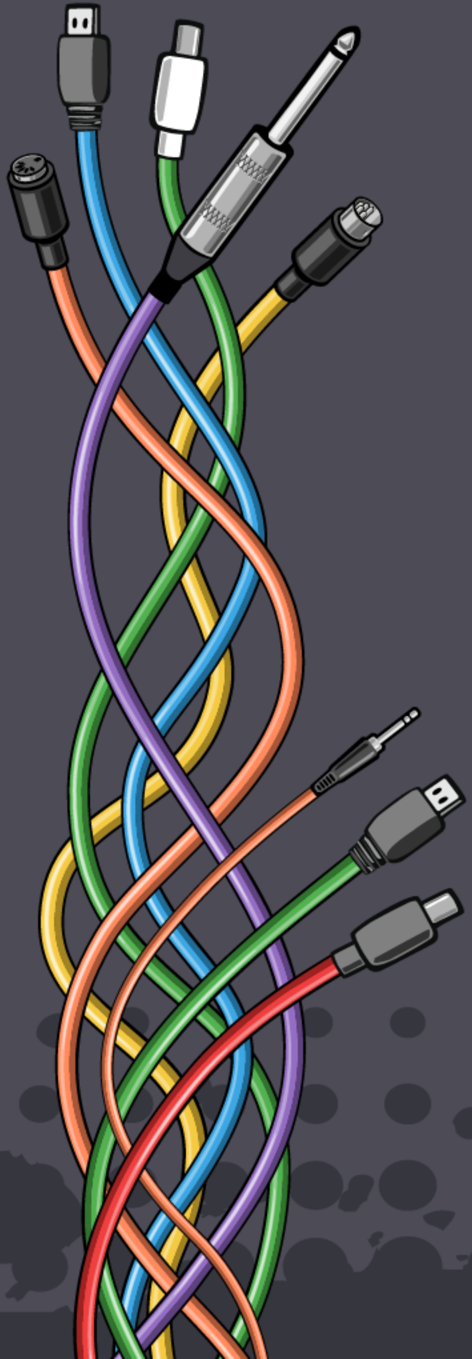


PHILTORCH

ACCELERATING AUTOMATIC DIFFERENTIATION OF DIGITAL FILTERS IN PYTORCH

CHIN-YUN YU

ADC25 Bristol

About me

- Yu, Chin-Yun （尤靖允）
- PhD student in AI & Music at Queen Mary University of London
- Research interests
  - Music Information Retrieval
  - Voice synthesis
  - Audio Effects Modelling
  - Binaural Audio
- https://iamycy.github.io/

Queen Mary
University of London

UK Research
and Innovation

Sources

- Publications
  - “Singing voice synthesis using differentiable LPC and glottal-flow-inspired wavetables”, ISMIR 2023
  - “Differentiable all-pole filters for time-varying audio systems”, DAFx24
  - “DiffVox: A differentiable model for capturing and analysing vocal effects distributions”, DAFx25
- Blog posts
  - “Introduction to Differentiable Audio Synthesiser Programming”, ISMIR 2023 Tutorial
  - “Notes on Differentiable TDF-II Filter”, personal blog
  - “Block-based Fast Differentiable IIR in PyTorch”, personal blog

Agenda

- What are differentiable filters
- Efficiency problem in PyTorch
- Proposed solution: PhilTorch
- Benchmarks
- Tips for efficient usage

What Are Digital Filters?

- In the Z domain

$$H(z) = \frac{b_0 + b_1 z^{-1} + \cdots + b_M z^{-M}}{1 + a_1 z^{-1} + \cdots + a_M z^{-M}}$$

- In the time domain

$$\begin{aligned} y(n) = \; & b_0 x(n) + b_1 x(n-1) + \cdots + b_M x(n-M) \\ & - a_1 y(n-1) - \cdots - a_M y(n-M) \end{aligned}$$

Usage

- Low/high/all-pass filters
- Shelving filters
- Peak filters
- Equalisers
- Subtractive synthesiser
- Phaser
- Linear prediction
- .etc

Differentiable Use Cases

- Differentiable Digital Signal Processing (DDSP)
  - Interpretable structure
  - strong inductive bias → highly efficient systems
- e.g.
  - Grey-box virtual analogue modelling
  - Synthesiser parameter retrieval

