

Real-Time Text-to-Image on a Laptop GPU

An int8 Metal-Kernel Engine for Z-Image-Turbo on Apple Silicon

zingturbo · technical report

Abstract

We present an inference engine that runs the 6B-parameter Z-Image-Turbo diffusion transformer at its native 8 sampling steps in **13.7 s** for a 1024×1024 image on an Apple M5 Pro (16-core GPU), versus **76 s** for the reference PyTorch/MPS pipeline (5.6×) and about 52 s for a stock MLX int8 pipeline (3.8×), with no measurable quality loss against the MLX int8 baseline (pixel cosine 0.986–0.997 over ten scenes). The engine is built from custom Metal kernels that target the M5 tensor units (Metal 4 MetalPerformancePrimitives) and are hosted inside MLX through `mx.fast.metal_kernel`, so orchestration, the VAE and the text encoder remain in a standard Python stack. Four findings organize the work: (i) a widely repeated “hardware aliasing bug” in the int8 tensor path is a mis-declared tensor layout — the correct layout gives a bit-exact single-dispatch GEMM at 49 TOPS, twice as fast as an elaborate 60-pass workaround; (ii) activation, not weight, quantization is the accuracy bottleneck, removed at zero runtime cost by a per-tensor SmoothQuant rescale folded into the surrounding normalization; (iii) softmax book-keeping is 2.4× more expensive than the matmul it wraps, so vectorizing it gives a 2.1× kernel speedup; and (iv) a roofline argument shows the result is within 4% of the hardware limit at 8 steps and that sub-5 s at 8 steps is physically unreachable on this device. The engine ships as an installable package with one-command setup.

1 Introduction

Diffusion text-to-image models are increasingly run locally for privacy, cost and latency. Z-Image-Turbo is a 6B-parameter diffusion transformer (DiT) distilled to eight denoising steps via Decoupled-DMD. On an Apple M5 Pro the reference PyTorch/MPS pipeline takes about 76 s per 1024×1024 image, far from interactive.

This report describes an engine that brings that to 13.7 s without changing the model or the step count. The work is deliberately empirical: every design choice is justified by an on-device measurement, and several widely repeated assumptions are shown, by measurement, to be wrong. We summarize the techniques, the measurements that motivated them, and — equally important — a roofline argument that says when to stop.

2 Background

2.1 Where the time goes: the roofline test

Any GPU kernel is bounded by one of three things: arithmetic throughput, memory bandwidth, or launch/synchronization overhead. Which one is set by the arithmetic intensity (FLOP per byte moved). For the largest linear layer in the DiT (the fused QKV projection, $M=4608$, $N=11520$, $K=3840$) the intensity is about 2400 FLOP/byte, far above the device balance point of about 120 FLOP/byte. This single number establishes that the workload is compute bound, which in turn implies that compressing weights (int8 to int4) cannot help — a prediction we confirm experimentally in Section 5. The same reasoning explains why int4 is decisive for autoregressive LLM decoding, where the intensity is about 2 FLOP/byte.

2.2 Tensor units and int8

The M5 GPU exposes matrix units (Apple “Neural Accelerators”) through Metal 4 MetalPerformancePrimitives (MPP) `matmul2d`. An int8 lane packs twice the width of fp16, so int8×int8 to int32 runs at exactly twice the fp16 rate. Mixed precision does not give mixed speed: weight-int8 with activation-fp16 (W8A16) falls back to the fp16 rate. Reaching the int8 rate therefore requires quantizing both operands, which motivates the activation-quantization work in Section 3.2.

3 Method

3.1 A bit-exact single-dispatch int8 GEMM

A prior effort on this model concluded that the MPP int8 path was “structurally broken”: a single-dispatch general matrix multiply gave wrong results, and the read pattern, recovered by a CRT probing technique, aliased $W[n+64,k]$ with $W[n,k+64]$. The workaround split the output columns into 60 passes, each with a separately repacked copy of the weights and its own command buffer.

