TILED FLASH LINEAR ATTENTION: MORE EFFICIENT LINEAR RNN AND XLSTM KERNELS

Maximilian Beck[1,2], **Korbinian Pöppel**[1,2], **Phillip Lippe**[2*], **Sepp Hochreiter**[1,2]
[1]ELLIS Unit Linz, Institute for Machine Learning, JKU Linz, Austria
[2]NXAI GmbH, Linz, Austria
{beck,poeppel,hochreit}@ml.jku.at

ABSTRACT

Linear RNNs with gating recently demonstrated competitive performance compared to Transformers in language modeling. Although their linear compute scaling in sequence length offers theoretical runtime advantages over Transformers, realizing these benefits in practice requires optimized custom kernels, as Transformers rely on the highly efficient FlashAttention kernels (Dao, 2024). Leveraging the chunkwise-parallel formulation of linear RNNs, FlashLinearAttention (FLA) (Yang & Zhang, 2024) shows that linear RNN kernels are faster than FlashAttention, by parallelizing over chunks of the input sequence. However, since the chunk size of FLA is limited, many intermediate states must be materialized in GPU memory. This causes high memory consumption and IO cost, especially for long-context pre-training. In this work, we present *Tiled Flash Linear Attention* (TFLA), a novel kernel algorithm for linear RNNs, that enables arbitrary large chunk sizes by introducing an additional level of sequence parallelization within each chunk. First, we apply TFLA to the xLSTM with matrix memory, the mLSTM (Beck et al., 2024). Second, we propose an mLSTM variant with sigmoid input gate and reduced computation for even faster kernel runtimes at equal language modeling performance. In our speed benchmarks, we show that our new mLSTM kernels based on TFLA outperform highly optimized FlashAttention, Linear Attention and Mamba kernels, setting a new state of the art for efficient long-context sequence modeling primitives.

1 INTRODUCTION

With the trend of training models of ever increasing size with large datasets on thousands of GPUs, it becomes increasingly important to optimize the model architecture as well as its low level implementations for modern hardware. Transformers (Vaswani et al., 2017), which are the core architecture of nowadays state-of-the-art models are highly optimized, but the computational requirements of self-attention scale quadratically with sequence length. This creates significant challenges for both training and inference on long context.

Recently, recurrent alternatives with linear scaling in sequence length (Beck et al., 2024; Sun et al., 2023; Dao & Gu, 2024; Yang et al., 2024b) promise efficiency gains, especially on long sequences and during inference while providing competitive performance. The success of these emerging recurrent architectures is based on two main pillars: (1) a parallel or chunkwise-parallel formulation, which is used in training mode when the full sequence is available beforehand instead of the recurrent formulation and (2) kernel implementations that are close to or exceed training speeds of FlashAttention (Dao, 2024).

Besides the standard recurrent execution, linear RNNs allow for a parallel formulation, which, like Attention, calculates all outputs in parallel. The parallel formulation leverages the insight from linear Attention (Katharopoulos et al., 2020), which showed that kernelized dot-product-based attention can be reinterpreted as a linear RNN with matrix-valued states. Due to the linear nature of the recurrence, the computation can be split into a recurrent part, which computes intermediate RNN states, and a parallel part, which fully utilizes the hardware for computing the outputs in between (Sun et al., 2023; Hua et al., 2022).

[*]Now at Google Deepmind

Yang et al. (2024b) show that their custom FlashLinearAttention (FLA) kernels based on the chunkwise-parallel formulation of linear RNNs provide faster runtimes than FlashAttention. This is achieved by first splitting the sequence into chunks and materializing the first RNN state of each chunk in GPU memory. Subsequently, in the parallel part they employ one level of sequence parallelism and compute the outputs for each chunk in parallel. For a small chunk size and long sequences, this leads to a large amount of intermediate states to be stored and loaded from GPU memory, which increases memory consumption and memory input/output (IO) cost. Since modern GPUs see a faster increase in computation throughput than memory bandwidth, it is essential to minimize large memory IO. A simple approach would be to increase the chunk size. However, the chunk size of FLA is limited by the physical SRAM available on the GPU.

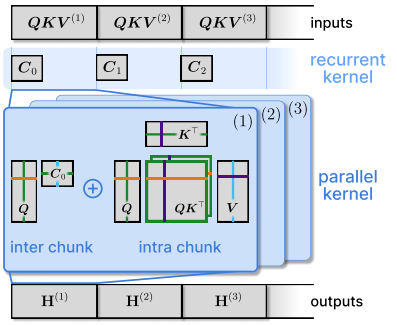


Figure 1: **Tiled Flash Linear Attention (TFLA)** consists of a recurrent kernel and a parallel kernel, which process the input sequence in chunks $\boldsymbol{QKV}^{(k)}$ (1^{st} level of sequence parallelism). The recurrent kernel materializes the memory state $\boldsymbol{C}_{k-1}$ for each chunk. The parallel kernel computes the output states $\mathbf{H}^{(k)}$ for all chunks. TFLA uses tiling for the 3 matrix-multiplications in the parallel kernel (2^{nd} level of sequence parallelism) to fully utilize the hardware and to prevent materialization of many memory states.

To solve this problem, we introduce *TiledFlashLinearAttention* (TFLA) which enables unlimited chunk sizes by introducing a second level of sequence parallelism via tiling matrix computations within each chunk. This enables fast kernels and allows us to efficiently balance memory consumption and IO vs. computation. In this paper, we implement our TiledFlashLinearAttention algorithm for the xLSTM with matrix memory – the mLSTM Beck et al. (2024). The mLSTM is a linear RNN that uses exponential gating with scalar gates per head, along with an additional normalizer state for output normalization. This gating mechanism has demonstrated competitive performance compared to Transformers and Mamba on language modeling tasks at moderate scales. However, for comparisons at even larger scales, efficient kernels that leverage the chunkwise-parallel formulation for the mLSTM were still missing. In our speed benchmarks, we show that our new mLSTM kernels based on TFLA outperform highly optimized Attention, Linear Attention and Mamba kernels.

After optimizing our kernels for the existing mLSTM computation, we seek ways to reduce kernel runtime by targeted modifications to the mLSTM. Towards this end, we propose *mLSTMsig*, an mLSTM with sigmoid input gate and reduced computation, that enables even faster kernel implementations at no performance drops on language modeling up to 1.4B parameter scale. Finally, motivated by the equal performance of both mLSTM variants, we perform an empirical study inspired by transfer function analysis from control theory (Ogata, 2010) to understand their differences and characteristics. We find that both mLSTM variants exhibit the same transfer behavior and, moreover, our analysis suggests that the input gate biases should be initialized at larger negative values. In extensive experiments on language modeling, we confirm that this initialization improves training stability as well as the overall performance of mLSTM models.

To summarize, in this work we make the following contributions: (1) We introduce *TiledFlashLinearAttention*, a new chunkwise-parallel kernel algorithm for Linear RNNs with two levels of sequence parallelism, that enables arbitrary large chunk sizes. (2) We introduce *mLSTMsig*, a faster mLSTM variant with sigmoid input gate with no performance losses up to 1.4B parameter scales. (3) We improve the training stability and performance of the mLSTM through careful gate initialization guided by our empirical transfer behavior analysis.

2 mLSTM Formulations

The mLSTM cell is the fully parallelizable part of the xLSTM (Beck et al., 2024). It has a matrix memory and exponential gating.

2.1 Recurrent Formulation

In its recurrent formulation, the mLSTM cell processes the series of input vectors $\boldsymbol{x}_t \in \mathbb{R}^d$ for time steps $t \in \{1, \dots, T\}$ mapping a state $(\boldsymbol{h}_{t-1}, \boldsymbol{C}_{t-1}, \boldsymbol{n}_{t-1}, m_{t-1})$ to a successor state $(\boldsymbol{h}_t, \boldsymbol{C}_t, \boldsymbol{n}_t, m_t)$ given an input $\boldsymbol{x}_t$. Here, $\boldsymbol{h}_t \in \mathbb{R}^{d_{hv}}$ denotes the hidden state, $\boldsymbol{C}_t \in \mathbb{R}^{d_{qk} \times d_{hv}}$ denotes the cell state responsible for long-term memory, $\boldsymbol{n}_t \in \mathbb{R}^{d_{qk}}$ denotes the normalizer state, and $m_t \in \mathbb{R}$ denotes the

max state. Together normalizer and max state control the magnitude of the exponential input gate and ensure stability (see Appendix D.1).

The recurrent mLSTM formulation is given by the following state update equations:

$$m_t = \max \left\{ \log \sigma(\tilde{\mathrm{f}}_t) + m_{t-1},\ \tilde{\mathrm{i}}_t \right\} \tag{1}$$

$$\boldsymbol{C}_t = \mathrm{f}_t\, \boldsymbol{C}_{t-1} + \mathrm{i}_t\, \boldsymbol{k}_t\, \boldsymbol{v}_t^\top \tag{2}$$

$$\boldsymbol{n}_t = \mathrm{f}_t\, \boldsymbol{n}_{t-1} + \mathrm{i}_t\, \boldsymbol{k}_t \tag{3}$$

$$\widetilde{\boldsymbol{h}}_t = \frac{\boldsymbol{C}_t^\top \left(\boldsymbol{q}_t / \sqrt{d_{qk}} \right)}{\max \left\{ \left| \boldsymbol{n}_t^\top \left(\boldsymbol{q}_t / \sqrt{d_{qk}} \right) \right|, \exp(-m_t) \right\}} \tag{4}$$

$$\boldsymbol{h}_t = \mathbf{o}_t \odot \mathrm{NORM}(\widetilde{\boldsymbol{h}}_t) \tag{5}$$

The scalar input and forget gate activations are computed as $\mathrm{f}_t = \exp(\log \sigma(\tilde{\mathrm{f}}_t) + m_{t-1} - m_t)$ and $\mathrm{i}_t = \exp(\tilde{\mathrm{i}}_t - m_t)$ with the pre-activations $\{\tilde{\mathrm{i}}_t, \tilde{\mathrm{f}}_t\} = \boldsymbol{w}_{\{\mathrm{i,f}\}}^\top \boldsymbol{x}_t + b_{\{\mathrm{i,f}\}}$ and the sigmoid function σ. The vector output gate $\mathbf{o}_t \in \mathbb{R}^{d_{hv}}$ is computed by $\mathbf{o}_t = \sigma(\tilde{\mathbf{o}}_t)$ with the pre-activations $\tilde{\mathbf{o}}_t = \boldsymbol{W}_{\mathbf{o}}\, \boldsymbol{x}_t + \boldsymbol{b}_{\mathbf{o}}$. The query, key, and value vectors $\boldsymbol{q}_t, \boldsymbol{k}_t \in \mathbb{R}^{d_{qk}}$, $\boldsymbol{v}_t \in \mathbb{R}^{d_{hv}}$ are computed as $\{\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t\} = \boldsymbol{W}_{\{q,k,v\}}\, \boldsymbol{x}_t + \boldsymbol{b}_{\{q,k,v\}}$. The norm layer NORM in (5) can be either RMS norm Zhang & Sennrich (2019) or LayerNorm (Ba et al., 2016). Typically, multiple of these cells operate simultaneously as parallel heads, similar to Transformers (Vaswani et al., 2017).

2.2 Chunkwise-Parallel Formulation

The chunkwise-parallel formulation is a trade-off between the parallel and the fully recurrent formulation. It has a recurrent part and a (quadratic) parallel part, with an overall sub-quadratic scaling in sequence length. Similar to the fully parallel formulation (see Appendix B.1), we assume that all inputs are available at once. We then split the sequence of length T into $N_c = \lceil T/L \rceil$ chunks of length L and use $k \in \{1, \ldots, N_c\}$ for the chunk index. We rearrange the input and forget gates, as well as the queries, keys, and values into chunkwise matrices, where the chunk index becomes the first dimension. For example, the forget gate pre-activations $\tilde{\mathbf{f}} \in \mathbb{R}^T$ are rearranged into a matrix $\tilde{\mathbf{f}} = (\tilde{\mathbf{f}}^{(1)}, \tilde{\mathbf{f}}^{(2)}, \ldots, \tilde{\mathbf{f}}^{(N_c)}) \in \mathbb{R}^{N_c \times L}$, where each row $\tilde{\mathbf{f}}^{(k)} = (\mathrm{f}_{(k-1)N_c+1}, \mathrm{f}_{(k-1)N_c+2}, \ldots, \mathrm{f}_{kN_c}) \in \mathbb{R}^L$ contains the pre-activations of the chunk k. The input gate pre-activations follow analogously. Similarly, the queries, keys and values are rearranged into chunkwise tensors $\boldsymbol{Q}, \boldsymbol{K} \in \mathbb{R}^{N_c \times L \times d_{qk}}$ and $\boldsymbol{V} \in \mathbb{R}^{N_c \times L \times d_{hv}}$. Here, the query matrix $\boldsymbol{Q}^{(k)} = (\boldsymbol{q}_{(k-1)N_c+1}, \ldots, \boldsymbol{q}_{kN_c}) \in \mathbb{R}^{L \times d_{qk}}$ contains the query vectors of chunk k. Keys, and values follow analogously. For notational simplicity we drop the leading N_c dimension and omit normalization layer and the output gate, i.e. consider $\widetilde{\boldsymbol{h}}_t$ as hidden state outputs.

Chunkwise Gates. Given the logarithmic forget gates $\bar{\mathbf{f}}^{(k)} = \log \sigma(\tilde{\mathbf{f}}^{(k)}) \in \mathbb{R}^L$ and input gates $\bar{\mathbf{i}}^{(k)} = \log \exp(\tilde{\mathbf{i}}^{(k)}) \in \mathbb{R}^L$, we can compute the chunkwise gates as

$$\mathrm{g}_k = \mathrm{sum}\left(\bar{\mathbf{f}}^{(k)}\right) \quad \in \mathbb{R}, \tag{6}$$

$$\mathbf{b}_k = \mathrm{cumsum}\left(\bar{\mathbf{f}}^{(k)}\right) \in \mathbb{R}^L, \tag{7}$$

$$\mathbf{a}_k = \mathrm{rev\_cumsum}\left(\bar{\mathbf{f}}^{(k)}\right) + \bar{\mathbf{i}}^{(k)} \in \mathbb{R}^L. \tag{8}$$

We refer to Appendix B.2 for more details on the chunkwise gate computation. The summed forget gates g_k contain the forget gate contribution of all forget gates within a chunk. The cumulative forget gate vectors $\mathbf{b}_k$ contain the forget gate contributions *from the beginning of the chunk up to the current time step* within the current chunk. The cumulative input gate vectors $\mathbf{a}_k$ contain the input gates for every timestep as well as the forget gate contributions *from the current time step to the end of the chunk*.

Inter-chunk Recurrent Contribution. The inter-chunk recurrence is given by

$$\boldsymbol{C}_k = \bar{\mathrm{g}}_k \boldsymbol{C}_{k-1} + \left(\bar{\mathbf{a}}_k \odot \boldsymbol{K}^{(k)}\right)^\top \boldsymbol{V}^{(k)} \tag{9}$$

$$\boldsymbol{n}_k = \bar{\mathrm{g}}_k \boldsymbol{n}_{k-1} + \left(\bar{\mathbf{a}}_k \odot \boldsymbol{K}^{(k)}\right)^\top \mathbf{1}, \tag{10}$$

where $\bar{\mathrm{g}}_k$ and $\bar{\mathbf{a}}_k$ are the stabilized chunkwise gates (see Appendix B.2 for details). This recurrent part resembles the fully recurrent formulation in Section 2.1, but instead of computing the intermediate states for every timestep t, we compute them directly for every L time steps without materializing the states in between.

Intra-chunk Parallel Contribution. The recurrent part is followed by the intra-chunk parallel contribution:

$$\widetilde{\mathbf{D}}^{(k)} = \begin{cases} -\infty & \text{for } i < j \\ \mathbf{b}_k - \mathbf{b}_k^\top + \bar{\mathbf{i}}^{(k)\top} & \text{for } i \geqslant j \end{cases} \tag{11}$$

$$\mathbf{S}^{(k)} = \frac{1}{\sqrt{d_{qk}}} \boldsymbol{Q}^{(k)} \boldsymbol{K}^{(k)\top} \tag{12}$$

$$\overline{\mathbf{S}}^{(k)} = \mathbf{S}^{(k)} \odot \mathbf{D}^{(k)}, \tag{13}$$

where $\mathbf{D}^{(k)} \in \mathbb{R}^{L \times L}$ is the stabilized gate matrix. Compared to the fully parallel part from Appendix B.1, the quadratic cost of the matrices $\mathbf{D}^{(k)}, \mathbf{S}^{(k)} \in \mathbb{R}^{L \times L}$ is greatly reduced, since the chunk size L is typically small compared to the sequence length T.

Output Computation. Finally, the contributions from the intra-chunk parallel part $\mathbf{H}_{\text{intra}}^{(k)}$ are combined with the inter-chunk recurrent part $\mathbf{H}_{\text{inter}}^{(k)}$ to obtain the hidden states $\mathbf{H}^{(k)} \in \mathbb{R}^{L \times d_{hv}}$ for each chunk k (see Figure 1):

$$\mathbf{H}_{\text{inter}}^{(k)} = \left(\bar{\mathbf{b}}_k \odot \frac{\boldsymbol{Q}^{(k)}}{\sqrt{d_{qk}}}\right) \boldsymbol{C}_{k-1} = \overline{\boldsymbol{Q}}^{(k)} \boldsymbol{C}_{k-1} \tag{14}$$

$$\mathbf{H}_{\text{intra}}^{(k)} = \overline{\mathbf{S}}^{(k)} \boldsymbol{V}^{(k)} \tag{15}$$

$$\mathbf{H}^{(k)} = \left(\mathbf{H}_{\text{inter}}^{(k)} + \mathbf{H}_{\text{intra}}^{(k)}\right) \,/\, \mathbf{h}_{\text{denom}}^{(k)}, \tag{16}$$

where $\mathbf{h}_{\text{denom}}^{(k)} \in \mathbb{R}^L$ is a normalization factor.

Appendix B.2 and B.3 provide a detailed description of the chunkwise-parallel forward and backward pass.

3 Tiled Flash Linear Attention

FlashLinearAttention (Yang et al., 2024b) introduces a fast kernel algorithm for the chunkwise formulation for Linear Attention (cf. Section 2.2 without gates) and shows that their implementation is faster than optimized FlashAttention Dao (2024). This speedup is achieved by single level sequence parallelism, where the states $\boldsymbol{C}_k$ are first materialized in GPU memory and then the outputs $\mathbf{H}^{(k)}$ are computed in parallel.

However, since FlashLinearAttention is limited in chunk size (typically $L = 64$), we have to materialize many states, where the number of states is $N_c = \lceil T/L \rceil$. This leads to high GPU memory consumption and a high memory IO cost, which poses challenges especially for long-context pre-training.

