Transformers are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality

Tri Dao [*1] **Albert Gu** [*2]

Abstract

While Transformers have been the main architecture behind deep learning's success in language modeling, state-space models (SSMs) such as Mamba have recently been shown to match or outperform Transformers at small to medium scale. We show that these families of models are actually quite closely related, and develop a rich framework of theoretical connections between SSMs and variants of attention, connected through various decompositions of a well-studied class of structured *semiseparable matrices*. Our state space duality (SSD) framework allows us to design a new architecture (**Mamba-2**) whose core layer is an a refinement of Mamba's selective SSM that is 2-8× faster, while continuing to be competitive with Transformers on language modeling.

1 Introduction

Transformers, in particular decoder-only models (e.g. GPT (Brown et al., 2020), Llama (Touvron et al., 2023)) which process input sequences in a causal fashion, are one of the main drivers of modern deep learning's success. Numerous approaches attempt to approximate the core attention layer to address its efficiency issues (Tay et al., 2022), such as scaling quadratically in sequence length during training and requiring a cache of size linear in sequence length during autoregressive generation. In parallel, a class of alternative sequence models, structured state-space models (SSMs), have emerged with linear scaling in sequence length during training and constant state size during generation. They show strong performance on long-range tasks (e.g. S4 (Gu et al., 2022a)) and recently matched or beat Transformers on language modeling (e.g. Mamba (Gu & Dao, 2023)) at small to moderate scale. However, the development of SSMs have appeared disjoint from the community's collective effort to improve Transformers, such as understanding them theoretically as well as optimizing them on modern hardware. As a result, it is more difficult to understand and experiment with SSMs compared to Transformers, and it remains challenging to train SSMs as efficiently as Transformers from both an algorithmic and systems perspective.

Our main goal is to develop a rich body of theoretical connections between structured SSMs and variants of attention. This will allow us to transfer algorithmic and systems optimizations originally developed for Transformers to SSMs, towards the goal of building foundation models that perform better than Transformers while scaling more efficiently in sequence length. A milestone contribution in this direction was the **Linear Attention (LA)** framework (Katharopoulos et al., 2020), which derived a connection between autoregressive attention and linear RNNs by showing the equivalence between "dual forms" of quadratic kernelized attention and a particular linear recurrence. This duality allows new capabilities such as the ability to have both efficient parallelizable training and efficient autoregressive inference. In the same spirit, this paper provides multiple viewpoints connecting linear-complexity SSMs with quadratic-complexity forms to combine the strengths of SSMs and attention.[1]

State Space Duality. Our framework connecting structured SSMs and variants of attention, which we call **structured state space duality** (SSD), is made through the abstractions of **structured matrices**: matrices with subquadratic parameters and multiplication complexity. We develop two broad frameworks for representing sequence models, one as matrix transformations and one as tensor contractions, which each reveal different perspectives of the duality. Our technical contributions include:

- We show an equivalence between state space models and a well-studied family of structured matrices called **semiseparable matrices**. This connection is at the heart our framework, revealing new properties and algorithms for SSMs. A central message of this paper is that *different methods of computing state space models can be reframed as various matrix multiplication algorithms on structured matrices*.
- We significantly improve the theory of linear attention (Katharopoulos et al., 2020). We first provide an incisive proof of its recurrent form through the language of tensor contractions, and then generalize it to a new family of **structured masked attention (SMA)**.
- We connect SSMs and SMA, showing that they have a large intersection that are duals of each other, possessing both SSM-like linear and attention-like quadratic forms.

[*]Equal contribution [1]Department of Computer Science, Princeton University [2]Machine Learning Department, Carnegie Mellon University. Correspondence to: Tri Dao<tri@tridao.me>, Albert Gu <agu@cs.cmu.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

[1]Technically speaking, these connections only relate to certain flavors of attention; the title of this paper is an homage to Katharopoulos et al. (2020) who first showed that "Transformers are RNNs".

Beyond its intrinsic theoretical value, our framework opens up a broad set of directions for understanding and improving sequence models.

Efficient Algorithms. First and most importantly, our framework exposes new efficient and easily-implementable algorithms for computing SSMs. We introduce a new **SSD algorithm**, based on block decompositions of semiseparable matrices, that takes advantage of both the linear SSM recurrence and quadratic dual form, obtaining optimal tradeoffs on all main efficiency axes (e.g. training and inference compute, memory usage, and ability to leverage matrix multiplication units on modern hardware). A dedicated implementation of SSD is $2-8\times$ faster than the optimized selective scan implementation of Mamba, while simultaneously allowing for much larger recurrent state sizes ($8\times$ the size of Mamba or even higher, with minimal slowdown). SSD is highly competitive with optimized implementations of softmax attention (FlashAttention-2 (Dao, 2024)), crossing over at sequence length 2K and $6\times$ faster at sequence length 16K.

Mamba-2. Additionally, inspired by the connection between SSMs and Transformers, we slightly modify the neural network architecture of Mamba by moving all data-dependent projections to occur in parallel at the beginning of the block. The combination of the modified parallel Mamba block, together with using SSD as the inner SSM layer, results in the **Mamba-2** architecture. We investigate Chinchilla scaling laws for Mamba-2 in the same setting as Mamba, finding that it Pareto dominates Mamba and Transformer++ in both perplexity and wall-clock time. We additionally train a family of Mamba-2 models at varying sizes on the Pile, showing that it matches or outperforms Mamba and open source Transformers on standard downstream evaluations. For example, Mamba-2 with 2.7B parameters trained on 300B tokens on the Pile outperforms Mamba-2.8B, Pythia-2.8B and even Pythia-6.9B trained on the same dataset.

Section 4 empirically validates Mamba-2 on language modeling, training efficiency, and a difficult multi-query associative recall task (Arora et al., 2024b). Finally, in Appendix A, we provide an extended related work and discuss potential research directions opened up by our framework.

Model code and pre-trained checkpoints are open-sourced at https://github.com/state-spaces/mamba.

2 Background and Overview

2.1 Structured State Space Models

Structured state space sequence models (S4) are a recent class of sequence models for deep learning that are broadly related to RNNs, CNNs, and classical state space models. They are inspired by a particular continuous system (1) that maps a 1-dimensional sequence $x \in \mathbb{R}^{\mathtt{T}} \mapsto y \in \mathbb{R}^{\mathtt{T}}$ through an implicit latent state $h \in \mathbb{R}^{(\mathtt{T},\mathtt{N})}$. A general discrete form of structured SSMs takes the form of equation (1).

$$h_t = Ah_{t-1} + Bx_t \quad \text{(1a)} \qquad h_t = A_t h_{t-1} + B_t x_t \quad \text{(2a)}$$
$$y_t = C^\top h_t \quad \text{(1b)} \qquad y_t = C_t^\top h_t \quad \text{(2b)}$$

where $A \in \mathbb{R}^{(\mathtt{N},\mathtt{N})}, B \in \mathbb{R}^{(\mathtt{N},1)}, C \in \mathbb{R}^{(\mathtt{N},1)}$. Structured SSMs are so named because the A matrix controlling the temporal dynamics must be *structured* in order to compute this sequence-to-sequence transformation efficiently enough to be used in deep neural networks. The original structures introduced were diagonal plus low-rank (DPLR) (Gu et al., 2022a) and diagonal (Gupta et al., 2022; Gu et al., 2022b; Smith et al., 2023), which remains the most popular structure.

In this work, we use the term state space model (SSM) to refer to structured SSMs. There are many flavors of such SSMs, with deep ties to several major paradigms of neural sequence models such as continuous-time, recurrent, and convolutional models (Gu et al., 2021).

Selective State Space Models. The form (2) where the parameters (A,B,C) can also vary in time was introduced in Mamba as the **selective SSM**. Compared to the standard LTI formulation (1), this model can selectively choose to focus on or ignore inputs at every timestep. It was shown to perform much better than LTI SSMs on information-dense data such as language, especially as its state size $\mathtt{N}$ increases allowing for more information capacity. However, it can only be computed in recurrent instead of convolutional mode, and requires a careful hardware-aware implementation to be efficient. Even so, it is still less efficient than hardware-friendly models such as CNNs and Transformers because it does not leverage matrix multiplication units, which modern accelerators such as GPUs and TPUs are specialized for.

While *time-invariant* SSMs are closely related to continuous, recurrent, and convolutional sequence models, they are not directly related to attention. In this paper, we show a deeper relationship between *selective* SSMs and attention, and use it to significantly improve the training speed of SSMs while simultaneously allowing for much larger state sizes $\mathtt{N}$.

Structured SSMs as Sequence Transformations.

Definition 2.1. *We use the term **sequence transformation** to refer to a parameterized map on sequences $Y = f_\theta(X)$ where $X,Y \in \mathbb{R}^{(\mathtt{T},\mathtt{P})}$ and θ is an arbitrary collection of parameters.* $\mathtt{T}$ *represents the sequence or* time *axis; subscripts index into the first dimension, e.g. $X_t, Y_t \in \mathbb{R}^{\mathtt{P}}$.*

Sequence transformations (e.g. SSMs, or self-attention) are the cornerstone of deep sequence models, where they are incorporated into neural network architectures (e.g. Transformers). The SSM in (1) or (2) is a sequence transformation with $\mathtt{P} = 1$; it can be generalized to $\mathtt{P} > 1$ by simply broadcasting across this dimension (in other words, viewing the input as $\mathtt{P}$ independent sequences and applying the SSM to each). One can think of $\mathtt{P}$ as a **head dimension**.

Definition 2.2. *We define the **SSM operator** $\mathsf{SSM}(A,B,C) = \mathsf{SSM}(A_{0:T}, B_{0:T}, C_{0:T})$ as the sequence transformation $X \in \mathbb{R}^{(\mathtt{T},\mathtt{P})} \mapsto Y \in \mathbb{R}^{(\mathtt{T},\mathtt{P})}$ defined by equation (2).*

In SSMs, the $\mathtt{N}$ dimension is a free parameter called the **state size** or state dimension. We also call it the **state expansion**

factor, because it expands the size of the input/output by a factor of N, with implications for the computational efficiency of these models.

Finally, we remark that many types of sequence transformations, such as attention, can be represented as a single matrix multiplication across the sequence dimension.

Definition 2.3. *We call a sequence transformation $Y = f_\theta(X)$ a **matrix transformation** if it can be written in the form $Y = M_\theta X$ where M is a matrix depending on the parameters θ. We identify the sequence transformation with the matrix M, and often drop the dependence on θ when clear from context.*

2.2 Attention

Attention broadly refers to a type of computation that assigns scores to every pair of positions in a sequence, allowing each element to "attend" to the rest. By far the most common and important variant of attention is softmax self-attention, which can be defined as

$$Y = \text{softmax}(QK^\top) \cdot V$$

for $Q, K, V \in \mathbb{R}^{(\mathtt{T}, \mathtt{P})}$. The mechanism of pairwise comparisons (induced by materializing $QK^\top$) leads to the characteristic quadratic training cost of attention.

Many variants of attention have been proposed, but all share the underlying core of these attention scores, with various approximations (Tay et al., 2022). The most important variant for this work is **linear attention** (Katharopoulos et al., 2020). Roughly speaking, this family of methods drops the softmax by folding it into a kernel feature map, and uses associativity of matrix multiplication to rewrite $(QK^\top) \cdot V = Q \cdot (K^\top V)$. Moreover, in the important case of causal (autoregressive) attention, they show that when the causal mask is incorporated into the left-hand side as $(L \circ QK^\top) \cdot V$, where L is the lower-triangular 1's matrix, then the right-hand side can be expanded as a recurrence. Several recent and concurrent works such as RetNet (Sun et al., 2023) and GateLoop (Katsch, 2023) strengthen this to more general forms of L (Appendix A). In this work, our formulation of structured masked attention will strongly generalize these ideas.

2.3 Structured Matrices

General matrices $M \in \mathbb{R}^{(\mathtt{T}, \mathtt{T})}$ require $\mathtt{T}^2$ parameters to represent and $O(\mathtt{T}^2)$ time to perform basic operations such as matrix-vector multiplication. **Structured matrices** are those that

(i) can be represented in subquadratic (ideally linear) parameters through a compressed representation, and

(ii) have fast algorithms (most importantly matrix multiplication) by operating directly on this compressed representation.

Perhaps the most canonical families of structured matrices are sparse and low-rank matrices. However, there exist many other families, such as Toeplitz, Cauchy, Vandermonde, and butterfly matrices, which have all been used in machine learning for efficient models (Thomas et al., 2018; Dao et al., 2019; Gu et al., 2022b; Fu et al., 2024). Structured matrices are a powerful abstraction for efficient representations and algorithms. In this work, we will show that SSMs are equivalent to another class of structured matrices that have not previously been used in deep learning, and use this connection to derive efficient methods and algorithms.

2.4 Overview: Structured State Space Duality

While this paper develops a much richer framework of connections between SSMs, attention, and structured matrices, we provide a brief summary of the main method, which is actually quite self-contained and simple algorithmically.

Recurrent (Linear) Form. The state space dual (SSD) layer can be defined as a special case of the selective SSM (2). The standard computation of an SSM as a recurrence (or parallel scan) can be applied, which has linear complexity in sequence length. Compared to the version used in Mamba, SSD has two minor differences:

- The structure on A is further simplified from diagonal to *scalar times identity* structure. Each A_t can also be identified with just a scalar in this case.
- We use a larger head dimension $\mathtt{P}$, compared to $\mathtt{P} = 1$ used in Mamba. Typically $\mathtt{P} = \{64, 128\}$ is chosen which is similar to conventions for modern Transformers.

Compared to the original selective SSM, these changes can be viewed as slightly decreasing the expressive power in return for significant training efficiency improvements. In particular, our new algorithms will allow the use of matrix multiplication units on modern accelerators.

Dual (Quadratic) Form. The dual form of SSD is a quadratic computation closely related to attention, defined as

$$(L \circ QK^\top) \cdot V \qquad L_{ij} = \begin{cases} a_i \times \cdots \times a_{j+1} & i \geq j \\ 0 & i < j \end{cases}$$

where a_i are input-dependent scalars bounded in $[0, 1]$.

Compared to standard softmax attention, there are two main differences

- The softmax is dropped.
- The attention matrix is multiplied elementwise-wise by an additional mask matrix L.

Both of these changes can be viewed as addressing problems in vanilla attention. For example, the softmax has been recently observed to cause problems in attention scores, such as the "attention sink" phenomenon (Xiao et al., 2024; Darcet et al., 2024). More importantly, the mask matrix L can be viewed as replacing the heuristic positional embeddings of Transformers with a different *data-dependent positional mask* that controls how much information is transfered across time.

More broadly, this form is an instance of our **structured masked attention** generalization of linear attention, defined in Appendix B.

Matrix Form and SSD Algorithm. The various forms of SSD are connected through a unified matrix representation, by

showing that SSMs have a matrix transformation form $Y = MX$ for a matrix $M_\theta \in \mathbb{R}^{(\mathtt{T},\mathtt{T})}$ that depends on $\theta = (A, B, C)$. In particular, the dual form of SSD is equivalent to naive (quadratic-time) multiplication by the matrix M, and the recurrent form is a particular efficient (linear-time) algorithm that leverages the structure in M.

Going beyond these, *any* algorithm for multiplication by M can be applied. Our proposed hardware-efficient SSD algorithm (Appendix C) is a new structured matrix multiplication method that involves block decompositions of M, which obtains better efficiency tradeoffs than either the pure linear or quadratic forms. It is relatively simple and easy-to-implement compared to general selective SSMs (Gu & Dao, 2023); Listing 1 provides a complete implementation in a few lines of code.

2.5 Notation

Throughout this paper, we prefer using precise notation that can be mapped to code.

Matrices and Vectors. We generally use lower case to denote vectors (i.e. tensors with a single axis) and upper case to denote matrices (i.e. tensors with more than one axes). We do not bold matrices in this work. Sometimes, if a matrix is tied or repeated along one axis (and hence can also be viewed as a vector), we may use either upper or lower case for it.[2] $\cdot$ denotes scalar or matrix multiplication while $\circ$ denotes Hadamard (elementwise) multiplication.

Indexing. We use Python-style indexing, e.g. $i:j$ refers to the range $(i,i+1,...,j-1)$ when $i<j$ and $(i,i-1,...,j+1)$ when $i>j$. For example, for any symbol v we let $v_{j:i}$ for $j \geq i$ denote the sequence $(v_j,...,v_{i+1})$. $[i]$ is equivalent to $0:i=(0,...,i-1)$. For shorthand, we also let $v^{\times}_{j:i}$ denote the product $v_j \times \cdots \times v_{i+1}$.[3]

Dimensions. To distinguish from matrices and tensors, we often use capital letters in typewriter fonts (e.g. $\mathtt{D},\mathtt{N},\mathtt{T}$) to denote dimensions and tensor shapes. Instead of the traditional notation $M \in \mathbb{R}^{T\times T}$ we frequently use $M \in \mathbb{R}^{(\mathtt{T},\mathtt{T})}$ to reflect tensor shapes in code.

Tensor Contractions. We will heavily rely on **tensor contraction** or **einsum** notation both for clarity and as a central tool in stating and proving our results. We assume the reader to be familiar with this notation, which is commonly used in modern tensor libraries such as `numpy`. For example, we can use $\mathsf{contract}(\mathtt{MN},\mathtt{NK}\to\mathtt{MK})$ to denote the matrix-matrix multiplication operator, and in our notation $\mathsf{contract}(\mathtt{MN},\mathtt{NK}\to\mathtt{MK})(X,Y)$ (which is equivalent to $X\cdot Y$) can be translated to code as `numpy.einsum('mn,nk->mk',X,Y)`.

[2] In this work, this happens only with the A parameter of SSMs.

[3] In some contexts, it is always clear that the notation $a_{i:j}$ or $A_{i:j}$ means $a^{\times}_{i:j}$, and the superscript is omitted.

3 Structured State Space Duality

3.1 The Matrix Transformation Form of SSMs

Recall that our definition of an SSM is defined as a parameterized map defined through (2). Our theoretical framework starts by simply writing this transformation as a matrix multiplication mapping the vectors $x \mapsto y \in \mathbb{R}^T$.

