Dense PFT / HKD Inf for an 8K Graphics Engine: Measured Speed, Numerical Accuracy, and Disclosure Boundaries

Technical benchmark report

November 5, 2026

Abstract

We examine two recorded experiments relevant to the proposed dense PFT/HKD Inf 8K graphics engine: a structured dense-training contraction and a matched 8K image-warp benchmark. On a 576-parameter polynomial network, five-node weighted evaluation reduced the median time for twenty dense Adam updates by factors of 1.288, 5.719, and 30.905 relative to full sample evaluation. Against a polynomial-moment implementation the factors were 1.385–1.411. Maximum final-parameter discrepancies were below 2.3×10^{-16}. Separately, the 8K PFT JPEG path achieved 1.186 times the speed of direct dense rendering but 0.945 times that of optimized separable dense rendering. These are distinct workloads: the training result is not an end-to-end 8K Renderdon speedup. A self-contained Python appendix reproduces every table from the recorded observations.

Keywords: Polynomial thinning (PFT); HKD Inf; polynomial loss contraction; Lagrange interpolation; dense neural network training; 8K image rendering; numerical accuracy; performance benchmarking; reproducible computation.

1 Evidence and scope

The dense results are taken from the November 5 polynomial-contraction audit, including its raw trials, numerical checks, and evaluator source[1]. The graphics results come from the October 27 matched 8K benchmark and its source-bound JSON record[2]. These local primary experiment records, rather than advertised engine performance, define the claims in this report. The term "8K" here means an output raster of 8192×5120 pixels; the input image was 1536×1024. This is existing-image polynomial warping, not native 8K neural synthesis. No recorded experiment supplied here combines the dense contraction with a complete Renderdon generation pipeline.

2 Dense contraction: mathematical condition and implementation

Let the complete per-example loss $\ell(\theta, t)$ be polynomial in t of degree at most d for every parameter vector along an open neighborhood of the training trajectory. Assume differentiable coefficients, fixed sample coordinates t_i, fixed empirical weights b_i, and $r = d + 1$ distinct valid nodes s_j. With Lagrange basis L_j, set

$$a_j = \sum_i b_i L_j(t_i).$$

Polynomial interpolation and finite summation give

$$\sum_i b_i \ell(\theta, t_i) = \sum_{j=1}^{r} a_j \ell(\theta, s_j), \qquad \nabla_\theta \sum_i b_i \ell(\theta, t_i) = \sum_{j=1}^{r} a_j \nabla_\theta \ell(\theta, s_j).$$

Thus equal initialization, optimizer state, regularization and update schedule produce the same deterministic dense-optimizer trajectory in exact arithmetic. Floating-point reassociation still requires numerical verification.

The tested model is $f_\theta(t) = W_2(W_1[1, t]^T)^{\odot 2}$ with $W_1 \in \mathbb{R}^{32\times 2}$ and $W_2 \in \mathbb{R}^{16\times 32}$. All 576 parameters are trained. Quadratic targets and squared error give a degree-four loss, so five nodes suffice. This contracts the sample dimension; it neither sparsifies trainable weights nor substitutes a sparse optimizer. A conventional coefficient/moment implementation is included as a stronger algebraic comparator. This is not evidence of exclusive ownership of the algebra.

A conditional work model makes the boundary explicit:

$$W_{\text{full}} = K(Nc + U), \quad W_{\text{PFT}} = A + K(rc + U + H), \quad W_{\text{full}} > W_{\text{PFT}} \iff K(N - r)c > A + KH.$$

Here A includes setup, U common work, and H contraction overhead. The condition is not an unconditional hardware-time guarantee. Larger sample count at fixed model size is the tested scaling variable; increased parameter density, diffusion-model applicability and higher visual accuracy were not tested.

3 Dense-training measurements

The saved environment is PyTorch 2.8.0, CPU, float64, one thread, seed 133. The audit does not identify a CPU model or a complete platform inventory for this experiment. Each mode was warmed for two steps, then run in five shuffled-order trials of twenty updates using ordinary dense Adam (learning rate 0.001, foreach disabled). Timings include forward, backward, updates and the initial-gradient capture. Model/optimizer construction and one-time data/moment setup are excluded. The setup column records the shared preparation block; it is not a per-mode end-to-end measurement. The largest 30.905-fold ratio is relative to full sample evaluation; the optimized

Table 1: Dense training: medians of five trials; ratios use unrounded seconds.

