Flash Inference: Near Linear Time Inference for Long Convolution Sequence Models and Beyond

Costin-Andrei Oncescu
Harvard University
`concescu@g.harvard.edu`

Sanket Purandare
Harvard University
`sanketpurandare@g.harvard.edu`

Stratos Idreos
Harvard University
`stratos@seas.harvard.edu`

Sham Kakade
Harvard University
`sham@seas.harvard.edu`

Abstract

While transformers have been at the core of most recent advancements in sequence generative models, their computational cost remains quadratic in sequence length. Several subquatratic architectures have been proposed to address this computational issue. Some of them, including long convolution sequence models (LCSMs), such as Hyena, address this issue at training time but remain quadratic during inference. We propose a method for speeding up LCSMs' inference to quasilinear time, identify the key properties that make this possible, and propose a general framework that exploits these. Our approach, inspired by previous work on relaxed polynomial interpolation, is based on a tiling which helps decrease memory movement and share computation. It has the added benefit of allowing for almost complete parallelization across layers of the position-mixing part of the architecture. Empirically, we provide a proof of concept implementation for Hyena-like settings, which gets up to $4\times$ improvement over standard inference.

1 Introduction

A lot of recent progress in deep learning, particularly in the form of large language models (LLMs) has been driven by the transformer architecture [Vaswani et al., 2017]. While these models have great quality, it comes at a computation cost which scales quadratically in sequence length - both during training and inference. This can become prohibitive for very long contexts and as such a number of alternative architectures with better computational scaling in context length have been proposed [Gu and Dao, 2023, Poli et al., 2023, Fu et al., 2024]. While most of these works have improved computational efficiency for training, some still scale quadratically in sequence length when it comes to inference, thus not improving asymptotically over transformers.

In this work, we propose a framework for optimizing inference efficiency for a general class of such models. As a case study, which inspired the method, we focus on long convolution sequence models (LCSMs) [Poli et al., 2023, Fu et al., 2022, Romero et al., 2021, Li et al., 2022, Karami and Ghodsi, 2024, Fu et al., 2023a]. However, our approach is not limited to LCSMs alone and we identify the properties that allow for such inference speedups in hope to guide the design of future architectures.

In the particular case of LCSMs (including Hyena), the building block of the architecture is that of convolving the input sequence with a sequence-length long, (potentially underparameterized) filter. If we let L be the sequence length (e.g. number of tokens in the case of LLMs), then a naive implementation of convolution during training would take $\Omega(L^2)$ FLOPs, but one can employ FFT

4th NeurIPS Efficient Natural Language and Speech Processing Workshop (ENLSP-IV 2024).

to bring that down to $O(L \log L)$. The issue that occurs during inference is that FFT cannot be used directly since the whole input sequence is not known ahead of time, but rather incrementally computed. Because of this, the naive inference approach goes up to $\Omega(L^2)$ - this is the apparent cost of moving from a *static* input to a *dynamic* one.

It turns out that in fact, at least in the case of "dynamic FFT", one can obtain $O(L \log^2 L)$ time complexity by using van der Hoeven [1997]'s result on relaxed polynomial convolution. Crucially, this result works by a direct reduction to several applications of FFT. This allows us to phrase a more general framework for turning training-efficient architectures to inference-efficient ones.

Our main contributions are:

- Proposing the first quasilinear-in-sequence-length $O(L \log^2 L)$, *exact* inference algorithm for LCSMs.
- Identifying the main features of LCSMs that allow for faster inference to propose a more general framework.
- Highlighting and exploiting the potential for dealing with most workload of different layers in parallel, rather than having to wait for all computation of a layer to finish before moving on to subsequent layers.
- Saving up on data movement thanks to the tiling approach used. We show how to save factors of 2 in several places, including activation storage, when the convolutional filters are data-independent. However, our method extends to any causal, data-dependent filters.
- Our framework provides an $O(L \log^2 L)$ inference algorithm which, when run empirically, yields up to $8\times$ time-efficiency improvement.

2 Setting and Related Work

Much recent effort has been put in designing more efficient alternatives to transformers [Tay et al., 2022]. A couple of well-performing classes are state space models (SSMs) [Gu et al., 2021, Gu and Dao, 2023] and convolution-based models[Fu et al., 2022, Poli et al., 2023]. In this work we focus on the latter class. To avoid a popular misconception, it is worth emphasizing that while any linear-time invariant (LTI) SSM has an equivalent convolutional filter, it is *not* true that any convolutional filter has an equivalent *low-dimensional* LTI SSM representation. It could be the case that the smallest LTI SSM equivalent to a length-L convolution filter has a dimension of L. However, such a dimension would deem an LTI SSM approach have quadratic time complexity both during training as well as during inference.

2.1 Notations and Definitions

In the most general form, we consider architectures obtained by stacking several position-mixing layers potentially interleaved with element-wise, feature-mixing modules, partially following Arora et al. [2023]'s notation. Put formally, let $\mathrm{block}_1, \dots \mathrm{block}_M : \mathbb{R}^D \to \mathbb{R}^D$ be feature-mixing modules and $\mathrm{mixer}_1 \dots \mathrm{mixer}_M : \mathbb{R}^{L \times D} \to \mathbb{R}^{L \times D}$ be position-mixing modules. One can think of the blocks as being combinations of MLPs, layernorms, skip connections and other modules of similar purpose and of mixers as playing the role of multi-head attention (MHA) in a transformer - the part where embeddings at different positions interact with each other. Assume these are all learnt from some well-defined parameterized classes. Here, D is the used embedding dimension and L can be any sequence length (unless positional embeddings are used, case in which it can only take values up to some L_{max}). We define the activations $a_0, \dots, a_M : \mathbb{R}^{L \times D} \to \mathbb{R}^{L \times D}$ at level $1 \le \ell \le M$ and position $1 \le i \le L$ as $(a_\ell(x))_i \triangleq \mathrm{block}_\ell(\mathrm{mixer}_\ell(a_{\ell-1}(x))_i)$ where $a_0(x) = x$ represents the input embeddings (L D-dimensional tokens) and the model output at position i is read from the last layer $(a_M(x))_i$.

We focus on autoregressive sequence models where for an input sequence $x_1, \dots x_p$, one generates the $(p+1)^{\text{th}}$ token by sampling from a distribution given by $(a_M(x))_p$ (usually via a linear map). In order for this generation procedure to be sensible, we constrain (by construction of mixers) the models to be causal, that is, $(a_\ell(x))_i$ is a function of only $x_{[1,i]}$ or, equivalently, $\mathrm{mixer}_\ell(y)_i$ is a function of $y_{[1..i]}$.

