



CONFERENCE

—NORTH AMERICA 2026—

From Bytes to Tensors: Efficient Media Processing with PyTorch in 2026

Nicolas Hug (META), Scott Schneider (META)

Plan

TorchCodec

decode & encode

TorchVision

image & video transforms

TorchAudio

audio transforms

Part 1: The landscape

Where these three libraries landed: who does what, and how they fit together

Part 2: Going fast

How to make your media pipeline fast with these three libraries, using existing and new features

All benchmarks: github.com/NicolasHug/ptc-talk-2026

TorchCodec: all media decoding & encoding

Images

Decode: JPEG, PNG, WebP, GIF, AVIF, HEIC

Encode: JPEG, PNG
CPU and **CUDA**

Video

Decode and encode

All **FFmpeg** codecs

CPU and **CUDA**

HDR, samplers

Audio

Decode and encode

All **FFmpeg** codecs

Resampling, remixing

Fast `WavDecoder`

TorchCodec: all media decoding & encoding

Images

Decode: JPEG, PNG, WebP, GIF, AVIF, HEIC

Encode: JPEG, PNG
CPU and **CUDA**

Video

Decode and **encode**

All **FFmpeg** codecs

CPU and **CUDA**

HDR, samplers

Audio

Decode and **encode**

All **FFmpeg** codecs

Resampling, remixing

Fast `WavDecoder`

New in 0.17: low-level decoding APIs → more in Part 2

- multi-threaded pipelines
- audio + video in one pass
- raw YUV access & custom kernels
- keyframe positions
- container-free decoding

TorchVision and TorchAudio: performant transforms

TorchVision

Transforms

resize, crop, flip, color jitter, AutoAugment, ...
images, videos, boxes, masks

~~Decoding & encoding~~

→ moved to TorchCodec

Models, datasets

maintenance mode

TorchAudio

Transforms

mel spectrogram, MFCC, resample, masking, ...

~~Decoding & encoding~~

→ moved to TorchCodec

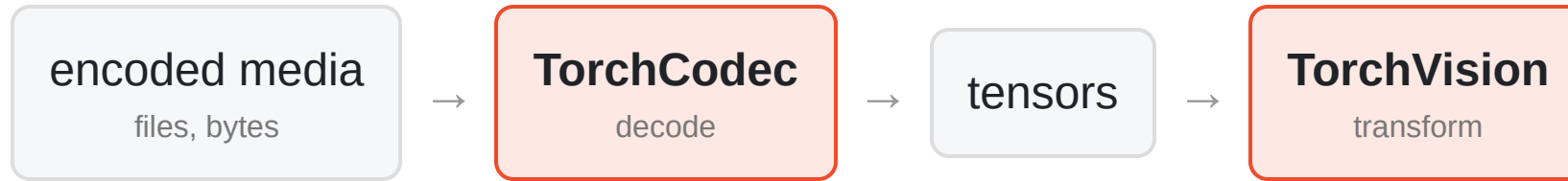
Models, datasets, pipelines

maintenance mode

Still on `torchvision.io` / `torchaudio` decoding? → TorchCodec [migration guide](#)

How they fit together

TorchCodec's **output** is TorchVision's **input**



```
import torch
from torchcodec.decoders import VideoDecoder
from torchvision.transforms import v2

clip = VideoDecoder("video.mp4")[10:20] # uint8 tensor, (10, C, H, W)

transform = v2.Compose([
    v2.RandomResizedCrop(224),
    v2.ToDtype(torch.float32, scale=True),
])
batch = transform(clip)
```

Same for audio: AudioDecoder → TorchAudio transforms (e.g. MelSpectrogram)

Encoding goes the other way: VideoEncoder, AudioEncoder, ...

All three libraries are ABI stable!

- A release **keeps working with the PyTorch versions that come after it**

BEFORE

...

