

From MP4 to Tensor: Optimizing Multi-Camera Video Data Loading for PyTorch Training

Kyle Huang, Mihai Aldén · NVIDIA, Alpamayo ML Infrastructure Team

1 Why decode live

Large-scale training can span **tens of petabytes** and **thousands of GPUs**, with six camera streams per sample. Pre-decoding or caching frames would create another dataset-scale copy and preparation pipeline. Instead, we decode the original MP4s live during training.

Tens of PB

training directly from compressed video

14×

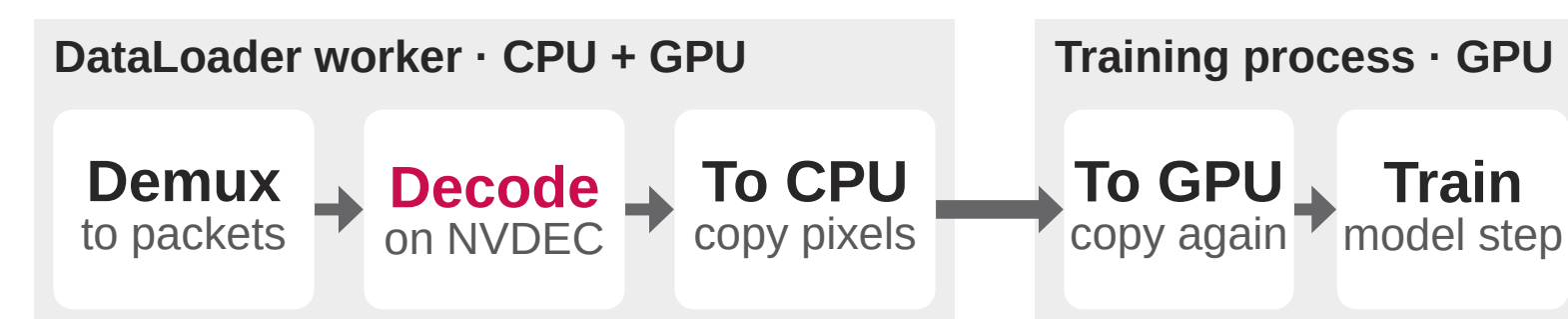