2.2 Self-attention

Self-attention is the building block of transformers [Vaswani et al., 2017]. In its simplest form, it implements a mixer by using three projection matrices $Q, K, V \in \mathbb{R}^{D\times D}$ to obtain three sequences of vectors $q, k, v \in \mathbb{R}^{L\times D}$ by projecting the input $y \in \mathbb{R}^{L\times D}$ via $q = yQ, k = yK, v = yV$ - these are called queries, keys and values, respectively. The (causal) self-attention operator $attention$: $\mathbb{R}^{L\times D} \to \mathbb{R}^{L\times D}$ is then given by:

$$attention(y)_j \triangleq \mathrm{mixer}(y)_j = \frac{\sum_{i=1}^{j} v_j \cdot e^{\langle q_j, k_i\rangle}}{\sum_{i=1}^{j} e^{\langle q_j, k_i\rangle}} = (v_{[1,j]})^\top \mathrm{softmax}(k_{[1,j]} q_j) \tag{1}$$

which represents, at position j, the average of the previous value vectors $v_{[1,j]}$ exponentially-weighted by how well the values' corresponding keys match j^{th} query. Put differently, the bigger $\langle q_i, k_j\rangle$ is, the more j^{th} output attends to i^{th} input. Note that both in the above notation, as well as in general, we assume any 1-dimensional tensor to be a column vector (for example $q_j \in \mathbb{R}^{D\times 1}$). In the transformer architecture this is how one "head" operates and each embedding is normally split into several such heads - this is called multihead attention (MHA). If we take the causality away (that is, the $i \le j$) for simplicity, one could think of attention as being $\mathrm{mixer}(y) = \mathrm{softmax}(qk^\top)v$ where $\mathrm{softmax}$ is computed along the rows of what is called the attention matrix: $qk^\top$.

2.3 Long Convolution Sequence Models (LCSMs)

LCSMs [Poli et al., 2023, Li et al., 2022, Gaido et al., 2024] work by creating a SISO (single-input-single-output) primitive to map an input sequence $y \in \mathbb{R}^L$ to $z \in \mathbb{R}^L$ via $z_t \mapsto \sum_{i=0}^{t} y_i \cdot k_{t-i}$ where k is the (possibly infinite, and of length at least L) convolution filter which is often parameterized by a smaller latent space Θ: $k = f(\theta)$ where $\theta \in \Theta, \dim\Theta \ll L$ is learnt. These convolution primitives operate on independent axes of the hidden dimensions to create a positional mixer: $\mathrm{mixer}(y)_{t,c} = \sum_{i=0}^{t} y_{i,c} \cdot k_{t-i,c}$. This assumes the filters to be independent, but shared ones are possible as well (as is the case for multi-head Hyena [Massaroli et al., 2024]).

For example, the Hyena architecture [Poli et al., 2023] maps back to our definitions if we assume the block functions, depending on the layer, are either MLPs or gates - that is, element-wise multiplications with a projection of activations at same position, but lower level. We do not focus on the details of the blocks since they all involve some $D \times D$ matrix-vector multiplication that is performed once for every layer and position $1 \le i \le L$ and thus scale as $\Theta(LD^2)$.

2.3.1 The Inference Problem

Whereas during training, the convolutions can be performed in $O(L \log L)$ time via FFT as the whole of y is known beforehand, during inference this is non-trivial. Suppose there is a prompt of size P and we want to generate $L - P$ more tokens. The bottleneck problem can be understood as computing, for every layer and every dimension, the below:

$$z_t \triangleq \sum_{i=1}^{t} y_i \cdot \rho_{t-i} \tag{2}$$

for all $1 \le t \le L$. To pre-fill the first P values $z_{[1..p]}$ at all levels of activations, since the first P inputs are known, one can perform FFT as they would normally do during train-time (incurring an $O(P \log P)$ time cost), but past P^{th} position it is important to note that y_i is not available before computing z_{i-1} - this is where *static* FFT breaks. Since dealing with the prompt can be done easily [Massaroli et al., 2024, Lemma 2.1] by essentially filling in all contributions of $y_{[1..P]}$ to $z_{[1..L]}$ and then forgetting the prompt ever existed, we henceforth assume $P = 0$.

2.3.2 Previous Work on Efficient Inference

Speeding up Equation 2 from the naive $\Omega(L^2)$ is the object of study of Massaroli et al. [2024]: they optimize the process by finding a *low-dimensional* LTI SSM whose equivalent convolution filter closely resembles k, thus getting an RNN-like problem to simulate. If the learnt SSM has size D', then this yields an $\Theta(LDD')$ algorithm for generating L tokens. This has the significant added benefit of not needing to store the activations (or even inputs) thus far which makes it memory efficient and practically time-efficient.

The downside, however, is that by definition this will only be an approximation of the learnt filter. More importantly, this approximation is a projection to a fundamentally smaller space of models, thus defeating the purpose of using LCSMs instead of LTI SSMs - it is only a different training procedure for an LTI SSM model. Furthermore, this approach assumes the filter k is data-independent - this is needed in order to undergo an expensive distillation procedure as a precomputation step. This can be qualitatively limiting as shown in Arora et al. [2023]. While we do store all activations, our proposed framework exactly simulates the architecture and data-dependent filters can also be accommodated.

3 The Flash Inference Framework

We propose a framework called Flash Inference to address the inference efficiency limitations of Hyena and other architectures sharing a few basic properties. The framework's main building tool is that of "tiling" - accounting for the influence of a range of inputs to a range of outputs all at once.

3.1 Assumptions

In order for our framework to apply, even partially, we need to assume a certain shape of the mixers involved:

A.1 Contribution-based The used mixers work by aggregating contributions of each input position to each subsequent output position. That is, for any mixer_ℓ, there exist an *associative* aggregation function $\mathrm{agg} : \mathcal{X}^* \to \mathcal{X}$, a read function $\mathrm{read} : \mathcal{X} \to \mathbb{R}^D$ and a contribution function *cont* : $\mathbb{R}^D \times \mathbb{N} \times \mathbb{N} \to \mathcal{X}$ such that:

$$\mathrm{mixer}(y)_i = \mathrm{read}(\mathrm{agg}(\mathrm{cont}(y, 1, i), \mathrm{cont}(y, 2, i), \ldots \mathrm{cont}(y, i, i))) \tag{3}$$

where, $\mathcal{X}$ is a set of intermediate states and read is a function to map those back to embeddings. For a sensible architecture, the size of $\mathcal{X}$ and cost of agg should be of order D and by agg being associative, we mean that $\mathrm{agg}(x_1, x_2, \ldots x_\tau) = \mathrm{agg}(\mathrm{agg}(x_1, \ldots x_i), \mathrm{agg}(x_{i+1}, \ldots x_\tau))$ for any $1 \le i < \tau$.