By definition, $h_0 = B_0 x_0$. By induction,

$$\begin{aligned} h_t &= A_t...A_1 B_0 x_0 + A_t...A_2 B_1 x_1 + ... \\ &\quad + A_t A_{t-1} B_{t-2} x_{t-2} + A_t B_{t-1} x_{t-1} + B_t x_t \\ &= \sum_{s=0}^{t} A^{\times}_{t:s} B_s x_s. \end{aligned}$$

Multiplying by C_t to produce y_t and vectorizing the equation over $t \in [T]$, we derive the matrix multiplication form of SSMs.

$$\begin{aligned} y_t &= \sum_{s=0}^{t} C_t^\top A^{\times}_{t:s} B_s x_s \\ y &= \mathsf{SSM}(A,B,C)(x) = Mx \\ M_{ji} &\coloneqq C_j^\top A_j \cdots A_{i+1} B_i. \end{aligned} \tag{3}$$

3.2 Semiseparable Matrices

M in equation (3) is a particular representation of a class of matrices known as semiseparable matrices. Semiseparable matrices are a fundamental matrix structure. We first define these matrices and their properties.

Definition 3.1. *A (lower triangular) matrix M is N-semiseparable if every submatrix contained in the lower triangular portion (i.e. on or below the diagonal) has rank at most N. We call N the* order *or* rank *of the semiseparable matrix.*

Definition 3.1, and other forms of related "separable" structure (e.g. quasiseparable matrices and other definitions of semiseparable matrices) are sometimes called **structured rank matrices** (or rank-structured matrices) because they are characterized by rank conditions on their submatrices. Semiseparable matrices have many structured representations including the hierarchical semiseparable (HSS), sequential semiseparable (SSS), and Bruhat forms (Pernet & Storjohann, 2018). We will primarily use the SSS form.

The Sequentially Semiseparable Representation.

Definition 3.2. *A lower triangular matrix $M \in \mathbb{R}^{T\times T}$ has a N-**sequentially semiseparable (SSS)** representation if it can be written in the form*

$$M_{ji} = C_j^\top A_j \cdots A_{i+1} B_i \tag{4}$$

for vectors $B_0,...,B_{T-1}, C_0,...,C_{T-1} \in \mathbb{R}^N$ and matrices $A_0,...,A_{T-1} \in \mathbb{R}^{N\times N}$.

We define the operator SSS so that $M = \mathsf{SSS}(A_{0:T}, B_{0:T}, C_{0:T})$.

A fundamental result of semiseparable matrices is that they are exactly equivalent to matrices with SSS representations, a well-established result in the literature.

Proposition 3.3. *The class of N-semiseparable matrices (Definition 3.1) and the class of matrices with an N-SSS representation coincide.*

Therefore in the rest of this paper we will conflate the structured matrix class (Definition 3.1) and a particular representation of it (Definition 3.2); we will always use this representation instead of other candidates. In turn we will use N-SS to refer to an N-semiseparable matrix in SSS form.

Semiseparable matrices are a fundamental matrix structure and have many important properties. They are deeply related to recurrences at large, and can be defined by multiple characterizations (e.g. Definitions 3.1 and 3.2) which reveal different connections and efficient algorithms for them.

1-SS Matrices: the Scalar SSM Recurrence.

We will single out the special case of 1-semiseparable matrices. Note that in this case, the C_j and B_i are scalars, and can be factored out of the SSS representation (4)

$$\mathsf{SSS}(a,b,c) = \mathsf{diag}(c) M \mathsf{diag}(b) \qquad \text{where} \qquad M_{ji} = a_{j:i}.$$

We also use lower-case to emphasize that the parameters are scalars in this case.

Since diagonal matrices are easy to handle (e.g. multiplication by a diagonal matrix is the same as elementwise scalar multiplication), we can ignore these terms. Thus our basic representation of a 1-SS matrix is $M_{ji} = a_{j:i}$ or

$$M = 1\mathsf{SS}(a_{0:T}) := \begin{bmatrix} 1 & & & & \\ a_1 & 1 & & & \\ a_2 a_1 & a_2 & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ a_{T-1}\dots a_1 & a_{T-1}\dots a_2 & \dots & a_{T-1} & 1 \end{bmatrix}.$$

The importance of 1-SS matrices lies in their equivalence to the minimal form of a scalar recurrence – the case of a degenerate SSM with state dimension $N = 1$ and no (B,C) projections. Note that multiplication $y = Mx$ can be computed by the recurrence

$$\begin{aligned} y_t &= a_{t:0} x_0 + \dots + a_{t:t} x_t \\ &= a_t (a_{t-1:0} x_0 + \dots + a_{t-1:t-1} x_{t-1}) + a_{t:t} x_t \\ &= a_t y_{t-1} + x_t. \end{aligned} \tag{5}$$

We thus also refer to matrix multiplication by 1-SS matrices as the **scalar SSM recurrence** or the `cumprodsum` (cumulative product sum; a generalization of cumulative productive and cumulative sum) operator. As the fundamental form of recurrence, multiplication by 1-SS matrices is important as a building block for our main algorithms.

3.3 State Space Models are Semiseparable Matrices

Recall that our definition of an SSM is defined as a parameterized map defined through Definition 2.1. The connection between SSMs and semiseparable matrices follows from simply writing this transformation as a matrix multiplication mapping the vectors $x \mapsto y \in \mathbb{R}^T$.

Equation (3) directly establishes the link between state space models and the sequentially semiseparable representation, which in turn are equivalent to semiseparable matrices in general (Proposition 3.3).

Theorem 3.4. *The state space model transformation $y = \mathsf{SSM}(A,B,C)(x)$ with state size $\mathtt{N}$ is identical to matrix multiplication by an $\mathtt{N}$-SS matrix in sequentially semiseparable representation $y = \mathsf{SSS}(A,B,C) \cdot x$.*

In other words the sequence transformation operator SSM (Definition 2.2) coincides with the matrix construction operator SSS (Definition 3.2), and we use them interchangeably (or sometimes SS as shorthand). Furthermore—by a twist of fate—structured state-space models and sequentially semiseparable matrices have the same acronyms, underscoring their equivalence! Conveniently we can use any of these acronyms SSM (state-space model or semiseparable matrix), SSS (structured state-space or sequentially semiseparable), or SS (state-space or semiseparable) interchangeably to unambiguously refer to either concept. However, we will generally use the convention that SSM refers to state space model, SS refers to semiseparable, and SSS refers to sequentially semiseparable.

3.4 Computing State Space Models through Structured Matrix Algorithms

The reason Theorem 3.4 is important is that it will allow us to *reduce the problem of efficient computation of SSMs (and other sequence models) into efficient algorithms for structured matrix multiplication.*

As defined in Section 2.3, semiseparable matrices are a classical type of structured matrix:

(i) They have compressed representations such as the SSS form which has only $O(T)$ instead of $O(T^2)$ parameters.

(ii) They have fast algorithms operating directly on the compressed representation.

Furthermore, the parameterization and matrix multiplication cost can be tight in the semiseparable order.

Proposition 3.5 (Pernet et al. (2023))**.** *An N-SS matrix of size T can be represented in $O(NT)$ parameters and has matrix-vector multiplication in time and space $O(NT)$.*

Surprisingly, Theorem 3.4 and Proposition 3.5 immediately imply that all SSMs have optimal asymptotic efficiency, even when A is unstructured.

Theorem 3.6. *Any state space model (Definition 2.2) of state size N on sequence length T can be computed in time $O(TN)$ (not accounting for potential preprocessing).*

We note that this result is new to the structured SSM literature and not obvious. In particular, given dense unstructured A_t matrices, the total representation alone is of size $O(\mathtt{TN}^2)$. Thus Theorem 3.6 states the non-trivial result that with a pre-processing step, even an unstructured SSM can be computed optimally efficiently, with upper bound matching the lower bound $O(\mathtt{TN})$ (the size of B and C).

3.5 SSD: The Linear (Recurrent) Mode

Proposition 3.5 can be easily seen in the case of diagonal structured SSMs (DSS,S4D,S5 (Gupta et al., 2022; Gu et al., 2022b; Smith et al., 2023)), simply by leveraging the state space model formulation (2) and unrolling the recurrence. We will focus on the even more structured case where A_t is a scalar-times-identity matrix; in this special case of a diagonal SSM, A_t can simply be identified with a scalar. We provide the formal tensor-contraction algorithm in (6), where the dimension $\mathtt{S}$ is equal to $\mathtt{T}$ (a different symbol is required for the contraction notation).

$$Z = \mathsf{contract}(\mathtt{SP},\mathtt{SN}\to\mathtt{SPN})(X,B) \qquad (\mathtt{S},\mathtt{P},\mathtt{N}) \qquad (6a)$$
$$H = \mathsf{contract}(\mathtt{TS},\mathtt{SPN}\to\mathtt{TPN})(L,Z) \qquad (\mathtt{T},\mathtt{P},\mathtt{N}) \qquad (6b)$$
$$Y = \mathsf{contract}(\mathtt{TN},\mathtt{TPN}\to\mathtt{TP})(C,H) \qquad (\mathtt{T},\mathtt{P}) \qquad (6c)$$

Here, $L \in \mathbb{R}^{(\mathtt{T},\mathtt{T})}$ is defined as $\mathsf{1SS}(A)$, or in other words $L_{0:\mathtt{T},0:\mathtt{T}} = \mathsf{1SS}(A_{0:\mathtt{T}})$ for $i \in [\mathtt{N}]$. This algorithm involves three steps corresponding to (2):

(i) expanding the input X by the input matrix B (6a),

(ii) unrolling independent scalar SSM recurrences (6b), and

(iii) contracting the hidden state H by the output matrix C (6c).

Note that we have used the equivalence between scalar SSMs and 1-SS matrix multiplication in step (6b).

Remark 1. *We note that (6) is a special case of the Mamba (S6) model. However, a naive implementation is slow because of the expanded tensors Z and H of size $(\mathtt{T},\mathtt{P},\mathtt{N})$; Gu & Dao (2023) introduced a hardware-aware implementation to avoid materializing these tensors.*

3.6 SSD: The Quadratic (Attention) Mode

We note that there is another way to compute an SSM exposed by our new matrix point of view. A naive computation of the matrix SSM representation (3) involves simply materializing the matrix $M = \mathsf{SSM}(A,B,C)$. This is a $(\mathtt{T},\mathtt{T})$ matrix, and therefore this naive algorithm will scale quadratically in sequence length. However, when the sequence length $\mathtt{T}$ is short, this can actually be more efficient than the linear algorithm due to constant factors and the hardware-friendliness of the computation pattern (e.g. leveraging matrix-matrix multiplications). Furthermore, for the case of scalar-identity structured SSMs above, this turns out to look very similar to a quadratic attention computation.

Note that in this case we can rearrange the matrix $M = \mathsf{SSM}(A,B,C)$ (equation (3)) as $M_{ji} = A_{j:i} \cdot (C_j^\top B_i)$, and this can be vectorized into $M = L \circ (CB^\top)$, where $B,C \in \mathbb{R}^{T\times N}$ and $L := \mathsf{1SS}(A)$.

Using this formulation, the full output $Y = MX$ can be computed by materializing M:

$$G = \mathsf{contract}(\mathtt{TN},\mathtt{SN}\to\mathtt{TS})(C,B) \qquad (\mathtt{T},\mathtt{S}) \qquad (7a)$$
$$M = \mathsf{contract}(\mathtt{TS},\mathtt{TS}\to\mathtt{TS})(G,L) \qquad (\mathtt{T},\mathtt{S}) \qquad (7b)$$
$$Y = \mathsf{contract}(\mathtt{TS},\mathtt{SP}\to\mathtt{TP})(M,X) \qquad (\mathtt{T},\mathtt{P}) \qquad (7c)$$

In equations, we can write this in the form $Y = (L \circ CB^\top)\cdot X$. But by renaming $(C,B,X) \to (Q,K,V)$, note that this is just Linear Attention (Section 2.2) with more general L!

3.7 Structured Masked Attention (SMA)

We have shown that a particular SSM computation (6) is equivalent to an attention-like computation (7). We can similarly derive the reduction in the other direction, starting from linear attention. We observe that this is a result of a deeper fact:

- They both compute the same function which is simply a particular *tensor contraction on four terms*.
- The quadratic and linear algorithms are simply *two different pairwise reduction orders*.

Linear attention starts from (7), and observes that for the special case where L is the causal mask, it can be rewritten into (6), which can be computed in linear time.

However, we observe that all that is necessary for (6) to be fast is for L to be any *structured matrix*, which by definition are those that have fast matrix multiplication (Section 2.3). This motivates our definition of SMA, a strong generalization of linear attention.

Definition 3.7. ***Structured masked attention (SMA)*** *is defined as a* function *on queries/keys/values Q,K,V as well as any* structured matrix *L, through the 4-way tensor contraction*

$$y = \mathsf{contract}(\mathtt{TN},\mathtt{SN},\mathtt{SP},\mathtt{TS}\to\mathtt{TP})(Q,K,V,L).$$

*The SMA **quadratic mode algorithm** is the sequence of pairwise contractions defined by (7), which corresponds to the standard (masked) attention computation.*

*The SMA **linear mode algorithm** is the sequence of pairwise contractions defined by (6), where step (6b) is optimized through the subquadratic structured matrix multiplication.*

The SSD linear/quadratic duality is a special case of SMA; this duality holds for *any* structured matrix L. SSD corresponds to the case when L is a 1-SS matrix, whence the linear form takes the form of an SSM recurrence.

The full SMA framework is developed in Appendix B.

3.8 Structured State-Space Duality

Structured state-space duality is summarized in Figure 1. Note that we use SSD to refer to both a model, and a theoretical framework. The *SSD model* is the intersection of state space models and structured masked attention, which can be described as either the class of scalar-identity SSMs or the class of 1-SS masked attention. The *SSD framework* refers to the rich interplay between our main objects of interest—state space models, structured masked attention, and structured matrices—and particularly the duality between recurrences and attention induced by two different tensor contraction orders.

3.9 State Space Duality: The Hybrid Mode

The power of developing the theoretical SSD framework between SSMs, attention, and structured matrices lies in using

Structured State Space Model		Structured Masked Attention	
C	(output matrix)	Q	(queries)
B	(input matrix)	K	(keys)
X	(input sequence)	V	(values)
$A_{j:i}$	(state matrix)	L_{ji}	(mask)
$\mathtt{N}$	(state expansion dim.)	$\mathtt{N}$	(kernel feature dim.)
H	(hidden states (6b))	SMA linear dual (6)	
$=L\cdot XB$	(linear mode)		
SSM quadratic dual (7)		G	(Gram matrix (7a))
		$=Q\cdot K^\top$	(quadratic mode)

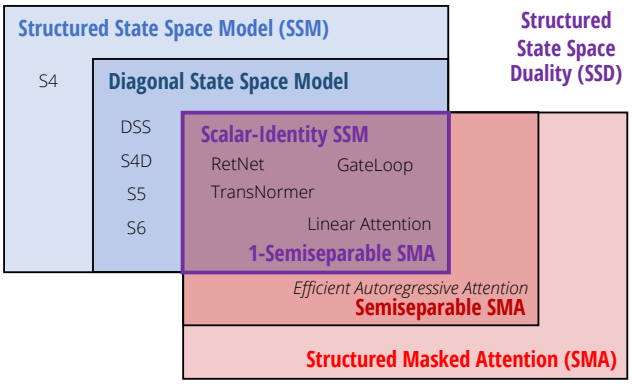

Figure 1: (**Structured State-Space Duality**.) This paper explores the rich relationships between state space models and attention through the bridge of structured matrices.

the connections to improve the models and algorithms. Our main computational result is an algorithm for computing SSD models that combines both the linear (primary, recurrent) mode and quadratic (dual, attention) mode. This algorithm is as computation efficient as SSMs (linear scaling in sequence length) and as hardware-friendly as attention (primarily uses matrix multiplications).

Theorem 3.8. *Consider an SSD model with state expansion factor* $\mathtt{N}$ *and head dimension* $\mathtt{P}=\mathtt{N}$. *There exists an algorithm for computing the model on any input* $X\in\mathbb{R}^{(\mathtt{T},\mathtt{P})}$ *which only requires* $O(\mathtt{TN}^2)$ *training FLOPs,* $O(\mathtt{TN})$ *inference flops,* $O(\mathtt{N}^2)$ *memory, and whose work is dominated by matrix multiplications.*

Note that all of these bounds are tight, because a state space model with state expansion $\mathtt{N}$ operating on a head size of $\mathtt{N}$ has total state size $\mathtt{N}^2$ (yielding the lower bounds for training and inference FLOPs of $O(\mathtt{TN}^2)$ and $O(\mathtt{N}^2)$ respectively). Furthermore the input X itself has $\mathtt{TN}$ elements, yielding the memory lower bound.

The main idea behind Theorem 3.8 is once again viewing the problem of computing a state space model as a semiseparable matrix multiplication, but leveraging its structure in a new way. Instead of computing the whole matrix in either recurrent or attention mode, we perform a *block decomposition* of the matrix. The diagonal blocks can be computed using the dual attention mode, which can be efficiently done with matrix multiplications, while the off-diagonal blocks can be factored by the rank-structure of semiseparable matrices and reduced to a smaller recurrence. Appendix C contains details of the algorithm which proves Theorem 3.8. We highlight that Listing 1 provides a self-contained implementation of the SSD algorithm. Compared to the general selective SSM of Gu & Dao (2023), this implementation is much simpler, and relatively efficient even in native PyTorch without requiring special low-level kernels.

3.10 The Mamba-2 Architecture

The Mamba-2 architecture makes small changes to the Mamba architecture, inspired by the connection between SSMs and attention (Figure 2). In Mamba, the selective SSM layer is viewed as a map from $X\mapsto Y$; the parameters A,B,C are viewed as subsidiary and are functions of the SSM input X. Thus these linear projections occur after the initial linear projection to create X.