To address this issue, we introduce TiledFlashLinearAttention (TFLA), which adds a second level of sequence parallelism and enables arbitrary large chunk sizes and hence reduces GPU memory consumption. We review the fundamentals on writing efficient kernels in Appendix C.1. Since we perform our experiments on NVIDIA GPUs, our review is targeted towards NVIDIA's terminology, though the principles also apply to other hardware. For a more extensive overview we refer to (Spector et al., 2024).

TiledFlashLinearAttention (TFLA) enables fast kernels and a trade-off between memory consumption and computational efficiency by introducing two levels of sequence parallelism (see Figure 5). The first level is the parallelisation over the chunks of the sequence, which requires to compute and materialize intermediate states $\boldsymbol{C}_k$ in GPU HBM. For this we use a recurrent kernel similar to previous work (Yang et al., 2024b). The second level is the parallelisation within each chunk, which is achieved by tiling the intra chunk attention matrix along the chunk dimension. This second level of parallelism enables large chunk sizes and hence reduces the memory consumption for the intermediate states as we have to store and load $N_c = \lceil T/L \rceil$ intermediate states in HBM on each kernel call, where T is the sequence length and L is the chunk size. In addition to the two levels of sequence parallelisation and the naive parallelisation over the batch and head dimension, TFLA also parallelizes over the embedding dimension, resulting in a massive parallelisation over five dimensions, which is crucial for achieving high performance on modern GPUs.

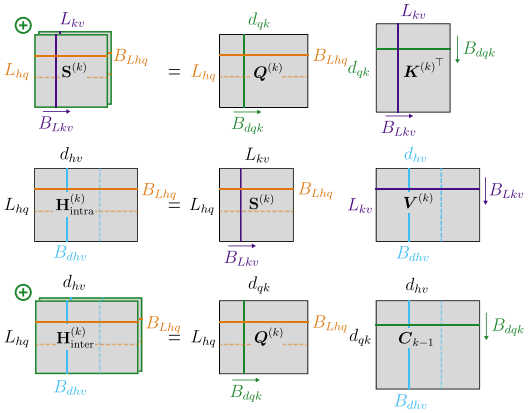


Figure 2: TFLA Forward Pass Tiling. We loop over B_{Lkv} and B_{dqk} (indicated by arrows) and parallelize over B_{Lhq} and B_{dhv} (indicated by dashed lines) blocks. $\bigoplus$ denotes block-wise accumulation.

Forward Pass. We review the matrix multiplication operations of the intra-chunk parallel part of the mLSTM in order to show how we efficiently parallelize these operations. For simplicity we omit the the gate computations and normalization, as these do not influence the work partitioning. We also omit the leading batch, head and chunk dimension, over which we can parallelize naively as they do not interact with the matrix multiplication (see Table 1).

In simplified form, the intra-chunk parallel forward pass of the mLSTM for a chunk k can be written as three matrix multiplications, which we fuse into a single kernel:

$$\underset{(L_{hq}\times d_{hv})}{\mathbf{H}^{(k)}} = \underbrace{\left(\underset{(L_{hq}\times d_{qk})}{\boldsymbol{Q}^{(k)}} \; \underset{(d_{qk}\times L_{kv})}{\boldsymbol{K}^{(k)\top}}\right) \underset{(L_{kv}\times d_{hv})}{\boldsymbol{V}^{(k)}}}_{\mathbf{H}^{(k)}_{\text{intra}}} + \underbrace{\underset{(L_{hq}\times d_{qk})}{\boldsymbol{Q}^{(k)}} \; \underset{(d_{qk}\times d_{hv})}{\boldsymbol{C}_{k-1}}}_{\mathbf{H}^{(k)}_{\text{inter}}} \tag{17}$$

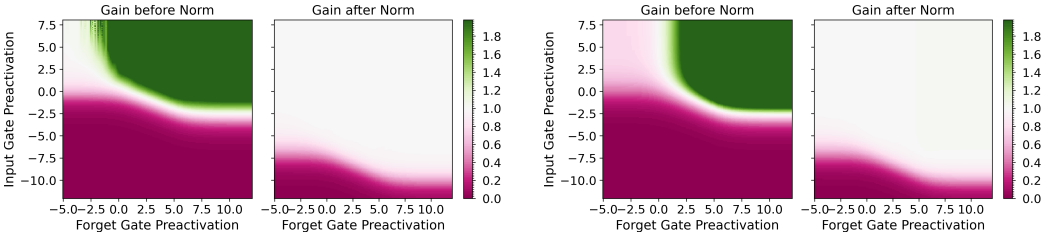
Figure 2 illustrates these matrix multiplications. In order to parallelize the matrix multiplications we introduce the block sizes B_{Lhq}, B_{Lkv}, B_{dqk} and B_{dhv} for the attention matrix, query, key, value and hidden state dimensions L_{hq}, L_{kv}, d_{qk} and d_{hv}, along which we either parallelize or accumulate over by using a loop inside the kernel.

For the forward pass $\mathbf{H}^{(k)}$ kernel we parallelize over the outer sequence dimension L_{hq} with $N_{Lhq} = L_{hq}/B_{Lhq}$ programs and the outer embedding dimension d_{hv} with $N_{dhv} = d_{hv}/B_{dhv}$ programs. We loop over the inner dimensions L_{kv} and d_{qk}, which are tiled by the block sizes B_{Lkv} and B_{dqk} respectively.

Tiled Computation. For the mLSTM we cannot simply accumulate the results of the matrix multiplications $\mathbf{H}^{(k)}_{\text{intra}}$ along the L_{kv} dimension and $\mathbf{H}^{(k)}_{\text{inter}}$ due to the stabilization of the exponential input gate with the max state m_t. The max state tracks the maximum of the forget and input gates over time and is used to stabilize the exponential input gate similar to the safe softmax computation (Milakov & Gimelshein, 2018). Since we compute the hidden state output $\mathbf{H}^{(k)}$ in blocks along the chunk size (i.e. time) dimension L_{kv}, we need to rescale during

Table 1: TFLA kernel parallelization and loop dimensions. Parallelization dimensions are indicated by P and loop dimensions by L. The last column shows the first two dimensions of the 3D kernel launch grid. The last dimension of all kernels is $N_{\text{chunk}} \cdot N_{\text{head}} \cdot N_{\text{batch}}$.

Kernel	L_{hq}	L_{kv}	d_{qk}	d_{hv}	Thread Block Grid
$\mathbf{H}^{(k)}$	P	L	L	P	$\left(\frac{d_{hv}}{B_{dhv}}, \frac{L_{hq}}{B_{Lhq}}, \dots\right)$
$\delta\boldsymbol{Q}^{(k)}$	P	L	P	L	$\left(\frac{d_{hv}}{B_{dhv}}, \frac{L_{hq}}{B_{Lhq}}, \dots\right)$
$\delta\boldsymbol{K}^{(k)}$	L	P	P	L	$\left(\frac{d_{hv}}{B_{dhv}}, \frac{L_{hq}}{B_{Lhq}}, \dots\right)$
$\delta\boldsymbol{V}^{(k)}$	L	P	L	P	$\left(\frac{d_{hv}}{B_{dhv}}, \frac{L_{hq}}{B_{Lhq}}, \dots\right)$



(a) mLSTMexp (Exponential Input Gate) (b) mLSTMsig (Sigmoid Input Gate)

Figure 3: Transfer behavior of the mLSTM before and after the RMS-norm layer (ϵ =1e-6) for different input and forget gate values. The color shows the gain of the mLSTM defined in (23). After the norm layer mLSTMexp and mLSTMsig exhibit the same transfer behavior.

accumulation of the block results for $\mathbf{H}^{(k)}_{\text{intra}}$ and the overall results into $\mathbf{H}^{(k)}$ in the same way as FlashAttention (Dao, 2024). We provide details on the rescaling in Section B.2. For the backward pass there is no rescaling necessary as we store the max states in the forward pass and reuse them in the backward pass.

The pseudocode for the forward pass of TFLA for the mLSTM is listed in Algorithm 1.

Backward Pass. The parallelization strategy for the backward pass of TFLA is more complex than the forward pass, since we need to compute three output tensors — the gradients for the queries, keys and values, of which each has an intra-chunk and inter-chunk part. However, in Section C.4 we show that the individual gradients can be mapped to three matrix multiplications similar to the forward pass. In TFLA, we then implement a separate kernel for each gradient and use the same work partitioning as in the forward pass but swap the loop and parallelization dimensions, accordingly. Table 1 summarizes the work partitioning of our TFLA kernels.

4 Faster mLSTM with Sigmoid Input Gate

The mLSTM with exponential gating (i.e. exponential input gate) introduced by Beck et al. (2024) requires to compute and keep track of two additional states, the normalizer state $\boldsymbol{n}_t$ and max state m_t, as we show in Appendix D.1.

Both will increase kernel runtime: The normalizer must be computed through summations, and tracking the max state throughout the tiled computation in TFLA (see Section 3 and C.2) prevents efficient fusing of loops within the kernel (see Appendix C.3).

Additionally, our analysis in Section 4.2 suggests to initialize the input gate biases at larger negative values (e.g. -10), such that the input gate pre-activations can grow slowly during training. We observe that most of these values stay below 0 during training (see Figure 11 in Appendix E). Therefore, we seek an alternative activation function which is similar to the exponential function in the negative range, but bounded in the positive range. This suggests to use the sigmoid function

$$\sigma(x) = \frac{1}{1+\exp(-x)} = \frac{\exp(x)}{\exp(x)+1}, \tag{18}$$

which converges to $\exp(x)$ for $x \to -\infty$ and 1 for $x \to \infty$.

4.1 mLSTM with Sigmoid Input Gate

The sigmoid function can be computed in two ways as given in equation 18. Depending on the sign of x it can be ensured that the argument of $\exp$ is always smaller than 0 to avoid numerical overflow. Therefore, we do not need to control the magnitude of x externally with a max state and as a consequence also drop the normalizer state (see Appendix D.1). This yields the mLSTM with sigmoid input gate (henceforth referred to as *mLSTMsig*) in its recurrent formulation as

$$\boldsymbol{C}_t = \sigma(\tilde{\mathrm{f}}_t)\, \boldsymbol{C}_{t-1} + \sigma(\tilde{\mathrm{i}}_t)\, \boldsymbol{k}_t\, \boldsymbol{v}_t^\top \tag{19}$$

$$\widetilde{\boldsymbol{h}}_t = \boldsymbol{C}_t^\top\, (\boldsymbol{q}_t / \sqrt{d_{qk}}) \tag{20}$$

$$\boldsymbol{h}_t = \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \mathrm{NORM}\left(\widetilde{\boldsymbol{h}}_t\right) \tag{21}$$

where the query, key, and value vectors $\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t$, and the gate preactivations $\tilde{\mathrm{i}}_t, \tilde{\mathrm{f}}_t, \tilde{\mathrm{o}}_t$ remain the same as for the mLSTM with exponential input gate (from now on referred to as *mLSTMexp*) in Section 2.1.

In Section 5.2, we confirm that our TFLA mLSTMsig forward kernel is over 30% faster than the mLSTMexp forward. We also show that mLSTMsig performs equally well compared to mLSTMexp in our language modeling experiments up to 1.4B parameters (see Section 5.1).

4.2 Normalization of mLSTM and Linear RNNs

Motivated by the performance of mLSTMsig, we seek to understand the differences between mLSTMsig and mLSTMexp empirically. To approach this, we draw inspiration from the concept of frequency response and transfer function analysis for control systems design, where typically the amplitude ratio or gain of output and input signals for different frequencies is considered (Ogata, 2010, Ch. 7). In our case, we analyze the transfer behavior of mLSTMsig and mLSTMexp for random inputs $\boldsymbol{q}_t, \boldsymbol{k}_t$ and $\boldsymbol{v}_t$ and different input gate and forget gate preactivation values $\tilde{\mathrm{i}}_t$ and $\tilde{\mathrm{f}}_t$.

We will see that the normalization layer $\boldsymbol{y} = \mathrm{NORM}(\boldsymbol{x})$, will play a crucial role in our analysis. The default norm layer in language modeling, the RMS norm (Zhang & Sennrich, 2019) with input vector input vector $\boldsymbol{x} \in \mathbb{R}^d$ and output vector $\boldsymbol{y} \in \mathbb{R}^d$ is defined as

$$\boldsymbol{y} = \frac{\boldsymbol{x}}{\mathrm{RMS}(\boldsymbol{x})} \odot \boldsymbol{\gamma}, \text{ where } \mathrm{RMS}(\boldsymbol{x}) = \sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}, \tag{22}$$

with with $\boldsymbol{\gamma} \in \mathbb{R}^d$ being a learnable scale parameter. The epsilon parameter $\epsilon \in \mathbb{R}$ is a small constant typically set to 1e-6 to avoid division by zero.

Transfer Behavior of the mLSTM. We analyze the transfer behavior by computing the gain of the mLSTM cells from random inputs sampled from $\mathcal{N}(0, 1)$ to hidden states before and after the norm layer for varying input and forget gate values. More specifically, we compute the gains G_{before} and G_{after} as

$$G_{\text{before}} = \frac{\|\widetilde{\boldsymbol{h}}_t\|_{\max}}{\|\boldsymbol{v}_t\|_{\max}} \quad \text{and} \quad G_{\text{after}} = \frac{\|\mathrm{NORM}(\widetilde{\boldsymbol{h}}_t)\|_{\max}}{\|\boldsymbol{v}_t\|_{\max}}, \tag{23}$$

where $\|\boldsymbol{x}\|_{\max} := \max(|\boldsymbol{x}_1|, \ldots, |\boldsymbol{x}_d|)$ and we average over the time dimension. For more details see App. D.2.

In Figure 3 we observe that the transfer behavior of mLSTMsig without normalizer is identical to mLSTMexp with normalizer and max state. Both exhibit a transition from suppressing ($G = 0$) to passing ($G = 1$) the signal at larger negative input gate preactivation values, which could partly explain the matching performance in our language modeling experiments.

Relation to other Gated Linear RNNs. Interestingly, almost all other gated linear RNN variants also place a normalization layer after the RNN cell Dao & Gu (2024); Sun et al. (2023); Qin et al. (2024b); Yang et al. (2024b). Often this is justified with improved training stability, but a more thorough discussion is missing (Lieber et al., 2024). Qin et al. (2022) analyze the effect of the norm layer after a non-gated, kernel-based linear attention (Katharopoulos et al., 2020) layer and show that this effectively prevents unbounded gradients. We also confirm that the norm layer has a significant impact on training stability and the gradient norm during training. In Section 5.1 we show that initializing the input gate bias at larger negative values, as suggested by our transfer behavior analysis in Figure 3, prevents large gradient norm variance and spikes during training.

Effect of Normalization on Gating in Linear RNNs. We hypothesize that at this point the normalization layer does not only have a stabilizing effect by controlling the magnitude of the layer activations through rescaling, but also actively participates in the information routing or gating mechanism of the linear RNN. For example, if the squared norm of $\boldsymbol{C}_t^\top \boldsymbol{q}$, which is controlled by input and forget gates through $\boldsymbol{C}_t^\top$, is smaller than the epsilon, the denominator in the $\mathrm{NORM}(\boldsymbol{x})$ layer is dominated by ϵ and the output moves towards zero (indicated by the purple area in Fig. 3). Hence, by moving through the x-y plane in Fig. 3, the gates could learn to suppress or amplify any input in the sequence.

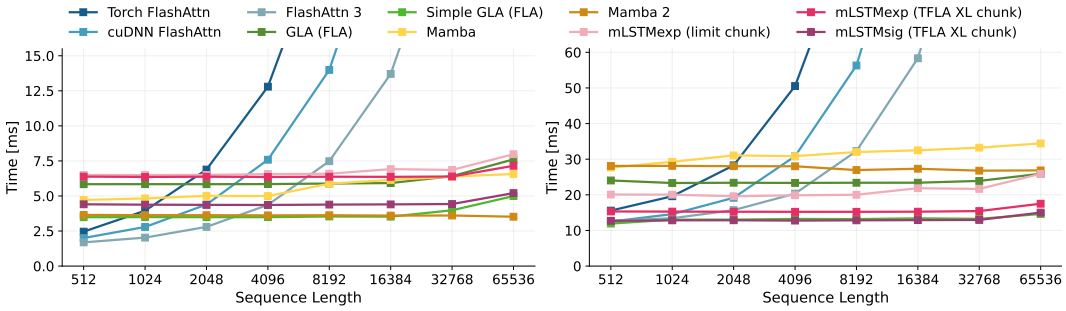


Figure 4: TFLA Kernel Runtime Benchmark for embedding dimension 4096 and 65,536 tokens on NVIDIA H100 GPUs. **Left:** Forward pass. **Right:** Forward and Backward pass. In training our TFLA kernels are faster than FlashAttention 3 for longer sequence lengths and more than two times faster than Mamba 2 kernels for all sequence lengths.

In Section D.2 we show additional experiments on the effect of varying the normalization layer epsilons and different modifications of the normalizers for the mLSTM.

5 Experiments

In this section we examine the performance of the two mLSTM variants mLSTMexp (mLSTM with exponential input gate) and mLSTMsig (mLSTM with sigmoid input gate). We compare two kernel algorithms: (1) `limit_chunk`: A kernel that is limited in chunk size L. (2) `xl_chunk`: Our TiledFlashLinearAttention (TFLA) kernels with unlimited chunk size. For details see Section 3.

We assess the performance of mLSTMsig compared to mLSTMexp in Section 5.1 and benchmark the runtime of our kernels against other baselines in Section 5.2. In App. E.1 we verify the numerical correctness of our kernels.

5.1 Language Modeling with mLSTM

We train three different model sizes (160M, 400M, 1.4B parameters) with context lengths 4096 and 8192 on the DCLM dataset (Li et al., 2024). We include Llama2 style Transformer models (Touvron et al., 2023b) as reference in our comparison and describe our experiment setup, model architecture and training recipe in Appendix E.2.

Performance in Language Modeling. We compare mLSTMsig and mLSTMexp models on next-token prediction with different number of heads or head dimensions. Table 2 and Table 5 show the results for context length 4096 and 8192, respectively. We find that our `limit_chunk` and `xl_chunk` kernels yield the same loss (up to small numerical deviations) for almost all head dimensions. For some head dimensions, we observe gradient norm or loss spikes for the `xl_chunk` kernels, which affect the final loss. As a main result we find that mLSTMsig performs equally well compared to mLSTMexp.

Table 2: Validation Perplexity at context length 4096. EXP and SIG denote mLSTMexp and mLSTMsig. LIMIT and XL correspond to `limit_chunk` and `xl_chunk` kernels.

