Systems and Algorithms for Convolutional Multi-Hybrid Language Models at Scale

Jerome Ku[2,*], Eric Nguyen[1,*], David W. Romero[3,*], Garyk Brixi[1], Brandon Yang[5], Anton Vorontsov[3], Ali Taghibakhshi[3], Amy X. Lu[4], Dave P. Burke[2], Greg Brockman[5,†], Stefano Massaroli[7,8], Christopher Ré[1], Patrick D. Hsu[2,4], Brian L. Hie[1,2], Stefano Ermon[1], Michael Poli[1,7,‡]

[1]Stanford University, [2]Arc Institute, [3]NVIDIA,
[4]University of California, Berkeley, [5]Independent Researcher, [7]Liquid AI, [8]RIKEN

Abstract

We introduce *convolutional multi-hybrid* architectures, with a design grounded on two simple observations. First, operators in hybrid models can be tailored to token manipulation tasks such as in-context recall, multi-token recall, and compression, with input-dependent convolutions and attention offering complementary performance. Second, co-designing convolution operators and hardware-aware algorithms enables efficiency gains in regimes where previous alternative architectures struggle to surpass Transformers. At the 40 billion parameter scale, we train end-to-end 1.2 to 2.9 times faster than optimized Transformers, and 1.1 to 1.4 times faster than previous generation hybrids. On H100 GPUs and model width 4096, individual operators in the proposed multi-hybrid StripedHyena 2 architecture achieve two-fold throughput improvement over linear attention and state-space models. Multi-hybrids excel at sequence modeling over byte-tokenized data, as demonstrated by the Evo 2 line of models. We discuss the foundations that enable these results, including architecture design, overlap-add blocked kernels for tensor cores, and dedicated all-to-all and point-to-point context parallelism strategies.

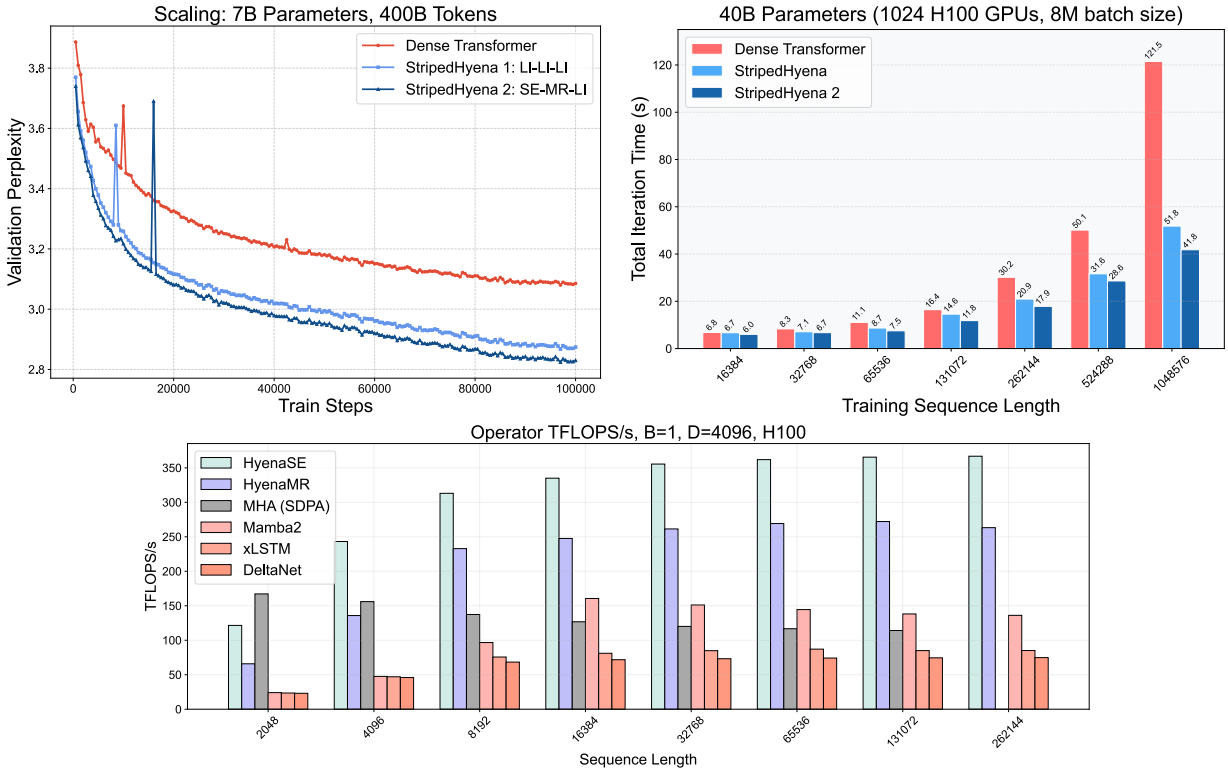


Figure 1: Scaling experiments, showing differences in perplexity and throughput of Transformers, multi-hybrids (StripedHyena 2), and other alternative operators.

* These authors contributed equally to this work.
† Current address: OpenAI.
‡Corresponding author: poli@stanford.edu.

1 Introduction

Architecture improvements for training language models at scale can be broadly categorized into several main groups. Tweaks to the attention mechanism to reduce the size of the kv cache such as GQA, MQA, MLA, sliding window and linear attention (Shazeer, 2019; Brown et al., 2020; Vaswani et al., 2021; Katharopoulos et al., 2020; Ainslie et al., 2023; Liu et al., 2024). Changes to the model for numerical stability, resilience to outliers and quantization such as pre-norm, SwiGLU, QK normalization (Zhang & Sennrich, 2019; Xiong et al., 2020; Shazeer, 2020). Finally, modifications that improve model capacity or recall at longer context such as RoPE, MoE (Shazeer et al., 2017; Su et al., 2024). Despite the broad interest in architecture improvement, remarkably few proposals, outside of the aforementioned methods, have stood the test of validation at scale.

A different approach is to introduce new *classes* of input-dependent operators to the standard mix of layers (self-attention and feed-forward) layers and optimize their composition, resulting in *hybrid* architectures. Hybrids promise improvements on both quality and efficiency, and have been proposed in various domains and with various mixtures of operators, typically with some combination of convolution and attention (Dai et al., 2021; Poli et al., 2023b), linear attention and attention (Fu et al., 2022; Fathi et al., 2023; Lieber et al., 2024; Glorioso et al., 2024), or local attention and attention (Child et al., 2019; Beltagy et al., 2020). In language modeling at scale, hybrids of convolutions, linear attention and attention have been validated through dedicated scaling laws (Poli et al., 2024) and large-scale model releases (Glorioso et al., 2024; Nguyen et al., 2024; Team et al., 2024).

Despite being a promising alternative, hybrids based on operators such as linear attention or state-space models have struggled to replace baseline Transformers as the de facto standard for language modeling due to a variety of reasons. One limitation is that these fixed-state operators realize efficiency gains only when applied to very long sequences, which is also where they drastically underperform full self-attention (Arora et al., 2023; Jelassi et al., 2024). Compared to Transformers, these methods are generally slower at the usual pretraining training regime: shorter contexts with larger and wider models. Furthermore, most of these approaches have been developed with the explicit goal of matching self-attention performance on in-context recall over longer sequences, but have only been successfully deployed in hybrids due to quality gaps. This has introduced redundancy in architectures, as multiple operators are optimized for the same capability: in-context recall.

This paper explores a fundamentally different approach. We advocate for model architecture designs that are both hybridization-aware and hardware-aware, combining different types of operators with complementary capabilities and computational costs, across a range of input and model sizes. Our approach is motivated by work on synthetics (Akyürek et al., 2024) and mechanistic design (Poli et al., 2024), showing how different operators in hybrids can specialize to subtasks such as recall, compression, multi-query recall, and fuzzy recall. For example, input-dependent convolutions excel at filtering noise and performing multi-token recall, useful for modeling byte-level data, whereas attention is optimized for targeted recall of information across longer sequences. We introduce *multi-hybrids*, architectures that combine strengths of multiple operator types.

We focus on StripedHyena 2, the first example of convolutional multi-hybrid architecture for sequence modeling validated on a series of experiments at scale (40 billion parameters, 9 trillion tokens). StripedHyena 2 is based on three different types of input-dependent convolutional operators: (i.) short, explicitly-parametrized hyena operators that maximize hardware utilization, specializing in local multi-token recall, (ii.) medium-length, regularized hyena operators tailored to efficient modeling across hundreds of tokens, and (iii.) long, implicit hyena operators that aggregate information over the entire sequence. We describe the algorithmic foundations of convolutional multi-hybrids, focusing on architecture design, kernels and context parallelism algorithms. As a motivating example, we will use the experiments behind the Evo 2 line of models (Brixi et al., 2025), built on top of StripedHyena 2. Evo 2 40B is a state-of-the-art foundation model for genomics, trained on byte-tokenized (nucleotide) sequences.

Outline In Section 2, we introduce the basic design ideas and describe the three primary operators behind convolutional multi-hybrids. We then discuss composition, filter grouping for improved hardware utilization, and showcase scaling at the thousand GPU and 40 billion parameter scale. In Section 3, we focus on architecture and algorithm co-design. Using filter grouping, we adapt overlap-add algorithms (Burrus & Parks, 1985) to tensor cores, introducing our implementation of a two-stage blocked kernel. We measure the performance gains at short and long context compared to efficient attention implementations (FlashAttention3 (Shah et al., 2024), SDPA) and other alternative operators such as linear attention and state-space

models: Mamba2 (Dao & Gu, 2024), xLSTM (Beck et al., 2024) and DeltaNet (Yang et al., 2024). In Section 4, we develop custom context parallelism methods for the different types of convolutions in our models. We introduce both peer-to-peer and all-to-all algorithms, including new channel-pipelined variants and FFT-based methods.

2 Multi-Hybrid Model Architecture

Notation Unless specified otherwise, (input) sequences are denoted with $x \in \mathbb{R}^{\ell \times d}$. We use subscripts to index in the time dimension, and Greek superscripts to index in the space (or width) dimension. To keep the notation compact, we also occasionally omit summation signs for repeated indices in longer tensor contractions e.g., $y_t^\alpha = A^{\alpha\beta} x_t^\beta$ is shorthand for $y_t^\alpha = \sum_\beta A^{\alpha\beta} x_t^\beta$.

2.1 Basic Design

We consider input-dependent convolutional operators that adhere to the following structure from the original Hyena work (Poli et al., 2023a):

$$
\begin{aligned}
q_t^\alpha &= T_{tt'}^\alpha (x_{t'}^\beta W^{\beta\alpha}) \\
k_t^\alpha &= H_{tt'}^\alpha (x_{t'}^\beta U^{\beta\alpha}) \\
v_t^\alpha &= K_{tt'}^\alpha (x_{t'}^\beta P^{\beta\alpha}) \\
y_t^\alpha &= (q_t^\beta G_{tt'}^\beta k_{t'}^\beta v_{t'}^\beta) M^{\beta\alpha}
\end{aligned}
\tag{1}
$$

where $T, H, K, G \in \mathbb{R}^{d \times \ell \times \ell}$ are Toeplitz matrices (corresponding to the convolution with the respective filters h_T, h_H, h_K, h_G), and $W, U, P, M \in \mathbb{R}^{d \times d}$ are dense matrices (parametrized as dense matrices or low-rank matrices). A schematic representation is provided in Figure 2.1.

In Hyena, the filters h_T, h_H, h_K are parametrized explicitly: the entries of the filters are learnable parameters, analogous to the approach of classical convolutional neural networks[1]. The inner filter h_G is instead parametrized implicitly, with the values obtained as a combination of basis functions or as outputs of a neural network (Romero et al., 2021). For this reason, computational primitives following the structure in Equation 1 have been also broadly referred to as long convolution operators.

We build on this basic structure, leaning into the design of convolution operators. The main insights are that not every input-dependent convolution in a hybrid should rely on long, implicit filters, and that convolutional operators should be tailored to run fast on target hardware.

Input-dependent convolutional operators The first class of input-dependent convolutions is Hyena-LI (long implicit), the closest relative to the original design. In Hyena-LI, the filters h_T, h_H, h_K remain short and explicit, while the inner filter is obtained as a linear combination of real exponentials $h_t = \sum_{n=1}^{d} R_n \lambda_n^{t-1}, R_n, \lambda_n \in \mathbb{R}$ (Massaroli et al., 2024). This is a real-valued, simplified version of a variety of other parametrizations (Orvieto et al., 2023; Gupta et al., 2022), with the addition of filter grouping (Section 2.2). Due to this choice, Hyena-LI retains the ability to switch to an recurrent parametrization for constant memory.

Next, we define Hyena-SE (short explicit), a variant with short, explicit filters in all its convolutions. When the filters are short[2], a simple explicit parametrization is sufficient to achieve convergence. Hyena-SE is key in achieving speedups across a range of input regimes, including short sequences, while still excelling at local, multi-token recall. With a hardware-aware implementation using tensor cores, Hyena-SE achieves the highest throughput of any sequence mixing operator (Section 3). Hyena-SE can also be utilized as a replacement for feed-forward layers.

