Simple linear attention language models balance the recall-throughput tradeoff

Simran Arora [*1] **Sabri Eyuboglu** [*1] **Michael Zhang** [*1] **Aman Timalsina** [2] **Silas Alberti** [1] **Dylan Zinsley** [2] **James Zou** [1] **Atri Rudra** [2] **Christopher Ré** [1]

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 10.36 accuracy points. We further develop IO-aware algorithms that enable BASED to provide 24× higher throughput on language generation than FlashAttention-2, when generating 1024 tokens using 1.3b parameter models. Overall, BASED expands the Pareto frontier of the throughput-recall tradeoff space beyond prior architectures.

[*]Equal contribution [1]Stanford University [2]University of Buffalo. Correspondence to: Simran Arora <simarora@stanford.edu>, Sabri Eyuboglu <eyuboglu@stanford.edu>, Michael Zhang <mzhang20@stanford.edu>.

Proceedings of the 2nd Efficient Systems for Foundation Models Workshop at the International Conference on Machine Learning (ICML), Vienna, Austria. PMLR 235, 2024. Copyright 2024 by the author(s).

1. Introduction

The choice of sequence mixer (*e.g.* attention, convolution) in a language model affects both its quality and efficiency (Arora et al., 2023a; Vaswani et al., 2017). Prior work shows that attention excels at *recall*, the ability to ground generations in previously seen tokens (Olsson et al., 2022; Arora et al., 2023a). On the other hand, the throughput of attention-based models is bottle-necked during training by quadratic compute complexity and during inference by aggressive memory consumption. The natural question is: *can we improve the real-world speed and memory-use of language models without comprising on quality?*

Recently, a number of architectures have been proposed that enable substantially higher throughput while competing with attention in perplexity (Wang et al., 2022; Gu and Dao, 2023; Yang et al., 2023; Poli et al., 2023; Peng et al., 2023). However, coarse metrics like overall perplexity can obscure important differences in model quality. For example, recent work shows that a specific class of architectures, *gated-convolutions*, despite complexity scaling sub-quadratically in sequence length, are asymptotically less efficient than attention at performing recall (Arora et al., 2023a). Building on this analysis, we evaluate a broader class of architectures across real-world recall-intensive tasks and show attention improves over a currently-popular attention-free alternative, Mamba, by 32.2 accuracy points (Table 1). [1]

Motivated by these observations, we explore the Pareto frontier of the tradeoff between high-recall and high-throughput models. We evaluate a range of architectures on a popular synthetic *associative recall* task (Arora et al., 2023a; Fu et al., 2023a; Olsson et al., 2022). Since generation throughput is bottle-necked by memory consumption, we vary hyperparameters (e.g. model dimension) that affect the size of the *recurrent* state during generation and demonstrate a fundamental recall-memory tradeoff that holds across architecture classes (Figure 2). Attention performs associative recall perfectly, but the recurrent state (*i.e.* the KV-cache) grows linearly with the sequence length. Sliding window

[1]Examples of recall-intensive tasks include information extraction, reading comprehension, summarization and code generation. These require using in context information (contrasting memorized information) during generation.

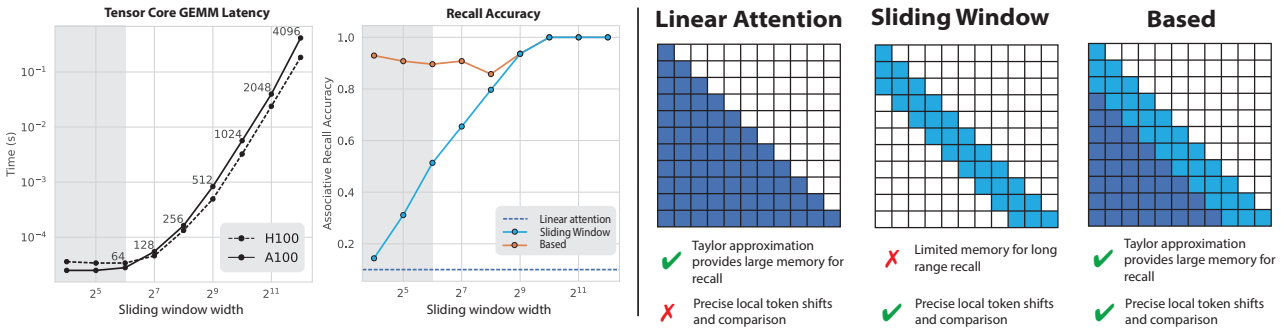

Figure 1. **BASED overview.** Combining linear attention with *tiny* sliding window softmax attention (e.g., 64 or 128 tokens in width) enables improved recall accuracy with limited efficiency overhead vs. smaller tile sizes. (*Left*) Time to execute Cutlass GEMMs (y) vs. sliding window attention size (x), with batch size 512 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.

attention (SWA) can cap the size of the recurrent state at the cost of worse long-range recall (Jiang et al., 2023). However, Mamba, a recently proposed SSM architecture expands the Pareto frontier beyond SWA. This begs the question: *are there other, perhaps simpler, models that can also expand the Pareto frontier?*

To reduce the memory consumption, we consider using two simple techniques: SWA and softmax-approximating linear attention. Our results on language modeling (Table 1) and synthetic recall experiments (Figure 1, center) suggest neither primitive alone suffices to navigate the Pareto frontier.

1. We find that *linear attention* alone struggles to solve associative recall (Figure 1, center). We hypothesize that this is because linear attention lacks the precision to perform local token shifts and comparisons (Fu et al., 2023a; Arora et al., 2023a).

2. In *sliding window attention*, associative recall 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 parallel 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 SWA 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 help model long-range token interactions in the sequence and (2) SWA handles the precise local shifts needed to perform associative recall.

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

1. Despite the theoretical efficiency benefits, *linear attention* implementations are often *slower* than well-optimized attention implementations (Dao et al., 2022). In BASED, we use the 2nd-order Taylor approximation of softmax as the linear attention feature map. 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). To make our attention competitive in real-world wall-clock time and memory usage, we provide hardware-efficient CUDA algorithms for generation prefill (Algorithm 1) and decoding (Algorithm 2). Relative to the baseline, our algorithm reduces data movement from HBM (slower-to-access memory) to SRAM (faster-to-access memory) by $\mathcal{O}(Nd^2)$ bytes and from SRAM to register by $O(Nd^3)$ bytes (Section 5).

2. *Sliding window attention* exploits tensor cores, specialized units on modern GPUs for performing matrix multiplications (GEMMs). While popular architectures use long window sizes (e.g. 4096 for Mistral-7B (Jiang et al., 2023)), we choose small $64 - 128$ windows, guided by hardware properties. In particular, it is important to keep the GPU tensor core occupancy high, while avoiding window sizes that drastically increase latency. Although tensor cores operate on 16×16 tiles, in Figure 1 (left), we see that the latency for performing 16×16 vs. 64×64 (or even 128×128) dimension matrix multiplications is similar because in the former setting, the tensor cores are not fully occupied. Beyond 128, the latency starts increasing quadratically with dimension.

In experiments, we show that BASED competes in quality with strong Transformer++ (Touvron et al., 2023) and SoTA sub-quadratic baselines in models up to the 1.3Bn parameters across language modeling on the Pile language, DNA modeling, and the LM Eval Harness (Gao et al., 2023). Beyond this, BASED outperforms a strong sub-quadratic

architecture, Mamba, on the associative recall slice of the Pile and in downstream recall-intensive tasks by 10.36 accuracy points. In efficiency, BASED enables up to 24× higher throughput than the strong FlashAttention-2 implementation on generation. Code for this work is provided at: `https://github.com/HazyResearch/based`.

2. Preliminaries and Related Work

We discuss the key relevant work in this section and provide an extended discussion in Appendix A.

Attention The *de facto* language modeling primitive, softmax attention (Vaswani et al., 2017) takes inputs $\boldsymbol{x} \in \mathbb{R}^{N \times d}$ of length N and head dimension d, and computes outputs $\boldsymbol{y} \in \mathbb{R}^{N \times d}$ via the softmax over projections $\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v} = \boldsymbol{x}\boldsymbol{W}_q, \boldsymbol{x}\boldsymbol{W}_k, \boldsymbol{x}\boldsymbol{W}_v$, *i.e.*,

$$\boldsymbol{y}_i = \sum_{j=1}^{i} \frac{\exp(\boldsymbol{q}_i^\top \boldsymbol{k}_j / \sqrt{d}) \boldsymbol{v}_j}{\sum_{m=1}^{i} \exp(\boldsymbol{q}_i^\top \boldsymbol{k}_m / \sqrt{d})} \tag{1}$$

in the causal case where $\boldsymbol{W}_q, \boldsymbol{W}_k, \boldsymbol{W}_v \in \mathbb{R}^{d \times d}$ are learnable matrices . While effective at recall (Arora et al., 2023a) and efficient to train (Eq 1 is parallelizable on GPUs and $\mathcal{O}(N)$ in memory with recent advances (Dao et al., 2022)), attention remains expensive for generation. For every new output $\boldsymbol{y}_n$, we require nd operations over a growing *KV-cache* of prior $\{\boldsymbol{k}_i, \boldsymbol{v}_i\}_{i=1}^{n-1}$. This results in larger memory consumption and lower-throughput for longer sequences.

Efficient attentions Various works thus try to improve on attention's efficiency without sacrificing quality. *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). While further popularized in large language models (Mistral, Jiang et al. (2023)), prior works either underperform full attention with sparse patterns that fail to capture dense interactions, or use large window sizes that still permit large KV-caches and subsequent inefficiency.

Meanwhile, *linear attentions* replace the softmax in standard attention with alternative kernel functions (Katharopoulos et al., 2020a; Choromanski et al., 2020; 2021; Qin et al., 2022a; Keles et al., 2023). 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., 2020b). 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). However, present linear attention feature maps either fail to match standard attention on recall or remain expensive to compute (Zhang et al., 2024). Linear attentions are slower in wall-clock time compared to optimized attention implementations (Dao et al., 2022).

The line of work studying how to combine sparse and linear attention into a single layer is also closely related to our work (Zaheer et al., 2020; Beltagy et al., 2020; Chen et al., 2021a; Zeng et al., 2022).

Attention alternatives Finally, various models use attention-free sequence mixers such as state-space models (SSMs) (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 and Dao, 2023) to rival attention performance while improving its efficiency. However, while recent such models can match attention in overall perplexity, further study suggests they may underperform Transformers on tasks such as recall and in-context learning (Arora et al., 2023a; Akyürek et al., 2024).

3. No Free Lunch: Memory-Recall Tradeoff

In this section, we demonstrate a fundamental tradeoff between a model's memory consumption during inference (*i.e.,* the size of its recurrent state) and its capacity to perform recall. We use a combination of experiments on synthetic data and theoretical analysis.

- **Empirical study of memory-recall tradeoff :** In Section 3.1, we evaluate a number of popular architecture classes (*e.g.* Mamba, Hyena) on a synthetic associative recall task, varying hyperparameters that affect the model's recurrent state size (Figure 2). Within each architecture class, we observe a clear tradeoff: the larger the recurrent state size, the better recall. However, for a fixed recurrent state size, performance is not consistent across architectures. We observe that some sequence mixers fall well-below the Pareto-frontier. This motivates the design of sequence mixers that can expand the Pareto frontier.

- **Lower bounds on memory required for recall**: In Section 3.2, we lower bound the recurrent state size required to perform exact recall with *any* recurrent model Theorem F.6. This analysis reinforces our empirical observations on the throughput-recall tradeoff.

3.1. Empirical study of memory-recall tradeoff

Setup. We use a synthetic AR task called Multi-Query Associative Recall (MQAR) (Arora et al., 2023a) to demonstrate the trade-off. In this task, input sequences consist of a number of key-value pairs followed by queries. For a given query, the model must recall the corresponding key-value pair from earlier in the sequence in order to predict the next token. For example, the correct output for input below would be 4, 6, 1, 2, 3:

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

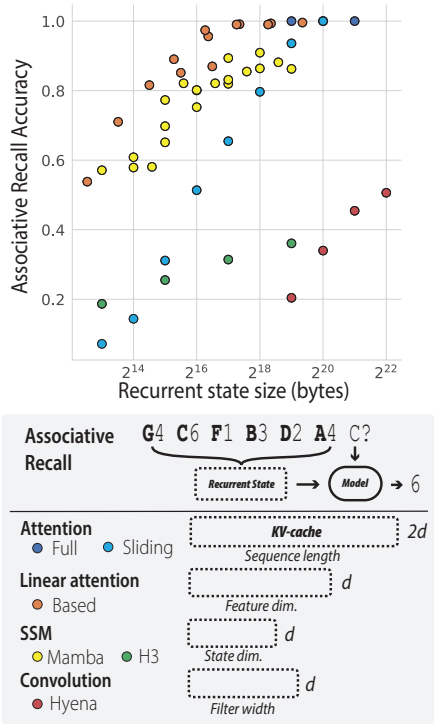

Figure 2. **Throughput (memory) - recall tradeoff.** x-axis shows state size (bytes) during generation; y-axis shows accuracy on the MQAR MQAR recall task (Arora et al., 2023a). For each architecture, we train several models varying hyperparameters that affect the recurrent state size (*e.g.* model dimension). The plot shows a fundamental tradeoff between the recurrent state size and recall capacity that applies to broad class of models (Arora et al., 2023a; Gu and Dao, 2023; Fu et al., 2023a).

We train on sequences of length 256 tokens containing between 4 and 64 key-value pairs. During evaluation, we measure accuracy on sequences of length 1,024 tokens containing between 4 and 256 key-value pairs.

We train and evaluate six sequence mixers: attention (Vaswani et al., 2017), sliding window attention (Beltagy et al., 2020), Mamba (Gu and 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 (e.g., in sliding window attention we vary the window width). We measure how MQAR accuracy varies with the size of the recurrent state and Appendix E.1 contains details on how state sizes are calculated.

Figures 2 and 3 can be reproduced or extended to new architectures using the scripts provided at `https://github.com/HazyResearch/zoology`.

Results In Figure 2, we demonstrate a fundamental tradeoff between recurrent state size and accuracy on MQAR that holds within and across architecture classes. Within each architecture class (*e.g.* H3 models), increasing the recurrent state size almost always leads to an improvement in accuracy. Across architecture classes, we see a tradeoff as well. Attention achieves perfect recall accuracy, but its recurrent state size grows with the length of the sequence. Other architecture classes like Mamba and H3 admit models with much smaller recurrent states, but these models have limited recall capacity.

Given a fixed recurrent state, not all architectures have the same recall capacity. Among architectures proposed in prior work, Mamba makes the best use of a limited memory budget. Notably, architectures with a convolutional view (*e.g.* Hyena and H3) fall well below the Pareto frontier. Our proposed architecture, BASED (introduced in Section 4), expands the Pareto-frontier beyond Mamba. By varying hyper-parameters that determine its state size (*e.g.* feature dimension and model dimension), we can smoothly navigate the tradeoff between efficient models and memory-hungry models with high recall capacity.

3.2. Theoretical Analysis

Our theoretical analysis provides further insight into the empirical observations described above. 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 F.12 in Appendix F).

Theorem 3.1. *Any recurrent model*[2] *depending causally on input* $\boldsymbol{u} \in \{0,1\}^{N \times d}$ *requires* $\Omega(N)$*-bits*[3] *in state size to solve* MQAR.

This result suggests that the tradeoff observed in Figure 2 is fundamental, not an artifact of architectural quirks.

Next, we focus on *gated-convolutions*, a broad class of architectures built from gating and convolutions (*e.g.* H3, Hyena, RWKV v4). To make progress in theoretically analyzing the broad set of gated convolution proposals, prior work develops a *canonical* gated-convolution, referred to as `BaseConv` which can provably simulate *any* architecture built from gating and convolution primitives.

Building on this work, we show that `BaseConv` cannot solve MQAR in constant-many layers (Theorem F.19 and Theorem F.29 in Appendix F).

Theorem 3.2. *Given an input sequence* $\boldsymbol{u} \in \{0,1\}^{3N \times d}$*, where* N *and* d *denote the sequence length and head dimension, respectively, a data-independent* `BaseConv` *model needs* $\log(2d)$*-layers to solve* MQAR *for* $d = \log_2(c)$*, where* c *denotes the vocabulary size*[4].

[2]For Mamba (Gu and Dao, 2023), see Corollary F.13.

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

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

Remark 3.3. For a class of input encodings that generalizes one-hot encodings termed as p-hot encodings (Definition F.22), input-dependent `BaseConv` needs at least $\lfloor \log(2p) \rfloor$-layers to solve MQAR where $d = p \cdot \sqrt[p]{c}$.

The above result is not as strong when $c \ll N$, for which we prove a complementary lower bound (Theorem F.14 in Appendix F):

Theorem 3.4. *Given an input $\boldsymbol{u} \in \{0,1\}^{N \times d}$ to the* MQAR *with any encoding such that $\log c \leq d \leq 2^{(\log N)^{1-\epsilon}}$ for $\epsilon > 0$, and c possible tokens from the vocabulary with $c \leq N$, a data-independent `BaseConv` model with model parameters taking $O(\log N)$ bits needs $\Omega(\epsilon \log \log N)$ layers to solve AR.*

In contrast, Arora et al. (2023a) show that attention solves MQAR in constant-many layers. This result helps to explain why the gated-convolution architectures (H3 and Hyena) in Figure 2 lie below the Pareto frontier established by newer architectures.

Note that Theorem 3.2 and Theorem 3.4 imply that we need $\Omega(\max(\log \log c, \log \log N))$ many `BaseConv` layers to solve MQAR. One might wonder if we can improve this lower bound. In Theorem F.30, we show that this is the best possible lower bound by showing that for certain settings, $O(\max(\log \log c, \log \log N))$ `BaseConv` layers *are* enough to solve MQAR.

Finally, we show that we can simulate linear attention (Katharopoulos et al., 2020a), the foundation of BASED, using `BaseConv` (Arora et al., 2023a) with a poly-log blowup in the number of layers (Proposition F.8 in Appendix F), pointing to the relative efficiency of linear attention over gated-convolution architectures.

4. The BASED Architecture

In this section, we introduce BASED. Our objective in designing this architecture is to demonstrate how we can navigate the Pareto-frontier of the memory-recall tradeoff using well-known architectural building blocks.

Softmax attention excels at recall, but since its recurrent state, the KV-cache, grows unconstrained with the length of sequence, it is stuck in the upper right quadrant of Figure 2. We study two simple approaches for constraining the size of attention's recurrent state: linear attention and sliding window attention. The recurrent state size of linear attention (*i.e.* attention without softmax) does not grow with the sequence length and can be modulated by changing simple hyperparameters (Katharopoulos et al., 2020a). With sliding window attention, we cap the recurrent state size to be the width of the window.

However, our experiments on real-world language modeling (Table 6) and synthetic associative recall (Figure 1 middle) suggest that neither primitive alone suffices to navigate the pareto frontier. Linear attention lacks the precision to perform local token shifts and comparisons (Fu et al., 2023b; Arora et al., 2023a). In sliding window attention, associative recall range is limited by the width of the windows (Figure 2, center). As we increase the window size, the recurrent state grows linearly and has a non-linear effect on speed during parallel training and inference (Figure 2, left).

BASED combines (1) softmax-approximating linear attention applied globally and (2) exact softmax attention applied locally in small sliding windows (Figure 1, right). This allows us to use softmax attention in surprisingly small sliding windows (*e.g.,* $64 - 128$ tokens) that recover 90.8% of full softmax attention's recall accuracy at 1e-5$\times$ its latency.

4.1. Taylor Linear Attention

By approximating softmax attention using linear feature maps, we can constrain the size of the recurrent state while maintaining global token interactions (*i.e.* each token depends on every token before it in the sequence).

Katharopoulos et al. (2020a); Tsai et al. (2019); Choromanski et al. (2020) show that we can select a feature map $\phi : \mathbb{R}^d \to \mathbb{R}^{\tilde{d}}$ such that $\phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j) \approx \exp(\boldsymbol{q}_i^\top \boldsymbol{k}_j / \sqrt{d})$. We can then rewrite the formula for softmax attention in Equation (1) as

$$\sum_{j=1}^{i} \frac{\phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j) \boldsymbol{v}_j}{\phi(\boldsymbol{q}_i) \sum_{j=1}^{i} \phi(\boldsymbol{k}_j)} = \frac{\phi(\boldsymbol{q}_i) \sum_{j=1}^{i} \left(\phi(\boldsymbol{k}_j)^\top \boldsymbol{v}_j\right)}{\phi(\boldsymbol{q}_i) \sum_{j=1}^{i} \phi(\boldsymbol{k}_j)} \tag{2}$$

where every query attends to every past key in $\mathcal{O}(Nd^2)$ time and space complexity. Furthermore, Katharopoulos et al. (2020b) show that linear attention has a fixed size recurrent state during 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, we can compute Equation (2) as

$$\boldsymbol{s}_i = \boldsymbol{s}_{i-1} + \phi(\boldsymbol{k}_i)^\top \boldsymbol{v}_i, \quad \boldsymbol{z}_i = \boldsymbol{z}_{i-1} + \phi(\boldsymbol{k}_i)^\top,$$
$$\boldsymbol{y}_i = \frac{\phi(\boldsymbol{q}_i) \boldsymbol{s}_i}{\phi(\boldsymbol{q}_i) \boldsymbol{z}_i} \tag{3}$$

where $\boldsymbol{s}_i \in \mathbb{R}^{d \times \tilde{d}}$ and $\boldsymbol{z}_i \in \mathbb{R}^{\tilde{d}}$.

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

$$\phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j) = 1 + \boldsymbol{q}_i^\top \boldsymbol{k}_j + \frac{(\boldsymbol{q}_i^\top \boldsymbol{k}_j)^2}{2} \tag{4}$$

While Zhang et al. (2024) note that picking a feature map with $\tilde{d} = d^2$ results in linear attention with $\mathcal{O}(Nd^3)$ time

and space complexity and large recurrent state of size $O(d^3)$, we can tradeoff efficiency for recall capacity by projecting queries and keys to smaller dimensions *i.e.,* $\boldsymbol{W}_q, \boldsymbol{W}_k \in \mathbb{R}^{d \times d'}$ with $d' = 16$. By changing d' we modulate the size of the recurrent state.

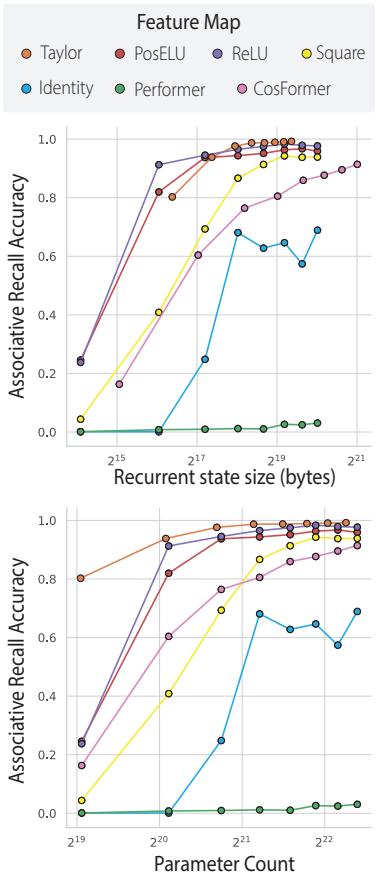

Figure 3. **Linear attention feature maps on AR.** x: state size (bytes) during generation or param. count; y: MQAR accuracy. This setting is harder than Figure 2 (256 key-value pairs).

How does the choice of feature map affect the memory-recall tradeoff? Prior work demonstrates the strong performance of the Taylor feature map on associative recall (Zhang et al., 2024). Building on this analysis, we evaluate a broad set of feature maps ($\phi_{\text{ReLU}}(x) = \max(x, 0)$, $\phi_{\text{PosELU}}(x) = \text{ELU}(x) + 1$, $\phi_{\text{Square}}(x) = x^2$, $\phi_{\text{Identity}}(x) = x$, $\phi_{\text{CosFormer}}$ as defined in (Qin et al., 2022a), and $\phi_{\text{Performer}}$ as defined in (Choromanski et al., 2020)) using the experimental setup described in Section 3.1. 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 6 informed our decision to use the Taylor approximation, though other simple feature maps may be effective as well.

4.2. Local Exact Attention with Small Sliding Windows

To efficiently model fine-grained local interactions, BASED uses sliding window attention (SWA) with window sizes set at small multiples of 16 (up 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.

However, unlike past SWA that keep w at sizes 256 (Parmar et al., 2018) to 4096 (Jiang et al., 2023), BASED uses only $w \leq 128$ to best exploit modern GPUs. In Section 5, we discuss how this "Tensor core-aware" window (TCWINDOW) achieves 1e-5× the latency than the $w = 4096$ windows in modern LLMs (*e.g.,* Mistral 7B (Jiang et al., 2023)).

While the small w in TCWINDOW enable fast local and exact attention, it presents a challenge for long range modeling. With just $w = 64$, for every layer of $w = 4096$ Mistral sliding window attention we would require 64 layers of BASED to achieve the same receptive field. Controlling for model depth and sequence length, Figure 2 indeed shows smaller w linearly decreasing in associative recall accuracy. BASED's global *linear attention* described above overcomes the lack of long-range modeling presented with low w.

Finally, we find that including gated convolution layers with short convolutions (e.g., filter size 3) gives additional benefit over only using TCWINDOW layers. Short convolutions can help perform local, precise shifts for token comparisons since they operate over the full sequence, while TCWINDOW does not. These local mixers can complement one-another.

