Blog Post

Linear Recurrences Accessible to Everyone

Felix Sarnthein ELLIS Institute Tubingen MPI-IS Tübingen ETH Zürich

1. The Challenge with Linear RNNs

- **Problem:** Investigating linear RNNs (like S4, Mamba, etc.) is difficult.
- **Why?** The parallel scan algorithm is not efficiently expressible in PyTorch.
- **Consequence:** Implementations are often hidden in complex CUDA kernels, limiting accessibility and research progress.
- **Insight:** Element-wise linear recurrences are the common operation across State Space Models (SSMs) and linear RNNs.
- **Proposal:** Use abstraction of linear recurrences to gain intuition for computational structure and make it accessible to a wider audience.

2. Abstraction of Linear Recurrence

- **Definition:** A simple linear update rule of inputs x_l, coefficients c_l, and outputs y_l, starting at $y_0 = x_0$ and iterating for $l = 0 \dots L-1$ steps:

$$y_l = y_{l-1} \cdot c_l + x_l$$

- **Matrix Form:** Unrolling yields a weighted sum with cumulative coefficients $\tilde{c}_{k,l}$ from k to l and $\tilde{c}_{l,l} = 1$:

$$y_l = \sum_{k=0}^{l} \left(\prod_{j=k+1}^{l} c_j \right) \cdot x_k = \sum_{k=0}^{l} \tilde{c}_{k,l} \cdot x_k$$

This describes a linear sequence mixer $y = f(x, c) = \tilde{C}^\top x$ with the lower triangular mixing matrix $\tilde{C}^\top = [\tilde{c}_{k,l}]_{k,l}$.

- **Backprop:** Given $\delta^{(y)} := \frac{\partial \mathcal{L}}{\partial y}^\top$, return $\delta^{(x)} := \frac{\partial \mathcal{L}}{\partial x}^\top$ and $\delta^{(c)} := \frac{\partial \mathcal{L}}{\partial c}^\top$. In the blog post, we derive gradients for back-propagation through time

$$\delta_k^{(x)} = \delta_{k+1}^{(x)} \cdot c_{k+1} + \delta_k^{(y)}$$
$$\delta_i^{(c)} = y_{i-1} \cdot \delta_i^{(x)}$$

In other words, this corresponds to a shifted reverse linear recurrence:

```
def linrec_bwd(d_outputs:Tensor, coeffs:Tensor, outputs:Tensor):
    d_inputs = linrec_fwd(d_outputs, shift(coeffs,-1), rev=True)
    d_coeffs = d_inputs * shift(outputs, 1)
    return d_inputs, d_coeffs
```

- **Versatility:** The abstraction generalizes important sequence operations:
  - Cumulative sums/products by setting $c = [1, \dots]$ resp. $x = [1, 0, \dots]$
  - Linear time-varying and invariant filters for RNNs such as S4, Mamba.