N	Full ms	PFT ms	Moment ms	Full/PFT	Moment/PFT	Setup ms
257	6.776	5.261	7.293	1.288	1.386	33.689
4097	28.498	4.983	6.902	5.719	1.385	0.461
16385	157.077	5.083	7.173	30.905	1.411	1.048

polynomial-moment comparator reduces that advantage to 1.411-fold. Five trials establish the reported medians, not statistical confidence or a universal speed guarantee. The smallest-grid setup exceeds its training-loop time, illustrating why excluded setup must not be hidden.

Accuracy here means numerical agreement with the full-evaluation trajectory. The checks compare initial gradients, parameters after twenty updates, and the final recorded loss (computed before the final update). They do not check every intermediate optimizer state or establish improved held-out accuracy. The acceptance tolerance was 10^{-10}. All listed errors pass it; none imply bitwise equality.

Table 2: PFT versus full evaluation: maximum recorded errors over five trials.

N	Gradient L_∞	Parameter L_∞	Absolute loss error
257	4.337e-18	1.110e-16	0.000e+00
4097	1.171e-17	1.943e-16	1.735e-18
16385	5.551e-17	2.220e-16	1.735e-18

4 8K graphics measurements

The October 27 run used macOS 15.7.4 ARM64, Python 3.9.6, NumPy 1.23.5, OpenCV 4.12.0 and libjpeg-turbo 3.1.0-70, with one OpenCV thread and OpenCL disabled. One warmup and three measured trials per path were randomized. JPEG used quality 98, 4:4:4 sampling, no progressive coding and no optimized Huffman option; PNG used compression level 1. Full-save timing includes input decode, rendering, encoding, file write and fsync, with warm filesystem cache. Subsequent read/decode and validation are excluded equally. For the same JPEG settings, direct dense/PFT is 1.186 whereas optimized dense/PFT is 0.945: PFT is slower than the optimized baseline in this run. The same-renderer PNG/JPEG full-save ratio is 3.584; this codec tradeoff must not be relabeled an algorithmic PFT-versus-dense acceleration. Three trials and one source image do not establish cross-scene performance.

For 8-bit RGB, pixel accuracy uses PSNR $= 10 \log_{10}(255^2/\text{MSE})$, maximum absolute channel error, and the number of pixels with at least one changed channel. The table separates warp error from lossy JPEG error. The output has 41,943,040 pixels, and none of these comparisons is pixel-exact. High PSNR establishes closeness to this reference raster, not semantic correctness, new-scene realism or temporal consistency. The native HKD kernel checks channel sums outside timing, and the HKD ALU reports measured values. Neither role establishes inference acceleration; a channel sum is not an injective image identity test.

Table 3: 8K full-save timings. Relative speed is optimized dense / listed path, within the same codec.

Path	Median s	Observed range s	Relative speed
Optimized dense JPEG	0.913394	0.903122–0.917848	1.000
Direct dense JPEG	1.146218	1.107282–1.208415	0.797
PFT JPEG	0.966365	0.915680–0.987749	0.945
Optimized dense PNG	3.462377	3.448335–3.511144	1.000
PFT PNG	3.463412	3.461146–3.617629	1.000

Table 4: Recorded full-raster image fidelity for the PFT JPEG path.

Comparison	PSNR dB	Max error	Changed pixels
PFT render / dense render	60.587	10	1499855
JPEG / PFT render	50.789	6	34531341
JPEG / dense render	50.431	11	34659923

5 Disclosure and reproducibility

The information disclosed in this paper is insufficient to reverse engineer Renderdon. Only experiment observations, analysis arithmetic and a small generic polynomial-loss example are disclosed. No Renderdon model weights, private synthesis implementation, training corpus, production pipeline, or complete engine reconstruction is provided. This is a disclosure-scope statement, not a formal security proof against all possible inference.

Appendix A is runnable with the Python standard library and regenerates every numeric table entry from embedded historical observations. It recomputes medians and ratios; error measurements are replayed from the audit, not remeasured from withheld tensors. It does not pretend that fixed historical wall times can be regenerated by running a timer on another machine. Appendix B exposes the relevant generic loss computation and an exact rational certificate without exposing the private renderer. A new performance experiment requires a fresh timed run and should be reported separately from this record.

The evidence-file SHA256 values are:

Dense results: `22db8549d8ae64ad9c402b1ef1673e0406a870e798d3167ba530dca2b06065b3`
Graphics results: `4e0b1dbbaaf0132563730746bc2f1ed994198160278656a7bf0d2ce3a3410972`

These bind the selected local records; they do not constitute independent replication. The original evaluators and environment metadata are identified in the bibliography, which is ordered by first citation.

6 Conclusion

The evidence supports efficient, numerically faithful contraction for this structured dense polynomial loss. It also records an 8K image-warp path whose advantage depends on the baseline: faster than direct dense evaluation but slower than optimized separable evaluation. An end-to-end dense PFT/HKD Inf Renderdon 8K speedup, improved visual accuracy, and general neural-generation applicability remain unestablished by these experiments.