In the case of self-attention this translates to having $\mathrm{read} \circ \mathrm{agg}$ simulate the softmax, by letting $\mathcal{X} = \mathbb{R}^D \times \mathbb{R}$, $\mathrm{agg} = \sum$ and

$$\mathrm{cont}(y, i, j) = (V y_i \cdot e^{y_j^\top Q K^\top y_i}, e^{y_j^\top Q K^\top y_i}) = (v_i \cdot e^{\langle k_i, q_j \rangle}, e^{\langle k_i, q_j \rangle})$$

that is, the exponentially weighted value vector along with the exponential weight. Finally one can use $\mathrm{read}(v, w) = v/w$ to implemnent the softmax normalization step.

In the case of LCSMs, one can simply choose $\mathcal{X} = \mathbb{R}^D$, read be the identity function, agg be the sum again and $\mathrm{cont}(y, i, j)_c = y_{i,c} \cdot k_{j-i,c}$.

A.2 Query-independent The contribution function $\mathrm{cont}(y, i, j)$ does not depend on $y_{[i+1,L]}$. Note that this is the case in LCSMs since $y_{i,c} \cdot k_{j-i,c}$ only depends on y_i.

3.2 Setting and Definitions

Suppose **A.1** holds and there exists an algorithm $\mathcal{A}$ that for any given input sequence y and indices $l \le r < l' \le r'$, computes the contributions of $y_{[l,r]}$ to outputs at every position $p \in [l', r']$:

$$\mathcal{A}(y, l, r, l', r')_{l' \le p \le r'} = \mathrm{agg}(\mathrm{cont}(y, l, p), \mathrm{cont}(y, l+1, p), \ldots \mathrm{cont}(y, r, p)).$$

Furthermore, for this choice of $\mathcal{A}$, let $\mathcal{T} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ such that for any $l \le r < l' \le r'$, evaluating $A(y, l, r, l', r')$ takes at most $\mathcal{T}(r - l + 1, r' - l' + 1)$ FLOPs.

If we dropped the $r < l'$ condition and set $l = l' = 1$ and $r = r' = L$, this algorithm would actually represent the procedure one needs to perform a forward pass during training - which we refer to as the *static* setting. In the case of LCSM's, FFT can then be used to get a runtime of $O(DL \log L)$. The naive implementation of the specification of $\mathcal{A}$ would yield a time complexity $\mathcal{T}$ given by $\mathcal{T}(L_1, L_2) = DL_1 \cdot L_2$, but it turns out that one can employ FFT as well:

Lemma 1. *If we are in an LCSM mixer setting, that is if* $\mathrm{cont}(y, i, j)_c = y_{i,c} \cdot k_{j-i,c}$*, then there exists an FFT-based implementation of* $\mathcal{A}$ *with* $\mathcal{T}$ *given by* $\mathcal{T}(L_1, L_2) = O(D \cdot (L_1 + L_2) \log(L_1 + L_2))$.

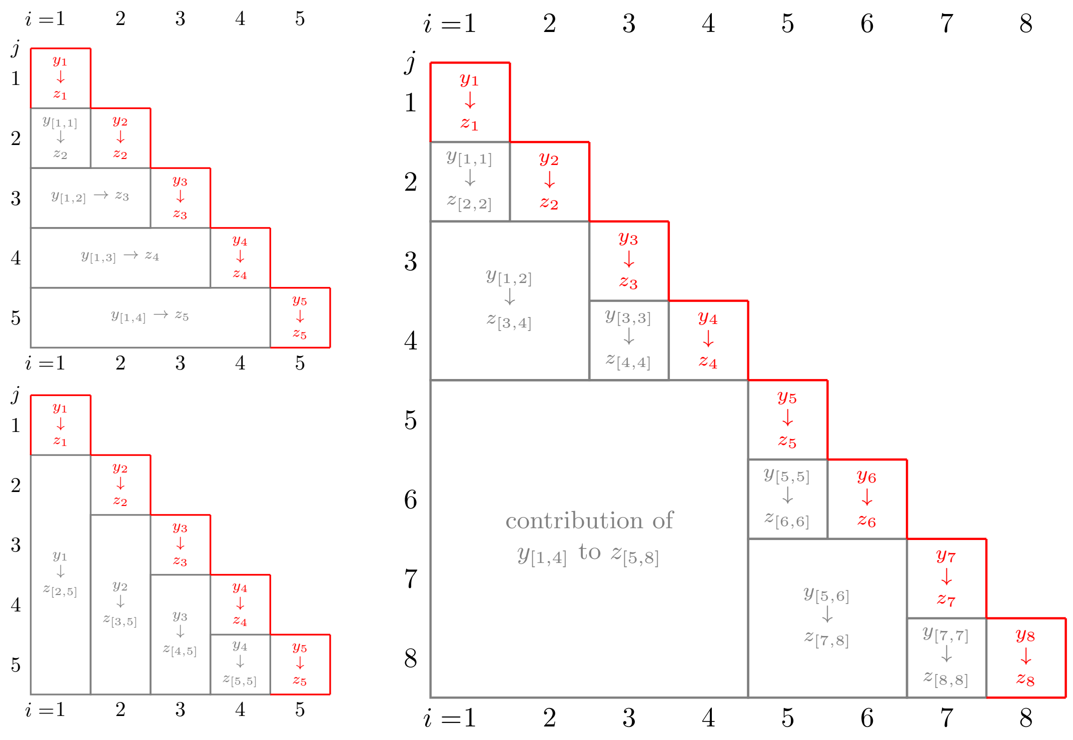


Figure 1: Cell (i, j) corresponds to the contribution of u_i to z_j. To compute z_j, all its line of contributions should've been accounted for. *(Left Top)* represents the standard (lazy) approach, *(Left Bottom)* represents the eager approach, and *(Right)* represents our suggested method.