Main Takeaways

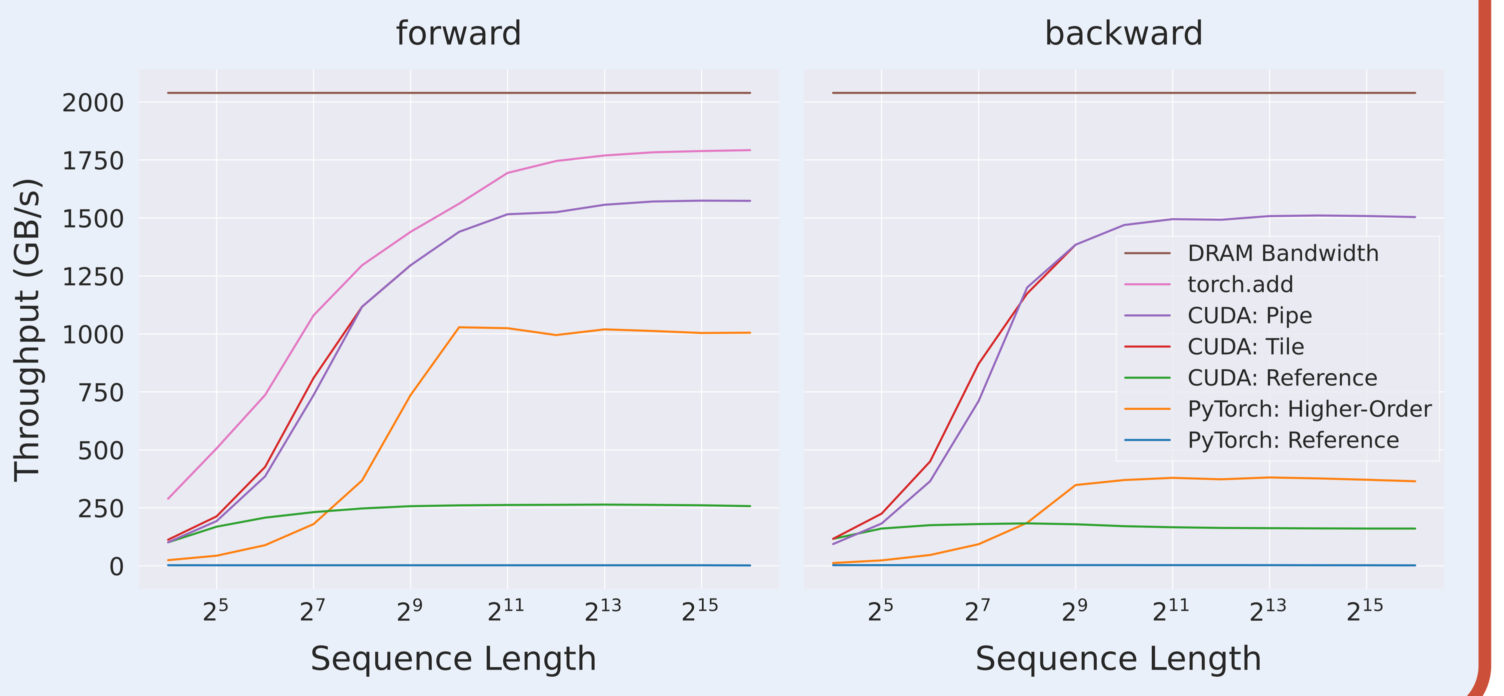
This blog post revisits **linear recurrences** for deep learning to

- provide intuitions for computational structure of linear RNNs
- facilitate research with minimal PyTorch/CUDA implementations

The **abstraction** of linear recurrences is intriguing because

- it acts as a generalization of cumulative sums/products and filters
- it supports dynamic recurrent computation and gating mechanisms
- the sequential overhead is almost negligible compared to `torch.add`

But: recent linear RNNs trade it for larger matrix hidden state sizes



3. Parallelizing Linear Recurrences

- **Parallel Scan (Blelloch, 1989):** A general algorithm to parallelize recurrences of associative operators. Only useful as high-level description. ⇒ We instantiate it for **element-wise** linear recurrences.
- **Two threads:** Split x and c into two parts of length $L' = L/2$.
  1. Compute local recurrences $[\bar{y}_{0,l}]_{l=0}^{L'-1}$ (from 0) and $[\bar{y}_{L',l}]_{l=L'}^{L-1}$ (from L').
  2. Return $y_l = \bar{y}_{0,l}$ for the first thread, then for the second combine:

$$y_l = \underbrace{\left(\sum_{k=0}^{L'-1} \tilde{c}_{k,L'-1} \cdot x_k \right)}_{=y_{L'-1}} \cdot \tilde{c}_{L'-1,l} + \underbrace{\sum_{k=L'}^{l} \tilde{c}_{k,l} \cdot x_k}_{=\bar{y}_{L',l}} \quad \text{for } L' \leq l < L.$$