References

[1] Local project benchmark archive. *Polynomial contraction audit: a dense toy network, not a video generator*. November 5, 2026. Results, source and review: `session_checks/pft_dense_review_20261105/results.json`; `session_checks/pft_dense_review_20261105/evaluate.py`; `session_checks/pft_dense_review_20261105/THEOREM_AND_REVIEW.txt`. Unpublished primary experiment record.

[2] Local project benchmark archive. *Matched 8K PFT/JPEG benchmark*. October 27, 2026, 09:48:18–09:49:42 UTC. `benchmark_results/pft_8k_matched_20261027/results.json`; `benchmark_results/pft_8k_matched_20261027/benchmark.py`; `benchmark_results/pft_8k_matched_20261027/REPORT.txt`. Unpublished primary experiment record.

A Exact table reproduction in Python

Copy the following verbatim into `reproduce_dense.py` and run `python3 reproduce_dense.py`. Output consists of the four tables' LaTeX body rows. The embedded arrays are raw recorded times in seconds; the accuracy fields are recorded maximum errors. No private input files, network access or renderer are required.

```
"""Replay recorded benchmark tables; no renderer is included.
>>> ratio(2.0, 4.0)
0.5
"""
import doctest
from statistics import median

def ratio(baseline, candidate):
    return baseline / candidate

DENSE = [{'n': 257,
  'setup': 0.03368895900000002,
  'raw': {'full': [0.0072830829999999125,
                   0.0068561660000003855,
                   0.006418166999999642,
                   0.006102708000000234,
                   0.006775625000000396],
          'thin': [0.005350374999999907,
                   0.0056638329999998405,
                   0.00494641600000012,
                   0.0052613749999999015,
                   0.005131708000000401],
          'moments': [0.007731875000000166,
                      0.007293374999999713,
                      0.00692237499999937,
                      0.007357041999999758,
                      0.0072424999999993744]},
  'error': {'parameter_error': 1.1102230246251565e-16,
            'gradient_error': 4.336808689942018e-18,
            'loss_error': 0.0}},
 {'n': 4097,
  'setup': 0.0004605419999998972,
  'raw': {'full': [0.03475529199999983,
                   0.028498124999999597,
                   0.02739633299999955,
                   0.02873804199999963,
                   0.026070374999999757],
          'thin': [0.00500125000000029,
                   0.00494037500000033,
                   0.005063459000000492,
                   0.004928374999999541,
                   0.004983458999999968],
          'moments': [0.00700491700000061,
                      0.006901957999999375,
                      0.006900082999999668,
                      0.006959874999999727,
                      0.0068544999999993195]},
  'error': {'parameter_error': 1.942890293094024e-16,
            'gradient_error': 1.1709383462843448e-17,
            'loss_error': 1.734723475976807e-18}},
 {'n': 16385,
  'setup': 0.0010479580000000155,
  'raw': {'full': [0.16376412500000015,
                   0.17857449999999986,
                   0.1570769160000003,
                   0.13543212500000035,
                   0.1369790420000001],
          'thin': [0.005101125000000373,
                   0.005082625000000007,
                   0.005340584000000703,
                   0.005018417000000497,
                   0.005009165999999787],
          'moments': [0.007172750000000505,
                      0.007393916999999917,
                      0.007121999999999851,
                      0.007024834000000091,
                      0.00739616600000037]},
  'error': {'parameter_error': 2.220446049250313e-16,
            'gradient_error': 5.551115123125783e-17,
            'loss_error': 1.734723475976807e-18}}]

GRAPHICS = {'dense_separable_jpeg': [0.9178477909999998,
                          0.9031223750000024,
                          0.9133937909999901],
 'sparse_separable_jpeg': [0.9156797499999954,
                           0.987748916000001,
                           0.9663650420000067],
```

```
 'dense_jpeg': [1.1462183340000038, 1.2084150419999986, 1.1072821670000081],
 'dense_separable_png': [3.462377458000006,
                         3.4483350419999965,
                         3.511143666999999],
 'sparse_separable_png': [3.6176293330000036,
                          3.461145917000003,
                          3.4634120830000086]}

QUALITY = {'render_vs_optimized_dense': {'psnr_db': 60.5872248610003,
                                         'max_channel_error': 10,
                                         'changed_pixels': 1499855,
                                         'pixel_exact': False,
                                         'quality_gate_passed': True},
 'decoded_vs_own_render': {'psnr_db': 50.78926009341028,
                           'max_channel_error': 6,
                           'changed_pixels': 34531341,
                           'pixel_exact': False,
                           'quality_gate_passed': True},
 'decoded_vs_optimized_dense': {'psnr_db': 50.4306971648192,
                                'max_channel_error': 11,
                                'changed_pixels': 34659923,
                                'pixel_exact': False,
                                'quality_gate_passed': True}}

def table_rows():
    dense, accuracy, graphics, quality = [], [], [], []
    for r in DENSE:
        m = {k: median(v) for k, v in r['raw'].items()}
        dense.append(
            f"{r['n']} & {1000*m['full']:.3f} & "
            f"{1000*m['thin']:.3f} & {1000*m['moments']:.3f} & "
            f"{ratio(m['full'],m['thin']):.3f} & "
            f"{ratio(m['moments'],m['thin']):.3f} & "
            f"{1000*r['setup']:.3f}")
        e = r['error']
        accuracy.append(
            f"{r['n']} & {e['gradient_error']:.3e} & "
            f"{e['parameter_error']:.3e} & {e['loss_error']:.3e}")
    cases = [
        ('Optimized dense JPEG', 'dense_separable_jpeg'),
        ('Direct dense JPEG', 'dense_jpeg'),
        ('PFT JPEG', 'sparse_separable_jpeg'),
        ('Optimized dense PNG', 'dense_separable_png'),
        ('PFT PNG', 'sparse_separable_png')]
    for label, key in cases:
        v = GRAPHICS[key]
        base = 'dense_separable_' + key.rsplit('_', 1)[1]
        graphics.append(
            f"{label} & {median(v):.6f} & "
            f"{min(v):.6f}--{max(v):.6f} & "
            f"{ratio(median(GRAPHICS[base]),median(v)):.3f}")
    for label, key in [
        ('PFT render / dense render', 'render_vs_optimized_dense'),
        ('JPEG / PFT render', 'decoded_vs_own_render'),
        ('JPEG / dense render', 'decoded_vs_optimized_dense')]:
        q = QUALITY[key]
        quality.append(
            f"{label} & {q['psnr_db']:.3f} & "
            f"{q['max_channel_error']} & {q['changed_pixels']}")
    return dict(dense=dense, accuracy=accuracy,
                graphics=graphics, quality=quality)

if __name__ == '__main__':
    assert doctest.testmod().failed == 0
    for name, rows in table_rows().items():
        print('% ' + name)
        for row in rows:
            print(row + r' \\')
```