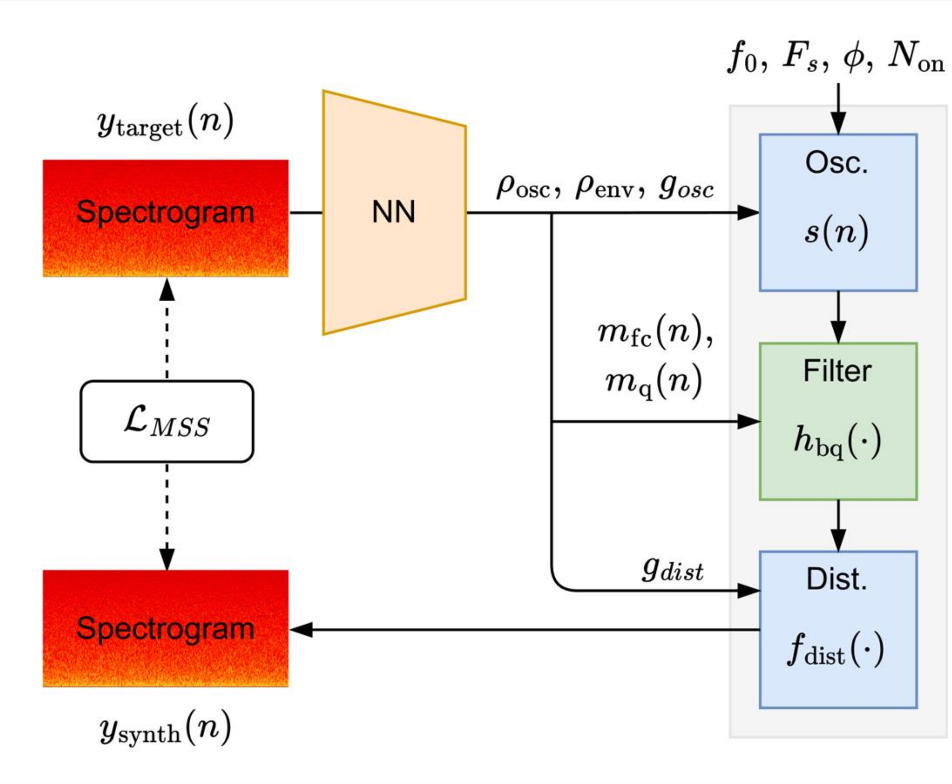


Fig. 3 from "Differentiable all-pole filters for time-varying audio systems"

Automatic Differentiation (AD) of Filters

- Differentiable everywhere? ✓
- Fast computation in PyTorch? ✗
- Problems
  - No native ops for recursions/autoregression

torch.nn.functional.conv1d

$$y(n) = \boxed{b_0 x(n) + b_1 x(n-1) + \cdots + b_M x(n-M)}$$

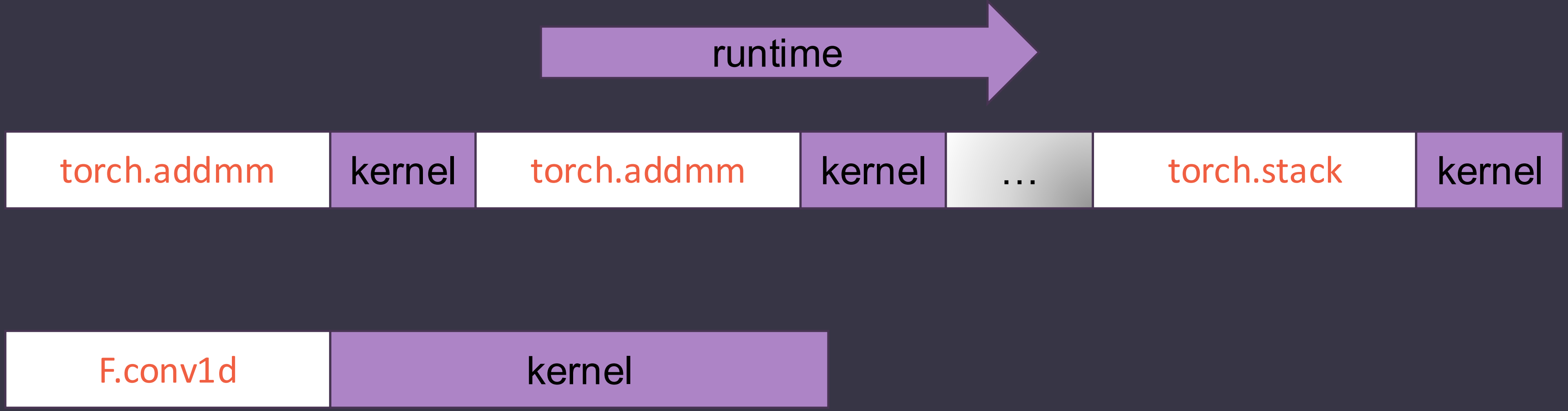
$$\boxed{-a_1 y(n-1) - \cdots - a_M y(n-M)}$$

torch.* ?