Additional architectural details for BASED are discussed in Appendix C and the hybridization of layers used in experiments are provided in Table 7. We include ablations of architectural choices in Table 6 and evaluate the overall quality and efficiency of BASED in Section 6.

5. Efficient Implementation

In this section we focus on the efficiency of BASED. A naïve implementation is *slower* than the most efficient standard attention implementations (shown in Figure 4) as it requires large amounts of high latency memory movement. We first describe preliminaries of the GPU execution model and memory hierarchy. We next present the baseline and our hardware-aware algorithms for linear attention in Section 5.2 and for sliding window attention in Appendix B.2.2.

5.1. Preliminaries

GPU operations, or *kernels*, are executed by many parallel threads. GPU *streaming multiprocessors* launch *thread blocks* at the software level. These blocks are divided into *warps* (*e.g.* 32 threads) that are assigned to cores at the hardware level. Threads need to read inputs into their *registers* to perform computations and write the outputs. The time taken to read and write is referred to as the IO cost.

Operations could either be memory or compute bound, depending on the time to load data vs. perform computations on loaded data. In designing our IO-aware algorithms, we would like to exploit two key properties of modern GPUs. First, tensor core units (fast matrix multiply units) achieve 312 TFLOP/s speeds relative to 19 TFLOP/s for the non-matrix multiply cores. Second, GPUs face a memory hierarchy with large amounts of slow-to-access memory and smaller amounts of fast-to-access memory. For instance, the hierarchy on a modern NVIDIA 80GB A100 GPU is: 80GB of HBM with 2 TB/s bandwidth, 80MB of L2 cache, 192KB of L1 cache / shared memory (implemented via SRAM) with 19 TB/s bandwidth per SM, and 256 KB of register file per SM (NVIDIA, 2022). Register memory is private to an executing thread, so threads need to write to shared memory to communicate data to other threads in the block. To reduce the IO cost, a key principle is to *fuse* multiple operations on the same data slice while it's in fast memory before writing it back to slow memory.

5.2. Taylor Exponential Linear Attention

Despite the theoretical efficiency, the popular linear attention implementations are less efficient than well-optimized softmax attention implementations when measured in real-world wall-clock time and memory usage (Dao et al., 2022). We next present hardware-aware algorithms to make Taylor linear attention efficient. We focus on two operations: (1) prefill (this section), corresponding to processing the prompt during generation or the forward pass during training, and (2) next token prediction during generation (Appendix B), which also requires updating the recurrent hidden state state.

In this section, we refer to the batch size as B, number of heads as H, head dimension as d, sequence length as N and feature dimension as d', following Section 4. For ease of notation, let $D = 1 + d' + d'^2$ in this section. Additional details for these algorithms are in Appendix B

Baseline Implementation The naïve implementation detailed in Appendix B only uses a CUDA kernel to compute the causal dot product between $\boldsymbol{q}$, $\boldsymbol{k}$, and $\boldsymbol{v}$ projections (Vyas et al., 2020), but computes the feature maps in python (non IO-aware). This is inefficient given the computation required for the feature map computation.

Analysis In overall IO cost, ignoring the input and output projections in the linear attention layer, this procedure requires $2BHND$ bytes for writing featurized $\boldsymbol{q}, \boldsymbol{k}$ to HBM. During the causal dot product, this requires $2BHND + BHNd$ bytes to read $\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}$ tiles and $BHNd$ bytes to write the result. Throughout the computation, $\mathcal{O}(BHNDd)$ bytes (note this is the shape KV state during the forward pass) are read in and out of thread registers to SRAM to update the running output and KV state at 19TB/s bandwidth.

Algorithm Our kernel computes *both* the feature map and causal dot product, detailed in Algorithm 1. Here we describe the key insights. First, to handle *causality* in the dot-product computation, for each *tile* of output $y_i \in \mathbb{R}^{16\times d}$, we split the computation as shown, where $\boldsymbol{q_i}, \boldsymbol{k_i}, \boldsymbol{v_i}$ are also now tiles of 16 tokens, handled in parallel by the kernel.

$$\boldsymbol{y_i} = \text{Causal}(\boldsymbol{q_i}^T \boldsymbol{k_i})\boldsymbol{v_i} + \boldsymbol{q_i} \sum_{j=0}^{i-1} (\boldsymbol{k_j} \boldsymbol{v_j})$$

where the first term uses the quadratic attention view and requires applying causal masking. The second term uses the linear view and its causality has already been handled.

Second the large KV-state, $\sum_{j=0}^{i-1}(\boldsymbol{k_j}\boldsymbol{v_j}), \in \mathbb{R}^{D\times d}$, needs to be stored as we iterate over the length-16 tiles. By partitioning across workers (warps), we can store the state *in thread registers* (fastest memory). The partitioning is restricted by (1) each warp has a limited quantity of threads and (2) warps cannot access the thread memory of other warps.

Analysis In IO cost, again ignoring the input and output projections in the linear attention layer, our procedure requires $2BHNd'$ bytes for reading q, k and $2BHNd$ bytes for reading v and writing output y between HBM and SRAM. Overall, our algorithm avoids in HBM $\mathcal{O}(2BHND)$ bytes in HBM to SRAM data movement. We further improve upon the baseline by storing the KV-state *in-register* to avoid the $\mathcal{O}(BHNDd)$ bytes in SRAM to register data movement.

End-to-end benchmarks for BASED implemented with these IO-aware algorithms are provided in Section 6. Micro-benchmarks for each kernel against the baseline implementations are provided in Appendix B.

6. Results

In this section, we present results for the following claims:

1. **Language modeling overall.** We evaluate architectures in pretraining on the Pile (Gao et al., 2020) and on standard natural language understanding benchmarks. We find BASED matches or outperforms prior sub-quadratic architectures across these settings.

2. **Language modeling recall.** BASED closes the gap to attention on the challenging associative recall slice of the Pile corpus (see Table 1). We apply these pretrained

Architecture	Params/Tokens	Efficiency Prefill Tok./ms ↑	Efficiency Generate Tok./ms ↑	Language Modeling (Pile) All Ppl. ↓	Language Modeling (Pile) AR Ppl. ↓	Language Modeling (Pile) Other Ppl. ↓	Info. Extraction SWDE Acc ↑	Info. Extraction FDA Acc ↑	QA SQUAD F1 ↑	Common LM-Evals Avg. Acc. ↑
Transformer++	1.33b/10b	103.50	0.99	**7.26**	**1.74**	**8.10**	**71.92**	**73.23**	**36.19**	**47.64**
BASED	1.35b/10b	**161.71**	24.28	7.43	1.87	8.26	48.06	24.41	30.46	46.68
Mamba	1.32b/10b	112.22	**25.69**	7.48	1.96	8.29	34.74	12.89	28.20	46.84
Transformer++	1.33b/50b	103.50	0.99	**6.28**	**1.65**	6.82	**76.50**	**80.47**	**43.47**	53.33
BASED	1.35b/50b	**161.71**	24.28	6.30	1.71	6.82	64.45	30.40	41.62	**53.81**
Mamba	1.32b/50b	112.22	**25.69**	**6.28**	1.74	**6.78**	52.75	18.51	35.92	53.50
Transformer++	360m/10b	207.77	23.82	**8.39**	**1.87**	**9.42**	**57.97**	**58.00**	**27.18**	**44.08**
BASED	363m/10b	**514.57**	**47.23**	8.65	2.07	9.64	29.16	11.71	25.07	43.03
Mamba	358m/10b	267.09	39.95	8.64	2.21	9.59	23.67	6.53	24.06	43.51
GLA	362m/10b	—	—	9.12	2.36	10.68	—	—	—	—
RWKV v5	362m/10b	—	—	9.79	2.40	10.90	—	—	—	—
H3	362m/10b	—	—	10.60	4.88	11.23	6.75	0.64	7.87	39.35
Transformer++	360m/30b	103.50	0.99	**7.68**	**1.80**	8.40	**70.75**	**63.79**	25.07	44.75
BASED	363m/30b	**161.71**	24.28	7.77	1.93	8.46	45.01	16.45	**32.67**	45.36
Mamba	358m/30b	112.22	**25.69**	7.73	2.02	**8.38**	27.63	8.71	26.71	**45.62**

Table 1. **Evaluation of pre-trained language models.** Models were trained on the same sets of 10b to 50b tokens drawn from the Pile corpus (Gao et al., 2020). We report inference throughput on $4,096$ tokens ($16,384$ for 360m param.) of pre-fill and $2,048$ tokens of recurrent generation for a subset of architectures. We report language model perplexity on the overall Pile test set as well as perplexity on two slices of the test set: associative recall tokens and other tokens (see Section 6.1, (Arora et al., 2023a)). We report zero-shot performance on three *recall-intensive* tasks: information retrieval on SWDE and FDA as well as question answering on SQUAD. Finally, we report average performance on the set of *LM Eval Harness* (Gao et al., 2023) common sense reasoning tasks used in Gu and Dao (2023), details in Appendix D. These tasks do not require significant recall capacity because the input text is typically very short. See Section 6.1. Some proposed architectures that do not implement recurrent views for generation are marked with a —.

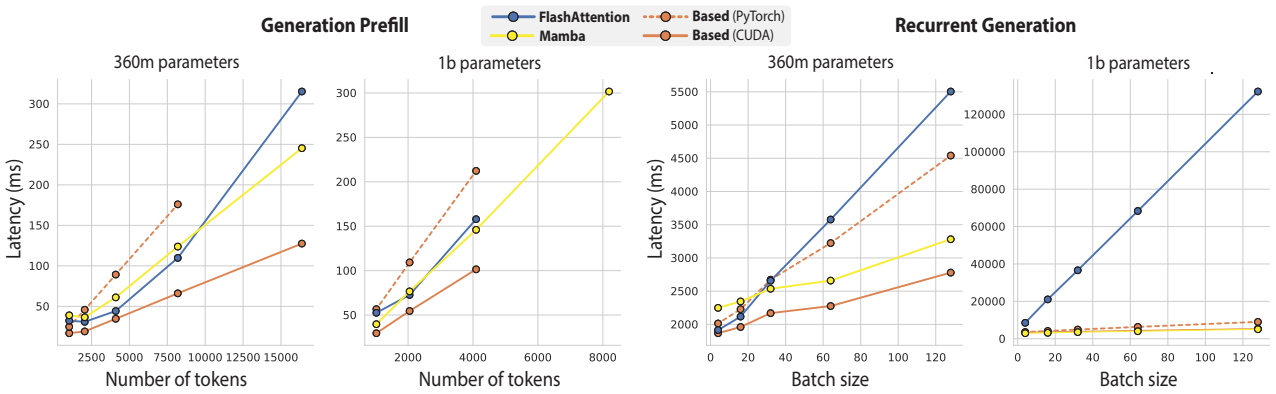

Figure 4. (**Left**) Throughput numbers for the varied prefill sequence lengths at a fixed batch size of 2. **Right** generation throughput at varied batch sizes at a fixed generation length of 1024 tokens. The y-axis shows the in latency (ms). Lines are cutoff when the model runs out of memory. We show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

models zero-shot to a suite of recall-intensive tasks (*e.g.* information extraction, QA), showing that BASED systematically outperforms Mamba by large margins (10.36 accuracy points at 1.3b parameters and 50b tokens).

3. **Generation throughput.** Our IO-aware implementation of recurrent generation in Based enables $40-60\%$ speedups relative to FlashAttention-2 and Mamba for prefill at 4k sequence length and up to $24\times$ higher throughput over FlashAttention-2 in generating 1024 tokens at batch size 128 (see Figure 4).

Baselines We compare to several key baselines at the 360m and 1.3b parameter scales, up to 50b tokens of training. We first consider Transformer++, Transformers with modern improvements such as rotary encodings (Su et al., 2023) and gated linear units (Touvron et al., 2023). We then consider a class of popular 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 the recently popular Mamba (Gu and Dao, 2023) and Gated Linear Attention (Yang et al., 2023) linear recurrent architectures with input-dependent recurrent-state updates. We give each architecture the Transformer++ improvements as relevant and use the implementations provided by prior work during training.

BASED combines familiar local and global sequence mixers to achieve high quality. We train BASED as a hybrid of $\approx 20\%$ linear attention, $\approx 20\%$ sliding window attention, and $\approx 60\%$ gated convolution layers as discussed in Appendix E.1. In contrast to recent baselines, BASED requires *no input-dependent decays* whatsoever.

6.1. Language Modeling Evaluations

Language Modeling Benchmarks We pretrain language models from scratch at two parameter scales (360m and 1.3b parameters) on the Pile (Gao et al., 2020). Each model sees the same tokens of pretraining data in the same order. The Pile data is tokenized using the GPT-2 BPE tokenizer (Radford et al., 2019). We measure perplexity on the Pile and report results in Table 1 and further experimental details are provided in Appendix E.1.

We additionally evaluate the pretrained models on key natural language understanding downstream benchmarks using the LM Eval Harness (SuperGLUE, ARC, PIQA, WinoGrande, HellaSwag, LAMBADA). A detailed breakdown of tasks and metrics can be found in Appendix D.

In both pretraining and on the downstream tasks, BASED consistently competes with the strongest Transformer++ and Mamba baselines. While these overall metrics are helpful, we next turn to a fine-grained analysis of recall and in-context learning ability on real-world data.

Recall Evaluations We evaluate our pretrained models on a suite of in-context learning tasks selected to test the downstream recall capacity in Table 1. These tasks fall into three categories: (1) **Real-world AR** Beyond perplexity scores, we slice the next token predictions on the Pile to understand each architecture's AR quality (Appendix E.1). (2) **Information extraction (IE)** SWDE and FDA are popular semi-structured and unstructured document IE benchmarks respectively (Wu et al., 2021; Deng et al., 2022; Arora et al., 2023b). SWDE has HTML for 8 Movie and 5 University websites (e.g. IMDB, US News) and annotations for 8-274 attributes per website (e.g., Movie `runtime`), and (3) **Question answering** from in-context passages.

We find BASED outperforms the baseline sub-quadratic architectures across these evaluations, closing the gap to Transformer++. These trends track the MQAR synthetic results from Section 3.1. We further observe that as we train for longer (more tokens), the improvements from BASED over Mamba grow (from 3.9 to 9.0 points on average at 360m scale and from 9.0 to 10.4 points at the 1.3b scale).

Quality Ablations In Table 6, we ablate the feature maps, feature dimensions, and sliding window and convolution dimensions using the same Pile setting as prior experiments. In feature maps, we consider replacing the Taylor approximation with CosFormer (Qin et al., 2022a) or Performers (Choromanski et al., 2020), and varying the state size using linear projections. We observe with larger sate size, CosFormer closes the gap to the Taylor map though note the projections increase the parameter count. In feature dimension, we find 24 and 32 provide diminishing returns. Further discussion is in Appendix E.1.

6.2. Efficiency Benchmarks

We benchmark the throughput of BASED, with and without our proposed IO-Aware algorithms (Section 5, Figure 4). We consider both the **forward pass / generation prefill** and **next token prediction** stages. Experiments were run using an H100 NVIDIA GPU and averaged over 20 repetitions.

End-to-end benchmarks Using our efficient implementation (Section 5), BASED achieves 56% faster prefill than FlashAttention-2 (Dao, 2023) and 44% faster than Mamba at $4k$ sequence length and 1.3b parameters (28% faster than FlashAttention-2 and 76% faster than Mamba at $360m$ parameters). We find that next token generation, with *no prefill*, provides $24\times$ higher throughput (tokens/second) over the highly optimized FlashAttention-2 implementation and achieves 95% and the throughput of the recurrent Mamba architecture at batch size 128 and 1.3b parameters (98% higher throughput vs. FlashAttention-2 and 118% higher throughput vs. Mamba at 360m parameters). All benchmarks is on a single NVIDIA H100 GPU, using CUDA cache graphs during next token prediction (NVIDIA, 2019).

In Figure 4, we also include results for the baseline implementation of BASED that uses the popular Fast Transformers CUDA kernel to compute the causal dot product (Vyas et al., 2020) (discussed in Section 5). The *custom kernel* introduced in our work unlocks the efficiency of BASED.

Micro benchmarks As the end-to-end BASED architecture is a hybrid architecture, we provide micro benchmarks of the individual kernels against key baseline implementations in Appendix B. Kernels are accessible at: `https://github.com/HazyResearch/ThunderKittens`.

7. Conclusion

This work identifies a fundamental tradeoff between recall, a critical skill for in-context learning, and throughput through theory and experiments. Attention performs recall perfectly, but requires retaining a KV cache that grows with the sequence length. As an alternative, BASED combines two simple techniques — local fine-grained attention and long-range linear attention via a Taylor approximation of the softmax exponential function – that are sub-quadratic during training and permit an efficient recurrent inference view. To enable wall clock efficiency, we introduce IO-aware algorithms for the Taylor linear attention inference that lead BASED to perform generation up to $24\times$ faster than FlashAttention-2 at the 1.3b parameter scale (generating 1024 tokens, batch size 128). Beyond competing in overall perplexity, BASED outperforms prior sub-quadratic architectures in recall quality by 10.36 accuracy points on average. Overall, our results show that BASED extends the Pareto frontier of the recall-throughput tradeoff space beyond prior architectures.

Acknowledgments

We thank Benjamin Spector, Dylan Zinsley, Songlin Yang, Daniel Fu, 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.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. We intend for BASED to aid in reducing the costs of machine learning and in unlocking new capabilities. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here. Detailed discussions of the risks of using and developing LLMs are in Bommasani et al. (2021); Weidinger et al. (2021).

References

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.

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.

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.

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

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

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.

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.

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.

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.

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.

Tri Dao. FlashAttention-2: Faster attention with better parallelism and work partitioning. 2023.

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.

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.

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, page 1–23, 2023.

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.

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.

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

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

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

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

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.

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.

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*, 2022a.

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)*, 2020b.

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*, pages 10630–10643, Online and Punta Cana, Dominican Republic, November 2021. Association for Computational Linguistics. doi: 10.18653/v1/2021.emnlp-main.830.

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

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.

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.

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.

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

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.

Ekin Akyürek, Bailin Wang, Yoon Kim, and Jacob Andreas. In-context language learning: Architectures and algorithms. 2024.

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.

Yao-Hung Hubert Tsai, Shaojie Bai, Makoto Yamada, Louis-Philippe Morency, and Ruslan Salakhutdinov. Transformer dissection: a unified understanding of transformer's attention via the lens of kernel. *arXiv preprint arXiv:1908.11775*, 2019.

NVIDIA. Nvidia H100 tensor core GPU architecture, 2022.

A. Vyas, A. Katharopoulos, and F. Fleuret. Fast transformers with clustered attention. 2020.

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.

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

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

Eric Wu, Kevin Wu, Roxana Daneshjou, David Ouyang, Daniel Ho, and James Zou. How medical ai devices are evaluated: limitations and recommendations from an analysis of fda approvals. *Nature Medicine*, 27:1–3, 04 2021.

Xiang Deng, Prashant Shiralkar, Colin Lockard, Binxuan Huang, and Huan Sun. Dom-lm: Learning generalizable representations for html documents. 2022.

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.

NVIDIA. Getting started with cuda graphs, 2019. URL https://developer.nvidia.com/blog/cuda-graphs/.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Laura Weidinger, John Mellor, Maribeth Rauh, Conor Griffin, Jonathan Uesato, Po-Sen Huang, Myra Cheng, Mia Glaese, Borja Balle, Atoosa Kasirzadeh, et al. Ethical and social risks of harm from language models. *arXiv preprint arXiv:2112.04359*, 2021.

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

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

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.

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

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

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, pages 14138–14148, 2021.

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, editors, *Advances in Neural Information Processing Systems*, 2021b.

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

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

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

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*, pages 7025–7041, Abu Dhabi, United Arab Emirates, December 2022b. Association for Computational Linguistics. doi: 10.18653/v1/2022.emnlp-main.473.

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

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

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.

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

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

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

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

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.

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.

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

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.

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

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*, pages 611–626, 2023.

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.

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

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)*, pages 250–263, 2019.

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

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

Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus), 2023.

Alex Wang, Yada Pruksachatkun, Nikita Nangia, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. *SuperGLUE: a stickier benchmark for general-purpose language understanding systems*. Curran Associates Inc., Red Hook, NY, USA, 2019.

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.

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

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

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

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

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.

Katarina Gresova, Vlastimil Martinek, David Cechak, Petr Simecek, and Panagiotis Alexiou. Genomic benchmarks: A collection of datasets for genomic sequence classification. *bioRxiv*, 2022. doi: 10.1101/2022.06.08.495248.

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, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 3047–3056, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics. doi: 10.18653/v1/N19-1309.

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

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

Swastik Kopparty. Topics in algorithms and complexity theory: Spring 2020. 2020.

The appendix is organized as follows:

1. Appendix A includes an extended related works discussion.
2. Appendix B includes details on the IO-aware implementation and benchmarking for BASED.
3. Appendix C includes additional discussion of BASED architectural details.
4. Appendix D provides additional experimental results.
5. Appendix E provides experimental details.
6. Appendix F includes theoretical results and proofs.

To facilitate reproducing this work we release:

1. Code for model training and inference at `https://github.com/HazyResearch/based`
2. Model checkpoints at `https://huggingface.co/collections/hazyresearch/`
3. CUDA kernels at `https://github.com/HazyResearch/ThunderKittens`
4. Code for synthetic MQAR experiments at `https://github.com/HazyResearch/zoology`

A. Extended Related Work

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., 2020a). 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., 2020b) (Equation (2)). 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) (Equation (3)).

Prior works propose different feature maps ϕ to improve linear attention modeling quality. (Katharopoulos et al., 2020a) 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., 2022a) 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 and 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 and Soricut, 2021). TransNormer (Qin et al., 2022b) apply normalizations such as LayerNorm (Ba et al., 2016) or RMSNorm (Zhang and 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 and 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 Section 3.2. We also study a broader set of architectures in this work beyond gated convolutions.

Gu and 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., 2020b; Gu and 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., 2020a; 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 and 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 and 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 and 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., 2020a) 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.

B. 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 the Section 6. Each kernel operates over 16×16 tiles of data, where dimension 16 is motivated by the matrix multiplication sizes computed by GPU tensor cores.

B.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., 2020a; Vyas et al., 2020). [5]. The listing below shows the baseline implementation for reference (where line 76-77 can be computed using pure PyTorch or the Fast Transformers kernel) (Zhang et al., 2024).

Micro Benchmark To complement the end-to-end architecture benchmarks in Section 6, we provide micro benchmark results for only the linear attention forward pass in Figure 5.