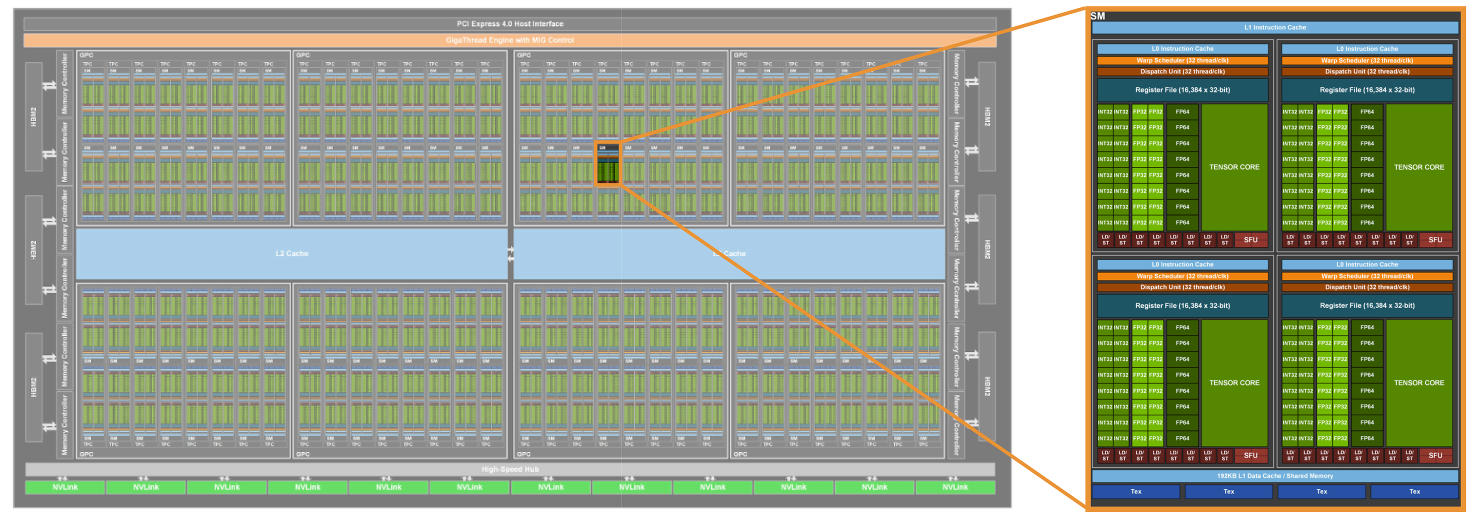
This requires communication of $y_{L'-1}$ and computing $\tilde{c}_{L'-1,l}$ for $L' \leq l$:

```
y[:L1] = linrec(x[:L1], c[:L1]) # thread 1
y[L1:] = linrec(x[L1:], c[L1:]) # thread 2
y[L1:] += y[L1 - 1] * cumprod(c[L1:]) # update thread 2
```