TorchAudio 2.9 → PyTorch 2.9 only
TorchAudio 2.10 → PyTorch 2.10 only

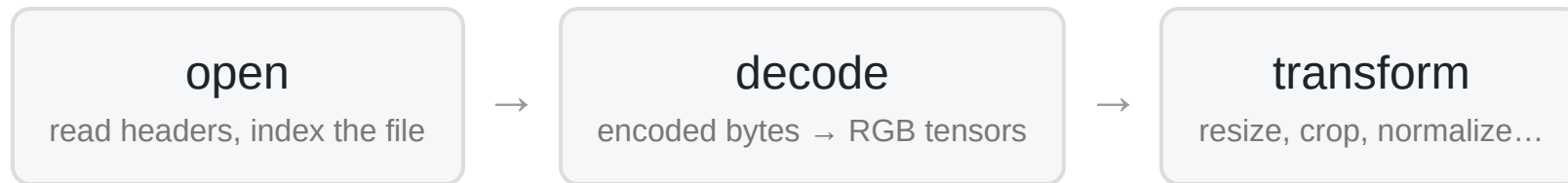
NOW

TorchAudio 2.11 → PyTorch 2.11, 2.12, ...

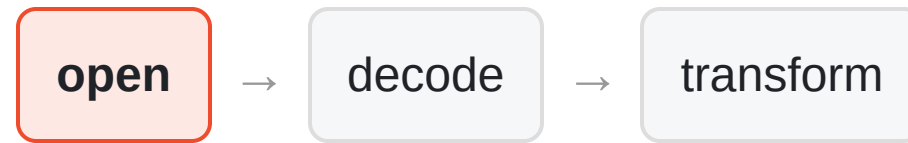
- No rebuild for every PyTorch release → release cadences no longer match PyTorch's, **and that's expected**

Earlier talk: *Clearing the Path Towards an ABI Stable PyTorch C++ Extension Ecosystem* - Sean McGovern, Chris Leonard, Jane Xu

Part 2: Going fast



Every stage can be the bottleneck. Let's go through them in order.



Opening the video

Open from anywhere

```
decoder = VideoDecoder("video.mp4") # path, URL, bytes, tensor of bytes
```

But also **file-like objects**. Don't download the whole file!

```
import fsspec

with fsspec.open("https://.../video.mp4") as f: # or s3://, gs://, ...
    decoder = VideoDecoder(f, seek_mode="approximate")
    frame = decoder[1000]
```

Only downloads the bytes around frame `1000` .

Exact vs approximate seek mode

```
decoder = VideoDecoder("long_video.mp4", seek_mode="approximate")
```

A tradeoff: **seek accuracy** vs **decoder creation speed**

	"exact" (default)	"approximate"
Creation	scans the whole file: slow on long videos	reads the header: fast
Seeking	frame-accurate	may be slightly off on some files

 **Benchmark:** remote 10 min 720p video (209 MB), create decoder + decode 16 frames

"exact" : **236 MB** downloaded

"approximate" : **28 MB** downloaded

Custom frame mappings: *exact and fast*

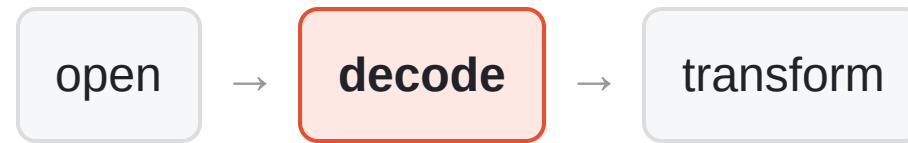
Pre-compute the index offline, reuse it every time you decode.

```
ffprobe -i video.mp4 -select_streams 0 -show_frames \
        -show_entries frame=pts,duration,key_frame -of json > mappings.json
```

```
with open("mappings.json") as f:
    decoder = VideoDecoder("video.mp4", custom_frame_mappings=f)
```

The best of both worlds:

- Seek accuracy like "exact"
- Creation speed like "approximate"



Decoding

GPU decoding: `device="cuda"`

```
decoder = VideoDecoder("video.mp4", device="cuda")
frames = decoder.get_frames_at([0, 10, 20]) # already on the GPU
print(decoder.cpu_fallback)                 # did we silently fall back to CPU?
```

Why it's (usually!) faster:

- **Fast raw decoding** on NVDEC, a dedicated hardware chip separate from the CUDA cores
- **Less PCIe traffic**: encoded packets are much smaller than decoded frames
- **YUV → RGB conversion**: expensive on CPU, extremely fast on GPU

 **Benchmark**: decode all 300 frames of a 10s video (laptop: RTX 4080)

	CPU	GPU	Speedup
1080p	1115 ms	379 ms	3×
4K	4408 ms	1446 ms	3×

On a warm cache!