Size	Tokens	Heads	Llama	EXP LIMIT	EXP XL	SIG XL
160M	19B	6		21.03	21.18	21.03
		12	20.89	21.03	21.06	21.05
400M	24B	4		16.66	16.66	16.67
		8		16.55	16.80	16.67
		16	16.85	16.60	16.61	16.61
1.4B	33B	4		13.31	13.35	13.34
		8		13.20	13.22	13.21
		16	13.64	13.20	13.87*	13.22

Effect of Input Gate Bias Initialization. We analyze the effect of the input gate bias initialization on training stability and performance of our mLSTM models in Appendix E.2. We observe in Figure 8 and 9, that initializing the input gate biases to -10 effectively mitigates large gradient norm spikes and variance during training for both mLSTMexp and mLSTMsig. We therefore conclude that the additional input gate not only improves performance (see Table 6), but also improves training stability, if initialized correctly.

Effect of Norm Layer Epsilon. In Appendix E.2 we investigate the effect of the norm layer epsilon on language modeling performance for mLSTMexp. Our transfer behavior analysis in Figure 3

suggests, that there exists an interplay between norm layer epsilon and input gate bias initialization. We confirm this in our grid search in Figure 10 and find that the best performing configuration is the default epsilon ϵ =1e-6 with input gate biases initialized to -10.

5.2 Kernel Benchmark

Finally, we compare the runtime of our mLSTM `limit_chunk` and TFLA `xl_chunk` kernels with kernel implementations of the state-of-the-art sequence modeling primitives FlashAttention (Dao, 2024; Shah et al., 2024), Mamba (Gu & Dao, 2024; Dao & Gu, 2024) and GLA Yang et al. (2024b). In Appendix E.3 we compare with other kernels from the FlashLinearAttention library (Yang & Zhang, 2024). We run our benchmarks on NVIDIA H100 GPUs.

Runtime Benchmark. We use the standard embedding dimension of 4096 for 7B parameter models and adapt the head dimensions per kernel accordingly. For example for FlashAttention we use 32 heads with head dim 128 and for the mLSTM we use 16 heads with head dim 256. Following the practice of Shah et al. (2024), we keep the number of tokens constant at 65,536 and vary sequence length and batch size accordingly. For further details see Appendix E.3.

Figure 4 shows the runtime benchmark results for forward pass only (left) and forward-backward pass (right). Our mLSTMexp TFLA `xl_chunk` kernels with two level sequence parallelism is about 25% faster than our `limit_chunk` kernels. Through targeted modifications of the input gate of the mLSTM we save computation and enable more efficient kernel implementations for the forward pass of mLSTMsig (see Sec. 4). This yields another speedup of over 30% for the forward pass of the mLSTMsig TFLA kernel over the mLSTMexp TFLA kernel. We perform additional runtime benchmarks for varying head dimensions and a more in-depth comparison to the FLA (Yang et al., 2024b) and LightningAttention2 (Qin et al., 2024a) kernels in Appendix E.3.

Runtime vs. Memory Trade-off. The chunk size parameter L balances the computation between the two levels of sequence parallelism (see Sec. 3). Smaller chunk sizes increase memory consumption because more chunks are materialized in memory, but they reduce the quadratic compute FLOPs in the parallel part. Larger chunk sizes have the opposite effect. They decrease memory consumption but increase quadratic compute FLOPs. In Figure 5 we measure this trade-off for our mLSTMsig TFLA `xl_chunk` kernels. By varying the chunk size parameter, our TFLA kernels can effectively balance memory vs. runtime.

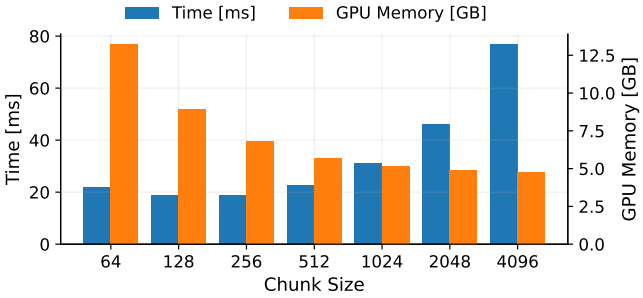


Figure 5: Memory vs. Runtime Trade-off of TFLA Forward-Backward Pass. We show the mLSTMsig for embedding dimension 4096 (8 heads with head dim 512), sequence length 8192 and batch size 4.

6 Related Work

TFLA builds on ideas from FlashAttention (Dao, 2024) and FlashLinearAttention (Yang et al., 2024b) and is designed for efficient mLSTM kernels (Beck et al., 2024), while being applicable also to other linear RNNs (Sun et al., 2023; Dao & Gu, 2024). We discuss this and other related works in Appendix A.

7 Conclusion and Future Work

With TiledFlashLinearAttention (TFLA) we introduce an algorithm for Linear RNN and mLSTM kernels with two levels of sequence parallelism. Our TFLA kernels for the mLSTM with exponential input gate (mLSTMexp) achieve state-of-the art kernel execution speeds, while remaining flexible to trade off GPU memory consumption and runtime. To further improve kernel runtimes, we propose mLSTMsig, a mLSTM variant with sigmoid input gate, that reduces computation and increases speed. Our experiments show that both mLSTM variants perform equally well on language modeling.

Although we enhance training stability through careful gate initialization informed by our empirical transfer behavior analysis, future work could explore instabilities arising from numerical errors in

kernel implementations in greater depth. Finally, the programming techniques and hardware features used to optimize FlashAttention (Shah et al., 2024) could also be leveraged by our TFLA algorithm. This makes us believe that TFLA has the potential to become a foundational primitive for future long-context language models.

REFERENCES

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

Maximilian Beck, Korbinian Pöppel, Markus Spanring, Andreas Auer, Oleksandra Prudnikova, Michael Kopp, Günter Klambauer, Johannes Brandstetter, and Sepp Hochreiter. xLSTM: Extended long short-term memory. In *Thirty-eighth Conference on Neural Information Processing Systems*, 2024.

Sid Black, Stella Biderman, Eric Hallahan, Quentin Anthony, Leo Gao, Laurence Golding, Horace He, Connor Leahy, Kyle McDonell, Jason Phang, Michael Pieler, USVSN Sai Prashanth, Shivanshu Purohit, Laria Reynolds, Jonathan Tow, Ben Wang, and Samuel Weinbach. GPT-NeoX-20B: An open-source autoregressive language model. In *Proceedings of the ACL Workshop on Challenges & Perspectives in Creating Large Language Models*, 2022.

James Bradbury, Roy Frostig, Peter Hawkins, Matthew James Johnson, Chris Leary, Dougal Maclaurin, George Necula, Adam Paszke, Jake VanderPlas, Skye Wanderman-Milne, and Qiao Zhang. JAX: composable transformations of Python+NumPy programs, 2018. URL http://github.com/jax-ml/jax.

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

T. Dao. FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning. In *The Twelfth International Conference on Learning Representations*, 2024.

T. Dao, D. Y. Fu, S. Ermon, A. Rudra, and C. Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Tri Dao and Albert Gu. Transformers are SSMs: Generalized models and efficient algorithms through structured state space duality. In *Forty-first International Conference on Machine Learning*, 2024.

Daniel Y. Fu, Hermann Kumbong, Eric Nguyen, and Christopher Ré. FlashFFTConv: Efficient convolutions for long sequences with tensor cores. In *International Conference on Learning Representations*, 2024.

A. Gu and T. Dao. Mamba: Linear-time sequence modeling with selective state spaces. In *International Conference on Learning Representations*, 2024.

Jonathan Heek, Anselm Levskaya, Avital Oliver, Marvin Ritter, Bertrand Rondepierre, Andreas Steiner, and Marc van Zee. Flax: A neural network library and ecosystem for JAX, 2024. URL http://github.com/google/flax.

Weizhe Hua, Zihang Dai, Hanxiao Liu, and Quoc Le. Transformer quality in linear time. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvari, Gang Niu, and Sivan Sabato (eds.), *Proceedings of the 39th International Conference on Machine Learning*, volume 162 of *Proceedings of Machine Learning Research*, pp. 9099–9117. PMLR, 17–23 Jul 2022.

A. Katharopoulos, A. Vyas, N. Pappas, and F. Fleuret. Transformers are RNNs: Fast autoregressive transformers with linear attention. In *Proceedings of the International Conference on Machine Learning*, 2020.

T. Katsch. GateLoop: Fully data-controlled linear recurrence for sequence modeling. *ArXiv*, 2311.01927, 2023.

Jeffrey Li, Alex Fang, Georgios Smyrnis, Maor Ivgi, Matt Jordan, Samir Gadre, Hritik Bansal, Etash Guha, Sedrick Keh, Kushal Arora, Saurabh Garg, Rui Xin, Niklas Muennighoff, Reinhard Heckel, Jean Mercat, Mayee Chen, Suchin Gururangan, Mitchell Wortsman, Alon Albalak, Yonatan Bitton, Marianna Nezhurina, Amro Abbas, Cheng-Yu Hsieh, Dhruba Ghosh, Josh Gardner, Maciej Kilian, Hanlin Zhang, Rulin Shao, Sarah Pratt, Sunny Sanyal, Gabriel Ilharco, Giannis Daras, Kalyani Marathe, Aaron Gokaslan, Jieyu Zhang, Khyathi Chandu, Thao Nguyen, Igor Vasiljevic, Sham Kakade, Shuran Song, Sujay Sanghavi, Fartash Faghri, Sewoong Oh, Luke Zettlemoyer, Kyle Lo, Alaaeldin El-Nouby, Hadi Pouransari, Alexander Toshev, Stephanie Wang, Dirk Groeneveld, Luca Soldaini, Pang Wei Koh, Jenia Jitsev, Thomas Kollar, Alexandros G. Dimakis, Yair Carmon, Achal Dave, Ludwig Schmidt, and Vaishaal Shankar. Datacomp-lm: In search of the next generation of training sets for language models. *arXiv*, 2406.11794, 2024.

Opher Lieber, Barak Lenz, Hofit Bata, Gal Cohen, Jhonathan Osin, Itay Dalmedigos, Erez Safahi, Shaked Meirom, Yonatan Belinkov, Shai Shalev-Shwartz, Omri Abend, Raz Alon, Tomer Asida, Amir Bergman, Roman Glozman, Michael Gokhman, Avashalom Manevich, Nir Ratner, Noam Rozen, Erez Shwartz, Mor Zusman, and Yoav Shoham. Jamba: A hybrid transformer-mamba language model. *arXiv*, 2403.19887, 2024.

Ilya Loshchilov and Frank Hutter. Decoupled weight decay regularization. In *International Conference on Learning Representations*, 2019.

M. Milakov and N. Gimelshein. Online normalizer calculation for softmax. *ArXiv*, 1805.02867, 2018.

MiniMax, Aonian Li, Bangwei Gong, Bo Yang, Boji Shan, Chang Liu, Cheng Zhu, Chunhao Zhang, Congchao Guo, Da Chen, Dong Li, Enwei Jiao, Gengxin Li, Guojun Zhang, Haohai Sun, Houze Dong, Jiadai Zhu, Jiaqi Zhuang, Jiayuan Song, Jin Zhu, Jingtao Han, Jingyang Li, Junbin Xie, Junhao Xu, Junjie Yan, Kaishun Zhang, Kecheng Xiao, Kexi Kang, Le Han, Leyang Wang, Lianfei Yu, Liheng Feng, Lin Zheng, Linbo Chai, Long Xing, Meizhi Ju, Mingyuan Chi, Mozhi Zhang, Peikai Huang, Pengcheng Niu, Pengfei Li, Pengyu Zhao, Qi Yang, Qidi Xu, Qiexiang Wang, Qin Wang, Qiuhui Li, Ruitao Leng, Shengmin Shi, Shuqi Yu, Sichen Li, Songquan Zhu, Tao Huang, Tianrun Liang, Weigao Sun, Weixuan Sun, Weiyu Cheng, Wenkai Li, Xiangjun Song, Xiao Su, Xiaodong Han, Xinjie Zhang, Xinzhu Hou, Xu Min, Xun Zou, Xuyang Shen, Yan Gong, Yingjie Zhu, Yipeng Zhou, Yiran Zhong, Yongyi Hu, Yuanxiang Fan, Yue Yu, Yufeng Yang, Yuhao Li, Yunan Huang, Yunji Li, Yunpeng Huang, Yunzhi Xu, Yuxin Mao, Zehan Li, Zekang Li, Zewei Tao, Zewen Ying, Zhaoyang Cong, Zhen Qin, Zhenhua Fan, Zhihang Yu, Zhuo Jiang, and Zijia Wu. MiniMax-01: Scaling foundation models with lightning attention. *arXiv*, 2501.08313, 2025.

Katsuhiko Ogata. *Modern control engineering*. Prentice-Hall electrical engineering series. Instrumentation and controls series. Prentice-Hall, Boston, 5th ed edition, 2010. ISBN 978-0-13-615673-4.

Antonio Orvieto, Samuel L Smith, Albert Gu, Anushan Fernando, Caglar Gulcehre, Razvan Pascanu, and Soham De. Resurrecting recurrent neural networks for long sequences. *arXiv*, 2303.06349, 2023.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Yang, Zach DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. *arXiv*, 1912.01703, 2019.

Korbinian Pöppel, Maximilian Beck, and Sepp Hochreiter. FlashRNN: I/O-aware optimization of traditional RNNs on modern hardware. In *The Thirteenth International Conference on Learning Representations*, 2025.

Zhen Qin, Xiaodong Han, Weixuan Sun, Dongxu Li, Lingpeng Kong, Nick Barnes, and Yiran Zhong. The devil in linear transformer. In Yoav Goldberg, Zornitsa Kozareva, and Yue Zhang (eds.), *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pp. 7025–7041, Abu Dhabi, United Arab Emirates, December 2022. Association for Computational Linguistics.

Zhen Qin, Songlin Yang, and Yiran Zhong. Hierarchically gated recurrent neural network for sequence modeling. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

Zhen Qin, Weigao Sun, Dong Li, Xuyang Shen, Weixuan Sun, and Yiran Zhong. Lightning attention-2: A free lunch for handling unlimited sequence lengths in large language models. *arXiv*, 2401.04658, 2024a.

Zhen Qin, Songlin Yang, Weixuan Sun, Xuyang Shen, Dong Li, Weigao Sun, and Yiran Zhong. HGRN2: Gated linear RNNs with state expansion. In *First Conference on Language Modeling*, 2024b.

Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI*, 2019.

Imanol Schlag, Kazuki Irie, and Jürgen Schmidhuber. Linear transformers are secretly fast weight programmers. In Marina Meila and Tong Zhang (eds.), *Proceedings of the 38th International Conference on Machine Learning*, volume 139 of *Proceedings of Machine Learning Research*, pp. 9355–9366. PMLR, 18–24 Jul 2021.

J. Shah, G. Bikshandi, Y. Zhang, V. Thakkar, P. Ramani, and T. Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *arXiv*, 2407.08608, 2024.

Noam Shazeer. GLU variants improve transformer. *arXiv*, 2002.05202, 2020.

Benjamin F. Spector, Simran Arora, Aaryan Singhal, Daniel Y. Fu, and Christopher Ré. ThunderKittens: Simple, fast, and adorable ai kernels. *arXiv*, 2410.20399, 2024.

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

Gemma Team. Gemma 2: Improving open language models at a practical size. *arXiv*, 2408.00118, 2024.

Philippe Tillet. Triton, 2024. URL https://github.com/triton-lang/triton.

Philippe Tillet, H. T. Kung, and David Cox. Triton: an intermediate language and compiler for tiled neural network computations. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2019, pp. 10–19, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367196. doi: 10.1145/3315508.3329973.

Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, Aurelien Rodriguez, Armand Joulin, Edouard Grave, and Guillaume Lample. Llama: Open and efficient foundation language models. *arXiv*, 2302.13971, 2023a.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. Llama 2: Open foundation and fine-tuned chat models. *arXiv*, 2307.09288, 2023b.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Ł ukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017.

Sharad Vikram, Chris Jones, and Sergei Lebedev. Jax-triton, 2022. URL https://github.com/jax-ml/jax-triton.

Roger Waleffe, Wonmin Byeon, Duncan Riach, Brandon Norick, Vijay Korthikanti, Tri Dao, Albert Gu, Ali Hatamizadeh, Sudhakar Singh, Deepak Narayanan, Garvit Kulshreshtha, Vartika Singh, Jared Casper, Jan Kautz, Mohammad Shoeybi, and Bryan Catanzaro. An empirical study of mamba-based language models. *arXiv*, 2406.07887, 2024.

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

Songlin Yang, Jan Kautz, and Ali Hatamizadeh. Gated Delta Networks: Improving Mamba2 with delta rule. *arXiv*, 2412.06464, 2024a.

Songlin Yang, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. Gated linear attention transformers with hardware-efficient training. In *Forty-first International Conference on Machine Learning*, 2024b.

Songlin Yang, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. Parallelizing linear transformers with the delta rule over sequence length. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024c.

Biao Zhang and Rico Sennrich. Root mean square layer normalization. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (eds.), *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

Appendix

TABLE OF CONTENTS

A Related Work

A.1 Relation to Flash Attention and Flash Linear Attention

Tiled Flash Linear Attention (TFLA) combines the idea of tiling one sequence dimension the attention matrix for better work partitioning (Dao, 2024) with the idea of dividing the sequence into chunks (Yang et al., 2024b). These two ideas yield the two levels of sequence parallelism for TFLA.

FlashAttention. FlashAttention (Dao et al., 2022) is an IO-aware implementation of softmax attention introduced by (Vaswani et al., 2017). It uses the idea of tiling to reduce the number of memory reads/writes between GPU high bandwidth memory (HBM) and GPU on-chip SRAM. In this way the quadratic attention matrix $\boldsymbol{QK}^\top$ is never materialized in HBM, which reduces the memory requirement from quadratic with sequence length to linear, and significantly speeds up the kernel due to reduced memory IO cost. However, the computation still remains quadratic with sequence length. FlashAttention 2 (Dao, 2024) improves the work partitioning by parallelizing the attention computation over the sequence dimension in addition to the naive parallelization over batch and head dimension. FlashAttention 3 (Shah et al., 2024) leverages new hardware features of recent GPU generations (e.g. NVIDIA Hopper GPUs) such as FP8 precision or exploiting asynchrony of Tensor cores and Tensor Memory Accelerators (TMA) to speed up FlashAttention.

TFLA is also IO-aware and parallelizes over one sequence dimension of the intra-chunk $\boldsymbol{QK}^\top$ matrix as the second level of sequence parallelism. New hardware features will also speed up future TFLA implementations.

FlashLinearAttention. FlashLinearAttention (FLA) (Yang et al., 2024b; Yang & Zhang, 2024) makes use of the fact that linear attention can be interpreted as linear RNN (Katharopoulos et al., 2020). It then leverages the chunkwise-parallel formulation of linear RNNs (Hua et al., 2022; Sun et al., 2023) for efficient kernel implementations, that process the sequence in chunks. More specifically, Yang et al. (2024b) propose two FLA variants: A version that materializes intermediate states in HBM and a non-materialization version. The materialization version consists of two kernels: The first is a recurrent kernel that materializes the first intermediate states of every chunk. The second kernel then processes all chunks in parallel and computes the outputs within the chunks. The non-materialization version was proposed concurrently by Qin et al. (2024a) and does not employ parallelism over the sequence dimension, but processes the inputs sequentially in chunks.

