arXiv:2605.08731v3 [cs.PF] 11 Aug 2026

Choosing a JPEG Decoder for PyTorch DataLoaders: Workload-Specific Throughput on Four CPUs

Vladimir Iglovikov
Ternaus
vladimir@ternaus.com

Dmitry Kosarevsky
Independent Researcher
if.kosarevsky@gmail.com

August 12, 2026

Abstract

A JPEG decoder benchmark can combine worker counts, CPUs, and datasets in one large result matrix. We simplify that comparison by fixing a PyTorch `DataLoader` at eight workers and asking one question: how much faster is each decoder than Pillow on the same CPU and JPEG workload?

We benchmark 12 Python decoders on four 16-vCPU Google Cloud platforms and two workloads from the Forchheim Image Database. The first contains 324 large camera originals. The second contains 1,668 originals and copies processed by social media services. Every decoder–CPU–workload setting has five isolated repetitions. JPEG bytes reside in memory, and each timed decode returns a complete three-channel RGB array.

The workload changes the practical result. On camera originals, the geometric mean across CPUs ranges from 1.00× to 1.09× Pillow. On the mixed workload, `simplejpeg`, `imagecodecs`, and `turbojpeg` reach 1.39–1.40× Pillow. The exact leader still changes by CPU. We therefore recommend comparing decoders on representative JPEGs, while using the fixed eight-worker results as a compact starting point. Full worker curves and absolute throughput tables are in the appendix.

1 Introduction

The JPEG decoder is one part of a training input pipeline. If that pipeline delivers the next batch too late, the GPU waits. A faster decoder can help in that case. The useful engineering question is how much decoder choice changes the rate at which a `DataLoader` supplies images.

This question is easy to bury in a benchmark matrix. Throughput changes with the decoder, worker count, CPU, and JPEG distribution. Reporting every combination gives a complete record, but makes the main result hard to see. We reduce the main comparison in three steps:

1. We discuss the two JPEG workloads separately.
2. We fix `num_workers` at eight for every decoder and CPU.
3. We divide each result by Pillow throughput on the same CPU and workload, then summarize the four CPU-specific ratios with a geometric mean.

Absolute images per second remain available in Appendix A. The fixed worker count is a comparison convention. It does not claim that eight workers minimizes training time.

The two workloads separate clearly. Decoder choice has a modest effect on the large camera originals and a much larger effect on the mixture of originals and smaller social-media copies. Hardware changes the exact ranking, but it does not erase this workload difference.

Contributions.

1. We compare 12 CPU JPEG decoders at one shared DataLoader setting on four 16-vCPU platforms and two real JPEG workloads.

2. We divide each result by Pillow throughput on the same CPU, so hardware speed does not dominate the decoder comparison.

3. We publish the full 3,725-bundle evidence archive, including worker sweeps, package versions, exact plans, and validation code.

2 Related work

FFCV [3], WebDataset [4], TorchData [10], NVIDIA DALI [5], and nvJPEG [6] address storage layout, sharding, preprocessing, or accelerator-side decoding. We study a narrower path: JPEG bytes decoded on the CPU and delivered through a PyTorch-style loader. The JPEG files are already in memory when timing starts, so storage speed does not enter the comparison.

Python pipelines expose several JPEG implementations through Pillow [8], OpenCV [7], torchvision [9, 11], Kornia [12], and wrappers around libjpeg-turbo [13]. Their backends, bindings, memory allocation, and thread controls differ. Our adapters convert every successful result to the same array format before counting the image as decoded.

3 Experimental setup

3.1 Two FODB workloads

The Forchheim Image Database (FODB) contains camera originals and copies processed by social-media services [2]. We select 12 scenes from all 27 cameras with a seeded ordering.

FODB-native contains 324 camera originals. The median image has 12.19 megapixels and occupies 3.74 MiB as a JPEG. This workload represents large, camera-generated files.

FODB-mixed starts from the same camera–scene pairs and adds copies from Facebook, Instagram, Telegram, Twitter, and WhatsApp. The timed common set contains 1,668 images with a median of 1.23 megapixels and 0.24 MiB. The mixture changes resolution, compression, metadata, and encoding history together. It supports a workload comparison, not a causal claim about any one JPEG property.

3.2 Decoders, CPUs, and output

We test `ajpegli`, `imagecodecs`, `imageio`, `jpeg4py`, `kornia-rs`, OpenCV, Pillow, PyVips, scikit-image, `simplejpeg`, `torchvision`, and `turbojpeg`. OpenCV, PyVips, and `torchvision` are explicitly limited to one internal thread. Runtime checks report one effective thread for the other decoders.

The four Google Cloud platforms each provide 16 vCPUs. The three x86 platforms are `c4-standard-16` with Intel Xeon 8581C, `c3d-standard-16` with AMD Zen 4, and `c4d-standard-16` with AMD Zen 5. The Arm platform is `c4a-standard-16` with Google Axion / Neoverse V2 [1]. Package versions are identical across platforms.

Every successful decode returns a complete C-contiguous RGB `uint8` array with shape `(height, width, 3)`. Compressed JPEG bytes are loaded into memory before validation, warmup, or timing. The loader uses batch size 1, pass-through collation, the `spawn` start method, prefetch factor 1, and persistent workers.