Finally, we introduce Hyena-MR (medium regularized), a variant with explicitly parametrized filters of length in the hundreds. While it can be difficult to optimize longer explicit convolutions, we find that a simple exponential decay regularizer i.e., $h_t = \hat{h}_t \lambda^{-\alpha t}$, where α is swept across channels and $\hat{h}_t$ is the learnable parameter, is sufficient for convergence. With filter grouping and an efficient implementation using tensor

[1] The presence of short explicit filters in the featurization step for query, key and value, first proposed for input-dependent convolution in (Poli et al., 2023a) and linear attention in (Peng et al., 2023), has also been later adopted by other modern operator variants

[2] In our experiments, shorter than 14. In our final runs at scale, we used a range of 4 to 7.

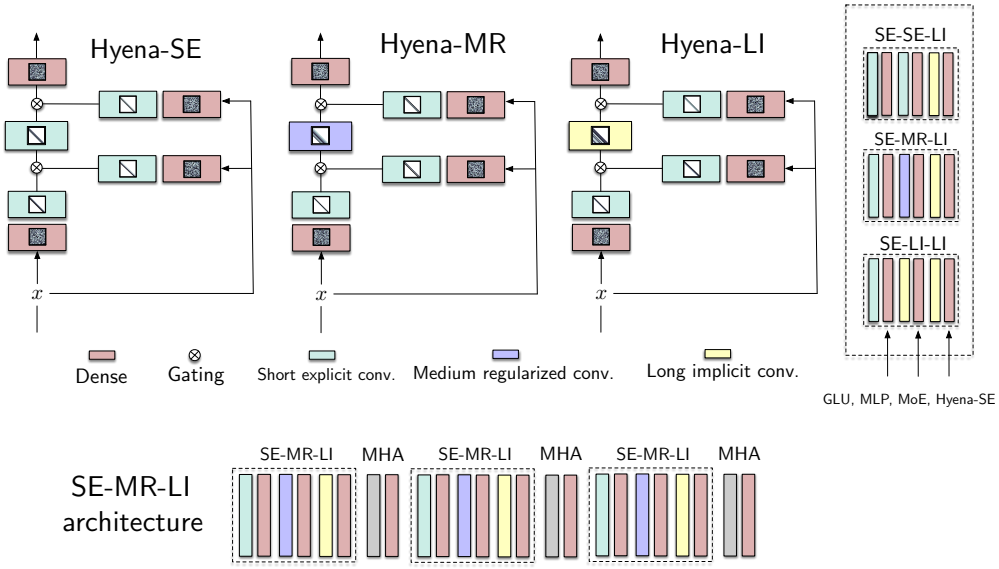


Figure 2.1: Overview of the convolutional operators forming the basis of StripedHyena 2:Hyena-SE (short explicit filters), Hyena-MR (medium regularized filters), Hyena-LI (long implicit filters). All operators use the Hyena structure (Poli et al., 2023a), tailoring the inner convolution parametrization for an improved balance of quality and efficiency. Given these operators, we explore different striped layouts.

cores, this variant remains significantly faster than linear attention and state-space models (Section 3). Hyena-MR is to Hyena-LI what sliding window attention is to the classic attention operator.

Since the filters in Hyena-MR and Hyena-SE are finite-impulse response (FIR), these operators trivially retain constant memory during autoregressive generation, analogous to sliding window attention.

Multi-hybrids interleave Hyena-SE, Hyena-MR and Hyena-LI to obtain the full architecture.

2.2 Additional Design Decisions

Next, we detail additional design aspects of convolutional multi-hybrids, including block layout, weight-sharing filter patterns, and effectiveness of context extension techniques.

Experiment configuration files and code are provided in Savanna, our fully open-source pretraining infrastructure for research on multi-hybrids: https://github.com/Zymrael/savanna.

Multi-hybrid block layout We explore the effect of block layouts on performance and training speed. Table 2.1 shows the results of training 7B multi-hybrid models on 400B tokens of OpenGenome2 (Brixi et al., 2025), each with different block layouts. We use our default inner filter lengths of 7 for SE and 128 for MR.

The blocks are repeated until target 7B parameter model depth (32) is achieved. All StripedHyena 2 models in addition interleave 5 MHA operators with the convolutional blocks. Validation perplexity measured after training of 400B tokens of byte-tokenized data (DNA sequences from OpenGenome2), with 7B parameter StripedHyena 2 models.

Layout	PPL@400B
MHA-MHA-MHA	3.09
LI-LI-LI	2.87
SE-SE-LI	2.88
SE-MR-LI	2.83

Table 2.1: Effect of different block layouts on pretraining at the 7B parameter scale.

At this scale, SE-MR-LI perform best on pretraining quality. Notably, we find that pure long convolution LI-LI-LI layouts can be replaced by SE-SE-LI blocks for little-to-no loss in quality and significant benefits to throughput. While SE-MR-LI block layouts provide a general stable baseline for multi-hybrids, we recommend ablating block layouts for new tasks or domains, particularly if parametrization hyperparameters such as filter length, decay strength, and initialization are modified.

Weight-sharing filter patterns We propose a grouped[3] design for input-dependent convolutional operators, where the filters are shared across groups of channels. Namely, let $\mathcal{G}$ be a group of channels of size d_g. Then,

$$\forall \alpha \in \mathcal{G} : y_t^\alpha = \sum_{j=0}^{t} h_{t-j}^{\mathcal{G}} x_j^\alpha.$$

The main benefit of our grouping is to enable efficient representation of the discrete convolution as a series of general matrix-matrix multiplications (`GEMM`) instead of general matrix-vector multiplications (`GEMV`). We use this property to co-design hardware-aware algorithms for our architecture (see Section 3). Grouping has minimal effect on quality (Section B.1).

Context extension We tested context extension up to 1 million sequence length on 7B and 40B multi-hybrids using techniques developed for rotary attention, such as *position interpolation* PI (Chen et al., 2023) and *adjusted base frequency* (ABF) (Xiong et al., 2023) and a combination of both. We found only minor differences in perplexity (Table 2.2), with all models capable of in-context recall at the target maximum context length. Recall results are shown in Figure B.2.

Extension Method	Context Length (K)					
	32	**65**	**131**	**262**	**524**	**1048**
Position Interpolation (PI)	2.785	2.763	2.750	-	-	-
PI + ABF	2.782	2.763	2.748	2.707	2.663	2.597

Table 2.2: Validation perplexity of 7B StripedHyena 2 architecture on `OpenGenome2` after midtraining extension with different techniques, terminating in extension at 1 million context length. Midtraining is performed on the base StripedHyena 2 7B trained on 2T tokens of `OpenGenome2` at byte resolution and 8192 context length (Evo 2 7B). The values are collected at the end of training, for a visualization of the trends see Figure B.2.

2.3 Measuring Throughput at Scale

Owing to its design incorporating FIR convolution operators such as Hyena-SE and Hyena-MR, multi-hybrids achieve consistent speedups compared to previous generation hybrids and Transformers.

Across the 7B and 40B parameter scales, StripedHyena 2 trains 1.2 to 2.9 times faster on a H100 cluster compared to our optimized Transformer based on a reference `Transformer Engine` implementation, collected during training with FP8 precision on dense (SwiGLUs, projections) and normalization layers (Figure 2.2). Notably, it also achieves speedups at shorter sequence lengths, compared to both Transformers and StripedHyena (Poli et al., 2023b), a previous generation hybrid. Given an comparable degree of optimization in the implementations, we expect similar or better gains on other training infrastructure. See Table C.1 for details on the measurement protocol.

We also report TFLOPS per second per GPU and MFU[4] in Figure 2.2. Since we use the same distributed settings for all architectures, we note a lower MFU for hybrids at longer sequence lengths. This is primarily due to a reduction in overall model FLOPS[5] caused by subquadratic scaling in sequence length. Consequently, we expect further tuning of distributed settings for multi-hybrids at long context to yield even larger speedups by using e.g., larger micro batch sizes to increase compute intensity of each rank. Further discussion on context parallelism implementation is provided in Section 4.

[3] Note that this approach is not the same as traditional grouped CNN layers, which instead mixes across channels in the same group.

[4] We use a reference number of 1000 TFLOPs per H100.

[5] We used actual model FLOPS instead of approximations, since most approximations are not accurate at very long context. For self-attention FLOPS, we used the estimate in Dao (2023).

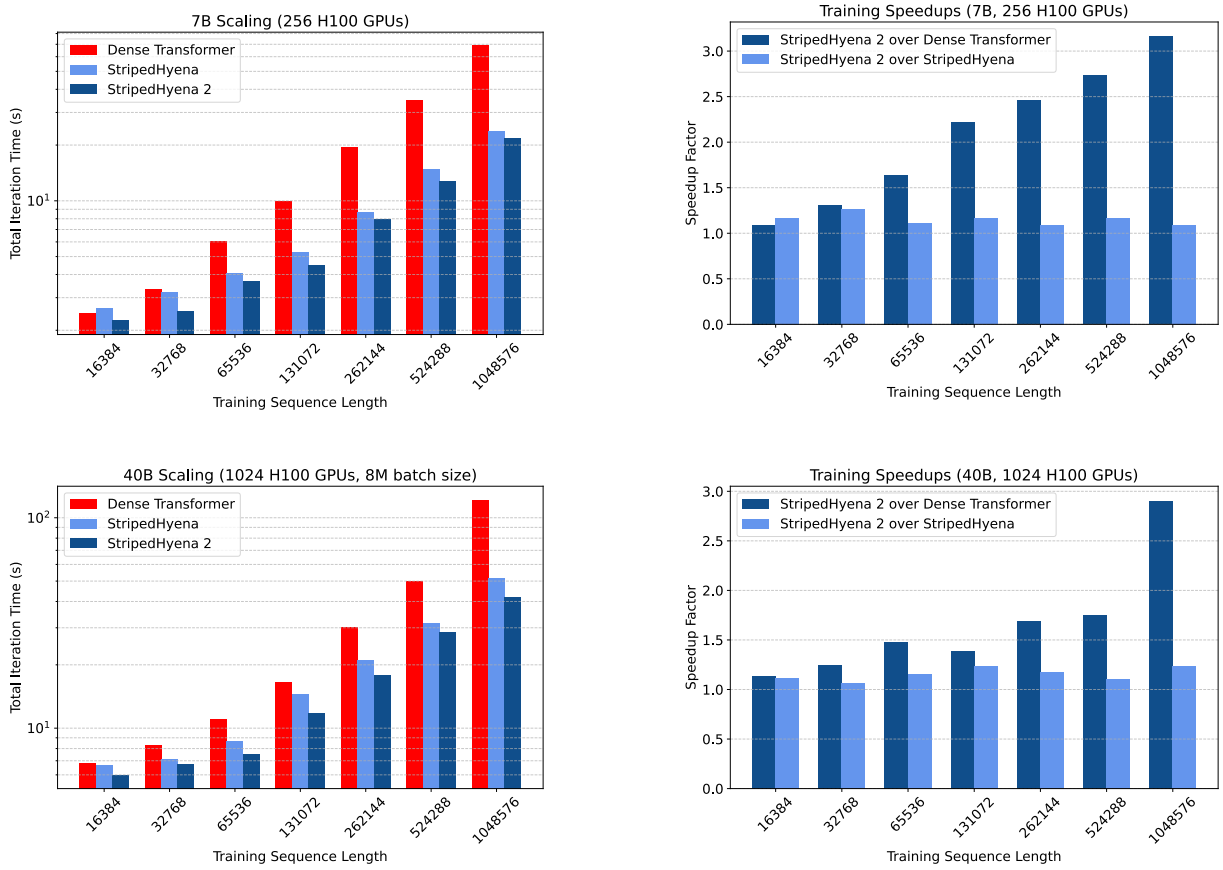


Figure 2.2: End-to-end iteration times (forward and backward) during training, collected on a large cluster of H100 SXM GPUs. See Table C.1 for details on the measurement protocol.

Convolutional multi-hybrids such as StripedHyena 2 similarly outscale other previous-generation hybrids, including those based on linear attention or state-space models. Latency and MFU at the operator level are provided in the following section.

3 Hardware-Aware Convolution Algorithms

Discrete convolutions are mathematically equivalent to matrix multiplications with Toeplitz matrices[6]. Convolutional operators in StripedHyena 2 operate in computational regimes that are quite different from traditional convolutional neural networks (CNNs), requiring custom algorithms:

- `Hyena-SE`: *short-explicit* hyena layers are based on a combination of grouped depthwise convolutions with explicitly parametrized shorter filters (e.g., length 7).
- `Hyena-MR`: *medium-regularized* hyena layers are based on a combination of grouped depthwise convolutions with longer filters (e.g., length 128), obtained by applying a regularization term to the filter weights.
- `Hyena-LI`: *long-implicit* hyena layers are based on a combination of grouped depthwise convolutions with longer *implicit* filters (as long as the sequence length) (Romero et al., 2021; Poli et al., 2023a), in the same family as other long convolutions.

Libraries such as PyTorch provide optimized implementations of short explicit convolutions via a variety of backends due to their common utilization in CNN architectures. These implementations span approaches such as im2col and Winograd algorithms (Chetlur et al., 2014; Vasudevan et al., 2017). However, existing implementations are not fully optimized for the convolution operators used in multi-hybrids, which rely mostly on depthwise convolutions of different lengths. When the convolution filter is very long, most algorithms rely on FFT-based methods. These approaches are known to suffer from lower hardware utilization,

[6] Algorithms for fast convolutions correspond to fast matrix multiplications schemes for Toeplitz matrices.

despite modifications to leverage tensor cores (Li et al., 2021; Fu et al., 2023). Instead of im2col or Winograd GEMM algorithms for convolutions, we focus on a direct multi-pass blocked approach that is co-designed to exploit filter grouping in Hyena.