faster training than OpenCV

≈83k

images/second on 192 GPUs

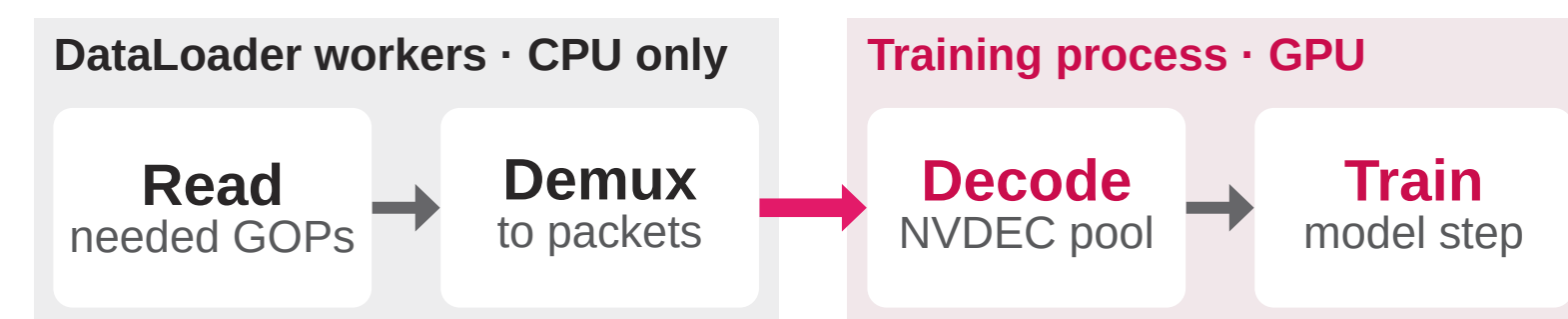
2 Decode out of the workers

Before: every DataLoader worker decoded

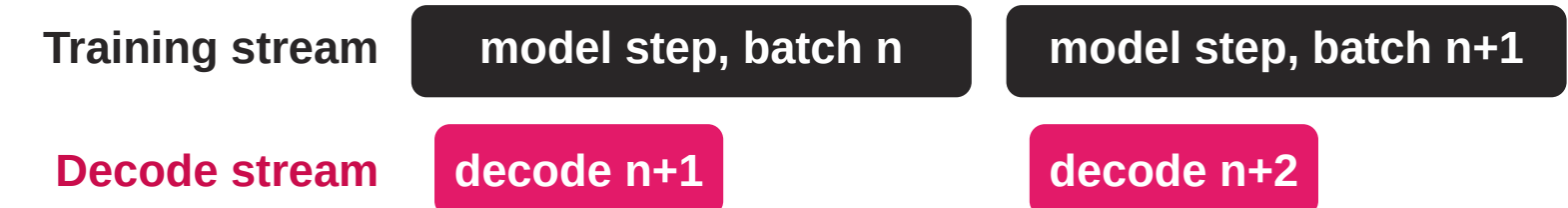


Each worker holds its own CUDA context and decoders, and every frame crosses PCIe twice.

After: workers demux, the trainer decodes



Compressed packets, not pixels, cross to the trainer. Decode runs on its own CUDA stream, one batch ahead:

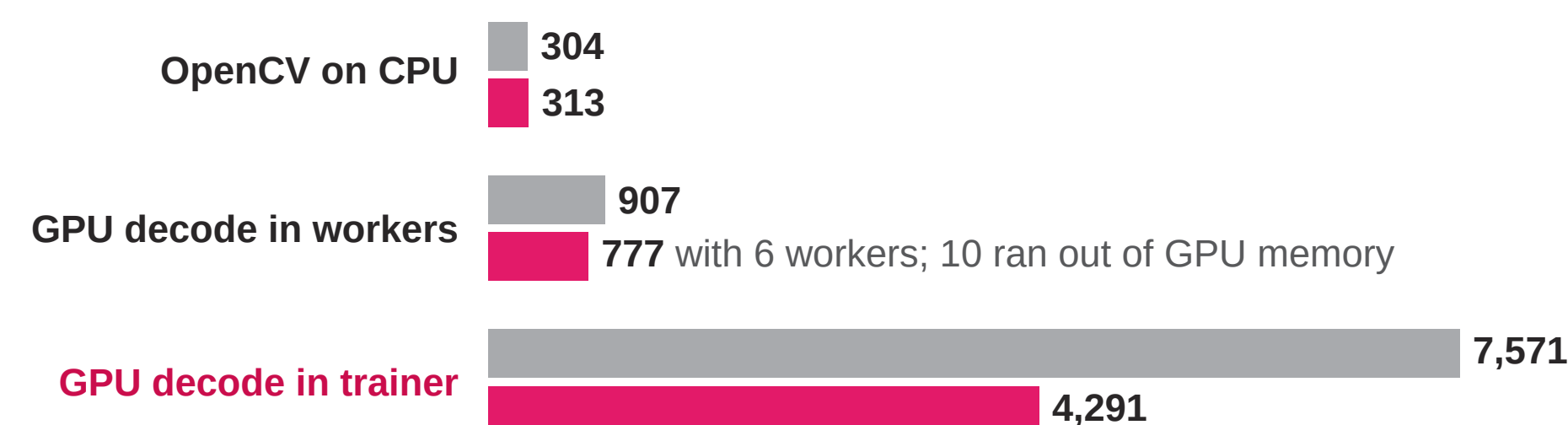


Decode is effectively hidden behind model compute.