Naïve AD of Recursions

```
h = zi
for xn in x.unbind(1):
    h = torch.addmm(xn, h, AT)
    results.append(h if out_idx is None else h[:, out_idx])
output = torch.stack(results, dim=1)
```

philtorch/lti/ssm.py, L127-131



PhilTorch Φ🔥

- A Python package for PyTorch
- Fast computation of linear discrete time filtering
- Same numerical robustness as SciPy
- Reverse- (and forward) mode AD
- Linear time-invariant filters
- Parameter-varying (time-varying) filters

```
pip install philtorch
```

Code

Things To Be Covered

- The state-space formulation
- Custom Ops for recursion
- Backpropagation
- Parallelisation

The State-Space Model (SSM)

… of a direct form (DF) II filter.

$$\boxed{\mathbf{v}(n+1) = \mathbf{A}\mathbf{v}(n) + \mathbf{B}x(n)}$$

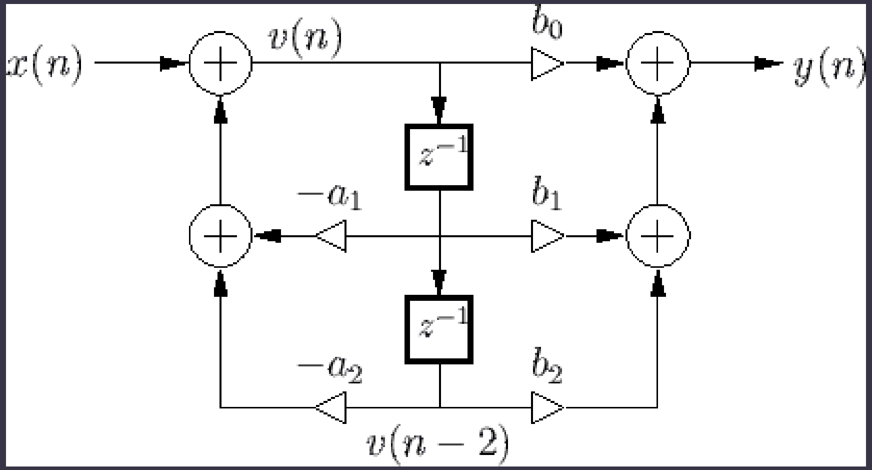
$$y(n) = \mathbf{C}^\top \mathbf{v}(n) + Dx(n)$$

$$\mathbf{A} = \begin{bmatrix} -\mathbf{a} & \\ \mathbf{I} & \mathbf{0} \end{bmatrix}, \qquad \mathbf{B} = \begin{bmatrix} 1 & \cdots & 0 \end{bmatrix}$$

$$\mathbf{C} = \begin{bmatrix} b_1 - b_0 a_1 & \dots & b_M - b_0 a_M \end{bmatrix}, \qquad D = b_0$$