3.1 Block Convolution

For a causal FIR filter of length ℓ_h applied to an input signal x of length ℓ (with typical $\ell \gg \ell_h$), the output at index t is

$$y_t = \sum_{k=0}^{t} h_{t-k}\, x_k, \quad \text{where } h_k = 0 \text{ for } k < 0 \text{ or } k \geq \ell_h. \tag{2}$$

This can be written in matrix form:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \vdots \\ y_{\ell-1} \end{bmatrix} = \underbrace{\begin{bmatrix} h_0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ h_1 & h_0 & 0 & 0 & 0 & \cdots & 0 \\ \vdots & h_1 & h_0 & 0 & 0 & \cdots & 0 \\ h_{\ell_h-1} & \vdots & h_1 & h_0 & 0 & \cdots & 0 \\ 0 & h_{\ell_h-1} & \vdots & h_1 & h_0 & \cdots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & h_{\ell_h-1} & \cdots & h_1 & h_0 \end{bmatrix}}_{T} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_{\ell-1} \end{bmatrix}. \tag{3}$$

Classical digital signal processing often handles FIR filters using *block convolution* methods (Burrus & Parks, 1985). One partitions both the input and the output signals into chunks of size ℓ_b, then multiplies $\ell_b \times \ell_b$ sub-blocks of T against these smaller segments. Once the filter's support is exceeded, many sub-blocks are purely zeros and can be skipped – an advantage when $\ell_h \ll \ell$.

Concretely, the input and output sequences are chunken into $x = (\hat{x}_0, \hat{x}_1, \dots)$, $y = (\hat{y}_0, \hat{y}_1, \dots)$ with

$$\hat{x}_k = (x_{k\ell_b}, \dots, x_{k\ell_b+\ell_b-1}), \quad \hat{y}_k = (y_{k\ell_b}, \dots, y_{k\ell_b+\ell_b-1}), \quad k = 0, 1, 2, \dots, \lceil \ell/\ell_b \rceil - 1. \tag{4}$$

The fully partitioned Toeplitz matrix factors into submatrices $H_0, H_1, \dots, H_{\lceil \ell/\ell_b \rceil - 1}$ of size $\ell_b \times \ell_b$. For instance,

$$H_0 = \begin{bmatrix} h_0 & 0 & \cdots & 0 \\ h_1 & h_0 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ h_{\ell_b-1} & \cdots & h_1 & h_0 \end{bmatrix}, \quad H_1 = \begin{bmatrix} h_{\ell_b} & h_{\ell_b-1} & \cdots & h_1 \\ h_{\ell_b+1} & h_{\ell_b} & \ddots & \vdots \\ \vdots & \ddots & \ddots & h_{\ell_b-1} \\ h_{2\ell_b-1} & \cdots & h_{\ell_b+1} & h_{\ell_b} \end{bmatrix}, \quad \dots \tag{5}$$

Hence,

$$T = \begin{bmatrix} H_0 & & & \\ H_1 & H_0 & & \\ H_2 & H_1 & H_0 & \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}. \tag{6}$$

Since $h_t = 0$ for $t \geq \ell_h$, any blocks with index greater than $\lceil (\ell_h - 1)/\ell_b \rceil + 1$ yields a zero submatrix. Hence, we only need to construct and multiply the non-zero submatrices $H_k, k = 0, 1, \dots, \lceil (\ell_h - 1)/\ell_b \rceil$. The output blocks then obey:

$$\hat{y}_n = \sum_{k=0}^{n} H_{n-k}\, \hat{x}_k = \sum_{k=0}^{\lceil (\ell_h-1)/\ell_b \rceil} H_k\, \hat{x}_{n-k}. \tag{7}$$

Note that *block convolution* (7) can be seen as a "convolution of convolutions", since each block H_k is itself a Toeplitz matrix and can be implemented efficiently by direct multiplication or by using fast convolution techniques (e.g., FFT-based methods when ℓ_b is large) within each block.

3.2 Simple Two-Stage Block Algorithm

For many `Hyena-SE` or `Hyena-MR` use cases, ℓ_h (the filter length) is much smaller than ℓ (the sequence length). When ℓ_h is also within about twice the chosen block size ℓ_b, a particularly efficient two-stage block algorithm can be used.

Let T be the Toeplitz matrix that applies a grouped depthwise FIR filter of length ℓ_h. Suppose we choose a block size ℓ_b such that $\ell_h \leq 2\ell_b$. Under this condition, T can be decomposed into a block-diagonal part plus an off-diagonal part (or "stage"):

$$T = \underbrace{\begin{bmatrix} H_0 & & \\ & H_0 & \\ & & \ddots \\ & & & H_0 \end{bmatrix}}_{\text{first stage}} + \underbrace{\begin{bmatrix} H_1 & & \\ & \ddots & \\ & & H_1 \end{bmatrix}}_{\text{second stage}}. \tag{8}$$

In particular:

- H_0 covers the points of the filter that align with the current chunk $\hat{x}_k$.
- H_1 covers the points of the filter that spills over from the previous chunk $\hat{x}_{k-1}$, capturing taps that straddle the boundary between adjacent chunks.

As an illustrative example, consider $\ell = 6$ (sequence length), $\ell_h = 4$ (filter length), and $\ell_b = 3$ (block size). The filter coefficients h_0, h_1, h_2, h_3 form the blocks:

$$H_0 = \begin{bmatrix} h_0 & 0 & 0 \\ h_1 & h_0 & 0 \\ h_2 & h_1 & h_0 \end{bmatrix}, \quad H_1 = \begin{bmatrix} h_3 & h_2 & h_1 \\ 0 & h_3 & h_2 \\ 0 & 0 & h_3 \end{bmatrix}.$$

The full Toeplitz matrix T decomposes as:

$$T = \underbrace{\begin{bmatrix} H_0 & 0 \\ 0 & H_0 \end{bmatrix}}_{\text{first stage}} + \underbrace{\begin{bmatrix} 0 & 0 \\ H_1 & 0 \end{bmatrix}}_{\text{second stage}} = \begin{bmatrix} h_0 & 0 & 0 & 0 & 0 & 0 \\ h_1 & h_0 & 0 & 0 & 0 & 0 \\ h_2 & h_1 & h_0 & 0 & 0 & 0 \\ h_3 & h_2 & h_1 & h_0 & 0 & 0 \\ 0 & h_3 & h_2 & h_1 & h_0 & 0 \\ 0 & 0 & h_3 & h_2 & h_1 & h_0 \end{bmatrix}.$$

This approach presents a number of advantages. Once loaded, H_0 and H_1 can be reused across multiple chunks of the input and, with our grouped operator design, also across multiple channels within the same group. This provides a convenient way to turn small `GEMV` operations into `GEMMs`, compared to other `GEMM` approaches for convolutions that rely on forming strided views of the input. Furthermore, the decomposition separates "current chunk" vs. "previous chunk" computations, which can be run in parallel or as a pipeline.

Analysis of the two-stage multiplication. Suppose we denote by $\hat{X}_n \in \mathbb{R}^{\ell_b \times d}$ the n-th input chunk (as in (7)), where d denotes both the group size and the tensor core dimension, and let $\hat{Y}_n \in \mathbb{R}^{\ell_b \times d}$ be the corresponding output chunk. Under a two-stage block convolution, each $\hat{Y}_n$ is computed as

$$\hat{Y}_n = H_0\,\hat{X}_n \;+\; H_1\,\hat{X}_{n-1}, \quad n = 0, 1, \ldots, \lceil \ell/\ell_b \rceil - 1, \quad (\text{with } \hat{X}_{-1} = 0 \text{ for } n = 0). \tag{9}$$

Here, H_0 captures the filter taps interacting with the "current" chunk $\hat{X}_n$, while H_1 handles the "spillover" from the preceding chunk $\hat{X}_{n-1}$. By construction, $\ell_h \leq 2\,\ell_b$ ensures that no additional off-diagonal blocks appear beyond H_1.

In the setting where all channels within a group share the same filter, the matrices $H_0 \in \mathbb{R}^{\ell_b \times \ell_b}$ and $H_1 \in \mathbb{R}^{\ell_b \times \ell_b}$ are common to every channel in the group. Consequently, instead of forming a block-diagonal structure over separate channels, one can directly operate on the full input block $\hat{X}_n \in \mathbb{R}^{\ell_b \times d_g}$, where d_g is the group size. In particular, if the tensor core is of size d_g and $\ell_b = d_g$, then (9) can be implemented as two full `GEMM` operations. This approach maximizes throughput and fully leverages tensor core utilization by processing the entire group in one efficient matrix multiplication.

3.2.1 Implementation

We report the kernel implementation of our two-stage blocked algorithm in Algorithm 1. The crux lies in the grouping structure, which enables data reuse and efficient computation using tensor cores dedicated hardware units on NVIDIA GPUs specialized for high throughput matrix multiplication. Without grouping, one would need to adopt a different strategy to transform depthwise convolutions from `GEMVs` to `GEMMs` to avoid lower throughput CUDA cores, for example forming an input view that parallelizes the first-stage multiplication with H_0 across all blocks.

Algorithm 1 Simple Two-Stage Blocked Hyena Convolution (Forward)

Require: Input $v,\ q,\ k \in \mathbb{R}^{\ell \times d}$, filter $h \in \mathbb{R}^{\ell_h}$, block size ℓ_b
Ensure: Output $y \in \mathbb{R}^{\ell \times d_g}$

Chunk inputs $v,\ q,\ k$ into blocks $v_i,\ q_i,\ k_i$ of size $\ell_b \times d_g$
for block $i = 0$ to $\lceil \ell / \ell_b \rceil - 1$ **do**
    Load $v_i,\ q_i,\ k_i,\ H_0,\ H_1$ to on-chip memory
    Initialize $y_i = 0$
    Optional: $v_i \leftarrow k_i \odot v_i$
    $y_i \leftarrow H_0 v_i$ ▷ First `GEMM`: block-diagonal
    if $i > 0$ **then**
        Load x_{i-1} to on-chip memory
        $y_i \leftarrow y_i + H_1 v_{i-1}$ ▷ Second `GEMM`: off-diagonal
    end if
    Optional: Compute $y_i \leftarrow q_i \odot y_i$
end for
return y

3.2.2 Profiling

Convolutional multi-hybrids models are designed to be efficient across a wide range of regimes, compared to both full attention and other subquadratic operators. We optimize for both short and long sequences, as pretraining is often performed at shorter sequence lengths to maximize throughput.

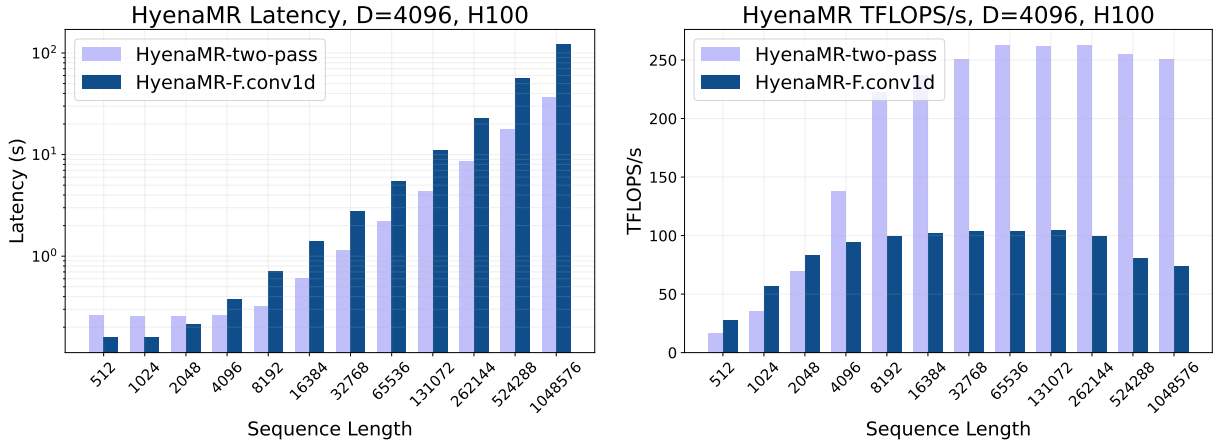


Figure 3.1: Forward latency and TFLOPS / second of Hyena-MR variants with filter length 128. We compare a baseline implementation using PyTorch convolutions and our two-stage blocked kernel, showing substantial improvements in latency and throughput.