TFLA uses the idea of chunking of the sequence for the first level of sequence parallelism.

A.2 Other Related Work

Other Hardware-Aware Optimizations. Optimized, hardware-aware implementations enable the exploration of new primitives or new model architectures. FlashRNN Pöppel et al. (2025) introduces a framework of IO-aware optimized CUDA kernels in order to simplify research on traditional, non-parallelizable RNNs. Mamba (Gu & Dao, 2024) enables large scale language modeling experiments (Waleffe et al., 2024) with an efficient parallel scan algorithm in their optimized CUDA kernels. FlashFFTConv (Fu et al., 2024) provides efficient implementations for FFT convolutions for modern hardware by reducing IO and leveraging specialized matrix multiply units. DeltaNet Yang et al. (2024c;a) introduces an efficient algorithm for training linear Transformers with the delta rule (DeltaNet) (Schlag et al., 2021), which enables to scale up DeltaNet to standard language modeling settings.

Our TFLA kernel algorithm provides an effective method to balance the runtime and memory for linear RNN kernels based on their chunkwise-parallel formulation, paving the way to even larger model training setups.

Gating mechanisms for Linear RNNs. Many different gating techniques for linear RNNs have been explored (Sun et al., 2023; Beck et al., 2024; Yang et al., 2024b; Gu & Dao, 2024; Dao & Gu, 2024; Qin et al., 2023; 2024b; Orvieto et al., 2023; Katsch, 2023). We propose mLSTMsig, a mLSTM variant with sigmoid input gate and analyze the transfer behavior, empirically.

B EXTENDED MLSTM FORMULATIONS

B.1 FULLY PARALLEL FORMULATION

For the parallel formulation it is assumed that all inputs are available at once. Then, the queries, keys and values $\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t$ can be stacked into the matrices $\boldsymbol{Q}, \boldsymbol{K} \in \mathbb{R}^{T \times d_{qk}}, \boldsymbol{V} \in \mathbb{R}^{T \times d_{hv}}$ in order to compute all hidden states $\mathbf{H} \in \mathbb{R}^{T \times d_{hv}}$ in parallel using the following equations:

$$\widetilde{\mathbf{D}} = \log \mathbf{F} + \tilde{\mathbf{I}} \tag{24}$$

$$\boldsymbol{m} = \max_j \widetilde{\mathbf{D}}_{ij}, \tag{25}$$

$$\mathbf{D} = \exp(\widetilde{\mathbf{D}} - \boldsymbol{m}) \tag{26}$$

$$\mathbf{S} = \frac{1}{\sqrt{d_{qk}}} \boldsymbol{Q}\boldsymbol{K}^\top \tag{27}$$

$$\overline{\mathbf{S}} = \mathbf{S} \odot \mathbf{D} \tag{28}$$

$$\boldsymbol{n} = \max\left(|\overline{\mathbf{S}}\, \mathbf{1}|, \exp(-\boldsymbol{m})\right) \tag{29}$$

$$\mathbf{H} = \left(\overline{\mathbf{S}} \odot \left(\boldsymbol{n}^{-1}\right)\right)\, \boldsymbol{V}, \tag{30}$$

where $\mathbf{1} \in \mathbb{R}^T$ is a vector of ones. The forget gate activation matrix $\mathbf{F} \in \mathbb{R}^{T \times T}$ is computed by

$$\mathbf{F}_{ij} = \begin{cases} 0 & \text{for } i < j \\ 1 & \text{for } i = j \\ \prod_{k=j+1}^{i} \sigma\left(\tilde{\mathrm{f}}_k\right) = \sum_{k=j+1}^{i} \log \sigma\left(\tilde{\mathrm{f}}_k\right) & \text{for } i > j \end{cases}. \tag{31}$$

Similarly, the input gate pre-activation matrix $\tilde{\mathbf{I}} \in \mathbb{R}^{T \times T}$ is given by

$$\tilde{\mathbf{I}}_{ij} = \begin{cases} 0 & \text{for } i < j \\ \tilde{\mathrm{i}}_j & \text{for } i \geqslant j \end{cases}. \tag{32}$$

Note that in contrast to the recurrent formulation, in the parallel formulation the states $\boldsymbol{C}_t$ are not materialized, i.e. computed explicitly. This comes at the cost of computing the quadratic matrices $\mathbf{D}, \mathbf{S} \in \mathbb{R}^{T \times T}$, with an overall quadratic scaling in sequence length T.

B.2 DETAILED CHUNKWISE-PARALLEL FORMULATION

In this section we provide more detailed formulas for the chunkwise-parallel formulation of the mLSTM from Section 2.2.

Chunkwise Gates. Given the logarithmic forget gates $\bar{\mathbf{f}}^{(k)} = \log \sigma(\tilde{\mathbf{f}}^{(k)}) \in \mathbb{R}^L$ and input gates $\bar{\mathbf{i}}^{(k)} = \log \exp(\tilde{\mathbf{i}}^{(k)}) \in \mathbb{R}^L$, we can compute the chunkwise gates as

$$\mathrm{g}_k = \mathrm{sum}\left(\bar{\mathbf{f}}^{(k)}\right) = \sum_{i=1}^{L} \bar{\mathrm{f}}_i^{(k)} \quad \in \mathbb{R}, \tag{33}$$

$$\mathbf{b}_k = \mathrm{cumsum}\left(\bar{\mathbf{f}}^{(k)}\right) \in \mathbb{R}^L, \text{ with } \quad \mathrm{b}_{k,j} = \sum_{i=1}^{j} \bar{\mathrm{f}}_i^{(k)} \text{ for } j = 1, 2, \ldots, L \tag{34}$$

$$\mathbf{a}_k = \mathrm{rev\_cumsum}\left(\bar{\mathbf{f}}^{(k)}\right) + \bar{\mathbf{i}}^{(k)} \in \mathbb{R}^L, \text{ with } \quad \mathrm{a}_{k,j} = \sum_{i=j+1}^{L} \bar{\mathrm{f}}_i^{(k)} + \bar{\mathrm{i}}_j^{(k)} \text{ for } j = 1, 2, \ldots, L. \tag{35}$$

Inter-chunk Recurrent Contribution. The inter-chunk recurrence is given by

$$m_k^{(\text{inter})} = \max\left\{\mathrm{g}_k + m_{k-1}^{(\text{inter})},\ \max\ \mathbf{a}_k\right\} \tag{36}$$

$$\boldsymbol{C}_k = \exp\left(\mathrm{g}_k + m_{k-1}^{(\text{inter})} - m_k^{(\text{inter})}\right)\boldsymbol{C}_{k-1} + \left(\exp\left(\mathbf{a}_k - m_k^{(\text{inter})}\right) \odot \boldsymbol{K}^{(k)}\right)^\top \boldsymbol{V}^{(k)} \tag{37}$$

$$\boldsymbol{n}_k = \exp\left(\mathrm{g}_k + m_{k-1}^{(\text{inter})} - m_k^{(\text{inter})}\right)\boldsymbol{n}_{k-1} + \left(\exp\left(\mathbf{a}_k - m_k^{(\text{inter})}\right) \odot \boldsymbol{K}^{(k)}\right)^\top \mathbf{1}. \tag{38}$$

In simplified form we can write the inter-chunk recurrence as

$$\boldsymbol{C}_k = \bar{\mathrm{g}}_k \boldsymbol{C}_{k-1} + \left(\bar{\mathbf{a}}_k \odot \boldsymbol{K}^{(k)}\right)^\top \boldsymbol{V}^{(k)} \qquad = \bar{\mathrm{g}}_k \boldsymbol{C}_{k-1} + \overline{\boldsymbol{K}}^{(k)^\top} \boldsymbol{V}^{(k)} \tag{39}$$

$$\boldsymbol{n}_k = \bar{\mathrm{g}}_k \boldsymbol{n}_{k-1} + \left(\bar{\mathbf{a}}_k \odot \boldsymbol{K}^{(k)}\right)^\top \mathbf{1} \qquad = \bar{\mathrm{g}}_k \boldsymbol{n}_{k-1} + \overline{\boldsymbol{K}}^{(k)^\top} \boldsymbol{V}^{(k)}. \tag{40}$$

with the running max state integrated into the gates.

Intra-chunk Parallel Contribution. The recurrent part is followed by the intra-chunk parallel contribution given by

$$\widetilde{\mathbf{D}}^{(k)} = \begin{cases} -\infty & \text{for } i < j \\ \mathbf{b}_k - \mathbf{b}_k^\top + \bar{\mathbf{i}}^{(k)^\top} & \text{for } i \geqslant j \end{cases} \tag{41}$$

$$\boldsymbol{m}_k^{(\text{intra})} = \max_j \widetilde{\mathbf{D}}_{ij}^{(k)} \tag{42}$$

$$\mathbf{D}^{(k)} = \exp(\widetilde{\mathbf{D}}^{(k)} - \boldsymbol{m}_k^{(\text{intra})}) \tag{43}$$

$$\mathbf{S}^{(k)} = \frac{1}{\sqrt{d_{qk}}} \boldsymbol{Q}^{(k)} \boldsymbol{K}^{(k)^\top} \tag{44}$$

$$\overline{\mathbf{S}}^{(k)} = \mathbf{S}^{(k)} \odot \mathbf{D}^{(k)}. \tag{45}$$

where $\exp$ is acting component-wise.

Output computation. The contributions from the intra-chunk parallel part $\mathbf{H}_{\text{intra}}^{(k)}$ are combined with the inter-chunk recurrent part $\mathbf{H}_{\text{inter}}^{(k)}$ to obtain the hidden states $\mathbf{H}^{(k)}$ for each chunk k (see Figure 1):

$$\boldsymbol{m}_k^{(\text{combine})} = \max\left\{\mathbf{b}_k + m_{k-1}^{(\text{inter})}, \boldsymbol{m}_k^{(\text{intra})}\right\} \tag{46}$$

$$\mathbf{H}_{\text{inter}}^{(k)} = \left(\exp\left(\mathbf{b}_k + m_{k-1}^{(\text{inter})} - \boldsymbol{m}_k^{(\text{combine})}\right) \odot \frac{\boldsymbol{Q}^{(k)}}{\sqrt{d_{qk}}}\right) \boldsymbol{C}_{k-1} \tag{47}$$

$$= \left(\overline{\mathbf{b}}_k \odot \frac{\boldsymbol{Q}^{(k)}}{\sqrt{d_{qk}}}\right) \boldsymbol{C}_{k-1} \tag{48}$$

$$= \overline{\boldsymbol{Q}}^{(k)} \boldsymbol{C}_{k-1} \tag{49}$$

$$\mathbf{H}_{\text{intra}}^{(k)} = \overline{\mathbf{S}}^{(k)} \boldsymbol{V}^{(k)} \tag{50}$$

$$\mathbf{H}^{(k)} = \frac{\left(\overline{\mathbf{b}}_k \odot (\boldsymbol{Q}^{(k)}/\sqrt{d_{qk}})\right) \boldsymbol{C}_{k-1} + \overline{\mathbf{S}}^{(k)} \boldsymbol{V}^{(k)}}{\max\left\{\left|\left(\overline{\mathbf{b}}_k \odot (\boldsymbol{Q}^{(k)}/\sqrt{d_{qk}})\right) \boldsymbol{n}_{k-1} + \overline{\mathbf{S}}^{(k)} \mathbf{1}\right|,\ \exp\left(-\boldsymbol{m}_k^{(\text{combine})}\right)\right\}} \tag{51}$$

$$= \frac{\overline{\boldsymbol{Q}}^{(k)} \boldsymbol{C}_{k-1} + \overline{\mathbf{S}}^{(k)} \boldsymbol{V}^{(k)}}{\max\left\{\left|\overline{\boldsymbol{Q}}^{(k)} \boldsymbol{n}_{k-1} + \overline{\mathbf{S}}^{(k)} \mathbf{1}\right|,\ \exp\left(-\boldsymbol{m}_k^{(\text{combine})}\right)\right\}} \tag{52}$$

$$= \left(\overline{\boldsymbol{Q}}^{(k)} \boldsymbol{C}_{k-1} + \overline{\mathbf{S}}^{(k)} \boldsymbol{V}^{(k)}\right) / \mathbf{h}_{\text{denom}}^{(k)}. \tag{53}$$

B.3 CHUNKWISE-PARALLEL BACKWARD PASS

In this section we provide a detailed description of the backward pass of the chunkwise-parallel mLSTM.

Gradients Through Normalizer States. Following Sun et al. (2023), we do not compute the gradients through the normalizer states $\boldsymbol{n}$. The gradients cancel out due to the Layer- or RMS-Norm on the mLSTM cell hidden states $\mathbf{H}$, since the normalizer state is constant over the embedding or feature dimension, which is the normalization dimension.

Inter-chunk Recurrent Backward Pass. Given the incoming memory cell state gradients from the next chunk $\delta\boldsymbol{C}_k$ and the hidden state output gradients $\delta\mathbf{H}^{(k)}$ for chunk k, we can compute the inter-chunk recurrent backward pass. The query, key and value gradients $\delta\boldsymbol{Q}_{\text{inter}}^{(k)}$, $\delta\boldsymbol{K}_{\text{inter}}^{(k)}$ and $\delta\boldsymbol{V}_{\text{inter}}^{(k)}$ of the inter-chunk recurrent part are computed by:

$$\delta\widetilde{\mathbf{H}}^{(k)} = \frac{\delta\mathbf{H}^{(k)}}{\mathbf{h}_{\text{denom}}^{(k)}} \tag{54}$$

$$\delta\boldsymbol{V}_{\text{inter}}^{(k)} = \overline{\boldsymbol{K}}^{(k)}\ \delta\boldsymbol{C}_k \tag{55}$$

$$\delta\overline{\boldsymbol{K}}^{(k)} = \boldsymbol{V}^{(k)}\ \delta\boldsymbol{C}_k^\top \tag{56}$$

$$\delta\boldsymbol{K}_{\text{inter}}^{(k)} = \delta\overline{\boldsymbol{K}}^{(k)} \odot \overline{\boldsymbol{a}}_k\ \mathbf{1}^\top \tag{57}$$

$$\delta\overline{\boldsymbol{Q}}^{(k)} = \delta\widetilde{\mathbf{H}}^{(k)}\ \boldsymbol{C}_{k-1}^\top \tag{58}$$

$$\delta\boldsymbol{Q}_{\text{inter}}^{(k)} = \frac{1}{\sqrt{d_{qk}}}\ \delta\overline{\boldsymbol{Q}}^{(k)} \odot \overline{\boldsymbol{b}}_k\ \mathbf{1}^\top \tag{59}$$

The memory cell state gradients $\delta\boldsymbol{C}_{k-1}$ have incoming contributions from the next timestep $\delta\boldsymbol{C}_{k-1}^{(\text{rec})}$ and output $\delta\boldsymbol{C}_{k-1}^{(\text{out})}$. They are given as

$$\delta\boldsymbol{C}_{k-1} = \delta\boldsymbol{C}_{k-1}^{(\text{rec})} + \delta\boldsymbol{C}_{k-1}^{(\text{out})} \tag{60}$$

$$= \overline{\mathrm{g}} \odot \delta\boldsymbol{C}_k + \overline{\boldsymbol{Q}}^{(k)^\top}\ \delta\widetilde{\mathbf{H}}^{(k)}. \tag{61}$$

Finally, we can compute the cumulative gate gradients $\delta\overline{\mathrm{g}}_k$, $\delta\boldsymbol{a}_k$ and $\delta\boldsymbol{b}_k$ for chunk k as

$$\delta\overline{\mathrm{g}}_k = \mathbf{1}^\top(\boldsymbol{C}_{k-1} \odot \delta\boldsymbol{C}_k)\ \mathbf{1} \tag{62}$$

$$\delta\mathrm{g}_k = \delta\overline{\mathrm{g}}_k \odot \overline{\mathrm{g}}_k \tag{63}$$

$$\delta\overline{\boldsymbol{a}}_k = (\delta\overline{\boldsymbol{K}}^{(k)} \odot \boldsymbol{K}^{(k)})\ \mathbf{1} \tag{64}$$

$$\delta\boldsymbol{a}_k = \delta\overline{\boldsymbol{a}}_k \odot \overline{\boldsymbol{a}}_k \tag{65}$$

$$\delta\overline{\boldsymbol{b}}_k = (\delta\overline{\boldsymbol{Q}}^{(k)} \odot \frac{\boldsymbol{Q}^{(k)}}{\sqrt{d_{qk}}})\ \mathbf{1} \tag{66}$$

$$\delta\boldsymbol{b}_k = \delta\overline{\boldsymbol{b}}_k \odot \overline{\boldsymbol{b}}_k. \tag{67}$$

Intra-chunk Parallel Backward Pass. Given the mLSTM hidden state output gradients $\delta\mathbf{H}^{(k)}$ the intra chunk query, key and value gradients $\delta\boldsymbol{Q}_{\text{intra}}^{(k)}$, $\delta\boldsymbol{K}_{\text{intra}}^{(k)}$ and $\delta\boldsymbol{V}_{\text{intra}}^{(k)}$ gradients are computed by

$$\delta\widetilde{\mathbf{H}}^{(k)} = \frac{\delta\mathbf{H}^{(k)}}{\mathbf{h}_{\text{denom}}^{(k)}} \tag{68}$$

$$\mathbf{S}^{(k)} = \frac{1}{\sqrt{d_{qk}}}\boldsymbol{Q}^{(k)}{\boldsymbol{K}^{(k)}}^{\top} \tag{69}$$

$$\overline{\mathbf{S}}^{(k)} = \mathbf{S}^{(k)} \odot \mathbf{D}^{(k)} \tag{70}$$

$$\delta\boldsymbol{V}_{\text{intra}}^{(k)} = {\overline{\mathbf{S}}^{(k)}}^{\top}\, \delta\widetilde{\mathbf{H}}^{(k)} \tag{71}$$

$$\delta\overline{\mathbf{S}}^{(k)} = \delta\widetilde{\mathbf{H}}^{(k)}\, {\boldsymbol{V}^{(k)}}^{\top} \tag{72}$$

$$\delta\mathbf{S}^{(k)} = \delta\overline{\mathbf{S}}^{(k)} \odot \mathbf{D}^{(k)} \tag{73}$$

$$\delta\boldsymbol{Q}_{\text{intra}}^{(k)} = \frac{1}{\sqrt{d_{qk}}}\, \delta\mathbf{S}^{(k)}\, \boldsymbol{K}^{(k)} \tag{74}$$

$$\delta\boldsymbol{K}_{\text{intra}}^{(k)} = \frac{1}{\sqrt{d_{qk}}}\, {\delta\mathbf{S}^{(k)}}^{\top}\, \boldsymbol{Q}^{(k)} \tag{75}$$

In order to compute the cumulative intra gate gradients, we compute the gradients through the gate matrix $\mathbf{D}^{(k)}$, which is computed from the cumulative forget gates

$$\boldsymbol{b}_k^{(q)} = \text{cumsum}(\bar{\mathbf{f}}_q^{(k)}) \in \mathbb{R}^{L_q} \tag{76}$$

$$\boldsymbol{b}_k^{(kv)} = \text{cumsum}(\bar{\mathbf{f}}_{kv}^{(k)}) \in \mathbb{R}^{L_{kv}}, \tag{77}$$

where we use the logarithmic forget gates $\bar{\mathbf{f}} = \log\sigma(\tilde{\mathbf{f}})$. We denote the dimensions as L_q and L_{kv} for the query and key-value dimensions, respectively. Omitting the masking operation, we compute the gate matrix as

$$\mathbf{D}^{(k)} = \boldsymbol{b}_k^{(q)}\, \mathbf{1}_{kv}^{\top} - \mathbf{1}_q\, {\boldsymbol{b}_k^{(kv)}}^{\top} + \mathbf{1}_q\, {\bar{\mathbf{i}}_{kv}^{(k)}}^{\top}, \tag{78}$$

where $\mathbf{1}_q \in \mathbb{R}^{L_q}$ and $\mathbf{1}_{kv} \in \mathbb{R}^{L_{kv}}$ are vectors of ones used to indicate broadcast operations, and $\bar{\mathbf{i}}_{kv}^{(k)} \in \mathbb{R}^{L_{kv}}$ are the logarithmic input gates for chunk k.