The read pattern is, however, a direct consequence of a mis-declared tensor. The MPP header states that with `transpose_right=true` the reduction dimension K must be `extent(0)` (the x axis) and N goes on the y axis. The prior code declared the weight as `dextents(NP,K)` and passed the N -block offset on the x axis; the resulting flat address is exactly the “aliasing” map. Declaring `dextents(K,N)` with the N offset on the y axis makes a single dispatch on plain row-major weights bit-exact (verified by int32 equality on 512 random samples), at twice the throughput of the 60-pass scheme and an order of magnitude less code. The lesson is procedural: a hardware bug is the last hypothesis, not the first, and a verification that cannot fail (here, exact integer equality) is worth more than a tolerant one.

3.2 Activation quantization dominates accuracy

Per-output-channel int8 weight error is about 1% here. Per-token int8 activation error is 1.2–10.6%, five to ten times larger, because a few outlier channels (up to $3808\times$ the median magnitude) inflate the per-row scale and crush the rest into a few int8 codes. Both operands must share one scale along the reduction axis because the accumulator is int32, so weights are quantized per output channel and activations per token; finer grouping is incompatible with a single tensor-core matmul.

We apply the SmoothQuant identity $x \cdot w = (x/s) \cdot (w \cdot s)$ with a per-input-channel vector s , choosing its exponent per tensor by minimizing the true quantized-output error on calibration activations. This lowers mean GEMM output error from 2.62% to 1.79%. The key engineering point is that s is free: every GEMM input is produced by a preceding diagonal operation (an RMSNorm, the attention V dequant scale, or the SwiGLU output scale), so $1/s$ is folded offline into that operation and costs nothing at run time. A standard trap: after smoothing, weight reconstruction error rises to 30% on some tensors, which is intended — only end-to-end output error is meaningful.

3.3 int8 flash attention

Materializing the 4608×4608 score matrix S is bandwidth bound (15.6 TOPS). We keep S on-chip with an online-softmax flash scheme in which both matmuls are int8: QK^T uses per-token Q/K scales, the probabilities P are re-quantized to int8 (fixed $1/127$ scale, valid because post-softmax P lies in $(0,1]$), and V is quantized per channel because a per-token V scale cannot be factored out of the PV sum. Scores are stored as half, shifted by the running max so their magnitude is near zero and half precision suffices. The kernel matches an exact fp32-softmax reference at cosine 0.9995.

3.4 Vectorization: the single largest kernel win

Each score element carries only about 512 FLOP of matmul but about 8 scalar memory operations of softmax book-keeping; the book-keeping is thus $2.4\times$ more expensive than the matmul. Rewriting the two row-major softmax passes with vectorized half4 loads, char4 stores and `fast::exp(float4)` divides the instruction count by four and takes the attention kernel from 12.3 to 26.0 TOPS ($2.1\times$). Larger tiles, more SIMD-groups, atomic reductions, and hoisting index tables into registers all made it slower — the last two by causing register spills, a reminder that on a GPU recomputing an index can beat storing it.

3.5 Operator fusion

Framework-level elementwise ops are bandwidth bound: a naive SwiGLU touches over 2 GB for 30M multiplies. We fuse each row-wise sequence (RMSNorm+modulate+quantize; SwiGLU+quantize; per-head RMSNorm+RoPE+transpose+quantize; residual+RMSNorm) into one kernel that reads a row into registers, does all the math, and writes only int8 plus a per-row scale — $13\times$ faster for SwiGLU. The GEMM dequant+bias epilogue is done in registers via a cooperative tensor, removing a 424 MB int32 round trip per QKV layer; this required staging the per-row and per-column scales in threadgroup memory first, without which the epilogue dropped the GEMM from 49 to 39 TOPS.