3.3 Main Result

Our main result can be phrased as follows:

Theorem 2. *Under the assumptions **A.1** and **A.2**, one can generate $L = 2^P$ tokens autoregressively by performing, per layer, $L-1$ black-box calls to $\mathcal{A}$ as well as L more calls to* cont, agg, read *and* block*. Concretely, there are $L/2 = 2^{P-1}$ calls to $\mathcal{A}$ of length 1 each, $L/4 = 2^{P-2}$ calls of length 2 each and so on up to 1 call of length $L/2$. Hence, neglecting the* cont, agg, read *and* block *part, the overall time complexity per mixer layer is:*

$$FLOPs = \sum_{q=0}^{P-1} 2^{P-1-q}\mathcal{T}(2^q, 2^q)$$

Furthermore, during every token-generation iteration, the calls to $\mathcal{A}$ across different layers can be performed in parallel as there is no data-dependency between them.

Here, we are being ignorant of the L calls to cont, agg, read and block because they only scale linearly with L - however the MLPs scale quadratically in D and therefore these can be the limiting factor when D is large in comparison to L.

Corollary 1. *In the case of LCSMs, by substituting $\mathcal{T}(L, L)$ with $O(D \cdot L \log L)$ as per Lemma 1, we get a per mixer-layer time complexity of $O(D \cdot P^2 2^P) = O(D \cdot L \log^2 L)$.*

3.4 The Proposed Algorithm

We derive inspiration from the the work of van der Hoeven [1997] regarding relaxed polynomial interpolation - they propose a fix to dealing with *dynamic* structure of 2 to achieve an overall $O(L \log^2 L)$ times. While their work is specific to convolutions (and thus can only translate to LCSMs), their approach can be generalized to act as a static-to-dynamic translator. This comes at the price of blowing up FLOPs by just $\log L$ over the static procedure.

Algorithm 1 Flash Inference for LCSMs

```
Require: filter $k \in \mathbb{R}^{M \times L \times D}$, $\text{block}_{[1,M]}$, first token $a_{1,0}$ and sampler
 1: Output: All activations $a_0, \dots, a_M \in \mathbb{R}^{L \times D}$ obtained by autoregressively sampling
 2: Initialize $a$ with zeros outside $a_{1,0}$
 3: for $i \leftarrow 1$ to $L - 1$ do
 4:     $U \leftarrow$ maximum power of 2 that divides $i$  # the side of the gray $i^{\text{th}}$ tile
 5:     for $\ell \leftarrow 1$ to $M$ do
 6:         # account for the contribution of $a_{\ell-1,i}$ to $a_{\ell,i}$ - red cell
 7:         $a_{\ell,i} = \text{block}_\ell(a_{\ell,i} + a_{\ell-1,i} * k_{\ell,0})$
 8:         # account for the contribution of $a_{\ell-1,[i-U+1,i]}$ to $a_{\ell,[i,i+U]}$ - gray tile
 9:         $a_{\ell,[i+1,i+U]} \mathrel{+}= \mathcal{A}(a_{\ell-1}, i-U+1, i, i+1, i+U)$
10:     end for
11:     # generate next token based on the output of last layer at position $i$
12:     $a_{0,i+1} = \text{sampler}(a_{M,i})$
13: end for
```

In describing the method, we will constrain ourselves to the LCSM setting, but all of the reasoning can be applied more broadly. The high-level idea is to tile the contributions into chunks that can be dealt with by calls to $\mathcal{A}$. However, as mentioned in Section 2.3.1, we need to be careful about the inputs being fully available at the time of calling $\mathcal{A}$ and this is the main problem to solve.

The crux of the algorithm is exploiting **A.2**: particularly, one can start accounting for $\text{cont}(y, i, j)$ as soon as y_i becomes available - this is in contrast to transformers where $\text{cont}(y, i, j)$ depends on q_j which depends on y_j. Hence, one can do work ahead of time, at the very extreme accounting for $\text{cont}(y, i, j)$ for all $j \geq i$ as soon as y_i is computed. This eager approach, however, still scales quadratically in L - the issue is that we are essentially incurring a cost of $\mathcal{T}(1, L - i + 1)$. The standard (lazy) implementation, which at time j iterates through all $i \leq j$ and accounts for their contribution essentially incurs a $\mathcal{T}(j, 1)$ cost. As noted in Lemma 1, by balancing the two inputs of $\mathcal{T}$, one can use FFT to lower $\mathcal{T}(L_1, L_2)$ significantly below $L_1 \cdot L_2$ and this yields biggest improvements when L_1 and L_2 are balanced. This motivates the fractal tiling from Figure 1. This tiling is different from the one proposed by van der Hoeven [1997] and saves a factor 2 by exploiting the fact that k is data-independent. However, as explored in Appendix B, one can use van der Hoeven [1997]'s approach to work with data-dependent filters.

The algorithm is illustrated in Algorithm 1. The outer loop (3) iterates through positions i: at each iteration, one computes all the activations at position i, does some (limited) eager work and samples next token. To do so, the inner loop (5 iterates through the several mixer-layers. It first accounts for the red cells in the diagram, that is, the direct dependency on previous layer's activations at position i (line 7). After computing the activation at position i, it also accounts for the contribution of the gray tile that has just been unlocked (line 9) at the current layer - this is only useful to save compute in the long term, while not being immediately necessary to sample the next token. The accounting works for considering (in-place) the influence of last U positions to next U ones. For example, during iteration $i = 6$, following the Figure 1, the gray cell has $5 \leq i \leq 6$ and $7 \leq j \leq 8$, corresponding to $U = 2$ defined as largest power of 2 that divides $i = 6$. Importantly, one can only account for a tile (red or gray) once all its i's are known at the previous layer.

3.5 Across-layer Parallelization

One important feature of the proposed method is that it allows for higher parallelization across layers. In particular, the gray lines can be taken out of the inner loop and performed in parallel across all layers at once, just after the loop - this is because all their inputs and outputs are disjoint. However, the red lines make need to be performed sequentially since $a_{\ell,i}$ is a function of $a_{\ell-1,i}$ for every ℓ.

3.6 Memory Considerations

As noted in Theorem 2, the overall size of the inputs to $\mathcal{A}$ in the proposed algorithm is only $O(L \log(L))$ - that is, we access on average activation values at $\log(L)$ positions as opposed to the naive implementations (such as lazy and eager approaches) that access $\Omega(L)$ positions on average. This directly translates to data movement improvements by a factor of up to $L/\log L$.

Furthermore the static memory overhead is minimal: at least in the LCSM case, one directly aggregates contributions to the same activation tensor, thus not using any extra storage. Hence, the only extra cost we pay is in the peak memory usage which is given by the largest tile - this is $O(MLD)$, but can be dropped to $O(LD)$ at essentially no cost by dropping parallelization for large tiles, as further discussed in Appendix E.