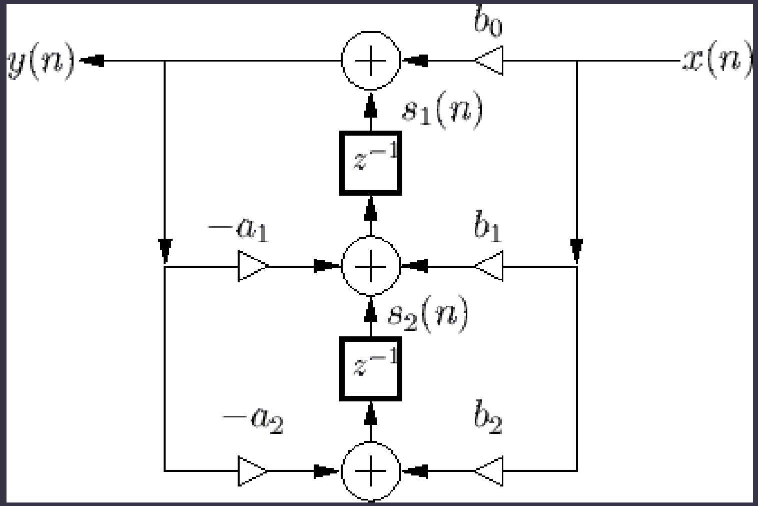
Different Direct Forms

- DF-II
  - Used in TorchAudio



$$H(z) = D + C^\top (zI - A)^{-1} B$$

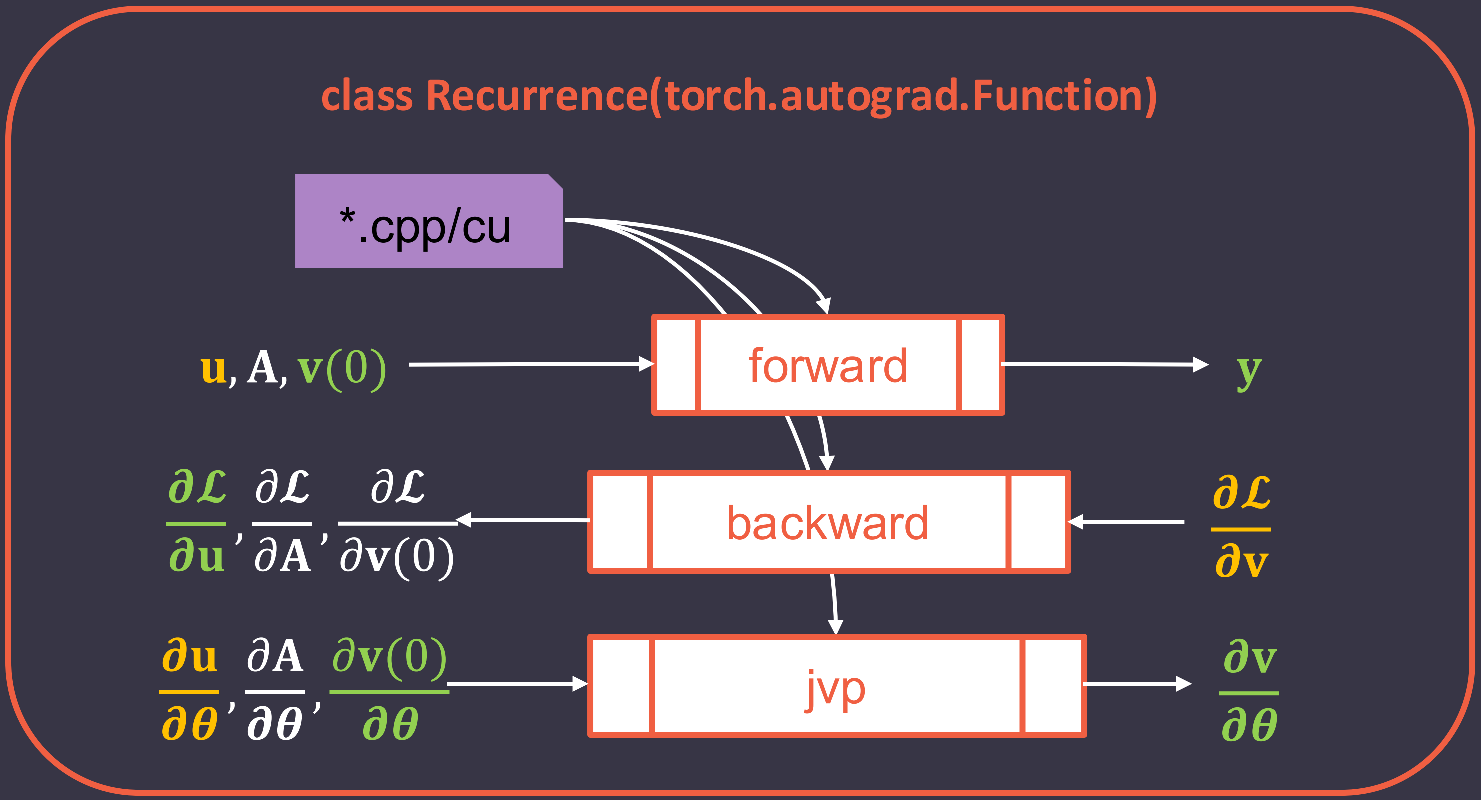
- Transposed DF-II (TDF-II)
  - Used in SciPy



$$H(z) = D + B^\top (zI - A^\top)^{-1} C$$

Image source: Julius Smith , "Introduction to Digital Filters with Audio Applications".

Writing Custom Operators in PyTorch



Output: Green
Input: Orange

$$
\begin{aligned}
&\mathbf{v}(n+1) \\
&= \mathbf{A}\mathbf{v}(n) + \mathbf{B}x(n) \\
&= \mathbf{A}\mathbf{v}(n) + \mathbf{u}(n)
\end{aligned}
$$

Backpropagation

…is a reverse-time recurrence with transposed **A.**

- Gradients for the input

$$\frac{\partial \mathcal{L}}{\partial \mathbf{u}(\mathrm{n})} = \left(\mathbf{A}^{\mathsf{T}} \frac{\partial \mathcal{L}}{\partial \mathbf{u}(n+1)}^{\mathsf{T}} + \frac{\partial \mathcal{L}}{\partial \mathbf{v}(n+1)}^{\mathsf{T}} \right)^{\mathsf{T}}$$