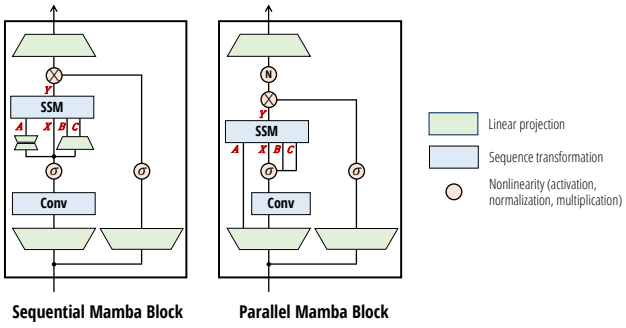

Figure 2: (**Architecture**.) The Mamba-2 block simplifies the Mamba block by removing sequential linear projections; the SSM parameters A,B,C are produced at the beginning of the block instead of as a function of the SSM input X. An additional normalization layer is added as in NormFormer (Shleifer et al., 2021), improving stability. The B and C projections only have a single head shared across the X heads, analogous to multi-value attention (MVA).

Since SSD is viewed as a map from $A,X,B,C\mapsto Y$, it therefore makes sense to produce A,X,B,C in parallel with a single projection at the beginning of the block. Note the analogy to standard attention architectures, where X,B,C correspond to the Q,K,V projections that are created in parallel.

4 Empirical Validation

We empirically evaluate Mamba-2 on synthetic recall tasks that have been challenging for recurrent models (Section 4.1), and standard language modeling pre-training and downstream evaluations (Section 4.2). We validate that our SSD algorithm is much more efficient than Mamba-1 (Section 4.3) and comparable to optimized attention for moderate sequence lengths.

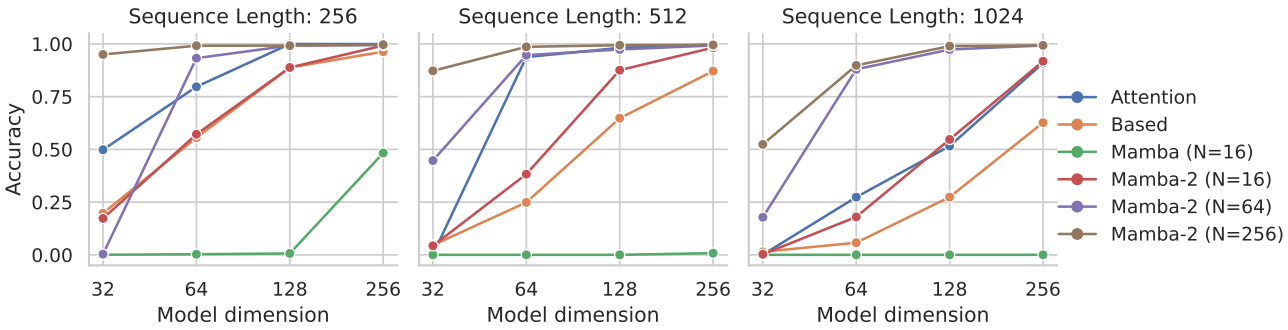

Figure 3: (**Multi-Query Associative Recall (MQAR)**). Associative recall tasks are challenging for SSMs, which must memorize all relevant information into their recurrent state. The SSD layer combined with improved architecture allows for much larger state sizes in Mamba-2, which performs significantly better than Mamba-1 and even vanilla attention.

Table 1: (**Zero-shot Evaluations**.) Best results for each size in bold, second best unlined. We compare against open source LMs with various tokenizers, trained for up to 300B tokens. Pile refers to the validation split, comparing only against models trained on the same dataset and tokenizer (GPT-NeoX-20B). For each model size, Mamba-2 outperforms Mamba, and generally matches Pythia at twice the model size. Full results in Table 3.

Model	Token.	Pile ppl ↓	LAMBADA ppl ↓	LAMBADA acc ↑	HellaSwag acc ↑	PIQA acc ↑	Arc-E acc ↑	Arc-C acc ↑	WinoGrande acc ↑	OpenBookQA acc ↑	Average acc ↑
Pythia-1B	NeoX	7.82	7.92	56.1	47.2	70.7	57.0	27.1	53.5	31.4	49.0
Mamba-790M	NeoX	7.33	6.02	**62.7**	**55.1**	**72.1**	**61.2**	**29.5**	56.1	34.2	53.0
Mamba-2-780M	NeoX	**7.26**	**5.86**	61.7	54.9	72.0	61.0	28.5	**60.2**	**36.2**	**53.5**
Hybrid H3-1.3B	GPT2	—	11.25	49.6	52.6	71.3	59.2	28.1	56.9	34.4	50.3
Pythia-1.4B	NeoX	7.51	6.08	61.7	52.1	71.0	60.5	28.5	57.2	30.8	51.7
RWKV4-1.5B	NeoX	7.70	7.04	56.4	52.5	72.4	60.5	29.4	54.6	34.0	51.4
Mamba-1.4B	NeoX	6.80	5.04	65.0	59.1	**74.2**	**65.5**	32.8	**61.5**	36.4	**56.4**
Mamba-2-1.3B	NeoX	**6.66**	**5.02**	**65.7**	**59.9**	73.2	64.3	**33.3**	60.9	**37.8**	**56.4**
Hybrid H3-2.7B	GPT2	—	7.92	55.7	59.7	73.3	65.6	32.3	61.4	33.6	54.5
Pythia-2.8B	NeoX	6.73	5.04	64.7	59.3	74.0	64.1	32.9	59.7	35.2	55.7
RWKV4-3B	NeoX	7.00	5.24	63.9	59.6	73.7	67.8	33.1	59.6	37.0	56.4
Mamba-2.8B	NeoX	6.22	4.23	69.2	66.1	75.2	**69.7**	36.3	63.5	**39.6**	59.9
Mamba-2-2.7B	NeoX	**6.09**	**4.10**	**69.7**	**66.6**	**76.4**	69.6	**36.4**	**64.0**	38.8	**60.2**

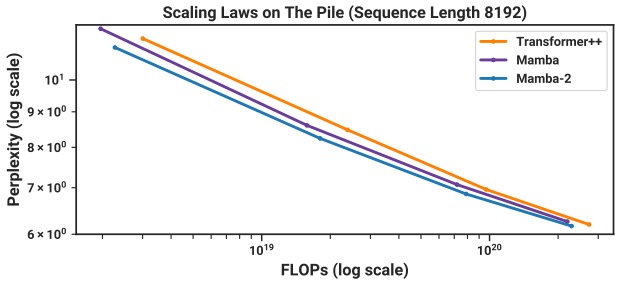

Figure 4: (**Scaling Laws**.) Models of size $\approx 125M$ to $\approx 1.3B$ parameters, trained on the Pile. Mamba-2 matches or exceeds the performance of Mamba as well as a strong "Transformer++" recipe. Compared to our Transformer baseline, Mamba-2 is Pareto dominant on performance (perplexity), theoretical FLOPs, and actual wall-clock time.

4.1 Synthetics: Associative Recall

Synthetic associative recall tasks have been popular for testing the ability of language models to look up information in their context. Broadly, they involve feeding autoregressive models pairs of key-value associations, and then prompting the model to produce the correct completion upon being shown a previously-seen key. The **multi-query associative recall (MQAR)** task is a particular formulation of this task that requires the model to memorize multiple associations (Arora et al., 2024a).

We compare on a challenging version of the MQAR setup from (Arora et al., 2024b), using a harder task, longer sequences, and smaller models. Our baselines include standard multi-head softmax attention as well as the Based architecture which combines convolutions, local attention, and a linear attention variant.

Results are shown in Figure 3. While Mamba-1 struggles on this task, Mamba-2 performs well across all settings.

4.2 Language Modeling

Following standard protocols in LLMs, we train and evaluate the Mamba-2 architecture on standard autoregressive language modeling against other architectures. We compare both pretraining metrics (perplexity) and zero-shot evaluations. The model sizes (depth and width) follow GPT3 specifications, from 125m to 2.7B. We use the Pile dataset (Gao et al., 2020), and follow the training recipe described in Brown et al. (2020). This follows the same setup as reported in Mamba (Gu & Dao, 2023);

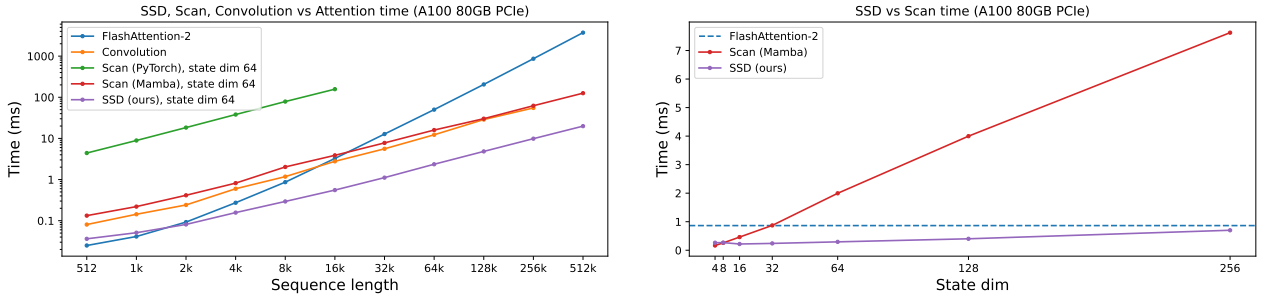

Figure 5: (**Efficiency Benchmarks**.) (*Left*) Our SSD is $2-8\times$ faster than a Mamba fused scan for large state expansion ($N=64$) and faster than FlashAttention-2 for sequence length 2k and above. (*Right*) Sequence length 4K: Increasing state expansion slows down the Mamba optimized scan implementation linearly. SSD can handle much larger state expansion factors without much slowdown.

training details are in Appendix E.

4.2.1 Scaling Laws

For baselines, we compare against both Mamba and its Transformer++ recipe (Gu & Dao, 2023), which is based on the PaLM and LLaMa architectures (e.g. rotary embedding, SwiGLU MLP, etc.). As Mamba has already demonstrated that it outperforms the standard Transformer architecture (GPT3 architecture) as well as recent subquadratic architectures (H3 (Dao et al., 2023), Hyena (Poli et al., 2023), RWKV-4 (Peng et al., 2023), RetNet (Sun et al., 2023)), we omit those in the plot for clarity (see Gu & Dao (2023) for comparisons).

4.2.2 Downstream Evaluations

Table 1 shows the performance of Mamba-2 on a range of popular downstream zero-shot evaluation tasks, compared to the most well-known open source models at these sizes, most importantly Pythia (Biderman et al., 2023) which were trained with the same tokenizer, dataset, and training length (300B tokens) as our models.

4.3 Speed Benchmarks

We benchmark the speed of the SSD algorithm against Mamba's scan implementation and FlashAttention-2 (Figure 5). SSD, thanks to its reformulation to use matrix multiplication as a subroutine, can exploit specialized matrix multiplication (matmul) units on GPUs, also known as tensor cores. As a result, it is 2-8$\times$ faster than Mamba's fused associative scan, which does not leverage matmul units. Due to its linear scaling in sequence length, SSD is faster than FlashAttention-2 starting at sequence length $2K$.

However, we note that the Mamba-2 model as a whole might not be as efficient to train as Transformer at short sequence length (e.g. at $2K$), since a Transformer with L layers would have $\frac{L}{2}$ MLP layers and $\frac{L}{2}$ attention layers, while a Mamba-2 model would have L SSD layers for the same number of parameters. Generally the MLP layers are very hardware efficient since they consist of simple matrix multiplication and pointwise linearity.

5 Conclusion

We proposed a theoretical framework based on well-studied classes of structured matrices that bridges the conceptual gap between SSMs and attention variants. This framework yields insights on how recent SSMs (e.g. Mamba) perform as well as Transformers on language modeling. Moreover, our theoretical tools provide new ideas to improve SSMs (and potentially Transformers) by connecting the algorithmic advances on both sides. As a demonstration, the framework guides our design of a new architecture (Mamba-2) at the intersection of SSMs and structured attention.

Acknowledgements

We thank Angela Wu for the suggestion on how to efficiently compute the gradient of Δ in a numerically stable manner. We thank Sukjun Hwang and Aakash Lahoti for assistance with the MQAR experiments.

Impact Statement

This paper presents work whose goal is to advance the field of Machine Learning. There are many potential societal consequences of our work, none which we feel must be specifically highlighted here.

References

Akyürek, E., Wang, B., Kim, Y., and Andreas, J. In-context language learning: Architectures and algorithms. In *The International Conference on Machine Learning (ICML)*, 2024.

Ali, A., Zimerman, I., and Wolf, L. The hidden attention of mamba models, 2024.

Arora, S., Eyuboglu, S., Timalsina, A., Johnson, I., Poli, M., Zou, J., Rudra, A., and Ré, C. Zoology: Measuring and improving recall in efficient language models. In *The International Conference on Learning Representations (ICLR)*, 2024a.

Arora, S., Eyuboglu, S., Zhang, M., Timalsina, A., Alberti, S., Zinsley, D., Zou, J., Rudra, A., and Ré, C. Simple linear attention language models balance the recall-throughput tradeoff. In *The International Conference on Machine Learning (ICML)*, 2024b.

Bahdanau, D., Cho, K., and Bengio, Y. Neural machine translation by jointly learning to align and translate. In *The International Conference on Learning Representations (ICLR)*, 2015.

Baker, G. A., Baker Jr, G. A., Graves-Morris, P., and Baker, S. S. *Pade Approximants: Encyclopedia of Mathematics and It's Applications, Vol. 59 George A. Baker, Jr., Peter Graves-Morris*, volume 59. Cambridge University Press, 1996.

Beck, M., Pöppel, K., Spanring, M., Auer, A., Prudnikova, O., Kopp, M., Klambauer, G., Brandstetter, J., and Hochreiter, S. xlstm: Extended long short-term memory. *arXiv preprint arXiv:2405.04517*, 2024.

Biderman, S., Schoelkopf, H., Anthony, Q. G., Bradley, H., O'Brien, K., Hallahan, E., Khan, M. A., Purohit, S., Prashanth, U. S., Raff, E., et al. Pythia: A suite for analyzing large language models across training and scaling. In *The International Conference on Machine Learning (ICML)*, pp. 2397–2430. PMLR, 2023.

Bisk, Y., Zellers, R., Gao, J., Choi, Y., et al. PIQA: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on Artificial Intelligence*, volume 34, 2020.

Black, S., Biderman, S., Hallahan, E., Anthony, Q., Gao, L., Golding, L., He, H., Leahy, C., McDonell, K., Phang, J., et al. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

Blelloch, G. E. Prefix sums and their applications. 1990.

Botev, A., De, S., Smith, S. L., Fernando, A., Muraru, G.-C., Haroun, R., Berrada, L., Pascanu, R., Sessa, P. G., Dadashi, R., et al. Recurrentgemma: Moving past transformers for efficient open language models. *arXiv preprint arXiv:2404.07839*, 2024.

Bradbury, J., Merity, S., Xiong, C., and Socher, R. Quasi-recurrent neural networks. *arXiv preprint arXiv:1611.01576*, 2016.

Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems (NeurIPS)*, 33:1877–1901, 2020.

Choromanski, K., Likhosherstov, V., Dohan, D., Song, X., Gane, A., Sarlos, T., Hawkins, P., Davis, J., Mohiuddin, A., Kaiser, L., et al. Rethinking attention with performers. In *The International Conference on Learning Representations (ICLR)*, 2021.

Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., et al. PaLM: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023. URL http://jmlr.org/papers/v24/22-1144.html.

Chung, J., Gulcehre, C., Cho, K., and Bengio, Y. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Clark, P., Cowhey, I., Etzioni, O., Khot, T., Sabharwal, A., Schoenick, C., and Tafjord, O. Think you have solved question answering? try ARC, the AI2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

Dao, T. FlashAttention-2: Faster attention with better parallelism and work partitioning. In *The International Conference on Learning Representations (ICLR)*, 2024.

Dao, T., Gu, A., Eichhorn, M., Rudra, A., and Ré, C. Learning fast algorithms for linear transforms using butterfly factorizations. In *The International Conference on Machine Learning (ICML)*, 2019.

Dao, T., Sohoni, N., Gu, A., Eichhorn, M., Blonder, A., Leszczynski, M., Rudra, A., and Ré, C. Kaleidoscope: An efficient, learnable representation for all structured linear maps. In *The International Conference on Learning Representations (ICLR)*, 2020.

Dao, T., Chen, B., Sohoni, N. S., Desai, A., Poli, M., Grogan, J., Liu, A., Rao, A., Rudra, A., and Ré, C. Monarch: Expressive structured matrices for efficient and accurate training. In *International Conference on Machine Learning*, pp. 4690–4721. PMLR, 2022.

Dao, T., Fu, D. Y., Saab, K. K., Thomas, A. W., Rudra, A., and Ré, C. Hungry hungry hippos: Towards language modeling with state space models. In *The International Conference on Learning Representations (ICLR)*, 2023.

Darcet, T., Oquab, M., Mairal, J., and Bojanowski, P. Vision transformers need registers. In *The International Conference on Learning Representations (ICLR)*, 2024.

De, S., Smith, S. L., Fernando, A., Botev, A., Cristian-Muraru, G., Gu, A., Haroun, R., Berrada, L., Chen, Y., Srinivasan, S., et al. Griffin: Mixing gated linear recurrences with local attention for efficient language models. *arXiv preprint arXiv:2402.19427*, 2024.

De Sa, C., Gu, A., Puttagunta, R., Ré, C., and Rudra, A. A two-pronged progress in structured dense matrix vector multiplication. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pp. 1060–1079. SIAM, 2018.

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

Fu, D. Y., Epstein, E. L., Nguyen, E., Thomas, A. W., Zhang, M., Dao, T., Rudra, A., and Ré, C. Simple hardware-efficient long convolutions for sequence modeling. *The International Conference on Machine Learning (ICML)*, 2023.

Gao, L., Biderman, S., Black, S., Golding, L., Hoppe, T., Foster, C., Phang, J., He, H., Thite, A., Nabeshima, N., Presser, S., and Leahy, C. The Pile: An 800GB dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020.