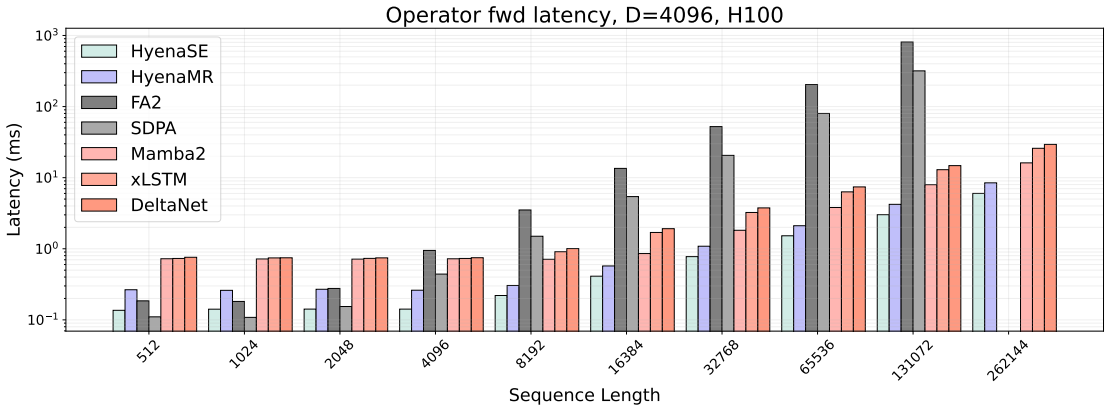


Figure 3.2: Forward latency and TFLOPs / second of Hyena-SE, Hyena-MR and other common operators in architecture design: *multi-head attention* (MHA) and linear attention variants. All values are collected at operator width 4096 (corresponding to model width at 7B parameters), on H100s. For MHA, we report both a highly optimized implementation for Hopper GPUs (PyTorch SDPA) as well as a previous generation implementation not optimized for Hopper GPUs (FlashAttention2) (Dao, 2023). All other operators use their official auto-tuned Triton kernels. Convolutional primitives remain efficient across sequence lengths, with substantially higher throughput than other operators, including efficient alternatives to MHA.

Measurement protocol We measure the latency and throughput of common operators in a batch size 1, varying sequence length, model width 4096 setting (corresponding to common operator width for 7B models), including input and output projections. We do not keep the total number of tokens constant, in contrast to the protocol used in FlashAttention 3 (Shah et al., 2024). Figure 3.2 and B.4 provide the results.

4 Training Multi-Hybrids on Long Sequences

Notation In the following, we consider an input of shape $[1, D, L]$, with batch size 1, hidden size D and length L and omit the leading 1. The discussion can be safely extended to inputs with larger batch sizes following standard data parallelism. For a CP group consisting of N_{cp} devices, the input is sharded along the sequence dimension and split across each of the devices in the group, so that each rank holds an input shard of shape $[D, L//N_{cp}]$.

4.1 Background

Context parallelism (CP) refers to a collection of distributed training techniques designed to handle the growing size of models and the increasing dimension of their inputs by processing segments of the full input sequence. Context parallelism complements other distributed training techniques such as data parallelism, tensor parallelism, sequence parallelism[7], pipeline parallelism and other strategies for partitioning of model parameters, gradients and optimizer states (Rajbhandari et al., 2020; Zhao et al., 2023).

`All-to-all` context parallelism In `a2a` context parallelism, each device is allowed to exchange data with every other device in the context parallel group to reconstruct the entire input sequence and hold hidden dimension splits on each CP rank instead. Concretely, the N_{cp} shards of shape $[D, L//N_{cp}]$ are redistributed among all devices, such that each device ends up with shards of shape $[H//N_{cp}L]$ instead. This allows each rank to independently carry out sequence mixing (e.g., attention, or convolutions). It is important to emphasize that the hidden dimension must be split in such a way that no additional communication is required to successfully complete the operation. Extended background is provided in Section A.2.1.

`Point-to-point` context parallelism While `a2a` CP allows processing long inputs across multiple devices, the cost of running the operator over the whole sequence can still be very expensive. Furthermore, naive `a2a` can lower utilization without appropriate overlap of communication and computation, as `a2a` calls

[7] Context and sequence parallelisms refer to different techniques by popular convention. Sequence parallelism distributes the sequence outside tensor parallel regions e.g., normalization layers

can take a significant amount of time with larger message sizes. To overcome some of these issues, *point-to-point* (`p2p`) context parallelism allows ranks to exchange data directly with a single peer at a time rather than broadcasting to all devices. These schemes essentially perform several rounds of blocked computation and communication. Section A.2.2 describes the common ring-based algorithm for attention (Liu et al., 2023).

4.2 Context Parallel Hyena Operators

Sequence mixing in Hyena operators is implemented via convolutions with different filter lengths, which need to be addressed appropriately when implementing context parallelism. We first present the general `a2a` and `p2p` CP formulations for general causal convolutions, followed by modifications specifically tailored to FFT convolutions.

`All-to-all` convolutions (Fig. 4.1) Let us consider an input sharded along the sequence dimension over $\texttt{N}_{\texttt{cp}}$ ranks, such that each rank holds a split of shape $[\texttt{D}, \texttt{L}//\texttt{N}_{\texttt{cp}}]$. Analogous to the general `a2a` context parallel formulation, we perform communication across all CP ranks so that input shards of shape $[\texttt{H}//\texttt{N}_{\texttt{cp}}, \texttt{L}]$ are held on each device, then convolve[8] each shard within the context parallel region. Finally, we perform an additional `a2a` operation.

For convolutional operators in multi-hybrids, additional considerations are needed. First, filters can be stored or materialized directly inside each context parallel region. In Hyena-SE, each context parallel rank stores $\texttt{H}//\texttt{N}_{\texttt{cp}}$ filters in the depthwise case, without grouping. Care must be taken to ensure filter groups are not split across context parallel ranks. In Hyena-MR and Hyena-LI, computation of the filters can be run in each context parallel region, keeping implicit or regularization parameters sharded. If `a2a` parallelism is used as the scheme of choice for the inner convolution, gating can be performed outside the context parallel region, to avoid communication overheads.

During the backward pass, gradients must be sent back to their incoming ranks during the forward call. Consequently, this involves calling two additional `a2a` calls, plus the backward function on the convolutional operation on each rank.

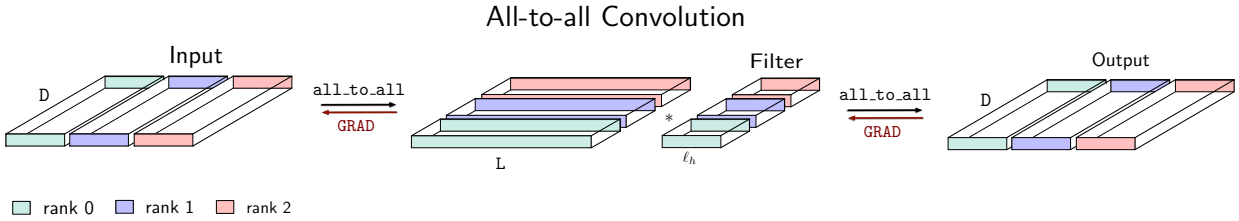


Figure 4.1: Diagram of computation and communication in `all-to-all convolutions`. This context parallelism strategy can be used in both inner hyena convolutions (corresponding to multiplication with $G_{tt'}$, Eq. 1) or featurizer convolutions ($T_{tt'}$, $H_{tt'}$, $K_{tt'}$). Filters are stored or computed in each context parallel rank to avoid communication overheads. The convolution inside the context parallel region can be computed with any algorithm e.g., FFT-based or direct.

[Extension] `All-to-all` channel-pipelined convolutions: One drawback of `a2a` methods is that communication latency can create bottlenecks when the message size is large, with a small fraction of time spent on compute. For `a2a`, this occurs when model size and sequence length grows. One strategy is to pipeline `a2a` calls and asynchronously overlap compute and communication using CUDA streams. Instead of pipelining over sequence length (Yao et al., 2024), we explore pipelining over channels to hide some of the communication latency. Concretely, we chunk inputs $[\texttt{D}, \texttt{L}//\texttt{N}_{\texttt{cp}}]$ into $\texttt{N}_{\texttt{pipe}}$ segments and run an asynchronous loop of `a2a` calls, scheduling to overlap compute and following `a2a` call.

`Point-to-point` convolutions (Fig. 4.2) Analogous to the self-attention case, `p2p` context parallel causal depthwise convolutions implements context parallelism while using communication with a single peer at a time. Let us consider the usual sharded input, with each rank holding $[\texttt{D}, \texttt{L}//\texttt{N}_{\texttt{cp}}]$, to be causally convolved

[8] Both causal and non-causal convolutions are supported in this case.

with a filter $[\texttt{D}, \ell_{\texttt{h}}]$. We detail the depthwise case, but the grouped depthwise case can be obtained as a simplification. For FIR filters, one can exploit locality of the operation to simplify the implementation compared to `p2p` attention. In particular, the causal convolution can be performed locally on most of the input elements without the need for communication. Only the first $\ell_{\texttt{h}}-\texttt{1}$ elements on each shard rely on computation to be performed on a different rank, namely the last $\ell_{\texttt{h}}-\texttt{1}$ elements of the previous shard. Note that each rank keeps or materializes copies of the convolutional filters, since each rank is responsible for computing convolutions across all `D` channels. This is in contrast to `a2a` schemes, which have ranks operate on chunks of channels independently. Figure 4.2 provides a schematic of the algorithm.

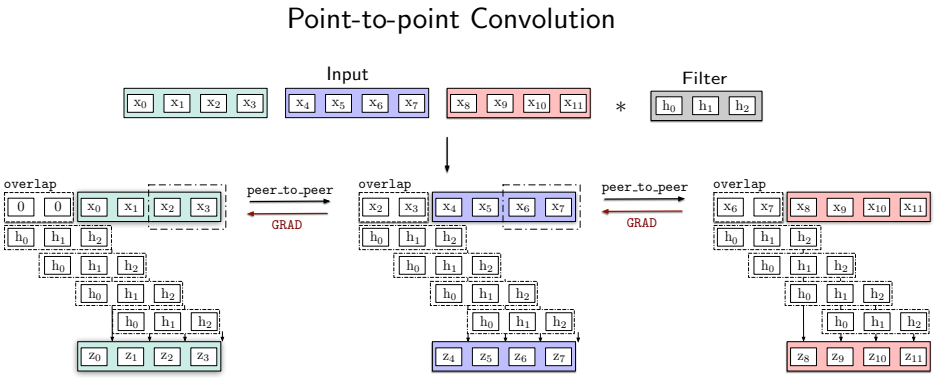


Figure 4.2: Diagram of computation and communication in `point-to-point convolutions`. This approach is best suited for FIR convolutions in Hyena-SE and Hyena-MR.

[Extension] `Point-to-point` **convolutions with overlapping communication:** Since only the first $\ell_{\texttt{h}} - \texttt{1}$ elements of each shard require inputs located on a different rank, we overlap local operations and `p2p` communication to further improve utilization. This process is illustrated in Supplementary Figure B.1. Instead of waiting for the overlap segment to arrive before computing the convolution, we start the local convolution with a zero-padded input and simultaneously start the communication of the previous segment. Once communication is concluded, an additional convolution over the right-padded overlap of shape $[\texttt{1}, \texttt{D}, \texttt{2}(\ell_{\texttt{h}}-\texttt{1})]$ and the convolutional kernel is performed. The result of this convolution is subsequently added to the first $\ell_{\texttt{h}} - \texttt{1}$ elements of the previous output. Interestingly, this algorithm relies on similar decomposition techniques as those involved in our two-stage block convolution approach (Section 3).

[Extension] `Point-to-point` **FFT Convolutions:** While the previous CP algorithms can also be used for Hyena-LI, convolutions with long filters are generally implemented via Fast Fourier Transform (FFT) algorithms. FFT convolution relies on the fact that convolution is equivalent to multiplication in the Fourier domain:

$$(x * h) = \mathsf{F}^{-1}(\mathsf{F}(x) \circ \mathsf{F}(h)), \tag{10}$$

where $\mathsf{F}, \mathsf{F}^{-1}$ are the Fourier and inverse Fourier transform, respectively.
FFTs require access to the entire input sequence. At first glance, one could think it mandatory to host the whole sequence in a single rank. Interestingly, it is possible to compute both the Fourier and inverse Fourier transform –and thus the FFT convolution– in a `p2p` fashion without ever hosting the whole sequence on a single device, by introducing communication during iterative steps of the FFTs. Unfortunately, without further optimizations, we generally observed `a2a` approaches to be faster for Hyena-LI. For completeness, we report the derivation in Appendix A.2.4).

5 Conclusion

In this paper, we introduce systems and algorithms for convolutional multi-hybrids, a new class of architecture for sequence modeling at scale. We discuss architecture design, block layout, kernels for fast convolutions on GPUs based on overlap-add schemes, and context parallelism strategies. Multi-hybrids excel at efficient modeling of byte and character-level data, and we expect their utilization to unlock new applications for

foundation models. Effectiveness of StripedHyena 2 is verified at scale (40 billion parameter, over 9 trillion tokens and 1 million context) with the Evo 2 line of models.

References

Joshua Ainslie, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. Gqa: Training generalized multi-query transformer models from multi-head checkpoints. *arXiv preprint arXiv:2305.13245*, 2023.

Ekin Akyürek, Bailin Wang, Yoon Kim, and Jacob Andreas. In-context language learning: Arhitectures and algorithms. *arXiv preprint arXiv:2401.12973*, 2024.

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

Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xlstm: Extended long short-term memory. *arXiv preprint arXiv:2405.04517*, 2024.

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

Saad Bouguezel, M Omair Ahmad, and MNS Swamy. An improved radix-16 fft algorithm. In *Canadian Conference on Electrical and Computer Engineering 2004 (IEEE Cat. No. 04CH37513)*, volume 2, pp. 1089–1092. IEEE, 2004.