- Gradients for coefficients & initial condition

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \sum_n \mathbf{v}(n) \frac{\partial \mathcal{L}}{\partial \mathbf{u}(n)}, \qquad \frac{\partial \mathcal{L}}{\partial \mathbf{v}(0)} = \frac{\partial \mathcal{L}}{\partial \mathbf{u}(0)} \mathbf{A}$$

Parallel implementation

Non-parallelisable and sequential ☹

$$\mathbf{v}(n+1) = \mathbf{A}(\mathbf{A}(\mathbf{A}(\dots) + \mathbf{u}(n-2)) + \mathbf{u}(n-1)) + \mathbf{u}(n)$$

Parallelisable ☺

- Parallel associative scan

$$(\mathbf{U}, \mathbf{x}) \oplus (\mathbf{V}, \mathbf{z}) \mapsto (\mathbf{VU}, \mathbf{Vx} + \mathbf{z})$$

$$(\mathbf{0}, \mathbf{v}(n+1)) = (\mathbf{0}, \mathbf{v}(0)) \oplus (\mathbf{A}, \mathbf{u}(0)) \oplus (\mathbf{A}, \mathbf{u}(1)) \oplus \cdots \oplus (\mathbf{A}, \mathbf{u}(n))$$

Diagonalised SSM

$$\tilde{\mathbf{v}}(n+1) = \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_M \end{bmatrix} \tilde{\mathbf{v}}(n) + \mathbf{P}^{-1}\mathbf{u}(n)$$

$$\mathbf{v}(n) = \mathbf{P}\tilde{\mathbf{v}}(n), \qquad \mathbf{A} = \mathbf{P} \begin{bmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_M \end{bmatrix} \mathbf{P}^{-1}$$

- Matrix multiplication $\rightarrow$ elementwise multiplications
- Equals to M first-order filters run in parallel

Alternative Solution w/o Custom Ops

- Sample unrolling
  - Andres Viso, “Implementing real-time Parallel DSP on GPUs”, ADC 2022.
- Pros: very efficient using torch.matmul
- Cons: still sequential

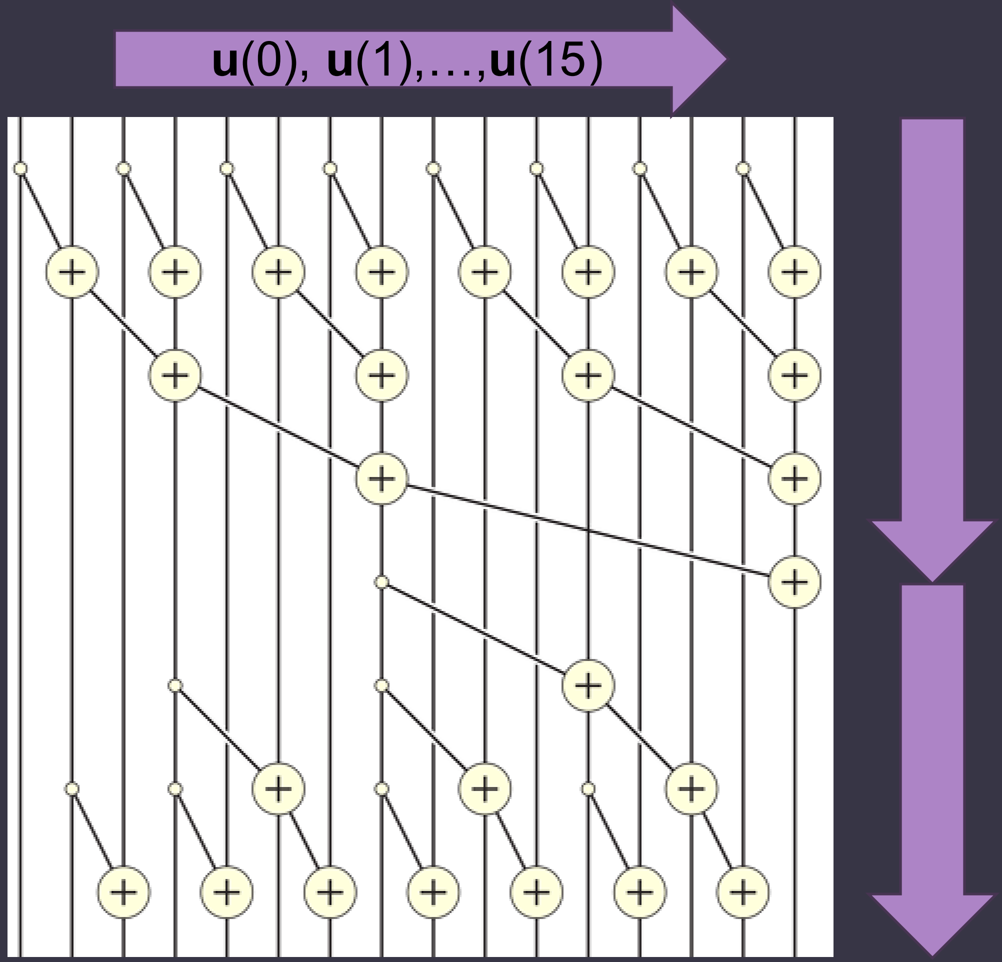
$$\begin{bmatrix} \mathbf{v}(n+1) \\ \vdots \\ \mathbf{v}(n+N-1) \\ \mathbf{v}(n+N) \end{bmatrix} = \begin{bmatrix} \mathbf{A} \\ \vdots \\ \mathbf{A}^{N-1} \\ \mathbf{A}^{N} \end{bmatrix} \mathbf{v}(n) + \begin{bmatrix} \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{A} & \mathbf{I} & \cdots & \mathbf{0} \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}^{N-1} & \mathbf{A}^{N-2} & \cdots & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u}(n) \\ \mathbf{u}(n+1) \\ \vdots \\ \mathbf{u}(n+N) \end{bmatrix}$$