We assume that users have already checked that their chosen decoder accepts their JPEGs and returns the color format their application expects.[1] The main results compare speed on the same input images for every decoder.

3.3 Fixed comparison and summary statistic

The main analysis uses eight DataLoader workers for every decoder, CPU, and workload. This gives 480 timed bundles:

$$12 \text{ decoders} \times 4 \text{ CPUs} \times 2 \text{ workloads} \times 5 \text{ repetitions}.$$

[1] A small compatibility audit is reported in Appendix D. It covers one malformed JPEG pattern and one four-component JPEG. These cases cannot establish general decoder reliability.

Each repetition runs in a fresh process and times one complete traversal.

For decoder d, CPU c, workload s, and repetition r, we report the paired ratio

$$R_{d,c,s,r} = \frac{\text{images/s}_{d,c,s,r}}{\text{images/s}_{\text{Pillow},c,s,r}}.$$

The figures show all five paired ratios on each CPU. A black diamond marks the geometric mean of the four CPU-specific mean ratios. This normalization keeps hardware speed from dominating the decoder comparison. It does not mean that the four CPUs behave the same.

4 Camera originals: decoder choice changes throughput modestly

The decoder gaps are small on camera originals. Across the four CPUs, the geometric mean ranges from 1.00× for Pillow to 1.09× for `torchvision`. PyVips follows at 1.08×. The other nine decoders fall between 1.02× and 1.06×.

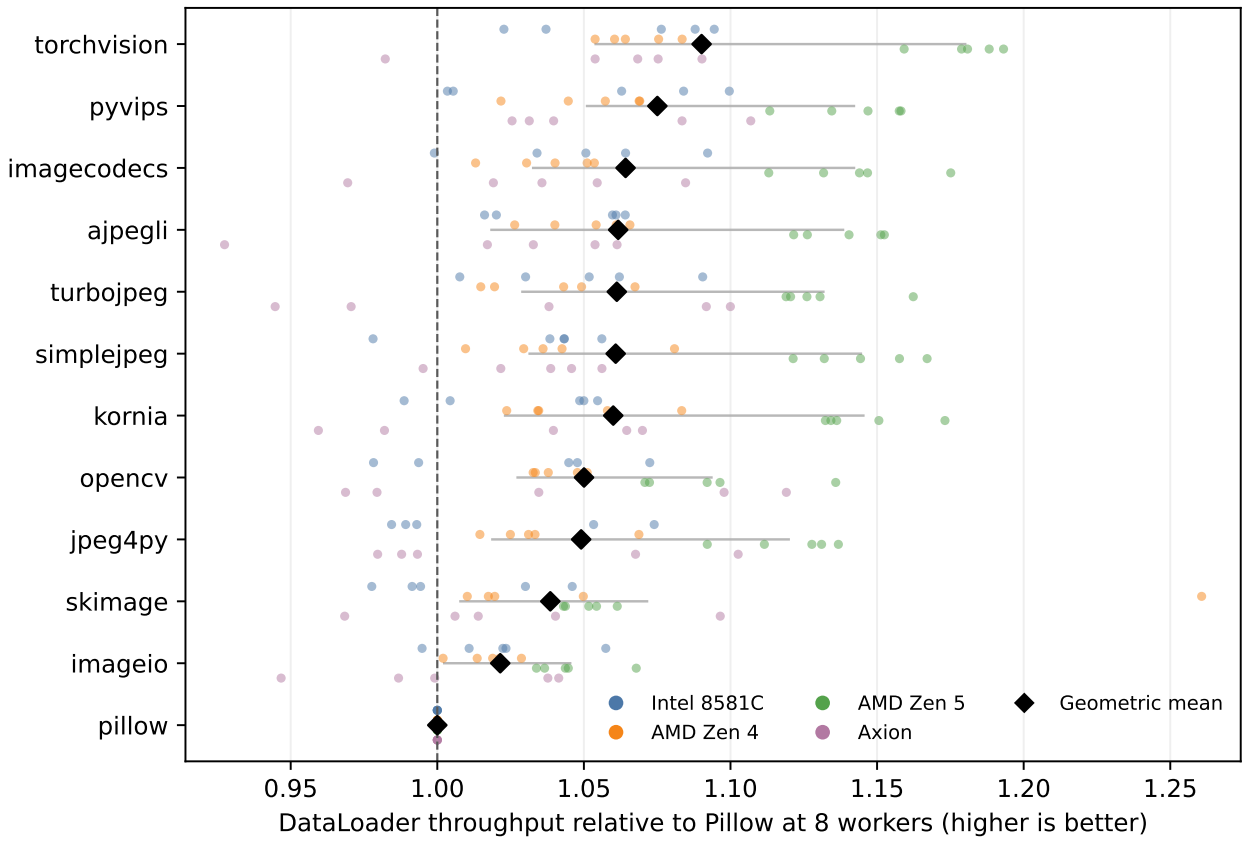


Figure 1: DataLoader throughput on 324 FODB camera originals at eight workers, relative to Pillow on the same 16-vCPU platform and repetition. Colored points show five paired repetitions on each CPU. Gray lines span the four CPU-specific mean ratios; black diamonds show their geometric mean. Every timed output is a complete RGB `uint8` array.