5 Results

Images/second · production model · 8 GPUs

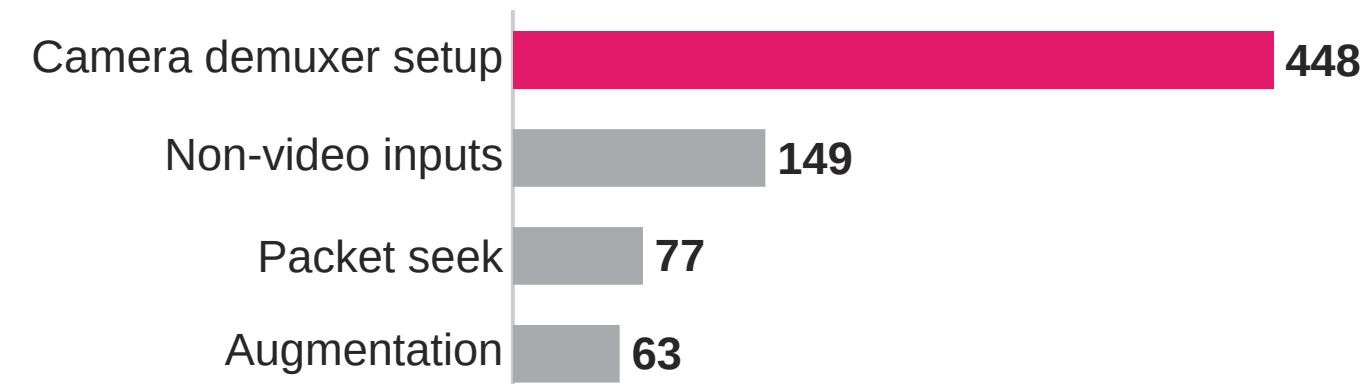
■ data loading only ■ training



Batch 12 per GPU. Six DataLoader workers per GPU is the most that fit beside the model.

3 After the split: demux became the bottleneck

DataLoader worker time per sample, before the demux fixes (ms)



Workers built **5.6 camera demuxers per sample**. Demux setup consumed **48% of worker time**, mostly waiting on file opens rather than CPU. Removing that work changed the whole host:

0.2%

time waiting on data (was 11–19%)

–54%

DataLoader worker CPU time

–93%

run-queue wait on the host

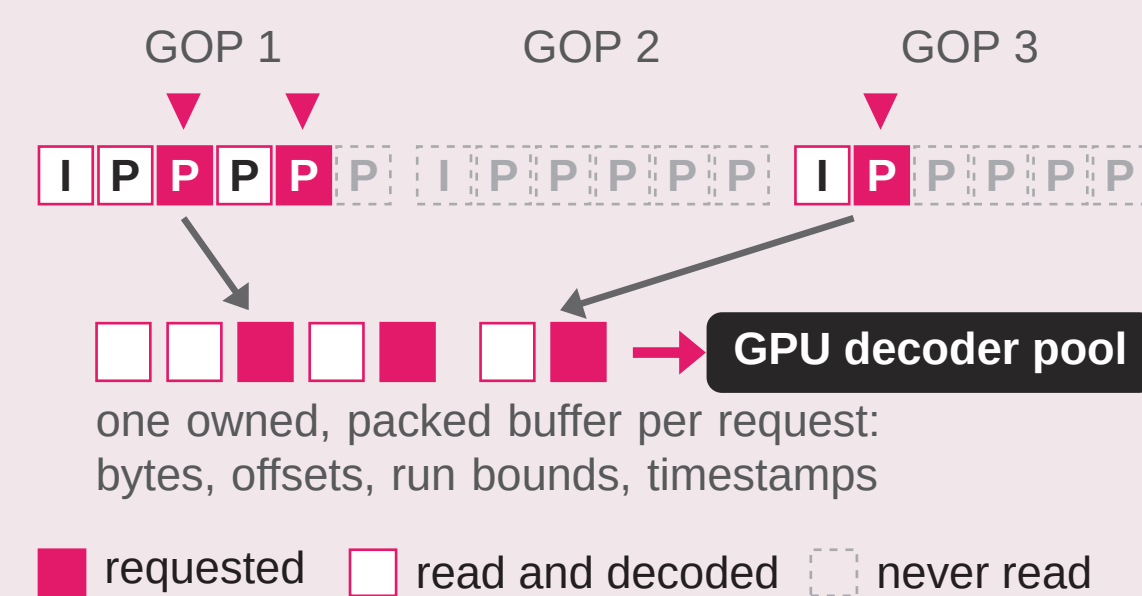
1.04 s

slowest step (was 8.11 s)