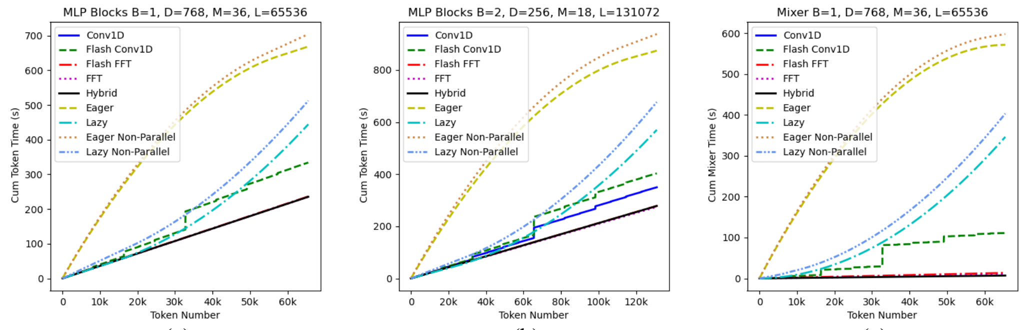
Lastly, if one is restricted to only using one GPU and the whole tensor of $M \times L \times D$ cannot fit it, it is possible to drop the requirement to only storing $M \times (L/2) \times D$ activations by paying some computational cost. This is further discussed in Appendix D.

4 Experiments

Throughout our experiments, we assumed the block's to be MLPs with a hidden dimension of $2D$ and GELU activation. Thus the computation scales as $O(MLD^2)$ in the block part. The eager and lazy approaches scale as $O(ML^2D)$ in the mixer part whereas our method scales as $O(MDL\log^2 L)$. These are FLOP counts, but practically hardware implications along with parallelization can greatly impact the end-to-end time.

In all our experiments, the weights of our model are initialized to noise and instead of sampling we just take the next $(i+1)^{\text{th}}$ token embedding to be $a_{M,i}$ plus some noise to avoid dependency on vocabulary size since that is out of the scope of our framework and will be negligible at scale.

In terms of notation, we introduce the batch dimension B and while M, D, L correspond to the number of layers, embedding dimension and number of tokens generated, respectively. We use U to refer to the square-tile length, as per line 4. We sweep over $B \in \{1, 2, 4, 8\}, M \in \{18, 36\}$, $D \in \{256, 768\}$ and generate up to $L \in \{2^{16}, 2^{17}\}$ tokens. All results are obtained by averaging over 10 runs following 3 runs of warm-up. We evaluate our approach on the latest Nvidia A-100 tensor core GPU with 312 peak TFLOPS for FP16, 2TB/s HBM2e memory with 80GB capacity.

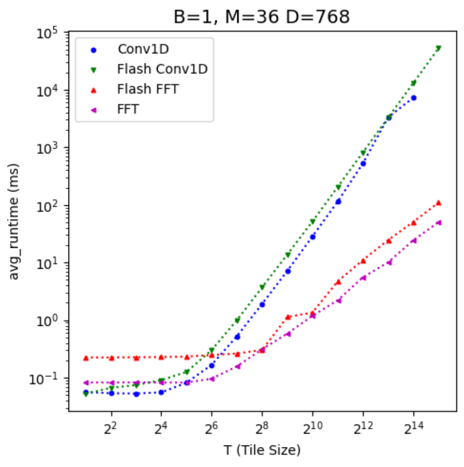


(a) (b) (c)

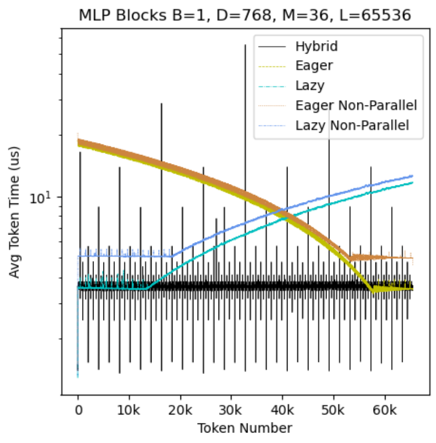
Figure 2: (ab)All our $\mathcal{A}$ implementations demonstrate significant speed-ups of **1.5-4**$\times$ over Lazy, Eager for all combinations of B, M, and D, both for MLP and non-MLP gating blocks. (c) When measuring only the time spent processing doing mixer-related computations, our methods outperform the naive ones by a factor of 10 - our methods become dominated by the block part.

For our baselines, (1) we consider the eager and lazy approaches described in 3.4, depicted in 1. We also consider their implementation exploiting our observation regarding parallelization across layers (3.5) (we denote the baselines as non-parallel and the parallel versions simply as lazy and eager).

Our Fast Inference framework explored various implementations of $\mathcal{A}$, explained in Section 4.1. Our best method is a *Hybrid* that dynamically chooses the optimal implementation of $\mathcal{A}$ depending on (B, D, M, U). Figures 2a, 2b show the cumulative time per token time for all our methods against the baselines, where *Hybrid* out performs the base lines consistently by **1.5-4** $\times$ in the end-to-end latency, for various combinations of (B, D, M, U) and two different types of gating. We show two



(a)



(b)

Figure 3: (a) Different implementations of $\mathcal{A}$ are optimal for different tile sizes creating a pareto optimal curve for Hybrid to choose, (b) Hybrid shows low variance in per-token response time compared to Lazy and Eager except at the tokens where large tiles are computed.

representative settings of (B, M, D) but we obtained consistent results throughout all our sweep (available in Supplementary Material).

4.1 $\mathcal{A}$ Implementations and Optimizations

Algorithm 1 assumes an implementation of primitive $\mathcal{A}$ that accounts for the contribution of a range of inputs to a range of outputs of a convolution. We considered 7 different implementations but, here, we only present results of the ones on the Pareto Frontier - that is, those optimal for at least some (B, N, D, U) setting. There are 4 such implementations, of two types:

(1) **Depthwise-separable 1D Convolution:** We use two types of implementations (a) *Conv1D* refers to the default PyTorch [Paszke et al., 2019] kernel, that requires explicit padding (2) *Flash Conv1D* refers to fused kernel by FlashFFTConv [Fu et al., 2023b] - these are asymptotically quadratic in tile size U.

(2) **FFT based convolution** We again use two implementations (a) PyTorch native, shown as *FFT*, that requires computing the FFTs of inputs, followed by pointwise multiplication and lastly performing the inverse FFT operation. (b) the second is provided by FlashFFTConv that does all the above in a single fused kernel shown as *FlashFFT*- both these scale quasilinearly in U.