The exact leader depends on the CPU. The fastest mean is `torchvision` on Intel and Zen 5, scikit-image on Zen 4, and PyVips on Axion. The top mean exceeds Pillow by about 6%–7% on Intel, Zen 4, and Axion, and by 18% on Zen 5. Absolute leader throughput ranges from 17.4 to 40.1 images/s across the four machines.

For this workload, decoder selection offers a limited gain compared with the variation between CPUs. A user processing similar camera originals can keep Pillow as a reasonable baseline and benchmark a small shortlist only when input throughput is already a measured bottleneck.

5 Mixed JPEGs: the leading decoders reach about 1.4 times Pillow

The mixed workload produces a clearer separation. The top group in Figure 2 contains `simplejpeg`, `imagecodecs`, and `turbojpeg`. Their geometric means are 1.40×, 1.40×, and 1.39× Pillow. Their CPU-specific mean ratios span 1.32×–1.53×.

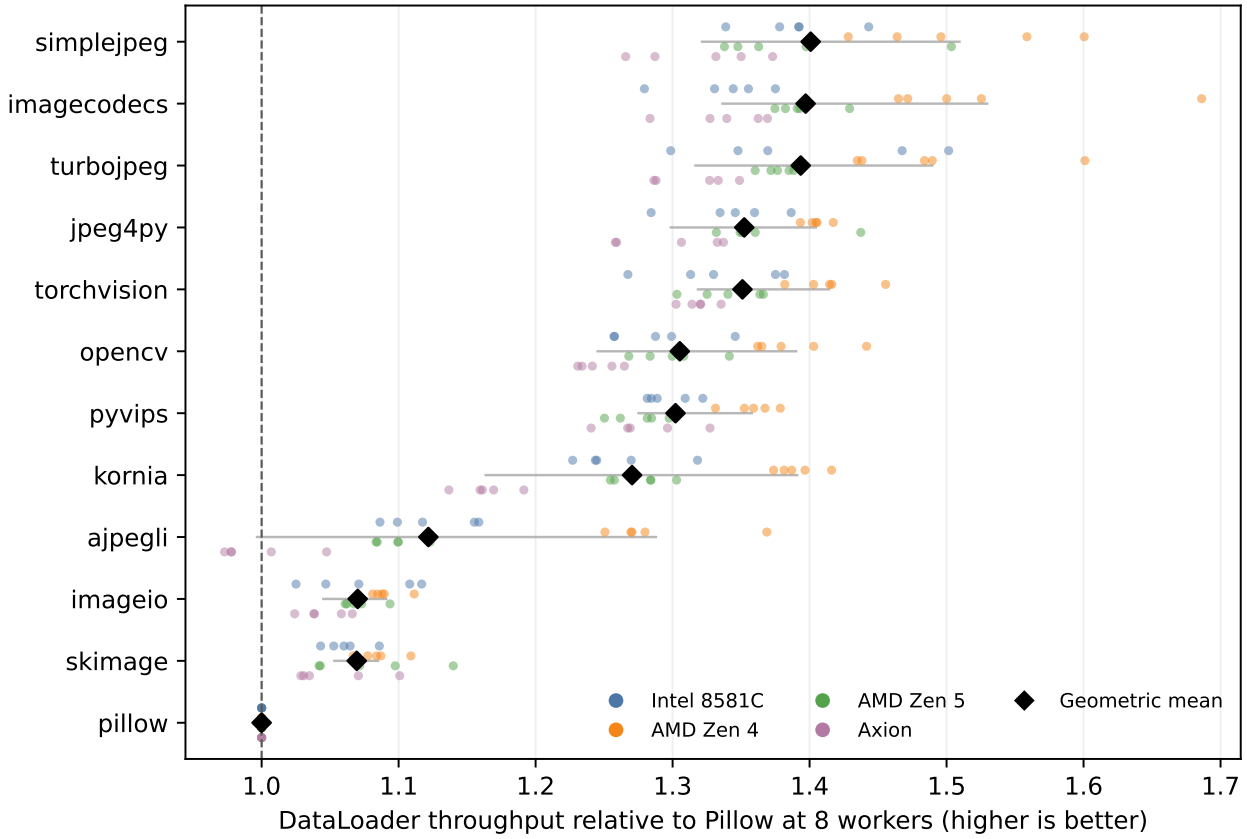


Figure 2: DataLoader throughput on 1,668 FODB originals and copies processed by social media services at eight workers, relative to Pillow on the same 16-vCPU platform and repetition. Colored points show five paired repetitions on each CPU. Gray lines span the four CPU-specific mean ratios; black diamonds show their geometric mean.

The leading library still changes by CPU. `turbojpeg` has the highest mean on Intel, while `imagecodecs` leads on Zen 4, Zen 5, and Axion. The gap within the top three is small compared with their gap over Pillow. Their absolute means range from about 58 to 110 images/s, depending on CPU.