GPU decoding: the NVDEC decoder cache

Creating a GPU decoder is expensive → TorchCodec **caches and reuses** them automatically

```
d1 = VideoDecoder("video1.mp4", device="cuda")    # creates a hardware decoder (slow)
...                                           # d1 goes out of scope → back to the cache
d2 = VideoDecoder("video2.mp4", device="cuda")    # same codec & resolution: reused (fast!)
```

What you need to do:

- **Nothing** by default: reuse is automatic
- More than 20 decoders in flight, or many resolutions? → `set_nvdec_cache_capacity(N)`

 **Benchmark:** create GPU decoder + decode 10 frames, 10s 1080p video (laptop: RTX 4080)

No cache	Warm cache	Speedup
272 ms	24 ms	11×

Matters even more when decoding many videos in parallel

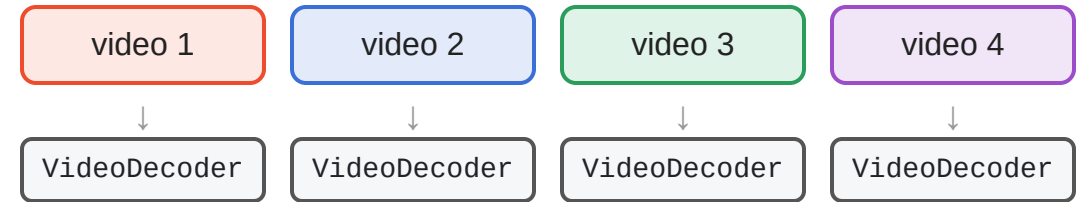
Parallel decoding: 4 strategies

① One long video: one decoder per chunk

video 1



② Many videos: one decoder per video



③ FFmpeg threads: decode upcoming frames ahead, into a buffer

```
VideoDecoder(path, num_ffmpeg_threads=4)
```

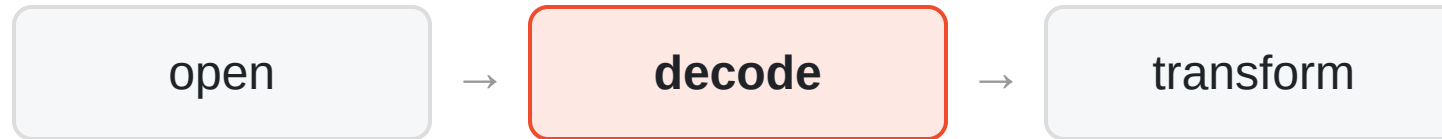
④ Low-level APIs: parallelize stage by stage. Next slides!

All strategies compose! ① and ② run on threads or processes: decoding releases the GIL.