William Brandon, Aniruddha Nrusimha, Kevin Qian, Zachary Ankner, Tian Jin, Zhiye Song, and Jonathan Ragan-Kelley. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431*, 2023.

Garyk Brixi, Matthew G. Durrant, Jerome Ku, Michael Poli, Greg Brockman, Daniel Chang, Gabriel A. Gonzalez, Samuel H. King, David B. Li, Aditi T. Merchant, Mohsen Naghipourfar, Eric Nguyen, Chiara Ricci-Tam, David W. Romero, Gwanggyu Sun, Ali Taghibakshi, Anton Vorontsov, Brandon Yang, Myra Deng, Liv Gorton, Nam Nguyen, Nicholas K. Wang, Etowah Adams, Stephen A. Baccus, Steven Dillmann, Stefano Ermon, Daniel Guo, Rajesh Ilango, Ken Janik, Amy X. Lu, Reshma Mehta, Mohammad R.K. Mofrad, Madelena Y. Ng, Jaspreet Pannu, Christopher Ré, Jonathan C. Schmok, John St. John, Jeremy Sullivan, Kevin Zhu, Greg Zynda, Daniel Balsam, Patrick Collison, Anthony B. Costa, Tina Hernandez-Boussard, Eric Ho, Ming-Yu Liu, Thomas McGrath, Kimberly Powell, Dave P. Burke, Hani Goodarzi, Patrick D. Hsu, and Brian L. Hie. Genome modeling and design across all domains of life with evo 2. 2025.

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

C Sidney Burrus and T Parks. Convolution algorithms. *Citeseer: New York, NY, USA*, 6:15, 1985.

Shouyuan Chen, Sherman Wong, Liangjian Chen, and Yuandong Tian. Extending context window of large language models via positional interpolation. *URL https://arxiv. org/abs/2306.15595*, 2023.

Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

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

Zihang Dai, Hanxiao Liu, Quoc V Le, and Mingxing Tan. Coatnet: Marrying convolution and attention for all data sizes. *Advances in neural information processing systems*, 34:3965–3977, 2021.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. *arXiv preprint arXiv:2307.08691*, 2023.

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

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

Mahan Fathi, Jonathan Pilault, Pierre-Luc Bacon, Christopher Pal, Orhan Firat, and Ross Goroshin. Block-state transformer. *arXiv preprint arXiv:2306.09539*, 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. *arXiv preprint arXiv:2212.14052*, 2022.

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

Paolo Glorioso, Quentin Anthony, Yury Tokpanov, James Whittington, Jonathan Pilault, Adam Ibrahim, and Beren Millidge. Zamba: A compact 7b ssm hybrid model. *arXiv preprint arXiv:2405.16712*, 2024.

Ankit Gupta, Albert Gu, and Jonathan Berant. Diagonal state spaces are as effective as structured state spaces. *Advances in Neural Information Processing Systems*, 35:22982–22994, 2022.

Shen-Jui Huang and Sau-Gee Chen. A high-throughput radix-16 fft processor with parallel and normal input/output ordering for ieee 802.15. 3c systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 59(8):1752–1765, 2012.

Sam Ade Jacobs, Masahiro Tanaka, Chengming Zhang, Minjia Zhang, Shuaiwen Leon Song, Samyam Rajbhandari, and Yuxiong He. Deepspeed ulysses: System optimizations for enabling training of extreme long sequence transformer models. *arXiv preprint arXiv:2309.14509*, 2023.

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

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

Binrui Li, Shenggan Cheng, and James Lin. tcfft: Accelerating half-precision fft through tensor cores. *arXiv preprint arXiv:2104.11471*, 2021.

Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, et al. Jamba: A hybrid transformer-mamba language model. *arXiv preprint arXiv:2403.19887*, 2024.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Dengr, Chong Ruan, Damai Dai, Daya Guo, et al. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *arXiv preprint arXiv:2405.04434*, 2024.

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

Stefano Massaroli, Michael Poli, Dan Fu, Hermann Kumbong, Rom Parnichkun, David Romero, Aman Timalsina, Quinn McIntyre, Beidi Chen, Atri Rudra, et al. Laughing hyena distillery: Extracting compact recurrences from convolutions. *Advances in Neural Information Processing Systems*, 36, 2024.

Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. *arXiv preprint arXiv:1812.06162*, 2018.

Eric Nguyen, Michael Poli, Matthew G Durrant, Brian Kang, Dhruva Katrekar, David B Li, Liam J Bartie, Armin W Thomas, Samuel H King, Garyk Brixi, et al. Sequence modeling and design from molecular to genome scale with evo. *Science*, 386(6723):eado9336, 2024.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. In *International Conference on Machine Learning*, pp. 26670–26698. PMLR, 2023.

Bo Peng, Eric Alcaide, Quentin Anthony, Alon Albalak, Samuel Arcadinho, Stella Biderman, Huanqi Cao, Xin Cheng, Michael Chung, Matteo Grella, et al. Rwkv: Reinventing rnns for the transformer era. *arXiv preprint arXiv:2305.13048*, 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. In *International Conference on Machine Learning*, pp. 28043–28078. PMLR, 2023a.

Michael Poli, Jue Wang, Stefano Massaroli, Jeffrey Quesnelle, Ryan Carlow, Eric Nguyen, and Armin Thomas. Stripedhyena: Moving beyond transformers with hybrid signal processing models, 12 2023b. *URL https://github.com/togethercomputer/stripedhyena*, 2023b.

Michael Poli, Armin W Thomas, Eric Nguyen, Pragaash Ponnusamy, Björn Deiseroth, Kristian Kersting, Taiji Suzuki, Brian Hie, Stefano Ermon, Christopher Ré, et al. Mechanistic design and scaling of hybrid architectures. *arXiv preprint arXiv:2403.17844*, 2024.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: Memory optimizations toward training trillion parameter models. In *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–16. IEEE, 2020.

David W Romero, Anna Kuzina, Erik J Bekkers, Jakub M Tomczak, and Mark Hoogendoorn. Ckconv: Continuous kernel convolution for sequential data. *arXiv preprint arXiv:2102.02611*, 2021.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv preprint arXiv:2407.08608*, 2024.

Noam Shazeer. Fast transformer decoding: One write-head is all you need. *arXiv preprint arXiv:1911.02150*, 2019.

Noam Shazeer. Glu variants improve transformer. *arXiv preprint arXiv:2002.05202*, 2020.

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

Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing*, 568:127063, 2024.

Daisuke Takahashi. *Fast Fourier transform algorithms for parallel computers*. Springer, 2019.

Jamba Team, Barak Lenz, Alan Arazi, Amir Bergman, Avshalom Manevich, Barak Peleg, Ben Aviram, Chen Almagor, Clara Fridman, Dan Padnos, et al. Jamba-1.5: Hybrid transformer-mamba models at scale. *arXiv preprint arXiv:2408.12570*, 2024.

Aravind Vasudevan, Andrew Anderson, and David Gregg. Parallel multi channel convolution using general matrix multiplication. In *2017 IEEE 28th international conference on application-specific systems, architectures and processors (ASAP)*, pp. 19–24. IEEE, 2017.

Ashish Vaswani, Prajit Ramachandran, Aravind Srinivas, Niki Parmar, Blake Hechtman, and Jonathon Shlens. Scaling local self-attention for parameter efficient visual backbones. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 12894–12904, 2021.

Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tieyan Liu. On layer normalization in the transformer architecture. In *International Conference on Machine Learning*, pp. 10524–10533. PMLR, 2020.

Wenhan Xiong, Jingyu Liu, Igor Molybog, Hejia Zhang, Prajjwal Bhargava, Rui Hou, Louis Martin, Rashi Rungta, Karthik Abinav Sankararaman, Barlas Oguz, et al. Effective long-context scaling of foundation models. *arXiv preprint arXiv:2309.16039*, 2023.

Songlin Yang and Yu Zhang. Fla: A triton-based library for hardware-efficient implementations of linear attention mechanism, January 2024. URL https://github.com/fla-org/flash-linear-attention.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. *arXiv preprint arXiv:2406.06484*, 2024.

Jinghan Yao, Sam Ade Jacobs, Masahiro Tanaka, Olatunji Ruwase, Aamir Shafi, Hari Subramoni, and Dhabaleswar K Panda. Training ultra long context language model with fully pipelined distributed transformer. *arXiv preprint arXiv:2408.16978*, 2024.

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

Yanli Zhao, Andrew Gu, Rohan Varma, Liang Luo, Chien-Chin Huang, Min Xu, Less Wright, Hamid Shojanazeri, Myle Ott, Sam Shleifer, et al. Pytorch fsdp: experiences on scaling fully sharded data parallel. *arXiv preprint arXiv:2304.11277*, 2023.

Systems and Algorithms for Convolutional Multi-Hybrid Language Models at Scale

Supplementary Material

Contents

Author Contributions

M.P. conceptualized the research; M.P. implemented the first version of the training infrastructure; J.K., E.N., D.W.R., G.Bri., B.Y., A.T., D.P.B., G.Bro., B.L.H., M.P., contributed to pretraining infrastructure; J.K., E.N., G.Bri., and M.P. designed pretraining experiments; M.P., E.N., designed the architectures; M.P. derived and implemented the first version of the two-stage block kernel; J.K. optimized the kernel and wrote the backward pass; S.M., M.P., extended the theory in section 3; D.W.R., B.Y., M.P., derived and implemented context parallelism for hyena and attention layers; D.W.R., derived and implemented the point-to-point scheme for spatial and FFT convolutions; M.P., G.Bri., A.V., wrote and optimized the inference stack; A.X.L., contributed to inference stack; C.R., P.D.H., B.L.H., S.E., M.P., supervised the project.

A Appendix: Additional Results

A.1 Additional Algorithms for Direct Convolution on Tensor Cores

Modern GPU accelerators include *tensor cores* capable of high-throughput dense matrix multiplications (`GEMM`). In the context of the two-stage block algorithm described in Section 3.2, an effective strategy is to recast small matrix-vector products into larger `GEMM` kernels that better exploit these tensor units. Below, we focus on a *single group of d_g channels*, which all share one depthwise FIR filter of length ℓ_h. We assume $\ell_h \le 2\,\ell_b$, so only two $(\ell_b \times \ell_b)$ Toeplitz blocks are needed for the filter:

$$H_0, \quad H_1 \in \mathbb{R}^{\ell_b \times \ell_b}.$$

These blocks respectively handle the "current-chunk" taps and the "spillover" taps from the preceding chunk. We provide here a different algorithm to transform direct convolutions into `GEMMs`, even without grouping. The idea is to parallelize across the chunks, rather than channels.

Block decomposition and dimensions. Let $\hat{X}_n \in \mathbb{R}^{\ell_b \times d_g}$, $\hat{Y}_n \in \mathbb{R}^{\ell_b \times d_g}$ denote the n-th input and output blocks (or *chunks*) for the group. Here, ℓ_b is the block size along the time dimension, and d_g is the number of channels in the group. According to the two-stage block convolution framework (cf. Section 3.2), each output block is given by

$$\hat{Y}_n \;=\; H_0\,\hat{X}_n \;+\; H_1\,\hat{X}_{n-1}, \qquad n \;=\; 0, 1, \ldots, \left\lceil \tfrac{\ell}{\ell_b} \right\rceil - 1, \tag{11}$$

with the convention that $\hat{X}_{-1} = 0$ (i.e., the "previous chunk" is zero for $n = 0$). In particular, H_0 and H_1 are constant for all chunks n once the filter has been fixed. Differently from the grouped algorithm proposed in the main text, this approach parallelizes across chunks, requiring the a reshape on the input.

Mapping to tensor cores. Equation (11) naturally translates into two matrix-matrix multiplications plus an elementwise addition:

$$\hat{Y}_n \;=\; \underbrace{H_0\,\hat{X}_n}_{(\ell_b \times \ell_b)\,\times\,(\ell_b \times d_g)} \;+\; \underbrace{H_1\,\hat{X}_{n-1}}_{(\ell_b \times \ell_b)\,\times\,(\ell_b \times d_g)}.$$

Because H_0 and H_1 remain unchanged for all n, the following procedure can be used to implement (11) efficiently on GPUs:

1. **Filter preload** Load H_0 and H_1 from global memory into low-latency on-chip memory (e.g., shared memory). This is a one-time overhead amortized across all chunks.
2. **Chunk read** Read the current chunk $\hat{X}_n$ (and the previous chunk $\hat{X}_{n-1}$ if $n > 0$) from global memory into local registers or shared memory.
3. **Tensor-core `GEMM`** Perform the matrix multiplications $H_0\,\hat{X}_n$ and $H_1\,\hat{X}_{n-1}$ on the tensor cores, accumulating the two partial results into $\hat{Y}_n$.
4. **Output writeback** Write the output block $\hat{Y}_n$ to global memory.

Cost model The floating-point cost per chunk follows directly from (11). Each chunk output $\hat{Y}_n$ requires two matrix multiplications of dimension $(\ell_b \times \ell_b)$ by $(\ell_b \times d)$ for a total of $2\,\ell_b^2\,d$ floating-point operations. Summing over all $\lceil \ell/\ell_b \rceil$ chunks, the total floating-point operations per sequence (for a single group) is $2\,\ell_b^2\,d\,\lceil \ell/\ell_b \rceil$.