The mixed workload changes several JPEG properties at once, so this experiment cannot identify the cause of the larger gap. The measured gap still shows that a decoder choice which matters little for camera originals can matter much more for another real JPEG population.

6 How to use these results

Start by matching the workload. For large camera originals, decoder choice changed the four-CPU geometric mean by at most 9% relative to Pillow. For the mixed workload, test `simplejpeg`, `imagecodecs`, and `turbojpeg` first. Keep Pillow in the comparison so the ratio remains easy to interpret.

Then run the shortlist on the deployment CPU and representative JPEGs. Hardware changes the exact leader, and a new JPEG distribution may change the ranking again.

Eight workers make the decoder comparison compact. They are a starting point, not a tuned training setting. If the model still waits for input, test nearby worker counts in the real pipeline and measure GPU idle time and wall-clock training time. Appendix B reports our wider worker sweep and shows why its result should not be treated as a universal default.

7 Limitations

FODB-native and FODB-mixed differ in resolution, compressed size, estimated quality, progressive encoding, source service, and item count. The experiment compares these two workloads as wholes. It cannot assign their different decoder gaps to one property.

The common-input rule removes 276 WhatsApp files from the mixed timing set because not every decoder accepts them. The timed population therefore contains 1,668 of the planned 1,944 images. This preserves a fair speed comparison, but changes the WhatsApp share.

The fixed eight-worker result measures DataLoader throughput, not training speed. The benchmark excludes storage, augmentation, tensor stacking, pinned-memory transfer, GPU decode, and model computation. Batch size 1 avoids padding images of different sizes and is not a proposed training default.

The hardware scope is four 16-vCPU Google Cloud machine types running the same Linux software environment. Five repetitions show run-to-run variation but cannot resolve small differences between nearly equal means.

8 Reproducibility

The public artifact is available at https://github.com/ternaus/imread_benchmark. It contains content-addressed datasets, benchmark plans, run bundles, validation code, and generated figures and tables. The full archive has 3,725 validated FODB speed bundles. The fixed eight-worker result in this paper is a balanced 480-bundle subset. Appendix F records the package and plan identities.

9 Conclusion

JPEG decoder choice depends on the workload. At eight DataLoader workers, the 12 tested decoders stay within 9% of Pillow by four-CPU geometric mean on large camera originals. On the mixed workload, the top three reach about 1.4× Pillow. Normalize within each CPU, inspect each workload separately, and use the result to choose a small local benchmark shortlist.

Acknowledgments

We thank Google and the Google Developer Program for providing Google Cloud credits that supported the benchmark runs.

References

[1] Google Cloud. Google Cloud compute engine machine families (c4 / c4d / c4a / t2a documentation). https://cloud.google.com/compute/docs/machine-types, 2026. Accessed 2026-05-02.

[2] Benjamin Hadwiger and Christian Riess. The forchheim image database for camera identification in the wild. In *Pattern Recognition. ICPR International Workshops and Challenges*, Lecture Notes in Computer Science, pages 500–515. Springer, 2021. doi: 10.1007/978-3-030-68780-9_40.

[3] Guillaume Leclerc, Andrew Ilyas, Logan Engstrom, Sung Min Park, Hadi Salman, and Aleksander Mądry. FFCV: Accelerating training by removing data bottlenecks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 12011–12020, 2023.

[4] Bob McElrath and Thomas Breuel. Webdataset: a PyTorch dataset (WebDataset) designed for streaming training. https://github.com/webdataset/webdataset, 2021. Accessed 2026-05-02.

[5] NVIDIA Corporation. NVIDIA DALI: GPU-accelerated data loading and image augmentation. https://developer.nvidia.com/dali, 2024. Accessed 2026-05-02.

[6] NVIDIA Corporation. nvJPEG: GPU-accelerated JPEG decode. https://developer.nvidia.com/nvjpeg, 2024. Accessed 2026-05-02.

[7] OpenCV Team. Open source computer vision library (OpenCV). https://opencv.org, 2024. Accessed 2026-05-02.

[8] Pillow Developers. Pillow: the friendly PIL fork. https://pillow.readthedocs.io/en/stable/, 2024. Accessed 2026-05-02.

[9] PyTorch Team. PyTorch. https://pytorch.org, 2024. Accessed 2026-05-02.

[10] PyTorch Team. Torchdata. https://github.com/pytorch/data, 2024. Accessed 2026-05-02.

[11] PyTorch Team. torchvision. https://pytorch.org/vision, 2024. Accessed 2026-05-02.

[12] Edgar Riba, Dmytro Mishkin, Daniel Ponsa, Ethan Rublee, and Gary Bradski. Kornia: differentiable computer vision in PyTorch. https://kornia.github.io, 2024. Accessed 2026-05-02.

[13] The libjpeg-turbo Project. libjpeg-turbo. https://libjpeg-turbo.org, 2024. Accessed 2026-05-02.

A Absolute throughput at eight workers

Tables 1 and 2 give the absolute values behind Figures 1 and 2. Each cell is the mean and sample standard deviation over five isolated repetitions. The final column is the geometric mean of the four CPU-specific paired ratios to Pillow.