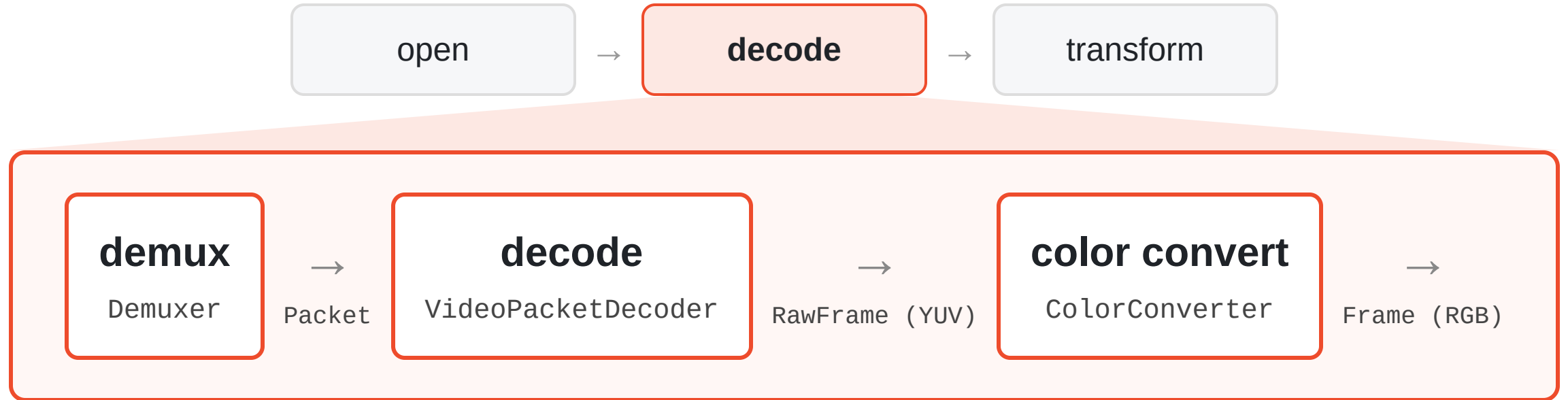
 **Benchmark:** decode all 18000 frames of a 10min 720p video (laptop CPU, 4 workers)

Sequential	① 4 threads	① 4 processes	③ 4 FFmpeg threads
32.7 s	8.7 s (3.7×)	9.3 s (3.5×)	15.3 s (2.1×)

Low-level decoding APIs: new in TorchCodec 0.17

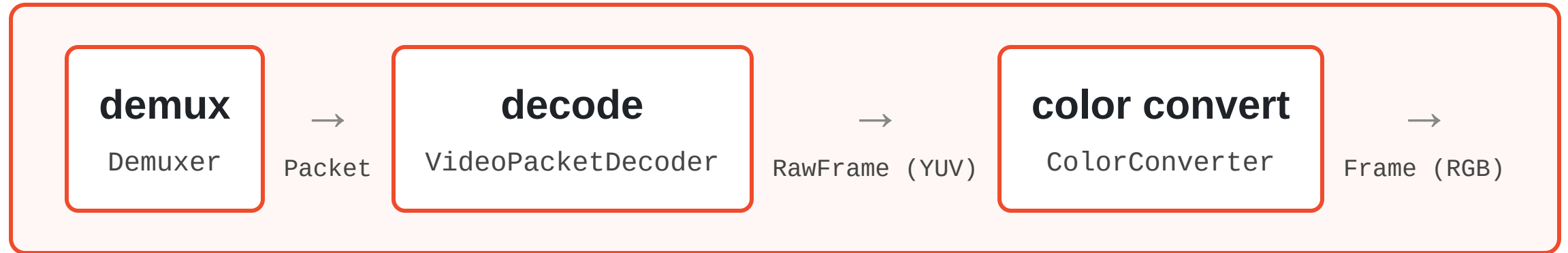


Low-level decoding APIs: new in TorchCodec 0.17



`VideoDecoder` : everything in one call. Low-level APIs: **more knobs to hill-climb on.**

Low-level APIs: the basic loop



```
from torchcodec.decoders import ColorConverter, Demuxer

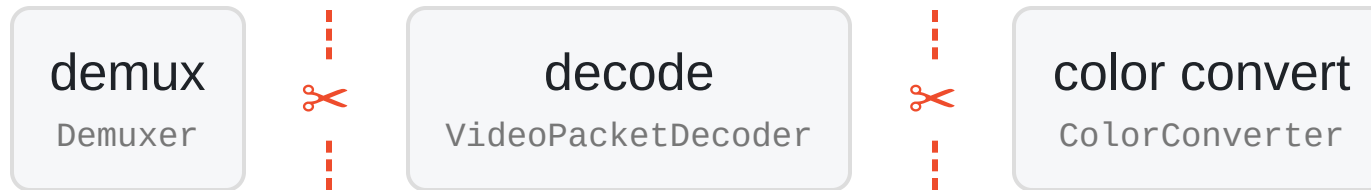
demuxer = Demuxer("video.mp4", streams="video")
packet_decoder = demuxer.streams[0].make_decoder()
color_converter = ColorConverter()

for packet in demuxer:
    for raw_frame in packet_decoder.decode(packet):
        frame = color_converter.convert(raw_frame)

# Same as VideoDecoder("video.mp4").get_all_frames()
```