Gao, L., Tow, J., Biderman, S., Black, S., DiPofi, A., Foster, C., Golding, L., Hsu, J., McDonell, K., Muennighoff, N., Phang, J., Reynolds, L., Tang, E., Thite, A., Wang, B., Wang, K., and Zou, A. A framework for few-shot language model evaluation, September 2021. URL https://doi.org/10.5281/zenodo.5371628.

Grazzi, R., Siems, J., Schrodi, S., Brox, T., and Hutter, F. Is mamba capable of in-context learning? *arXiv preprint arXiv:2402.03170*, 2024.

Gu, A. *Modeling Sequences with Structured State Spaces*. Phd thesis, Stanford University, 2023.

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

Gu, A., Dao, T., Ermon, S., Rudra, A., and Ré, C. HIPPO: Recurrent memory with optimal polynomial projections. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Gu, A., Johnson, I., Goel, K., Saab, K., Dao, T., Rudra, A., and Ré, C. Combining recurrent, convolutional, and continuous-time models with the linear state space layer. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

Gu, A., Goel, K., and Ré, C. Efficiently modeling long sequences with structured state spaces. In *The International Conference on Learning Representations (ICLR)*, 2022a.

Gu, A., Gupta, A., Goel, K., and Ré, C. On the parameterization and initialization of diagonal state space models. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022b.

Gu, A., Johnson, I., Timalsina, A., Rudra, A., and Ré, C. How to train your HIPPO: State space models with generalized basis projections. In *The International Conference on Learning Representations (ICLR)*, 2023.

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

Hillis, W. D. and Steele Jr, G. L. Data parallel algorithms. *Communications of the ACM*, 29(12):1170–1183, 1986.

Hochreiter, S. and Schmidhuber, J. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., de Las Casas, D., Hendricks, L. A., Welbl, J., Clark, A., et al. An empirical analysis of compute-optimal large language model training. *Advances in Neural Information Processing Systems (NeurIPS)*, 35:30016–30030, 2022.

Jelassi, S., Brandfonbrener, D., Kakade, S. M., and Malach, E. Repeat after me: Transformers are better than state space models at copying. In *The International Conference on Machine Learning (ICML)*, 2024.

Katharopoulos, A., Vyas, A., Pappas, N., and Fleuret, F. Transformers are RNNs: Fast autoregressive Transformers with linear attention. In *International Conference on Machine Learning*, pp. 5156–5165. PMLR, 2020.

Katsch, T. Gateloop: Fully data-controlled linear recurrence for sequence modeling. *arXiv preprint arXiv:2311.01927*, 2023.

Lee-Thorp, J., Ainslie, J., Eckstein, I., and Ontanon, S. Fnet: Mixing tokens with fourier transforms. *arXiv preprint arXiv:2105.03824*, 2021.

Lei, T. When attention meets fast recurrence: Training language models with reduced compute. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pp. 7633–7648, 2021.

Lei, T., Zhang, Y., Wang, S. I., Dai, H., and Artzi, Y. Simple recurrent units for highly parallelizable recurrence. *arXiv preprint arXiv:1709.02755*, 2017.

Li, Y., Cai, T., Zhang, Y., Chen, D., and Dey, D. What makes convolutional models great on long sequence modeling? In *The International Conference on Learning Representations (ICLR)*, 2023.

Lieber, O., Lenz, B., Bata, H., Cohen, G., Osin, J., Dalmedigos, I., Safahi, E., Meirom, S., Belinkov, Y., Shalev-Shwartz, S., et al. Jamba: A hybrid transformer-mamba language model. *arXiv preprint arXiv:2403.19887*, 2024.

Lu, C., Schroecker, Y., Gu, A., Parisotto, E., Foerster, J., Singh, S., and Behbahani, F. Structured state space models for in-context reinforcement learning. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Ma, X., Zhou, C., Kong, X., He, J., Gui, L., Neubig, G., May, J., and Zettlemoyer, L. Mega: Moving average equipped gated attention. In *The International Conference on Learning Representations (ICLR)*, 2023.

Martin, E. and Cundy, C. Parallelizing linear recurrent neural nets over sequence length. In *The International Conference on Learning Representations (ICLR)*, 2018.

Mihaylov, T., Clark, P., Khot, T., and Sabharwal, A. Can a suit of armor conduct electricity? a new dataset for open book question answering. *arXiv preprint arXiv:1809.02789*, 2018.

Orvieto, A., Smith, S. L., Gu, A., Fernando, A., Gulcehre, C., Pascanu, R., and De, S. Resurrecting recurrent neural networks for long sequences. In *The International Conference on Machine Learning (ICML)*, 2023.

Paperno, D., Kruszewski, G., Lazaridou, A., Pham, N.-Q., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., and Fernández, R. The LAMBADA dataset: Word prediction requiring a broad discourse context. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, pp. 1525–1534, 2016.

Park, J., Park, J., Xiong, Z., Lee, N., Cho, J., Oymak, S., Lee, K., and Papailiopoulos, D. Can mamba learn how to learn? a comparative study on in-context learning tasks. In *The International Conference on Machine Learning (ICML)*, 2024.

Peng, B., Alcaide, E., Anthony, Q., Albalak, A., Arcadinho, S., Cao, H., Cheng, X., Chung, M., Grella, M., GV, K. K., et al. RWKV: Reinventing RNNs for the Transformer era. *arXiv preprint arXiv:2305.13048*, 2023.

Peng, B., Goldstein, D., Anthony, Q., Albalak, A., Alcaide, E., Biderman, S., Cheah, E., Ferdinan, T., Hou, H., Kazienko, P., et al. Eagle and finch: Rwkv with matrix-valued states and dynamic recurrence. *arXiv preprint arXiv:2404.05892*, 2024.

Peng, H., Pappas, N., Yogatama, D., Schwartz, R., Smith, N. A., and Kong, L. Random feature attention. In *The International Conference on Learning Representations (ICLR)*, 2021.

Pernet, C. and Storjohann, A. Time and space efficient generators for quasiseparable matrices. *Journal of Symbolic Computation*, 85:224–246, 2018.

Pernet, C., Signargout, H., and Villard, G. Exact computations with quasiseparable matrices. *arXiv preprint arXiv:2302.04515*, 2023.

Poli, M., Massaroli, S., Nguyen, E., Fu, D. Y., Dao, T., Baccus, S., Bengio, Y., Ermon, S., and Ré, C. Hyena hierarchy: Towards larger convolutional language models. In *The International Conference on Machine Learning (ICML)*, 2023.

Press, O., Smith, N., and Lewis, M. Train short, test long: Attention with linear biases enables input length extrapolation. In *International Conference on Learning Representations*, 2022.

Qin, Z., Han, X., Sun, W., Li, D., Kong, L., Barnes, N., and Zhong, Y. The devil in linear transformer. *arXiv preprint arXiv:2210.10340*, 2022a.

Qin, Z., Sun, W., Deng, H., Li, D., Wei, Y., Lv, B., Yan, J., Kong, L., and Zhong, Y. CosFormer: Rethinking softmax in attention. In *The International Conference on Learning Representations (ICLR)*, 2022b.

Qin, Z., Han, X., Sun, W., He, B., Li, D., Li, D., Dai, Y., Kong, L., and Zhong, Y. Toeplitz neural network for sequence modeling. In *The International Conference on Learning Representations (ICLR)*, 2023a.

Qin, Z., Li, D., Sun, W., Sun, W., Shen, X., Han, X., Wei, Y., Lv, B., Luo, X., Qiao, Y., et al. Transnormerllm: A faster and better large language model with improved transnormer. *arXiv preprint arXiv:2307.14995*, 2023b.

Qin, Z., Yang, S., and Zhong, Y. Hierarchically gated recurrent neural network for sequence modeling. *Advances in Neural Information Processing Systems*, 36, 2023c.

Qin, Z., Yang, S., Sun, W., Shen, X., Li, D., Sun, W., and Zhong, Y. Hgrn2: Gated linear rnns with state expansion. *arXiv preprint arXiv:2404.07904*, 2024.

Rahimi, A. and Recht, B. Random features for large-scale kernel machines. *Advances in Neural Information Processing Systems (NeurIPS)*, 20, 2007.

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

Schlag, I., Irie, K., and Schmidhuber, J. Linear Transformers are secretly fast weight programmers. In *The International Conference on Machine Learning (ICML)*, pp. 9355–9366. PMLR, 2021.

Shleifer, S., Weston, J., and Ott, M. NormFormer: Improved Transformer pretraining with extra normalization. *arXiv preprint arXiv:2110.09456*, 2021.

Smith, J. T., Warrington, A., and Linderman, S. W. Simplified state space layers for sequence modeling. In *The International Conference on Learning Representations (ICLR)*, 2023.

Su, J., Lu, Y., Pan, S., Murtadha, A., Wen, B., and Liu, Y. Roformer: Enhanced Transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

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

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

Thomas, A., Gu, A., Dao, T., Rudra, A., and Ré, C. Learning compressed transforms with low displacement rank. In *Advances in Neural Information Processing Systems (NeurIPS)*, pp. 9052–9060, 2018.

Tolstikhin, I. O., Houlsby, N., Kolesnikov, A., Beyer, L., Zhai, X., Unterthiner, T., Yung, J., Steiner, A., Keysers, D., Uszkoreit, J., et al. MLP-Mixer: An all-MLP architecture for vision. *Advances in Neural Information Processing Systems (NeurIPS)*, 34: 24261–24272, 2021.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., and Polosukhin, I. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

Wang, S., Li, B. Z., Khabsa, M., Fang, H., and Ma, H. Linformer: Self-attention with linear complexity. *arXiv preprint arXiv:2006.04768*, 2020.

Xiao, G., Tian, Y., Chen, B., Han, S., and Lewis, M. Efficient streaming language models with attention sinks. In *The International Conference on Learning Representations (ICLR)*, 2024.

Xiong, Y., Zeng, Z., Chakraborty, R., Tan, M., Fung, G., Li, Y., and Singh, V. Nyströmformer: A nyström-based algorithm for approximating self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 2021.

Yang, S., Wang, B., Shen, Y., Panda, R., and Kim, Y. Gated Linear Attention Transformers with hardware-efficient training. In *The International Conference on Machine Learning (ICML)*, 2024.

Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., and Choi, Y. HellaSwag: Can a machine really finish your sentence? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 2019.

Zhai, S., Talbott, W., Srivastava, N., Huang, C., Goh, H., Zhang, R., and Susskind, J. An attention free Transformer. *arXiv preprint arXiv:2105.14103*, 2021.

Zhang, M., Bhatia, K., Kumbong, H., and Ré, C. The hedgehog & the porcupine: Expressive linear attentions with softmax mimicry. In *The International Conference on Learning Representations (ICLR)*, 2024.

Zheng, L., Wang, C., and Kong, L. Linear complexity randomized self-attention mechanism. In *International Conference on Machine Learning*, pp. 27011–27041. PMLR, 2022.

A Related Work and Discussion

The state space duality framework bridges connections between SSMs, structured matrices, and attention. We discuss in more depth the relations between SSD and these concepts more broadly. Using ideas from each of the viewpoints, we also suggest some directions that the SSD framework can be extended in future work.

A.1 State Space Models

Structured state space models can be characterized along the axes

(i) whether it is time-invariant or time-varying.

(ii) the dimensionality of the system.

(iii) the structure on the recurrent transitions A.

SSD can be described as a selective SSM with SISO dimensions and scalar-identity structure.

Time Variance (Selectivity). The original structured SSMs (S4) were linear time-invariant (LTI) systems (Gu, 2023; Gu et al., 2022a) motivated by continuous-time online memorization (Gu et al., 2020; 2021; 2023). Many variants of structured SSMs have been proposed (Gupta et al., 2022; Gu et al., 2022b; Smith et al., 2023; Ma et al., 2023; Dao et al., 2023), including several that drop the recurrence and focus on the convolutional representation of LTI SSMs (Li et al., 2023; Poli et al., 2023; Fu et al., 2023; Qin et al., 2023a).

SSD is a time-varying structured SSM, also known as a **selective SSM** introduced in Mamba (Gu & Dao, 2023). Selective SSMs are closely related to gating mechanisms of RNNs, including classical RNNs such as the LSTM (Hochreiter & Schmidhuber, 1997) and GRU (Chung et al., 2014) as well as more modern variants such as the QRNN (Bradbury et al., 2016), SRU (Lei et al., 2017; Lei, 2021), RWKV (Peng et al., 2023), HGRN (Qin et al., 2023c), and Griffin (De et al., 2024; Botev et al., 2024). These RNNs differ in their parameterizations in various ways, most importantly in the lack of a state expansion.

Dimensionality and State Expansion. An important characteristic of SSD, shared by previous SSMs in its lineage (S4, H3, Mamba), is that it is a **single-input single-output (SISO)** system where input channels are processed independently. This leads to a much larger effective state size of $\texttt{ND}$ where $\texttt{N}$ is the SSM state size (also called state expansion factor) and $\texttt{D}$ is the standard model dimension. Traditional RNNs either have $\texttt{N}=1$ or are multi-input multi-output (MIMO) with dense B,C matrices, either of which leads to a smaller state. While MIMO SSMs have been shown to work well in some domains (Smith et al., 2023; Orvieto et al., 2023; Lu et al., 2023), Mamba showed that state expansion is crucial for information-dense domains such as language. One of the main advantages of SSD is allowing for even larger state expansion factors without slowing down the model. Many subsequent works have since adopted state expansion (Appendix A.4).

Structure. Compared to previous structured SSMs, the main restriction of SSD is on the expressivity of the state transitions A_t. We note that more general SSMs, such as the case of diagonal A_t, have the same theoretical efficiency as SSD, but are less hardware-friendly. This is because the dual quadratic form loses its attention-like interpretation and becomes more difficult to compute. Thus compared to Mamba, SSD differs only in a slightly more restrictive form of diagonal A_t, and trades off this expressivity for improved hardware efficiency (and ease of implementation).

We hypothesize that it may be possible to refine our structured matrix algorithms to improve to the general diagonal SSM case as well.

A.2 Structured Matrices

The first viewpoint of the state space duality adopts the viewpoint of these models as **matrix sequence transformations** or "matrix mixers": sequence transformations (Definition 2.1) that can be represented as matrix multiplication (by a $\texttt{T}\times\texttt{T}$ matrix) along the sequence dimension $\texttt{T}$.

Several such matrix mixers have been proposed before, where the primary axis of variation is the representation of the matrix. These include MLP-Mixer (Tolstikhin et al., 2021) (unstructured matrix), FNet (Lee-Thorp et al., 2021) (Fourier Transform matrix), M2 (Dao et al., 2019; 2020; 2022; Fu et al., 2024) (butterfly/monarch matrix), Toeplitz matrices (Poli et al., 2023; Qin et al., 2023a), and even more exotic structures (De Sa et al., 2018; Thomas et al., 2018).

An important characterization is that efficient (sub-quadratic) matrix sequence transformations are exactly those which have *structured matrix mixers*. A core result of the SSD framework is viewing SSMs as matrix mixers with a particular structure – semiseparable matrices (Section 3.3). The linear vs. quadratic duality then takes the form of structured matrix multiplication vs. naive matrix multiplication.

The structure matrix representation led to our efficient SSD algorithm through block decompositions of particular semiseparable matrices (Appendix C). We note that semiseparable matrices are well-studied in the scientific computing literature, and incorporating those ideas may be a promising avenue for more improvements to state space models. We also suggest that focusing on the matrix

mixer viewpoint can lead to more fruitful directions for sequence models, such as designing principled non-causal variants of Mamba, or finding ways to characterize and bridge the gap between softmax attention and sub-quadratic models through analyzing their matrix transformation structure.

A.3 (Linear) Attention

Compared to standard (causal) attention, SSD has only two main differences.

First, SSD does not use the softmax activation of standard attention (Bahdanau et al., 2015; Vaswani et al., 2017), which is what gives attention its quadratic complexity. When the softmax is dropped, the sequence can be computed with linear scaling through the linear attention framework (Katharopoulos et al., 2020).

Second, SSD multiplies the logits matrix by an input-dependent 1-semiseparable mask. Thus this mask can be viewed as replacing the softmax in standard attention.

This semiseparable mask can also be viewed as providing positional information. The elements a_t act as "gates" in the RNN sense, or a "selection" mechanism (see discussion in Mamba paper), and their cumulative products $a_{j:i}$ control how much interaction is allowed between positions i and j. Positional embeddings (e.g. sinusoidal (Vaswani et al., 2017), AliBi (Press et al., 2022), and RoPE (Su et al., 2021)) are an important component of Transformers that are often viewed as heuristics, and the 1-SS mask of SSD can be seen as a more principled form of relative positional embeddings. We note that this view was also posited concurrently by GateLoop (Katsch, 2023).

The second viewpoint of state space duality is a special case of our more general structured masked attention (SMA) framework, where the duality is revealed as different contraction orderings on a simple 4-way tensor contraction. SMA is a strong generalization of linear attention that is much more general than SSD as well; other forms of structured masks may lead to more variants of efficient attention with different properties than SSD.

Beside leading to new models, these connections to attention can lead to other directions for understanding SSMs. For example, we are curious whether the phenomenon of attention sinks (Darcet et al., 2024; Xiao et al., 2024) exist for Mamba models, and more broadly whether interpretability techniques can be transferred to SSMs (Ali et al., 2024).

Finally, many other variants of linear attention have been proposed (Schlag et al., 2021; Peng et al., 2021; Choromanski et al., 2021; Qin et al., 2022a;b; Zheng et al., 2022; Zhang et al., 2024; Arora et al., 2024a;b) (see Appendix B.1.3 for descriptions of several of these), and we expect that many techniques can be transferred to SSMs.

We emphasize that SSD **does not generalize standard softmax attention**, or any other transformation on the attention kernel matrix that does not have a finite feature map ψ. Compared to general attention, SSD's advantage is having a controllable state expansion factor $\texttt{N}$ that compresses the history, compared to quadratic attention's cache of the entire history scaling with sequence length $\texttt{T} \gg \texttt{N}$. Concurrent work has starting studying the tradeoffs of these representations, for example on copying and in-context learning tasks (Akyürek et al., 2024; Jelassi et al., 2024; Grazzi et al., 2024; Park et al., 2024). We note that Mamba-2 significantly improves on Mamba on some of these capabilities (e.g., as demonstrated by MQAR results in Section 4.1), but more remains to be understood.