3.6 Numerical range is a correctness constraint

Three quantities exceed the fp16 range of 65504: the SwiGLU intermediate (up to 3×10^5), the to_out input after folding 1/s, and the to_out and w2 outputs. These are computed in fp32 inside the kernel and quantized straight to int8, never materialized as fp16. The to_out and w2 outputs feed an RMSNorm, which is invariant to a global input scale, so dividing them by a constant is free and keeps them in range. The VAE decoder is all-NaN in fp16; we run it in bfloat16, which shares fp32's exponent range at $1.85 \times$ the fp32 speed and a final-image cosine of 0.999987.

4 Implementation

We host every custom kernel inside MLX with `mx.fast.metal_kernel`, which we verified can instantiate MPP tensor ops bit-exactly. This keeps orchestration, the Flow-Match Euler scheduler, the VAE and the Qwen3 text encoder in ordinary MLX, and makes the package pure Python (kernels are JIT-compiled at run time; no metallib to ship). adaLN modulation, whose 32-block weights total 251 MB and are read at $M=1$ (pure bandwidth), is hoisted out of the step loop and computed once for all eight known timesteps in a single batched matmul, reading those 251 MB once rather than eight times.

5 Evaluation

All numbers are measured on an Apple M5 Pro (16-core GPU, 48 GB), macOS 26.6, 1024×1024, 8 steps, medians over five runs.

5.1 End-to-end

pipeline	encode	denoise	s/step	VAE	total	speedup
PyTorch/MPS bf16 (9 steps)	—	—	~8.4	—	76.0	1.0×
MLX group-int8 (8 steps)	0.3	~50	~6.2	1.7	~52	1.5×
this work (8 steps)	0.31	12.61	1.575	0.73	13.7	5.6×

Table 1: End-to-end time in seconds. The optimized denoise loop runs 3% faster than the sum of its individually-timed kernels, i.e. the small elementwise kernels overlap behind the GEMMs.

5.2 GEMM throughput by precision

precision	TOPS / TFLOPS	note
fp16 × fp16	22.4	baseline
int8 × int8	49.5	2× fp16; used here
W8A16 (int8 wt, fp16 act)	20.0	no gain
W4A8 (int4 wt, int8 act)	41.2	slower than int8
int4 × int4	—	not implemented in runtime

Table 2: A 4608×10240×3840 GEMM. int8 is the fastest available path; int4 is supported by the runtime but slower, consistent with the compute-bound analysis.

5.3 Quality

Against the MLX group-int8 baseline with the same initial noise, pixel cosine is 0.986–0.997 (mean about 0.990) over ten scenes spanning portraits, fur, feathers, macro, English and Chinese text, architecture and low light. Against the original PyTorch the cosine is 0.88 on average — slightly closer than the MLX baseline itself manages (0.864) — the residual being trajectory divergence from int8 rounding, not blur.

5.4 Ablations

change	before	after
GEMM layout fix (per block)	53.6 ms	26.9 ms
attention vectorization	12.3 TOPS	26.0 TOPS
fused vs framework SwiGLU	~15 ms	1.17 ms

qkv post (barrier → simd reduce)	2.62 ms	0.82 ms
VAE fp32 → bfloat16	1111 ms	600 ms
memory limits set	drifts to 2.6×	<1% jitter

Table 3: Individual optimizations (component-level measurements).