The gradients are computed as

$$\delta\mathbf{D}^{(k)} = \delta\overline{\mathbf{S}}^{(k)} \odot \mathbf{S}^{(k)} \tag{79}$$

$$\delta\boldsymbol{b}_k^{(q)} = \delta\mathbf{D}^{(k)}\, \mathbf{1}_{kv} \tag{80}$$

$$\delta\boldsymbol{b}_k^{(kv)} = -{\delta\mathbf{D}^{(k)}}^{\top}\, \mathbf{1}_q \tag{81}$$

$$\delta\bar{\mathbf{i}}_{kv}^{(k)} = {\delta\mathbf{D}^{(k)}}^{\top}\, \mathbf{1}_q. \tag{82}$$

Combined input and gate gradients. The intra and inter chunk gradients are combined by summing up the contributions. This yields for the query, key and value gradients

$$\delta\boldsymbol{Q}^{(k)} = \delta\boldsymbol{Q}_{\text{inter}}^{(k)} + \delta\boldsymbol{Q}_{\text{intra}}^{(k)} \tag{83}$$

$$\delta\boldsymbol{K}^{(k)} = \delta\boldsymbol{V}_{\text{inter}}^{(k)} + \delta\boldsymbol{K}_{\text{intra}}^{(k)} \tag{84}$$

$$\delta\boldsymbol{V}^{(k)} = \delta\boldsymbol{V}_{\text{inter}}^{(k)} + \delta\boldsymbol{V}_{\text{intra}}^{(k)}. \tag{85}$$

The input and forget gate gradients $\bar{\mathbf{i}}^{(k)}$ and $\bar{\mathbf{f}}^{(k)}$ can be computed from the cumulative gate gradients $\delta\mathrm{g}_k$, $\delta\boldsymbol{b}_k$ and $\delta\boldsymbol{a}_k$ with the following equalities

$$\delta\bar{\mathbf{f}}^{(k)} = \delta\mathrm{g}_k \tag{86}$$

$$\delta\bar{\mathbf{f}}^{(k)} = \text{rev\_cumsum}(\delta\boldsymbol{b}_k) \tag{87}$$

$$\delta\bar{\mathbf{f}}^{(k)} = \text{rev\_cumsum}(\delta\boldsymbol{a}_k) \tag{88}$$

$$\delta\bar{\mathbf{i}}^{(k)} = \delta\boldsymbol{a}_k \tag{89}$$

C Extended Tiled Flash Linear Attention

C.1 GPU Fundamentals

A GPU (Graphics Processing Unit) is a specialized processor designed to efficiently handle large-scale parallel computation tasks, such as matrix multiplications in neural networks. These tasks are divided into small programs called kernels, that are executed on GPUs. A kernel loads data from high bandwidth memory (HBM), performs work on it, and writes the results back to HBM. For writing efficient kernels, it is important to understand the software hierarchy of the GPU, which closely follows its physical hardware hierarchy.

GPU Hierarchy. At the lowest level the GPU runs multiple Threads, operating on small but fast register memory in parallel. On the software side usually multiple (e.g. 32) Threads are grouped together into Warps. Again, multiple Warps are grouped into Thread blocks which together execute a kernel on a physical core, called streaming multiprocessor (SM). Warps or Threads within the same Thread block can communicate data through special on-chip shared memory (SRAM). When executing a kernel, a grid (with typically 3 dimensions) of Thread blocks that run in parallel is launched on the GPU. All Thread blocks have access to the large but slow off-chip high-bandwidth memory (HBM), which has both the largest latency and least bandwidth of all GPU memory. *For efficient kernels it is important to minimize memory read and writes from and to HBM.*

Specialized Compute Units. Modern GPUs have specialized compute units – called tensor cores – that accelerate matrix multiplications on GPUs. Tensor cores have most of the GPU compute and are accessed at the warp or block level. *For efficient kernels it is important to maximize tensor core utilization.*

Triton Language. Triton is a GPU kernel programming language with an associated compiler, that provides a Python-based environment for GPU programming. The user can load data from HBM via a `tl.load` instruction and store data to HBM via `tl.store`. `tl.dot` is an instruction, that leverages tensor cores for matrix multiplications. While this Triton interface of increases productivity in writing very fast custom kernels, peak performance can be achieved sometimes only with CUDA kernels. We write our kernels in Triton and leave a CUDA implementation for future work. In contrast to NVIDIAs programming model CUDA, which provides access to all levels of the GPU hierarchy, Triton programs operate on the Thread block level and hide register and thread management from the user. Therefore, we describe TFLA on the more abstract Thread block or program level in the following section.

C.2 Tiled Computation

For the tiled computation of the intra-chunk hidden state contribution $\mathbf{H}_{\text{intra}}$ within a chunk, we consider blocks of the matrix $\mathbf{S} = \begin{bmatrix}\mathbf{S}^{(1)} & \mathbf{S}^{(2)}\end{bmatrix}$ and the gate matrix $\mathbf{D} = \begin{bmatrix}\mathbf{D}^{(1)} & \mathbf{D}^{(2)}\end{bmatrix}$, with $\mathbf{S}^{(i)}, \mathbf{D}^{(i)} \in \mathbb{R}^{B_{Lhq} \times B_{Lkv}}$. Here, the superscript i denotes the block index along the L_{kv} dimension (and not the chunk index). Similarly, we consider blocks of the value matrix $\boldsymbol{V} = \begin{bmatrix}\boldsymbol{V}^{(1)} \\ \boldsymbol{V}^{(2)}\end{bmatrix}$, with $\boldsymbol{V}^{(i)} \in \mathbb{R}^{B_{kv} \times B_{dhv}}$. We then accumulate the unnormalized hidden state blocks $\mathbf{H}^{(i)}_{\text{intra,num}} \in \mathbb{R}^{B_{Lkv} \times B_{dhv}}$ and the corresponding normalizer $\boldsymbol{l}^{(i)} \in B_{Lkv}$ as

$$\boldsymbol{m}^{(1)} = \max_j \widetilde{\mathbf{D}}^{(1)}_{ij} \tag{90}$$

$$\boldsymbol{l}^{(1)} = (\mathbf{S}^{(1)} \odot \exp(\widetilde{\mathbf{D}}^{(1)} - \boldsymbol{m}^{(1)}))\, \mathbf{1} \tag{91}$$

$$\mathbf{H}^{(1)}_{\text{intra,num}} = (\mathbf{S}^{(1)} \odot \exp(\widetilde{\mathbf{D}}^{(1)} - \boldsymbol{m}^{(1)}))\, \boldsymbol{V}^{(1)} \tag{92}$$

$$\boldsymbol{m}^{(2)} = \max\left(\boldsymbol{m}^{(1)},\ \max_j \widetilde{\mathbf{D}}^{(2)}_{ij}\right) \tag{93}$$

$$\boldsymbol{l}^{(2)} = \exp(\boldsymbol{m}^{(1)} - \boldsymbol{m}^{(2)})\, \boldsymbol{l}^{(1)} + (\mathbf{S}^{(2)} \odot \exp(\widetilde{\mathbf{D}}^{(2)} - \boldsymbol{m}^{(2)}))\, \mathbf{1} \tag{94}$$

$$\mathbf{H}^{(2)}_{\text{intra,num}} = \exp(\boldsymbol{m}^{(1)} - \boldsymbol{m}^{(2)})\, \mathbf{H}^{(1)}_{\text{intra,num}} + (\mathbf{S}^{(2)} \odot \exp(\widetilde{\mathbf{D}}^{(2)} - \boldsymbol{m}^{(2)}))\, \boldsymbol{V}^{(2)}. \tag{95}$$

After computing this intra-chunk part, we need to do one more rescaling step to combine the intra-chunk and inter-chunk parts of the hidden state output $\mathbf{H}^{(k)}$ since $\mathbf{H}^{(k)}_{\text{intra}}$ and $\mathbf{H}^{(k)}_{\text{inter}}$ were computed with different max states. Therefore, we compute the final hidden state output $\mathbf{H}^{(k)}$ as

$$\boldsymbol{m}_k^{(\text{combine})} = \max\left\{\mathbf{b}_k + m_{k-1}^{(\text{inter})}, \boldsymbol{m}_k^{(2)}\right\} \tag{96}$$

$$\mathbf{H}^{(k)} = \frac{\overline{\boldsymbol{Q}}^{(k)}\boldsymbol{C}_{k-1} + \exp\left(\boldsymbol{m}_k^{(2)} - \boldsymbol{m}_k^{(\text{combine})}\right)\overline{\mathbf{S}}^{(k)}\boldsymbol{V}^{(k)}}{\max\left\{|\overline{\boldsymbol{Q}}^{(k)}\boldsymbol{n}_{k-1} + \exp\left(\boldsymbol{m}_k^{(2)} - \boldsymbol{m}_k^{(\text{combine})}\right)\boldsymbol{l}_k^{(2)}|,\ \exp\left(-\boldsymbol{m}_k^{(\text{combine})}\right)\right\}}, \tag{97}$$

where we assume that $\boldsymbol{m}_k^{(2)}$ is the block maximum and $\boldsymbol{l}_k^{(2)}$ is the normalizer after the last B_{Lkv} block of the intra-chunk computation for chunk k.

C.3 TFLA Forward Pass

For notational simplicity we drop the k index for the query, key and value matrices as $\boldsymbol{Q} \in \mathbb{R}^{L_{hq}\times d_{qk}}$, $\boldsymbol{K} \in \mathbb{R}^{L_{kv}\times d_{qk}}$ and $\boldsymbol{V} \in \mathbb{R}^{L_{kv}\times d_v}$, respectively. We make use of reweighting (as discussed in Appendix C.2) in order to keep track of the maximum value over the gate matrix tiles, similar to (Dao et al., 2022).

The forward pass algorithm of TFLA for one thread block is described in Algorithm 1.

Note that the loop in line 27 of Algorithm 1 is the same as the loop in line 6. In both loops we load the same blocks of the matrix $\boldsymbol{Q}$. Fusing these loops would avoid loading this data twice. Unfortunately, fusing these loops efficiently is problematic due to the online computation of the maximum $\boldsymbol{m}_{old}$ and $\boldsymbol{m}_{new}$ in the loop in line 4 and the dependence of $\boldsymbol{m}_k^{(\text{combine})}$ and $\overline{\mathbf{b}}_k$ on the final $\boldsymbol{m}_{new}$ (see Appendix D.1 and C.2).

We address this issue in Section 4 by modifying the input gate of the mLSTM.

C.4 TFLA Backward Pass

For the TFLA backward pass, we need to compute the gradients of the queries, keys and values $\delta\boldsymbol{Q}^{(k)}, \delta\boldsymbol{K}^{(k)}$ and $\delta\boldsymbol{V}^{(k)}$. Omitting the gate computations and normalization, we write a simplified version of these gradients as

$$\underset{(L_{hq}\times d_{qk})}{\delta\boldsymbol{Q}^{(k)}} = \underbrace{\left(\underset{(L_{hq}\times d_{hv})}{\delta\mathbf{H}^{(k)}}\ \underset{(d_{hv}\times L_{kv})}{\boldsymbol{V}^{(k)\top}}\right)\underset{(L_{kv}\times d_{qk})}{\boldsymbol{K}^{(k)}}}_{\delta\boldsymbol{Q}^{(k)}_{\text{intra}}} + \underbrace{\underset{(L_{hq}\times d_{hv})}{\delta\mathbf{H}^{(k)}}\ \underset{(d_{hv}\times d_{qk})}{\boldsymbol{C}_{k-1}^{\top}}}_{\delta\boldsymbol{Q}^{(k)}_{\text{inter}}} \tag{98}$$

$$\underset{(L_{kv}\times d_{qk})}{\delta\boldsymbol{K}^{(k)}} = \underbrace{\left(\underset{(L_{kv}\times d_{hv})}{\boldsymbol{V}^{(k)}}\ \underset{(d_{hv}\times L_{hq})}{\delta\mathbf{H}^{(k)\top}}\right)\underset{(L_{hq}\times d_{qk})}{\boldsymbol{Q}^{(k)}}}_{\delta\boldsymbol{K}^{(k)}_{\text{intra}}} + \underbrace{\underset{(L_{kv}\times d_{hv})}{\boldsymbol{V}^{(k)}}\ \underset{(d_{hv}\times d_{qk})}{\delta\boldsymbol{C}_k^{\top}}}_{\delta\boldsymbol{K}^{(k)}_{\text{inter}}} \tag{99}$$

$$\underset{(L_{kv}\times d_{hv})}{\delta\boldsymbol{V}^{(k)}} = \underbrace{\left(\underset{(L_{kv}\times d_{qk})}{\boldsymbol{K}^{(k)}}\ \underset{(d_{qk}\times L_{hq})}{\boldsymbol{Q}^{(k)\top}}\right)\underset{(L_{hq}\times d_{hv})}{\delta\mathbf{H}^{(k)}}}_{\delta\boldsymbol{V}^{(k)}_{\text{intra}}} + \underbrace{\underset{(L_{kv}\times d_{qk})}{\boldsymbol{K}^{(k)}}\ \underset{(d_{qk}\times d_{hv})}{\delta\boldsymbol{C}_k}}_{\delta\boldsymbol{V}^{(k)}_{\text{inter}}}. \tag{100}$$

We see that each of the query, key and value gradients has a similar structure as the forward pass in equation (17). They can be computed with the same work partitioning scheme, where we parallelize over the outer chunk size and outer embedding dimension of the matrix multiplications and loop over the inner dimensions, respectively. For example, for the key gradients $\delta\boldsymbol{K}^{(k)}$ we parallelize over the outer chunk size L_{kv} and the outer embedding dimension d_{qk} and loop over the inner dimensions L_{hq} and d_{hv}. Table 1 summarizes the TFLA work partitioning scheme for the forward and backward pass kernels.

Algorithm 1 TFLA Intra-Chunk Forward Pass for mLSTMexp ($\mathbf{H}^{(k)}$ Kernel)

Require: Matrices $\boldsymbol{Q} \in \mathbb{R}^{L_{hq} \times d_{qk}}$, $\boldsymbol{K} \in \mathbb{R}^{L_{kv} \times d_{qk}}$, $\boldsymbol{V} \in \mathbb{R}^{L_{kv} \times d_{hv}}$.
States $\boldsymbol{C}_{k-1} \in \mathbb{R}^{d_{qk} \times d_v}$, $\boldsymbol{n}_{k-1} \in \mathbb{R}^{d_{qk}}$.
Input- and cumulative forget gate vectors $\mathbf{i}_k$, $\mathbf{b}_k \in \mathbb{R}^{L_{hq}}$.
Block sizes B_{dqk}, B_{dhv}, B_{Lhq} and B_{Lkv}, where $B_{Lhq} \geqslant B_{Lkv}$.
Block Q index i_{Lq} and Block HV index i_{dhv}.