A.4 Related Models

We finally highlight a growing body of recent and concurrent work that have developed sequence models very similar to Mamba and Mamba-2.

- RetNet (Sun et al., 2023) and TransNormerLLM (Qin et al., 2023b) generalize Linear Attention using decay terms instead of a cumulative sum, and propose dual parallel/recurrent algorithms as well as a hybrid "chunkwise" mode. These algorithms can be seen as an instantiation of SSD where A_t is time-invariant (constant for all t); in the SMA interpretation, the mask matrix L would be a decay matrix $L_{i,j} = \gamma^{i-j}$. These models also differ architecturally in various ways. For example, since they were derived from an attention-centric perspective they preserve the multi-head attention (MHA) pattern; since Mamba-2 was derived from an SSM-centric pattern it preserves the multi-value attention (MVA) or multi-expand SSM (MES) pattern, which we show to be better.
- GateLoop (Katsch, 2023) concurrently proposed using input-dependent decay factors A_t, and developed the same dual quadratic form as in SSD which they call a "surrogate attention" form.
- Gated Linear Attention (GLA) (Yang et al., 2024) proposed a variant of linear attention with data-dependent gates, along with efficient algorithms to compute a chunkwise mode and hardware-aware implementations.
- HGRN (Qin et al., 2023c) introduced an RNN with input-dependent gates, which was improved to incorporate state expansion in HGRN2 (Qin et al., 2024).
- Griffin (De et al., 2024) and RecurrentGemma (Botev et al., 2024) showed that an RNN with input-dependent gating, combined with local attention, can be very competitive with strong modern Transformers. Jamba also showed that combining Mamba with

a few layers of attention performs very well on language modeling (Lieber et al., 2024).

- xLSTM (Beck et al., 2024) improves the xLSTM by adopting the idea of state expansion and other gating, normalization, and stabilization techniques.
- RWKV(-4) (Peng et al., 2023) is an RNN based on a different linear attention approximation (the attention-free Transformer (Zhai et al., 2021)). It has recently been improved to the RWKV-5/6 (Eagle and Finch) architectures (Peng et al., 2024) by adopting the ideas of selectivity and state expansion.

B Structured Masked Attention: Generalizing Linear Attention with Structured Matrices

In this section we revisit the linear attention framework from first principles. The main results in this section are a simple tensor-contraction-based proof of linear attention (Proposition B.1), and our generalized abstraction of structured masked attention in Definition B.2.

- Appendix B.1 sets up our framework for variants of attention, with a particular focus on kernel attention and masked kernel attention.
- Appendix B.2 provides our first main attention result, a simple proof of linear attention through the lens of tensor contractions.
- Appendix B.3 defines structured masked attention, our generalization of prior attention variants through structured matrices.

B.1 The Attention Framework

B.1.1 Attention

The basic form of (single-head) attention is a map on three sequences of vectors $(Q,K,V) \mapsto Y$.

$$\begin{aligned} Q &= \mathsf{input} && (\mathtt{T},\mathtt{N}) \\ K &= \mathsf{input} && (\mathtt{S},\mathtt{N}) \\ V &= \mathsf{input} && (\mathtt{S},\mathtt{P}) \\ G &= QK^\top && (\mathtt{T},\mathtt{S}) \\ M &= f(G) && (\mathtt{T},\mathtt{S}) \\ Y &= GV && (\mathtt{T},\mathtt{P}) \end{aligned} \tag{8}$$

We use "shape annotations" to indicate the dimensions of tensors, e.g. $Q \in \mathbb{R}^{(\mathtt{T},\mathtt{N})}$. In this general form, $\mathtt{S}$ and $\mathtt{T}$ represent *source* and *target* sequence lengths, $\mathtt{N}$ represents the *feature dimension*, and $\mathtt{P}$ represents the *head dimension*.

The most common variant of **softmax attention** uses a softmax activation $f = \mathsf{softmax}$ to normalize the rows of the G matrix.

B.1.2 Self-Attention

Our treatment is motivated by the most important case of self-attention, where

(i) the source and target sequences are the same (i.e. $\mathtt{S} = \mathtt{T}$),

(ii) usually the feature and head dimensions are the same (i.e. $\mathtt{N} = \mathtt{P}$),

(iii) and Q,K,V are generated by linear projections on the same input vector ($Q = W_Q \cdot X, K = W_K \cdot X, V = W_V \cdot X$).

However, our presentation abstracts away these choices and begins from the Q,K,V matrices.

Remark 2. *Our focus is on the self-attention case with equal head and feature dimensions (i.e. $\mathtt{S} = \mathtt{T}$ and $\mathtt{N} = \mathtt{P}$), which should be used as the running example. We define the general formulation of attention not only so that our framework captures variants such as cross-attention, but also because separating the notation for dimensions (e.g. $\mathtt{S}$ and $\mathtt{T}$) makes the contraction notation proofs of our main results in this section more clear.*

Remark 3. *While attention is usually framed as an operation on these three inputs Q,K,V which are viewed symmetrically, the input and output dimensions in (8) indicate otherwise. In particular, the feature dimension $\mathtt{N}$ is not present in the output; therefore in the case when $\mathtt{S} = \mathtt{T}$ (e.g. self-attention), we view V as the main input, so that (8) defines a proper sequence transformation $V \mapsto Y$ (Definition 2.1).*

B.1.3 Kernel Attention

The step where the softmax function is applied to the Gram matrix G can be decomposed into two parts:

1. Exponentiating the G matrix.
2. Normalizing the G matrix on the $\mathtt{S}$ axis.

We can ignore the normalization term for now, as it amounts to simply passing in $V=1$ and dividing. The exponentiation term can be viewed as a kernel transformation: there is an (infinite-dimensional) feature map φ such that $\exp(QK^\top)=\varphi(Q)\varphi(K)^\top$. By abstracting away the feature map into the definition of Q and K itself (i.e. define Q,K as the post-transformed versions), we can ignore the softmax transformation, and assume that Q,K are arbitrarily generated by kernel feature maps and potentially $\mathtt{N}\neq\mathtt{P}$.

Many instantiations of kernel attention have been proposed, including:

- The original Linear Attention (Katharopoulos et al., 2020) defines the kernel feature map as an arbitrary pointwise activation function, such as $x\mapsto 1+\mathsf{elu}(x)$.
- Random Feature Attention (RFA) (Peng et al., 2021) chooses the kernel feature map to approximate softmax attention (i.e. the $\exp$ feature map) using the random Fourier feature approximation of Gaussian kernels (Rahimi & Recht, 2007). This involves random projections (i.e. multiplying Q and K by a random projection W and applying the activation $x\mapsto(\cos(x),\sin(x))$.
- Performer (Choromanski et al., 2021) proposes the fast attention via positive orthogonal random features (FAVOR+). The positive random features (PRF) part chooses the kernel feature map to be a random projection followed by the feature map $x\mapsto 2^{-1/2}(\exp(x),\exp(-x))$. This choice is motivated so that the kernel elements are positive-valued and provably approximates the softmax attention. [It also proposes choosing the random projections in orthogonal directions, which we do not consider.]
- cosFormer (Qin et al., 2022b) augment RFA with a cosine reweighting mechanism that incorporates positional information to emphasize locality. This effectively passes Q_t,K_t through the feature map $x\mapsto(x\cos(\pi t/2T),\sin(\pi t/2T))$.
- Linear Randomized Attention (Zheng et al., 2022) generalize RFA from the perspective of importance sampling, and generalize it to provide better estimates of the full softmax kernel (rather than just the $\exp$-transformed numerator).

Other related attention variants include Linformer (Wang et al., 2020) and Nyströformer (Xiong et al., 2021), which both use low-rank approximations of the attention matrix M (and are thus compatible with equation (8)), through random projections (Johnson-Lindenstrauss) and kernel approximation (the Nyström method) respectively.

B.1.4 Masked (Kernel) Attention

Let L be a mask of shape $(\mathtt{T},\mathtt{S})$. Most commonly, in the *autoregressive* self-attention case when $\mathtt{S}=\mathtt{T}$, L may be a lower-triangular matrix of 1's representing a *causal mask*. Besides enforcing causality, many other types of masks can be applied – in particular various sparsity patterns such as banded, dilated, or block diagonal – which are motivated by reducing the complexity of dense attention.

Masked attention is usually written in matrix notation as

$$y=(L\circ(QK^\top))\cdot V. \tag{9}$$

More precisely, with shape annotations and breaking this down into the precise sequence of computations:

$$\begin{aligned} G&=QK^\top && (\mathtt{T},\mathtt{S})\\ M&=G\circ L && (\mathtt{T},\mathtt{S})\\ Y&=MV && (\mathtt{T},\mathtt{P}) \end{aligned} \tag{10}$$

Our improved derivation of attention variants in this section starts by noticing that this formula can be written as a *single contraction*:

$$Y=\mathsf{contract}(\mathtt{TN},\mathtt{SN},\mathtt{SP},\mathtt{TS}\to\mathtt{TP})(Q,K,V,L) \tag{11}$$

and the algorithm in (10) can be reframed as computing (11) by a particular ordering of pairwise contractions

$$G=\mathsf{contract}(\mathtt{TN},\mathtt{SN}\to\mathtt{TS})(Q,K) \qquad (\mathtt{T},\mathtt{S}) \tag{12a}$$

$$M=\mathsf{contract}(\mathtt{TS},\mathtt{TS}\to\mathtt{TS})(G,L) \qquad (\mathtt{T},\mathtt{S}) \tag{12b}$$

$$Y=\mathsf{contract}(\mathtt{TS},\mathtt{SP}\to\mathtt{TP})(M,V) \qquad (\mathtt{T},\mathtt{P}) \tag{12c}$$

B.2 Linear Attention

Linear attention, and many other variants of efficient attention, is often motivated by changing the order of matrix associativity in the core attention computation $(QK^\top)V=Q(K^\top V)$. However when the mask is added, the derivation is somewhat less straightforward (for example, the original paper (Katharopoulos et al., 2020) and variants (Sun et al., 2023) state the formula without proof).

Roughly, the linear attention method claims that the following formula is equivalent to (9), which must be verified by expanding the sum and tracking indices carefully.

$$Y = Q \cdot \mathsf{cumsum}(K^\top V) \tag{13}$$

Proposition B.1 ((Katharopoulos et al., 2020)). *Autoregressive kernel attention, i.e. masked kernel attention with the causal mask, can be computed in $O(T)$ time by a recurrence taking constant time per step.*

B.2.1 A Tensor Contraction Proof of Linear Attention

We present a simple and rigorous derivation of linear attention that will also immediately reveal how to generalize it. The main idea is to perform the contraction (11) in an alternate order. We avoid ambiguous matrix notation and work directly with contraction notation:

$$Z = \mathsf{contract}(\mathtt{SP},\mathtt{SN} \to \mathtt{SPN})(V,K) \qquad (\mathtt{S},\mathtt{P},\mathtt{N}) \tag{14a}$$

$$H = \mathsf{contract}(\mathtt{TS},\mathtt{SPN} \to \mathtt{TPN})(L,Z) \qquad (\mathtt{T},\mathtt{P},\mathtt{N}) \tag{14b}$$

$$Y = \mathsf{contract}(\mathtt{TN},\mathtt{TPN} \to \mathtt{TP})(Q,H) \qquad (\mathtt{T},\mathtt{P}) \tag{14c}$$

Intuitively, we interpret this contraction order as follows.

The first step (14a) performs an "expansion" into more features, by a factor of the feature dimension $\mathtt{N}$. The third step (14c) contracts the expanded feature dimension away. If K is viewed as the input (Remark 3), then V and Q perform the expansion and contraction, respectively.

The second step is the most critical, and explains the *linear* part of linear attention. First notice that (14b) is just a direct matrix multiplication by L (since the $(\mathtt{P},\mathtt{N})$ axes can be flattened). Also note that this is the only term that involves both $\mathtt{T}$ and $\mathtt{S}$ axes, hence should have $\Omega(\mathtt{TS})$ complexity (i.e. quadratic in sequence length). However, when the mask L is the standard causal attention mask (lower triangular 1's), matrix-vector multiplication by L is identical to a feature-wise cumulative sum

$$y = \begin{bmatrix} 1 & & \\ \vdots & \ddots & \\ 1 & \dots & 1 \end{bmatrix} x \quad \iff \quad \begin{aligned} y_0 &= x_0 \\ y_t &= y_{t-1} + x_t \end{aligned}.$$

B.3 Structured Masked Attention

With the tensor contraction perspective of masked attention (14), we can immediately see that the crux of the original linear attention is the fact that *matrix-vector multiplication by the causal mask is equivalent to the cumulative sum operator*.

However, we observe that there is no reason the attention mask has to be all 1's. All that is necessary for linear attention to be fast is for L to be a *structured matrix*, which by definition are those that have fast matrix multiplication. In particular, we can use *any mask matrix* L that has sub-quadratic (ideally linear) matrix-vector multiplication, which would have the same complexity as standard linear attention by speeding up the bottleneck equation (14b).

Definition B.2. ***Structured masked attention (SMA)*** *(or* ***structured attention*** *for short) is defined as a* function *on queries/keys/values Q,K,V as well as any* structured matrix *L (i.e. has sub-quadratic matrix multiplication), through the 4-way tensor contraction*

$$Y = \mathsf{contract}(\mathtt{TN},\mathtt{SN},\mathtt{SP},\mathtt{TS} \to \mathtt{TP})(Q,K,V,L).$$

The SMA ***quadratic mode algorithm*** *is the sequence of pairwise contractions defined by (12), which corresponds to the standard (masked) attention computation.*

The SMA ***linear mode algorithm*** *is the sequence of pairwise contractions defined by (14), where step (14b) is optimized through the subquadratic structured matrix multiplication.*

We can instantiate structured masked attention to any given class of matrix structure. Some examples include (Figure 6):

- Linear attention uses a causal mask.
- RetNet (Sun et al., 2023) uses a decay mask $L_{ij} = \gamma^{i-j} \cdot \mathbb{I}[j \geq i]$ for some decay factor $\gamma \in [0,1]$.
- The decay mask could be generalized to a Toeplitz matrix $L_{ij} = \alpha_{i-j}$ for some learnable (or input-dependent) set of parameters $\alpha \in \mathbb{R}^{\mathtt{T}}$. This can be interpreted as a form of relative positional encoding, reminiscent of other methods such as AliBi (Press et al., 2022) but multiplicative instead of additive.
- Another variant could use a Fourier matrix $L_{ij} = \omega^{ij/\mathtt{T}}$ to encode positional structure a different way.

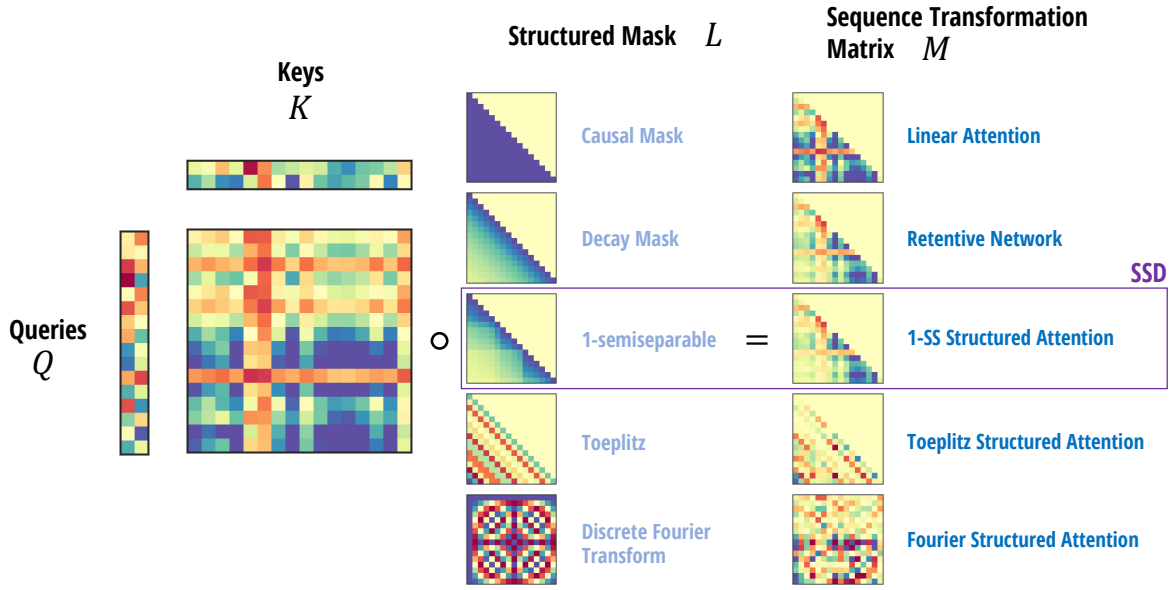

Figure 6: (**Structured Masked Attention**.) SMA constructs a masked attention matrix $M = QK^\top \circ L$ for any structured matrix L, which defines a matrix sequence transformation $Y = MV$. All instances of SMA have a dual subquadratic form induced by a different contraction ordering, combined with the efficient structured matrix multiplication by L. Previous examples include Linear Attention (Katharopoulos et al., 2020) and RetNet (Sun et al., 2023). Beyond SSD (1-semiseparable SMA), the focus of this paper, many other potential instantiations of structured attention are possible.

B.3.1 Summary: The Dual Forms of Masked Attention

Standard (masked kernel) attention is often conflated between a function and an algorithm. Separating this distinction presents a clear way to understand different variants of attention.

- We view **masked attention** as a particular *function* (11).
- The standard **quadratic attention** computation (12) can be viewed as an *algorithm* to compute the function.
- **Linear attention** (14) is an alternate algorithm to compute the same function.

Moreover, in this case

- The masked attention function is simply a particular *contraction on four terms*.
- The quadratic and linear attention algorithms are simply *two different orders to perform the contractions*.