Extend to Offline Filtering

1. Compute by hopping through $\mathbf{v}(N), \mathbf{v}(2N) \ldots$
   - $\mathbf{v}(n+N) = \mathbf{A}^N \mathbf{v}(n) + \begin{bmatrix} \mathbf{A}^{N-1} & \mathbf{A}^{N-2} & \cdots & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{u}(n) \\ \mathbf{u}(n+1) \\ \vdots \\ \mathbf{u}(n+N-1) \end{bmatrix}$
2. Compute the rest in parallel:
   - $\begin{bmatrix} \mathbf{v}(n+1) \\ \mathbf{v}(n+2) \\ \vdots \\ \mathbf{v}(n+N-1) \end{bmatrix} = \begin{bmatrix} \mathbf{A} & \mathbf{I} & \mathbf{0} & \cdots & \mathbf{0} \\ \mathbf{A}^2 & \mathbf{A} & \mathbf{I} & \cdots & \mathbf{0} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{A}^{N-1} & \mathbf{A}^{N-2} & \mathbf{A}^{N-3} & \cdots & \mathbf{I} \end{bmatrix} \begin{bmatrix} \mathbf{v}(n) \\ \mathbf{u}(n) \\ \mathbf{u}(n+1) \\ \vdots \\ \mathbf{u}(n+N-2) \end{bmatrix}$

Note: We can apply step 1 repeatedly (it's a recursive definition).

Assemble Parallel Associative Scan



Step 1 (4 times, N=2)

$$O(n) \longrightarrow O(\log(n))$$

Step 2 (4 times)

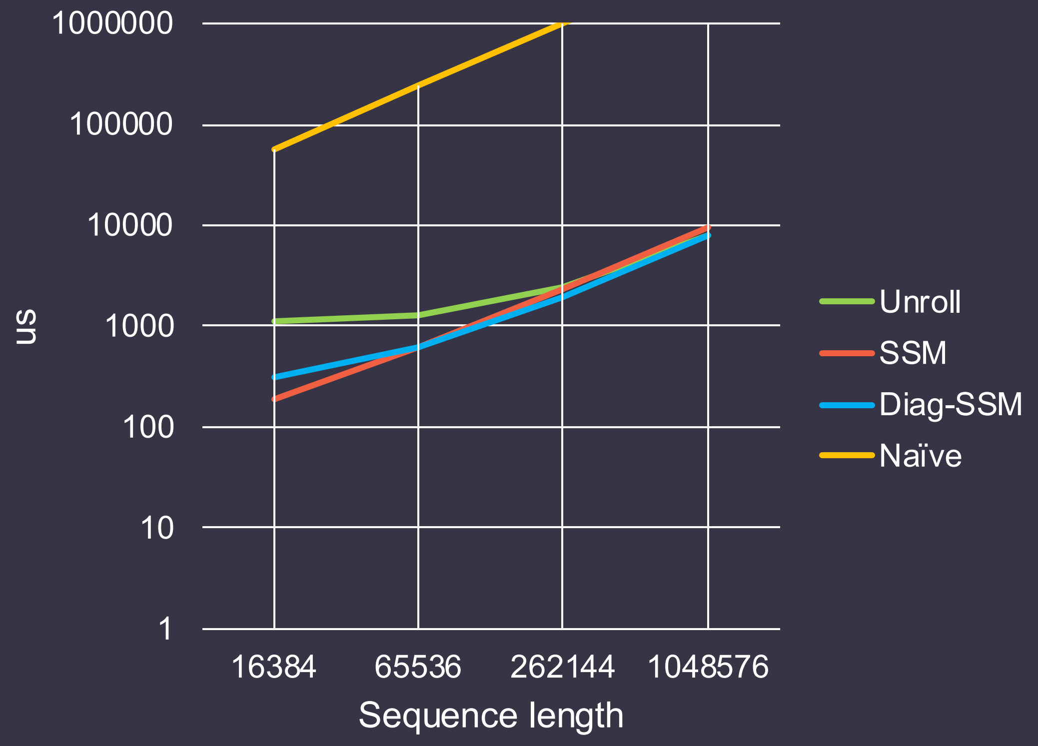
Image source: https://en.wikipedia.org/wiki/Prefix_sum