Optimizations:

(1) Since Conv1D is implemented using the cross-correlation operator it requires the convolution kernel to sliced to the convolution length and flipped on the sequence dimension. We precompute this for all $\log(L)$ tile sizes and cache them.

(2) For PyTorch based FFT we precompute the FFT for the convolution kernel for all $\log(L)$ tile sizes.

(3) The calls to $\mathcal{A}$ are always performed parallelly across the M layers - hence, we measure the time cost as a function of B, D, M, U since these are the variables defining the performance of dealing with the M gray tiles for an iteration.

4.2 Hybridization

We evaluate the different convolution implmentations for all settings of B, D, M, U and observe that each of these four implementation lies on the pareto frontier curve of tile size vs latency as shown in Figure 3a. Our *Hybrid* approach dynamically chooses the best $\mathcal{A}$ implementation for a given tile size U based on the isolated empirically-measured efficiency of each implementation. Consequently, it is the best-performing method as seen in Figure 2.

4.3 Summary

This significant speed-up can be attributed to:

(1) The significantly lower $O(L \log^2 L)$ FLOPs required for our tile computation approach compared to the $\Omega(L^2)$ FLOPs required by *Eager* and *Lazy* counterparts. This is shown in Figure 2c where methods based on our tiling outperform by a large constant the quadratic methods in terms of time spent on the mixer components.

(2) Drastically reduced activation memory access from $\Omega(L^2)$ for *Eager* and *Lazy* to $O(L \log L)$ *Hybrid*. This can be shown through the performance of *Flash Conv1D* which outperforms lazy and eager by a margin although it also performs $\Omega(L^2)$ FLOPs - it does so in a more memory-friendly and kernel-optimizable way.

(3) Parallel tile computation across M layers by all our $\mathcal{A}$ implementations. One can notice improvements of $10 - 20\%$ in the Eager and Lazy implementations alone.

(4) The dynamic choice of best $\mathcal{A}$ for given tile U - hybrid outperforming any method using a fixed implementation.

(5) Implementation specific optimizations described in Section 4.1.

5 Conclusion and Further Work

We propose a framework for performing inference in certain autoregressive sequence models. Among such models, LCSMs such as Hyena, are noteworthy: there, our framework provides an $O(L \log^2 L)$ inference algorithm which, when run empirically, yields up to $\times 4$ time-efficiency improvement. The framework exploits a causal, fractal tiling that helps save on data movement and share computation. Moreover, it allows for almost-complete across-layers parallelization of mixer-related workload.

An interesting future direction to pursue is that of designing architectures that fit our framework requirements and thus get fast-inference by construction. Furthermore, in the class of LCSMs, we have noted that one can achieve the same theoretical complexity if filters are made data-dependent, and, while previous works Arora et al. [2023], Karami and Ghodsi [2024] have shown the potential for these, they are not yet causal so looking into how to make filters data-dependent in a causal way is another promising direction.

6 Acknowledgments

This work was partially enabled by use of the Kempner AI cluster. Sham Kakade acknowledges: this work has been made possible in part by a gift from the Chan Zuckerberg Initiative Foundation to establish the Kempner Institute for the Study of Natural and Artificial Intelligence. SK and CO also acknowledge support from the Office of Naval Research under award N00014-22-1-2377, and the National Science Foundation Grant under award #IIS 2229881.

References

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

D. Fu, S. Arora, J. Grogan, I. Johnson, E. S. Eyuboglu, A. Thomas, B. Spector, M. Poli, A. Rudra, and C. Ré. Monarch mixer: A simple sub-quadratic gemm-based architecture. *Advances in Neural Information Processing Systems*, 36, 2024.

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

D. Y. Fu, E. L. Epstein, E. Nguyen, A. W. Thomas, M. Zhang, T. Dao, A. Rudra, and C. Ré. Simple hardware-efficient long convolutions for sequence modeling. In *International Conference on Machine Learning*, pages 10373–10391. PMLR, 2023a.

D. Y. Fu, H. Kumbong, E. Nguyen, and C. Ré. Flashfftconv: Efficient convolutions for long sequences with tensor cores. *arXiv preprint arXiv:2311.05908*, 2023b.

M. Gaido, S. Papi, M. Negri, and L. Bentivogli. How do hyenas deal with human speech? speech recognition and translation with confhyena. *arXiv preprint arXiv:2402.13208*, 2024.

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

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

M. Karami and A. Ghodsi. Orchid: Flexible and data-dependent convolution for sequence modeling. *arXiv preprint arXiv:2402.18508*, 2024.

Y. Li, T. Cai, Y. Zhang, D. Chen, and D. Dey. What makes convolutional models great on long sequence modeling? *arXiv preprint arXiv:2210.09298*, 2022.

S. Massaroli, M. Poli, D. Fu, H. Kumbong, R. Parnichkun, D. Romero, A. Timalsina, Q. McIntyre, B. Chen, A. Rudra, et al. Laughing hyena distillery: Extracting compact recurrences from convolutions. *Advances in Neural Information Processing Systems*, 36, 2024.

A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

M. Poli, S. Massaroli, E. Nguyen, D. Y. Fu, T. Dao, S. Baccus, Y. Bengio, S. Ermon, and C. Ré. Hyena hierarchy: Towards larger convolutional language models. *arXiv preprint arXiv:2302.10866*, 2023.

D. W. Romero, A. Kuzina, E. J. Bekkers, J. M. Tomczak, and M. Hoogendoorn. Ckconv: Continuous kernel convolution for sequential data. *arXiv preprint arXiv:2102.02611*, 2021.

Y. Tay, M. Dehghani, D. Bahri, and D. Metzler. Efficient transformers: A survey. *ACM Computing Surveys*, 55(6):1–28, 2022.

J. van der Hoeven. Lazy multiplication of formal power series. In *Proceedings of the 1997 international symposium on Symbolic and algebraic computation*, pages 17–20, 1997.

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

A Proofs

Lemma. *If we are in an LCSM mixer setting, that is if* $\mathrm{cont}(y, i, j)_c = y_{i,c} \cdot k_{j-i,c}$*, then there exists an FFT-based implementation of* $\mathcal{A}$ *with* $\mathcal{T}$ *given by* $\mathcal{T}(L_1, L_2) = O(D \cdot (L_1 + L_2)\log(L_1 + L_2))$.

