-8$	
Precision	BFloat16	
Warmup	1%	
Learning rate decay	Cosine	
Learning rate (min, base)	8e-5, 8e-4	
Global batch size	256	
Weight decay	0.1	
Num Layers	46	
Hidden Size	1024	2048
RMSNorm	True	
Norm Epsilon	$1e-5$	
Dt State	16	
Dt (Min, Max)	$(0.001, 0.1)$	
Dt Init. Strategy	Random	
Dt Init. Floor	$1e-4$	
Dt Scale	1.0	
Dt Softplus	True	
Projection Expansion Factor	2	
Short Conv Filter Size	4	

Table 8: Hyena Training Settings

	355M
Optimizer	Adam
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.95$
Optimizer eps	$1e-8$
Precision	BFloat16
Warmup	1%
Learning rate decay	Cosine
Learning rate (min, base)	8e-5, 8e-4
Global batch size	256
Weight decay	0.1
Num Layers	29
Hidden Size	1024
Num Heads	1
MLP Width	2
Short Conv. Filter Size	3
Exp. Mod. Decay (Fast, Slow)	0.3, 1.2
Filter Sine Freq. (w)	14
Filter Order	64
Filter Inner MLP	2

Table 9: Hyena Training Settings

	355M
Optimizer	Adam
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.99$
Optimizer eps	$1e-8$
Precision	BFloat16
Warmup	1%
Learning rate decay	Cosine
Learning rate (min, base)	8e-5, 8e-4
Global batch size	256
Weight decay	0.1
Num Layers	24 (No Attention Layers)
Hidden Size	1024
Num Heads	16
MLP Width	4

Table 10: Hyena Training Settings

	355M
Optimizer	Adam
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.99$
Optimizer eps	$1e-8$
Precision	BFloat16
Warmup	1%
Learning rate decay	Cosine
Learning rate (min, base)	8e-5, 8e-4
Global batch size	256
Weight decay	0.1
Num Layers	19
Hidden Size	1024
MLP Width	3.5

Table 11: Gated Linear Attention (GLA) Training Settings

	355M
Optimizer	Adam
Optimizer momentum	$\beta_1, \beta_2 = 0.9, 0.95$
Optimizer eps	$1e-8$
Precision	BFloat16
Warmup	1%
Learning rate decay	Cosine
Learning rate (min, base)	8e-5, 8e-4
Global batch size	256
Weight decay	0.1
Num Layers	24
Hidden Size	1024
Num Heads	4
MLP Width	2