It is known that contraction orderings can make large differences in computation complexity, leading to the quadratic vs. linear split. Just as state space models are a transformation that can be computed in multiple ways, with dual quadratic vs. linear forms (Section 3.4), linear attention has a similar duality that results from two contraction orders.

C A Hardware-Efficient Algorithm for SSD Models

The benefits of developing the theoretical SSD framework between SSMs, attention, and structured matrices lies in using the connections to improve the models and algorithms. In this section, we show how various algorithms for computing SSD models efficiently can be derived from various algorithms for computing structured matrix multiplication.

Remark 4 (1-SS matrix multiplication)**.** *Before the defails of the algorithm, we recommend also reading Appendix D, which covers a variety of efficient algorithms for the* scalar SSM *setting (in other words, 1-semiseparable matrix multiplication) from the perspective of matrix decompositions. These algorithms not only provide an alternative perspective on classical algorithms and support one of our main themes—efficient algorithms through structured matrix factorizations—but 1-semiseparable matrix multiplication is also necessary as an intermediate step in the full SSD algorithm.*

C.1 Block Decomposition of Semiseparable Matrices

To begin, we partition the matrix M into a $\frac{\mathtt{T}}{\mathtt{Q}} \times \frac{\mathtt{T}}{\mathtt{Q}}$ grid of submatrices of size $\mathtt{Q} \times \mathtt{Q}$, for some block size $\mathtt{Q}$. Note that the off-diagonal blocks are low-rank by the defining property of semiseparable matrices (Definition 3.1).[4]

$$\text{(Block Decomposition)} \quad M = \begin{bmatrix} M^{(0,0)} & & & \\ M^{(1,0)} & M^{(1,1)} & & \\ \vdots & \vdots & \ddots & \\ M^{(\mathtt{T}/\mathtt{Q}-1,0)} & M^{(\mathtt{T}/\mathtt{Q}-1,1)} & \dots & M^{(\mathtt{T}/\mathtt{Q}-1,\mathtt{T}/\mathtt{Q}-1)} \end{bmatrix}$$

$$\text{(Diagonal Block)} \quad M^{(j,j)} = \mathsf{SSM}(A_{j\mathtt{Q}:(j+1)\mathtt{Q}}, B_{j\mathtt{Q}:(j+1)\mathtt{Q}}, C_{j\mathtt{Q}:(j+1)\mathtt{Q}})$$

$$\text{(Low-Rank Block)} \quad M^{(j,i)} = \begin{bmatrix} C_{j\mathtt{Q}}^\top A_{j\mathtt{Q}:j\mathtt{Q}-1} \\ \vdots \\ C_{(j+1)\mathtt{Q}-1}^\top A_{(j+1)\mathtt{Q}-1:j\mathtt{Q}-1} \end{bmatrix} A_{j\mathtt{Q}-1:(i+1)\mathtt{Q}-1} \begin{bmatrix} B_{i\mathtt{Q}}^\top A_{(i+1)\mathtt{Q}-1:i\mathtt{Q}} \\ \vdots \\ B_{(i+1)\mathtt{Q}-1}^\top A_{(i+1)\mathtt{Q}-1:(i+1)\mathtt{Q}-1} \end{bmatrix}^\top$$

This is easiest illustrated through an example, e.g. for $\mathtt{T} = 9$ and decomposing into chunks of length $\mathtt{Q} = 3$. The shaded cells are low-rank factorizations of the off-diagonal blocks of the semiseparable matrix.

$$\left[\begin{array}{ccc|ccc|ccc} C_0^\top A_{0:0} B_0 & & & & & & & & \\ C_1^\top A_{1:0} B_0 & C_1^\top A_{1:1} B_1 & & & & & & & \\ C_2^\top A_{2:0} B_0 & C_2^\top A_{2:1} B_1 & C_2^\top A_{2:2} B_2 & & & & & & \\ \hline C_3^\top A_{3:0} B_0 & C_3^\top A_{3:1} B_1 & C_3^\top A_{3:2} B_2 & C_3^\top A_{3:3} B_3 & & & & & \\ C_4^\top A_{4:0} B_0 & C_4^\top A_{4:1} B_1 & C_4^\top A_{4:2} B_2 & C_4^\top A_{4:3} B_3 & C_4^\top A_{4:4} B_4 & & & & \\ C_5^\top A_{5:0} B_0 & C_5^\top A_{5:1} B_1 & C_5^\top A_{5:2} B_2 & C_5^\top A_{5:3} B_3 & C_5^\top A_{5:4} B_4 & C_5^\top A_{5:5} B_5 & & & \\ \hline C_6^\top A_{6:0} B_0 & C_6^\top A_{6:1} B_1 & C_6^\top A_{6:2} B_2 & C_6^\top A_{6:3} B_3 & C_6^\top A_{6:4} B_4 & C_6^\top A_{6:5} B_5 & C_6^\top A_{6:6} B_6 & & \\ C_7^\top A_{7:0} B_0 & C_7^\top A_{7:1} B_1 & C_7^\top A_{7:2} B_2 & C_7^\top A_{7:3} B_3 & C_7^\top A_{7:4} B_4 & C_7^\top A_{7:5} B_5 & C_7^\top A_{7:6} B_6 & C_7^\top A_{7:7} B_7 & \\ C_8^\top A_{8:0} B_0 & C_8^\top A_{8:1} B_1 & C_8^\top A_{8:2} B_2 & C_8^\top A_{8:3} B_3 & C_8^\top A_{8:4} B_4 & C_8^\top A_{8:5} B_5 & C_8^\top A_{8:6} B_6 & C_8^\top A_{8:7} B_7 & C_8^\top A_{8:8} B_8 \end{array}\right]$$

$$= \left[\begin{array}{ccc|ccc|ccc} C_0^\top A_{0:0} B_0 & & & & & & & & \\ C_1^\top A_{1:0} B_0 & C_1^\top A_{1:1} B_1 & & & & & & & \\ C_2^\top A_{2:0} B_0 & C_2^\top A_{2:1} B_1 & C_2^\top A_{2:2} B_2 & & & & & & \\ \hline & \begin{bmatrix} C_3^\top A_{3:2} \\ C_4^\top A_{4:2} \\ C_5^\top A_{5:2} \end{bmatrix} A_{2:2} \begin{bmatrix} B_0^\top A_{2:0} \\ B_1^\top A_{2:1} \\ B_2^\top A_{2:2} \end{bmatrix}^\top & & C_3^\top A_{3:3} B_3 & & & & & \\ & & & C_4^\top A_{4:3} B_3 & C_4^\top A_{4:4} B_4 & & & & \\ & & & C_5^\top A_{5:3} B_3 & C_5^\top A_{5:4} B_4 & C_5^\top A_{5:5} B_5 & & & \\ \hline & \begin{bmatrix} C_6^\top A_{6:5} \\ C_7^\top A_{7:5} \\ C_8^\top A_{8:5} \end{bmatrix} A_{5:2} \begin{bmatrix} B_0^\top A_{2:0} \\ B_1^\top A_{2:1} \\ B_2^\top A_{2:2} \end{bmatrix}^\top & & \begin{bmatrix} C_6^\top A_{6:5} \\ C_7^\top A_{7:5} \\ C_8^\top A_{8:5} \end{bmatrix} A_{5:5} \begin{bmatrix} B_3^\top A_{5:3} \\ B_4^\top A_{5:4} \\ B_5^\top A_{5:5} \end{bmatrix}^\top & & C_6^\top A_{6:6} B_6 & & \\ & & & & & & C_7^\top A_{7:6} B_6 & C_7^\top A_{7:7} B_7 & \\ & & & & & & C_8^\top A_{8:6} B_6 & C_8^\top A_{8:7} B_7 & C_8^\top A_{8:8} B_8 \end{array}\right]$$

From here we can reduce the problem into these two parts. These can also be interpreted as dividing the output of a "chunk" $y_{j\mathtt{Q}:(j+1)\mathtt{Q}}$ into two components: the effect of inputs within the chunk $x_{j\mathtt{Q}:(j+1)\mathtt{Q}}$, and the effect of inputs before the chunk $x_{0:j\mathtt{Q}}$.

C.2 Diagonal Blocks

The diagonal blocks are easy to handle, because they are simply self-similar problems of a smaller size. The j-th block represents computing the answer $\mathsf{SSM}(A_R, B_R, C_R)(x_R)$ for the range $R = j\mathtt{Q}:(j+1)\mathtt{Q} = (j\mathtt{Q}, j\mathtt{Q}+1, \dots, j\mathtt{Q}+\mathtt{Q}-1)$. The key is that this block can be computed using any desired method. In particular, for small chunk lengths $\mathtt{Q}$, this problem is computed more efficiently using the dual quadratic SMA form. Additionally, the chunks can be computed in parallel.

These subproblems can be interpreted as: what is the output per chunk *supposing that the initial state (to the chunk) is* 0. In other words for chunk j, this computes the correct outputs taking into account only the chunk inputs $x_{j\mathtt{Q}:(j+1)\mathtt{Q}}$.

[4] Note that the block decomposition is valid even with partitions of varying size, e.g. if $\mathtt{Q} \nmid \mathtt{T}$, but we assume even divisibility for simplicity.

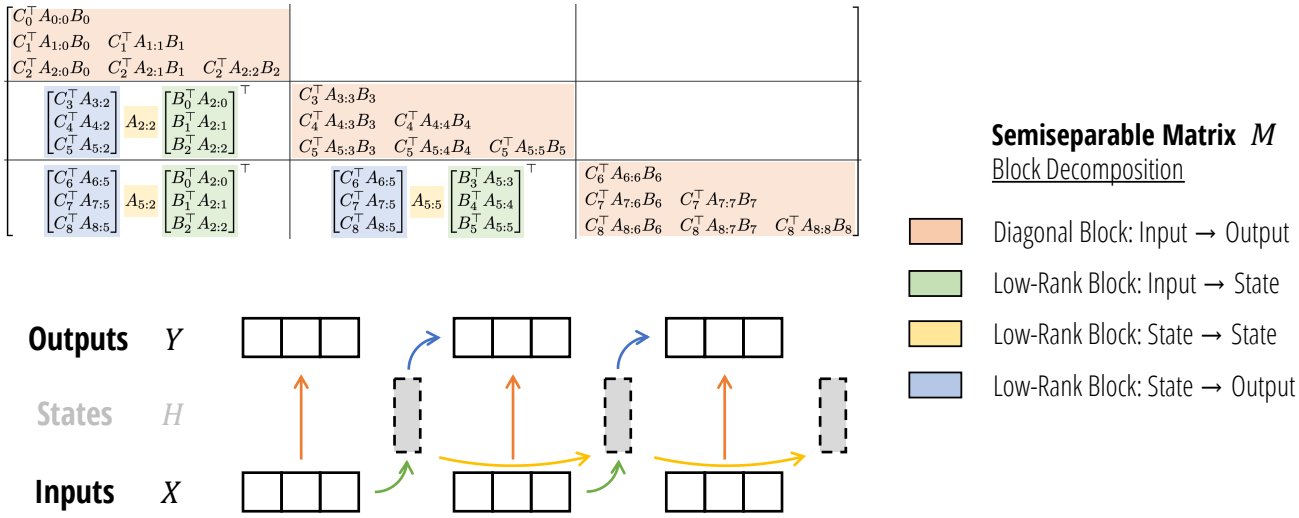

Figure 7: (**SSD Algorithm**.) By using the matrix transformation viewpoint of state space models to write them as semiseparable matrices (Section 3.3), we develop a more hardware-efficient computation of the SSD model through a block-decomposition matrix multiplication algorithm. The matrix multiplication also has an interpretation as a state space model, where blocks represent chunking the input and output sequence. Diagonal blocks represent intra-chunk computations and the off-diagonal blocks represent inter-chunk computations, factored through the SSM's hidden state.

C.3 Low-Rank Blocks

The low-rank factorizations consist of 3 terms, and there are correspondingly three pieces of the computation. In this factorization, we will use the terminology

- The terms like $\begin{bmatrix} B_0^\top A_{2:0} \\ B_1^\top A_{2:1} \\ B_2^\top A_{2:2} \end{bmatrix}^\top$ are called the right factors or B-block-factors.
- The terms like $A_{5:2}$ are called the center factors or A-block-factors.
- The terms like $\begin{bmatrix} C_6^\top A_{6:5} \\ C_7^\top A_{7:5} \\ C_8^\top A_{8:5} \end{bmatrix}$ are called the left factors or C-block-factors.

Right Factors. This step computes the multiplication by the right B-block-factors of the low-rank factorization. Note that for each chunk, this is a $(\mathtt{N},\mathtt{Q})$ by $(\mathtt{Q},\mathtt{P})$ matrix multiplication, where $\mathtt{N}$ is the state dimension and P is the head dimension. The result is a $(\mathtt{N},\mathtt{P})$ tensor for each chunk, which has the same dimensionality as the expanded hidden state h.

This can be interpreted as: what is the final state per chunk *supposing that the initial state (to the chunk) is* 0. In other words this computes $h_{j\mathtt{Q}+\mathtt{Q}-1}$ assuming that $x_{0:j\mathtt{Q}}=0$.

Center Factors. This step computes the effect of the center A-block-factors terms in the low-rank factorization. In the previous step, the final states per chunk have total shape $(\mathtt{T}/\mathtt{Q},\mathtt{N},\mathtt{P})$. This is now multiplied by a 1-SS matrix generated by $A^\times_{2\mathtt{Q}-1:\mathtt{Q}-1}, A^\times_{3\mathtt{Q}-1:2\mathtt{Q}-1}, \ldots, A^\times_{\mathtt{T}-1:\mathtt{T}-\mathtt{Q}-1}$.

This step can be computed by any algorithm for computing 1-SS multiplication (also known as the scalar SSM scan or `cumprodsum` operator).

This can be interpreted as: what is the actual final state per chunk *taking into account all previous inputs*; in other words, this computes the true hidden state $h_{j\mathtt{Q}}$ taking into account all of $x_{0:(j+1)\mathtt{Q}}$.

Left Factors. This step computes the multiplication by the left C-block-factors of the low-rank factorization. For each chunk, this can be represented by a matrix multiplication $\mathsf{contract}(\mathtt{QN},\mathtt{NP}\to\mathtt{QP})$.

This can be interpreted as: what is the output per chunk *taking into account the correct initial state* $h_{j\mathtt{Q}-1}$, *and supposing the inputs* $x_{j\mathtt{Q}:(j+1)\mathtt{Q}}$ *are* 0. In other words for chunk j, this computes the correct outputs taking into account only the prior inputs $x_{0:j\mathtt{Q}}$.

Listing 1 Full PyTorch example of the state space dual (SSD) model.

```
def segsum(x):
    """Naive segment sum calculation. exp(segsum(A)) produces a 1-SS matrix,
       which is equivalent to a scalar SSM."""
    T = x.size(-1)
    x_cumsum = torch.cumsum(x, dim=-1)
    x_segsum = x_cumsum[..., :, None] - x_cumsum[..., None, :]
    mask = torch.tril(torch.ones(T, T, device=x.device, dtype=bool), diagonal=0)
    x_segsum = x_segsum.masked_fill(~mask, -torch.inf)
    return x_segsum

def ssd(X, A, B, C, block_len=64, initial_states=None):
    """
    Arguments:
        X: (batch, length, n_heads, d_head)
        A: (batch, length, n_heads)
        B: (batch, length, n_heads, d_state)
        C: (batch, length, n_heads, d_state)
    Return:
        Y: (batch, length, n_heads, d_head)
    """
    assert X.dtype == A.dtype == B.dtype == C.dtype
    assert X.shape[1] % block_len == 0

    # Rearrange into blocks/chunks
    X, A, B, C = [rearrange(x, "b (c l) ... -> b c l ...", l=block_len) for x in (X, A, B, C)]

    A = rearrange(A, "b c l h -> b h c l")
    A_cumsum = torch.cumsum(A, dim=-1)

    # 1. Compute the output for each intra-chunk (diagonal blocks)
    L = torch.exp(segsum(A))
    Y_diag  = torch.einsum("bclhn,bcshn,bhcls,bcshp->bclhp", C, B, L, X)

    # 2. Compute the state for each intra-chunk
    # (right term of low-rank factorization of off-diagonal blocks; B terms)
    decay_states = torch.exp((A_cumsum[:, :, :, -1:] - A_cumsum))
    states = torch.einsum("bclhn,bhcl,bclhp->bchpn", B, decay_states, X)

    # 3. Compute the inter-chunk SSM recurrence; produces correct SSM states at chunk boundaries
    # (middle term of factorization of off-diag blocks; A terms)
    if initial_states is None:
        initial_states = torch.zeros_like(states[:, :1])
    states = torch.cat([initial_states, states], dim=1)
    decay_chunk = torch.exp(segsum(F.pad(A_cumsum[:, :, :, -1], (1, 0))))
    new_states = torch.einsum("bhzc,bchpn->bzhpn", decay_chunk, states)
    states, final_state = new_states[:, :-1], new_states[:, -1]

    # 4. Compute state -> output conversion per chunk
    # (left term of low-rank factorization of off-diagonal blocks; C terms)
    state_decay_out = torch.exp(A_cumsum)
    Y_off = torch.einsum('bclhn,bchpn,bhcl->bclhp', C, states, state_decay_out)

    # Add output of intra-chunk and inter-chunk terms (diagonal and off-diagonal blocks)
    Y = rearrange(Y_diag+Y_off, "b c l h p -> b (c l) h p")
    return Y, final_state
```

C.4 Computational Cost

We define the notation $\mathsf{BMM}(\mathtt{B},\mathtt{M},\mathtt{N},\mathtt{K})$ to define a batched matrix multiplication $\mathsf{contract}(\mathtt{MK},\mathtt{KN}\to\mathtt{MN})$ with batch dimension $\mathtt{B}$. From this notation we can infer three aspects of the efficiency:

- *Computation cost*: total of $O(\mathtt{BMNK})$ FLOPs.
- *Memory cost:* total of $O(\mathtt{B}(\mathtt{MK}+\mathtt{KN}+\mathtt{MN}))$ space.