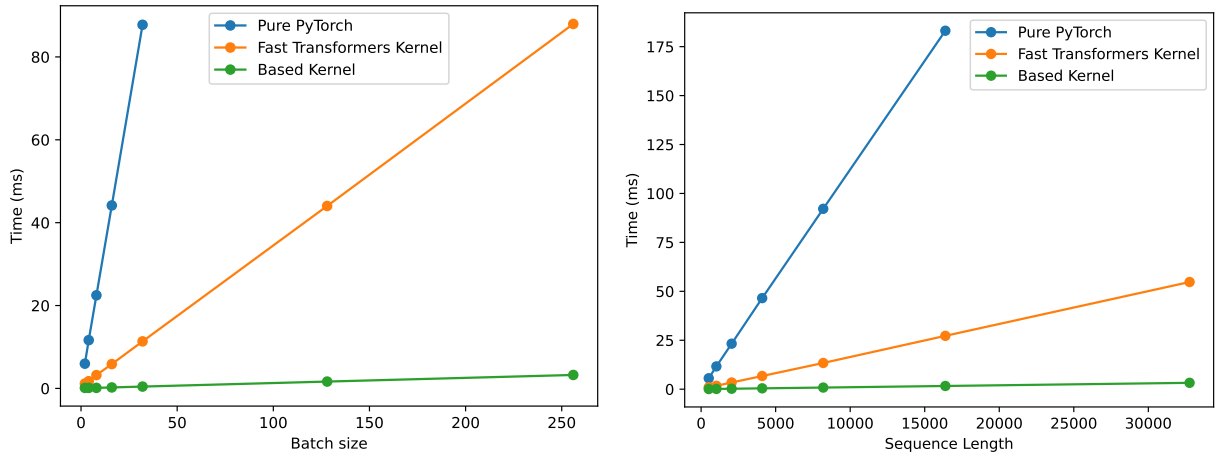

Figure 5. Time (ms) for different ways of computing the Taylor linear attention forward pass — using Pure PyTorch (shown in the Listing and introduced in (Zhang et al., 2024)), Fast Transformers kernel (as indicated in the listing) (Vyas et al., 2020; Katharopoulos et al., 2020b), or our BASED kernel (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.

```
from einops import rearrange
import torch
from torch import nn

class TaylorExp(nn.Module):
    """
    Feature map to compute 2nd-order Taylor approx. of exp(q^T k / sqrt(d))
    """

    def __init__(self, input_dim, head_dim_idx, temp=None, eps=1e-12):
        super().__init__()

        self.input_dim = input_dim
        self.head_dim_idx = head_dim_idx
        self.temp = 1.0 if temp is None else temp
```

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

```
        self.eps = eps

        self.r2 = math.sqrt(2)
        self.rd = math.sqrt(self.input_dim)
        self.rrd = math.sqrt(self.rd)

    def forward(self, x: torch.Tensor):
        # Get 2nd-order terms (rearrange(x * x), '... m n -> ... (m n)')
        x2 = (x.unsqueeze(-1) * x.unsqueeze(-2)).flatten(start_dim=-2) / self.r2
        term1 = torch.ones(x[..., :1].shape).to(x.device)
        term2 = x / self.rrd
        term3 = x2 / self.rd
        terms = [term1, term2, term3]
        return torch.cat(t for t in terms), dim=self.head_dim_idx)


class TaylorLinAttn(nn.Module):
    def __init__(self):
        super().__init__()
        self.d_model = d_model
        self.feature_dim = 16
        self.num_heads = 16
        self.num_key_value_heads = 16
        self.head_dim = self.d_model // self.num_key_value_heads
        self.eps = 1e-12

        feature_map_kwargs = {
            "input_dim": self.feature_dim,
            "head_dim_idx": -1,
            "eps": 1e-12,
        }
        self.feature_map = TaylorExp(**feature_map_kwargs)
        self.proj_q = nn.Linear(
            self.d_model, self.feature_dim * self.num_heads, bias=False
        )
        self.proj_k = nn.Linear(
            self.d_model, self.feature_dim * self.num_heads, bias=False
        )
        self.proj_v = nn.Linear(
            self.d_model, self.num_key_value_heads * self.head_dim, bias=False
        )
        self.proj_o = nn.Linear(
            self.num_heads * self.head_dim, self.d_model, bias=False
        )

    def forward(self, hidden_states: torch.Tensor, *args, **kwargs):
        b, l, _ = hidden_states.size()
        q = self.proj_q(hidden_states)
        k = self.proj_k(hidden_states)
        v = self.proj_v(hidden_states)
        q = q.view(b, l, self.num_heads, self.feature_dim).transpose(1, 2)
        k = k.view(b, l, self.num_key_value_heads, self.feature_dim).transpose(1, 2)
        v = v.view(b, l, self.num_key_value_heads, self.head_dim).transpose(1, 2)

        # Linear attention
        q, k = self.feature_map(q), self.feature_map(k)
        q, k, v = q.unsqueeze(-2), k.unsqueeze(-2), v.unsqueeze(-1)

        # Compute attention causal (alternatively use the Fast Transformers kernel)
        num = (q * (k * v).cumsum(dim=2)).sum(dim=-1)
        denom = (q * k.cumsum(dim=2)).sum(dim=-1) + self.eps
        y = (num / denom)

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

```
        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:

$$\boldsymbol{o}_i = \sum_{j=1}^{i} \frac{\phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j)\boldsymbol{v}_j}{\phi(\boldsymbol{q}_i)\sum_{j=1}^{i}\phi(\boldsymbol{k}_j)} = \frac{\phi(\boldsymbol{q}_i)\sum_{j=1}^{i}\left(\phi(\boldsymbol{k}_j)^\top \boldsymbol{v}_j\right)}{\phi(\boldsymbol{q}_i)\sum_{j=1}^{i}\phi(\boldsymbol{k}_j)} \tag{5}$$

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 $\phi : \mathbb{R}^d \to \mathbb{R}^{d^2}$ such that

$$\phi(\boldsymbol{q}_i)^\top \phi(\boldsymbol{k}_j) = 1 + \boldsymbol{q}_i^\top \boldsymbol{k}_j + \frac{(\boldsymbol{q}_i^\top \boldsymbol{k}_j)^2}{2} \tag{6}$$

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 $\mathrm{q_a}, \mathrm{q_b}, \mathrm{k_a}, \mathrm{k_b}, \mathrm{q_{frag}}, \mathrm{k_{frag}}, \mathrm{v_{frag}}$ in register to hold chunks or *tiles* of the q, k, v inputs. We initialize data structures $A0, A1, A2$ in SRAM and $\mathrm{a0_{frag}}$, $\mathrm{a1_{accum}}$, $\mathrm{qA2_{accum}}$ 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 $\mathrm{n_{blocks}}$, where in each loop from 1 to $\mathrm{n_{blocks}}$, 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 $\mathrm{n_{tiles}} = 128$ and $\mathrm{n_{blocks}} = 16$. 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.

Overall approach Our overall approach is to compute $\boldsymbol{o_i}$ by splitting the $\boldsymbol{q}, \boldsymbol{k}, \boldsymbol{v}$ multiplications as such:

$$\boldsymbol{y_i} = \mathrm{Causal}(\boldsymbol{q_i^T k_i})\boldsymbol{v_i} + \boldsymbol{q_i}\sum_{j=0}^{i-1}(\boldsymbol{k_j v_j})$$

where the first term uses the quadratic attention view and requires applying causal masking. Imagining the square attention matrix, we refer to the first term as computing the interactions **on the diagonal**. The second term uses the linear view and its causality has already been handled. We refer to this term as **off-diagonal**.

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 (6)). 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 (i.e., a *quadratic attention* view). 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 (5)). 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. $\mathrm{q_{frag}}$). To accomplish

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^Tv
  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_aA2_0 and $q_bA2_1 \rightarrow \mathrm{qA2_{accum}}$
    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 for
end for
End. Store y. Optionally store $A0$, $A1$, $A2$ (comprising the "KV state") for generation. ▷ SRAM to HBM

this, in Algorithm 1, we multiply ${\mathrm{k}_{\mathrm{frag}}}^T$ and $\mathrm{v}_{\mathrm{frag}}$ 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 $\mathrm{q}_{\mathrm{frag}}$ in register with this KV state to update the final output for the 1st up until the current $\mathrm{n}_{\mathrm{blocks}}$. Note that we maintain the running KV state corresponding to the 1st order term in $A1$ shared memory for the next iteration along $\mathrm{n}_{\mathrm{blocks}}$.

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 ($\mathcal{O}(d^2D)$ 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 (5)), 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.

B.2. Next Token Prediction

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

B.2.1. Taylor linear attention recurrent update

During next token prediction, an important consideration is how to efficiently update the recurrent state $KV_t \in \mathbb{R}^{BHDd}$ at timestep t. The expensive operation during next token prediction is computing the outer product between projected hidden states $k_{t+1} \in \mathbb{R}^{BHD}$ and $v_{t+1} \in \mathbb{R}^{BHd}$. The outer product requires $\mathcal{O}(BHDd)$ computation and space, and the result is summed with KV_t to produce KV_{t+1}. We provide an IO-aware algorithm for the state updates in Algorithm 2. This algorithm incurs $\mathcal{O}(BHD + BHd)$ bytes of HBM to SRAM data movement (to load the q, k, v projections).

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.