Decoder	Intel	Zen 4	Zen 5	Axion	Geo. mean
`torchvision`	18.7 ± 0.6	17.4 ± 0.1	40.1 ± 0.5	28.2 ± 0.7	1.09×
`pyvips`	18.4 ± 0.9	17.1 ± 0.3	38.8 ± 0.4	28.3 ± 0.4	1.08×
`imagecodecs`	18.4 ± 0.5	16.9 ± 0.3	38.8 ± 0.5	27.6 ± 0.2	1.06×
`ajpegli`	18.3 ± 0.6	17.1 ± 0.2	38.7 ± 0.6	27.2 ± 0.9	1.06×
`turbojpeg`	18.4 ± 0.7	16.9 ± 0.2	38.4 ± 0.5	27.5 ± 1.1	1.06×
`simplejpeg`	18.1 ± 0.6	16.9 ± 0.3	38.9 ± 0.5	27.6 ± 0.9	1.06×
`kornia`	18.0 ± 0.6	17.0 ± 0.3	38.9 ± 0.5	27.4 ± 0.7	1.06×
`opencv`	18.0 ± 0.7	17.0 ± 0.1	37.1 ± 0.7	27.8 ± 1.0	1.05×
`jpeg4py`	17.9 ± 0.8	16.9 ± 0.3	38.0 ± 0.4	27.4 ± 0.8	1.05×
`skimage`	17.7 ± 0.5	17.4 ± 1.7	35.7 ± 0.3	27.4 ± 0.8	1.04×
`imageio`	17.9 ± 0.3	16.6 ± 0.3	35.5 ± 0.4	26.8 ± 0.8	1.02×
`pillow`	17.5 ± 0.3	16.3 ± 0.2	34.0 ± 0.3	26.8 ± 1.0	1.00×

Table 1: FODB-native DataLoader throughput in images/s at eight workers. Higher is better.

Decoder	Intel	Zen 4	Zen 5	Axion	Geo. mean
`simplejpeg`	59.6 ± 2.4	59.1 ± 2.7	110.1 ± 4.3	87.2 ± 3.0	1.40×
`imagecodecs`	57.3 ± 1.7	59.9 ± 3.6	110.5 ± 2.8	88.2 ± 2.3	1.40×
`turbojpeg`	59.8 ± 2.6	58.3 ± 2.8	109.1 ± 2.0	86.8 ± 1.8	1.39×
`jpeg4py`	57.5 ± 1.5	55.0 ± 0.6	108.2 ± 2.3	85.6 ± 1.4	1.35×
`torchvision`	57.2 ± 2.0	55.4 ± 1.2	106.2 ± 0.9	87.0 ± 1.2	1.35×
`opencv`	55.3 ± 1.1	54.4 ± 1.7	103.0 ± 2.5	82.1 ± 0.8	1.31×
`pyvips`	55.6 ± 1.3	53.1 ± 0.3	101.1 ± 1.3	84.4 ± 2.8	1.30×
`kornia`	54.0 ± 1.0	54.4 ± 0.7	101.2 ± 1.2	76.7 ± 1.0	1.27×
`ajpegli`	48.2 ± 1.7	50.4 ± 1.7	87.0 ± 1.9	65.7 ± 0.9	1.12×
`imageio`	46.0 ± 1.7	42.7 ± 0.6	84.9 ± 1.9	68.9 ± 0.7	1.07×
`skimage`	45.5 ± 1.0	42.5 ± 0.5	85.5 ± 3.5	69.4 ± 1.2	1.07×
`pillow`	42.9 ± 1.0	39.1 ± 0.4	79.3 ± 1.2	66.0 ± 1.3	1.00×

Table 2: FODB-mixed DataLoader throughput in images/s at eight workers. Higher is better.

B Worker-count sensitivity

The full experiment starts with 0, 2, 4, and 8 workers. A rule fixed before the next stage adds 12 workers when the highest base-grid mean occurs at eight. This selects 87 of 96 decoder–CPU–workload combinations. A second rule adds 16 workers when the 12-worker mean is at least 5% above the 8-worker mean and all five paired repetitions avoid a decrease. This selects 58 combinations.

The mixed workload reaches its highest observed mean at 16 workers in 43 of 48 decoder–CPU combinations and at 12 in the other five. Native images peak at 12 workers in 35 combinations, zero in nine, eight in two, and 16 in two. These are DataLoader-only results. Training workers share CPU and memory with the rest of the pipeline, so the observed peaks do not define a training default.