Before → after, one run per mode on 8 GPUs.

4 Remove work without changing the frames

Demux: read only the GOP runs a request needs



- **No probe, no scan:** read the MP4 header and seek index instead.
- **One demux call per request:** return one owned, packed packet buffer.
- **GOP read-ahead:** under direct I/O, one aligned fill of up to 512 KiB replaces many 32 KiB reads inside a GOP run.
- **GOP 6:** any frame needs at most 5 earlier frames: 2.4× faster decode than GOP 30, ~30% more storage.

–87%

stream probe 101 → 13.5 ms

–99.9%

GOP-size scan 79 → 0.04 ms

3.5×

fewer bytes read 5.3 → 1.5 GiB/step

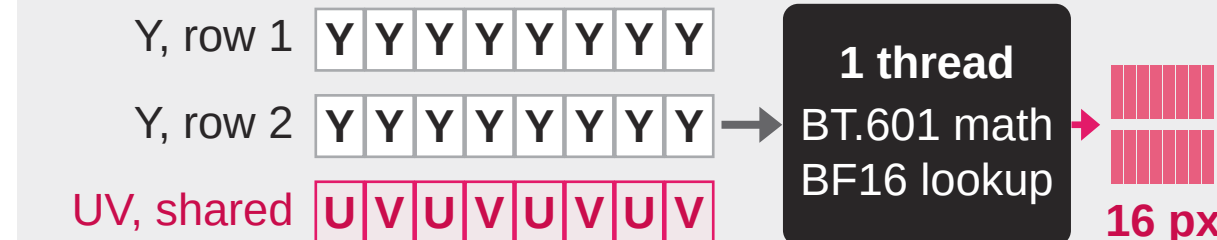
1/2

the read calls with read-ahead

All changes are bit-exact against the reference: same packets, frames and loss. The decoder pool uses 2.4–3 GB per GPU. Kernel numbers: H100, 192 frames at 1920×768.

Decode: pooled NVDEC and one fused NV12 → BF16 kernel

Retained NV12 buffer, one selected frame



Three 8-byte loads feed 16 BF16 RGB pixels, written straight into the model input: 1.5 bytes read per pixel, the NV12 minimum. The reference kernel read 3.

Pool path (CUDA events order steps, guard buffers)



2.2×

faster conversion 3.32 → 1.51 ms

+65%

kernel bandwidth 1086 → 1791 GB/s

3.9 GiB

GPU memory freed (no staged RGB)

+3.3%

end to end with fused conversion

- **Batch decode:** pair a CPU worker with an NVDEC decoder per slot; release the GIL.
- **Copy only selected frames:** references still decode but are not copied out.
- **Touch each pixel once:** 8 columns × 2 rows per thread share one chroma load.
- **One pass:** crop, BT.601, BF16 lookup and padding; no RGB8 or float temporaries.

What the optimized pipeline changes

+24.7%

end-to-end throughput from faster demux 5 paired runs, 8 GPUs

≈83k

images/second in a 3-hour training run 192 GPUs

–88%

network traffic at scale 192 GPUs, direct I/O

Result: direct MP4 decode keeps up with training without a decoded-frame cache or offline conversion pipeline. Workers read and demux only the compressed data needed; the trainer batches NVDEC and fused conversion on a separate CUDA stream.

The +24.7% measures the later demux optimization on the production training path.