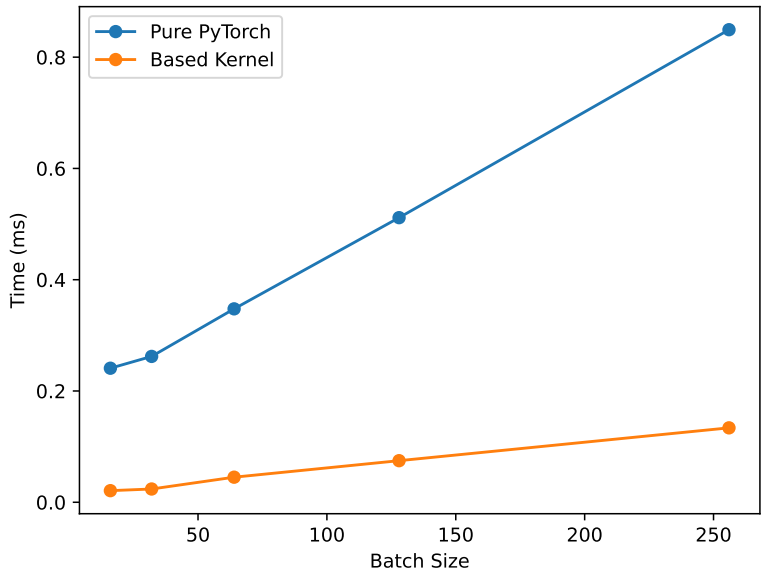

Figure 6. Time (ms) for computing the Taylor linear attention recurrent update using Pure PyTorch (shown in the Listing and introduced in (Zhang et al., 2024)) vs. our BASED kernel (Algorithm 2). Benchmarking uses 16 feature dimension, 16 heads, 64 head dimension, and focuses on the *numerator* of the linear attention. Each point represents the median across 10 iterations is measured on a single NVIDIA H100 GPU.

```
from einops import rearrange
import torch
from torch import nn

def step(self, kv_state: torch.Tensor, k_state: torch.Tensor, q: torch.Tensor, k: torch.
    Tensor, v: torch.Tensor):
        """
        Compute linear attention with recurrent view
        -> 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.
        """
        b, h, l, d = q.shape
        assert l == 1, f'q.shape is {q.shape} but should be ({b}, {h}, 1, {d})'
        # Expand dims for broadcasting to compute linear attention
        q, k, v = q.unsqueeze(-2), k.unsqueeze(-2), v.unsqueeze(-1)

        kv_state += k[:, :, -1:] * v[:, :, -1:]
        k_state  += k[:, :, -1:]
```

```
        # Compute linear attention
        num = (q * kv_state).sum(dim=-1)
        y = num / ((q * k_state).sum(dim=-1) + self.eps)

        y = rearrange(y, 'b h l d -> b l (h d)').to(q.dtype)
        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 $\text{batch} \times \text{heads}$ parallel computations, with $\text{n}_{\text{warps}} = 8$ warps per block.
Within a block:
Define $\text{n}_{\text{threads}} = \text{n}_{\text{warps}} \times 32$ ▷ Assuming 32 threads per warp
Define $\text{buffer}_{\text{size}} = \text{n}_{\text{warps}} \times 8 \times d$
Define $\text{total}_{\text{batches}} = \frac{D}{\text{n}_{\text{warps}}\times 8}$ ▷ E.g. $\text{total}_{\text{batches}} = 5$ if $D = 320$; For k, $\frac{320}{5} = 64$ values per batch
Define $\text{tic} = 0, \text{toc} = 1$

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 $2 \times \text{buffer}_{\text{size}}$) for storing blocks of kv_{state}
Create SRAM buffer o (Size d) for output.
Create SRAM buffer A (Size $\text{n}_{\text{warps}} \times d$) for intermediate computation

Create register buffer v_{reg} (Size 2) to store V data
Create register 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 $B_{kvs}[\text{tic}] \leftarrow \text{kv}_{\text{state}}$ ▷ Load $(1 \times 64) \times 64$ of the $(\text{total}_{\text{batches}} \times 64) \times 64$ 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 $\text{v}_{\text{reg}}[m] \leftarrow v[j]$ ▷ SRAM to Register; Now $v[j]$ is stored in thread $j \mod 32$
end for

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

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

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

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

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 $A[\text{warpid}][j] \leftarrow \text{A}_{\text{reg}}[m]$ ▷ Register to SRAM
end for

for Thread $j; j < d; j+= \text{n}_{\text{threads}}$ **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 $w \in [0..\text{n}_{\text{warps}}]$ **do**
    Sum the $n_j+= A[w][j]$ across ▷ Need to combine results across warps
  end for
  Store $o[j] \leftarrow n_j$
end for
Write output o ▷ SRAM to HBM

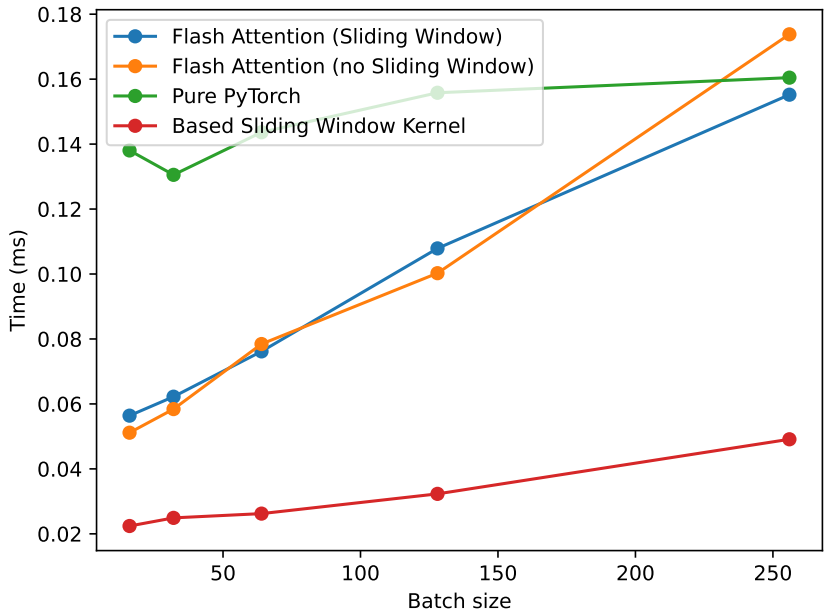

Figure 7. Time (ms) for different ways of computing sliding window attention next token prediction — using PyTorch, Flash Attention (which supports a sliding window function), or our inference kernel. Each point represents the median across query tokens at different token positions in the generation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

B.2.2. Sliding window attention

Next we motivate the choice of window size for TCWindow. In contrast to sliding-window style models such as the popular Mistral models, which use large window sizes $w = 4096$ (Jiang et al., 2023), Based chooses a window size based on hardware specifications. GPU tensor cores operate on 16×16 tiles. Large GEMMs are compute bound (for e.g. in long-context attention). But, we need sufficient occupancy to hide the latency of the tensor core units. Figure 1 (Right) shows 64×64 dimension matrix multiplications are approximately the same latency as 16×16. Based sets w to use 64×64 tiles (Figure 1). To distinguish from prior sliding windows, we refer to this approach as TCWindow. We use the Flash Attention sliding window implementation during training (Dao, 2023) and in Appendix B Algorithm 3, we provide an IO-aware algorithm of TCWindow for next token prediction. The naïve sliding window approach reads and writes $\mathcal{O}(BHwd)$ bytes between SRAM and HBM between each step of the attention computation. Our approach fuses computation in thread registers to improve upon the baselines.

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. 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 $\text{batch} \times \text{heads}$ parallel computations, with $\text{n}_{\text{warps}} = 4$ warps per block.
Within a block:
Define tile size T ▷ $T = 16$ in BASED
Define $\text{n}_{\text{threads}} = \text{n}_{\text{warps}} \times 32$ ▷ 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 q_{reg}, k_{reg}, v_{reg} to hold data during fused computation in-register.
Create register fragments w_{reg} (size $1 \times 4T$) and wv_{reg} (size $4T \times 1$) to store intermediate computation in-register.
Create register fragment o_{reg} (size $4T \times 1$) to store output in-register.

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

Initialize w_{reg} to zero
$\text{w}_{\text{reg}} \leftarrow \text{q}_{\text{reg}}\text{k}_{\text{reg}}$ ▷ Matrix-vector (GEMV) multiplication

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

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

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

C. Extended Architecture Details

In this section, we describe two additional architectural details for that can enable small improvements in language model perplexity. We emphasize, however, that the combination of Taylor linear attention and TCWINDOW layers alone is sufficient to come within 0.1 perplexity points of our best models using these additional components (Table 6).

Convolution. We find that replacing some of the linear attention and TCWINDOW layers with gated convolution layers enables small improvements in language modeling performance. A gated convolution layer uses a combination of gating (Hadamard product, elementwise product) and convolution operations. In , we use `BaseConv` layers (Arora et al., 2023a) with short convolutions and a SilU non-linearity (Hendrycks and Gimpel, 2023). By keeping the convolutions short (*e.g.* width 3), we keep the recurrent state size for these layers low and improve throughput. The projections expand the dimensionality by a factor $c = 4$.

$$\boldsymbol{y} := (\underbrace{\boldsymbol{u} \cdot \boldsymbol{W}_1 + \boldsymbol{b}_1}_{\textbf{Linear Projection}} \odot \sigma \underbrace{\boldsymbol{h} * \boldsymbol{u} \cdot \boldsymbol{W}_2 + \boldsymbol{b}_2}_{\textbf{Convolution}}) \cdot \boldsymbol{W}_3 + \boldsymbol{b}_3 \tag{7}$$

where $\boldsymbol{u} \in \mathbb{R}^{N \times d}$ is a projected input, $h \in \mathbb{R}^{N \times cd}$ is a learned filter, $\odot$ is the Hadamard product, and $\boldsymbol{W}_1, \boldsymbol{W}_2 \in \mathbb{R}^{d \times cd}$, $\boldsymbol{W}_3 \in \mathbb{R}^{cd \times d}$, $\boldsymbol{b}_1, \boldsymbol{b}_2 \in \mathbb{R}^{cd}$, and $\boldsymbol{b}_3, \in \mathbb{R}^{d}$ define weights and biases of three linear projections.

Decay. Recent recurrent architectures include the use of *decay* terms, implemented in a variety of ways (Gu et al., 2021; Sun et al., 2023; Gu and Dao, 2023; Yang et al., 2023). As intuition, decay terms control how much a token should attend to "recent" tokens vs. "early" tokens in the sequence. Prior work falls in two categories: using input-independent (Gu et al., 2021; Sun et al., 2023, inter alia.) or input-dependent (Gu and Dao, 2023; Yang et al., 2023) decay rates. The latter offers improved quality, but requires the use of a parallel scan during sequence processing (Gu and Dao, 2023).

Instead, we explore a coarser input-dependent decay technique for the linear attention layer, avoiding the parallel scan. We first use a unique decay rate *per head*, fixed across all inputs. We introduce a linear projection that takes in the inputs $\in \mathbb{R}^{Nxd}$ and projects to $\mathbb{R}^{N \times h}$, where N is the sequence length, d is the model dimension, and H is the number of heads. We use the result of this projection to scale the attention combination across heads.

In our main experiments Table 1, we use no decay when training the models to 50b and 30b tokens. We observe that decay can help small in our Table 6 ablations, but removing the decay does not affect the overall trends for BASED relative to other architectures.

D. Extended Results

D.1. Extended empirical study of memory-recall tradeoff

In Figure 8, we provide additional experimental results using the setup described in Section 3.1. The results in Figure 8 include additional efficient architectures beyond those in Figure 3 and Figure 2. Specifically we include NystromFormer (Xiong et al., 2021), BigBird (Zaheer et al., 2020), and ScatterBrain (Chen et al., 2021a).

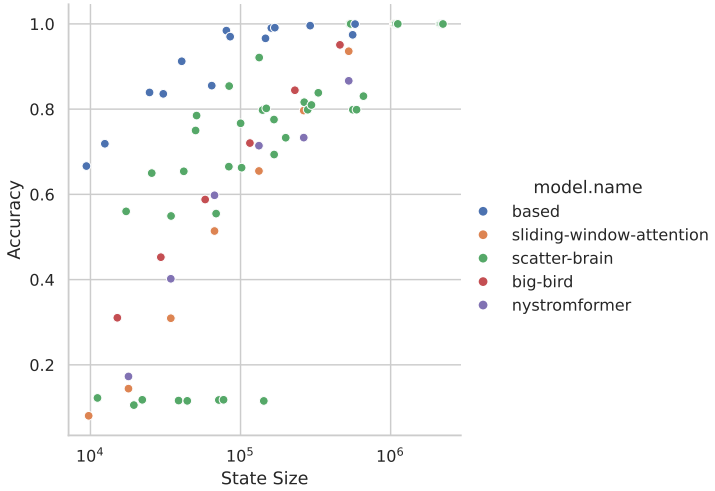

Figure 8. **Extended Throughput (memory) - recall tradeoff.** x-axis shows state size (bytes) during generation; y-axis shows accuracy on the MQAR recall task (Arora et al., 2023a). For each architecture, we train several models varying hyperparameters that affect the recurrent state size (*e.g.* model dimension). The plot shows a fundamental tradeoff between the recurrent state size and recall capacity that applies to broad class of models.

D.2. Downstream Language Results

To further evaluate BASED's performance in language modeling, we evaluate the PILE-pretrained models on several downstream tasks that test general natural language understanding.

Architecture	Params/Tokens	LAMBADA Ppl. ↓	LAMBADA Acc. ↑	HellaSwag Acc. Norm. ↑	PIQA Acc ↑	Arc-E Acc ↑	Arc-C Acc. Norm. ↑	WinoGrande Acc. ↑	Average Acc. ↑
Transformer++ (LLaMa)	1.33b/10b	11.12	49.10	39.29	66.16	51.68	26.19	53.43	47.64
BASED	1.35b/10b	12.35	46.96	39.11	66.32	50.72	26.54	50.43	46.68
Mamba	1.32b/10b	13.11	46.13	39.41	66.38	52.36	25.94	50.83	46.84
Transformer++ (LLaMa)	1.33b/50b	7.38	57.50	49.62	70.46	57.58	27.99	56.83	53.33
BASED	1.35b/50b	6.96	57.85	50.79	71.65	58.84	28.75	55.80	53.81
Mamba	1.32b/50b	7.19	57.56	50.94	71.87	59.39	28.41	53.83	53.50
Transformer++ (LLaMa)	360m/10b	18.39	42.52	33.48	63.98	46.04	24.49	53.99	44.08
Transformer (Pythia)	356m/10b	25.17	37.16	31.32	63.76	44.82	23.8	51.54	42.08
BASED	363m/10b	21.80	38.66	33.43	64.42	45.79	24.66	51.22	43.03
Mamba	358m/10b	20.23	39.65	33.63	65.02	47.01	25.00	50.75	43.51
H3	362m/10b	57.59	23.58	30.62	63.11	45.20	23.29	50.28	39.35
Transformer++ (LLaMa)	360m/30b	15.79	44.44	36.90	66.05	48.27	20.56	52.25	44.75
BASED	363m/30b	14.43	45.20	37.41	67.46	49.45	21.42	51.22	45.36
Mamba	358m/30b	14.27	45.06	38.02	66.38	50.55	20.01	51.70	45.62

Table 2. **Downstream evaluation of pre-trained language models.** The same set of models as in Table 1, all were trained on the same data drawn from the Pile (Gao et al., 2020), evaluated zero-shot using the default LM-Eval Harness settings from EleutherAI (Gao et al., 2023). These averages are computed across the 6 tasks, excluding LAMBADA perplexity. These averages are included in Table 1.

LM-Eval Harness Standard Tasks We use the same protocol as (Gu and Dao, 2023; Yang et al., 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)

Model	Shots	BoolQ	CB		COPA	MultiRC	ReCoRD		RTE	WiC	WSC	Avg
		Acc. ↑	Acc. ↑	F1 ↑	Acc. ↑	Acc. ↑	F1 ↑	EM ↑	Acc. ↑	Acc. ↑	Acc. ↑	
Based (363m/10b)	0	59.0	41.1	19.4	69.0	54.9	14.5	14.0	52.0	50.0	36.5	45.7
	1	57.5	37.5	26.8	68.0	52.5	19.9	19.2	47.7	50.9	49.0	47.2
	5	56.6	44.6	28.9	73.0	53.6	24.9	24.1	48.7	51.1	39.4	48.0
Transformer++ (360m/10b)	0	57.3	41.1	21.3	67.0	57.0	16.6	16.1	53.8	50.0	37.5	46.3
	1	54.2	39.3	25.3	69.0	51.5	22.2	21.6	50.9	47.0	55.8	47.8
	5	50.7	58.9	49.9	64.0	46.9	24.2	23.6	47.3	52.2	51.9	48.9
Mamba (358m/10b)	0	57.5	35.7	24.4	71.0	57.2	18.8	18.3	52.4	50.0	36.5	46.6
	1	51.1	39.3	27.4	71.0	52.9	21.6	21.0	46.6	46.2	52.9	46.9
	5	41.1	37.5	23.6	69.0	49.2	20.4	19.9	48.4	51.7	51.9	45.2

Table 3. **Few-shot downstream evaluation on SuperGLUE 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 on the SuperGLUE benchmark (Wang et al., 2019) using the LM eval harness by EleutherAI (Gao et al., 2023). When computing the average, we first average the metrics by task and then average across tasks.

		HG38 PPL ↓		
Model	Params	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 4. **DNA modeling performance on the HG38 dataset.** All models are pretrained from scratch for 10Bn tokens at $N = 1$k, 4k, and 8k sequence lengths respectively. We report results after hyperparameter sweeping the learning rate for each architecture.

- 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 and Dao, 2023). We report results in Table 2. For both 360 million and 1.3 billion parameter models, BASED performs competitively with recent and state-of-the art architectures, including Mamba and Transformer++ (LLaMa).

SuperGLUE Fewshot Results In order to understand in-context-learning performance, we next perform few-shot evaluations on the SuperGLUE benchmark (Wang et al., 2019) for BASED, Mamba and Transformer++ in Table 3. Each model was evaluated on all tasks using under 0 shot (*i.e.*, number of in-context examples), 1 shot and 5 shot prompting, respectively. Transformer++ and BASED both see monotonic improvement from increasing the number of shots. For Mamba, however, albeit getting a slight improvement from 0-shot to 1-shot, it performs worse on 5-shot than even on 0-shot. This result suggests that the limited recall ability observed in Mamba could also impact few-shot abilities.

D.3. DNA Modeling

Towards understanding the capability of BASED beyond natural English language, we next evaluate each architecture on its ability to model DNA sequences.

Pretraining In Table 4, we evaluate architectures on the HG38 (human genome) benchmark at $1k$, $4k$, and $8k$ sequence lengths used in prior architecture evaluations (Nguyen et al., 2023; Gu and Dao, 2023). The DNA tasks uses a byte-level tokenizer wherein the vocabulary consists of characters corresponding to the nucleotide bases. We find BASED is competitive with state-of-the-art architectures across evaluated sequence lengths.

Downstream DNA Classification We further evaluate how different architectures compare for DNA modeling. We take the pretrained models described and evaluate them on DNA sequence classification using a popular benchmark (GenomicBenchmarks) (Gresova et al., 2022) in Table 5. We find similar performance across tasks, indicating that prior matching in quality during pretraining transfers to downstream classification. For reference, we also include results from (Nguyen et al., 2023). Although not directly comparable to due differences in tokenization, the evaluations suggest BASED

Dataset	Enhancer Cohn	Enhancer Ens	Human Reg.	Non-TATA Promoters	Human OCR Ens.
CNN	69.5	68.9	93.3	84.6	68.0
DNABERT	74.0	85.7	88.1	85.6	75.1
GPT	70.5	83.5	91.5	87.7	73.0
HyenaDNA	74.2	89.2	**93.8**	96.6	**80.9**
Transformer++	73.4	**89.5**	89.9	94.4	79.5
Mamba	73.0	-	-	96.6	-
Based	**74.6**	**89.5**	89.5	**96.8**	79.0

Table 5. **Downstream evaluation of pre-trained DNA models on GenomicsBenchmarks (Gresova et al., 2022).** We report top-1 classification accuracy (%) with pretrained models (Transformer++, Mamba, BASED) along with prior reported results in (Nguyen et al., 2023). We find the similar quality-matching in pretraining transfers to downstream tasks. Modern architectures are also able to achieve state-of-the-art results on the classification tasks.

can perform strongly on different modalities, and that recent sequence modeling architectures are also able to outperform or compete with prior state-of-the-art on evaluated DNA tasks.

D.4. BASED Quality Ablations

Our objective with BASED is to measure the throughput and recall of the simplest possible linear attention model that achieves strong performance. Therefore, we ablate the key design decisions — choice of feature map, feature dimension for the Taylor map, use of sliding window and convolutions — to understand their contributions to the quality of BASED. We ablate using the Pile dataset (Gao et al., 2020) with the same number of tokens and data ordering as the prior experiments.

In **feature map ablations**, we consider the CosFormer (Qin et al., 2022a) and Performers (Choromanski et al., 2020) feature maps, which have been demonstrated as strong choices in prior work (Zhang et al., 2024). We also include a baseline that expands the state size using learned projections and applies CosFormer towards comparing to the larger state size of the Taylor map. For these baselines, we keep the rest of the BASED architecture the same (*i.e.* in the number of linear attention layers and hybridization with sliding window and gated convolution layers). We observe that with the larger state size, CosFormer quality is increasingly competitive with the Taylor map. We note that expanding the state size requires increasing the model's overall parameter count (due to the learned projections) for CosFormer, in contrast to the Taylor map.

Next, we ablate the **feature dimension**, holding the feature map fixed to the Taylor map. We find larger feature dimension improves quality, with diminishing returns going from 24 to 32 dimension. Note that feature dimension $\sqrt{1024} = 32$, where 1024 is the attention model dimension at the 360 parameter scale in our experiments.

Next, the ablations show that eliminating the **convolutions** and/or the **sliding window attention** degrades quality. We observe that adding *either* convolutions or sliding window helps on the associative recall slice relative to *neither* (e.g. 2.29 AR Ppl. on the Pile with *neither* vs. 2.09 or 2.11 with sliding window *or* convolutions.). Increasing the window size from 0 to 64 vs. 64 to 128 (also an efficient design point in Figure 1, left) continues to help quality, but with marginal improvements.

Finally, we ablate the use of the input-dependent decay strategy introduced in Appendix C. In our main results Table 1, we use **no input-dependent decay** whatsoever when training to 30b and 50b tokens for the 360m and 1.3b parameter models respectively. At 10b tokens, we use the decay strategy and provide ablations without decay in Table 6. We find that the decay can provide a small boost in quality, but removing the decay does not affect the overall trends.

	Hyperparameters				Language Modeling (Pile)			Info. Extraction		QA
Feat. Map	Feat. Dim.	Sliding	Convs.	Decay	All Ppl. ↓	AR Ppl. ↓	Other Ppl. ↓	SWDE Acc. ↑	FDA Acc. ↑	SQUAD Acc. ↑
Taylor Exp. (2nd)	16 (153)	✓(64)	✓	✓	**8.65**	**2.07**	**9.64**	**29.16**	**11.71**	25.07
Performer	16 (16)	✓(64)	✓	✓	9.08	8.53	11.62	8.10	0.36	7.47
CosFormer	16 (32)	✓(64)	✓	✓	9.03	2.42	9.98	19.35	7.71	24.63
CosFormer	64 (128)	✓(64)	✓	✓	8.82	2.18	9.80	25.47	9.07	**27.85**
Taylor Exp. (2nd)	32 (561)	✓(64)	✓	✓	**8.56**	**2.00**	**9.57**	**37.62**	12.89	**26.74**
Taylor Exp. (2nd)	24 (325)	✓(64)	✓	✓	8.58	2.02	9.58	34.38	**20.87**	24.77
Taylor Exp. (2nd)	16 (153)	✓(64)	✓	✓	8.65	2.07	9.64	29.16	11.71	25.07
Taylor Exp. (2nd)	8 (45)	✓(64)	✓	✓	8.77	2.18	9.75	23.40	12.79	22.35
Taylor Exp. (2nd)	16 (153)	✓(64)	✓	✓	**8.65**	2.07	**9.64**	**29.16**	11.71	25.07
Taylor Exp. (2nd)	16 (153)	✓(64)	✓	✗	**8.65**	**2.04**	9.66	22.95	**12.34**	**27.45**
Taylor Exp. (2nd)	16 (153)	✗	✓	✓	8.91	2.11	9.94	28.62	10.16	24.5
Taylor Exp. (2nd)	16 (153)	✓(64)	✗	✓	8.74	2.09	9.74	24.66	2.36	18.87
Taylor Exp. (2nd)	24 (325)	✗	✗	✓	9.49	2.29	10.58	19.62	8.71	11.33
Taylor Exp. (2nd)	16 (153)	✓(128)	✓	✓	**8.61**	**2.06**	**9.60**	**32.13**	**14.39**	**31.84**
Taylor Exp. (2nd)	16 (153)	✓(64)	✓	✓	8.65	2.07	9.64	29.16	11.71	25.07

Table 6. **Ablations.** All models are 362M param variants of the BASED architecture described in Section 4, trained to 10 billion tokens on the Pile. We ablate the hyperparameters central to the design of BASED: (1) the choice of feature map ϕ (see Section 4.1), (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 4.1), (3) the use of local sequence mixers (sliding window attention and short convolutions), and (4) the data-dependent decay defined in Section 4.

E. Experimental Details

E.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 pre-training 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 7. 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 B (Vyas et al., 2020; Katharopoulos et al., 2020b). We use no input-dependent decay whatsoever when training the models to 30b and 50b tokens at 360m and 1.3b parameters respectively.
- **Transformer++ (Touvron et al., 2023)** We refer to the modern Llama architecture with Rotary encodings, RMSNorm and SwiGLU as Transformer++, following prior work (Gu and Dao, 2023; Yang et al., 2023). We train using the the specifications in Table 8 using the Flash Attention training code provided here: `https://github.com/Dao-AILab/flash-attention/tree/main` (Dao, 2023).
- **Mamba (Gu and Dao, 2023)** We train using the specifications in Table 9, where the parameters are sourced from the Appendix of (Gu and 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 10, 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 11. 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 12 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 13. 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.

E.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 3.1. 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 and 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.

$$\text{sizeof}(\boldsymbol{s}_i) = (d+1) \times \tilde{d} \tag{8}$$

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:

$$\text{sizeof}(\boldsymbol{s}_i) = (d+1) \times (1 + \frac{3d'}{2} + \frac{d'^2}{2}) \tag{9}$$

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.

$$\text{sizeof}(\boldsymbol{s}_i) = 2 \times d \times N \tag{10}$$

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.

$$\text{sizeof}(\boldsymbol{s}_i) = 2 \times d \times \min(N, k_{\text{sliding}}) \tag{11}$$

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.

$$\text{sizeof}(\boldsymbol{s}_i) = 2 \times d \times d_{\text{state}} \tag{12}$$

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}.

$$\text{sizeof}(\boldsymbol{s}_i) = d \times d_{\text{state}} \tag{13}$$

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.

$$\text{sizeof}(\boldsymbol{s}_i) = d \times N \tag{14}$$

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

E.3. Language Model Evaluation

In this section, we provide details on each of the evaluations (columns) reported in Tables 1 and 6.

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**

F. Theoretical Results

F.1. Introduction

Our focus in this section will be on the theoretical results of the paper. Specifically, we will show the equivalence of models `Based` and `Mamba` (Gu and Dao, 2023) with `BaseConv`, 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 pair-wise Hadamard product of columns of $\mathbf{M}$ as $\boldsymbol{M} \circ \boldsymbol{M} \in \mathbb{R}^{n\times m^2}$, where