Initialize $\boldsymbol{m}_{old}$, $\boldsymbol{m}_{new} \in \mathbb{R}^{L_q}$ to $-\infty$ in SRAM.
▷ Compute intra-chunk contribution
Initialize accumulators $\mathbf{H}_{\text{intra}} \in \mathbb{R}^{B_{Lhq} \times B_{dv}}$ and $\mathbf{n}^{(\text{intra})} \in \mathbb{R}^{B_{Lhq}}$ in SRAM.
Load $\mathbf{b}_k^{(q)} \in \mathbb{R}^{B_{Lhq}}$ from HBM to SRAM.
for $i = 1$ to $\left\lfloor \frac{(i_{Lq}+1) \cdot B_{Lhq}}{B_{Lkv}} \right\rfloor$ **do**
  Initialize accumulator $\mathbf{S}^{(i)} \in \mathbb{R}^{B_{Lhq} \times B_{Lkv}}$ in SRAM.
  for $j = 1$ to $\left\lceil \frac{d_{qk}}{B_{dqk}} \right\rceil$ **do**
    Load $\boldsymbol{Q}^{(j)} \in \mathbb{R}^{B_{Lhq} \times B_{dqk}}$ and $\boldsymbol{K}^{(j)} \in \mathbb{R}^{B_{Lkv} \times B_{dqk}}$ from HBM to SRAM.
    Accumulate $\mathbf{S}^{(i)} \mathrel{+}= \boldsymbol{Q}^{(j)} {\boldsymbol{K}^{(j)}}^{\top}$.
  end for
  Load $\mathbf{b}_k^{(kv)} \in \mathbb{R}^{B_{Lkv}}$ and $\mathbf{i}_k^{(kv)} \in \mathbb{R}^{B_{Lkv}}$ from HBM to SRAM.
  Compute $\widetilde{\mathbf{D}}^{(i)} = \mathbf{b}_k^{(q)} - {\mathbf{b}_k^{(kv)}}^{\top} + {\mathbf{i}_k^{(kv)}}^{\top} \in \mathbb{R}^{B_{Lhq} \times B_{Lkv}}$.
  if $i \cdot B_{Lkv} \geqslant i_{Lq} \cdot B_{Lhq}$ **then**
    Apply causal mask to $\widetilde{\mathbf{D}}^{(i)}$.
  end if
  Compute $\boldsymbol{m}_{new} = \text{maximum}\{\boldsymbol{m}_{old}, \text{rowmax } \widetilde{\mathbf{D}}^{(i)}\}$.
  Compute $\mathbf{D}^{(i)} = \exp(\widetilde{\mathbf{D}}^{(i)} - \boldsymbol{m}_{new})$.
  Compute $\overline{\mathbf{S}}^{(i)} = \frac{1}{\sqrt{d_{qk}}} \mathbf{S} \odot \mathbf{D}^{(i)}$.
  Load $\boldsymbol{V}^{(i)} \in \mathbb{R}^{B_{Lkv} \times B_{dhv}}$ for Block i_{dhv} from HBM to SRAM.
  Accumulate $\mathbf{H}_{\text{intra}} = \exp(\boldsymbol{m}_{old} - \boldsymbol{m}_{new}) \cdot \mathbf{H}_{\text{intra}} + \overline{\mathbf{S}}^{(i)} \boldsymbol{V}$.
  Accumulate $\mathbf{n}^{(\text{intra})} = \exp(\boldsymbol{m}_{old} - \boldsymbol{m}_{new}) \cdot \mathbf{n}^{(\text{intra})} + \overline{\mathbf{S}}^{(i)} \mathbf{1}$.
  Update $\boldsymbol{m}_{old} = \boldsymbol{m}_{new}$.
end for
▷ Compute inter-chunk contribution
Load $m_{k-1}^{(\text{inter})} \in \mathbb{R}$ from HBM to SRAM.
Compute $\boldsymbol{m}_k^{(\text{combine})} = \text{maximum}\left\{\mathbf{b}_k^{(q)} + m_{k-1}^{(\text{inter})}, \boldsymbol{m}_{new}\right\}$.
Compute $\overline{\mathbf{b}}_k = \exp\left(\mathbf{b}_k^{(q)} + m_{k-1}^{(\text{inter})} - \boldsymbol{m}_k^{(\text{combine})}\right)$.
Initialize accumulators $\mathbf{H}_{\text{inter}} \in \mathbb{R}^{B_{Lhq} \times B_{dhv}}$ for Block i_{dhv} and $\mathbf{n}^{(\text{inter})} \in \mathbb{R}^{B_{Lhq}}$ in SRAM.
▷ Note: This is the same loop as the inner one above. They cannot be merged because of the max state computation.
for $j = 1$ to $\left\lceil \frac{d_{qk}}{B_{dqk}} \right\rceil$ **do**
  Load $\boldsymbol{Q}^{(j)} \in \mathbb{R}^{B_{Lhq} \times B_{dqk}}$ and $\boldsymbol{C}_{k-1}^{(j)} \in \mathbb{R}^{B_{dqk} \times B_{dhv}}$ for Block i_{dhv} from HBM to SRAM.
  Compute $\overline{\boldsymbol{Q}}^{(j)} = \frac{1}{\sqrt{d_{qk}}} \boldsymbol{Q}^{(j)} \odot \mathbf{b}_k^{(q)}$.
  Accumulate $\mathbf{H}_{\text{inter}} \mathrel{+}= \overline{\boldsymbol{Q}}^{(j)} \boldsymbol{C}_{k-1}^{(j)}$.
  Load $\boldsymbol{n}_{k-1}^{(j)} \in \mathbb{R}^{B_{dqk}}$.
  Accumulate $\mathbf{n}^{(\text{inter})} \mathrel{+}= \overline{\boldsymbol{Q}}^{(j)} \boldsymbol{n}_{k-1}^{(j)}$.
end for
▷ Combine inter- and intra-chunk contributions
Compute $\mathbf{H}^{(\text{comb})} = \mathbf{H}_{\text{intra}} + \exp(\boldsymbol{m}_{new} - \boldsymbol{m}_k^{(\text{combine})})\, \mathbf{H}_{\text{inter}}$.
Compute $\mathbf{n}^{(\text{comb})} = \mathbf{n}^{(\text{intra})} + \exp(\boldsymbol{m}_{new} - \boldsymbol{m}_k^{(\text{combine})})\, \mathbf{n}^{(\text{inter})}$.
Compute $\mathbf{H}^{(k)} = \frac{\mathbf{H}^{(\text{comb})}}{\max\left\{|\mathbf{n}^{(\text{comb})}|, \exp(-\boldsymbol{m}_k^{(\text{combine})})\right\}}$.
Store $\mathbf{H}^{(k)}$, $\mathbf{n}^{(\text{comb})}$ and $\boldsymbol{m}_k^{(\text{combine})}$ to HBM.

D EXTENDED MLSTM WITH SIGMOID INPUT GATE

D.1 STABILIZATION OF THE EXPONENTIAL INPUT GATE

In this section we show how the exponential input gate is stabilized with the max state m_t (Beck et al., 2024). The stabilization is based on the idea of Safe Softmax (Milakov & Gimelshein, 2018). We will see that the max state stabilization ensures that the argument of the exponential input gate activation is always smaller than 1. We will also see that the normalizer state guarantees cancellation of the max state, so that the overall outputs of the mLSTM remain unaffected by the max state.

Without stabilization mLSTM hidden state output is computed as

$$\boldsymbol{h}_t = \tilde{\mathbf{o}}_t \odot \frac{\boldsymbol{C}_t^\top \boldsymbol{q}_t}{\max\left\{\left|\boldsymbol{n}_t^\top \boldsymbol{q}_t\right|, 1\right\}}, \tag{101}$$

where we omit the scaling factor $\sqrt{d_{qk}}$ for $\boldsymbol{q}$. To simplify we also omit the lower bound and the absolute value on the dot product in the denominator. We obtain

$$\boldsymbol{h}_t = \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\boldsymbol{C}_t^\top \boldsymbol{q}_t}{\boldsymbol{n}_t^\top \boldsymbol{q}_t}. \tag{102}$$

Inserting the update formulas for the memory cell state $\boldsymbol{C}_t$ and the normalizer state $\boldsymbol{n}_t$ gives

$$\boldsymbol{h}_t = \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\left(\sigma(\tilde{\mathrm{f}}_t)\, \boldsymbol{C}_{t-1} + \exp(\tilde{\mathrm{i}}_t)\, \boldsymbol{k}_t\, \boldsymbol{v}_t^\top\right)^\top \boldsymbol{q}_t}{\left(\sigma(\tilde{\mathrm{f}}_t)\, \boldsymbol{n}_{t-1} + \exp(\tilde{\mathrm{i}}_t)\, \boldsymbol{k}_t\right)^\top \boldsymbol{q}_t}. \tag{103}$$

We now show that from this unstabilized version of the mLSTM we can derive the stabilized form in three steps. At first we use the identity $\sigma(\tilde{\mathrm{i}}) = \exp(\log(\sigma(\tilde{\mathrm{f}}_t)))$, extend the fraction in equation (103) by $\exp(-m_t)$ and select $m_t = \max\{\log(\sigma(\tilde{\mathrm{f}}_t)),\, \tilde{\mathrm{i}}_t\}$ to be the maximum of the two arguments of the exponential function. This gives

$$\boldsymbol{h}_t = \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\boldsymbol{C}_t^\top \boldsymbol{q}_t \cdot \exp(-m_t)}{\boldsymbol{n}_t^\top \boldsymbol{q}_t \cdot \exp(-m_t)} = \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\left(\exp(\log(\sigma(\tilde{\mathrm{f}}_t)) - m_t)\, \boldsymbol{C}_{t-1} + \exp(\tilde{\mathrm{i}}_t - m_t)\, \boldsymbol{k}_t\, \boldsymbol{v}_t^\top\right)^\top \boldsymbol{q}_t}{\left(\exp(\log(\sigma(\tilde{\mathrm{f}}_t)) - m_t)\, \boldsymbol{n}_{t-1} + \exp(\tilde{\mathrm{i}}_t - m_t)\, \boldsymbol{k}_t\right)^\top \boldsymbol{q}_t}. \tag{104}$$

In this way, we ensure that the arguments of the exponential function are always smaller than 1, such that numerical overflow due to large values can never occur.

As next step we reparameterize $\boldsymbol{C}_t$ and $\boldsymbol{n}_t$ to $\widetilde{\boldsymbol{C}}_t$ and $\widetilde{\boldsymbol{n}}_t$.

$$\begin{aligned} \widetilde{\boldsymbol{C}}_t &= \boldsymbol{C}_t \exp(-m_t) \quad &\rightarrow\ \widetilde{\boldsymbol{C}}_{t-1} &= \boldsymbol{C}_{t-1} \exp(-m_{t-1}) \ \Leftrightarrow\ \boldsymbol{C}_{t-1} &= \widetilde{\boldsymbol{C}}_{t-1} \exp(m_{t-1}) \\ \widetilde{\boldsymbol{n}}_t &= \boldsymbol{n}_t \exp(-m_t) \quad &\rightarrow\ \widetilde{\boldsymbol{n}}_{t-1} &= \boldsymbol{n}_{t-1} \exp(-m_{t-1}) \ \Leftrightarrow\ \boldsymbol{n}_{t-1} &= \widetilde{\boldsymbol{n}}_{t-1} \exp(m_{t-1}) \end{aligned} \tag{105}$$

Finally, we replace $\boldsymbol{C}_t$ and $\boldsymbol{n}_t$ with the stabilized states $\widetilde{\boldsymbol{C}}_t$ and $\widetilde{\boldsymbol{n}}_t$ in the recurrence. We arrive at

$$\begin{aligned} \boldsymbol{h}_t &= \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\left(\exp(\log(\sigma(\tilde{\mathrm{f}}_t)) + m_{t-1} - m_t)\, \widetilde{\boldsymbol{C}}_{t-1} + \exp(\tilde{\mathrm{i}}_t - m_t)\, \boldsymbol{k}_t\, \boldsymbol{v}_t^\top\right)^\top \boldsymbol{q}_t}{\left(\exp(\log(\sigma(\tilde{\mathrm{f}}_t)) + m_{t-1} - m_t)\, \widetilde{\boldsymbol{n}}_{t-1} + \exp(\tilde{\mathrm{i}}_t - m_t)\, \boldsymbol{k}_t\right)^\top \boldsymbol{q}_t} \\ &= \sigma\left(\tilde{\mathbf{o}}_t\right) \odot \frac{\widetilde{\boldsymbol{C}}_t^\top \boldsymbol{q}_t}{\widetilde{\boldsymbol{n}}_t\, \boldsymbol{q}_t} \end{aligned} \tag{106}$$

Now we choose the max state as $m_t = \max\{\log(\sigma(\tilde{\mathrm{f}}_t)) + m_{t-1},\, \tilde{\mathrm{i}}_t\}$ and arrive at the stabilized mLSTM formulas by changing the denominator to $\max\left\{\left|\widetilde{\boldsymbol{n}}_t^\top \boldsymbol{q}_t\right|, \exp(m_{t-1})\right\}$. We have to add $\exp(m_{t-1})$ also to the right side of the maximum, so that it cancels out.

To summarize, we see that the normalizer is necessary for the max state to cancel out and the exponential input gate argument bounded through the max state.

D.2 Empirical Transfer Behavior Analysis of the mLSTM

We provide details on the transfer behavior analysis of mLSTMexp and mLSTMsig in Section 4.2.

Experiment Setup. We analyze the transfer behavior of the mLSTM for a single head and a single input sequence of length T =512. The inputs are for the queries, keys and values $\boldsymbol{q}_t$, $\boldsymbol{k}_t$ and $\boldsymbol{v}_t$ are sampled from the standard normal distribution $\mathcal{N}(0, 1)$. We set the head dimensions to d_{qk} =128 and d_{hv} =128. As norm layer $\mathrm{NORM}(\boldsymbol{x})$ we use the RMS-norm. Changing the norm to layernorm does not alter the results, as for this experiment we set the mean of the inputs to zero. For every plot we measure the gains G_{before} and G_{after} (as defined in (23)) for input and forget gate preactivation values in the ranges [-12, 8] and [-5, 12], respectively.

Effect of Normalization Layer Epsilon on Transfer Behavior. Based on our analysis on the normalization layer after the gated linear RNN operation in Section 4.2, we hypothesize that the normalization layer and especially the norm epsilon ϵ is integral to the gating mechanism. In this experiment we probe the effect of the epsilon value on the transfer behavior of the mLSTM. Figure 6a and Figure 7a show the transfer behavior of mLSTMexp and mLSTMsig for ϵ =[1e-2, 1e-6, 1e-8], respectively.

We observe that the epsilon acts in the same way for mLSTMexp and mLSTMsig. Increasing ϵ causes an offset of the gain in positive y-direction, increasing ϵ in negative y-direction. We set our default value ϵ =1e-6, which yields the best performance in our experiments (see Sec. 5.1).

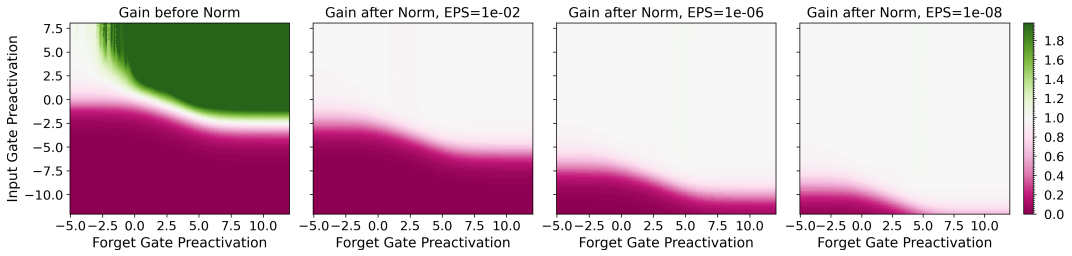
Normalizers of mLSTMexp and mLSTMsig. In this experiment we test the effect of different normalizers $\boldsymbol{n}$ in Equation 29 for mLSTMexp and mLSTMsig. The parallel formulation in Section B.1 is presented for the mLSTM with exponential input gate, but applies similarly to the mLSTM with sigmoid input gate. For the default mLSTMsig, we set the normalizer to $\boldsymbol{n} = \mathbf{1}$ and modify the calculation of the gate matrix $\mathbf{D}$ for sigmoid input gates.

In Figure 6 we show the results of different normalizers for the mLSTM with exponential input gate. Only the default mLSTMexp with correct normalizer and max state (in Fig. 6a) shows a transfer behavior that depends on the input gate.

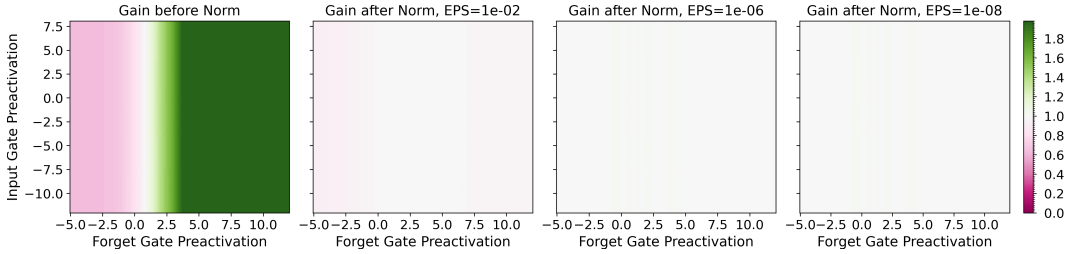
In contrast, in Figure 7a and 7b we observe that incorporating a normalizer similar to mLSTMexp (excluding the max state) into mLSTMsig does not alter its transfer behavior.

The other two normalizer variants for mLSTMsig in Figure 7c and 7d show a clearly different transfer behavior and do not train successfully. Similarly, the variants in Figure 6b and 6c also fail to train successfully.

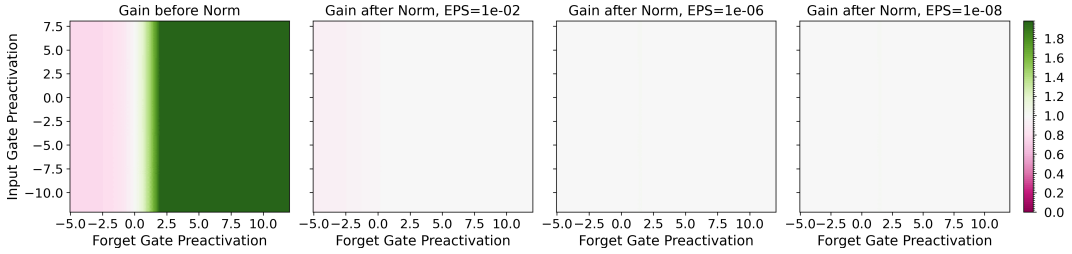
In summary, we find that if the mLSTM exhibits the characteristic gate dependent transfer behavior it trains successfully and shows good performance in our language modeling experiments. In order to achieve this behavior for the mLSTMexp we need to normalize correctly as derived in Section D.1. Adding a normalizer to the mLSTMsig does not change performance and transfer behavior, if the normalizer incorporates a lower bound on the dot-product $\boldsymbol{n}_t^\top \boldsymbol{q}_t$. However, our default mLSTMsig omits the normalizer in order to reduce computational cost and runtime.



(a) mLSTMexp — $\boldsymbol{n} = \max\left(|\overline{\mathbf{S}}\,\mathbf{1}|, \exp(-\boldsymbol{m})\right)$ (default)

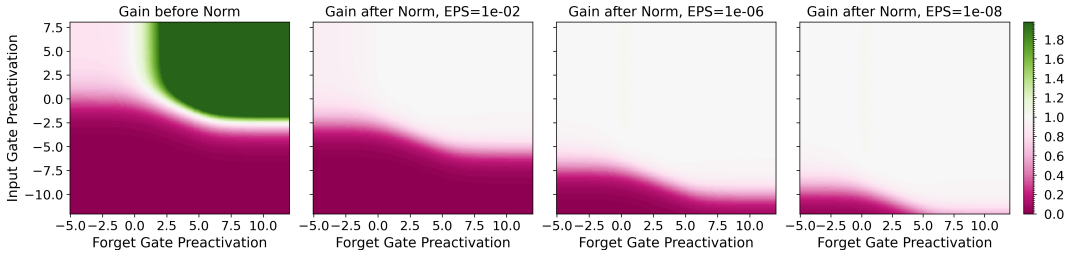


(b) mLSTMexp — $\boldsymbol{n} = \max\left(|\overline{\mathbf{S}}\,\mathbf{1}|, 1\right)$ (wrong $\boldsymbol{m}$ state)

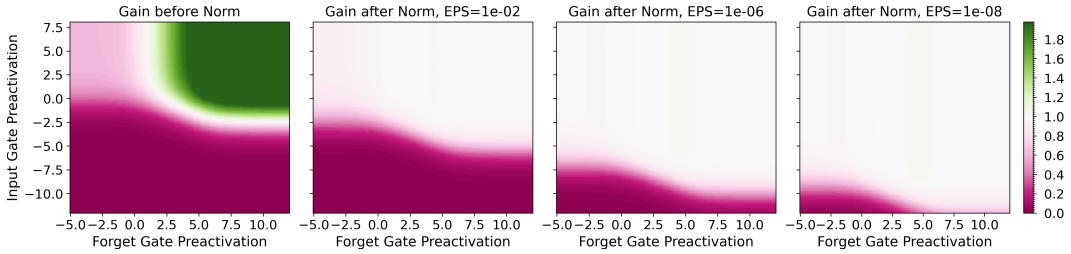


(c) mLSTMexp — $\boldsymbol{n} = \mathbf{1}$ (no normalizer)

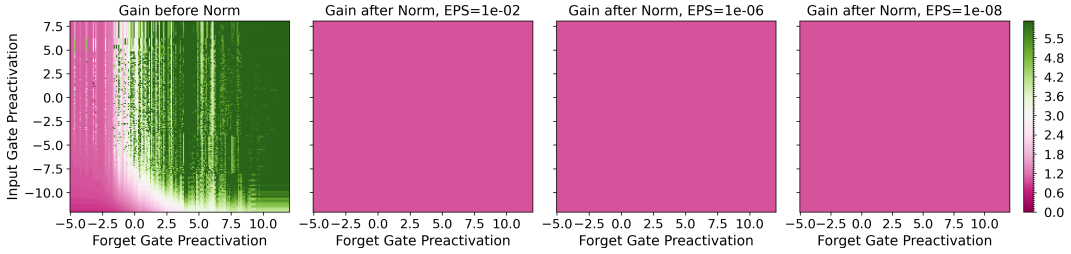
Figure 6: Transfer behavior of the **mLSTM with exponential input gate** for different normalization layer epsilons (EPS) and different normalizer variants. Only the default normalization shows the input gate dependent transfer behavior. Varying the normalization layer epsilon causes a shift of the gain curve in y-direction.