A.2 Context Parallel Methods

A.2.1 All-to-All Attention

In the case of self-attention, where channels `D` are split into groups (heads), and operations happen independently over each head, the sharding is done such that an integer number of heads is held on each device. Once the operation is complete, a second `a2a` exchange is used to return to the input shape of $[\texttt{1}, \texttt{H}, \texttt{L//N}_{\texttt{CP}}]$. `a2a` context parallelism has been used in attention parallelization strategies, such as DeepSpeed Ulysses (Jacobs et al., 2023).

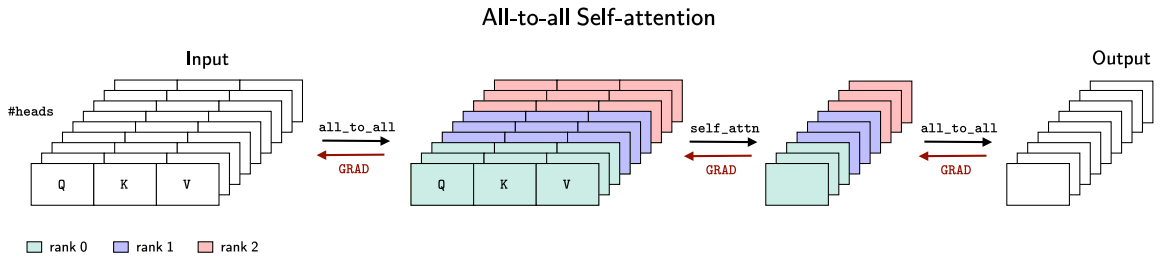


Figure A.1: Diagram of computation and communication in `all-to-all self-attention`. Attention heads are split across devices such that an integer number of heads is held on each device. Then, self-attention is performed locally on each device and subsequently, the output is exchanged across all ranks to reconstruct the original shape of the input.

A.2.2 Point-to-Point Attention

Ring attention methods propose a `p2p` solution for self-attention. For simplicity, let us assume that the query, key and value projections of the input are all of same size as the input shape $[\mathtt{D}, \mathtt{L}//\mathtt{N_{cp}}]$. Ring-Attention arranges CP ranks in a ring, each holding a portion of the query. Consequently, Ring Attention (RA) (Liu et al., 2023) passes portions of the key and value tensors, each of shape $[\mathtt{D}, \mathtt{L}//\mathtt{N_{cp}}]$ following the ring arrangement. At each stage, self-attention is performed between the query stationed in each device and the transmitted key-value portions. Global statistics are updated online on each device based on each upcoming key value portions to compute the softmax. After $\mathtt{N_{cp}}$ steps, each consisting of $\mathtt{N_{cp}}$ parallel `p2p` calls, the shards of the queries held on each device will have seen all key-value portions, and each device will hold a shard of the final attention result. Combining this result with the online softmax operation, full attention operation can be concluded without ever holding the whole sequence on a single device.

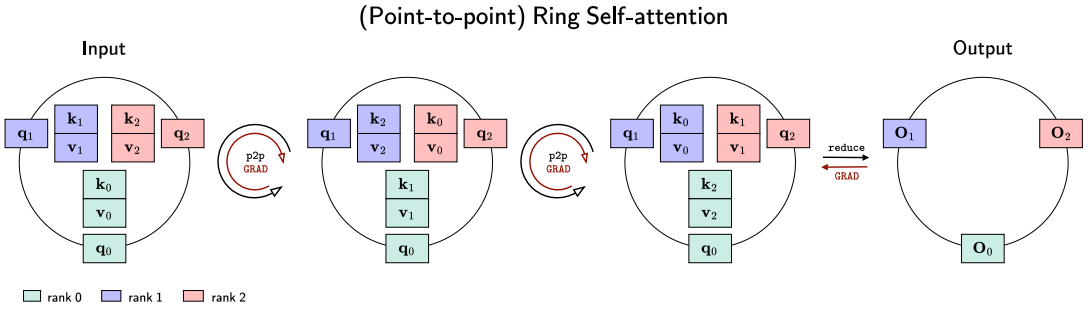


Figure A.2: Diagram of computation and communication in `(point-to-point) ring self-attention`. The key and value chunks are communicated in a ring arrangement until all devices have seen all of the chunks. Once all chunks have been seen, partial results can be reduced to complete the attention operation.

A.2.3 Considerations for Causal Models.

When causal transformers are used, e.g., for autoregressive tasks, computations follow a triangular structure. Brandon et al. (2023) noticed that this triangular structure causes load imbalances in Ring-Attention in causal autoregressive settings. To overcome this problem, they propose to shard the input across CP devices using 2× as many shards as CP ranks and accommodate 2 shards on each CP rank using their introduced *striped ordering*. For example, for $\{\mathbf{x}_j\}_{j=1}^{2\mathtt{N_{cp}}}$ shards with $\mathtt{N_{cp}}$=4, shards are organized as: $[\mathbf{x}_0, \mathbf{x}_4]$, $[\mathbf{x}_1, \mathbf{x}_5]$, $[\mathbf{x}_2, \mathbf{x}_6]$, $[\mathbf{x}_3, \mathbf{x}_7]$ on each rank respectively. Dubey et al. (2024) proposed an improved load balancing strategy for the training of Llama-3 by distributed shards following a *zig-zag splitting*. Specifically, for the same case with $\{\mathbf{x}_j\}_{j=1}^{2\mathtt{N_{cp}}}$ shards and $\mathtt{N_{cp}}$=4, shards are organized as: $[\mathbf{x}_0, \mathbf{x}_7]$, $[\mathbf{x}_1, \mathbf{x}_6]$, $[\mathbf{x}_2, \mathbf{x}_5]$, $[\mathbf{x}_3, \mathbf{x}_4]$ on each rank respectively.

For the training of StripedHyena 2, we adopt the zig-zag splitting of Dubey et al. (2024). We note that the choice of sharding strategy has no major implications for any of the `a2a` and `p2p` CP strategies introduced before. For `a2a`, the zigzag arrangement must now be considered when reconstructing the sequence. For `p2p`, two buffers must be kept on each rank, each representing the attention operation on each split.

A.2.4 Point-to-point FFT Convolutions

As mentioned in the main text, the idea behind `p2p` Context Parallelism is to perform an operation over an input of shape $[\mathtt{1}, \mathtt{D}, \mathtt{L}]$ sharded along the sequence dimension onto $\mathtt{N_{cp}}$ devices, each holding a shard of shape $[\mathtt{1}, \mathtt{D}, \mathtt{L}//\mathtt{N_{cp}}]$, without ever holding the whole sequence on a single device.

This section heavily relies on the FFT derivation and the multiple Radix-N algorithms presented in Takahashi (2019). We refer interested readers to that textbook for additional details.

Primer on the Fast Fourier Transform. The name Fast Fourier Transform (FFT) comes from an algorithm that, as its name indicates, allows computing the Discrete Fourier Transform fast. Computing the Discrete Fourier Transform naively has quadratic complexity. However, the FFT is able to achieve the same result with $\mathcal{O}(\mathtt{L}\log\mathtt{L})$ complexity, by using a divide and conquer formulation.

To understand the FFT, we start from the Discrete Fourier Transform (DFT) defined as:

$$y(k) = \mathrm{DFT}_l(x) = \sum_{j=0}^{l-1} x(j)\omega_l^{jk}, \quad 0 \le k \le l-1, \tag{12}$$

where $\omega_l{=}e^{-2\pi i/l}$ and $i{=}\sqrt{-1}$, applied to an input x of length $l{=}4$. The DFT of x can be calculated as:

$$\begin{bmatrix} y(0) \\ y(1) \\ y(2) \\ y(3) \end{bmatrix} = \begin{bmatrix} \omega^0 & \omega^0 & \omega^0 & \omega^0 \\ \omega^0 & \omega^1 & \omega^2 & \omega^3 \\ \omega^0 & \omega^2 & \omega^4 & \omega^6 \\ \omega^0 & \omega^3 & \omega^6 & \omega^9 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}. \tag{13}$$

Importantly, the terms ω_l^{jk} are not independent. In fact, there is a relation $\omega_l^{jk}{=}\omega_l^{jk \bmod l}$, which we can use to rewrite Eq. 13 as follows:

$$\begin{bmatrix} y(0) \\ y(1) \\ y(2) \\ y(3) \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega^1 & \omega^2 & \omega^3 \\ 1 & \omega^2 & \omega^0 & \omega^2 \\ 1 & \omega^3 & \omega^2 & \omega^1 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}. \tag{14}$$

At this point, we can observe that there are several values that repeat themselves. Using some algebra and reorganizing the position of the output positions in the vector, we arrive at the following decomposition:

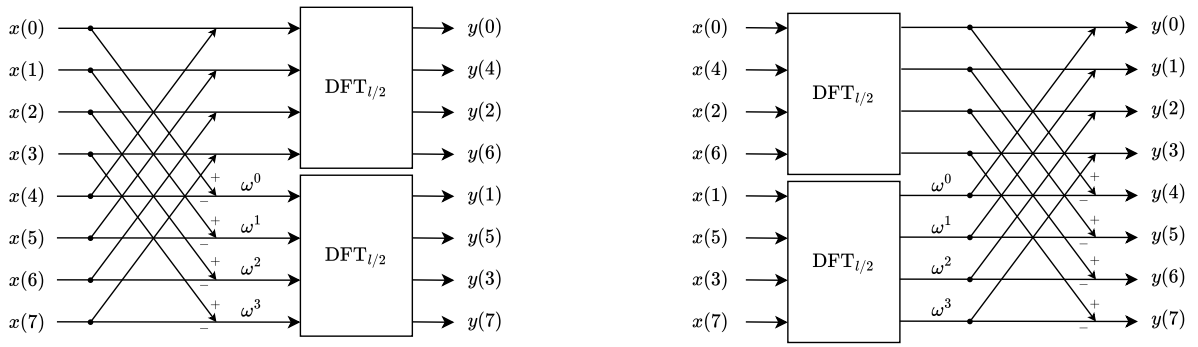
$$\begin{bmatrix} y(0) \\ y(2) \\ y(1) \\ y(3) \end{bmatrix} = \begin{bmatrix} 1 & \omega^0 & 0 & 0 \\ 1 & \omega^2 & 0 & 0 \\ 0 & 0 & 1 & \omega^1 \\ 0 & 0 & 1 & \omega^3 \end{bmatrix} \begin{bmatrix} 1 & 0 & \omega^0 & 0 \\ 0 & 1 & 0 & \omega^0 \\ 1 & 0 & \omega^2 & 0 \\ 0 & 1 & 0 & \omega^2 \end{bmatrix} \begin{bmatrix} x(0) \\ x(1) \\ x(2) \\ x(3) \end{bmatrix}. \tag{15}$$

Looking at this decomposition, we observe two important things: First, the first of the matrices is a blockwise matrix, corresponding to two DFTs of half the size of the initial sequence length. In the general case, it holds that, an l-point DFT can be decomposed onto two $\frac{l}{2}$-point DFTs followed by some arithmetic operations.[9] Specifically, for an input x of length l, divided onto two chunks $x(j)$, $x(j+l/2)$, its DFT is given by:

$$\begin{aligned} y(k) &= \mathrm{DFT}_{l/2}\big(x(j) + x(j+l/2)\big) \\ y(k+1) &= \mathrm{DFT}_{l/2}\left(\omega_l^j\big(x(j) - x(j+l/2)\big)\right). \end{aligned} \tag{16}$$

Secondly, it is important to note that while the values of x are organized sequentially in Eq. 15, the values of its DFT y have been permuted following a bit reversal order. In the general case, there exist two types of FFT depending on whether the input is assumed to be organized sequentially –in which case the output is bit reversed–, or whether the input is assumed to be bit reversed –in which case the output is organized sequentially. These are known as *Decimation-in-Frequency* (DiF) and *Decimation-in-Time* (DiT) FFT algorithms, respectively. Equation 16 depicts the DiF FFT algorithm.

[9] While this observation is already enough for us to describe `p2p` FFT convolutions, it is worth noticing that the FFT repeats this process until the input can no longer be split, i.e., $l{=}2$. This recursive split procedure is that makes the FFT fast.



(a) DiF-FFT butterfly diagram

(b) DiT-FFT butterfly diagram

Figure A.3: Butterfly diagrams for DiT and DiF FFTs.

Butterfly diagrams. A powerful, intuitive way to visualize the flow of data in DiF and DiT FFTs are butterfly diagrams. Butterfly diagrams illustrate the data exchange in DiF and DiT FFTs, which are based on two operations:

$$\begin{aligned} &\textbf{DiF Butterfly:} \\ &X = X + Y, \\ &Y = (X - Y)\ \omega^j \end{aligned} \tag{17}$$

$$\begin{aligned} &\textbf{DiT Butterfly:} \\ &X = X + \omega^j\ Y, \\ &Y = X - \omega^j\ Y \end{aligned} \tag{18}$$

The butterfly diagrams of DiF and DiT Fast Fourier Transforms are shown in Fig. A.3.