$$\begin{aligned}(\boldsymbol{M} \circ \boldsymbol{M})[:,i] &:= \mathbf{M}[:,j] \odot \mathbf{M}[:,k] \quad \text{for} \quad i \in [m^2], \\ j &= \left\lfloor \frac{i-1}{m} \right\rfloor + 1, \quad k = (i-1) \mod m + 1.\end{aligned} \tag{15}$$

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 $X \triangleq \{x_1, x_2, \dots, x_n\}$ 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 F.1. 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.

F.2. The Models

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

F.2.1. Based

The `Based` model combines two layer types: `BaseConv` and `LinearAttention` defined below.

Definition F.2 (`BaseConv` (Arora et al., 2023a))**.** Given an input sequence $\boldsymbol{u} \in \mathbb{R}^{N\times d}$, where N is the sequence length and d is the model dimension, a learned weight matrix $\boldsymbol{W}^B \in \mathbb{R}^{d\times d}$ and biases $\boldsymbol{B}^B, \boldsymbol{B}^K \in \mathbb{R}^{N\times d}$ and a matrix of convolution filters $\boldsymbol{K} \in \mathbb{R}^{N\times d}$, a `BaseConv`layer computes the following:

$$\boldsymbol{z}^{\texttt{BaseConv}} := (\boldsymbol{u}\boldsymbol{W}^B + \boldsymbol{B}^B) \odot \boldsymbol{K} * \boldsymbol{u} + \boldsymbol{B}^K \in \mathbb{R}^{N\times d}, \tag{16}$$

where the convolutions are applied across the input length N.

Definition F.3 (`LinearAttention` (Katharopoulos et al., 2020b))**.** Given an input sequence $\boldsymbol{u} \in \mathbb{R}^{N\times d}$, where N is the sequence length and d is the model dimension, a set of linear projections[6] $\texttt{Projection}_q, \texttt{Projection}_k \in \mathbb{R}^{d\times d'}, \texttt{Projection}_v \in \mathbb{R}^{d\times d}$, where d' is the feature dimension, the `LinearAttention` layer computes the following:

$$\boldsymbol{z}^{\texttt{LinearAttention}} := \overline{\boldsymbol{Q}}\,\overline{\boldsymbol{K}}^\top \boldsymbol{V} \in \mathbb{R}^{N\times d}, \tag{17}$$

[6]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 $\boldsymbol{B} \in \mathbb{R}^{m\times n}$.

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

$$\begin{aligned}\overline{\boldsymbol{Q}} &= [\boldsymbol{1}, \boldsymbol{Q}, \boldsymbol{Q} \circ \boldsymbol{Q}] \in \mathbb{R}^{N \times (1+d'+d'^2)},\\ \overline{\boldsymbol{K}} &= [\boldsymbol{1}, \boldsymbol{Q}, \boldsymbol{K} \circ \boldsymbol{K}] \in \mathbb{R}^{N \times (1+d'+d'^2)}.\end{aligned}$$

F.2.2. MAMBA

We now introduce the `Mamba` model from (Gu and Dao, 2023).

Definition F.4 (`Mamba` (Gu and Dao, 2023))**.** Given an input sequence $\boldsymbol{u} \in \mathbb{R}^{N \times d}$, where N is the sequence length and d is the model dimension, the `Mamba` layer computes the following:

$$\boldsymbol{z}^{\texttt{Mamba}} := \texttt{SSM}(\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C})(\boldsymbol{u}) \in \mathbb{R}^{N \times d}, \tag{18}$$

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

$$\begin{aligned}\overline{\boldsymbol{A}} &:= \exp \Delta \boldsymbol{A},\\ \overline{\boldsymbol{B}} &:= \Delta \boldsymbol{A}^{-1}(\exp \Delta \boldsymbol{A} - \boldsymbol{I}) \cdot \Delta \boldsymbol{B},\\ &= \boldsymbol{A}^{-1}(\exp \Delta \boldsymbol{A} - \boldsymbol{I}) \cdot \boldsymbol{B},\end{aligned} \tag{19}$$

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

$$\begin{aligned}\boldsymbol{B} &:= \texttt{Linear}_{N \times \overline{d}}(\boldsymbol{u}) \in \mathbb{R}^{\overline{d}},\\ \boldsymbol{C} &:= \texttt{Linear}_{N \times \overline{d}}(\boldsymbol{u}) \in \mathbb{R}^{\overline{d}},\\ \Delta &:= \texttt{Linear}_{N \times d}(\boldsymbol{u}) \in \mathbb{R}\end{aligned} \tag{20}$$

for $i \in [N]$. It is important to note here that the parameters $\overline{\boldsymbol{B}}, \boldsymbol{C}, \Delta$ are causal[7] 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 $\overline{\boldsymbol{A}}_i \in \mathbb{R}^{\overline{d} \times \overline{d}}$ is inherited from Δ_i in equation 19 and we denote $\overline{\boldsymbol{B}}[i,:] =: \boldsymbol{B}_i, \overline{\boldsymbol{C}}[i,:] =: \boldsymbol{C}_i$.

Finally, the `SSM` in equation 18 is realized as a linear recurrence. That is, for every $(i,j) \in [N] \times [d]$, we have

$$\begin{aligned}\boldsymbol{h}[i,j] &= \overline{\boldsymbol{A}}_i \boldsymbol{h}[i-1,j] + \overline{\boldsymbol{B}}_i \boldsymbol{u}[i,j]\\ \boldsymbol{z}[i,j] &= \boldsymbol{C}_i^\top \boldsymbol{h}[i,j]\end{aligned} \tag{21}$$

where $\boldsymbol{h}[i,j] \in \mathbb{R}^{\overline{d}}, \boldsymbol{z}[i,j] \in \mathbb{R}$ denote the latent state and the output of the `SSM` in Equation (18), respectively.

F.3. Equivalency to `BaseConv`

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 `BaseConv` 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 17 and equation 18 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 F.5. An $N, L, d, \tilde{N}, \tilde{d}-$ 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

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

3. all the individual gated convolution layers take in $\tilde{N} \times \tilde{d}$ matrices and output $\tilde{N} \times \tilde{d}$ matrices. We refer to the tuple $(\tilde{N}, \tilde{d})$ as the *inner dimension* of the model.

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

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

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

Theorem F.6 ((Arora et al., 2023a), Theorem 4.2)**.** *For any (nd, s, Δ, w)-arithmetic circuit $\mathcal{C}$, there exists an equivalent $N, \Delta', d, \tilde{N}, \tilde{d} - \texttt{BaseConv}$ with $N = n, \Delta' = \mathcal{O}(\Delta \log w)$, $\tilde{N} = \mathcal{O}(w), \tilde{d} = d$ that simulates $\mathcal{C}$.*

Remark F.7. 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 $\boldsymbol{u}[i,j]$.

We now present the results showing equivalency between the models in Appendix F.2 and the `BaseConv` layer in equation 16 using Theorem F.6.

Proposition F.8. *Given an input $\boldsymbol{u} \in \mathbb{R}^{N \times d}$, there exists an equivalent $N, O(\log^2(Nd)), d, O(N(d + d'^2), O(\max(d, d'^2)) - \texttt{BaseConv}$ that computes the output of the `LinearAttention` layer with feature dimension d', cf. Equation (17).*

Proof. For the matrices $\boldsymbol{Q}, \boldsymbol{K} \in \mathbb{R}^{N \times d'}, \boldsymbol{V} \in \mathbb{R}^{N \times d}$ with the corresponding projection matrices $\boldsymbol{W}^Q, \boldsymbol{W}^k \in \mathbb{R}^{d \times d'}, \boldsymbol{W}^V \in \mathbb{R}^{d \times d}$, a single `BaseConv` 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 $\boldsymbol{B}^\ell \equiv \mathbb{1}^{N \times d}$, 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 $(i,j) \in [N] \times [d'^2]$ of $\boldsymbol{Q} \circ \boldsymbol{Q}, \boldsymbol{K} \circ \boldsymbol{K} \in \mathbb{R}^{N \times d'^2}$. From equation 15, observe that each entry of $\boldsymbol{M} \circ \boldsymbol{M}$ can be written as the product of entries from $\boldsymbol{M}$. Hence we have

$$\begin{aligned} (\boldsymbol{Q} \circ \boldsymbol{Q})[i,j] &\equiv \boldsymbol{Q}[i,k] \cdot \boldsymbol{Q}[i,\ell] \\ (\boldsymbol{K} \circ \boldsymbol{K})[i,j] &\equiv \boldsymbol{K}[i,k] \cdot \boldsymbol{K}[i,\ell] \end{aligned} \tag{22}$$

for $k = \left\lfloor \frac{j-1}{d'} \right\rfloor + 1, \quad \ell = (j-1) \mod d' + 1$. 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 $k = \left\lfloor \frac{j-1}{d'} \right\rfloor + 1, \quad \ell = (j-1) \mod d' + 1$ so that $(\boldsymbol{Q} \circ \boldsymbol{Q}) = \boldsymbol{Q}_1 \odot \boldsymbol{Q}_2$, which, *mutatis mutandis*, also applies to $(\boldsymbol{K} \circ \boldsymbol{K})$ 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 `BaseConv` 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 $N, O(1), d, O(N(d + d'^2), O(\max(d, d'^2)) - \texttt{BaseConv}$ model. Moreover, the concatenated matrices $\overline{\boldsymbol{Q}}, \overline{\boldsymbol{K}} \in \mathbb{R}^{N \times (1 + d' + d'^2)}$ then take the addition of the computed components so far which again takes $O(1)$ layers of `BaseConv`.

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

$$\boldsymbol{z}_{i,j}(\boldsymbol{u}) \equiv \sum_{m \in [1 + d' + d'^2], n \in [N]} \overline{\boldsymbol{Q}}[i,m] \cdot \overline{\boldsymbol{K}}[n,m] \cdot \boldsymbol{V}[n,j]. \tag{23}$$

Thus, we can derive the arithmetic circuit that computes $\boldsymbol{z}_{i,j}(\boldsymbol{u})$ by taking in the outputs of the `BaseConv` 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 $\log \lceil Nd \rceil$ depth. Each term inside the sum requires two multiplication gates with depth 2, each of which serve as inputs to the circuit with size Nd computing the sum. Moreover, there are $N \cdot d$ such output gates each of which is computed in parallel resulting in a circuit of size $O(N \cdot d)$, depth $O(\log(Nd))$ and width $O(Nd)$. O Overall, applying Theorem F.6 then results in an equivalent $N, O(\log^2(Nd)), d, O(N(d + d'^2), O(\max(d, d'^2)) - \texttt{BaseConv}$ model that computes $\boldsymbol{z}$. □

F.4. 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 $\boldsymbol{u}[0 \cdots 3N-1] \triangleq \{\boldsymbol{k}_0, \boldsymbol{v}_0, \boldsymbol{q}_0, \ldots, \boldsymbol{k}_{N-1}, \boldsymbol{v}_{N-1}, \boldsymbol{q}_{N-1}\}$ 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 $1 \leq i \leq N-1$, whether there exists $0 \leq j < i$ such that $\boldsymbol{q}_i \equiv \boldsymbol{k}_j$, and if so, output $\boldsymbol{v}_j$.

F.4.1. 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 F.9 ((Jayram et al., 2008))**.** *The one-way randomized communication complexity*[8] *of the index problem for sending an n-length bit string is $\Omega(n)$.*

F.4.2. Lower Bound for Recurrent Models

We now use Theorem F.9 to first provide a lower bound on the number of bits required by the following class of models to solve AR.

Definition F.10 (Recurrent Models)**.** A model $\mathcal{M}$ taking an input $\boldsymbol{u} \in \mathbb{R}^{N \times d}$, 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, $\boldsymbol{Z}_{\mathcal{M}}^i \in \mathbb{R}^{\tilde{d}}$, with $\tilde{d}$ denoting the state size, is determined exclusively by the preceding elements of the input $\boldsymbol{u}[0 \ldots i-1]$. The state $\boldsymbol{Z}_{\mathcal{M}}^i$ 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, $\boldsymbol{Z}_{\mathcal{M}}^i(\boldsymbol{u}) = \phi(\boldsymbol{u}[0 \ldots i-1])$, 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:

$$\boldsymbol{Z}_{\mathcal{M}}^i(\boldsymbol{u}) = f_{\mathcal{M}}^i(\boldsymbol{Z}_{\mathcal{M}}^{i-1}, \boldsymbol{u}[i]), \tag{24}$$

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

Remark F.11. Note that Definition F.10 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 F.12. *Any recurrent model $\mathcal{M}$ (Definition F.10) that solves AR requires $\max_i \boldsymbol{Z}_{\mathcal{M}}^i$ 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:

$$\{j, \boldsymbol{x}_j\}_{j=0}^{N-1}, i. \tag{25}$$

Next, consider the following one-way protocol for solving the index problem using the regressive model $\mathcal{M}$. Alice with their access of $\boldsymbol{x} \in \{0,1\}^N$ generate an input for AR (without the query) as in equation 25. Alice then runs the model $\mathcal{M}$ on $\{i, \boldsymbol{x}_j\}_{j=0}^{N-1}$ and sends the memory content of running the model $\mathcal{M}$ to Bob. This should include the state $\boldsymbol{Z}_{\mathcal{M}}^{N-1}$ of size $\tilde{d}$ as we can reasonably assume that both have access to the set of functions $\{f_{\mathcal{M}}^j\}_{j \in [N]}$. 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.

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

[8]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 $2/3$.

That is, the total number of bits that are communicated in this protocol is $\boldsymbol{Z}_{\mathcal{M}}^{N-1}$. Now, if $\max_j \boldsymbol{Z}_{\mathcal{M}}^{j}$ 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 F.9 and hence, we conclude that the model $\mathcal{M}$ solving AR also needs $\Omega(N)$ bits. □

Corollary F.13. *Given an input $\boldsymbol{u} \in \mathbb{R}^{N\times d}$ to the AR problem, a causal* `Mamba` *model with all entries in its computation taking $O(1)$ bits needs $d + \overline{d} \geq \Omega(N)$ to solve AR.*

Proof. We will first show that causal `Mamba` is a recurrent model. To see this, first observe equation 21 and note the fact that the input-dependent parameters $\overline{\boldsymbol{A}}, \overline{\boldsymbol{B}}, \boldsymbol{C}, \Delta$ are causal as mentioned in Definition F.4.

Next, due to equation 21, in order to compute $\boldsymbol{z}_{N,:} \in \mathbb{R}^d$, we need $\boldsymbol{C}_N \in \mathbb{R}^{\overline{d}}, \overline{\boldsymbol{B}}_N \in \mathbb{R}^{\overline{d}}$ and $\Delta_N \in \mathbb{R}^d$ along with $\boldsymbol{h}[N-1,:] \in \mathbb{R}^{\overline{d}}$. Here, we have the $(N-1)$-st state $\boldsymbol{Z}_{\texttt{Mamba}}^{N-1} \in \mathbb{R}^{3\overline{d}+d}$ given by

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

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:

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

Thus, due to Theorem F.12, we can conclude that $\boldsymbol{Z}_{\texttt{Mamba}}^{N-1}$ does require $\Omega(N)$-bits to solve AR. Finally, assuming each entry of $\boldsymbol{Z}_{\texttt{Mamba}}^{N-1}$ needs $O(1)$ bits to represent, the overall state $\boldsymbol{Z}_{\texttt{Mamba}}^{N-1}$ needs $O(d+\overline{d})$ to represent, which completes the proof of the claim. □

F.4.3. Lower Bound on the Number of Layers for AR

Next, we will again use Theorem F.9 to provide a better bound on the number of layers required to solve AR. (Note that since AR is a special case of MQAR, the result below immediately implies Theorem 3.4.)

Theorem F.14. *Given an input $\boldsymbol{u} \in \{0,1\}^{N\times d}$ to the AR problem with any encoding such that $\log c \leq d \leq 2^{(\log N)^{1-\epsilon}}$ for $\epsilon > 0$, and c possible tokens from the vocabulary with $c \leq N$, a data-independent* `BaseConv` *model with model parameters taking $O(\log N)$ bits needs $\Omega(\epsilon \log\log N)$ layers to solve AR.*

Proof. For a `BaseConv` model that solves AR using L layers, by definition, there exists a polynomial $P(\boldsymbol{u})$ of degree at most 2^L that solves AR for any $\boldsymbol{u} \in \{0,1\}^{N\times d}$[9]. This is because for the output of the ith layer of `BaseConv`, given by $\boldsymbol{Z}_{\texttt{BaseConv}}^{i}$, we have

$$\boldsymbol{Z}_{\texttt{BaseConv}}^{i}(\boldsymbol{Y}_{\mathcal{M}}^{i-1}) \equiv P^i(\boldsymbol{Z}_{\texttt{BaseConv}}^{i-1}), \quad \deg(P^i) \leq 2,$$

for some polynomial P^i of degree 2 which simply takes the inner products allowing the model to solve AR, where $\boldsymbol{Z}_{\texttt{BaseConv}}^{0} := \boldsymbol{u}$. Further, for such a model with L layers, by composition, the output of the i-th layer for $i \in [L]$ is also a polynomial over the input $\boldsymbol{u}$ and has degree at most 2^i. At the end, we have a polynomial $P(\boldsymbol{u})$ of degree $\leq 2^L$ for $\boldsymbol{u} \in \{0,1\}^{N\times d}$. As in the proof of Theorem F.12, again take the instance instance $(\boldsymbol{x}, i)$ of the index problem with $\boldsymbol{x} \in \{0,1\}^N$ and the corresponding instance of the AR problem as before

$$\boldsymbol{u} := \{j, \boldsymbol{x}_j\}_{j=0}^{N-1}, i. \tag{26}$$

Next, we build the following one-way protocol for solving the index problem using the `BaseConv` model from the hypothesis that it solves AR. Alice with their access of $\boldsymbol{x} \in \{0,1\}^N$ will again generate an input $\boldsymbol{u}$ for AR (without the query) as in equation 26.

Alice first takes the values $\boldsymbol{a} := \boldsymbol{u}[0:N-2,:] \in \{0,1\}^{(N-1)\times d}$ and substitutes these known $(N-1)d$ values to define the following polynomial:

$$Q(\boldsymbol{u}_{N-1,0}, \ldots, \boldsymbol{u}_{N-1,d-1}) = P(\boldsymbol{a}, \boldsymbol{u}_{N-1,0}, \ldots, \boldsymbol{u}_{N-1,d-1}). \tag{27}$$

[9] Since `BaseConv` is data independent, note that the polynomial $P(\cdot)$ is defined once we fix N and d.

Here, note that Q is a polynomial in d variables that correspond to the values $\boldsymbol{u}[N-1,:]$ that Bob has and trivially has degree $D \leq 2^L$. Now, Alice can run the model $\mathcal{M}$, retrieve the coefficients of Q, and send it to Bob. Since we assume that P solves AR, Bob can take the coefficients of Q and substitute $\boldsymbol{u}[N-1,:]$ to Q to compute $P(\boldsymbol{u})$ which is the associated value of i.

Here, the polynomial Q that Alice sends has at most d^{2^L} coefficients as each term in Q can have degree at most 2^L. If each such coefficient has B bits, then using Theorem F.9, the total number of bits being communicated must satisfy $B \cdot d^{2^L} \geq \Omega(N)$. This follows from the fact that if $B \cdot d^{2^L} \leq o(N)$, then since the associated value of i in equation 26 is the answer to the indexing problem, we have shown that a one-way communication protocol for solving the index problem uses $o(N)$ communication complexity, which then contradicts Theorem F.9. Thus, we must have

$$B \cdot d^{2^L} \geq \Omega(N) \implies 2^L \log(d) \geq \log \frac{N}{B} - O(1).$$

Taking logarithm of both sides then yields

$$\begin{aligned} L &\geq \log \frac{\log \frac{N}{B}}{\log d} - O(1) \geq \log \frac{\log N - \log B}{\log d} - O(1) \\ &\geq \log \frac{\log N - \log B}{(\log N)^{1-\epsilon}}, \end{aligned} \tag{28}$$

where we use the fact that $d \leq 2^{(\log N)^{1-\epsilon}}$ for any $\epsilon > 0$ in equation 28.

Moreover, as the model parameters are assumed to be $O(\log N)$ bits, any coefficient in Q should have absolute value at most $2^{O(\log N)} \cdot Nd^{2^L}$ as each coefficient can be a product of at most Nd variables. That is, for some $\alpha > 0$, we have the following bound on each coefficient:

$$2^B \leq (N^{\alpha+1}d)^{2^L} \leq (N^{(\alpha+2)})^{2^L}$$

where the last equality uses the fact that $d \leq N$. We thus have

$$\log(B) \leq \log(\alpha+2) + L + \log\log N. \tag{29}$$

Substituting equation 29 to equation 28, we get

$$L \geq \log \frac{\log N - \log(\alpha+2) - L - \log\log N}{(\log N)^{1-\epsilon}} \tag{30}$$

Now, if $L > \log\log N$, we are done. Otherwise, if $L \leq \log\log N$, then we can substitute this to equation 30 to get

$$\begin{aligned} L &\geq \log \frac{\log N - \log(\alpha+2) - 2\log\log N}{(\log N)^{1-\epsilon}} \\ &= \log\log N - \log(\alpha+2) - 2\log\log N - (1-\epsilon)\log\log N \end{aligned} \tag{31}$$

We now claim that first term in equation 31 satisfies the following:

$$\log\log N - \log(\alpha+2) - 2\log\log N \geq (1 - \frac{\epsilon}{2})\log\log N. \tag{32}$$

To see this, note that, for sufficiently large enough N, the following holds:

$$\frac{\log N}{2} \geq \log(\alpha+2) + 2\log\log N,$$

hence, we get

$$\log\log N - \log(\alpha+2) - 2\log\log N \geq \log\frac{\log N}{2} \geq \log\log N - 1 \geq (1 - \frac{\epsilon}{2})\log\log N.$$

This proves the claim in equation 32. Finally, using equation 32, equation 31 leads to the following:

$$L \geq (1 - \frac{\epsilon}{2})\log\log N - (1-\epsilon)\log\log N = \frac{\epsilon}{2}\log\log N,$$

which still provides the lower bound $L = \Omega(\epsilon \log\log N)$, as desired. □

Remark F.15. We remark that it is possible to extend Theorem F.14 to any model whose output from each layer is a polynomial of some degree $\Delta \geq 2$ ot get a lower bound of $\Omega(\epsilon \log\log N / \log \Delta)$.

F.4.4. 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 `BaseConv` 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 F.16. *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:

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

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

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

However, since both $\boldsymbol{q}_{\lfloor i/3 \rfloor}, \boldsymbol{k}_{\lfloor j/3 \rfloor} \in \{-1,1\}^d$, we have $\boldsymbol{q}_{\lfloor i/3 \rfloor}, \boldsymbol{k}_{\lfloor j/3 \rfloor} \leq d$ 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 $\mathbf{Z} := \textsc{ReLU}(\mathbf{Q}\mathbf{K}^\top + \mathbf{B})$ so that we have

$$\mathbf{Z}[i,j] = \mathbb{1}\{\boldsymbol{q}_{\lfloor i/3 \rfloor} \equiv \boldsymbol{k}_{\lfloor j/3 \rfloor}\}.$$

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 $\boldsymbol{Z}' \in \mathbb{R}^{3N \times d}$:

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

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

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

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 `BaseConv` to the degree of the polynomial that it computes.

Lemma F.17. *For any input sequence $\boldsymbol{u}$, there exists a multilinear polynomial equivalent (over Boolean inputs) to the polynomial computed by L layers of `BaseConv` with degree at most 2^L.*

Proof. Let $P(\boldsymbol{u})$ be the polynomial computed by L layers of `BaseConv`. Since the output of a single layer of `BaseConv` 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 F.18. *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.

$$\boldsymbol{x}^{ij} = \boldsymbol{q}_i \texttt{ xnor } \boldsymbol{k}_j := \boldsymbol{q}_i \wedge \boldsymbol{k}_j \vee \neg \boldsymbol{q}_i \wedge \neg \boldsymbol{k}_j \text{ for } i > j, \tag{33}$$

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

$$\boldsymbol{y}^{ij} := \bigwedge_{k \in [d]} \boldsymbol{x}_k^{ij}, i > j. \tag{34}$$

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

$$\boldsymbol{z}_k^{ij} := \boldsymbol{y}_{ij} \wedge [\boldsymbol{v}_j]_k. \tag{35}$$

Thus, the output of the circuit can be represented as

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

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 34, 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 34 as follows.

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

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

$$\boldsymbol{z}^{ij}(\boldsymbol{u}) := \prod_{k \in [d]} \boldsymbol{u}_{ik}\boldsymbol{u}_{jk} + (1 - \boldsymbol{u}_{ik})(1 - \boldsymbol{u}_{jk})\boldsymbol{u}_{(i+1)k} \quad \text{for each } k \in [d], i > j, \tag{36}$$

where $\boldsymbol{u}[i + 1, :] \equiv \boldsymbol{v}_j$. It is then easy to observe that equation 36 is multi-linear and has degree $2d + 1$. □

We are now ready to provide the lower bound.

Theorem F.19. *A data-independent* `BaseConv` *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 F.18, 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 `BaseConv` model with L layers that exactly solves MQAR. Then, due to Lemma F.17, this yields a multilinear polynomial $P(\boldsymbol{u})$ of degree at most 2^L. Here, if $L \leq \log(2d)$, then the resulting `BaseConv` 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 `BaseConv` model needs $\geq \log(2d)$-layers to solve MQAR. □

F.5. Lower Bound on the Number of Layers for $d \geq \log_2 c$ with Specific Encodings

F.5.1. 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 F.20. *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: $\boldsymbol{u} := \{(\boldsymbol{u}_1, \mathbb{1}, \boldsymbol{u}_1), (\boldsymbol{u}_2, \mathbb{1}, \boldsymbol{u}_2)\}$ and solve EQ using L layers with M_{MQAR} returning $\mathbb{1}$ iff there is a match. □

Due to Proposition F.20, we obtain the following corollary.