Low-level APIs: multi-threaded pipelines

Each stage **releases the GIL** → cut the pipeline into threads at any stage boundary:

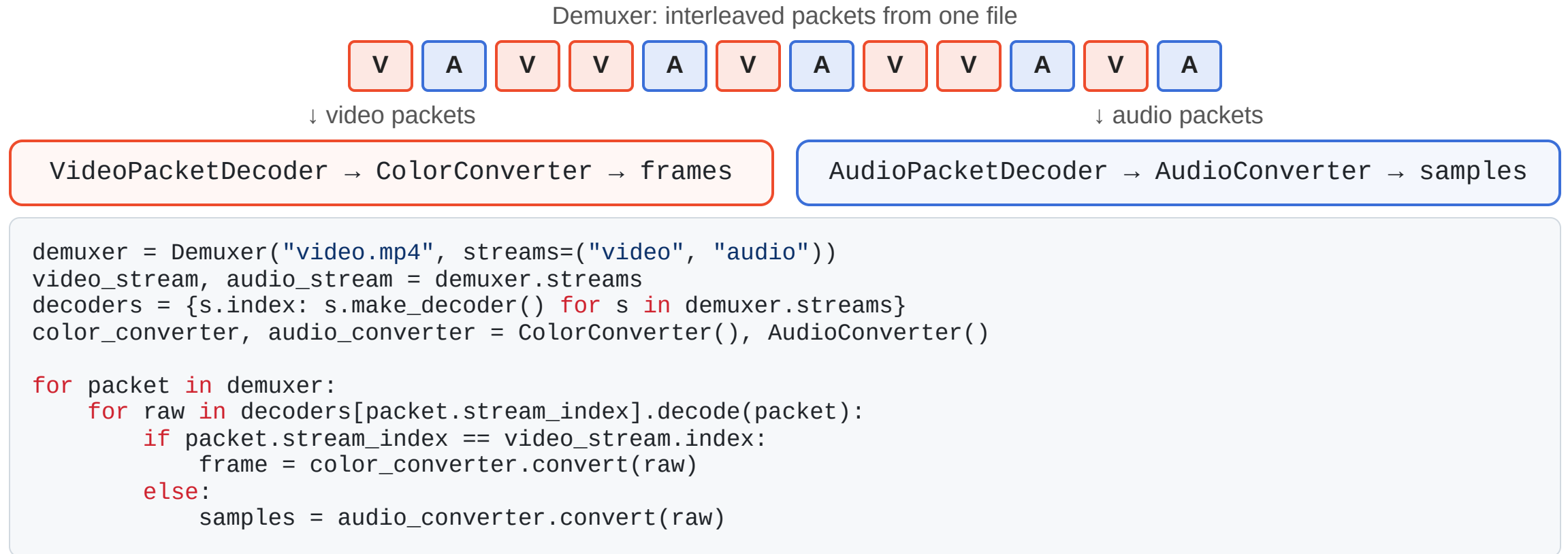


```
# prefetch(): run the upstream stages on a background thread, through a bounded queue
raw_frames = prefetch(decode(packet_decoder, demux(demuxer))) # producer
frames = color_convert(color_converter, raw_frames)           # consumer
```