- *Parallelization:* larger M,N,K terms can leverage specialized matrix multiplication units on modern accelerators.

Center Blocks. The cost of the quadratic SMA computation consists of three steps (equation (7)):

- Computing the kernel matrix $C^\top B$, which has cost $\mathsf{BMM}(\mathtt{T/Q},\mathtt{Q},\mathtt{Q},\mathtt{N})$.
- Multiplying by the mask matrix, which is an elementwise operation on tensors of shape $(\mathtt{T/Q},\mathtt{Q},\mathtt{Q})$.
- Multiplying by the X values, which has cost $\mathsf{BMM}(\mathtt{T/Q},\mathtt{Q},\mathtt{P},\mathtt{N})$

Low-Rank Blocks: Right Factors. This step is a single matrix multiplication with cost $\mathsf{BMM}(\mathtt{T/Q},\mathtt{N},\mathtt{P},\mathtt{Q})$.

Low-Rank Blocks: Center Factors. This step is a scalar SSM scan (or 1-SS multiplication) of length $\mathtt{T/Q}$ on $(\mathtt{N},\mathtt{P})$ independent channels. The work of this scan is $\mathtt{TNP/Q}$, which is negligible compared to the other factors.

Note that because of the blocking which reduces the length of the sequence from T to $\mathtt{T/Q}$, this scan has Q times smaller cost than a pure SSM scan (e.g. the selective scan of Mamba). Thus we observe that on most problem lengths, other algorithms (Appendix D) may be more efficient or much easier to implement without a significant slowdown. For example, a naive implementation of this via 1-SS matrix multiplication has cost $\mathsf{BMM}(1,\mathtt{T/Q},\mathtt{NP},\mathtt{T/Q})$, which is much easier to implement and can be more efficient than a naive recurrence/scan implementation.

Low-Rank Blocks: Left Factors. This step is a single matrix multiplication with cost $\mathsf{BMM}(\mathtt{T/Q},\mathtt{Q},\mathtt{P},\mathtt{N})$.

Total Cost. If we set $\mathtt{N}=\mathtt{P}=\mathtt{Q}$ (in other words the state dimension, head dimension, and chunk length are equal), then all BMM terms above become $\mathsf{BMM}(\mathtt{T/N},\mathtt{N},\mathtt{N},\mathtt{N})$. The computational chacteristics of this are:

- Total FLOP count of $O(\mathtt{TN}^2)$.
- Total memory of $O(\mathtt{TN})$.
- The work *consists primarily of matrix multiplications* on matrices of shape $(\mathtt{N},\mathtt{N})$.

Notice that the memory consumption is tight; the inputs and outputs x,y have shape $(\mathtt{T},\mathtt{P})=(\mathtt{T},\mathtt{N})$. Meanwhile the flop count reflects an extra factor of N, which is cost incurred by the autoregressive state size and is common to all models.

Aside from the matmuls, there is a scalar SSM scan on $\mathtt{NP}=\mathtt{N}^2$ features and sequence length $\mathtt{T/Q}$. This has cost $O(\mathtt{T/QN}^2)$ FLOPs and $O(\log(\mathtt{T/Q}))$ depth. Although it does not use matrix multiplications, it is still parallelizable and the total work done is negligible compared to the other steps; this has a negligible cost in our GPU implementation.

Comparison to Pure SSM and Attention Models. Quadratic attention is also very hardware efficient by only leveraging matrix multiplications, but has $\mathtt{T}^2N$ total FLOPs. Its slower computation speed at both training and inference can directly be seen as a consequence of having a larger state size – standard attention has a state size scaling with sequence length T because it caches its history and does not compress its state.

Linear SSMs have $\mathtt{TNP}=\mathtt{TN}^2$ total FLOPs, which is the same as SSD. However, a naive implementation requires a state expansion (14a) that materializes extra memory, and a scalar operation (14b) that does not leverage matrix multiplications.

	Attention	SSM	**SSD**
State size	$\mathtt{T}$	$\mathbf{N}$	$\mathbf{N}$
Training FLOPs	$\mathtt{T}^2\mathtt{N}$	$\mathbf{TN}^2$	$\mathbf{TN}^2$
Inference FLOPs	$\mathtt{TN}$	$\mathbf{N}^2$	$\mathbf{N}^2$
(Naive) memory	$\mathtt{T}^2$	$\mathtt{TN}^2$	$\mathbf{TN}$
Matrix multiplication	✓		✓

We note that many other matrix decompositions are possible (for example, see Appendix D for a compendium of algorithms for 1-SS multiplication through different structured matrix decompositions) which may lead to more algorithms for SSDs that could be better for other specialized settings. Even more broadly, we note that semiseparable matrices have a rich literature and many more representations besides the SSS form that we use (Definition 3.2), and even more efficient algorithms may be possible.

D Efficient Algorithms for the Scalar SSM Scan (1-SS Multiplication)

In this section we flesh out various algorithms for computing the scalar SSM scan, through the lens of structured matrix decompositions. The scalar SSM scan is defined as computing the recurrent part of the discrete SSM (5), in the case when $N=1$ (i.e. A is a scalar). This is commonly used to compute SSMs recurrently; in particular, the case of structured SSMs where A is diagonally structured reduces down to this operation, such as in the S5 (Smith et al., 2023) and S6 (Gu & Dao, 2023) models.

The goal of this section is to support a central theme of this paper that *efficient algorithms for sequence models can be viewed as structured matrix multiplication algorithms*. The various matrix decomposition ideas we show here are related to ideas used to derive fast SSM algorithms (Appendix C), as well as directly used as a subroutine.

D.1 Problem Definition

Let $a : (\mathtt{T})$ and $b : (\mathtt{T})$ be sequences of scalars. The **scalar SSM scan** is defined as

$$h_t = a_t h_{t-1} + b_t. \tag{15}$$

Here h_{-1} can be an arbitrary value representing the previous *hidden state* to the SSM recurrence; unless otherwise specified, we assume $h_{-1} = 0$.

We also call equation (15) the **`cumprodsum`** (cumulative product sum). Note that the `cumprodsum` reduces to the `cumprod` (cumulative product) when $b = 0$ is the additive identity and it reduces to the `cumsum` (cumulative sum) when $a = 1$ is the multiplicative identity.

Finally, note that in vectorized form we can write

$$\begin{aligned} h &= Mb \\ M &= \begin{bmatrix} 1 & & & & \\ a_1 & 1 & & & \\ a_2 a_1 & a_2 & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ a_{\mathtt{T}-1} \dots a_1 & a_{\mathtt{T}-1} \dots a_2 & \dots & a_{\mathtt{T}-1} & 1 \end{bmatrix} \end{aligned}$$

In other words, this is simply the matrix-vector product by a 1-SS matrix M.

Therefore we have three ways of viewing this fundamental primitive operation that are all equivalent:

- A (scalar) SSM scan.
- A `cumprodsum`.
- A 1-SS matrix-vector multiplication .

D.2 Classical Algorithms

We first describe the two classical ways of computing the SSM scan (15), previously used by prior work.

D.2.1 Sequential Recurrence

The recurrent mode simply computes (15) one timestep t at a time. From the perspective of 1-SS multiplication, this was also described in Section 3.5.

D.2.2 Parallel Associative Scan

Second, an important observation is that this recurrence can be turned into an associative scan (Martin & Cundy, 2018; Smith et al., 2023). This fact is not completely obvious. For example, S5 defined the correct associative scan operator and then showed associativity of the operator through rote calculation.

A slightly cleaner way to see that this is computable with an associative scan is to turn the multi-term recurrence into a single-term recurrence on a hidden state of size 2 instead of 1:

$$\begin{aligned} h_t &= a_t h_{t-1} + b_t \\ \begin{bmatrix} h_t \\ 1 \end{bmatrix} &= \begin{bmatrix} a_t & b_t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} h_{t-1} \\ 1 \end{bmatrix}. \end{aligned}$$

Then computing all the h_t is the same as taking the cumulative products of these 2×2 matrices. Since matrix multiplication is associative, this can be computed with an associative scan. The associative binary operator is simply matrix multiplication on these particular matrices:

$$\begin{bmatrix} a_t & b_t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} a_s & b_s \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} a_t a_s & a_t b_s + b_t \\ 0 & 1 \end{bmatrix}.$$

Equating the top row yields the same associative scan operator as defined by S5:

$$(a_t,b_t)\otimes(a_s,b_s)=(a_t a_s,a_t b_s+b_t). \tag{16}$$

The reason why associative scans are important is that they can be parallelized using a divide-and-conquer algorithm (Blelloch, 1990). We omit the details of this algorithm, and instead show that the entire associative SSM scan algorithm can be derived from scratch through matrix decompositions (Appendix D.3.5).

D.3 Efficient Algorithms via Structured Matrix Decompositions

We discuss several algorithms for computing the SSM scan, all through the lens of finding structured matrix decompositions of the 1-SS matrix M. These algorithms or computation modes include

- A *dilated* mode where information is propagated 1,2,4,8,... steps at a time.
- A *state-passing* mode where information is propagated forward in chunks.
- A *fully recurrent* mode that increments one step at a time, which is a special case of the state-passing mode.
- A *block decomposition* parallel mode where M is divided into hierarchical blocks.
- A *scan* mode where M is divide into equal size blocks and reduced recursively.

D.3.1 Dilated Mode

This mode factors the 1-SS matrix in a particular way involving increasing "strides". This is best illustrated through a concrete example:

$$M=\begin{bmatrix}
a_{0:0} & & & & & & & \\
a_{1:0} & a_{1:1} & & & & & & \\
a_{2:0} & a_{2:1} & a_{2:2} & & & & & \\
a_{3:0} & a_{3:1} & a_{3:2} & a_{3:3} & & & & \\
a_{4:0} & a_{4:1} & a_{4:2} & a_{4:3} & a_{4:4} & & & \\
a_{5:0} & a_{5:1} & a_{5:2} & a_{5:3} & a_{5:4} & a_{5:5} & & \\
a_{6:0} & a_{6:1} & a_{6:2} & a_{6:3} & a_{6:4} & a_{6:5} & a_{6:6} & \\
a_{7:0} & a_{7:1} & a_{7:2} & a_{7:3} & a_{7:4} & a_{7:5} & a_{7:6} & a_{7:7}
\end{bmatrix}$$

$$=\begin{bmatrix}
a_{0:0} & & & & & & & \\
 & a_{1:1} & & & & & & \\
 & & a_{2:2} & & & & & \\
 & & & a_{3:3} & & & & \\
a_{4:0} & & & & a_{4:4} & & & \\
 & a_{5:1} & & & & a_{5:5} & & \\
 & & a_{6:2} & & & & a_{6:6} & \\
 & & & a_{7:3} & & & & a_{7:7}
\end{bmatrix}
\begin{bmatrix}
a_{0:0} & & & & & & & \\
 & a_{1:1} & & & & & & \\
a_{2:0} & & a_{2:2} & & & & & \\
 & a_{3:1} & & a_{3:3} & & & & \\
 & & a_{4:2} & & a_{4:4} & & & \\
 & & & a_{5:3} & & a_{5:5} & & \\
 & & & & a_{6:4} & & a_{6:6} & \\
 & & & & & a_{7:5} & & a_{7:7}
\end{bmatrix}
\begin{bmatrix}
a_{0:0} & & & & & & & \\
a_{1:0} & a_{1:1} & & & & & & \\
 & a_{2:1} & a_{2:2} & & & & & \\
 & & a_{3:2} & a_{3:3} & & & & \\
 & & & a_{4:3} & a_{4:4} & & & \\
 & & & & a_{5:4} & a_{5:5} & & \\
 & & & & & a_{6:5} & a_{6:6} & \\
 & & & & & & a_{7:6} & a_{7:7}
\end{bmatrix}$$

Note that this closely resembles the computation of dilated convolutions.

We also note that this factorization shows that 1-SS matrices are a special case of butterfly matrices, another broad and fundamental type of structured matrix (Dao et al., 2019; 2020).

Remark 5. *This algorithm is sometimes described as a "work-inefficient but more parallelizable" prefix sum algorithm (Hillis & Steele Jr, 1986), becauses it uses $O(\mathtt{T}\log\mathtt{T})$ operations but has half the depth/span as the work-efficient associative scan algorithm.*

D.3.2 State-Passing (Chunkwise) Mode

This mode can be viewed as a generalization of the standard recurrent mode where instead of passing forward the recurrent state h one step at a time, we compute the answer on chunks of arbitrary length k and pass the state through the chunk. This can also be derived from a simple block decomposition of the 1-SS matrix.

Remark 6. *While we call this "state-passing" to refer to how states are passed from one local segment to another, this is related to the "chunkwise" algorithms proposed by related models (Sun et al., 2023; Yang et al., 2024).*

Consider computing $h=Mb$ in "chunks": for some index $k\in[\mathtt{T}]$, we want to compute $h_{0:k}$ or the output up to index k, and have a way to reduce the problem to a smaller problem on indices $[k:\mathtt{T}]$.

We write M as

$$M = \begin{bmatrix} a_{0:0} & & & & & & & \\ a_{1:0} & a_{1:1} & & & & & & \\ \vdots & & \ddots & & & & & \\ a_{k-1:0} & \cdots & \cdots & a_{k-1:k-1} & & & & \\ a_{k:0} & \cdots & \cdots & a_{k:k-1} & a_{k:k} & & & \\ \vdots & & & \vdots & \vdots & \ddots & & \\ a_{\mathrm{T}-1:0} & \cdots & \cdots & a_{\mathrm{T}-1:k-1} & a_{\mathrm{T}-1:k} & \cdots & a_{\mathrm{T}-1:\mathrm{T}-1} \end{bmatrix}$$

Let the upper-left triangle be M_L, lower-right be M_R (left and right subproblems), and lower-left be M_C. Divide up b into $b_L = b_{0:k}$ and $b_R = b_{k:\mathrm{T}}$ in the same way. Note that

$$Mb = \begin{bmatrix} M_L b_L \\ M_R b_R + M_C b_L \end{bmatrix}$$

Also, M_C has the rank-1 factorization (this is essentially the defining property of semiseparable matrices)

$$M_C = \begin{bmatrix} a_{k:k} \\ \vdots \\ a_{\mathrm{T}-1:k} \end{bmatrix} a_k \begin{bmatrix} a_{k-1:0} & \cdots & a_{k-1:k-1} \end{bmatrix}$$

Thus

$$M_C b_L = \begin{bmatrix} a_{k:k} \\ \vdots \\ a_{\mathrm{T}-1:k} \end{bmatrix} a_k \cdot (Mb)_{k-1}.$$

Here we think of $(Mb)_{k-1} = h_{k-1}$ as the "final state" of the left chunk, because the row vector in M_C's factorization is the same as the final row of M_L. Furthermore, note that the column vector in M_C's factorization is the same as the final column of M_R.Thus

$$M_R b_R + M_C b_L = M_R \begin{bmatrix} a_k h_{k-1} + b_k \\ b_{k+1} \\ \vdots \\ b_{\mathrm{T}-1} \end{bmatrix}$$

Finally, we use the observation that M_L and M_R are self-similar to the original matrix M; the answers for these two smaller 1-SS matrix multiplications can be performed arbitrarily using any algorithm. In total, the algorithm proceeds as follows:

1. Compute the left half of the answer $h_{0:k}$ using any desired method (i.e. any of the methods for 1-SS multiplication from this section).
2. Compute the final state h_{k-1}.
3. Increment the state by one step to modify b_k.
4. Compute the right half of the answer $h_{k:\mathrm{T}}$ using any desired method.

In other words, we compute the left subproblem as a black box, pass its final state on to the right problem, and compute the right subproblem as a black box.

The utility of this method comes from more complicated settings, such as in the general N-semiseparable case, and when the input b has an additional "batch" dimension (or in other words this is a matrix-matrix instead of matrix-vector multiplication). In this case, we can use an alternate algorithm for the chunks (corresponding to MM by M_L and M_R) that does not materialize the full hidden states h. Instead, we skip the hidden states and directly compute the final state h_{k-1} in an alternate way, then "pass" the state to the next chunk.

Complexity. This method can be very work-efficient because steps 2-3 takes only constant time. Therefore assuming the two subproblems (steps 1 and 4) are linear time, the whole method takes linear time.

The downside is that this is also sequential.

D.3.3 Fully Recurrent Mode

Note that the fully recurrent mode, where the recurrence is evolved one step at a time (15), is simply an instantiation of the state-passing mode with chunk size $k=1$.

D.3.4 (Parallel) Block Decomposition Mode

This uses the same matrix decomposition as the state-passing mode, but computes subproblems in a different order that trades off computation for parallelization.

As usual, we write M as

$$M = \begin{bmatrix} 1 & & & & \\ a_1 & 1 & & & \\ a_2a_1 & a_2 & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ a_{\mathtt{T}-1}...a_1 & a_{\mathtt{T}-1}...a_2 & ... & a_{\mathtt{T}-1} & 1 \end{bmatrix} = \begin{bmatrix} 1 & & & & \\ -a_1 & 1 & & & \\ 0 & -a_2 & 1 & & \\ \vdots & \vdots & \ddots & \ddots & \\ 0 & 0 & ... & -a_{\mathtt{T}-1} & 1 \end{bmatrix}^{-1}$$

The key observation is again that the bottom-left quadrant of M is rank-1. Aside from inspection, another way to see this is by using the RHS, observing that the bottom-left quadrant of it is a trivial rank-1 matrix (it is all 0 except the top-right corner is $-a_{\mathtt{T}/2}$), and using the Woodbury inversion formula to see that the bottom-left corner of the LHS must also be rank 1. This also provides a way to deduce the rank-1 factorization, which can be verified through inspection:

$$\begin{aligned} M_{\text{lower-left-quadrant}} &= \begin{bmatrix} (a_{\mathtt{T}/2}...a_1) & ... & a_{\mathtt{T}/2} \\ \vdots & \ddots & \vdots \\ (a_{\mathtt{T}-1}...a_{\mathtt{T}/2}a_{\mathtt{T}/2-1}...a_1) & ... & (a_{\mathtt{T}-1}...a_{\mathtt{T}/2}) \end{bmatrix} \\ &= \begin{bmatrix} a_{\mathtt{T}/2} \\ \vdots \\ a_{\mathtt{T}-1}...a_{\mathtt{T}/2} \end{bmatrix} \begin{bmatrix} (a_{\mathtt{T}/2-1}...a_1) & ... & a_{\mathtt{T}/2-1} & 1 \end{bmatrix}. \end{aligned}$$

A second observation is that *this matrix is self-similar*: any principle submatrix has the same form. In particular, the top-left and bottom-right quadrants are both 1-SS matrices.