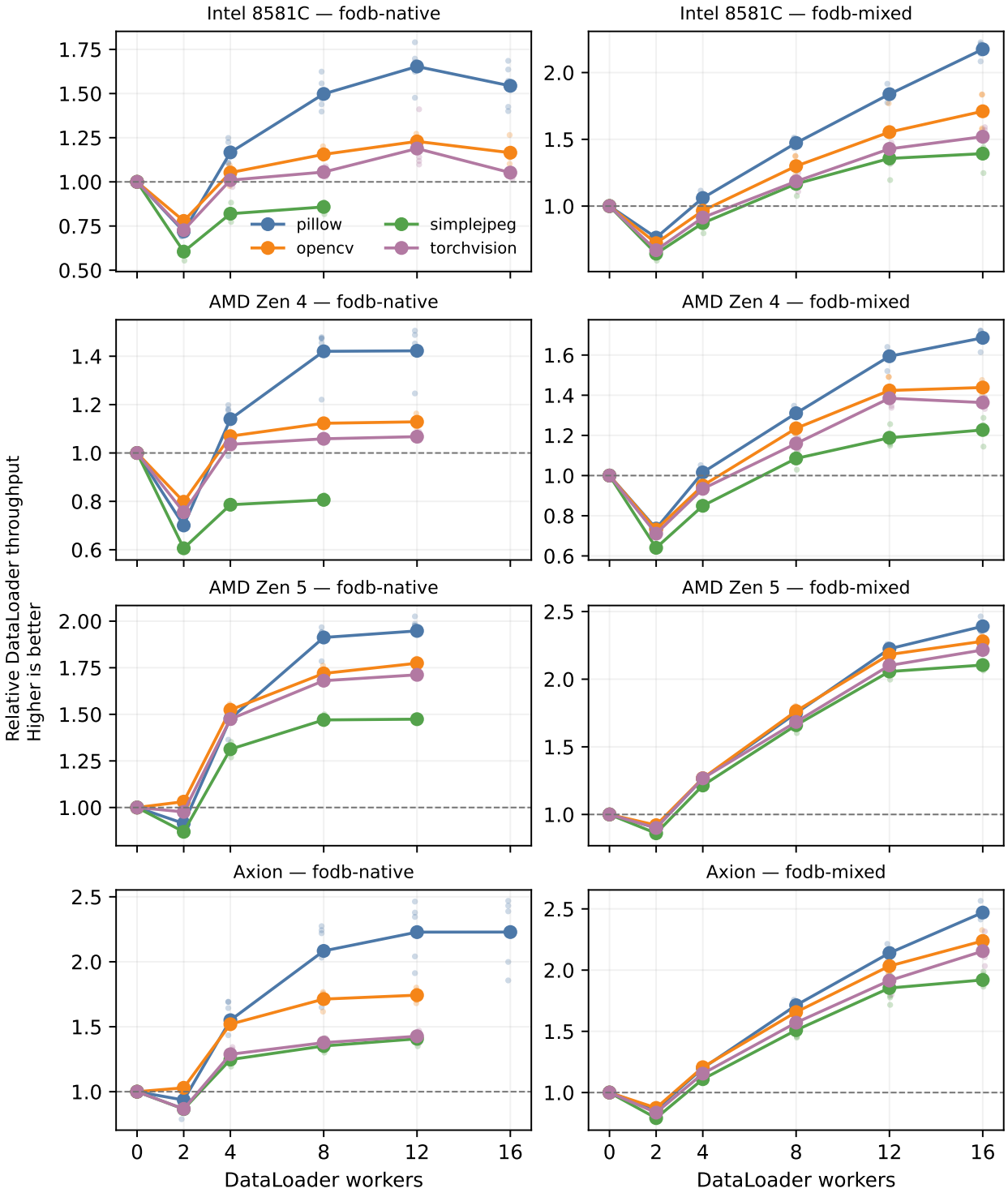


Figure 3: Worker scaling for four representative controlled-thread decoders. Throughput is divided by the same decoder at zero workers; higher is better. Lines are means of five repetition-paired ratios, and translucent points show each ratio. Every curve contains 0, 2, 4, and 8 workers. The staged rules determine whether 12- and 16-worker points appear.

Platform	Native	Mixed	Changed
Intel 8581C	0: 4, 12: 7, 16: 1	12: 1, 16: 11	11/12
AMD Zen 4	0: 5, 8: 1, 12: 6	12: 3, 16: 9	11/12
AMD Zen 5	8: 1, 12: 11	12: 1, 16: 11	12/12
Axion	12: 11, 16: 1	16: 12	11/12

Table 3: Highest observed mean worker counts for the 12 controlled-thread decoders. An entry such as 16: 11 means that 11 decoders peak at 16 workers. Changed counts decoders whose peak differs between workloads.

C One-thread choice versus a tuned DataLoader

The fastest controlled one-thread decoder and the fastest decoder after worker tuning differ in five of eight CPU–workload pairs. Choosing the one-thread leader and then tuning only its workers loses 0.9%–7.9% of mean DataLoader throughput in those five pairs. The losses stay within 10%, but the complete decoder ranking can change by a large margin.

Workload	Platform	One-thread leader	Tuned leader	Loss	Rank corr.
FODB-native	Intel 8581C	`turbojpeg`	`turbojpeg` (0 workers)	0.0%	0.66
FODB-native	AMD Zen 4	`imagecodecs`	`imagecodecs` (0 workers)	0.0%	0.90
FODB-native	AMD Zen 5	`imagecodecs`	`torchvision` (12 workers)	5.0%	0.13
FODB-native	Axion	`torchvision`	`jpeg4py` (12 workers)	7.9%	0.23
FODB-mixed	Intel 8581C	`imagecodecs`	`torchvision` (16 workers)	0.9%	0.54
FODB-mixed	AMD Zen 4	`imagecodecs`	`imagecodecs` (12 workers)	0.0%	0.77
FODB-mixed	AMD Zen 5	`imagecodecs`	`turbojpeg` (16 workers)	3.6%	0.88
FODB-mixed	Axion	`imagecodecs`	`torchvision` (16 workers)	5.7%	0.75

Table 4: Fastest controlled one-thread decoder versus fastest tuned DataLoader decoder. Lower loss is better. Rank correlation covers all 12 decoders.