Take away

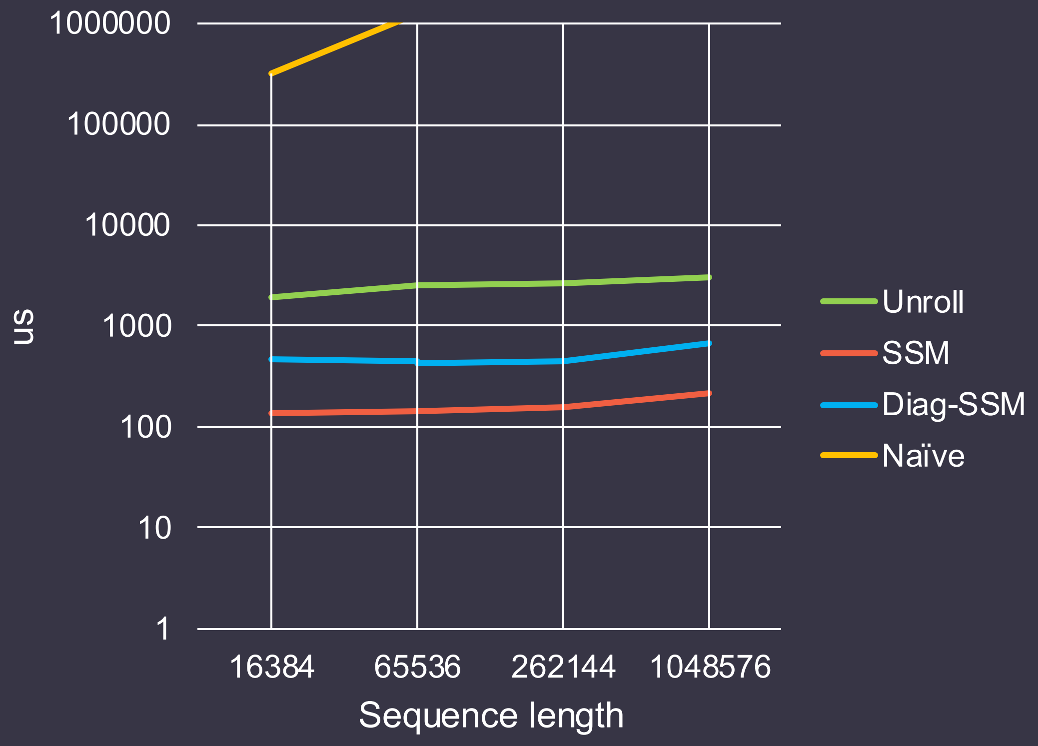
1. > 1000 small kernels → write custom extension
2. Associative? → parallelisable!
3. You can do both

Benchmarks (M=2, forward)