This provides an easy way to perform the matrix multiplication by M: recurse on the two halves (i.e. top-left and bottom-right) in parallel, and then account for the bottom-left submatrix. This "combination" step in the divide-and-conquer algorithm is easy since the submatrix is rank 1. This leads to a parallel algorithm.

Complexity. Like the state-passing algorithm, this method uses the same block decompositions of the rank-structured semiseparable matrices. The difference is that we recurse on both subproblems in parallel, while the state-passing algorithm handles the left and then right subproblems. This lowers the depth/span of the algorithm from linear to $\log(\mathtt{T})$. The tradeoff is that the combination step (accounting for the rank-1 bottom-left submatrix) requires linear instead of constant work, so the total work is $O(\mathtt{T}\log\mathtt{T})$ instead of linear.

Note also that in the recursion, we can stop at any time and compute the subproblems in any other way. This is a main idea behind the SSD algorithm (Appendix C), where we switch to the dual *quadratic attention* formulation on small subproblems.

D.3.5 Associative Scan Mode

The state passing (chunkwise) algorithm has linear work, but also involves sequential operations.

The block matrix reduction and dilated modes are parallelizable: they have $\log \mathtt{T}$ depth/span. However, they do extra work ($O(\mathtt{T}\log\mathtt{T})$.

As noted in Appendix D.2.2, there is an algorithm that achieves both $O(\log\mathtt{T})$ depth and $O(\mathtt{T})$ work by leveraging the associative scan (also called prefix scan) algorithm (Baker et al., 1996). This algorithm is most easily seen from the SSM scan or `cumprodsum` view, and even then is not obvious: it requires separately deriving an associative operator (16), and then leveraging the parallel/associative/prefix scan algorithm as a black box (Blelloch, 1990).

Here we show that it is actually possible to derive this parallel scan from leveraging a different matrix decomposition:

$$M = \left[\begin{array}{cc|cc|cc|cc} a_{0:0} & & & & & & & \\ a_{1:0} & a_{1:1} & & & & & & \\ \hline a_{2:0} & a_{2:1} & a_{2:2} & & & & & \\ a_{3:0} & a_{3:1} & a_{3:2} & a_{3:3} & & & & \\ \hline a_{4:0} & a_{4:1} & a_{4:2} & a_{4:3} & a_{4:4} & & & \\ a_{5:0} & a_{5:1} & a_{5:2} & a_{5:3} & a_{5:4} & a_{5:5} & & \\ \hline a_{6:0} & a_{6:1} & a_{6:2} & a_{6:3} & a_{6:4} & a_{6:5} & a_{6:6} & \\ a_{7:0} & a_{7:1} & a_{7:2} & a_{7:3} & a_{7:4} & a_{7:5} & a_{7:6} & a_{7:7} \end{array}\right]$$

$$= \left[\begin{array}{cc|cc|cc|cc} a_{0:0} & & & & & & & \\ a_{1:0} & a_{1:1} & & & & & & \\ \hline \begin{bmatrix} a_{2:2} \\ a_{3:2} \end{bmatrix} a_{2:1} \begin{bmatrix} a_{1:0} \\ a_{1:1} \end{bmatrix}^\top & & a_{2:2} & & & & & \\ & & a_{3:2} & a_{3:3} & & & & \\ \hline \begin{bmatrix} a_{4:4} \\ a_{5:4} \end{bmatrix} a_{4:1} \begin{bmatrix} a_{1:0} \\ a_{1:1} \end{bmatrix}^\top & & \begin{bmatrix} a_{4:4} \\ a_{5:4} \end{bmatrix} a_{4:3} \begin{bmatrix} a_{3:2} \\ a_{3:3} \end{bmatrix}^\top & & a_{4:4} & & & \\ & & & & a_{5:4} & a_{5:5} & & \\ \hline \begin{bmatrix} a_{6:6} \\ a_{7:6} \end{bmatrix} a_{6:1} \begin{bmatrix} a_{1:0} \\ a_{1:1} \end{bmatrix}^\top & & \begin{bmatrix} a_{6:6} \\ a_{7:6} \end{bmatrix} a_{6:3} \begin{bmatrix} a_{3:2} \\ a_{3:3} \end{bmatrix}^\top & & \begin{bmatrix} a_{6:6} \\ a_{7:6} \end{bmatrix} a_{6:1} \begin{bmatrix} a_{5:4} \\ a_{5:5} \end{bmatrix}^\top & & a_{6:6} & \\ & & & & & & a_{7:6} & a_{7:7} \end{array}\right]$$

Now we proceed in three stages.

Stage 1. First we compute the answers for each of the diagonal blocks in the multiplication Mb. This produces two numbers, but the first element is unchanged. For example, the second block is going to compute b_2 and $a_3 b_2 + b_3$

Stage 2. Now consider each of the 2×2 blocks factored as a rank-1 matrix in the strictly lower triangular part of the matrix. Note that each of the right side row vectors is the same as the bottom row vector in the diagonal block in its column: in particular the $[a_{1:0}\, a_{1:1}]$, $[a_{3:2}\, a_{3:3}]$, and $[a_{5:4}\, a_{5:5}]$ rows.

Therefore we already have the answers to these from Stage 1, which is the second element of all $\mathtt{T}/2$ subproblems in Stage 1. If we call this array of elements b' (of half the size of b), then we need to multiply b' by the 1-SS matrix generated by $a_{3:-1}$,$a_{3:1}$,$a_{5:3}$,$a_{7:5}$.

Stage 3. Finally, each of the answers to Stage 2 can be broadcast into two final answers by multiplying by the left-side column vectors: in particular the $[a_{2:2}\, a_{3:2}]^\top$, $[a_{4:4}\, a_{5:4}]^\top$, and $[a_{6:6}\, a_{7:6}]^\top$ vectors.

Note that this can be slightly modified with some off-by-one shifting of the indices. An equivalent way to view this algorithm is as the three-step matrix factorization

$$M = \begin{bmatrix} a_{0:0} & & & & & & & \\ a_{1:0} & a_{1:1} & & & & & & \\ a_{2:0} & a_{2:1} & a_{2:2} & & & & & \\ a_{3:0} & a_{3:1} & a_{3:2} & a_{3:3} & & & & \\ a_{4:0} & a_{4:1} & a_{4:2} & a_{4:3} & a_{4:4} & & & \\ a_{5:0} & a_{5:1} & a_{5:2} & a_{5:3} & a_{5:4} & a_{5:5} & & \\ a_{6:0} & a_{6:1} & a_{6:2} & a_{6:3} & a_{6:4} & a_{6:5} & a_{6:6} & \\ a_{7:0} & a_{7:1} & a_{7:2} & a_{7:3} & a_{7:4} & a_{7:5} & a_{7:6} & a_{7:7} \end{bmatrix}$$

$$= \begin{bmatrix} a_{0:0} & & & & & & & \\ & a_{1:1} & & & & & & \\ & a_{2:1} & a_{2:2} & & & & & \\ & & & a_{3:3} & & & & \\ & & & a_{4:3} & a_{4:4} & & & \\ & & & & & a_{5:5} & & \\ & & & & & a_{6:5} & a_{6:6} & \\ & & & & & & & a_{7:7} \end{bmatrix} \begin{bmatrix} a_{0:0} & & & & & & & \\ & a_{1:1} & & & & & & \\ & & a_{2:2} & & & & & \\ & a_{3:1} & & a_{3:3} & & & & \\ & & & & a_{4:4} & & & \\ & a_{5:1} & & a_{5:3} & & a_{5:5} & & \\ & & & & & & a_{6:6} & \\ & a_{7:1} & & a_{7:3} & & a_{7:5} & & a_{7:7} \end{bmatrix} \begin{bmatrix} a_{0:0} & & & & & & & \\ a_{1:0} & a_{1:1} & & & & & & \\ & & a_{2:2} & & & & & \\ & & a_{3:2} & a_{3:3} & & & & \\ & & & & a_{4:4} & & & \\ & & & & a_{5:4} & a_{5:5} & & \\ & & & & & & a_{6:6} & \\ & & & & & & a_{7:6} & a_{7:7} \end{bmatrix}$$

Note that Stage 1 and Stage 3 require $O(\mathtt{T})$ work, while Stage 2 reduces to a self-similar problem of half the size. It is easy to check that this requires $O(\mathtt{T})$ total work and $O(\log \mathtt{T})$ depth/span.

Table 2: (**Scaling Law Model Sizes**.) Our model sizes and hyperparameters for scaling experiments. (Model dimension and number of heads applies only to Transformer models.)

PARAMS	n_layers	d_model	n_heads / d_head	TRAINING STEPS	LEARNING RATE	BATCH SIZE	TOKENS
125M	12	768	12 / 64	4800	6e-4	0.5M tokens	2.5B
350M	24	1024	16 / 64	13500	3e-4	0.5M tokens	7B
760M	24	1536	16 / 96	29000	2.5e-4	0.5M tokens	15B
1.3B	24	2048	32 / 64	50000	2e-4	0.5M tokens	26B

Remark 7. *In fact, it is possible to see that the computation graph of this algorithm is identical to that of the associative scan algorithm described in Appendix D.2.2. The key takeaway is that instead of the steps of (1) recognizing that M defines a recurrence (2) observing that the recurrence can be defined with an associative binary operator; there is a completely different perspective of simply finding a structured matrix decomposition algorithm for M.*

E Experimental Details

E.1 MQAR Details

We use a harder version of the task introduced in Based (Arora et al., 2024b) where tokens that are not query/key/values are replaced with random tokens. We also use more key-value pairs, longer sequences, and smaller model sizes than the usual variant of MQAR used by prior work; all of these changes make the task harder.

For each sequence length $\texttt{T} \in \{256, 512, 1024\}$, we use $\texttt{T}/4$ key-value pairs. The total vocab size is 8192.

We use a form of curriculum training where training cycles through datasets using $(\texttt{T}/32, \texttt{T}/16, \texttt{T}/8, \texttt{T}/4)$ key-value pairs, where each dataset has $2^{18} \approx 250000$ examples, for a total of 8 epochs through each dataset (total of $2^{28} \approx 270M$ examples). The total batch size is $2^{18} \approx 0.25M$ tokens (e.g. for $\texttt{T} = 1024$, the batch size is 256).

All methods use 2 layers with default settings; the attention baseline additionally receives positional embeddings. For each method, we sweep over model dimensions $\texttt{D} = \{32, 64, 128, 256\}$ and learning rates $\{10^{-3.5}, 10^{-2}, 10^{-2.5}\}$. We use a linear decay schedule that drops on every epoch (e.g. the last epoch would have a learning rate $1/8$ of the maximum/starting learning rate).

E.2 Scaling Law Details

All models were trained on the Pile. For the scaling law experiments, we use the GPT2 tokenizer.

Model Sizes. Table 2 specifies the model sizes we use for scaling laws following GPT3 (Brown et al., 2020), First, we changed the batch size of the 1.3B model from 1M tokens to 0.5M tokens for uniformity. Second, we changed the number of training steps and total tokens to roughly match Chinchilla scaling laws (Hoffmann et al., 2022), which specify that training tokens should increase proportionally to model size.

Training Recipes. All models used the AdamW optimizer with

- gradient clip value 1.0
- weight decay 0.1
- no dropout
- linear learning rate warmup with cosine decay

By default, the peak learning rate is the GPT3 specification.

Compared to GPT3 recipe, we use an "improved recipe", inspired by changes adopted by popular large language models such as PaLM (Chowdhery et al., 2023) and LLaMa (Touvron et al., 2023). These include:

- linear learning rate warmup with cosine decay to $1e-5$, with a peak value of $5\times$ the GPT3 value
- no linear bias terms
- RMSNorm instead of LayerNorm
- AdamW hyperparameter $\beta = (.9, .95)$ (the GPT3 value) instead of the PyTorch default of $\beta = (.9, .999)$

E.3 Downstream Evaluation Details

To evaluate downstream performance of fully trained, we train Mamba-2 on 300B tokens on the Pile, using the GPT-NeoX (Black et al., 2022) tokenizer.

We use the same hyperparameters as the scaling experiments, except with batch size 1M for the 1.3B and 2.7B model. For the

2.7B model, we also follow GPT3 specification (32 layers, dimension 2560).

For all models, we use 5x the learning rate of the corresponding GPT3 model.

For downstream evaluation, we use the LM evaluation harness from EleutherAI (Gao et al., 2021), on the same tasks as Mamba (Gu & Dao, 2023) with one additional one:

- LAMBADA (Paperno et al., 2016)
- HellaSwag (Zellers et al., 2019)
- PIQA (Bisk et al., 2020)
- ARC-challenge (Clark et al., 2018)
- ARC-easy: an easy subset of ARC-challenge
- WinoGrande (Sakaguchi et al., 2021)
- OpenBookQA (Mihaylov et al., 2018)

Table 3: (**Zero-shot Evaluations**.) Best results for each size in bold, second best unlined. We compare against open source LMs with various tokenizers, trained for up to 300B tokens. Pile refers to the validation split, comparing only against models trained on the same dataset and tokenizer (GPT-NeoX-20B). For each model size, Mamba-2 outperforms Mamba, and generally matches Pythia at twice the model size.

Model	Token.	Pile ppl ↓	LAMBADA ppl ↓	LAMBADA acc ↑	HellaSwag acc ↑	PIQA acc ↑	ARC-E acc ↑	ARC-C acc ↑	WinoGrande acc ↑	OpenBookQA acc ↑	Average acc ↑
Hybrid H3-130M	GPT2	—	89.48	25.8	31.7	64.2	44.4	**24.2**	50.6	27.0	38.2
Pythia-160M	NeoX	29.64	38.10	33.0	30.2	61.4	43.2	24.1	51.9	29.2	39.0
Mamba-130M	NeoX	10.56	**16.07**	**44.3**	35.2	64.5	**48.0**	**24.2**	51.9	28.8	42.4
Mamba-2-130M	NeoX	**10.48**	16.86	43.9	**35.3**	**64.9**	47.4	**24.2**	**52.1**	**30.6**	**42.6**
Hybrid H3-360M	GPT2	—	12.58	48.0	41.5	68.1	51.4	24.7	54.1	31.6	45.6
Pythia-410M	NeoX	9.95	10.84	51.4	40.6	66.9	52.1	24.6	53.8	30.0	45.6
Mamba-370M	NeoX	8.28	8.14	55.6	46.5	69.5	**55.1**	**28.0**	55.3	30.8	48.7
Mamba-2-370M	NeoX	**8.21**	**8.02**	**55.8**	**46.9**	**70.5**	54.9	26.9	**55.7**	**32.4**	**49.0**
Pythia-1B	NeoX	7.82	7.92	56.1	47.2	70.7	57.0	27.1	53.5	31.4	49.0
Mamba-790M	NeoX	7.33	6.02	**62.7**	**55.1**	**72.1**	**61.2**	**29.5**	56.1	34.2	53.0
Mamba-2-780M	NeoX	**7.26**	**5.86**	61.7	54.9	72.0	61.0	28.5	**60.2**	**36.2**	**53.5**
GPT-Neo 1.3B	GPT2	—	7.50	57.2	48.9	71.1	56.2	25.9	54.9	33.6	49.7
Hybrid H3-1.3B	GPT2	—	11.25	49.6	52.6	71.3	59.2	28.1	56.9	34.4	50.3
OPT-1.3B	OPT	—	6.64	58.0	53.7	72.4	56.7	29.6	59.5	33.2	51.9
Pythia-1.4B	NeoX	7.51	6.08	61.7	52.1	71.0	60.5	28.5	57.2	30.8	51.7
RWKV4-1.5B	NeoX	7.70	7.04	56.4	52.5	72.4	60.5	29.4	54.6	34.0	51.4
Mamba-1.4B	NeoX	6.80	5.04	65.0	59.1	**74.2**	**65.5**	32.8	**61.5**	36.4	**56.4**
Mamba-2-1.3B	NeoX	**6.66**	**5.02**	**65.7**	**59.9**	73.2	64.3	**33.3**	60.9	**37.8**	**56.4**
GPT-Neo 2.7B	GPT2	—	5.63	62.2	55.8	72.1	61.1	30.2	57.6	33.2	53.2
Hybrid H3-2.7B	GPT2	—	7.92	55.7	59.7	73.3	65.6	32.3	61.4	33.6	54.5
OPT-2.7B	OPT	—	5.12	63.6	60.6	74.8	60.8	31.3	61.0	35.2	55.3
Pythia-2.8B	NeoX	6.73	5.04	64.7	59.3	74.0	64.1	32.9	59.7	35.2	55.7
RWKV4-3B	NeoX	7.00	5.24	63.9	59.6	73.7	67.8	33.1	59.6	37.0	56.4
Mamba-2.8B	NeoX	6.22	4.23	69.2	66.1	75.2	**69.7**	36.3	63.5	**39.6**	59.9
Mamba-2-2.7B	NeoX	**6.09**	**4.10**	**69.7**	**66.6**	**76.4**	69.6	**36.4**	**64.0**	38.8	**60.2**
GPT-J-6B	GPT2	–	4.10	68.3	66.3	75.4	67.0	36.6	64.1	38.2	59.4
OPT-6.7B	OPT	–	4.25	67.7	67.2	76.3	65.6	34.9	65.5	37.4	59.2
Pythia-6.9B	NeoX	6.51	4.45	67.1	64.0	75.2	67.3	35.5	61.3	38.0	58.3
RWKV4-7.4B	NeoX	6.31	4.38	67.2	65.5	76.1	67.8	37.5	61.0	40.2	59.3