Corollary F.21. *Any lower bound* $\overline{L}$ *on the number of layers* L *of* `BaseConv` *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 F.19 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.

F.5.2. The p-Hot Encoding for $p \geq 1$

Definition F.22 ((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 F.23 (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 F.24. The condition specified in Definition F.23 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 F.25. *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. □

Proposition F.26. *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 F.25, 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. □

Lemma F.27. *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 $\boldsymbol{u}^i := (\boldsymbol{u}^i_0, \ldots, \boldsymbol{u}^i_{p-1})$ for $i \in \{1, 2\}$, where for each $0 < j < p$ such that we have

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

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

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

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

□

Proposition F.28. *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:

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

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 F.26 and Lemma F.27. □

Theorem F.29. *A data-independent* `BaseConv` *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 F.21 that it suffices to show a lower bound for the EQ problem. Moreover, we know from Proposition F.28 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 `BaseConv` model with L layers that exactly solves EQ in the p-hot encoding setting. Then, due to Lemma F.17 and Proposition F.26, 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 `BaseConv` 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 `BaseConv` model needs $\geq \lfloor \log(2p) \rfloor$-layers to solve EQ. □

F.6. Upperbound on MQAR with sub-logarithmically many BaseConv layers

Setup: For an input $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V} \in \mathbb{R}^{N \times d}$, the MQAR problem is computing $(\boldsymbol{C} \odot (\boldsymbol{Q}\boldsymbol{K}^\top)) \times \boldsymbol{V}$ where $\boldsymbol{C} \in \mathbb{R}^{N \times N}$ is a lower triangular matrix with 1s in all possible non-zero positions:

$$\boldsymbol{C}[i,j] \equiv \begin{cases} 1 & \text{if } j \leq i \\ 0 & \text{otherwise.} \end{cases}$$

Further, we define the following notation:

$$b = \lceil \log(N+1) \rceil,$$

$$\overline{b} = \lceil \log(d+1) \rceil,$$

$$\overline{N} = \max(N, d).$$

Finally, we define $\text{bin}(i) \in \{0,1\}^b$ to be the binary representation of $1 \leq i \leq N$ and $\text{bin}(j) \in \{0,1\}^{\overline{b}}$ to be the binary representation of $1 \leq j \leq d$. All vectors are assumed to be in column form and all row and column indices will start from 1.

We assume that:

(i) Each row of $\boldsymbol{Q}$ and $\boldsymbol{K}$ use 1-hot encoding and

(ii) Each query matches with at most one key.

We show that BaseConv can compute the MQAR problem with $O(\log\log(\overline{N}))$ layers:

Theorem F.30. *The MQAR problem with* 1*-hot encoded tokens, at most one key match per query, and* $\overline{N} \geq 8$ *can be solved with* $\texttt{BaseConv}(N, O(\log\log(\overline{N})), d, O(\overline{N}\log(\overline{N})), O(\overline{N}\log(\overline{N})))$.

F.6.1. BASECONV PRIMITIVES

In this section we show some basic primitives that will be helpful in proving Theorem F.30.

We define $\boldsymbol{e}_i^{(j)}$ to be the the ith standard basis vector with a dimension of j (remember, both are one based indexing).

We first define the primitives and then show how to implement them using BaseConv.

Note that if a convolution matrix, denoted with $\boldsymbol{h}$ are given as a single column, all columns of this matrix are identical as as defined. Specifically, if given $\boldsymbol{h} \in \mathbb{R}^N$ and $\boldsymbol{x} \in \mathbb{R}^{N \times d}$we have $\boldsymbol{h} * \boldsymbol{h}$ to denote $\boldsymbol{k} * \boldsymbol{x}$ where $\boldsymbol{k}[:, j] = \boldsymbol{h} \forall j \in [d]$.

Definition F.31. $\texttt{repeat_columns}(\boldsymbol{y}, r)$
INPUT: $\boldsymbol{y} \in \mathbb{R}^{N' \times d'r}, r \in \mathbb{Z}^+$.
OUTPUT: $\boldsymbol{z} \in \mathbb{R}^{N' \times d'r}$, where $\boldsymbol{z}$ has each of the first d' columns of $\boldsymbol{y}$ repeated r times. In other words,

$$\boldsymbol{y} \equiv \begin{pmatrix} \overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(1)}}}, \ldots, \overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(d')}}}, \overset{\uparrow}{\underset{\downarrow}{?}}, \ldots, \overset{\uparrow}{\underset{\downarrow}{?}} \end{pmatrix}, \boldsymbol{z} \equiv \begin{pmatrix} \underbrace{\overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(1)}}}, \ldots \overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(1)}}}}_{\text{r times}}, \ldots, \underbrace{\overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(d')}}}, \ldots, \overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(d')}}}}_{\text{r times}} \end{pmatrix}.$$

Definition F.32. $\texttt{repeat_matrix}(\boldsymbol{y}, r)$
INPUT: $\boldsymbol{y} \in \mathbb{R}^{N'r \times d'}, r \in \mathbb{Z}^+$ such that $\boldsymbol{y}[N'+1:,:] = \boldsymbol{0}^{N'(r-1) \times d'}$.
OUTPUT: $\boldsymbol{z} \in \mathbb{R}^{N'r \times d'}$, where $\boldsymbol{z}$ is the first N' rows of $\boldsymbol{y}$ repeated r times. In other words,

$$\boldsymbol{y} \equiv \begin{pmatrix} \leftarrow \boldsymbol{y}[1:N',:] \rightarrow \\ \hline \boldsymbol{0}^{N' \times d'} \\ \hline \vdots \\ \hline \boldsymbol{0}^{N' \times d'} \end{pmatrix}, \boldsymbol{z} \equiv \left.\begin{pmatrix} \leftarrow \boldsymbol{y}[1:N',:] \rightarrow \\ \hline \leftarrow \boldsymbol{y}[1:N',:] \rightarrow \\ \hline \vdots \\ \hline \leftarrow \boldsymbol{y}[1:N',:] \rightarrow \end{pmatrix}\right\} r \text{ times}$$

Definition F.33. $\texttt{cumulative_sum}(\boldsymbol{y})$
INPUT: $\boldsymbol{y} \in \mathbb{R}^{N' \times d'}$.
OUTPUT: $\boldsymbol{z} \in \mathbb{R}^{N' \times d'}$, where each row of $\boldsymbol{z}$ is the sum of all rows of $\boldsymbol{y}$ with a smaller or equal index. In other words,

$$\boldsymbol{y} \equiv \begin{pmatrix} \leftarrow \boldsymbol{y}_1 \rightarrow \\ \hline \vdots \\ \hline \leftarrow \boldsymbol{y}_i \rightarrow \\ \hline \vdots \\ \hline \leftarrow \boldsymbol{y}_{N'} \rightarrow \end{pmatrix}, \boldsymbol{z} \equiv \begin{pmatrix} \leftarrow \boldsymbol{y}_1 \rightarrow \\ \hline \vdots \\ \hline \leftarrow \sum_{j=1}^{i} \boldsymbol{y}_j \rightarrow \\ \hline \vdots \\ \hline \leftarrow \sum_{j=1}^{N'} \boldsymbol{y}_j \rightarrow \end{pmatrix}.$$

Definition F.34. `sum_all_columns`$(\boldsymbol{y})$
INPUT: $\boldsymbol{y} \in \mathbb{R}^{N' \times d'}$.
OUTPUT: $\boldsymbol{z} \in \mathbb{R}^{N' \times d'}$,
where the first column of $\boldsymbol{z}$ has the sum of all columns of $\boldsymbol{y}$ and the rest are all zeros. In other words,

$$\boldsymbol{y} \equiv \left(\overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(1)}}}, \overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(2)}}}, \ldots, \overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(d')}}} \right), \boldsymbol{z} \equiv \left(\overset{\uparrow}{\underset{\downarrow}{\sum_{j=1}^{d'} \boldsymbol{y}^{(j)}}}, \overset{\uparrow}{\underset{\downarrow}{\mathbf{0}}}, \ldots, \overset{\uparrow}{\underset{\downarrow}{\mathbf{0}}} \right).$$

Definition F.35. `sum_column_blocks`$(\boldsymbol{y}, B)$
INPUT: $\boldsymbol{y} \in \mathbb{R}^{N' \times d'}, B \in \mathbb{Z}$ such that B divides d'.
OUTPUT: $\boldsymbol{z} \in \mathbb{R}^{N' \times d'}$, where of column blocks the first column block of $\boldsymbol{z}$ is the sum of all column blocks and the rest are zero. In other words,

$$\boldsymbol{z}[:, 1:B] \equiv \sum_{j=0}^{\frac{d'}{B}-1} \boldsymbol{y}[:, jB+1:(j+1)B]$$

$$\boldsymbol{z}[:, j] = \mathbf{0}^{N'} \text{ for all } j > B.$$

Note that `sum_all_columns`$(\boldsymbol{y})$ = `sum_column_blocks`$(\boldsymbol{y}, 1)$.

Definition F.36. `one_hot_encoding`$(\boldsymbol{y}, d)$
INPUT: $\boldsymbol{y} \in \mathbb{R}^{N' \times N'}$.
OUTPUT: $\boldsymbol{z} \in \mathbb{R}^{N' \times N'}$, where the first $\lceil \log(N') \rceil$ columns of each row of $\boldsymbol{y}$ represent a binary encoding $\overline{y}_i \in [1, N']$ which is converted to a 1-hot encoding. In other words,

$$\boldsymbol{y} \equiv \begin{pmatrix} \text{bin}(\overline{y}_1)^\top, \leftarrow ? \rightarrow \\ \hline \vdots \\ \hline \text{bin}(\overline{y}_i)^\top, \leftarrow ? \rightarrow \\ \hline \vdots \\ \hline \text{bin}(\overline{y}_{N'})^\top, \leftarrow ? \rightarrow \end{pmatrix} \qquad \boldsymbol{z} \equiv \begin{pmatrix} \mathbf{e}_{\overline{y}_1}^\top \\ \hline \vdots \\ \hline \mathbf{e}_{\overline{y}_i}^\top \\ \hline \vdots \\ \hline \mathbf{e}_{\overline{y}_{N'}}^\top \end{pmatrix}$$

Definition F.37. `remember`$(\boldsymbol{y}, r, t, f)$
INPUT: $\boldsymbol{y} \in \mathbb{R}^{N' \times d'}, r \in \mathbb{Z}, t \in \mathbb{Z}, f : \mathbb{R}^{t-r} \rightarrow \mathbb{R}^{t-r+s}, \boldsymbol{v}_1 \in \mathbb{R}^r, \boldsymbol{x} \in \mathbb{R}^{t-r}$, where $\boldsymbol{y}$ is defined as below.

OUTPUT: $\boldsymbol{z} \in \mathbb{R}^{N' \times d'}$, which is defined as follows:

$$\boldsymbol{y} \equiv \begin{pmatrix} \leftarrow \boldsymbol{v}_1 \rightarrow \\ \hline \leftarrow \boldsymbol{x} \rightarrow \\ \hline \mathbf{0}^{s \times d'} \\ \hline \leftarrow \boldsymbol{v}_2 \rightarrow \\ \hline \mathbf{0} \\ \hline \vdots \\ \hline \mathbf{0} \end{pmatrix} \qquad \boldsymbol{z} \equiv \begin{pmatrix} \leftarrow \boldsymbol{v_1} \rightarrow \\ \hline \leftarrow f(\boldsymbol{x}) \rightarrow \\ \hline \leftarrow \boldsymbol{v}_2 \rightarrow \\ \hline \mathbf{0} \\ \hline \vdots \\ \hline \mathbf{0} \end{pmatrix}$$

Recall, that $\texttt{shift-down}(\boldsymbol{y}, s)$ and $\texttt{shift-up}(\boldsymbol{y}, s)$ will shift the matrix $\boldsymbol{y}$ down or up by s rows, respectively.

Proposition F.38 ((Arora et al., 2023a)). *For any $\boldsymbol{y} \in \mathbb{R}^{N \times d}$, there exist $N, 1, d, N, d -$ `BaseConv` and $N, 3, d, N, d -$ `BaseConv` that computes $\texttt{shift_down}(\boldsymbol{y}, s)$ and $\texttt{shift_up}(\boldsymbol{y}, s)$ for any $s \leq N$.*

Now we will show how each primitive is implemented in terms of BASECONV layers.

Proposition F.39 (The Repeat Columns primitive). *For any $\boldsymbol{y} \in \mathbb{R}^{N' \times d'r}$ and any $r \in \mathbb{Z}^+$ there exists a $N', 1, d'r, N', d'r -$ `BaseConv` that computes $\texttt{repeat_columns}(\boldsymbol{y}, r)$.*

Proof. Define

$$\boldsymbol{z} \leftarrow \text{BASECONV}(\boldsymbol{y}, \boldsymbol{W}, \mathbf{0}^{N' \times d'}, \mathbf{0}^{N' \times d'}, \mathbf{1}^{N' \times d'}),$$

where $\boldsymbol{W} \in \mathbb{R}^{d' \times d'}$ is defined as:

$$\boldsymbol{W}[i, j] = \begin{cases} 1 & \text{if } \left\lceil \frac{i}{r} \right\rceil = j \\ 0 & \text{otherwise.} \end{cases}$$

Then note that the output of this layer is:

$$\boldsymbol{z} = (\boldsymbol{y}\boldsymbol{W} + \mathbf{0}^{N' \times d'}) \odot (\mathbf{0}^{N' \times d'} * \boldsymbol{y} + \mathbf{1}^{N' \times d'}) = \boldsymbol{y}\boldsymbol{W} = \left(\underbrace{\overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(1)}}}, \ldots \overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(1)}}}}_{\text{r times}}, \ldots, \underbrace{\overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(d')}}}, \ldots, \overset{\uparrow}{\underset{\downarrow}{\boldsymbol{y}^{(d')}}}}_{\text{r times}} \right),$$

where the last equality follows from the definition of $\boldsymbol{W}$. □

Proposition F.40 (The Repeat Matrix primitive). *For any $\boldsymbol{y} \in \mathbb{R}^{N'r \times d'}$ and any $r \in \mathbb{Z}^+$ there exists a $N'r, 1, d', N'r, d' -$ `BaseConv` that computes $\texttt{repeat_matrix}(\boldsymbol{y}, r)$.*

Proof. Define

$$\boldsymbol{z} \leftarrow \text{BASECONV}(\boldsymbol{y}, \mathbf{0}^{d' \times d'}, \mathbf{1}^{N'r \times d'}, \boldsymbol{h}, \mathbf{0}^{N'r \times d'}),$$

where $\boldsymbol{h} \in \mathbb{R}^{N' \times d'}$ is defined as:

$$\boldsymbol{h}(X) \equiv \sum_{j=0}^{r-1} \boldsymbol{x}^{N'j}.$$

The computation of the convolution will result in:

$$\begin{aligned}\boldsymbol{h} * \boldsymbol{y} &= \text{coeff}((\boldsymbol{y}(X)) \cdot (1 + X^{N'} + \ldots + X^{N' \times (r-1)}) \\ &= \text{coeff}(\boldsymbol{y}(X) + \boldsymbol{y}(X) \cdot X^{N'} + \ldots + \boldsymbol{y}(X) \cdot (X^{N' \times (r-1)})) \\ &= \boldsymbol{y} + \texttt{shift-down}(\boldsymbol{y}, N') + \ldots + \texttt{shift-down}(\boldsymbol{y}, rN').\end{aligned}$$

In the above, the final equality follows from Proposition F.38. The output of this layer will compute:

$$\boldsymbol{z} = (\boldsymbol{y} \cdot \mathbf{0}^{d' \times d'} + \mathbf{1}^{N' \times d'}) \odot (\boldsymbol{h} * \boldsymbol{y} + \mathbf{0}^{N' \times d'}) = \boldsymbol{h} * \boldsymbol{y} = \begin{pmatrix} \boldsymbol{y}[1 : N', :] \\ \hline \vdots \\ \hline \boldsymbol{y}[1 : N', :] \end{pmatrix}.$$

□

In the above, the last equality follows from Proposition F.38 and the fact that $\boldsymbol{y}[N' + 1 :, :] = \mathbf{0}^{((N'-1) \times r) \times d')}$

Proposition F.41 (The Cumulative Sum primitive)**.** *For any $\boldsymbol{y} \in \mathbb{R}^{N' \times d'}$ there exists a $N', 1, d', N', d' -$ `BaseConv` that computes* `cumulative_sum`$(\boldsymbol{y})$.

Proof. Define

$$\boldsymbol{z} \leftarrow \textsc{BaseConv}(\boldsymbol{y}, \mathbf{0}^{d' \times d'}, \mathbf{1}^{N' \times d'}, \boldsymbol{h}, \mathbf{0}^{N' \times d'}),$$

where $\boldsymbol{h} \in \mathbb{R}^{N' \times d'}$ is defined as:

$$\boldsymbol{h}(X) \equiv \sum_{j=0}^{N'-1} \boldsymbol{x}^j.$$

The computation of the convolution will result in:

$$\begin{aligned}\boldsymbol{h} * \boldsymbol{y} &= \text{coeff}((\boldsymbol{y}(X)) \cdot (1 + X + \ldots + X^{N'-1}) \\ &= \text{coeff}(\boldsymbol{y}(X) + \boldsymbol{y}(X) \cdot X + \ldots + \boldsymbol{y}(X) \cdot X^{N'-1}) \\ &= \boldsymbol{y} + \texttt{shift-down}(\boldsymbol{y}, 1) + \ldots + \texttt{shift-down}(\boldsymbol{y}, N' - 1).\end{aligned}$$

The output of this layer is:

$$\begin{pmatrix} \leftarrow \boldsymbol{y}_1 \rightarrow \\ \hline \vdots \\ \hline \leftarrow \sum_{j=1}^{i} \boldsymbol{y}_j \rightarrow \\ \hline \vdots \\ \hline \leftarrow \sum_{j=1}^{N} \boldsymbol{y}_j \rightarrow \end{pmatrix}.$$

□

Proposition F.42 (The Sum All Columns primitive)**.** *For any $\boldsymbol{y} \in \mathbb{R}^{N' \times d'}$ there exists a $N', 1, d', N', d' -$ `BaseConv` that computes* `sum_all_columns`$(\boldsymbol{y})$.

Proof. Define

$$\boldsymbol{z} \leftarrow \textsc{BaseConv}(\boldsymbol{y}, \boldsymbol{W}, \mathbf{0}^{N' \times d'}, \mathbf{0}^{N' \times d'}, \mathbf{1}^{N' \times d'}),$$

where $\boldsymbol{W} \in \mathbb{R}^{d' \times d'}$ is defined as:

$$\boldsymbol{W}[i,j] \equiv \begin{cases} 1 & \text{if } j = 1 \\ 0 & \text{otherwise.} \end{cases}$$

The output of this layer will be:

$$(\boldsymbol{y}\boldsymbol{W} + \mathbf{0}^{N' \times d'}) \odot (\mathbf{0}^{N' \times d'} * \boldsymbol{y} + \mathbf{1}^{N' \times d'}) = \boldsymbol{y}\boldsymbol{W} = \left(\overset{\uparrow}{\underset{\downarrow}{\textstyle\sum_{j=1}^{d'} \boldsymbol{y}^{(j)}}}, \overset{\uparrow}{\underset{\downarrow}{\mathbf{0}}}, \ldots, \overset{\uparrow}{\underset{\downarrow}{\mathbf{0}}} \right),$$

where the last equality follows from the definition of $\boldsymbol{W}$. □

Proposition F.43 (The Sum Block Columns primitive)**.** *For any $\boldsymbol{y} \in \mathbb{R}^{N' \times d'}$ and B that divides d' there exists a $N', 1, d', N', d'$ – `BaseConv` that computes `sum_column_blocks`$(\boldsymbol{y}, B)$.*

Proof. Define

$$\boldsymbol{z} \leftarrow \textsc{BaseConv}(\boldsymbol{y}, \boldsymbol{W}, \mathbf{0}^{N' \times d'}, \mathbf{0}^{N' \times d'}, \mathbf{1}^{N' \times d'}),$$

where $\boldsymbol{W} \in \mathbb{R}^{d' \times d'}$ is defined as:

$$\boldsymbol{W}[i,j] \equiv \begin{cases} 1 & \text{if } j \leq B \text{ and } j - B\lfloor \frac{i}{B} \rfloor \\ 0 & \text{otherwise.} \end{cases}$$

The output of this layer will be:

$$\begin{aligned} (\boldsymbol{y}\boldsymbol{W} + \mathbf{0}^{N' \times d'}) \odot (\mathbf{0}^{N' \times d'} * \boldsymbol{y} + \mathbf{1}^{N' \times d'}) &= \boldsymbol{y}\boldsymbol{W} \\ &= \left(\overset{\uparrow}{\underset{\downarrow}{\textstyle\sum_{j=0}^{\frac{d'}{B}-1} \boldsymbol{y}[:, jB+1]}}, \ldots, \overset{\uparrow}{\underset{\downarrow}{\textstyle\sum_{j=0}^{\frac{d'}{B}-1} \boldsymbol{y}[:, jB+B-1]}}, \underbrace{\mathbf{0}, \ldots, \mathbf{0}}_{d' - \lfloor \frac{d'}{D} \rfloor \text{times}} \right), \end{aligned}$$

where the last equality follows from definition of $\boldsymbol{W}$.

□

Proposition F.44 (The 1-hot primitive)**.** *For any $\boldsymbol{y} \in \mathbb{R}^{N' \times N'}$ there exists a $N', \lceil \log\log(N') \rceil + O(1), N', 2N' \lceil \log N' \rceil, N'$ – `BaseConv` that computes `one_hot_encoding`$(\boldsymbol{y})$.*

Proof. We first give a sketch of the proof. Each row has a binary representation of a number which we want to convert to it's 1-hot encoding. In order to do this, we need to know which position in the 1-hot encoded vector needs to be 1. We need to extract information from each bit, which details which subset of positions the 1-hot encoded vector could potentially have as 1. Concretely, if the least significant bit is 0, only the even position can be represented, if that same bit is 1, only the odd positions can be represented. This pattern continues for all bits in the binary number. Each bit in the binary representation gets its own row. Finally, we take the bit wise ANDs of each row of the same binary representation to get the resulting 1-hot encoded vector. Next, we present the details.

First compute $\boldsymbol{z}_1 \in \mathbb{R}^{2N' \lceil \log N' \rceil \times N'}$ defined as:

$$\boldsymbol{z}_1 \leftarrow \texttt{repeat_matrix}(\boldsymbol{y}, 2\lceil \log(N') \rceil).$$

Define

$$\boldsymbol{z}_2 \leftarrow \texttt{BaseConv}(\boldsymbol{z}_1, \boldsymbol{I}^{N' \times N'}, \mathbf{0}^{2N' \lceil \log N' \rceil \times N'}, \mathbf{0}^{2N' \lceil \log N' \rceil \times N'}, \boldsymbol{b}_2^2),$$

where $\boldsymbol{b}_2 \in \mathbb{R}^{2N'\lceil \log N' \rceil \times N'}$ is defined as for $1 \leq i \leq 2N'\lceil \log N' \rceil$ and $1 \leq j \leq N'$:

$$\boldsymbol{b}_2^2[i,j] \equiv \begin{cases} 1 & \text{if } \left\lfloor \frac{i}{N'} \right\rfloor \mod \lceil \log(N') \rceil + 1 = j \\ 0 & \text{otherwise.} \end{cases}$$

Note that $\boldsymbol{z}_1$ has $2\log(N')$ copies of the binary representations in the first column block. When we zero them out to get $\boldsymbol{z}_2$, the $(i \mod \log(N'))$th matrix stores the value of the ith bit with all others being set to zero.
Compute $\boldsymbol{z}_3 \in \mathbb{R}^{2N'\lceil \log N' \rceil \times N'}$ by storing the sum all columns in the first column. Being that each row only has a single non-zero entry, this is equivalent to moving every one of these non-zero entries to the first column. Define

$$\boldsymbol{z}_3 \leftarrow \texttt{sum_all_columns}(\boldsymbol{z}_2).$$

Compute $\boldsymbol{z}_4 \in \mathbb{R}^{2N'\lceil \log N' \rceil \times N'}$ by copying the first column to all other columns. Define

$$\boldsymbol{z}_4 \leftarrow \texttt{repeat_columns}(\boldsymbol{z}_3, N').$$

Then, we will take a Hadamard product of $\boldsymbol{z}_4$ with a binary representation matrix. Define

$$\boldsymbol{z}_5 \leftarrow \texttt{BaseConv}\left(\boldsymbol{z}_4, \boldsymbol{0}^{N' \times N'}, \boldsymbol{b}_1^5, 1, \boldsymbol{0}^{2N'\lceil \log N' \rceil \times N'}\right).$$

In the above, $\boldsymbol{b}_1^5$ has the positions where a binary number with that has that bit set to a 1 could fall(and 0 if it in the bottom half of the matrix. We will define it in blocks where $1 \leq i \leq 2\lceil \log N' \rceil, 1 \leq k \leq N', 1 \leq j \leq N'$

$$\boldsymbol{b}_1^5[(i,k),j] \equiv \begin{cases} 1 & \text{if } j \mod 2^i \geq 2^{i-1} \text{ and } i \leq \lceil \log N' \rceil \\ 1 & \text{if } j \mod 2^i < 2^{i-1} \text{ and } i > \lceil \log N' \rceil \\ 0 & \text{otherwise.} \end{cases}$$

Next, we will combine the two representations:

$$\boldsymbol{z}_6 \leftarrow \texttt{BaseConv}\left(\boldsymbol{z}_5, \boldsymbol{0}^{N' \times N'}, \boldsymbol{b}_1^6, \boldsymbol{h}^6, \boldsymbol{0}^{2N'\lceil \log N' \rceil \times N'}\right),$$

where $\boldsymbol{b}_1^6, \boldsymbol{h}^6$ are defined as:

$$\boldsymbol{b}_1^6 = \begin{pmatrix} \boldsymbol{0}^{(N'\lceil \log N' \rceil) \times N'} \\ \hline \boldsymbol{1}^{(N'\lceil \log N' \rceil) \times N'} \end{pmatrix}, \boldsymbol{h}^6 = \begin{pmatrix} \boldsymbol{e}_1^{(N'\lceil \log N' \rceil)} \\ \hline \boldsymbol{e}_1^{(N'\lceil \log N' \rceil)} \end{pmatrix}.$$

We now specify the results of this kernel:

$$\begin{aligned} \boldsymbol{h}^6 * \boldsymbol{z}_5 &= \text{coeff}\, 1 + X^{N'\lceil \log N' \rceil} \cdot \boldsymbol{z_5}(x) \\ &= \text{coeff}\, \boldsymbol{z_5}(x) + \boldsymbol{z_5}(x) \cdot X^{N'\lceil \log N' \rceil} \\ &= \boldsymbol{z_5} + \texttt{shift-down}(\boldsymbol{z_5}, X^{N'\lceil \log N' \rceil}) \end{aligned}$$

By combining the possible positions based on a 0 or 1 being present in the binary number, $\boldsymbol{z}_6$ now stores the expanded binary representation in each row block in the bottom half of the matrix. We move this to the top half as shown below:

$$\boldsymbol{z}_7 \leftarrow \texttt{shift-up}(\boldsymbol{z}_6, N'\lceil \log N' \rceil).$$

Finally we do a bit wise multiplication between corresponding rows. For $0 \leq m < \lceil \log \log N' \rceil$ such that on the m'th iteration the following function is performed:

$$\boldsymbol{z}'_{8,m} \leftarrow \textsc{BaseConv}(\boldsymbol{z}_{8,m-1}, \boldsymbol{I}^{2N'\lceil \log N' \rceil \times N'}, \boldsymbol{0}^{2N'\lceil \log N' \rceil \times N'}, \boldsymbol{h}_m, \boldsymbol{0}^{2N'\lceil \log N' \rceil \times N'})$$

$$\boldsymbol{z}_{8,m} \leftarrow \texttt{shift-up}(\boldsymbol{z}'_{8,m}, N'2^m)$$

Where $\boldsymbol{z}_{8,-1} \equiv \boldsymbol{z}_7$ and $\boldsymbol{h}_m$ is defined below:

$$\boldsymbol{h}_m = \begin{pmatrix} \boldsymbol{e}_1^{(2^m)} \\ \hline \boldsymbol{e}_1^{(2^m - 2N'\lceil \log N' \rceil)} \end{pmatrix}.$$

The computation of this convolution will result in:

$$\begin{aligned} \boldsymbol{h}_m * \boldsymbol{z}_{8,m} &= \text{coeff}1 + X^{(2^m)} \cdot \boldsymbol{z}_{8,m}(x) \\ &= \text{coeff}\boldsymbol{z}_{8,m}(x) + \boldsymbol{z}_{8,m}(x) \cdot X^{(2^m)} \\ &= \boldsymbol{z}_{8,m} + \texttt{shift-down}(\boldsymbol{z}_{8,m}, X^{(2^m)}) \end{aligned}$$

$\boldsymbol{z}_7$ holds the possible positions of of the "1" in the one hot vector for each bit in the binary representation. This step takes a bit-wise and of the rows corresponding to the same binary representation so that we are left with the 1-hot encoding of the original binary representation. The idea is that for each bit in each binary representation of each value, there is a row in $\boldsymbol{z}_7$ that represents the possible positions, in the form of a bitmap, that this binary number can encode. When we bitwise AND all of these rows together, we are left with the position that satisfies all the constraints and is therefore the index that these binary numbers encoded.

