Liquid AI Treehack Challenges

1 Operator Optimization Champion

Focus on our Track 2 challenge by optimizing some of key operators for next-gen architectures—convolutions and recurrences—on the Samsung Galaxy S24. We'll measure both peak memory and latency using our profiling scripts. The most optimized implementation (memory, latency across given input sizes) that still delivers accurate results will be crowned the champion.

1.1 Background

We consider the following classes of sequence processing core primitives: *convolutions* and *linear recurrences*. These core primitives are designed such to manipulate sequences of variable length (yet consistent between input and outputs) and fixed number of channels.

Convolutions. Let "$*$" denote the convolution operator. In signal processing and deep learning alike, one often encounters the *causal* linear convolution of a filter h (which may extend indefinitely) with an input sequence x of length ℓ and d channels:

$$y[i,n] = \sum_{j=0}^{i} h[i-j,n]x[j,n] \quad \text{for all} \quad i \in [\ell], n \in [d].$$

For *short* filters, an effective approach is to construct the sparse Toeplitz matrix $T[i,j,n] = h[i-j,n]$ from the filter and multiply it by x. In essence, one "unrolls" the filter into overlapping segments—each of size equal to the sequence length—so that a standard matrix multiplication yields the convolution result[1]. However, when the filter grows longer (or when sequence lengths become large), it is often more efficient to apply the convolution in the frequency domain using the *Fast Fourier Transform* (FFT). By transforming both the signal and filter to the frequency domain, the convolution becomes elementwise multiplication (subject to zero-padding to handle boundaries), and then an inverse FFT recovers the time-domain result.

In PyTorch, one can use "nn.Conv1d" to handle many of these details automatically. Internally, for short filters, PyTorch may use a direct method, forming or conceptually approximating the Toeplitz matrix and leverage fast matrix-matrix operations on optimized backends. For very long filters, libraries can switch to FFT-based implementations.

```
def matmul_convolution(x: torch.Tensor,
                       h: torch.Tensor) -> torch.Tensor:
  T = toeplitz(h)
  y = T @ x
  return y

def fft_convolution(x: torch.Tensor,
                    h: torch.Tensor) -> torch.Tensor:
    N = x.shape[-1] + h.shape[-1]
    X = rfft(x, n=N)
    H = rfft(h, n=N)
    Y = X * H
```

[1]This is mathematically equivalent to unrolling the input.

```
    y = irfft(Y, n=N)
    y = y[..., :x.shape[-1]]
    return y
```

For an FFT-based approach on longer kernels, one could explicitly:
1. Zero-pad both the filter and the input signal (usually to the next power of 2 for efficiency).
2. Compute their FFTs (of a suitable length).
3. Multiply the spectra elementwise.
4. Apply the inverse FFT.
5. Trim the result to match the desired output length.

Either strategy yields the same mathematical convolution defined above; the choice depends on practical considerations (filter size, sequence length, and available hardware optimizations).

Hints:
- Specialize your implementation to input and filter shapes e.g., direct methods for short convs (materialize the sparse Toeplitz matrix, or use the direct for loop) and fft for long convolutions

Linear Recurrences. These are the staple of classic signal processing and control systems and can also be used as building blocks for some types of model architectures. We consider a heavily simplified model of the form

$$y[i,n] = a[i,n]y[i-1,n] + x[i,n] \quad \text{for } i \in [\ell], n \in [d]$$

given the initial state $y[-1,n]$.

```
def linear_recurrence(x: torch.Tensor,
                 y_init: torch.Tensor,
                      a: torch.Tensor) -> torch.Tensor:
    """
    Args:
        x      (Tensor): Input sequence of shape (ell, d).
        y_init (Tensor): Initial state of shape (d,).
        a      (Tensor): Time-varying coefficients of shape (ell, d).
    Returns:
        y      (Tensor): State sequence of shape (ell, d), starting from y0.
    """
    ell, d = x.shape
    # Allocate output sequence
    y = torch.zeros((ell+1, d), dtype=x.dtype, device=x.device)
    # Set initial state
    y[0] = y_init
    # Iterate through timesteps
    for i in range(1, ell):
        for n in range(d):
          y[i,n] = a[i,n] * y[i-1,n] + x[i,n]
    return y[1:]
```

Hints:
- Look at different algorithms for linear recurrences: linear scan, parallel scan, tiled scan algorithms for recurrences.
- the convolution can also be written as a linear recurrence with state size equal to `filter length - 1` (companion form).