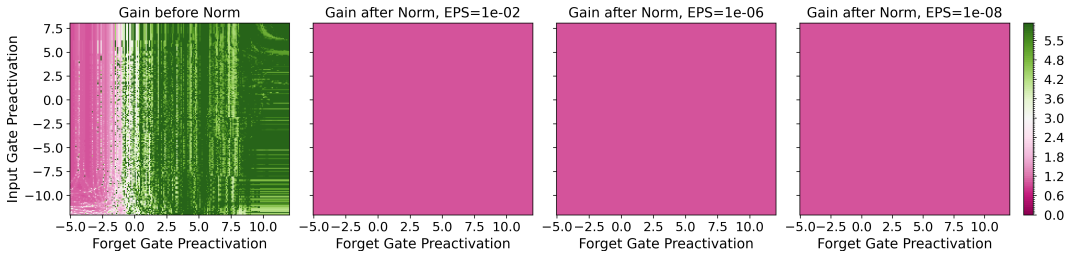
(a) mLSTMsig — $\boldsymbol{n} = \mathbf{1}$ (default)



(b) mLSTMsig — $\boldsymbol{n} = \max\left(|\overline{\mathbf{S}}\ \mathbf{1}|, 1\right)$ (same normalizer as mLSTMexp)



(c) mLSTMsig — $\boldsymbol{n} = |\overline{\mathbf{S}}\ \mathbf{1}|$ (no max – different colorbar)



(d) mLSTMsig — $\boldsymbol{n} = \overline{\mathbf{S}}\ \mathbf{1}$ (no $|\cdot|$ – different colorbar)

Figure 7: Transfer behavior of the **mLSTM with sigmoid input gate** for different normalization layer epsilons (EPS) and different normalizer variants. Removing the normalizer from mLSTMsig (which is our default setting in (a)) has no effect on the transfer behavior. If the normalizer is added, it should be bounded by 1 (see (b)). Varying the normalization layer epsilon causes a shift of the gain curve in y-direction.

E EXTENDED EXPERIMENTS

In this section, we provide additional experiments and details to Section 5.

E.1 NUMERICAL VALIDATION OF TFLA KERNELS

Before we begin our experiments on langauge modeling, we first verify that our kernels yield the same result as a reference implementation in pure JAX based on the fully parallel formulation (see Appendix B.1).

Validation Perplexity Match (Table 3). We compare the validation perplexity at the end of training for 160M parameter mLSTMexp and mLSTMsig models trained on 19B tokens. We use context length 4096 since the parallel JAX implementation go out-of-memory for longer contexts. Model architecture and training recipe follows or general setup described in Appendix E.2.

In Table 3 we confirm that our kernels yield the same results as our reference implementation in JAX.

Table 3: Validation Perplexity for 160M parameter models at context length 4096 trained on 19B tokens.

	EXP			SIG	
HEADS	JAX PARALLEL	LIMIT CHUNK	XL CHUNK	JAX PARALLEL	XL CHUNK
6	21.02	21.03	21.18	21.01	21.05
12	21.01	21.03	21.07	21.02	21.06

E.2 EXTENDED LANGUAGE MODELING EXPERIMENTS WITH MLSTM

In this section we provide details on our experiment setup, model architecture and training recipe and add additional performance results on context length 8192 as well as analyze the effect of the epsilon parameter in the norm layer.

Software and Hardware Setup. We run our language modeling experiments in JAX 0.4.34 (Bradbury et al., 2018) and use FLAX 0.9.0 (Heek et al., 2024) to implement our models. We implement our kernels in Triton 3.1.0 (Tillet et al., 2019; Tillet, 2024) and use JAX-Triton 0.2.0 (Vikram et al., 2022) to integrate the kernels into JAX. Our kernel benchmark experiments are run in PyTorch 2.5.1 (Paszke et al., 2019), because most kernel baselines are available in PyTorch. All experiments are run on NVIDIA H100 80GB GPUs.

Model Architecture. The model architecture for mLSTMexp and mLSTMsig follows the design of most dense Transformer decoder only large language models (Radford et al., 2019; Brown et al., 2020; Touvron et al., 2023a;b).

An embedding layer, is followed by a stack of blocks and a language model head that produces the output logits (i.e. the values before softmax), which typically consists of a normalization layer and a linear (unembedding) layer. We apply logit soft-capping (Team, 2024), such that the value of the logits stay between $-c$ and c for a specific cap value c. We choose c =30. The logits are capped with the following function:

$$\text{softcap}(\boldsymbol{x}) = c \cdot \tanh(\boldsymbol{x}/c) \tag{107}$$

We use the GPT-NeoX tokenizer (Black et al., 2022) with vocabulary size 50257 and do not tie the weights for the embedding layers and and the last (unembedding) layer.

Each block consists of two layers, where each layer has skip a connection and a normalization layer before the layer input (i.e. we use the pre-norm block architecture). As normalization layer we use the RMS-norm (Zhang & Sennrich, 2019) with epsilon ϵ =1e-6.

The first layer is a sequence-mix layer, that mixes the tokens along the sequence or time dimension. For standard Transformers this is the Attention operation (Vaswani et al., 2017). In our case, we

replace Attention by the mLSTM operation with exponential or sigmoid input gate. Similar to Attention, mLSTM processes each token in multiple parallel heads. The second layer in the block is a feedforward linear layer that mixes the tokens per timestep channelwise. We use the SwiGLU feedforward linear layers (Shazeer, 2020; Touvron et al., 2023a).

For the mLSTM we set the head dimension for the queries and keys to be half of the values, i.e. $d_{qk} = 0.5\, d_{hv}$. We use Layernorm (Ba et al., 2016) as $\mathrm{NORM}(\boldsymbol{x})$ operation with epsilon ϵ =1e-6 in our experiments. [1] We apply soft-capping from equation (107) on the input and forget gate preactivations, as we found that this improves training stability. For the gate preactivations we set c =15.

We provide the remaining model parameters in Table 4.

Training Recipe. We train our models with the AdamW optimizer (Loshchilov & Hutter, 2019) with β_1 =0.9, β_2 =0.95 and ϵ =1e-8. We use learning rates and batch sizes as specified in Table 4. We apply a weight decay of 0.1 to all linear layers (including the last linear layer or unembedding) and exclude biases and the token embeddings from weight decay. We clip the gradient norm at 0.5. We use a cosine learning rate scheduler with a linear warmup for the first 750 steps and decay to 0.1 of the peak learning rate, followed by a linear cooldown to 0 for the last 1000 steps. We list the number of training steps for every model size in Table 4. During pre-training we ensure that no information is leaked across document borders by resetting the memory states at the beginning of each new document. We implement this by manually setting the forget gate preactivations to a large negative values at the beginning of each new document.

Table 4: Training and Model Architecture Hyperparameters for our model sizes 160M, 400M and 1.4B.

Model Size	Blocks	Embedding Dim	Heads	Head Dim	LR	Batch Size	Steps	Tokens 4k ctx	Tokens 8k ctx
160M	12	768	6 12	128 64	3e-3	128	36k	19B	38B
400M	24	1024	8 16	128 64	1e-3	128	46k	24B	48B
1.4B	24	2048	4 8 16	512 256 128	8e-4	256	31k	33B	65B

Additional Performance Results (Table 5). In Table 5 we show the validation perplexity for mLSTMexp and mLSTM for context length 8192 (the results for context length 4096 are shown in Table 2). For some head dimension configurations we observed irrecoverable gradient norm spikes during training (indicated by -).

Effect of Trainable Input Gate (Table 6). We investigate the effect of the input gate on the performance. Table 6 shows that having the input gate learnable consistently improves performance for both mLSTMexp and mLSTMsig.

Effect of Input Gate Bias Initialization (Figure 8 and 9). In our transfer behavior analysis in Section 4.2 we find that there is a transition from suppressing the signal to passing the signal at negative input gate values of around -8 (see Figure 3). Since we initialize the weights of the gates $\boldsymbol{w}_{\{\mathrm{i,f}\}}$ to 0, the biases of the input and forget gates determine the actual position in the x-y plane in the beginning of training. Initially, with input gate biases initialized to 0, we observe a high gradient norm variance, which was more pronounced for mLSTMsig (see Figure 8a and 9a).

Therefore, we test to initialize the input gate biases at larger negative values. The forget gate biases are initialized equally spaced in the range [3,6]. As the weights $\boldsymbol{w}_{\{\mathrm{i,f}\}}$ grow during training, so do

[1] We confirmed empirically that the type of normalization layer does not affect the performance as well as our qualitative results on transfer behavior and gradient norm variance. Therefore, we generally prefer RMS-norm as it faster.

Table 5: Validation Perplexity at context length 8192. EXP and SIG denote mLSTMexp and mLSTMsig. LIMIT and XL correspond to `limit_chunk` and `xl_chunk` kernels. - indicates that the run experienced irrecoverable loss spikes during training.

SIZE	TOKENS	HEADS	LLAMA	EXP LIMIT	EXP XL	SIG XL
160M	38B	6	19.99	20.29	20.43	20.46
		12		20.31	20.42	20.52
400M	48B	8	16.05	15.91	16.01	16.08
		16		15.95	16.01	-
1.4B	65B	4	12.97	12.69	12.71	12.91
		8		12.62	12.65	12.67
		16		12.59	-	12.75

Table 6: Validation Perplexity for 160M mLSTMs at context length 4096 with learnable and fixed input gate (bias initialized at -10).

INPUT GATE	EXP LIMIT	SIG XL
FIXED	21.23	21.24
LEARNABLE	20.95	21.04

the gate preactivations and the model could learn to gradually move into the dynamical region of Figure 3, where the input signal is passed.

Indeed, as we observe in Figure 8 and 9 initializing the input gate biases to -10 effectively mitigates gradient norm spikes and reduces high gradient norm variance during training for both mLSTMexp and mLSTMsig. We therefore conclude that the additional input gate not only improves performance (see Table 6), but also improves training stability, if initialized correctly.

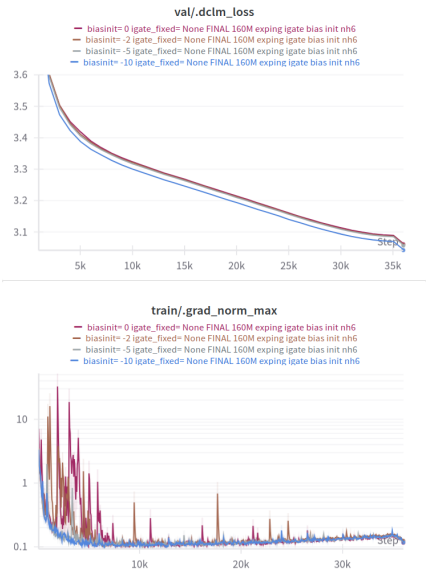
We use the `limit_chunk` kernel for mLSTMexp and our `xl_chunk` kernel for mLSTMsig and confirm that we obtain the same behavior with the `xl_chunk` kernel for mLSTMexp.

Effect of Normalization Layer Epsilon on Performance (Figure 10). In our empirical transfer behavior analysis of the mLSTM in Section 4.2 and D.2 we find that the transfer behavior depends on the input and forget gate preactivations, as well as the normalization layer epsilon (see Figure 6a and 7a). Therefore, we perform a grid search over different normalization layer epsilons and input gate bias initializations for the mLSTM with exponential input gate with 160M parameters and 6 heads at context length 4096. We show the results in Figure 10.

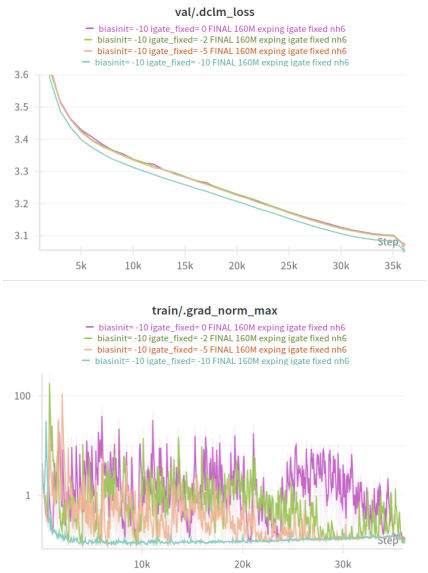
We observe that there is a diagonal region from norm layer epsilon and input gate bias (ϵ, b_i)=(1e-6, -10) to (1e-4, -5) with improved performance. This indicate that if we increase the norm layer epsilon we can or should also increase the input gate bias initialization, as the shift of the gain curve in positive y-direction for larger epsilons in Figure 6a suggests. This supports our hypothesis in Section 4.2, that the norm layer is important for the gating mechanism.

We use (ϵ, b_i)=(1e-6, -10) as our default configuration.

Input Gate Activations over Training (Figure 11). We show the maximum input gate preactivations (maximum over batch, sequence and head dimension) over training for mLSTMexp and mLSTMsig with 160M parameters in Figure 11. Both models have the input gate bias initialized to -10.

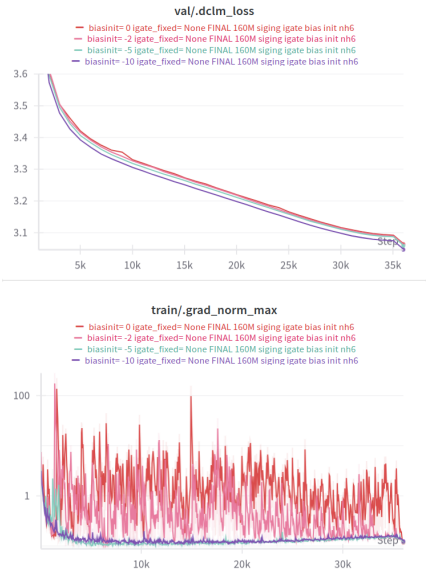


(a) mLSTMexp with Trainable Input Gate

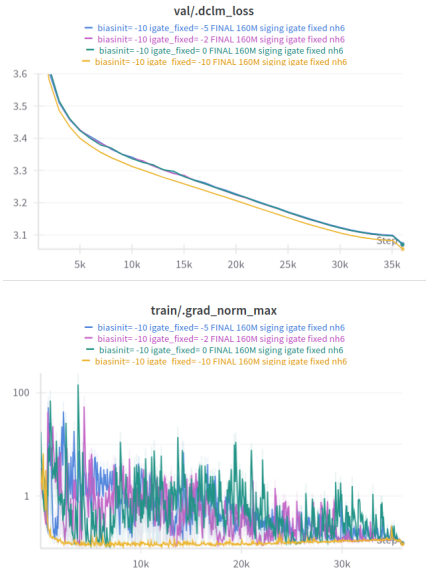


(b) mLSTMexp with Fixed Input Gate

Figure 8: Trainable and fixed **exponential input gate** for bias initializations [0, -2, -5, -10] and norm epsilon ϵ =1e-6.



(a) mLSTMsig with Trainable Input Gate



(b) mLSTMsig with Fixed Input Gate

Figure 9: Trainable and fixed **sigmoid input gate** for bias initializations [0, -2, -5, -10] and norm epsilon ϵ =1e-6.

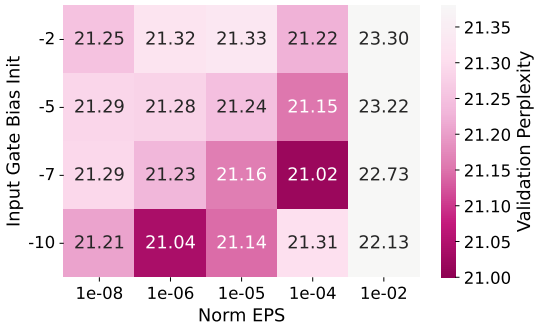


Figure 10: Validation Perplexity of mLSTMexp with 160M parameters with 6 heads. Grid search over norm layer epsilon and input gate bias initialization. The diagonal region of improved performance indicates, that there exists an interplay between the norm layer epsilon and input gate bias initialization. This supports the hypothesis that the norm layer is important for the gating mechanism.

./../figures/Layer activation 160M_cropped.png

Figure 11: Maximum input gate pre-activation values $\tilde{\mathrm{i}}_t$ over training for mLSTMexp and mLSTMsig with 160M parameters. Maximum taken over batch, sequence and head dimension. Both models have the input gate bias initialized to -10. In most cases the input gate pre-activations remain below zero.

E.3 Extended Kernel Benchmark

In this section, we provide details on our benchmark setup and add additional benchmark results.

Details on GPU Memory Measurement. In Figure 5 and 12 we measure the GPU memory used by the kernels. For this, we use the PyTorch torch.cuda.max_memory_allocated API to measure the peak memory allocated during one kernel iteration. We make sure that the memory statistics are reset after each iteration and that the PyTorch caches are cleared before the start of each benchmark.

Details on the Runtime Benchmark (Figure 4). In our TFLA kernel runtime benchmark in Section 5.2, Figure 4 we report the median runtime of 30 iterations, after 10 warmup iterations in milliseconds. We run all kernels in `bfloat16` precision.