B Public contraction and accuracy-check code

The following independent certificate runs with the Python standard library. It verifies all five polynomial moments exactly on a 17-point rational grid; it is a mathematical unit check, not a reproduction of hardware timings.

```
"""Exact polynomial-moment certificate.
>>> certificate()
True
"""
import doctest
```

```
from fractions import Fraction as F

def certificate():
    nodes = [F(-1), F(-1, 2), F(0), F(1, 2), F(1)]
    grid = [F(i, 8) - 1 for i in range(17)]
    def basis(j, x):
        value = F(1)
        for k, s in enumerate(nodes):
            if k != j:
                value *= (x-s)/(nodes[j]-s)
        return value
    weights = [sum(basis(j, x) for x in grid)/len(grid)
               for j in range(5)]
    for k in range(5):
        assert sum(x**k for x in grid)/len(grid) == sum(
            w*s**k for w, s in zip(weights, nodes))
    return True

if __name__ == '__main__':
    assert doctest.testmod().failed == 0
    assert certificate()
```

The core evaluator operations below are excerpted from the dense audit (with presentation-only spacing). In the source, `a,b` are dense trainable tensors, `weights` solve the five-moment linear system, and `moment_matrix` contains the empirical moments through degree four. The same dense Adam optimizer is applied to all three paths. This is a generic experimental model, not Renderdon's architecture.

```
# Full and five-node paths differ only in sample evaluation/weights.
output = (features[mode] @ a.T).square() @ b.T
per_sample = (output - targets[mode]).square().mean(dim=1)
loss = (per_sample.mean() if mode == 'full'
        else (per_sample * weights).sum())
loss.backward()
opt.step()

# Alternative conventional polynomial-moment loss:
coeff = b @ torch.stack([
    a[:, 0]**2, 2*a[:, 0]*a[:, 1], a[:, 1]**2], dim=1)
residual = coeff - target_coeff
loss = ((residual @ moment_matrix) * residual).sum() / 16

# Numerical checks after paired runs:
pe = float((full['params'] - thin['params']).abs().max())
ge = float((full['gradient'] - thin['gradient']).abs().max())
le = abs(full['loss'] - thin['loss'])
assert pe < 1e-10 and ge < 1e-10 and le < 1e-10
```