Intel i7 7700K, 1 thread

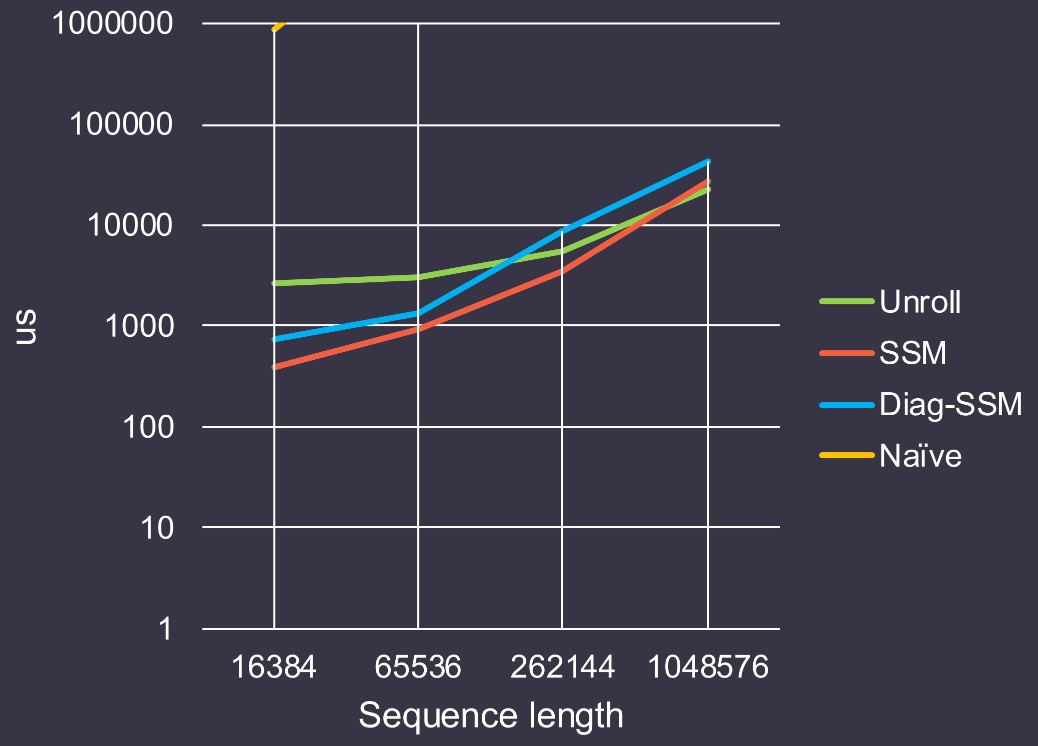


RTX 5060 Ti

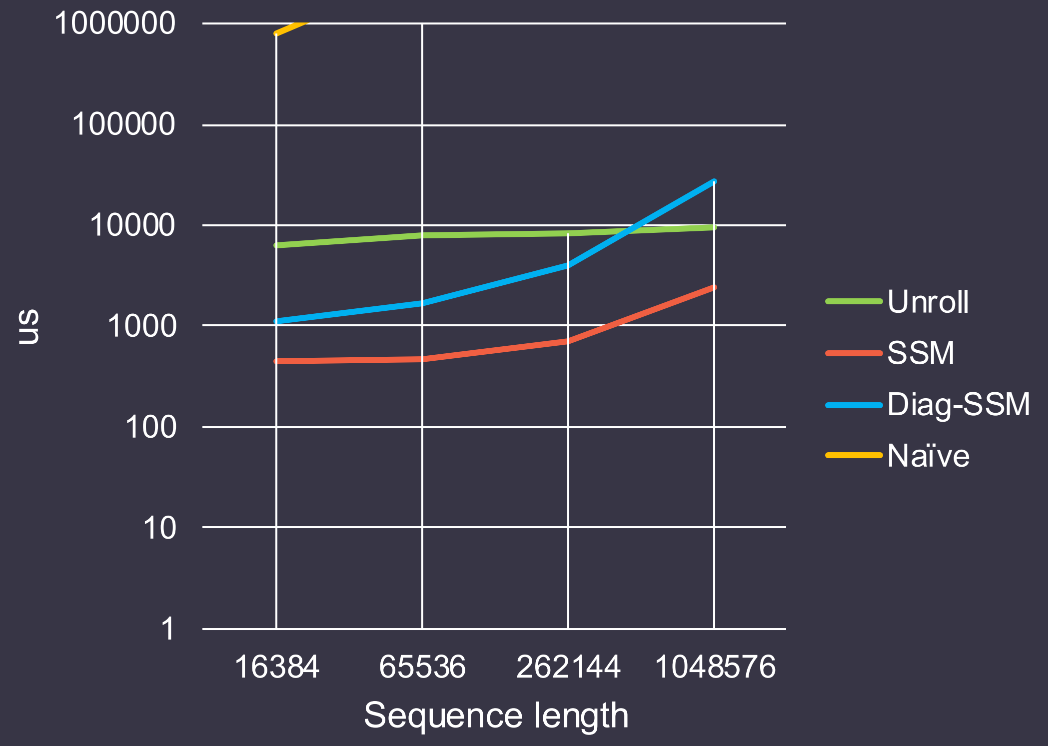


Benchmarks (M=2, backpropagation)

Intel i7 7700K, 1 thread



RTX 5060 Ti



Tips For Using PhilTorch

- PhilTorch has fast kernel for M = 1 or 2.
  - Sample unrolling for M > 2.
- Decompose your filter into second-order sections (SOS)
  - Cascaded SOS 👍
  - Parallel SOS 👍 👍
- Check out the example!

Estimate lowpass filter