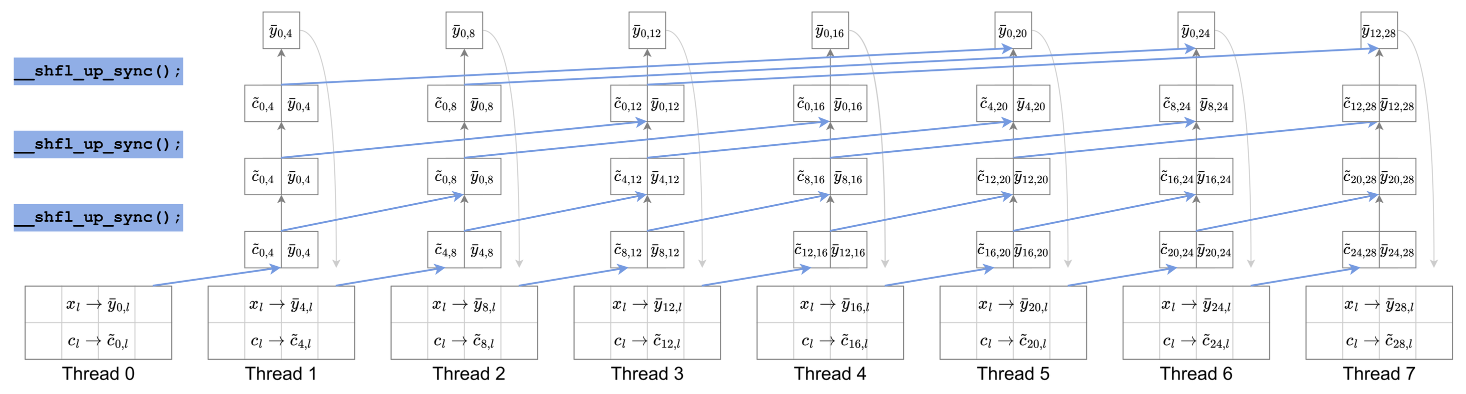
- **T threads:** Split x and c into parts of length $L' = L/T$
  1. Compute local recurrences in $O(L/T)$ sequential steps.
  2. Communicate transition elements $y_{L' \cdot t_{id} - 1}$ in a tree like structure. The depth of the tree and the number of sequential steps is in $O(\log T)$.
  3. Combine transition with output elements in $O(L/T)$ steps.
- **Complexity:** reduces from $O(L)$ to $O(L/T + \log T)$ sequential steps. ⇒ For $T = L$ threads this recovers the fully parallel scan in $O(\log L)$.

4. Mapping onto GPU Architectures

- **CUDA Background:**
  - ∼ 128 Streaming Multiprocessors (SMs) per GPU perform independent computation (e.g. per batch and channel for elementwise operations).
  - up to 2048 threads per SM share ∼ 128 CUDA cores, 65536 registers, and ∼ 256 shared memory to perform coordinated computation.



- **Tiled Processing:** up to 1024 threads process up to 16 elements.



- **Pipelined Processing:** sequentially load and process tiles of static size

5. Tuning & Benchmarking

- **Implementations:** we provide reference implementations
  - in PyTorch: sequential and via higher-order operation
  - in CUDA: sequential, tiled, and piped with PyTorch interface
- **Configs:** we statically compile for $L' \in \{4, 8, 16\}$ and #threads $\in \{32, 64, 128, 256, 512, 1024\}$ and thus tile sizes up to 16384.
- **Tuning:** we measure runtime and throughput to determine the optimal config for sequences of length 16 to 65536. Surprisingly, tiles of sizes 512 or 1024 are typically the best performing. ⇒ it is more efficient to sequentially process tiles than to increase parallelism with huge tiles.
- **Benchmarking:** we compare throughput (GB/s) of our implementations
  - observe initial speed-ups from translating PyTorch for-loop to CUDA
  - PyTorch higher-order operation cannot reach beyond 1000 GB/s
  - CUDA implementations are only slightly slower than `torch.add`
  - linear recursions are memory bound, similar to `torch.add`

  Measurements with `torch==2.5.1` on an NVIDIA A100-SXM4-80GB.

6. Discussion

- **Sequential Overhead:** is negligible compared to fully element-wise `torch.add` because of memory bandwidth.
- **Memory-Bound Operations:** present an opportunity to perform computations 'for free' once the memory is transferred from RAM/HBM.
- **Limitations:** recurrence needs to be element-wise on small hidden states
- **Open Questions:** how to increase expressivity per transferred byte?
  - is state-expansion the most efficient way to enhance memory in RNNs?
  - how to efficiently design dense linear RNNs with increased expressivity?
  - where to re-introduce non-linearities in order to improve expressivity?
  - which parametrizations are able to learn long-range interactions?

In conclusion, linear recurrences present a simple mathematical object with surprising modeling expressivity and computational opportunities.

Code & Contact

A simple CUDA extension template for PyTorch prototyping is available.

`github.com/safelix/linrec`
`felix.sarnthein@tue.ellis.eu` `@flxsa.bsky.social`