We use the standard embedding dimension of 4096 for 7B Transformer models for our benchmark. Since different models and kernels have different default input sizes at this embedding dimension, we adapt the head dimension, number of heads and remaining input dimensions for each kernel accordingly. Following the practice of Shah et al. (2024) we keep the number of tokens constant at 65,536 and vary the sequence length (i.e. $T = [512, 1024, 2048, 4096, 8192, 16384, 32768, 65536]$) and batch size accordingly (i.e. $N_{\text{batch}} = 65536/T$).

We benchmark the following mLSTM kernels:

- **mLSTMexp (FLA `limit_chunk`):** Our own baseline kernel for the mLSTM with exponential input gate with limited chunk size based on FLA. Similar to FLA this kernel employs only single level sequence parallelism across chunks. We report the best performing chunk size of 64. The chunk size of 128 would still fit in SRAM, but is considerably slower.
- **mLSTMexp (TFLA `xl_chunk`):** TFLA kernel for the mLSTM with exponential input gate with two levels of sequence parallelism. We set the chunk size to the best performing chunk size of 128.
- **mLSTMsig (TFLA `xl_chunk`):** TFLA kernel for the mLSTM with sigmoid input gate. We set the chunk size to 128, but find chunk size 256 to perform equally well in terms of runtime (see Fig. 12 and 5).

For all our mLSTM kernels we use 16 heads, which results in head dimension $d_{hv} = 4096/16 = 256$ for the values. Similar to GLA (Yang et al., 2024b), we set the query and key head dimension to $d_{qk} = d_{hv}/2$, i.e. $d_{qk} = 128$.

We compare our mLSTM kernels with the following baselines:

- **Torch FlashAttention:** PyTorch 2.5.1 implementation of FlashAttention 2.
  Accessed via `SDPBackend.FLASH_ATTENTION` [2]
- **cuDNN FlashAttention:** NVIDIA cuDNN implementation of FlashAttention 2 integrated in PyTorch 2.5.1.
  Accessed via `SDPBackend.CUDNN_ATTENTION`.
- **FlashAttention 3:** FlashAttention 3 implementation[3], which has been optimized for NVIDIA H100 GPUs (Shah et al., 2024).
- **GLA (FLA):** Gated Linear Attention Triton kernel based on the FlashLinearAttention algorithm with one level of sequence parallelism (Yang et al., 2024b). Implementation from the official FLA repository, version 0.1[4]
- **Simple GLA (FLA):** A simple version of GLA with scalar forget gates per head. This primitive is not published as a new sequence modeling primitive but serves as a reference implementation for kernels for RetNet (Sun et al., 2023) or Mamba 2 (Dao & Gu, 2024) in the FLA library Yang & Zhang (2024). Therefore, we find it interesting to add it as baseline. Implementation from the official FLA repository, version 0.1

[2] See torch.nn.attention.SDPBackend

[3] See `https://github.com/Dao-AILab/flash-attention`

[4] See `https://github.com/fla-org/flash-linear-attention`

- **Mamba:** Mamba CUDA kernel Gu & Dao (2024). Implementation from the official Mamba repository, version 2.2.4.
- **Mamba 2:** Mamba 2 Triton kernels Dao & Gu (2024). Implementation from the official Mamba repository, version 2.2.4.[5]

For all FlashAttention baselines we use 32 heads with head dimension 128 for queries, keys and values. For the FlashLinearAttention (FLA) kernels GLA and Simple GLA, we use the identical head configuration as for our TFLA mLSTM kernels (i.e. 16 heads, $d_{hv} = 256$, $d_{qk} = 128$). For Mamba, we use our embedding dimension of 4096 and set the state dimension to 16 similar to Gu & Dao (2024). For Mamba 2, we use their default head dimension of 64 and set the number of heads to $4096/64 = 64$. Note that smaller head dimension can yield faster runtimes (see Figure 14).

We show the results of this benchmark for varying sequence length and constant number of tokens in Figure 4. When comparing the forward pass runtime only, we find that Mamba2 and Simple GLA kernels are slightly faster than our mLSTMsig kernels. However, this difference is within 1 ms. In training, when forward and backward pass runtime is measured, our TFLA kernels are faster than FlashAttention 3 for longer sequence lengths and more than two times faster than Mamba 2 kernels for all sequence lengths. Only Simple GLA (FLA) can keep up in training speed with our TFLA mLSTM kernels. Therefore, we compare the runtime and memory usage for a larger head dimension in Figure 12 and find that this comes at the cost of almost 2 times the GPU memory usage compared to our TFLA mLSTM kernels. These memory savings are achieved by leveraging a larger chunk size, enabled through the two levels of sequence parallelism outlined in Section 3.

Runtime and Memory Comparison with FLA Kernels (Figure 12). In this experiment we compare the runtime and memory consumption of our TFLA mLSTM kernels with prominent kernels from the Flash Linear Attention library. We use a similar setup to our previous benchmark, but perform this comparison with 8 heads at a larger head dimension of 512 for the values and 256 for the queries and keys, since both Beck et al. (2024) and Yang et al. (2024b) report better language modeling performance for larger head dimensions.

In addition to GLA (chunk) and Simple GLA (chunk), we also compare with GLA (fused) which is the non-materialization version of Gated Linear Attention (GLA) (Yang et al., 2024b).

The non-materialization version of GLA has been also proposed by Qin et al. (2024a) as Lightning Attention-2 (see also Section A). For the forward pass it fuses the inter- and intra-chunk part of the chunkwise-parallel Linear Attention formulation (see Section 2.2) and therefore does not materialize the hidden states in GPU memory.

Interestingly, in our experiments we find that even though the non-materialization version uses the least GPU memory of all FLA kernels, it is neither faster nor more memory efficient in training than our TFLA mLSTM kernels (see Figure 12). While Simple GLA is slightly faster (within 3 ms or 15%), it uses almost twice the GPU memory compared to our TFLA mLSTM kernels.

Runtime and Memory Comparison with LightningAttention2 Kernels (Figure 13). Similar to the previous experiment, we compare the runtime and memory consumption of our TFLA mLSTM kernels with LightningAttention2 (Qin et al., 2024a). LightningAttention2 is the core of the recent hybrid large language model MiniMax-01, which combines lightning attention (a linear attention variant with data independent decay) with softmax attention (MiniMax et al., 2025). MiniMax-01 is proposed as a very efficient long-context language model, which makes the comparison between LightningAttention2 and our TFLA mLSTM kernels interesting.

LightningAttention2 also uses the chunkwise-parallel formulation for linear RNNs (see Section 2.2). However, in contrast to Simple GLA and TFLA it does not split the computation in a recurrent and parallel part, but instead processes all chunks fully recurrent (see Section A for more details).

We find that LightningAttention2 supports only identical head dimensions for queries, keys and values up to 128. For this reason, we discuss this comparison separately from the other experiments. We compare our TFLA mLSTM kernels with LightningAttenion2 for 32 and 64 heads, corresponding to head dimension 128 and 64. We keep the number of tokens fixed to 65536 and vary sequence length and batch size in the same way as above.

[5] See https://github.com/state-spaces/mamba

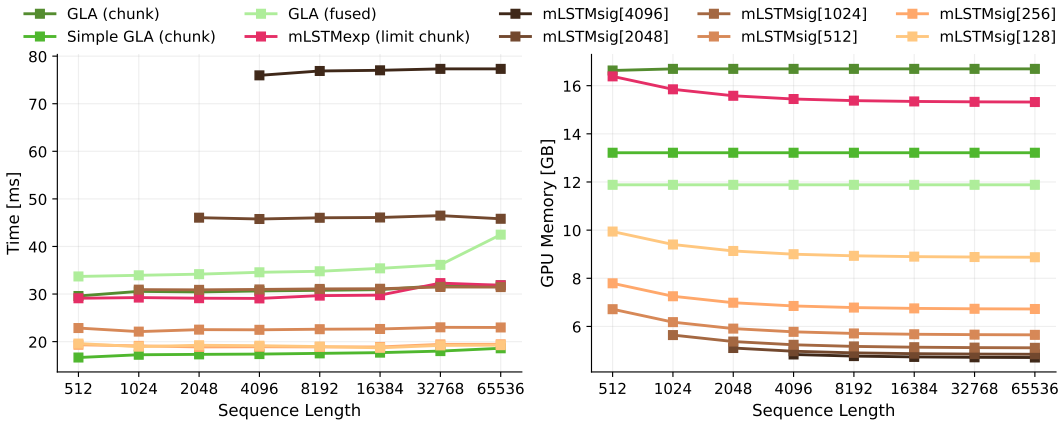


Figure 12: Runtime and Memory Comparison with FLA Kernels.
Left: Runtime (Forward Backward Pass). **Right:** GPU Memory Usage.
We use 8 heads and head dimension of 512 for values, and 256 for queries and keys. Simple GLA (the fastest FLA kernel in our experiments) is slightly faster than our TFLA mLSTMsig kernels but uses almost twice as much GPU memory.

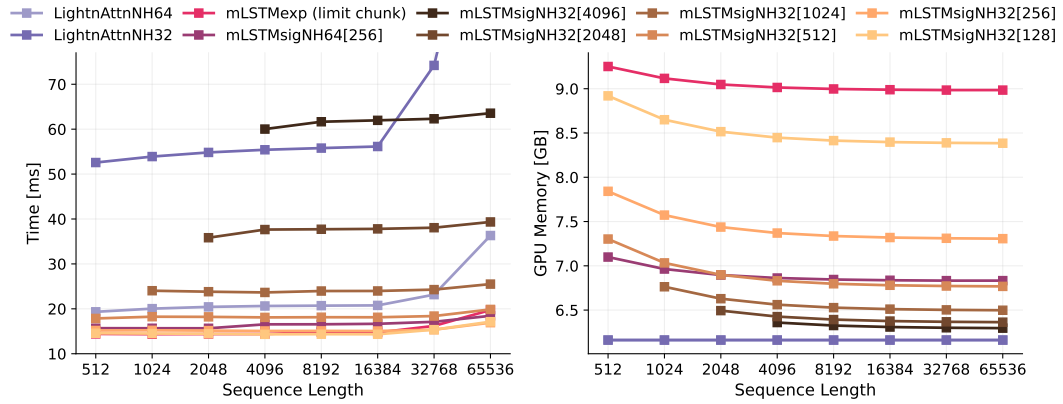


Figure 13: Runtime and Memory Comparison with LightningAttention2.
Left: Runtime (Forward Backward). **Right:** GPU Memory.
We use 32 and 64 heads with head dimension 128 and 64 for queries, keys and values. LightningAttention has the least memory usage of all kernels, but is more than 3 times slower than our TFLA mLSTM at the larger head dimension of 128.

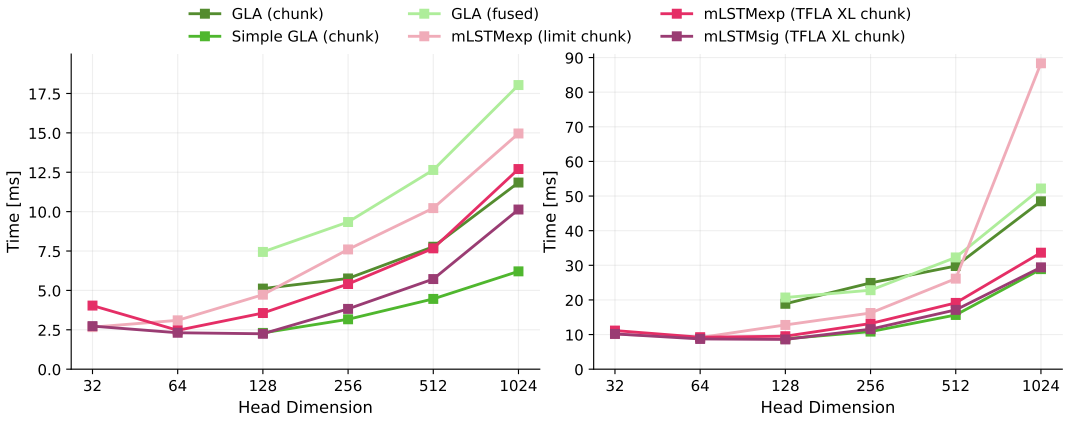


Figure 14: Head Dimension Benchmark for FLA and TFLA mLSTM kernels.
Left: Forward Pass. **Right:** Forward and Backward Pass.
We measure the runtime for sequence length 8192 and batch size 4 for different head dimensions. We use the same head dimension for queries, keys and values. Our TFLA mLSTM kernels show fast runtimes even for very large head dimensions.

We show the results in Figure 13. Since LightningAttention does not materialize intermediate states, it has the least GPU memory usage with 6.2 GB. However, this GPU memory efficiency comes at the cost of a more than 3 times longer runtime compared to our TFLA mLSTMsig kernel with chunk size 256, which uses about 7.3 GB of GPU memory. This highlights that there exists a trade-off between GPU memory usage and runtime for linear RNN kernels based on the chunkwise-parallel formulation. Our experiments demonstrate that our TFLA kernel algorithm provides an effective method to balance this trade-off via the chunk size parameter (see Figure 5).

Runtime Benchmark for Varying Head Dimensions (Figure 14). It has been reported in several other works that larger head dimensions (compared to common Self-Attention head dimensions) lead to improved language modeling performance for linear RNNs (Sun et al., 2023; Beck et al., 2024; Yang et al., 2024b). Consequently, it is desirable for linear RNN kernels to be fast and efficient across a wide range of head dimensions. In this experiment, we evaluate whether our new TFLA kernels exhibit this property.

We vary the head dimension from 32 to 1024 and adapt the number of heads for a total embedding dimension of 4096 and measure the runtime for inputs of sequence length 8192 and batch size 4. We use the same head dimension for queries, keys and values.

For the FLA kernels the head dimensions 32 and 64 did not run, due to Triton compiler errors. As the FLA library is still being developed at the time of writing this paper, we expect this to be fixed soon.

We observe that for small head dimensions (i.e. 32 and 64) our mLSTM limit chunk kernel is as fast as our TFLA mLSTM kernels in training.

In summary, our results in Figure 14 confirm that our TFLA kernels achieve fast runtimes across a wide range of head dimensions.

F FLOP AND MEMORY OPERATION COUNTS

We count the number of FLOPs in a forward pass (with batch size 1) of the mLSTM with exponential and sigmoid input gate. We use a factor of 2 to describe the multiply accumulate cost.

We do not count FLOPs that belong to recomputation, that happens within kernels. For example, when we parallelize across the embedding dimension in the forward kernel $\mathbf{H}^{(k)}$, each of the d_{hv}/B_{dhv} blocks recomputes the matrix $\mathbf{S}$. Similarly, we do not count the additional memory-loading operations that are necessary for the recomputations. During training, we typically have fixed context lengths. Therefore, we do not count loading the initial state and storing the final state.

The mLSTM with sigmoid input gate does not have a normalizer and a max state. Therefore, it has fewer FLOPs and memory operations compared to mLSTM with exponential input gate.

We use factors denoted as F_X to describe the number of FLOPs for operation X (e.g. F_exp for the exponential function). By default, we set all of these factors to 1.

We use the factors bytes_X to denote the size of each element in the tensor (e.g. bytes_QKV for the query, key and value tensors).

F.1 FLOPS FOR THE MLSTM WITH EXPONENTIAL INPUT GATE

- Inter-chunk recurrent:
  - **Chunkwise gates:** num_heads × num_chunks × (0.5×chunk_size × (chunk_size + 1) + 2×chunk_size)
  - **Gates & max state:** num_heads × num_chunks × (3 + F_max + F_exp + chunk_size × (3 + 2 × F_exp))
  - **Numerator:** num_heads × num_chunks × (2×d_qk × d_v + 4×chunk_size × d_qk × d_v + 3×chunk_size × d_qk)
  - **Denominator:** num_heads × num_chunks × (2×d_qk + 4×chunk_size × d_qk)
- Intra-chunk parallel:
  - **Gate matrix:** num_heads × num_chunks × (0.5 × chunk_size × (chunk_size + 1) + chunk_size × chunk_size × (3 + F_mask + F_max + F_exp) + chunk_size × (1 + F_max))
  - **Gated Attn logits:** num_heads × num_chunks × 2×chunk_size × chunk_size × (1 + d_qk)
  - **Numerator:** num_heads × num_chunks × 2×chunk_size × chunk_size × d_v
  - **Denominator:** num_heads × num_chunks × 2 × chunk_size × chunk_size
  - **Output combination:** num_heads × num_chunks × (chunk_size × (1 + F_max) + chunk_size × (2 + F_abs + F_exp + F_max + 2×d_v))

F.2 MEMORY OPERATIONS FOR THE MLSTM WITH EXPONENTIAL INPUT GATE

- Inter-chunk recurrent:
  - **Load:** num_heads × num_chunks × chunk_size × ((d_qk + d_v)×bytes_QKV + 2×bytes_If)
  - **Store:** num_heads × num_chunks × (d_qk×d_v + d_qk + 1) ×bytes_Cnm
- Intra-chunk parallel:
  - **Load:** num_heads × num_chunks × chunk_size × ((2×d_qk + d_v)×bytes_QKV + 2×bytes_If) + num_heads × num_chunks × (d_qk×d_v + d_qk + 1) ×bytes_Cnm
  - **Store:** num_heads × num_chunks × chunk_size × ((d_v×bytes_QKV + 2×bytes_Cnm)

F.3 FLOPs FOR THE mLSTM WITH SIGMOID INPUT GATE

- Inter-chunk recurrent:
  - **Chunkwise gates:** num_heads × num_chunks × (0.5×chunk_size × (chunk_size + 1) + 2×chunk_size)
  - **Gates:** num_heads × num_chunks × (F_exp + (2 × chunk_size + 1) × F_sig))
  - **Numerator:** num_heads × num_chunks × (2×d_qk × d_v + 4×chunk_size × d_qk × d_v + 3×chunk_size × d_qk)
- Intra-chunk parallel:
  - **Gate matrix:** num_heads × num_chunks × (0.5 × chunk_size × (chunk_size + 1) + chunk_size × chunk_size × (2 + F_mask + F_exp))
  - **Gated Attn logits:** num_heads × num_chunks × (2×chunk_size × chunk_size × (1 + d_qk))
  - **Numerator:** num_heads × num_chunks × (2×chunk_size × chunk_size × d_v)
  - **Output combination:** num_heads × num_chunks × (2 × chunk_size × d_v)

F.4 MEMORY OPERATIONS FOR THE mLSTM WITH SIGMOID INPUT GATE

- Inter-chunk recurrent:
  - **Load:** num_heads × num_chunks × chunk_size × ((d_qk + d_v)×bytes_QKV + 2×bytes_If)
  - **Store:** num_heads × num_chunks × d_qk×d_v ×bytes_Cnm
- Intra-chunk parallel:
  - **Load:** num_heads × num_chunks × chunk_size × ((2×d_qk + d_v)×bytes_QKV + 2×bytes_If) + num_heads × num_chunks × d_qk×d_v ×bytes_Cnm
  - **Store:** num_heads × num_chunks × chunk_size × d_v×bytes_QKV