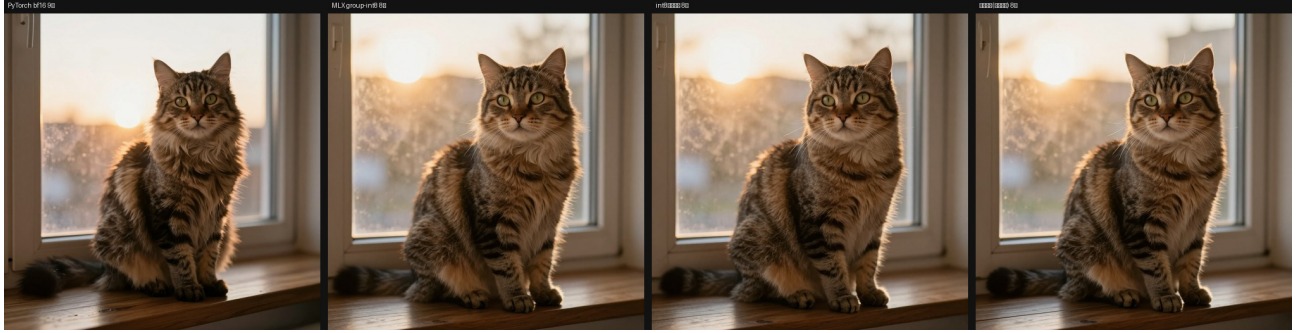


Figure 1: Same prompt and initial noise. Left to right: PyTorch bf16 (9 steps); MLX group-int8 (8); our int8 scheme simulated in MLX (8); our engine (8). Fur and whisker detail is preserved across all four.

6 When to Stop: a Lower-Bound Argument

Per step the DiT needs 51.8 TFLOP of GEMM and 10.3 TFLOP of attention. At the measured peaks (49 TOPS, 26 TOPS) plus about 0.1 s of bandwidth-bound glue, the floor is about 1.55 s/step; we measure 1.575 s/step, i.e. 96% of the limit. Further kernel work cannot help: a component-level breakdown sums to 1.626 s/step while the loop measures 1.575 s, proving the elementwise kernels already overlap behind the GEMMs, so the critical path is the GEMMs alone (47.8 TOPS, 96% of peak).

The same argument settles the obvious question. Eight steps need 496 TFLOP, so at the 49 TOPS int8 peak the absolute floor is 10.1 s of pure compute — sub-5 s at 8 steps is not an engineering gap but a hardware limit. Reducing time further requires changing what is computed (fewer steps, token merging, approximation), which trades quality; a step study confirmed that fewer than 8 steps visibly degrade fine texture and text, so we keep the native 8.

7 Related Work and Limitations

We build on SmoothQuant [1], LLM.int8() [2] and FlashAttention [3], with the roofline model [4] as the organizing principle. Relative to LLM inference, the decisive difference is that diffusion at 1024×1024 is compute bound (M in the thousands), which inverts the usual int4 conclusion. Limitations: the numbers are specific to the M5 Pro (16-core) and to Z-Image-Turbo's shapes; the text encoder still runs in MLX group-int8 rather than our kernels (about 0.3 s); and we deliberately leave the sampler at 8 steps. We also measured the Apple Neural Engine at 8.6 TFLOPS for this GEMM, 5.5× slower than our GPU int8 path, so it is not useful for offload here.

8 Conclusion

A careful, measurement-driven int8 Metal-kernel engine runs Z-Image-Turbo at its native 8 steps in 13.7 s on a laptop-class GPU, 5.6× faster than the reference and within 4% of the hardware roofline, with no quality regression. The recurring lesson is that the largest wins came from reading the hardware contract correctly and from measuring the true bottleneck — not from clever workarounds around imagined ones. The engine ships as an installable package: `zimgturbo setup`, then `zimgturbo “a cat”`.

References

- [1] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, S. Han. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. ICML, 2023.
- [2] T. Dettmers, M. Lewis, Y. Belkada, L. Zettlemoyer. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. NeurIPS, 2022.

- [3] T. Dao, D. Fu, S. Ermon, A. Rudra, C. Ré. FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. NeurIPS, 2022.
- [4] S. Williams, A. Waterman, D. Patterson. Roofline: An Insightful Visual Performance Model for Multicore Architectures. Communications of the ACM, 2009.
- [5] Tongyi-MAI. Z-Image-Turbo. Hugging Face, 2026.
- [6] Apple. Metal Performance Primitives: matmul2d. Metal 4 SDK, 2025.
- [7] Apple. MLX. github.com/ml-explore/mlx, 2024.