 **Benchmark:** decode all 300 frames of a 10s 1080p video (laptop CPU)

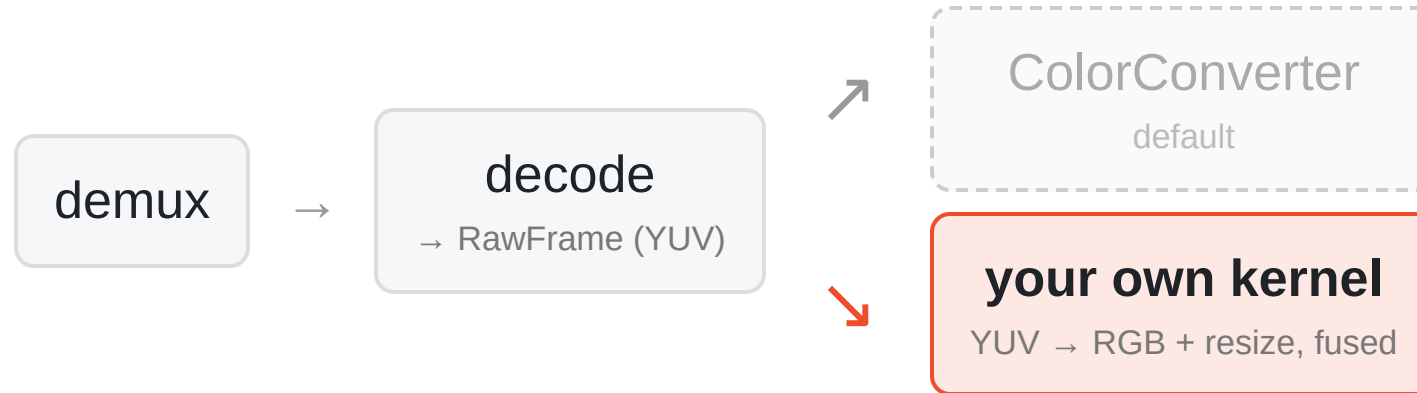
VideoDecoder	Low-level, sequential	Low-level, producer / consumer
1457 ms	1458 ms	940 ms (1.5×)

Low-level APIs: audio + video in one pass



- **Read the file once**, instead of once per stream
- Also works for several video streams or audio tracks: `streams="all"`

Low-level APIs: raw YUV frames



```
for raw_frame in packet_decoder.decode(packet):  
    Y, U, V = raw_frame.planes          # the decoder's own planes: no conversion, no copy  
    out = my_yuv_to_rgb_and_resize(Y, U, V, size=(224, 224))  # e.g. a Triton kernel
```

- **Fuse** color conversion with your transforms (e.g. resize) in one kernel, or train directly in YUV
- Or just train on YUV!
- 10-bit HDR → `uint16` planes, every bit intact

Low-level APIs: keyframe positions

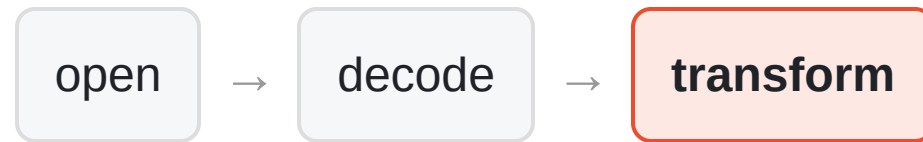
- Keyframes are the **cheapest frames to decode**
- Great for **samplers** and thumbnails: sample at or near keyframes for fast training data

```
demuxer = Demuxer("video.mp4")
index = demuxer.streams[0].scan()      # one pass over the packets, no decoding
key_frame_times = index.pts_seconds[index.key_frame_indices]

demuxer.seek(key_frame_times[3])      # lands right on a keyframe
```