The inverse DFT. The inverse DFT (iDFT) is defined as:

$$x(j) = \text{iDFT}_l(y) = \frac{1}{l}\sum_{k=0}^{l-1} y(k)\omega_l^{-jk}, \quad 0 \le j \le l-1. \tag{19}$$

When compared to the DFT (Eq. 12), we observe that the only differences are (i) the normalization factor $\frac{1}{l}$, and (ii) the minus sign in the ω^{-jk}. Luckily, this means that the exact same operation as well as butterfly diagrams can be used, with two minor modifications: all ω^j terms are exchanged by an ω^{-j} term, and the normalization by $\frac{1}{l}$ must be considered.

A.2.5 Point-to-point FFT Convolutions with CP=2

Looking at the FFT formulation, we can observe that the FFT is computed by performing independent FFTs on two independent splits of the input, followed by some point-wise operations over these splits. Interestingly, this setting exactly resembles the case of a distributed `p2p` setting, where each split is held in a different device. Nonetheless, there is an important impediment due to the organization of the data *after* the FFT is performed. Assuming the conventional sequential splitting of the input across CP ranks, once a distributed FFT is performed, the output of the FFT would be bit-reversed across all ranks. Consequently, in order to restore the sharding distribution of the input, an additional `a2a` call would be required.

Luckily, this permutation of the output is not a problem when performing FFT Convolutions. Since the FFT convolution is composed of an FFT followed by an inverse FFT, it is sufficient to use a forward (DiF) FFT –that generates a bit-reversed output– followed by a DiF inverse FFT algorithm to generate an output that follows the same organization as the input. As a result, after the FFT convolution is finished, both the input and the output will be sharded in the same manner.

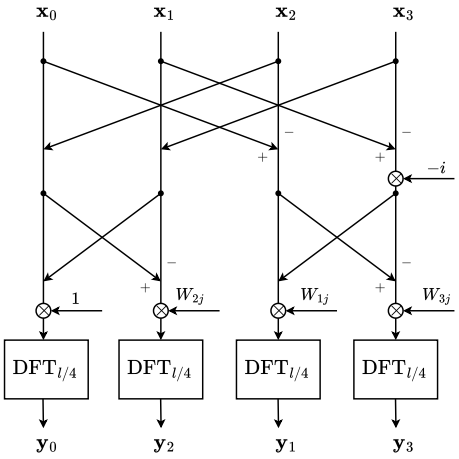
Listing 1 shows a minimalist reference implementation of the `p2p` FFT convolution with simulated sharding for $\texttt{N}_{\texttt{cp}}$=2.

A.3 Extending `p2p` FFT Convolutions to larger CP sizes

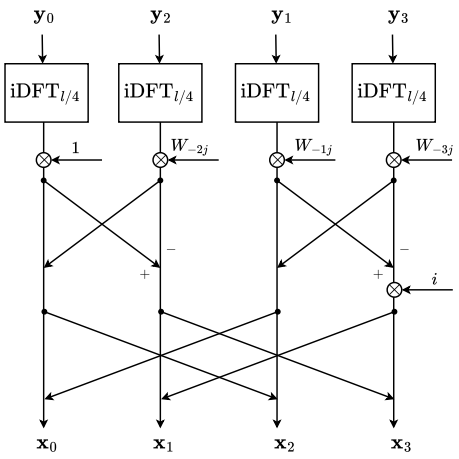
The previous section illustrates how a `p2p` FFT convolution can be computed for a CP group with $\texttt{N}_{\texttt{cp}}$=2 devices. In this section, we show how to extend this procedure to CP groups with $\texttt{N}_{\texttt{cp}}$>2 devices.

Listing 1 Minimal implementation of the FFT and iFFT with a simulated CP group of size 2.

```python
def bit_reversal(xarray: torch.Tensor, size: int, log2length: int):
    """Applies a bit-reversal permutation to the input array."""
    reversed_indices = vectorized_bit_reversal_indices(size, log2length, xarray.device)
    return xarray[..., reversed_indices]

def dif_radix2_fft(x: torch.Tensor):
    """Applies the radix-2 Decimation-in-Frequency (DIF) FFT to the input.

    Parameters:
    - x (torch.Tensor): The input tensor of shape [B, H, L]

    Returns:
    - tuple(torch.Tensor, torch.Tensor): The sharded bit-reversed fft of the input.
    """
    # Split the input (to simulate a CP group of size 2).
    _x_0, _x_1 = rearrange(x,"... (n1 n2) -> ... n1 n2", n1 = 2, n2 = N // 2).unbind(dim=-2)

    # Twiddle factors
    k = torch.arange(N // 2, device=x.device)
    W = torch.exp(-1j * 2.0 * torch.pi * k / N)

    # Apply butterfly operations
    x_0 = _x_0 + _x_1
    x_1 = (_x_0 - _x_1) * W

    # Compute FFT on both halves.
    fft_x_0 = torch.fft.fft(x_0, dim=-1), N // 2, int(math.log2(N // 2)))
    fft_x_1 = torch.fft.fft(x_1, dim=-1), N // 2, int(math.log2(N // 2)))
    return fft_x_0, fft_x_1

def dif_radix2_ifft(fft_x_0, fft_x_1: torch.Tensor):
    """Applies the radix-2 iFFT assuming that the input is a bit-reversed FFT, and returns
       a non-reversed input tensor, i.e., x = dif_radix2_ifft(dif_radix2_fft(x))."""

    # Compute iFFT on both halves (it internally performs normalization by 1 / n // 2).
    _x_0 = torch.fft.ifft(fft_x_0, dim=-1)
    _x_1 = torch.fft.ifft(fft_x_1, dim=-1)

    # Twiddle factors
    k = torch.arange(N // 2, device=_x_1.device)
    W = torch.exp(1.j * 2.0 * torch.pi * k / N)

    # Apply butterfly operations
    x_0 = _x_0 + W * _x_1
    x_1 = _x_0 - W * _x_1

    # Normalize by seq. length of this stage (2) & concat results for verification.
    return 0.5 * torch.cat([x_0, x_1], dim=-1)

assert torch.allclose(x, dif_radix2_ifft(dif_radix2_fft(x)).real)
```



(a) `p2p` FFT for $\texttt{N}_{\texttt{cp}}$=4 devices.

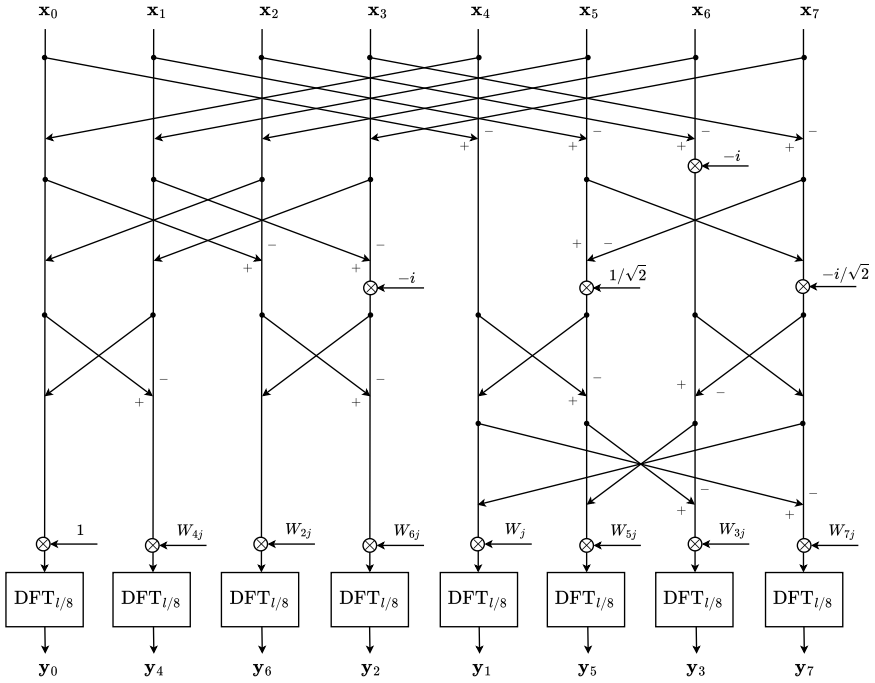


(b) `p2p` inverse FFT for $\texttt{N}_{\texttt{cp}}$=4 devices.

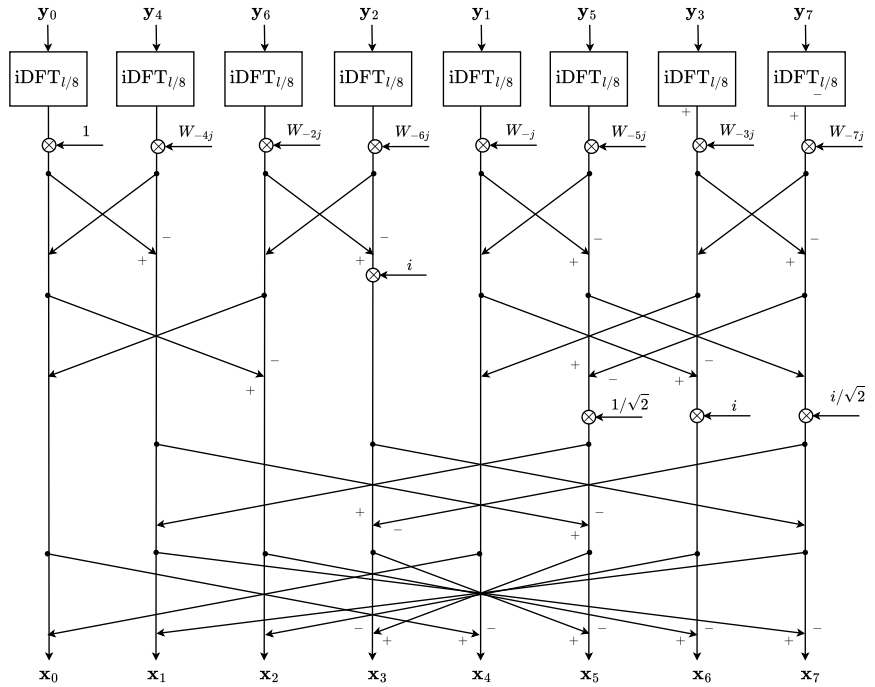
Figure A.4: Butterfly diagrams for the distributed `p2p` FFT and inverse FFT for $\texttt{N}_{\texttt{cp}}$=4. $\mathbf{x}_0, ..., \mathbf{x}_3$ represent the shards of the input $\mathbf{x}$ held on each CP rank. Devices are represented by the vertical lines. Note that after the FFT, the outputs $\{\mathbf{y}_j\}_{j=0}^{3}$ are bit-reversed over CP ranks. However, after combining it with the inverse FFT, the same sharding distribution as the input is obtained. FFT convolutions are computed by performing the FFT of both the input and the (sharded) convolutional kernel, multiplying them in the Fourier domain, and returning the result back to the spatial domain.

Radix-`N` FFT. Before we continue, we must introduce the concept of a Radix-`N` FFT algorithm. Simply put, a Radix-`N` FFT algorithm computes a l-point FFT by decomposing it onto `N` independent $l//\texttt{N}$-point FFTs followed by pointwise operations. In other words, a Radix-`N` FFT algorithm generalizes the splitting process illustrated in the previous section for a value of `N`=2 to values `N`>2. Furthermore, just as for the Radix-2 FFT algorithm, there exist DiT and DiF implementations for several values of `N`. Takahashi (2019) provides an exceptional description of Radix-`N` FFT algorithms for $\texttt{N} \in [3, 4, 5, 8]$, and many algorithms exist for many other values of `N`, e.g., `N`=16 (Bouguezel et al., 2004; Huang & Chen, 2012). An important difference from the `N`=2 case, is that the Radix-`N` algorithms introduces `N` different sets of twiddle factors. For `N`=2, we had two sets of values $W_{0j} = [\omega_{0\times 0}, ..., \omega_{(\frac{l}{2}-1)\times 0}] = [1, ..., 1]$, and $W_{1j} = [\omega_{0\times 1}, ..., \omega_{(\frac{l}{2}-1)\times 1}] = [1, \omega^1, \omega^2, ..., \omega^{\frac{l}{2}}]$ applied to the first and second splits, respectively (Fig. A.3). For a general value of `N`, we utilize `N` sets of twiddle factors, $\{W_{nj}\}_{n=0}^{\texttt{N}-1}$, defined as $W_{nj} = [\omega_{0\times n}, ..., \omega_{(\frac{l}{\texttt{N}}-1)\times n}]$.

Extension to $\texttt{N}_{\texttt{cp}}$>2 devices. Following the same formulation used for `N`=2, we can compose a DiF Radix-`N` FFT and an inverse DiF Radix-`N` FFT to perform convolutions in a distributed setting. Specifically, given an input of shape $[1, \texttt{D}, \texttt{L}]$ sharded on a CP group with $\texttt{N}_{\texttt{cp}}$ devices, the `p2p` FFT convolution is implemented by using Radix-$\texttt{N}_{\texttt{cp}}$ (DiF) FFT and inverse (DiF) FFT algorithms to compute the FFT convolution without holding the whole sequence in a single device. Schematic butterfly diagrams for the implementation of `p2p` FFT convolutions for $\texttt{N}_{\texttt{cp}}$=4, and $\texttt{N}_{\texttt{cp}}$=8 devices are provided in Figs. A.4, A.5.