Proof. For each dimension independently, consider performing a convolution between $y_{[l,r]}$ and $k_{[l'-r,r'-l]}$. It holds that for each $j \in [l', r']$ and $i \in [l, r]$, $j - i$ is in the range of k. On the other hand, truncating the output of the convolution appropriately, one can keep only the corresponding outputs for each $j \in [l', r']$. The overall size of the FFT is then given by the sum of the lengths, which is at most $r-l+1+(r'-l)-(l'-r)+1 = 2(r-l+1)+(r'-l'+1)-1 = O((r-l+1)+(r'-l'+1))$. Since an FFT of order L runs in $O(L \log L)$ and we do this independently for each dimension $1 \leq c \leq D$, the conclusion follows. □

Theorem. *Under the assumptions* ***A.1*** *and* ***A.2****, one can generate* $L = 2^P$ *tokens autoregressively by performing, per layer,* $L - 1$ *black-box calls to* $\mathcal{A}$ *as well as* L *more calls to* cont, agg, read *and* block. *Concretely, there are* $L/2 = 2^{P-1}$ *calls to* $\mathcal{A}$ *of length* 1 *each,* $L/4 = 2^{P-2}$ *calls of length* 2 *each and so on up to* 1 *call of length* $L/2$*. Hence, neglecting the* cont, agg, read *and* block *part, the overall time complexity per mixer layer is:*

$$\mathit{FLOPs} = \sum_{q=0}^{P-1} 2^{P-1-q}\mathcal{T}(2^q, 2^q)$$

Furthermore, during every token-generation iteration, the calls to $\mathcal{A}$ *across different layers can be performed in parallel as there is no data-dependency between them.*

Here, we are being ignorant of the L calls to cont, agg, read and block because they only scale linearly with L - however it is worth mentioning that when d is large the MLPs scale quadratically in d and therefore these can be the limiting factor for L's up to a point.

Proof. Figure 1 shows why the tiling used in Algorithm 1 covers every pair of contributions exactly once and in the correct order (remember, we only assumed agg to be associative, not necessarily commutative). The more general algorithm is illustrated in Algorithm 2 and follows the same shape as its LCSM counterpart. The only difference is that $\mathcal{X}$ is not assumed to be $\mathbb{R}^D$ necessarily, so one cannot store the intermediate accumulated states in the same place they store the activations (though they could reuse some of the space since only $a_{\ell,i}$ and $x_{\ell,i}$ never need to co-exist). We use $x_{\ell,i}$ to store the inner part of Equation 3, namely $x_{\ell,i}$ incrementally computes $\mathrm{agg}(\mathrm{cont}(a_{\ell-1}, 1, i), \mathrm{cont}(a_{\ell-1}, 2, i), \dots \mathrm{cont}(a_{\ell-1}, i, i))$. Finally, we explicitly moved the gray tile calls to happen after the inner loop and in parallel - this simply shows the last point of the theorem, namely that the calls to $\mathcal{A}$ can be done parallelly across layers.

Algorithm 2 Generic Flash Inference

```
Require: Functions $\mathcal{A}$, cont, agg, read, $\mathrm{block}_{[1,M]}$, first token $a_{1,0}$ and sampler
1:  Output: All activations $a_0, \dots, a_M \in \mathbb{R}^{L \times D}$ obtained by autoregressively sampling
2:  Initialize $x_1, \dots x_M : \mathcal{X}^{L \times D}$ with elements neutral to agg
3:  for $i \leftarrow 1$ to $L-1$ do
4:      $U \leftarrow$ maximum power of 2 that divides $i$  # the side of the $i^{\text{th}}$ gray tile
5:      for $\ell \leftarrow 1$ to $M$ do
6:          # account for the contribution of $a_{\ell-1,i}$ to $x_{\ell,i}$ - red cell
7:          $x_{\ell,i} = \mathrm{agg}(x_{\ell,i}, \mathrm{cont}(a_{\ell-1}, i, i))$
8:          $a_{\ell,i} = \mathrm{block}_\ell(\mathrm{read}(x_{\ell,i}))$
9:      end for
10:     # account for the contribution of $a_{\ell-1,[i-U+1,i]}$ to $x_{\ell,[i+1,i+U]}$ - gray tile
11:     parallelly across $\ell \leftarrow 1$ to $M$ do:
12:         $x_{\ell,[i+1,i+U]} = \mathrm{agg}(x_{\ell,[i+1,i+U]}, \mathcal{A}(a_{\ell-1}, i-U+1, i, i+1, i+U))$
13:     # generate next token based on the output of last layer at position $i$
14:     $a_{0,i+1} = \mathrm{sampler}(a_{M,i})$
15: end for
```

Only part remaining to be proved is the right counting of calls to $\mathcal{A}$ per layer: calls of length 2^r happen whenever 2^r divides i but 2^{r+1} does not - that is, for $i \in \{2^r, 3 \cdot 2^r, 5 \cdot r^2, \dots (2^{P-r}-1) \cdot 2^r\}$, so 2^{P-r-1} calls. □

Corollary. *In the case of LCSMs, by substituting* $\mathcal{T}(L, L)$ *with* $O(D \cdot L \log L)$ *as per Lemma 1, we get a per mixer-layer time complexity of* $O(D \cdot P^2 2^P) = O(D \cdot L \log^2 L)$*.*

Proof. By applying Theorem 2, we get:

$$\mathrm{FLOPs} = \sum_{q=0}^{P-1} 2^{P-1-q}\mathcal{T}(2^q, 2^q) = O\left(\sum_{q=0}^{P-1} 2^{P-1-q} D \cdot 2^q q\right) = O\left(\sum_{q=0}^{P-1} D \cdot 2^{P-1} \cdot q\right) =$$
$$O(D \cdot P^2 2^P) = O(D \cdot L \log^2 L)$$

□

B Extension to Data-Dependent Filters

The reason why data-dependent filters are harder to deal with is that $\mathrm{cont}(y, i, j) = y_i \cdot k_{j-i}$ depends on both y_i and on what k_{j-i} depends on. By only assuming causality, we can only access k_{j-i} after y_{j-i} has been computed. This stops Algorithm 1 from working out-of-the-box since when i is a power of 2, in order to account for the contribution of $a_{\ell-1,[1,i]}$ to $a_{\ell,[i+1,2i]}$ one will need access to $k_{\ell,[1,2i-1]}$ which is not guaranteed to be accessible (only $k_{\ell,[1,i]}$ can be assumed to be known).