□

Proposition F.45 (The Remembering Primitive)**.** *For any $\boldsymbol{x} \in \mathbb{R}^{n \times d'}, \boldsymbol{v}_1 \in \mathbb{R}^{r \times d'}, \boldsymbol{v}_2 \in \mathbb{R}^{m-r}$ where $n = t - r$ contained in some $\boldsymbol{y} \in \mathbb{R}^{N' \times d'}$ such that $\boldsymbol{v}_1$ is in the first r rows, $\boldsymbol{x}$ is in the next n rows, 0s fill up the next s rows, and $\boldsymbol{v}_2$ are in the next $m - r$ rows, for some $3n + 3m + 2s + 2t \leq N'$ so that for $\boldsymbol{h} \in \mathbb{R}^{n \times d}$ and $\boldsymbol{W} \in \mathbb{R}^{d' \times d'}$ with $\boldsymbol{x} * \boldsymbol{h} \in \mathbb{R}^{(n+s) \times d'}$ and $\boldsymbol{v} * \boldsymbol{h} \in \mathbb{R}^{(m+t) \times d'}$, where $\boldsymbol{v} \in \mathbb{R}^{m \times d}$ is defined as $\boldsymbol{v}_2 + \texttt{shift_down}(\boldsymbol{v}_1, m - r)$, there exists a $N', 8, d', N', d' - \texttt{BaseConv}$ that computes $\texttt{remember}(\boldsymbol{y}, r, t, f)$, where f can be implemented in 1 layer of* BaseConv *through the parameters $\boldsymbol{W} \in \mathbb{R}^{d' \times d'}, \boldsymbol{h} \in \mathbb{R}^{N' \times d'}, \boldsymbol{b}_1 \in \mathbb{R}^{N' \times d'}, \boldsymbol{b}_2 \in \mathbb{R}^{N' \times d'}$ as defined below:*

$$f(\boldsymbol{u}) = \begin{pmatrix} \boldsymbol{u}\boldsymbol{W} \\ \boldsymbol{0}^{s \times d'} \end{pmatrix} + \begin{pmatrix} \boldsymbol{b}_1 \\ \boldsymbol{1}^{s \times d'} \end{pmatrix} \odot \boldsymbol{u} * \boldsymbol{h} + \begin{pmatrix} \boldsymbol{b}_2 \\ \boldsymbol{0}^{s \times d'} \end{pmatrix}$$

Proof. First, we will convert the $\boldsymbol{y}$ so that it stores $\boldsymbol{v}$ in consecutive rows to get $\boldsymbol{z}_1$. Recall

$$\boldsymbol{y} = \begin{pmatrix} \leftarrow \boldsymbol{v}_1 \rightarrow \\ \hline \leftarrow \boldsymbol{x} \rightarrow \\ \hline \boldsymbol{0}^{s \times d'} \\ \hline \leftarrow \boldsymbol{v}_2 \rightarrow \\ \hline \boldsymbol{0}^{(N'-m-s-n) \times d'} \end{pmatrix}.$$

We compute $\boldsymbol{z}_1$ as:

$$\boldsymbol{z}_1 \leftarrow \textsc{BaseConv}(\boldsymbol{y}, \boldsymbol{0}^{d' \times d'}, \boldsymbol{b}_1^1, \boldsymbol{h}^1, \boldsymbol{0}^{N' \times d'}),$$

where the kernels $\boldsymbol{h}^1 \in \mathbb{R}^{N' \times d'}$ and $\boldsymbol{b}_1^1 \in \mathbb{R}^{N' \times d'}$ are given by:

$$\boldsymbol{h}^1 \leftarrow \begin{pmatrix} \boldsymbol{e}_1^{(n)} \\ \hline \mathbf{0}^m \\ \hline \mathbf{0}^s \\ \hline \boldsymbol{e}_1^{(n)} \\ \hline \mathbf{0}^n \\ \hline \dots \\ \hline \mathbf{0}^n \end{pmatrix}, \boldsymbol{b}_1^1 \leftarrow \begin{pmatrix} \mathbf{0}^{r \times d'} \\ \hline \mathbf{1}^{n \times d'} \\ \hline \mathbf{0}^{s \times d'} \\ \hline \mathbf{1}^{(m-r) \times d'} \\ \hline \mathbf{1}^{r \times d'} \\ \hline \mathbf{0}^{n \times d'} \\ \hline \mathbf{0}^{s \times d'} \\ \hline \mathbf{0}^{(m-r) \times d'} \\ \hline \dots \\ \hline \mathbf{0}^{n \times d'} \end{pmatrix}.$$

We now specify the result of this kernel:

$$\begin{aligned} \boldsymbol{h}^1 * \boldsymbol{y} &= \text{coeff}(1 + X^{n+s+m}) \cdot \boldsymbol{v}_1(X) + \boldsymbol{x}(X) \cdot X^r + \boldsymbol{v}_2(X) \cdot X^{n+s+r} \\ &= \text{coeff}\boldsymbol{v}_1(X) + \boldsymbol{x}(X) \cdot X^r + \boldsymbol{v}_2 \cdot X^{n+s+r} + \boldsymbol{v}_1(X) \cdot X^{n+s+m} + \boldsymbol{x}(X) \cdot X^{n+s+m+r} + \boldsymbol{v}_2 \cdot X^{2n+2s+m+r} \\ &= \boldsymbol{v}_1 + \texttt{shift-down}(\boldsymbol{x}, r) + \texttt{shift-down}(\boldsymbol{v}_2, n+s+r) + \texttt{shift-down}(\boldsymbol{v}_1, n+s+m) \\ &+ \texttt{shift-down}(\boldsymbol{x}, n+s+m+r) + \texttt{shift-down}(\boldsymbol{v}_2, 2n+2s+m+r). \end{aligned}$$

With this we have:

$$\boldsymbol{z}_1 = \boldsymbol{y} \cdot \mathbf{0}^{d' \times d'} + \boldsymbol{b}_1^1 \odot \boldsymbol{h}^1 * \boldsymbol{y} + \mathbf{0}^{N' \times d'} = \boldsymbol{b}_1^1 \odot \boldsymbol{h}^1 * \boldsymbol{y} = \begin{pmatrix} \mathbf{0}^{r \times d'} \\ \hline \mathbf{1}^{n \times d'} \\ \hline \mathbf{0}^{s \times d'} \\ \hline \mathbf{1}^{(m-r) \times d'} \\ \hline \mathbf{1}^{r \times d'} \\ \hline \mathbf{0}^{n \times d'} \\ \hline \mathbf{0}^{s \times d'} \\ \hline \mathbf{0}^{(m-r) \times d'} \\ \hline \dots \\ \hline \mathbf{0}^{n \times d'} \end{pmatrix} \odot \begin{pmatrix} \boldsymbol{v}_1 \\ \hline \boldsymbol{x} \\ \hline \mathbf{0}^{s \times d'} \\ \hline \boldsymbol{v}_2 \\ \hline \boldsymbol{v}_1 \\ \hline \boldsymbol{x} \\ \hline \mathbf{0}^{s \times d'} \\ \hline \boldsymbol{v}_2 \\ \hline \dots \\ \hline \mathbf{0}^{n \times d'} \end{pmatrix} = \begin{pmatrix} \mathbf{0}^{r \times d'} \\ \hline \boldsymbol{x} \\ \hline \mathbf{0}^{s \times d'} \\ \hline \boldsymbol{v}_2 \\ \hline \boldsymbol{v}_1 \\ \hline \mathbf{0}^{n \times d'} \\ \hline \mathbf{0}^{s \times d'} \\ \hline \mathbf{0}^{(m-r) \times d'} \\ \hline \dots \\ \hline \mathbf{0}^{n \times d'} \end{pmatrix} = \begin{pmatrix} \mathbf{0}^{r \times d'} \\ \hline \boldsymbol{x} \\ \hline \mathbf{0}^{s \times d'} \\ \hline \boldsymbol{v} \\ \hline \mathbf{0}^{n \times d'} \\ \hline \mathbf{0}^{s \times d'} \\ \hline \mathbf{0}^{(m-r) \times d'} \\ \hline \dots \\ \hline \mathbf{0}^{n \times d'} \end{pmatrix}.$$

Next compute $\boldsymbol{z}_2$:

$$\boldsymbol{z}_2 \leftarrow \texttt{shift-up}(\boldsymbol{z}_1, r),$$

as seen in (Arora et al., 2023a) Proposition F.38.

At this point $\boldsymbol{z}_2$ looks like:

$$\begin{pmatrix} \boldsymbol{x} \\ \hline \mathbf{0}^{s\times d'} \\ \hline \boldsymbol{v} \\ \hline \mathbf{0}^{n\times d'} \\ \hline \vdots \\ \hline \mathbf{0}^{n\times d'} \end{pmatrix}$$

Next, we will apply f to $\boldsymbol{z}_2$ to get $f(\boldsymbol{x})$ but also retain a unchanged version of $\boldsymbol{v}$. Define

$$\boldsymbol{z}_3 \leftarrow \textsc{BaseConv}(\boldsymbol{z}_2, \boldsymbol{W}, \boldsymbol{b}_1^3, \boldsymbol{h}^3, \boldsymbol{b}_2^3),$$

with the kernels $\boldsymbol{h}^3$, $\boldsymbol{b}_1^3$, and $\boldsymbol{b}_2^3$ for this layer are given by:

$$\boldsymbol{h}^3 \leftarrow \begin{pmatrix} \boldsymbol{h} \\ \hline \mathbf{0}^{s\times d'} \\ \hline \mathbf{0}^{m\times d'} \\ \hline \mathbf{0}^{t\times d'} \\ \hline \boldsymbol{e}_1^{(n)} \\ \hline \mathbf{0}^{s\times d'} \\ \hline \mathbf{0}^{n\times d'} \end{pmatrix}, \boldsymbol{b}_1^3 \leftarrow \begin{pmatrix} \boldsymbol{b}_1 \\ \hline \mathbf{1}^{s\times d'} \\ \hline \mathbf{0}^{m\times d'} \\ \hline \mathbf{0}^{t\times d'} \\ \hline \mathbf{0}^{n\times d'} \\ \hline \mathbf{0}^{s\times d'} \\ \hline \mathbf{1}^{m\times d'} \end{pmatrix}, \boldsymbol{b}_2^3 \leftarrow \begin{pmatrix} \boldsymbol{b}_2 \\ \hline \mathbf{0}^{s\times d'} \\ \hline \mathbf{0}^{m\times d'} \\ \hline \mathbf{0}^{t\times d'} \\ \hline \mathbf{0}^{n\times d'} \\ \hline \mathbf{0}^{s\times d'} \\ \hline \mathbf{0}^{m\times d'} \end{pmatrix}.$$

Remember that $\boldsymbol{W}, \boldsymbol{h}, \boldsymbol{b}_1, \boldsymbol{b}_2$ come from the definition of f.
We specify the result of this kernel as:

$$\begin{aligned}
\boldsymbol{h}^3 * \boldsymbol{z}_2 &= \text{coeff}(\boldsymbol{h}(X) + X^{n+s+m+t}) \cdot \boldsymbol{x}(X) + \boldsymbol{v}(X) \cdot X^{n+s} \\
&= \text{coeff}\boldsymbol{h} \cdot \boldsymbol{x}(X) + \boldsymbol{h} \cdot \boldsymbol{v}(X) \cdot X^{n+s} + \boldsymbol{x}(X) \cdot X^{n+s+m+t} + \boldsymbol{v}(X) \cdot X^{2n+2s+m+t} \\
&= \boldsymbol{h} * \boldsymbol{x} + \texttt{shift-down}(\boldsymbol{h} * \boldsymbol{v}, n+s) \\
&+ \texttt{shift-down}(\boldsymbol{x}, n+s+m+t) + \texttt{shift-down}(\boldsymbol{v}, 2n+2s+m+t).
\end{aligned}$$

With this we have

$$
\begin{aligned}
\boldsymbol{z}_3 = \boldsymbol{z}_2\boldsymbol{W} + \boldsymbol{b}_1^3 \odot \boldsymbol{h}^3 * \boldsymbol{z}_2 + \boldsymbol{b}_2^3 &= \begin{pmatrix} \boldsymbol{x}\boldsymbol{W} \\ \hline \mathbf{0}^{s\times d'} \\ \hline \boldsymbol{v}\boldsymbol{W} \\ \hline \mathbf{0}^{t\times d'} \\ \hline \mathbf{0}^{n\times d'} \\ \hline \mathbf{0}^{s\times d'} \\ \hline \mathbf{0}^{m\times d'} \end{pmatrix} + \begin{pmatrix} \boldsymbol{b}_1 \\ \hline \mathbf{1}^{s\times d'} \\ \hline \mathbf{0}^{m\times d'} \\ \hline \mathbf{0}^{t\times d'} \\ \hline \mathbf{0}^{n\times d'} \\ \hline \mathbf{0}^{s\times d'} \\ \hline \mathbf{1}^{m\times d'} \end{pmatrix} \odot \begin{pmatrix} \boldsymbol{h} * \boldsymbol{x} \\ \hline \\ \boldsymbol{h} * \boldsymbol{v} \\ \hline \\ \boldsymbol{x} \\ \hline \mathbf{0}^{s\times d'} \\ \hline \boldsymbol{v} \end{pmatrix} + \begin{pmatrix} \boldsymbol{b}_2 \\ \hline \mathbf{0}^{s\times d'} \\ \hline \mathbf{0}^{m\times d'} \\ \hline \mathbf{0}^{t\times d'} \\ \hline \mathbf{0}^{n\times d'} \\ \hline \mathbf{0}^{s\times d'} \\ \hline \mathbf{0}^{m\times d'} \end{pmatrix} \\
&= \begin{pmatrix} \begin{pmatrix} \boldsymbol{x}\boldsymbol{W} \\ \mathbf{0}^{s\times d'} \end{pmatrix} + \begin{pmatrix} \boldsymbol{b}_1 \\ \mathbf{1}^{s\times d'} \end{pmatrix} \odot \boldsymbol{h} * \boldsymbol{x} + \begin{pmatrix} \boldsymbol{b}_2 \\ \mathbf{0}^{s\times d'} \end{pmatrix} \\ \hline \begin{pmatrix} \boldsymbol{v}\boldsymbol{W} \\ \mathbf{0}^{t} \end{pmatrix} \odot \boldsymbol{h} * \boldsymbol{v} \\ \hline \mathbf{0}^{n\times d'} \\ \hline \mathbf{0}^{s\times d'} \\ \hline \boldsymbol{v} \end{pmatrix}.
\end{aligned}
$$

Note that $\begin{pmatrix} \boldsymbol{x}\boldsymbol{W} \\ \mathbf{0}^{s\times d'} \end{pmatrix} + \begin{pmatrix} \boldsymbol{b}_1 \\ \mathbf{0}^{s\times d'} \end{pmatrix} \odot \boldsymbol{h} * \boldsymbol{x} + \begin{pmatrix} \boldsymbol{b}_2 \\ \mathbf{0}^{s\times d'} \end{pmatrix}$ is $f(\boldsymbol{x})$ as defined. This next step will mask out duplicate and unnecessary $\boldsymbol{x}$ and $\boldsymbol{v}$ values. Define

$$
\boldsymbol{z}_4 \leftarrow \textsc{BaseConv}(\boldsymbol{z}_3, \boldsymbol{I}^{N'\times d'}, \mathbf{0}^{N'\times d'}, \mathbf{0}^{N'\times d'}, \boldsymbol{b}_2^4),
$$

where the kernel $\boldsymbol{b}_2^4 \in \mathbb{R}^{N'\times d'}$ for this layer is given by:

$$
\boldsymbol{b}_2^4 \leftarrow \begin{pmatrix} \mathbf{1}^{(n+s)\times d'} \\ \hline \mathbf{0}^{(m+t)\times d'} \\ \hline \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{1}^{(m)\times d'} \end{pmatrix}.
$$

We will specify the output of this layer:

$$\boldsymbol{z}_4 = \boldsymbol{z}_3 \odot \boldsymbol{b}_2^4 = \begin{pmatrix} f(\boldsymbol{x}) \\ \hline \begin{pmatrix} \boldsymbol{v}\boldsymbol{W} \\ \boldsymbol{0}^t \end{pmatrix} \odot \boldsymbol{h} * \boldsymbol{v} \\ \hline \boldsymbol{0}^{(n+s)\times d'} \\ \hline \boldsymbol{v} \end{pmatrix} \odot \begin{pmatrix} \boldsymbol{1}^{(n+s)\times d'} \\ \hline \boldsymbol{0}^{(m+t)\times d'} \\ \hline \boldsymbol{0}^{(n+s)\times d'} \\ \hline \boldsymbol{1}^{m\times d'} \end{pmatrix} = \begin{pmatrix} f(\boldsymbol{x}) \\ \hline \boldsymbol{0}^{(m+t)\times d'} \\ \hline \boldsymbol{0}^{(n+s)\times d'} \\ \hline \boldsymbol{v} \end{pmatrix}.$$

In the next step we will reorder the information such that $f(\boldsymbol{x})$ and $\boldsymbol{v}$ are contained in contiguous rows by copying it with a convolution. Define

$$\boldsymbol{z}_5 \leftarrow \textsc{BaseConv}(\boldsymbol{z}_4, \boldsymbol{0}^{d'\times d'}, \boldsymbol{b}_1^5, \boldsymbol{h}^5, \boldsymbol{0}^{N'\times d'}),$$

wehre the kernels $\boldsymbol{h}^5 \in \mathbb{R}^{N'\times d'}$ and $\boldsymbol{b}_1^5 \in \mathbb{R}^{N'\times d'}$ for this layer is given by:

$$\boldsymbol{h}^5 \leftarrow \begin{pmatrix} \boldsymbol{e}_1^{(n)} \\ \hline \boldsymbol{0}^{s\times d'} \\ \hline \boldsymbol{0}^{m\times d'} \\ \hline \boldsymbol{0}^{t\times d'} \\ \hline \boldsymbol{e}_1^{(n)} \\ \hline \boldsymbol{0}^{s\times d'} \\ \hline \boldsymbol{0}^{n\times d'} \end{pmatrix}, \boldsymbol{b}_1^5 \leftarrow \begin{pmatrix} \boldsymbol{0}^{(n+s)\times d'} \\ \hline \boldsymbol{0}^{(m+t)\times d'} \\ \hline \boldsymbol{1}^{(n+s)\times d'} \\ \hline \boldsymbol{1}^{m\times d'} \\ \hline \boldsymbol{0}^{(m+t)\times d'} \\ \hline \boldsymbol{0}^{(n+s)\times d'} \\ \hline \boldsymbol{0}^{m\times d'} \end{pmatrix}.$$

$$\begin{aligned} \boldsymbol{h}^5 * \boldsymbol{z}_4 &= \text{coeff}(1 + X^{n+s+m+t}) \cdot f(\boldsymbol{x})(X) + \boldsymbol{v}(X) \cdot X^{2n+2s+m+t} \\ &= \text{coeff}(f(\boldsymbol{x})(X) + f(\boldsymbol{x})(X) \cdot X^{n+s+m+t} + \boldsymbol{v}(X) \cdot X^{2n+2s+m+t} + \boldsymbol{v}(X) \cdot X^{3n+3s+2m+2t}) \\ &= f(\boldsymbol{x}) + \texttt{shift-down}(f(\boldsymbol{x}), n+s+m+t) + \texttt{shift-down}(\boldsymbol{v}, 2n+2s+m+t) \\ &+ \texttt{shift-down}(\boldsymbol{v}, 3n+3s+2m+2t). \end{aligned}$$

$$\boldsymbol{z}_5 = \boldsymbol{b}_1^5 \odot \boldsymbol{h}^5 * \boldsymbol{z}_4 = \begin{pmatrix} \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{0}^{(m+t)\times d'} \\ \hline \mathbf{1}^{(n+s)\times d'} \\ \hline \mathbf{1}^{m\times d'} \\ \hline \mathbf{0}^{(m+t)\times d'} \\ \hline \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{0}^{m\times d'} \end{pmatrix} \odot \begin{pmatrix} f(\boldsymbol{x}) \\ \hline \mathbf{0}^{(m+t)\times d'} \\ \hline f(\boldsymbol{x}) \\ \hline \boldsymbol{v} \\ \hline \mathbf{0}^{(m+t)\times d'} \\ \hline \mathbf{0}^{(n+s)\times d'} \\ \hline \boldsymbol{v} \end{pmatrix} = \begin{pmatrix} \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{0}^{(m+t)\times d'} \\ \hline f(\boldsymbol{x}) \\ \hline \boldsymbol{v} \\ \hline \mathbf{0}^{(m+t)\times d'} \\ \hline \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{0}^{m\times d'} \end{pmatrix}.$$

The next step we will duplicate the entries so we can position $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ in the same position relative to the enacted upon portion of the matrix as in the input. Define

$$\boldsymbol{z}_6 \leftarrow \textsc{BaseConv}(\boldsymbol{z}_5, \mathbf{0}^{d'\times d'}, \boldsymbol{b}_1^6, \boldsymbol{h}^6, \mathbf{0}^{N'\times d'}),$$

where the kernels $\boldsymbol{h}^6$ and $\boldsymbol{b}_1^6$ for this layer is given by:

$$\boldsymbol{h}^6 \leftarrow \begin{pmatrix} \boldsymbol{e}_1^{(2n+2s+2m+t)} \\ \hline \boldsymbol{e}_1^{(N'-2n-2s-2m-t)} \end{pmatrix}, \boldsymbol{b}_1^6 \leftarrow \begin{pmatrix} \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{0}^{(m+t)\times d'} \\ \hline \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{0}^{(m-r)\times d'} \\ \hline \mathbf{1}^{r\times d'} \\ \hline \mathbf{1}^{(n+s)\times d'} \\ \hline \mathbf{1}^{(m-r)\times d'} \\ \hline \mathbf{0}^{r\times d'} \end{pmatrix}.$$

Specifically the convolution does this:

$$\begin{aligned} \boldsymbol{h}^6 * \boldsymbol{z}_5 &= \text{coeff}(1 + X^{2n+2s+2m+t}) \cdot ((f(\boldsymbol{x}))(X)) \cdot X^{n+m+s+t} + \boldsymbol{v}(X) \cdot X^{2n+m+2s+t} \\ &= \text{coeff}((f(\boldsymbol{x}))(X)) \cdot X^{n+m+s+t} + \text{coeff}((f(\boldsymbol{x})(X)) \cdot X^{3n+3m+3s+2t}) \\ &+ \boldsymbol{v}(X) \cdot X^{2n+1m+2s+t} + \boldsymbol{v}(X) \cdot X^{4n+3m+4s+2t} \\ &= \texttt{shift-down}((f(\boldsymbol{x})(X)), n+m+s+t) \\ &+ \texttt{shift-down}((f(\boldsymbol{x})(X)), 3n+3m+3s+2t) \\ &+ \texttt{shift-down}(\boldsymbol{v}, 2n+m+2s+t) + \texttt{shift-down}(\boldsymbol{v}, 4n+3m+4s+2t). \end{aligned}$$

$$\boldsymbol{z}_6 = \boldsymbol{b}_1^6 \odot \boldsymbol{h}^6 * \boldsymbol{z}_5 = \begin{pmatrix} \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{0}^{(m+t)\times d'} \\ \hline \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{0}^{(m-r)\times d'} \\ \hline \mathbf{1}^{r\times d'} \\ \hline \mathbf{1}^{(n+s))\times d'} \\ \hline \mathbf{1}^{(m-r)\times d'} \\ \hline \mathbf{0}^{r\times d'} \end{pmatrix} \odot \begin{pmatrix} \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{0}^{(m+t)\times d'} \\ \hline f(\boldsymbol{x}) \\ \hline \boldsymbol{v}_2 \\ \hline \boldsymbol{v}_1 \\ \hline f(\boldsymbol{x}) \\ \hline \boldsymbol{v}_2 \\ \hline \boldsymbol{v}_1 \end{pmatrix} = \begin{pmatrix} \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{0}^{(m+t)\times d'} \\ \hline \mathbf{0}^{(n+s)\times d'} \\ \hline \mathbf{0}^{(m-r)\times d'} \\ \hline \boldsymbol{v}_1 \\ \hline f(x) \\ \hline \boldsymbol{v}_2 \\ \hline \mathbf{0}^{r\times d'} \end{pmatrix}.$$

Finally we get,

$$\boldsymbol{z}_7 \leftarrow \texttt{shift-up}(\boldsymbol{z}_6, 2n + 2m + 2s + t - r)$$

The final output of this layer is:

$$\boldsymbol{z}_7 \leftarrow \begin{pmatrix} \boldsymbol{v}_1 \\ \hline f(x) \\ \hline \boldsymbol{v}_2 \\ \hline \mathbf{0} \\ \hline \vdots \\ \hline \mathbf{0} \end{pmatrix},$$

which is our final output. □

Corollary F.46. *Let $\boldsymbol{y}$ be as in Proposition F.45 but now let f be implemented with* `BaseConv`(N', L, d', N', d'). *Then* `remember`$(\boldsymbol{y}, r, t, f)$ *where $t - r = n$ can be implemented with* `BaseConv`$(N', O(L), d', N', d')$.

Proof. The Remember primitive can be used to implement any number of BaseConv layers. As shown, the remember primitive can perform a BaseConv operation on a portion of a matrix while maintaining the values of the rest. This output matrix can then be fed through another remember primitive repeatedly such that any number of BaseConv layers can be performed through remember. □

Definition F.47. Being that primitives 1-7 can be implemented using BaseConv layers and that remember can apply BaseConv to a continuous subsection of a matrix, we can implement these primitives on subsections of any matrix "through" remember. This will be represented as `Remember`(i, j, f) where i and j are the start and end rows that will be effected, respectively, and f is the function which will be applied to them.

F.6.2. Proof of Theorem F.30

We first mathematically state the major steps, after which we will show how to implement each step using BaseConv layers.

1. INPUT: $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$.
   OUTPUT: $\boldsymbol{Q}, \boldsymbol{K}', \boldsymbol{V}$ where $\boldsymbol{K}' \in \mathbb{R}^{N \times db}$ is defined below.
   In steps 1.1 and 1.2, for $1 \leq i \leq N$, replace each 1 in $\boldsymbol{K}[i,:]$ by $\text{bin}(i)^\top$ to get $\overline{\boldsymbol{K}} \in \mathbb{R}^{N \times db}$. Then in step 1.3 then we compute $\boldsymbol{K}'$ where every row is the sum of all previous rows and itself in $\overline{\boldsymbol{K}}$.

   1.1 INPUT: $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$.
   OUTPUT: $\boldsymbol{Q}, \overline{\boldsymbol{K}}', \boldsymbol{V}$ where $\overline{\boldsymbol{K}}' \in \mathbb{R}^{N \times db}$ is defined below.

$$\overline{\boldsymbol{K}}' := \texttt{repeat_columns}(\boldsymbol{K}, b).$$

   1.2 INPUT: $\boldsymbol{Q}, \overline{\boldsymbol{K}}', \boldsymbol{V}$.
   OUTPUT: $\boldsymbol{Q}, \overline{\boldsymbol{K}}, \boldsymbol{V}$ where $\mathbf{B} \in \mathbb{R}^{N \times db}, \overline{\boldsymbol{K}}' \in \mathbb{R}^{N \times db}$ are defined below.

$$\mathbf{B}[i, jb+1:(j+1)b] := \text{bin}(i)^\top \text{ for all } 1 \leq i \leq N \text{ and } 1 \leq j \leq d. \tag{37}$$
$$\overline{\boldsymbol{K}} := \mathbf{B} \odot \overline{\boldsymbol{K}}'.$$

   1.3 INPUT: $\boldsymbol{Q}, \overline{\boldsymbol{K}}, \boldsymbol{V}$.
   OUTPUT: $\boldsymbol{Q}, \boldsymbol{K}', \boldsymbol{V}$ where $\boldsymbol{K}' \in \mathbb{R}^{N \times db}$ is defined below.

$$\boldsymbol{K}'[i,:] := \sum_{j=1}^{i} \overline{\boldsymbol{K}}[j,:] \text{ for all } 1 \leq i \leq N.$$

2. INPUT: $\boldsymbol{Q}, \boldsymbol{K}', \boldsymbol{V}$.
   OUTPUT: $\overline{\boldsymbol{M}}, \boldsymbol{V}$ where $\overline{\boldsymbol{M}} \in \mathbb{R}^{N \times db}$ is defined as follows.
   In steps 2.1-2.3, compute $\overline{\boldsymbol{M}}[i,:]$ so that $\overline{\boldsymbol{M}}[i, 1:b] = \text{bin}(j)^\top$ where $\boldsymbol{Q}[i,:] = \boldsymbol{K}[j,:]$ for every $1 \leq i \leq N$. Note that by assumption (ii) only one such j exists.

   2.1 INPUT: $\boldsymbol{Q}, \boldsymbol{K}', \boldsymbol{V}$.
   OUTPUT: $\boldsymbol{Q}', \boldsymbol{K}', \boldsymbol{V}$.
   Compute $\boldsymbol{Q}' \in \mathbb{R}^{N \times db}$ to be $\boldsymbol{Q}$ with each column repeated b times.

$$\boldsymbol{Q}' = \texttt{repeat_columns}(\boldsymbol{Q}, b).$$

   2.2 INPUT: $\boldsymbol{Q}', \boldsymbol{K}', \boldsymbol{V}$.
   OUTPUT: $\overline{\boldsymbol{M}}', \boldsymbol{V}$.
   Compute $\overline{\boldsymbol{M}}'$ as it is defined below:

$$\overline{\boldsymbol{M}}' = \boldsymbol{Q}' \odot \boldsymbol{K}'.$$

   Some column block of $\overline{\boldsymbol{M}}'$ holds $\text{bin}(j)$ such that $\boldsymbol{Q}[i,:]$ matches $\boldsymbol{K}[j,:]$, we now move it to the first column block in step 2.3.

   2.3 INPUT: $\overline{\boldsymbol{M}}', \boldsymbol{V}$.
   OUTPUT: $\overline{\boldsymbol{M}}, \boldsymbol{V}$. Compute $\overline{\boldsymbol{M}}$ as it is defined below:

$$\overline{\boldsymbol{M}} = \texttt{sum_column_blocks}(\overline{\boldsymbol{M}}', b).$$

   The first b entries in the i'th row $\overline{\boldsymbol{M}}$ holds $\text{bin}(j)$ such that $\boldsymbol{Q}[i,:]$ matches $\boldsymbol{K}[j,:]$.

3. INPUT: $\overline{\boldsymbol{M}}, \boldsymbol{V}$.
   OUTPUT: $\boldsymbol{L}, \boldsymbol{V}$ where $\boldsymbol{L} \in \mathbb{R}^{N \times N}$.
   Compute $\boldsymbol{L}$ from $\overline{\boldsymbol{M}}$ as defined below:

$$\boldsymbol{L} = \boldsymbol{C} \odot (\boldsymbol{Q}\boldsymbol{K}^\top).$$

   Compute $\boldsymbol{L} \in \mathbb{R}^{N \times N}$ from $\overline{\boldsymbol{M}}$ such that the binary representation of j in the i'th block is converted into 1-hot encoding of j in the i'th row of $\overline{\boldsymbol{M}}$. Define

$$\boldsymbol{L} \leftarrow \texttt{one_hot_encoding}(\overline{\boldsymbol{M}})$$

So we now have computed $\boldsymbol{L} = \boldsymbol{C} \odot (\boldsymbol{Q}\boldsymbol{K}^\top)$. All that's left to do is to compute $\boldsymbol{L} \times \boldsymbol{V}$. While mathematically this is a simple operation, we will implement this in multiple steps so that it is easy to implement this with BaseConv layers on input $\left(\frac{\boldsymbol{L}}{\boldsymbol{V}}\right)$.

4. INPUT: $\boldsymbol{L}, \boldsymbol{V}$.
   OUTPUT: $\boldsymbol{L}, \overline{\boldsymbol{V}}$, where $\overline{\boldsymbol{V}} \in \mathbb{R}^{N\bar{b} \times d}$ is defined below.
   In steps 4.1 and 4.2, compute $\overline{\boldsymbol{V}}$ from $\boldsymbol{V}$ where the v'th column holds the information for bin$(v) \in \{0,1\}^{\bar{b}}$, for every $1 \leq v \leq d$:

   4.1 INPUT: $\boldsymbol{L}, \boldsymbol{V}$.
   OUTPUT: $\boldsymbol{L}, \overline{\boldsymbol{V}}_1$, where $\overline{\boldsymbol{V}}_1 \in \mathbb{R}^{N\bar{b} \times d}$ is defined below.

$$\overline{\boldsymbol{V}}_1 := \texttt{repeat_matrix}(\boldsymbol{V}, \bar{b}).$$

   4.2 INPUT: $\boldsymbol{L}, \overline{\boldsymbol{V}}_1$.
   OUTPUT: $\boldsymbol{L}, \overline{\boldsymbol{V}}$, where $\overline{\boldsymbol{V}}_1 \in \mathbb{R}^{N\bar{b} \times d}$ is defined below. First, define $\mathbf{B}' \in^{N\bar{b} \times d}$ for $1 \leq i \leq \bar{b}, 1 \leq k \leq N, 1 \leq j \leq d$ as:

$$\mathbf{B}'[(i,k),j] \equiv \begin{cases} 1 & \text{if } j \mod 2^i \geq 2^{i-1} \\ 0 & \text{otherwise.} \end{cases} \tag{38}$$

   Then

$$\overline{\boldsymbol{V}} := \overline{\boldsymbol{V}}_1 \odot \mathbf{B}'.$$

5. INPUT: $\boldsymbol{L}, \overline{\boldsymbol{V}}$.
   OUTPUT: $\boldsymbol{L}, \boldsymbol{V}'$, where $\boldsymbol{V}' \in \mathbb{R}^{N\bar{b} \times d}$ is defined below.
   Compute $\boldsymbol{V}'$ from $\overline{\boldsymbol{V}}$ such that all the non-zero encodings of columns of $\overline{\boldsymbol{V}}$ are moved to 1st column and the other columns are zeroed out, specifically for $1 \leq i \leq Nb$:

$$\boldsymbol{V}'[:,i] := \begin{cases} \sum_{j=1}^{d} \overline{\boldsymbol{V}}[:,j] & \text{if } i = 1 \\ \mathbf{0}^{N\bar{b}} & \text{otherwise.} \end{cases}$$

   The summation of each column of $\overline{\boldsymbol{V}}$ performs this desired move action because each row of $\overline{\boldsymbol{V}}$ has at most 1 non zero value as each row of $\mathbf{B}'$ is a 1-hot encoding.

6. INPUT: $\boldsymbol{L}, \boldsymbol{V}'$.
   OUTPUT: $\boldsymbol{L}, \boldsymbol{V}_1$ where $\boldsymbol{V}_1 \in \mathbb{R}^{N \times N\bar{b}}$ is defined below.
   Compute $\boldsymbol{V}_1$ from $\boldsymbol{V}'$ in steps 6.1-6.6 such that for $1 \leq k \leq N$ and $\boldsymbol{W} \in \mathbb{R}^{N\bar{b} \times N\bar{b}}$:

$$\boldsymbol{W}[i,j] = \begin{cases} 1 & \text{if } i = (j-1)N + \left\lfloor \frac{i-1}{N} \right\rfloor \mod N\bar{b} + 1 \\ 0 & \text{otherwise.} \end{cases}$$

$$\boldsymbol{V}_1[k,:] := (\boldsymbol{V}'[:,1]^\top)\boldsymbol{W}.$$

   $\boldsymbol{W}$ is a permutation matrix which reorders the values such that values relating to the same index rather than values relating to the same copy of the matrix, made in step 4.1, are adjacent.

   6.1 INPUT: $\boldsymbol{L}, \boldsymbol{V}'$.
   OUTPUT: $\boldsymbol{L}, \boldsymbol{V}_2$.
   Compute $\boldsymbol{V}_2 \in \mathbb{R}^{N\bar{b} \times N\bar{b}}$ from $\boldsymbol{V}'$ where $\boldsymbol{V}_2$ has the 1st column of $\boldsymbol{V}'$ repeated $N\bar{b}$ times, as defined below for $1 \leq j \leq N\bar{b}$:

$$\boldsymbol{V}_2[:,j] = \boldsymbol{V}'[:,1]$$

6.2 INPUT: $\boldsymbol{L}, \boldsymbol{V}_2$.
OUTPUT: $\boldsymbol{L}, \boldsymbol{V}_3$.
Compute $\boldsymbol{V}_3 \in \mathbb{R}^{N\bar{b} \times N\bar{b}}$ from $\boldsymbol{V}_2$ by zeroing out all but the diagonal:

$$\boldsymbol{V}_3 = \boldsymbol{I}^{Nb \times Nb} \odot \boldsymbol{V}_2.$$

6.3 INPUT: $\boldsymbol{L}, \boldsymbol{V}_3$.
OUTPUT: $\boldsymbol{L}, \boldsymbol{V}_4$.
Compute $\boldsymbol{V}_4 \in \mathbb{R}^{N\bar{b} \times N\bar{b}}$ from $\boldsymbol{V}_3$ and $\boldsymbol{W}$ such that the values are permuted such that the values grouped by which row they represent, rather than the matrix repeat number they were in. Where $\boldsymbol{V}_4$ is defined as:

$$\boldsymbol{V}_4 = \boldsymbol{V}_3 \boldsymbol{W}.$$

6.4 INPUT: $\boldsymbol{L}, \boldsymbol{V}_4$.
OUTPUT: $\boldsymbol{L}, \boldsymbol{V}_5$.
Compute $\boldsymbol{V}_5 \in \mathbb{R}^{N\bar{b} \times N\bar{b}}$ from $\boldsymbol{V}_4$ such that $\boldsymbol{V}_5[1, :]$ is the sum of all rows by making $\boldsymbol{V}_5[N\bar{b}, :]$ the sum of all rows of $\boldsymbol{V}_4$ followed by zeroing the rest of rows. I.e. for all $1 \leq i, j \leq N\bar{b}$:

$$\boldsymbol{V}_5[i, :] := \begin{cases} \sum_{k=1}^{N\bar{b}} \boldsymbol{V}_4[k, :] & \text{if } i = N\bar{b} \\ \boldsymbol{0} & \text{otherwise.} \end{cases}$$

6.5 INPUT: $\boldsymbol{L}, \boldsymbol{V}_5$.
OUTPUT: $\boldsymbol{L}, \boldsymbol{V}_6$.
Compute $\boldsymbol{V}_6 \in \mathbb{R}^{N\bar{b} \times N\bar{b}}$ from $\boldsymbol{V}_5$ such that $\boldsymbol{V}_6[1, :] = \boldsymbol{V}_5[N\bar{b}, :]$ and the rest of the rows be zeroed out. I.e. for all $1 \leq i, j \leq N\bar{b}$:

$$\boldsymbol{V}_6[i, :] := \begin{cases} \boldsymbol{V}_5[N\bar{b}, :] & \text{if } i = 1 \\ \boldsymbol{0} & \text{otherwise.} \end{cases}$$

6.6 INPUT: $\boldsymbol{L}, \boldsymbol{V}_6$.
OUTPUT: $\boldsymbol{L}, \boldsymbol{V}_1$.
Compute $\boldsymbol{V}_1$ from $\boldsymbol{V}_6$ by copying $\boldsymbol{V}_6[1, :]$ to rest of the rows.

$$\boldsymbol{V}_1[i, :] := \boldsymbol{V}_6[1, :] \text{ for all } 1 \leq i \leq N.$$

At this point, each row of $\boldsymbol{V}_1$ now has the same values as the first column of $\boldsymbol{V}'$ permuted in the way that was stated in 6.3.

7. INPUT: $\boldsymbol{L}, \boldsymbol{V}_1$.
OUTPUT: $\overline{\boldsymbol{L}}$ where $\overline{\boldsymbol{L}} \in R^{N \times Nb}$ is defined below.
Compute $\overline{\boldsymbol{L}}$ such that the single 1 in $\boldsymbol{L}[i, :]$ (say at position $0 \leq j < N$) is replaced by $\text{bin}(j)^\top$. In other words, the i'th query were to match a key in position j, if the i'th row of $\overline{\boldsymbol{L}}$ would have a representation of the matching value at the j'th block.

7.1 INPUT: $\boldsymbol{L}, \boldsymbol{V}_1$.
OUTPUT: $\boldsymbol{L}', \boldsymbol{V}_1$
Compute $\boldsymbol{L}' \in R^{N \times N\bar{b}}$ by repeating each column of $\boldsymbol{L}$ $\bar{b}$ times:

$$\boldsymbol{L}'[:, j] = \boldsymbol{L}\left[:, \left\lceil \frac{j}{\bar{b}} \right\rceil\right] \text{ if } 1 \leq j \leq N\bar{b}.$$

7.2 INPUT: $\boldsymbol{L}', \boldsymbol{V}_1$.
OUTPUT: $\overline{\boldsymbol{L}}$
Compute $\overline{\boldsymbol{L}} \leftarrow \boldsymbol{L}' \odot \boldsymbol{V}_1$.

8. INPUT: $\overline{\boldsymbol{L}}$.
OUTPUT: $\overline{\boldsymbol{L}}_1 = \boldsymbol{L} \times \boldsymbol{V}$, the final output.

8.1 INPUT: $\overline{\boldsymbol{L}}$.
OUTPUT: $\overline{\boldsymbol{L}}_2 \in \mathbb{R}^{N \times (N \times \overline{b})}$ by summing up $\overline{b}$ chunks of columns of $\overline{\boldsymbol{L}}$ and store the result in 1st block column and zero out remaining columns to get $\overline{\boldsymbol{L}}_2$, specifically for $1 \leq j \leq \overline{b}$:

$$\overline{\boldsymbol{L}}_2[:, (i,j)] \equiv \begin{cases} \sum_{k=1}^{N} \boldsymbol{L}[:, (k,j)] & \text{if } i = 1 \\ \mathbf{0}^{N \times \overline{b}} & \text{otherwise.} \end{cases}$$

8.2 INPUT: $\overline{\boldsymbol{L}}_2$
OUTPUT: $\overline{\boldsymbol{L}}_1$ by replacing the binary representation in 1st block column by corresponding 1-hot encoding. Compute $\overline{\boldsymbol{L}}_1$ from $\overline{\boldsymbol{L}}_2$ such that for $1 \leq i \leq N$, $\overline{\boldsymbol{L}}_1[i,:] = \boldsymbol{e}_\ell^{(d)}$ where $\overline{\boldsymbol{L}}_2[i,(1,:)] = \text{bin}(\ell)^\top$.

Next, using the primitives defined in Appendix F.6.1, we show how each step above would be implemented with BASECONV layers. Instead of 3 separate matrices as shown in the math layout, we will have a single matrix, $\mathbf{Y} \in \mathbb{R}^{3N \times d}$ which contains the information of all three.

$$\mathbf{Y} = \begin{pmatrix} \boldsymbol{Q} \\ \hline \boldsymbol{K} \\ \hline \boldsymbol{V} \end{pmatrix}.$$

The internal dimension will be $(4\overline{N}\log(\overline{N}), \overline{N}\log(\overline{N}))$. For notational convenience, we define

$$z = \overline{N}\log(\overline{N}).$$

Specifically, the input $\boldsymbol{Y} \in \mathbb{R}^{3z \times d}$ has the top left matrix of $3N \times d$ holding $\begin{pmatrix} \boldsymbol{Q} \\ \boldsymbol{K} \\ \boldsymbol{V} \end{pmatrix}$.

Define

$$\mathbf{Y}_0 \leftarrow \text{BASECONV}(\mathbf{Y}, \mathbf{0}^{4z \times z}, \boldsymbol{b}_1^0, \boldsymbol{h}_0, \mathbf{0}^{4z \times z}),$$

where $\boldsymbol{b}_1^0 \in \mathbb{R}^{4z \times z}$ and $\boldsymbol{h}_0 \in \mathbb{R}^{4z \times z}$ are defined as:

$$\boldsymbol{b}_1^0 \leftarrow \begin{pmatrix} \mathbf{1}^{N \times z} \\ \hline \mathbf{0}^{(z-N) \times z} \\ \hline \mathbf{1}^{N \times z} \\ \hline \mathbf{0}^{(z-N) \times z} \\ \hline \mathbf{1}^{N \times z} \\ \hline \mathbf{0}^{(z-N) \times z} \\ \hline \mathbf{0}^{N \times z} \\ \hline \mathbf{0}^{(z-N) \times z} \end{pmatrix}, \boldsymbol{h}_0 \leftarrow \begin{pmatrix} \boldsymbol{e}_1^{(z-N)} \\ \hline \boldsymbol{e}_1^{(z-N)} \\ \hline \boldsymbol{e}_1^{(z-N)} \\ \hline \mathbf{0}^{z+3N} \end{pmatrix}.$$

The vectors that make up the convolution have dimension $z - N$ because $\boldsymbol{K}$ and $\boldsymbol{V}$ are not top left justified in the original matrix. This produces a matrix where $\boldsymbol{Q}, \boldsymbol{K}$ and $\boldsymbol{V}$ all sit in the top left position in their own sub-matrix surrounded by

zeros. The structure is:

$$\mathbf{Y}_0 \equiv \begin{pmatrix} \boldsymbol{Q}^p \\ \hline \boldsymbol{K}^p \\ \hline \boldsymbol{V}^p \\ \hline \mathbf{0}^{z \times z} \end{pmatrix}.$$

where

$$\boldsymbol{Q}^p = \begin{pmatrix} \boldsymbol{Q}, \mathbf{0}^{N \times (z)} \\ \mathbf{0}^{(z-N) \times (z)} \end{pmatrix}, \boldsymbol{K}^p = \begin{pmatrix} \boldsymbol{K}, \mathbf{0}^{N \times (z)} \\ \mathbf{0}^{(z-N) \times (z)} \end{pmatrix}, \boldsymbol{V}^p = \begin{pmatrix} \boldsymbol{V}, \mathbf{0}^{N \times (z)} \\ \mathbf{0}^{(z-N) \times (z)} \end{pmatrix}.$$

1. Compute $\mathbf{Y}_1$ in steps 1.1-1.2.

   1.1 Compute $\mathbf{Y}_1' \leftarrow \texttt{remember}(\mathbf{Y}_0, z+1, 2z, f_1)$ where f_1 is defined as:

   $$\texttt{repeat-columns}(\overline{\mathbf{Y}}_0, b).$$

   This results in

   $$\mathbf{Y}_1' \equiv \begin{pmatrix} \boldsymbol{Q}^p \\ \hline \overline{\boldsymbol{K}}' \\ \hline \boldsymbol{V}^p \\ \hline \mathbf{0}^{z \times z} \end{pmatrix}.$$

   This repeats the columns of $\boldsymbol{K}$ $\bar{b}$ times and doesn't change $\overline{\boldsymbol{Q}}$ or $\overline{\boldsymbol{V}}$ with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Corollary F.46.

   1.2 Compute $\mathbf{Y}_1'' \leftarrow \texttt{remember}(\mathbf{Y}_1', z+1, 2z, f_1')$ where f_1' is defined as:

   $$\textsc{BaseConv}(\overline{\mathbf{Y}}_1', \boldsymbol{I}^{z \times z}, \mathbf{0}^{z \times z}, \mathbf{0}^{z \times z}, \mathbf{B}).$$

   Where $\mathbf{B}$ is defined as it was in equation 37. This results in

   $$\mathbf{Y}_1'' \equiv \begin{pmatrix} \boldsymbol{Q}^p \\ \hline \overline{\boldsymbol{K}} \\ \hline \boldsymbol{V}^p \\ \hline \mathbf{0}^{z \times z} \end{pmatrix}.$$

   This replaces each 1 in $\boldsymbol{K}$ with the binary representation of the row with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Proposition F.45.

   1.3 Compute $\mathbf{Y}_1 \leftarrow \texttt{remember}(\mathbf{Y}_1'', z+1, 2z, f_1'')$ where f_1'' is defined as:

   $$\texttt{cumulative_sum}(\overline{\mathbf{Y}}_1'').$$

   This results in

   $$\mathbf{Y}_1 \equiv \begin{pmatrix} \boldsymbol{Q}^p \\ \hline \boldsymbol{K}' \\ \hline \boldsymbol{V}^p \\ \hline \mathbf{0}^{z \times z} \end{pmatrix}.$$

2. Compute $\mathbf{Y}_2$ in steps 2.1-2.3.

   2.1 Compute $\mathbf{Y}_2' \leftarrow \texttt{Remember}(\mathbf{Y}_1, 1, z, f_2)$ where f_2 is defined as:

$$\texttt{repeat-columns}(\overline{\mathbf{Y}}_1, b).$$

   This results in

$$\mathbf{Y}_2' \equiv \begin{pmatrix} \boldsymbol{Q}' \\ \hline \boldsymbol{K}' \\ \hline \boldsymbol{V}^p \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

   This repeats the columns of $\boldsymbol{Q}$ $\bar{b}$ times with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Corollary F.46.

   2.2 Compute $\mathbf{Y}_2$ as the Hadamard product of the first and second position stored in the second position. This can be done with the following substeps:

$$\mathbf{Y}_2 \leftarrow \textsc{BaseConv}(\overline{\mathbf{Y}}_2', \boldsymbol{I}^{z\times z}, \mathbf{0}^{4z\times z}, \boldsymbol{h}^2, \mathbf{0}^{4z\times z}).$$

   Where $\boldsymbol{h}^2 \in \mathbb{R}^{4z\times z}$ is defined as:

$$\boldsymbol{h}^2 \equiv \begin{pmatrix} \mathbf{0}^{z\times z} \\ \hline \boldsymbol{e}_1^{(z)} \\ \hline \mathbf{0}^{z\times z} \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

   This layer computes:

$$\mathbf{Y}_2'' = \mathbf{Y}_2' \odot \boldsymbol{h}^2 * \mathbf{Y}_2' = \begin{pmatrix} \boldsymbol{Q}' \\ \hline \boldsymbol{K}' \\ \hline \boldsymbol{V} \\ \hline \mathbf{1}^z \end{pmatrix} \odot \begin{pmatrix} \mathbf{0}^{z\times z} \\ \hline \boldsymbol{Q}' \\ \hline \boldsymbol{K}' \\ \hline \boldsymbol{V} \end{pmatrix} \equiv \begin{pmatrix} \mathbf{0}^{z\times z} \\ \hline \boldsymbol{Q}' \odot \boldsymbol{K}' \\ \hline \boldsymbol{K}' \odot \boldsymbol{V} \\ \hline \boldsymbol{V} \end{pmatrix}.$$

   Next, we mask out the unnecessary position using

$$\mathbf{Y}_2^3 \equiv \textsc{BaseConv}(\overline{\mathbf{Y}}_2'', \boldsymbol{b}_1^2, \mathbf{0}^{4z\times z}, \boldsymbol{e}_1^{4z\times z}, \mathbf{0}^{4z\times z}).$$

   Where $\boldsymbol{b}_1^2 \in \mathbb{R}^{4z\times z}$ is defined as:

$$\boldsymbol{b}_1^2 \equiv \begin{pmatrix} \mathbf{0}^{z\times z} \\ \hline \mathbf{1}^{z\times z} \\ \hline \mathbf{0}^{z\times z} \\ \hline \mathbf{1}^{z\times z} \end{pmatrix}.$$

This layer computes:

$$\mathbf{Y}_2^3 = \boldsymbol{b}_1^2 \odot \mathbf{Y}_2'' = \begin{pmatrix} \mathbf{0}^{z\times z} \\ \hline \mathbf{1}^{z\times z} \\ \hline \mathbf{0}^{z\times z} \\ \hline \mathbf{1}^{z} \end{pmatrix} \odot \begin{pmatrix} \mathbf{0}^{z\times z} \\ \hline \boldsymbol{Q}' \odot \boldsymbol{K}' \\ \hline \boldsymbol{K}' \odot \boldsymbol{V} \\ \hline \boldsymbol{V} \end{pmatrix} \equiv \begin{pmatrix} \mathbf{0}^{z\times z} \\ \hline \boldsymbol{Q}' \odot \boldsymbol{K}' \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V} \end{pmatrix} \equiv \begin{pmatrix} \mathbf{0}^{z\times z} \\ \hline \overline{\boldsymbol{M}}' \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V} \end{pmatrix}.$$

Finally, we shift up:

$$\mathbf{Y}_2^4 \leftarrow \texttt{shift-up}(\mathbf{Y}_2^3,\ z).$$

This results in:

$$\mathbf{Y}_2^4 = \begin{pmatrix} \overline{\boldsymbol{M}}' \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V} \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

This was done using $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via the vanilla BaseConv layers and Proposition F.38.

2.3 Sum up block columns in the first position to move the binary representations to the first column blocks through $\texttt{remember}(\mathbf{Y}_2^4, 1, z, f_2)$. Where f_2 is defined as:

$$\texttt{sum_column_blocks}(\overline{\mathbf{Y}}_2^4, b).$$

This results in:

$$\mathbf{Y}_2 \equiv \begin{pmatrix} \overline{\boldsymbol{M}} \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V} \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

Through this step, we have computed $\overline{\boldsymbol{M}}$ with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Corollary F.46..

3. Compute $\mathbf{Y}_3 \leftarrow \texttt{Remember}(\mathbf{Y}_2, 1, z,\ f_3)$ where f_3 is defined by:

$$\texttt{one_hot_encoding}(\overline{\mathbf{Y}}_2)$$

This step was computed with $\texttt{BaseConv}(4z, O(\lceil \log\log \overline{N} \rceil, z, 4z, z)$ via Corollary F.46. This converts $\overline{\boldsymbol{M}}$ to be 1-hot encoded in $O(\overline{N} \log \overline{N})$ BaseConv layers. This results in:

$$\mathbf{Y}_3 \equiv \begin{pmatrix} \boldsymbol{L} \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V} \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

We move this binary representation to the first column block with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Proposition F.38.

4. Compute $\mathbf{Y}_4$ in steps 4.1 and 4.2.

4.1 Compute $\mathbf{Y}_4' \leftarrow \texttt{remember}(\mathbf{Y}_3, 2z+1, 3z, f_4')$ where f_4' is defined as:

$$\texttt{repeat-matrix}(\overline{\mathbf{Y}}_3, \bar{b})$$

This results in:

$$\begin{pmatrix} \boldsymbol{L} \\ \hline \mathbf{0}^{z\times z} \\ \hline \overline{\boldsymbol{V}}' \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

We repeat $\boldsymbol{V}$ matrix $\bar{b}$ times with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Corollary F.46.

4.2 Compute $\mathbf{Y}_4 \leftarrow \texttt{remember}(\mathbf{Y}_4', 2z+1, 3z, f_4)$ where f_4 is defined as:

$$\texttt{BaseConv}(\overline{\mathbf{Y}}_4', \boldsymbol{I}^{z\times z}, \mathbf{0}^{z\times z}, \mathbf{0}^{z\times z}, \mathbf{B}').$$

Where $\mathbf{B}'$ is defined as it was in equation 38.
This results in:

$$\begin{pmatrix} \boldsymbol{L} \\ \hline \mathbf{0}^{z\times z} \\ \hline \overline{\boldsymbol{V}} \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

This step can be done with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Proposition F.45.

5. Compute $\mathbf{Y}_5 \leftarrow \texttt{remember}(\mathbf{Y}_4, 2z+1, 3z, f_5)$)where f_5 is defined as:

$$\texttt{sum-all-columns}(\overline{\mathbf{Y}}_4)$$

This results in:

$$\mathbf{Y}_5 \equiv \begin{pmatrix} \boldsymbol{L} \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V}' \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

Now each row of $\boldsymbol{V}'$ has that one moved to the first column if it existed with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Corollary F.46.

6. Compute $\mathbf{Y}_6$ through in steps 6.1-6.6.

6.1 Compute $\mathbf{Y}_6' \leftarrow \texttt{remember}(\mathbf{Y}_5, 2z+1, 3z, f_6')$ where f_6' is defined as:

$$\texttt{repeat_columns}(\overline{\mathbf{Y}}_5, N\bar{b}).$$

This results in:

$$\mathbf{Y}_6' \equiv \begin{pmatrix} \boldsymbol{L} \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V}_2 \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

Here we repeat the columns of $\boldsymbol{V}'$ with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Corollary F.46.

6.2 Compute $\mathbf{Y}_6'' \leftarrow \texttt{remember}\,(\mathbf{Y}_6', 2z+1, 3z, f_6'')$ where f_6'' is defined as:

$$\texttt{BaseConv}\,(\overline{\mathbf{Y}}_6', \mathbf{0}, \boldsymbol{I}^{z\times z}, \boldsymbol{e}_1^{(z)}, \mathbf{0}).$$

This results in:

$$\mathbf{Y}_6'' \equiv \begin{pmatrix} \boldsymbol{L} \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V}_3 \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

Here we zeroed out everything except the main diagonal in $\boldsymbol{V}$ with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Proposition F.45.

6.3 Compute $\mathbf{Y}_6^3 \leftarrow \texttt{remember}\,(\mathbf{Y}_6'', 2z+1, 3z, f_6^3)$ where f_6^3 is defined as:

$$\texttt{BaseConv}\,(\overline{\mathbf{Y}}_6'', \boldsymbol{W_6}, \mathbf{0}^{4z\times z}, \mathbf{0}^{4z\times z}, \mathbf{1}^{4z\times z}).$$

Where $\boldsymbol{W}_6$ is defined as it was in math step 6.
This results in:

$$\mathbf{Y}_6^3 \equiv \begin{pmatrix} \boldsymbol{L} \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V}_4 \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

Here, because we can only repeat whole matrices and not each row, we are permuting the values on the main diagonal to reorganize them to be as if they were repeated each row at a time with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Corollary F.46.

6.4 Compute $\mathbf{Y}_6^4 \leftarrow \texttt{remember}\,(\mathbf{Y}_6^3, 2z, 3z-1, f_6^4)$ where f_6^4 is defined as:

$$\texttt{cumulative_sum}\,(\overline{\mathbf{Y}}_6^3).$$

This results in:

$$\mathbf{Y}_6^4 \equiv \begin{pmatrix} \boldsymbol{L} \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V}_5 \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

Here, we use `cumulative_sum` so that the final row in the matrix stores each value in each column with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Corollary F.46.

6.5 Compute $\mathbf{Y}_6^5 \leftarrow \texttt{remember}\,(\mathbf{Y}_6^4, 2z, 3z-1, f_6^5)$ where f_6^5 is defined as:

$$\texttt{shift_up}\,(\overline{\mathbf{Y}}_6^4, N\bar{b}-1).$$

This results in:

$$\mathbf{Y}_6^5 \equiv \begin{pmatrix} \boldsymbol{L} \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V}_6 \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

Here, we shift this final row up to be in the first row with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Proposition F.38.

6.6 Compute $\mathbf{Y}_6 \leftarrow \texttt{remember}(\mathbf{Y}_6^5, 2z, 3z-1, f_6^6)$ where f_6^6 is defined as:

$$\texttt{cumulative_sum}(\overline{\mathbf{Y}}_6^5).$$

This results in:

$$\mathbf{Y}_6 \equiv \begin{pmatrix} \boldsymbol{L} \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V}_1 \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

Here we copied the first row to each row in $\boldsymbol{V}$ with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Corollary F.46. This results in:

$$\mathbf{Y}_{\mathbf{6}} \equiv \begin{pmatrix} \boldsymbol{L} \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V}_1 \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

This step is computed with with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Corollary F.46.

7. Compute $\mathbf{Y}_7$ in steps 7.1-7.3.

7.1 Compute $\mathbf{Y}_7' \leftarrow \texttt{remember}(\mathbf{Y}_6, 1, z, f_7)$ where f_7 is defined as:

$$\texttt{repeat_columns}(\overline{\mathbf{Y}}_6, \bar{b}).$$

This results in:

$$\mathbf{Y}_{\mathbf{7}}' \equiv \begin{pmatrix} \boldsymbol{L}' \\ \hline \mathbf{0}^{z\times z} \\ \hline \boldsymbol{V}_1 \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

Here we repeat the columns of $\boldsymbol{L}$ with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Corollary F.46.

7.2 We want to Hadamard the first and third position which we can do with:

$$\mathbf{Y}_7'' \equiv \textsc{BaseConv}(\mathbf{Y}_7', \boldsymbol{I}^{z\times z}, \mathbf{0}^{4z\times z}, \boldsymbol{h}^7, \mathbf{0}^{4z\times z}).$$

Where $\boldsymbol{h}^7 \in \mathbb{R}^{z\times z}$ is defined as:

$$\boldsymbol{h}^7 \equiv \begin{pmatrix} \mathbf{0}^{z\times z} \\ \hline \mathbf{0}^{z} \\ \hline \boldsymbol{e}_1^z \\ \hline \mathbf{0}^{z} \end{pmatrix}.$$

Here we Hadamard the $\boldsymbol{L}$ and $\boldsymbol{V}_1$ together. Which results in:

$$\boldsymbol{Y}_7'' = \begin{pmatrix} \mathbf{0}^{z\times z} \\ \hline \mathbf{0}^{z\times z} \\ \hline \overline{\boldsymbol{L}} \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}$$

This is done with with $\texttt{BaseConv}(4z, 1, z, 4z, z)$ via a single BaseConv layer.

7.3 Finally, we $\texttt{shift-up}(\mathbf{Y}_7'', 2z)$. This results in:

$$\mathbf{Y}_7 \equiv \begin{pmatrix} \overline{\boldsymbol{L}} \\ \hline \mathbf{0}^{z} \\ \hline \mathbf{0}^{z} \\ \hline \mathbf{0}^{z} \end{pmatrix}.$$

This is done with with $\texttt{BaseConv}(4z, O(1), z, 4z, z)$ via Proposition F.38.

8. Convert $\mathbf{Y}_7$ to the final output.

8.1 Here we will move everything to the first block using:

$$\mathbf{Y}_8' \equiv \texttt{sum_column_blocks}(\overline{\mathbf{Y}}_7, b).$$

$$\mathbf{Y}_{\mathbf{8}}' \equiv \begin{pmatrix} \overline{\boldsymbol{L}}_2 \\ \hline \mathbf{0}^{z\times z} \\ \hline \mathbf{0}^{z\times z} \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

This is done with with $\texttt{BaseConv}(4z, 1, z, 4z, z)$ via Corollary F.46.

8.2 Compute $\mathbf{Y}_8 \leftarrow \texttt{remember}(\mathbf{Y}_8', 1, z, f_8)$ where f_8 is defined as:

$$\texttt{one_hot_encoding}(\overline{\mathbf{Y}}_8').$$

This converts $\overline{\boldsymbol{L}}$ to be 1-hot encoded in $O(\overline{N}\log\overline{N})$ BaseConv layers. This results in:

$$\mathbf{Y}_{\mathbf{8}} \equiv \begin{pmatrix} \overline{\boldsymbol{L}}_1 \\ \hline \mathbf{0}^{z\times z} \\ \hline \mathbf{0}^{z\times z} \\ \hline \mathbf{0}^{z\times z} \end{pmatrix}.$$

This is done with with $\texttt{BaseConv}(4z, O(\lceil\log\log\overline{N}\rceil), z, 4z, z)$ via Corollary F.46.

Overall cost: We can solve the MQAR problem with $\texttt{BaseConv}(4z, O(\lceil\log\log\overline{N}\rceil) + O(1), z, 4z, z)$ by stacking the layers presented above.

Table 7. 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. `BaseConv` Layers	17	22
`BaseConv` Projection Expansion Factor	4	
`BaseConv` Filter Size	3	
`BaseConv` Activation	SiLU	

Table 8. 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 9. 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 10. 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 11. 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 12. 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 13. 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