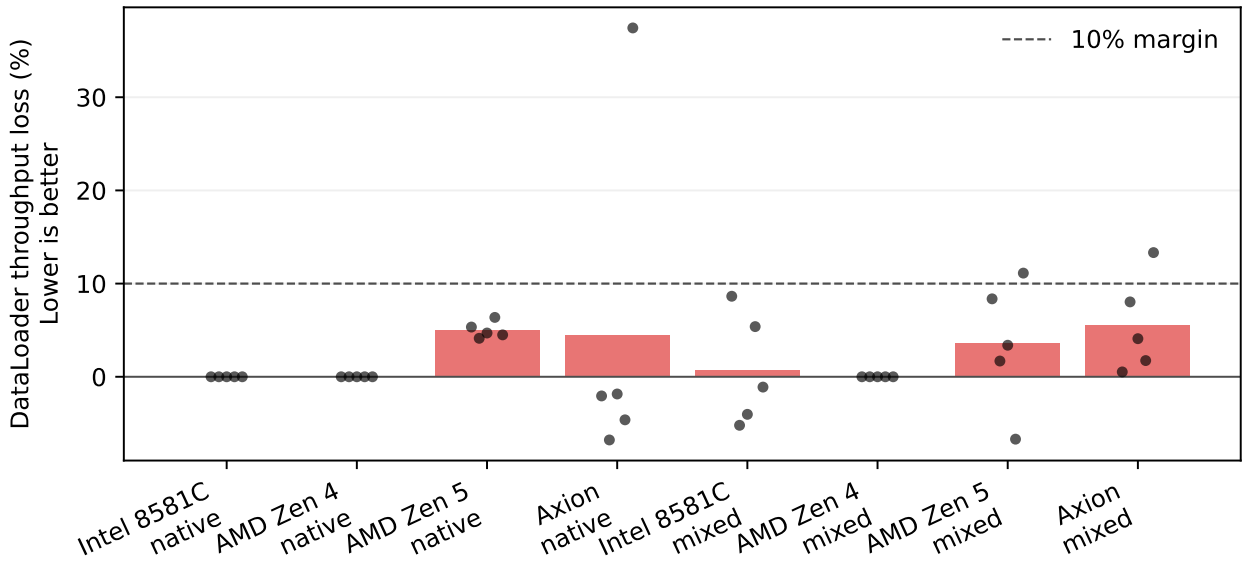


Figure 4: DataLoader throughput lost by choosing the fastest controlled one-thread decoder and then tuning its worker count. Bars are mean losses; black points show five repetition-matched values. Red marks a change in the fastest decoder. The 10% line is a practical margin, not a confidence threshold.

D Compatibility scope

The speed comparison assumes that the application has already validated its JPEG population and required output. We ran two narrow checks to document why that assumption matters. The first contains 276 progressive WhatsApp JPEGs with empty DHT markers. The second is one valid four-component JPEG and asks whether the decoder returns the benchmark's three-channel RGB array.

Decoder	Empty-DHT	Four-component	Both
`ajpegli`	0/276	0/1	0/277
`imagecodecs`	276/276	0/1	276/277
`imageio`	276/276	0/1	276/277
`jpeg4py`	276/276	0/1	276/277
`kornia`	276/276	0/1	276/277
`opencv`	276/276	1/1	277/277
`pillow`	276/276	1/1	277/277
`pyvips`	276/276	0/1	276/277
`simplejpeg`	276/276	1/1	277/277
`skimage`	276/276	0/1	276/277
`torchvision`	276/276	0/1	276/277
`turbojpeg`	276/276	0/1	276/277

Table 5: Results for two compatibility checks. Entries show files passed out of files tested. The checks cover two observed cases and do not measure general decoder reliability.

OpenCV, Pillow, and `simplejpeg` pass both checks under the pinned Linux environment. Other applications may need different color conversion, error recovery, or fallback behavior. They should repeat these checks on their own data. This small audit is not a general ranking.

E Timed workload composition

Every decoder is timed on the same images. Native retains all 324 planned files. Mixed retains 1,668 of 1,944. All 276 exclusions are progressive WhatsApp copies with estimated quality 69. The timed mixed set contains 324 images from the original, Facebook, Instagram, Telegram, and Twitter sources, and 48 from WhatsApp.