The modified algorithm is shown in Algorithm 3 and precisely follows the tiling of van der Hoeven [1997] and, thus, its correctness transfers. Note that rather than using the $\mathcal{A}$ algorithm, it directly uses the untruncated convolution (implementable via FFT) - in the contribution space this corresponds to a parallelogram tile rather than rectangle - and this cannot be simulated via a rectangle, although the other way around is possible. This implementation performs convolutions between 2 sequences of length U each and uses the whole output and thus requires an order $2U$ FFT. Note that this happens twice for a given i when $i + 1$ is not a power of 2 (almost always). Algorithm 1, on the other hand, only performs a convolution between a length-U sequence and a length-$2U$ sequence. However, as noted in Appendix C, we can get away without padding and thus performing only one order $2U$ FFT. Thus, our proposed tiling improves FLOPs by a factor of 2, but it requires the kernel to be data-independent.

Algorithm 3 Flash Inference for LCSMs with data-dependent filters

```
Require: first token $a_{\cdot,0}$, $k_{\cdot,0}$, $\text{block}_{[1,M]}$ and sampler
 1: Output: All activations $a_0, \ldots, a_M \in \mathbb{R}^{L\times D}$ obtained by autoregressively sampling
 2: Initialize $a$ with zeros outside $a_{\cdot,0}$
 3: for $i \leftarrow 1$ to $L-1$ do
 4:     $U \leftarrow$ maximum power of 2 that divides $(i+1)$
 5:     for $\ell \leftarrow 1$ to $M$ do
 6:         Compute $k_{\ell,i}$ as a causal function of data $a_{\ell-1,[1,i]}$ as per model specification
 7:         # account for the newly available contributions
 8:         $a_{\ell,i} = \text{block}_\ell(a_{\ell,i} + a_{\ell-1,i} * k_{\ell,0} + a_{\ell-1,0} * k_{\ell,i})$
 9:         # account for some eager contributions
10:         if $i+1 = U$ then
11:             # downgrade the power of 2 by half if $i+1$ is already a power of 2
12:             $U \leftarrow U/2$
13:             $a_{\ell,[2U,4U-2]} \mathrel{+}= CONV(a_{\ell-1,[U,2U-1]}, k_{\ell,[U,2U-1]})$
14:         else
15:             $a_{\ell,[i+1,i+2U-1]} \mathrel{+}= CONV(a_{\ell-1,[U,2U-1]}, k_{\ell,[i-U+1,i]})$
16:             $a_{\ell,[i+1,i+2U-1]} \mathrel{+}= CONV(k_{\ell,[U,2U-1]}, a_{\ell-1,[i-U+1,i]})$
17:         end if
18:     end for
19:     # generate next token based on the output of last layer at position $i$
20:     $a_{0,i+1} = \text{sampler}(a_{M,i})$
21: end for
```

C Implementation Improvements

In Algorithm1, the calls to $\mathcal{A}$ all have a shape of $\mathcal{A}(y, i-U+1, i, i+1, i+U)$ where U is a power of 2 that divides i. Following the proof of Lemma 1, we note that to implement $\mathcal{A}$, we need to perform the convolution of a segment of length U of y and a prefix of length $2U$ of k. However, following the convolution, of the $3U-1$ outputs, indexed $[0, 3U-2]$, we are only interested in the middle U of them, namely indices $[U, 2U-1]$. The canonical way of performing convolutions via FFT involves padding by enough $0s$ and using an order large enough to store the whole output which rounded up to the closest power of 2 would mean an order $4U$ FFT. However, using a $2U$ FFT call, which will perform a cyclical convolution, is enough for our purposes, since the values of interest are not affected by the cyclicity - that is, when folding outputs at $[2U, 3U-2]$ onto $[0, U-2]$, we do not intersect $[U, 2U-1]$. Furthermore, there are only $\log L$ different lengths of prefixes of k involved in these convolution so one could, in fact, cache the DFTs for these as a precomputation step, thus dropping the number of DFT's per convolution from 3 to 2, speeding up by a further $\times 1.5$ factor.

D Storing Only Half of the Activations

If the whole activation tensor ($M \times L \times D$) cannot fit the memory of one GPU, we can save a factor of 2 in the L-axis. To do this, observe that after $(L/2)^{\text{th}}$ iteration completes we never need to look back to any activation at position $i \leq L/2$. Hence, one can hope to reuse the space used for the first half to store the second half of activations. While doing this directly by having $\mathcal{A}$ work as an in-place operator may not be possible, one way to achieve this is by not applying $\mathcal{A}$ parallelly across all layers, but rather applying it sequentially for this largest tile - similarly in nature to gradient accumulation - always placing the output back into the input slot. This way, the peak extra memory required for the call to $\mathcal{A}$ does not scale with M (though it exceeds $L \times D$).

As we get the output of $\mathcal{A}(a_0, 1, L/2, L/2+1, L)$, which has a shape of $(L/2) \times D$, we overwrite a_0 by it and then move on to the next layer. We can do this because the contribution of $a_{0,i}$ to $a_{1,i}$ has already been properly accounted for so the first half of a_0 truly becomes irrelevant for next layers and iterations to come. Hence, a_0 contains the intended second half of a_1. We then proceed to do the same thing for a_1 and so on. At the end, $a_{[0 \ldots M-1]}$ will contain the second halves of $a_{[1 \ldots M]}$, so we can shift these by 1, emptying a_0 and now finally having a represent the second halves of activations - the first halves have been discarded.

Although per layer, one will have a peak memory containing both a_ℓ, the output, and whatever other necessary temporary values needed to perform $\mathcal{A}$, this does not scale with M - because we essentially reuse the extra space needed by $\mathcal{A}$ across different layers. We seemingly pay the cost of not performing the calls to $\mathcal{A}$ in parallel across layers, but if storing the $M \times L \times D$ tensor of activations was an issue, then one would very likely have to make this sort of compromise to be able to run $\mathcal{A}$ in the first place.

E Discussion on memory

In Section 3.6, we discussed the peak memory usage being given by the gray calls to $\mathcal{A}$. Since each call to $\mathcal{A}$ performs D FFTs of length L each (when dealing with the biggest tile of side $L/2$), the naive implementation would use up to $MD \cdot L$ extra memory to perform these FFT calls. However, this assumes the biggest gray tile is being processed in parallel across all layers and across all dimensions at once. We do not need to maximize parallelization on all sizes of tiles: for the largest ones, we can process the tiles at different layers sequentially by reusing the space, thus dropping to an overhead of $O(LD)$, or even just $O(L)$ if one is to treat sequentially the different dimensions. Dropping parallelization could theoretically yield a slowdown, but that is only when we are not memory bandwidth-bound which is the case if one has to worry about allocating more than $O(LD)$ - in that case, the FFT calls are anyway not happening in parallel. Hence, one can easily drop the peak memory overhead to $O(LD)$ or even $O(L)$ without incurring an actual time cost.