Low-level APIs: what else they unlock

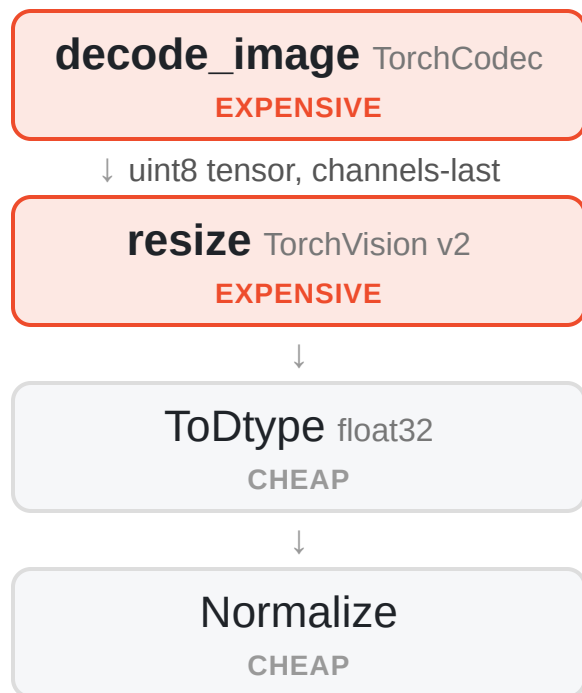
- **Live streams and pipes:** sources with no duration and no end
- **Decoding raw packet data**, without a demuxer or container: in progress, [tell us what you need!](#)



Transforming

(mostly images, but this applies to videos too!)

Fast image pipelines: just 3 rules



1. **Decode with TorchCodec:** `decode_image`
2. **Use** `torchvision.transforms.v2`
3. **Resize on** `uint8`, before converting to float

Why? You want **SIMD decoding** (`libjpeg-turbo`) and **SIMD resizing** (AVX2, NEON: needs `uint8`, channels-last tensors). That's what `decode_image` + `transforms.v2` give you.

TorchCodec + TorchVision: fast by default

```
from torchcodec.decoders import decode_image
from torchvision.transforms import v2

img = decode_image("image.jpg")          # ① decode with TorchCodec

transforms = v2.Compose([                # ② use v2
    v2.Resize(size=(224, 224)),           # ③ resize first, on uint8
    v2.ToDtype(torch.float32, scale=True), # then convert to float
    v2.Normalize(mean=..., std=...),
])
out = transforms(img)
```

 **Benchmark:** decode + resize + normalize a 1080p JPEG (laptop CPU, 1 thread)

PIL pipeline	PIL-SIMD pipeline	TorchCodec + TorchVision (above)
10.0 ms	4.4 ms	4.0 ms

- PIL-SIMD: x86 only (no NEON SIMD), and still pays the PIL → tensor copy.

Results are the same as PIL's!

- Use `resize(..., antialias=True)` (the default)
- vs PIL: **over 99% of pixel values are identical**, the others differ by **1 or 2** (out of 255)
- **Bitwise identical** to PIL-SIMD
- What matters for models: **consistent** interpolation mode and antialias between training and inference
- **NEW** in `TorchVision 0.27`: **Lanczos interpolation!** Alongside bilinear and bicubic. All SIMD.

Audio: WavDecoder

```
from torchcodec.decoders import WavDecoder  
  
samples = WavDecoder("audio.wav").get_all_samples()
```

- Reads WAV data directly, **bypassing FFmpeg** entirely

 **Benchmark:** create decoder + decode a WAV file (laptop CPU)

	AudioDecoder	WavDecoder	Speedup
5s clip , 16kHz mono	2.7 ms	0.05 ms	50×
10min , 48kHz stereo	202 ms	81 ms	2.5×
1s from a 10min file	16.7 ms	0.06 ms	280×

And on the way out: encoding

```
from torchcodec.encoders import Encoder

encoder = Encoder()
video = encoder.add_video(height=1080, width=1920, frame_rate=30, device="cuda")
audio = encoder.add_audio(sample_rate=44100, num_channels=2)

with encoder.open_file("output.mp4"):
    video.add_frames(frames)
    audio.add_samples(samples)
```

- Multiple video and audio streams in one container
- Mix CPU and CUDA video streams, feed data incrementally
- Simpler one-shot encoders too: VideoEncoder , AudioEncoder , JpegEncoder , PngEncoder

TorchCodec 1.0 is coming!

Tell us what you need

Feedback and feature requests: [TorchCodec](#) · [TorchVision](#) · [TorchAudio](#)

Docs and tutorials

- [TorchCodec docs](#) · [tutorials](#)
- [TorchVision transforms](#)
- [TorchAudio docs](#)

From this talk

- [Benchmarks](#)
- [Blog post: the media processing landscape](#)
- [TorchCodec performance tips](#)