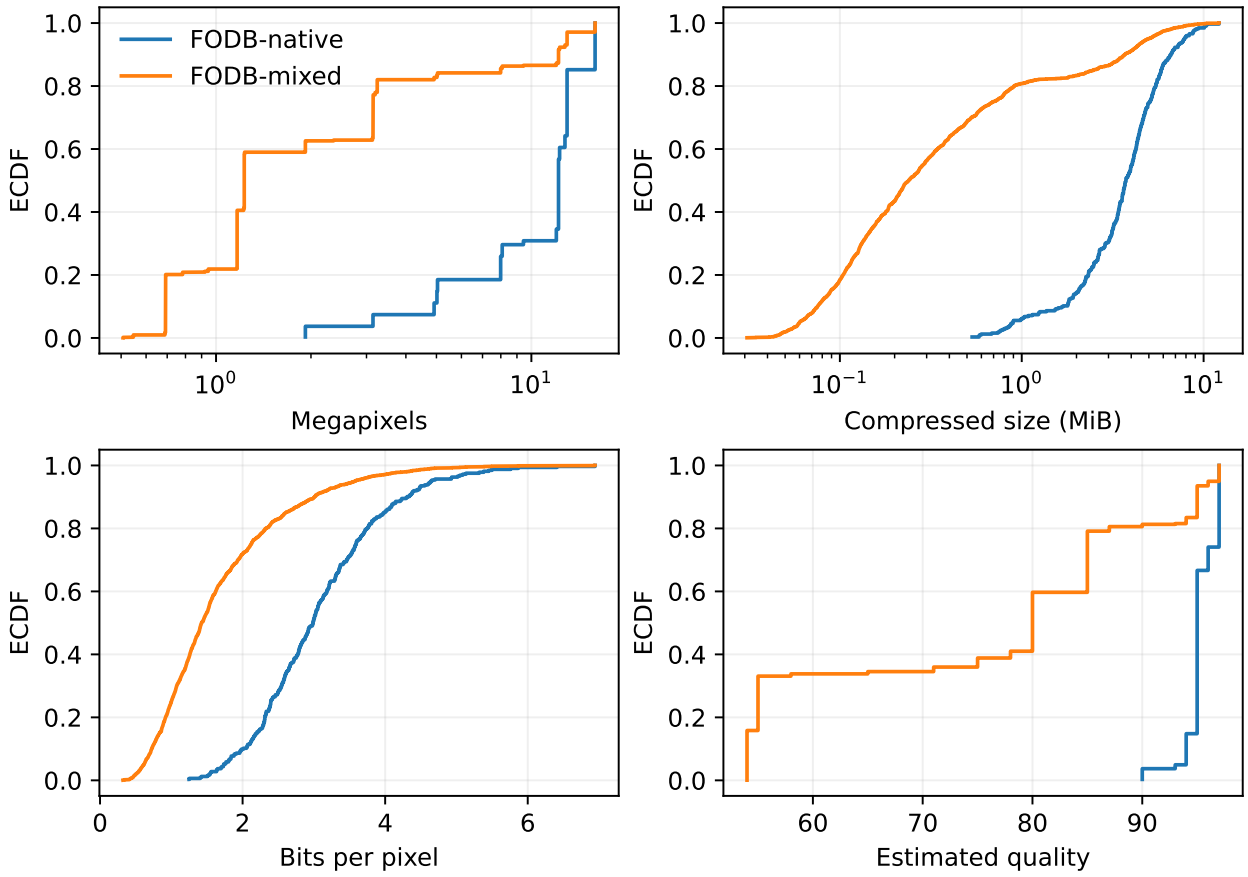


Figure 5: Empirical distributions for the two timed workloads. Resolution, compressed size, bits per pixel, estimated quality, and progressive encoding change together. The figure describes the workload shift and does not isolate a causal factor.

F Evidence receipt and package versions

The complete speed archive contains 3,725 bundles: 1,745 FODB-native and 1,980 FODB-mixed. Every selected configuration has five fresh-process repetitions. The analysis validates the plan IDs, runner revision, package identities, output contract, platform identity, repetition count, timed images, and expected bundle count before generating paper assets.

Dataset package `925d57ed84cb07e12f290e6c1612b61ca892fabea729091bfabd0acf3661cc9f`

Native base plan `6c746a5c21978f75c58c8c779f6628c2645a199b63489da9b9f1a9bb4f335c8f`

Mixed base plan `e04f54dc177cadf86cb99a9fc34b950afb5b1abee5b46af040bafc8a4aa7910b`

Runner revision `52a0bea5a2f44079883c9a41472b3add285955f2b471c37674ddd2fb5d4da6ff`

Table 6 lists the package versions recorded on every platform. The libjpeg-turbo build reports version 3.2.0, build 20260630.

Distribution	Version
ajpegli	1.0.0
imagecodecs	2026.6.26
imageio	2.37.4
jpeg4py	0.1.4
kornia-rs	0.1.14
opencv-python-headless	5.0.0.93
pillow	12.3.0
pyturbojpeg	2.5.0
pyvips	3.1.1
scikit-image	0.26.0
simplejpeg	1.9.0
torch	2.13.0+cpu
torchvision	0.28.0+cpu

Table 6: Pinned package versions for the FODB experiment.

G Thread-control check

OpenCV, PyVips, and `torchvision` are measured both with package-default thread behavior and with one requested internal thread. Default PyVips reports up to 16 effective threads in the one-thread decode protocol, and default `torchvision` reports up to eight. The main comparison therefore uses the explicit one-thread configurations for these three libraries. Default thread results remain in the generated evidence for applications that preserve package defaults.