(a) `p2p` FFT for $\texttt{N}_{\texttt{cp}}$=8 devices.



(b) `p2p` inverse FFT for $\texttt{N}_{\texttt{cp}}$=8 devices.

Figure A.5: Butterfly diagrams for the distributed `p2p` FFT and inverse FFT for $\texttt{N}_{\texttt{cp}}$=8. $\mathbf{x}_0, ..., \mathbf{x}_7$ represent the shards of the input $\mathbf{x}$ held on each CP rank. Devices are represented by the vertical lines. Note that after the FFT, the outputs $\{\mathbf{y}_j\}_{j=0}^{7}$ are bit-reversed over CP ranks. However, after combining it with the inverse FFT, the same sharding distribution as the input is obtained. FFT convolutions are computed by performing the FFT of both the input and the (sharded) convolutional kernel, multiplying them in the Fourier domain, and returning the result back to the spatial domain.

A.4 Two-Pass Algorithm

Backward kernel: To compute filter gradients in the backward pass, one requires global accumulation. Instead of limiting the computation to a single kernel, we implement gradient calculation using back-to-

back kernels. The first performs a partial accumulation of the filter gradient by block, maintaining the same overall structure as the forward kernel, while the second kernel calculates the final result as a reduction of these partial gradients. Importantly, we take care to write out the partially accumulated gradients in coalesced format to enable a simple vectorized reduction as a second step.

Fast materialization of Toeplitz factors In the listing below, we report code to efficiently materialize the Toeplitz factors using Triton.

B Appendix: Supplementary Figures

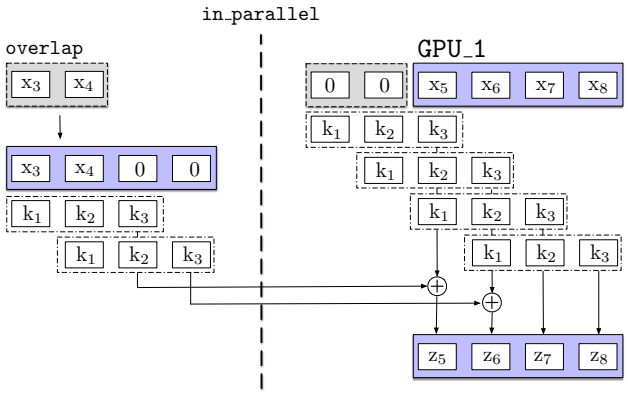


Figure B.1: Diagram of computation in our overlapped `point-to-point convolution` scheme.

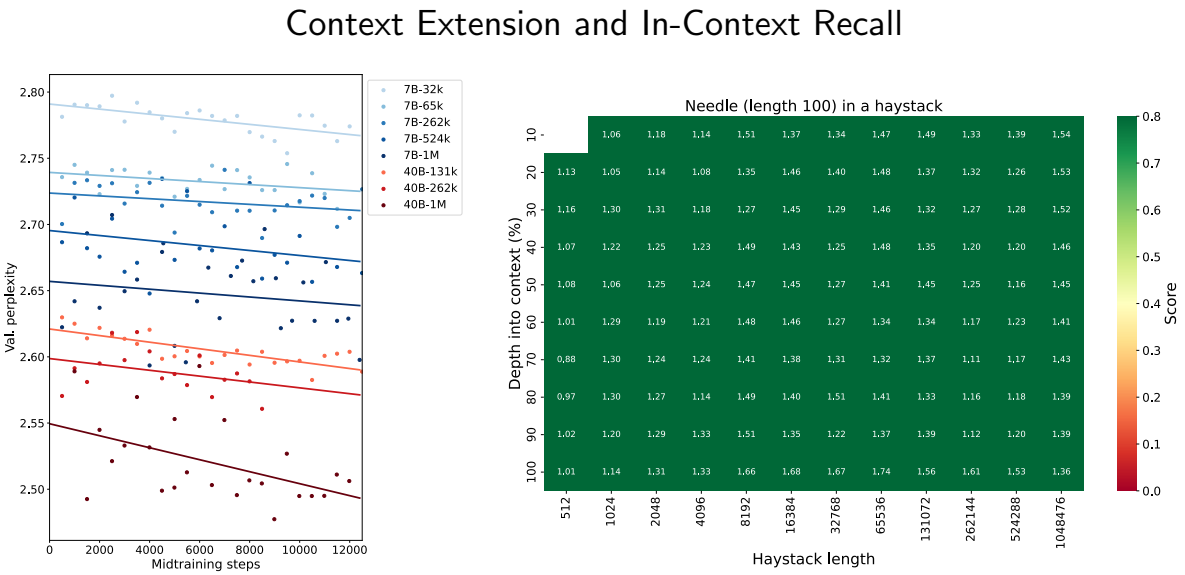


Figure B.2: **[Left]:** Validation perplexity on `OpenGenome2` after midtraining extension with different techniques, at model scales of 7B and 40B. The extensions are performed on the base Evo 2 7B and 40B models. We also provide a linear fit at each scale. **[Right]:** Recall performance of 40B 1M measured via the needle-in-the-haystack task described in (Brixi et al., 2025)

Listing 2 Masked loading of Toeplitz matrix factors $T_h^{(0)}$ and $T_h^{(1)}$ in Triton.

```
import triton
import triton.language as tl

@triton.jit
def toeplitz_idx(
    FILTER_LEN: tl.constexpr,
    CHUNK_SIZE: tl.constexpr,
    TOEPLITZ_TYPE: tl.constexpr = "toeplitz",
):

    if TOEPLITZ_TYPE == "zeroth_factor":
        r_idx = tl.arange((FILTER_LEN - 1), CHUNK_SIZE + (FILTER_LEN - 1))[None, :]
    elif TOEPLITZ_TYPE == "first_factor":
        r_idx = (
            tl.arange((FILTER_LEN - 1), CHUNK_SIZE + (FILTER_LEN - 1))[None, :]
            - CHUNK_SIZE
        )
    else:
        tl.static_assert(False, "Invalid ToeplitzType")
    c_idx = tl.arange(0, CHUNK_SIZE)[:, None]
    idx = r_idx - c_idx
    return idx

@triton.jit
def load_toeplitz(
    h_ptr,
    FILTER_LEN: tl.constexpr,
    CHUNK_SIZE: tl.constexpr,
):
    t_idx = toeplitz_idx(FILTER_LEN, CHUNK_SIZE, "toeplitz")
    mask = (0 <= t_idx) & (t_idx < FILTER_LEN)

    T = tl.load(
        h_ptr + group_num * FILTER_LEN + t_idx,
        mask=mask,
        other=0.0,
        eviction_policy="evict_last",
    )

    return T
```

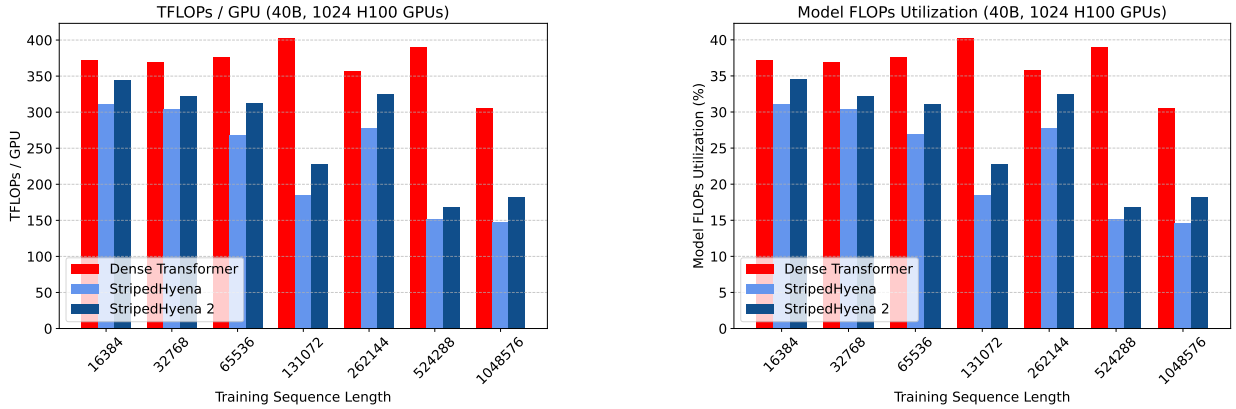


Figure B.3: MFU and TFLOPS / s / GPU of 40B models with same distributed configuration and different architectures. For StripedHyena 2, we achieve peak MFU of 34% at 16K context. See Table C.1 for details on the measurement protocol.

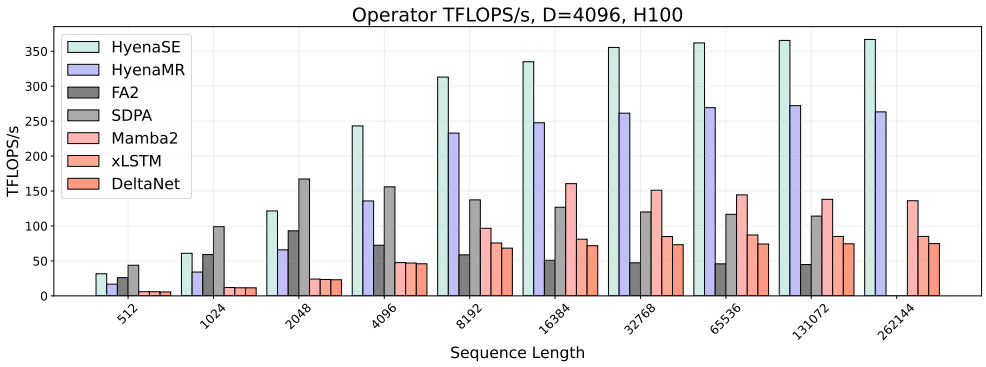


Figure B.4: Forward TFLOPs / second of Hyena-SE, Hyena-MR and other common operators in architecture design.

C Appendix: Methods

Setting	Value
Tensor Parallel	2, 2, 8, 8, 16, 16, 32
Sequence Parallel	True
Context Parallel	1, 1, 1, 1, 1, 2, 2
Global Batch Size	4M tokens
Hardware	H100 SXM
GPU Count	256

Setting	Value
Tensor Parallel	8, 8, 8, 8, 16, 32, 64
Sequence Parallel	True
Context Parallel	1, 1, 1, 2, 2, 2, 2
Global Batch Size	8M tokens
Hardware	H100 SXM
GPU Count	2048

Table C.1: **[Left]:** Baseline distributed configuration used for 7B parameter model measurements in Figure 2.2 at 16K, 33K, 65K, 131K, 262K, 524K, 1M sequence length. **[Right]:** Baseline distributed training configuration used for 40B parameter model measurements in Figure 2.2 at 16K, 33K, 65K, 131K, 262K, 524K, 1M sequence length. We also verify scaling on 2048 with the same settings, doubling batch size 16M. We observe similar throughput multipliers even on 2048.

C.1 Pretraining Experiments

Methodology For all training experiments, we use Savanna. Configuration files are available in the repository, at the following frozen commit hash: `5f9fdb`. We train on the `OpenGenome 2` dataset (Brixi

et al., 2025). For throughput measurements, we report the settings in dedicated tables (Table C.1). We use critical batch size estimation (McCandlish et al., 2018) to determine a batch size for the 7 billion parameter runs. We train in mixed precision using FP8 for dense layers and norms.

Effect of grouped convolution We train 7 billion parameter StripedHyena 2 SE-MR-LI models on 400 billion tokens `OpenGenome2`, with group sizes 1 (baseline) and 16, and observe no significant difference in convergence. We also explore the effect of filter grouping in Hyena on smaller models, including simpler hybrids using only Hyena-SE and Hyena-MR. Group sizes larger than 64 introduce minimal degradation in quality, more visible in smaller models. A smaller number of independent filters also reduces the granularity of the regularization in Hyena-MR, initialized to span different values across filters. The configuration files to replicate the experiments are available in Savanna: `configs/7b-final-test/`.

Context extension Full configuration files are available in `configs/context-scaling`. We evaluate recall during midtraining context extension with the needle-in-a-haystack task described in (Brixi et al., 2025). Results are shown in Figure B.2.

Replacing feed-forward-layers with convolutions In early designs, we also trained variants of the architecture where every feed-forward layer (MLP, SwiGLU) following every hyena or MHA operator had been replaced with Hyena-SE. We observe generally improved convergence for these models with a small decrease in throughput. Given these findings, coupled with the higher throughput of Hyena-SE compared to MHA or state-space models, we expect future multi-hybrid designs to also optimize the ratio of MLPs and Hyena-SE (or their MoE variants). Configuration files are available at: `configs/model/evo2/ablations`.

C.2 Profiling

Operator profiling All operators use their official kernels[10]. For DeltaNet, we report the latency of the kernel provided in the Flash Linear Attention (Yang & Zhang, 2024) repository. For Mamba2 (Dao & Gu, 2024), we use the official kernels provided by the authors.

Figure 3.2 and B.4 shows the results, with Hyena-SE and Hyena-MR providing graceful scaling to longer sequence lengths, with high throughput even at batch size 1. Figure 3.1 shows a direct comparison of latencies and TFLOPS / second for the Hyena-MR with different underlying implementations. Direct convolutions using the proposed two-stage approach generally outperform PyTorch convolutions.

[10] We use the mLSTM kernels developed for xLSTM-7B.