Gerbil: A Pure-WGSL WebGPU Inference Engine for Private, Key-Free, On-Device Language and Speech Models

Eyal Toledano
eyal@tryhamster.com

July 23, 2026

Abstract

On-device language-model inference costs nothing per token, keeps data private, runs offline, and needs no API key. Yet the dominant browser stacks reach the GPU through a translation layer (an ONNX runtime or a deep-learning compiler) that is heavy to ship and brittle on mobile Safari. We present **Gerbil**, a WebGPU inference engine written directly in WGSL: a runtime intermediate representation lowers to hand-written compute shaders that execute *the same compute* in a browser and under Node via Google's Dawn, with a selective safetensors loader and INT4 weights.

In a same-base-model head-to-head (Qwen3.5-0.8B, each engine in its own native 4-bit format), Gerbil's hand-written WGSL stack decodes 2.1× faster than WebLLM's compiler-generated stack on desktop and 1.5× faster on an iPad, and is the only one of three engines that runs the model at all on an iPhone, where WebLLM OOM-crashes and a WASM/CPU engine cannot load it. This advantage reflects the *combined* effect of several design choices that differ across the engines at once (kernel authorship, quantization format, KV-cache layout, op-fusion schedule, and memory management), not kernel authorship in isolation; we are explicit that decomposing the format-versus-kernel contribution would need a matched-format ablation we do not run. We trace the regime that makes these levers matter to our central characterization, a measured anatomy of the decode token: on desktop the cost is kernel occupancy and workgroup geometry plus a fixed, reclaimable per-submit overhead—not raw FLOPs and not dispatch count—while on mobile WebKit every dispatch is a submit-and-drain, making dispatch count the binding constraint and kernel fusion the dominant lever there. Owning the entire forward pass further enables a GPU-side vocabulary-pruning trick with byte-identical output, adapter-based "flavors" that specialize a shared cached base from a few-megabyte LoRA (merged into the base before quantization, so decode carries no adapter overhead), three FFT-free bit-exact neural-codec speech engines (one of them a text-prompted encoder-decoder), and a deployment method under which 0.5–1.5 GB models load on iOS, where the platform CDN otherwise stalls (a quantified multi-device success-rate sweep remains future work). Gerbil shows that a hand-written WGSL engine, with no external inference runtime, is a practical foundation for private, key-free, on-device language and speech models across the desktop and mobile web.

1 Introduction

Running a language model on the user's own device changes its economics and its privacy posture at once. There is no per-token cost, no server to operate, no data leaving the device, and no API key to provision; the model works on a plane or behind a firewall. The browser is the most widely deployed application runtime on earth, and *WebGPU* now exposes its GPU for general compute [W3C, 2024]. The question this paper answers is how to build a *good* on-device inference engine on that substrate.

The established browser stacks answer it indirectly. TRANSFORMERS.JS reaches the GPU through ONNX Runtime Web [Hugging Face, 2023, ONNX Runtime authors, 2024]; WEBLLM reaches it through a deep-learning compiler (TVM/MLC) [Ruan et al., 2024, MLC team, 2023, Chen et al., 2018]. Both interpose a general-purpose translation layer between the model and the shader. That layer is large to ship and, as we document in §2, fragile on mobile Safari, where memory limits and CDN quirks turn "it works on desktop" into "the tab dies." Our position is that for a fixed, small family of on-device models the translation layer is not worth its cost: writing the compute shaders directly is *less* code, ships smaller, and, because the engine owns the whole forward pass, unlocks systems optimizations a black-box runtime cannot reach.

Gerbil is that engine. A runtime intermediate representation (IR) describes a model as a graph of typed tensor ops; an architecture registry generates the IR from a HuggingFace config; an executor lowers each op to a hand-written WGSL compute shader and dispatches it on WebGPU. The exact same TypeScript and WGSL run in a browser and under Node (via Dawn), so a measurement taken in one is meaningful in the other. Weights are loaded selectively from safetensors and quantized to INT4.

Contributions.

- **A pure-WGSL, runtime-free engine** (§3): runtime IR → WGSL → WebGPU, identical compute in browser and Node/Dawn, with a selective safetensors loader and INT4 weights; greedy output is byte-identical across runtimes.
- **Flavors: adapter-based specialization over a shared base** (§3): a tuned model ships not as a full checkpoint but as a few-megabyte PEFT LoRA adapter [Hu et al., 2022] over a base that is cached once, so a new specialization is a megabyte-scale artifact rather than a gigabyte-scale download. Adapters load on the fly (`loadModel(id, {adapter})`) and hot-swap on an already-warm engine and base (`loadAdapter`/`getAdapter`) without reloading or re-downloading the multi-hundred-megabyte base. The correction $W' = W + (\alpha/r)\,BA$ is realized by a *merge-at-load* path that folds the adapter into the raw float weights *before* INT4 quantization, so decode carries zero adapter overhead and a fused flavor runs byte-for-byte like any other checkpoint.
- **A measured anatomy of the decode token** (§4): whole-pass GPU timestamps and command-buffer-doubling experiments decompose the ~ 3.3 ms token into $\sim 92\%$ kernel execution (governed by workgroup occupancy and geometry at production shapes, with a $+27\%$ single-fix occupancy repair) and a fixed ~ 0.48 ms per-submit overhead that batching amortizes; on WebKit, per-dispatch submits make dispatch count the constraint instead, splitting the optimization playbook by platform.
- **Vocab-pruned decoding with a GPU argmax remap** (§5): a kept token-id prefix plus a special/byte-fallback tail, with the index→token-id remap performed *inside* the argmax kernel so the no-readback pipelined path survives; +10–16% WebKit / +8% Dawn, byte-identical.
- **FFT-free on-device neural-codec TTS** (§6): three TTS engines on one convolutional kernel family, a NanoCodec (FSQ + causal HiFi-GAN), an IBM DAC decoder, and the first encoder-decoder backbone in the engine (a Flan-T5 text encoder cross-attending into a delay-pattern Parler audio decoder), all validated bit-exact (Parler waveform NCC 1.000 vs PyTorch), with a second tokenizer family (Unigram/SentencePiece, 1110/1110 exact ids) and INT4 weights by default.
- **A same-base-model, native-format head-to-head** (§8): on one base model (Qwen3.5-0.8B, each engine in its own native 4-bit format), the Gerbil stack beats the compiler-

generated peer by 2.1× on desktop (178.8 vs. 84.5 tok/s) and 1.5× on iPad (46.3 vs. 30.1); the WASM/CPU engine cannot load the model at all and WebLLM OOM-crashes the iPhone (3/3), so Gerbil is the only one of the three that runs the 0.8B across desktop, iPad, *and* iPhone (a reliability result, not just speed). The throughput ratio is a whole-stack difference, not an isolated kernel-authorship effect (§8).

- **A mobile-deployment method for the iOS cold-load stall** (§7): a self-hosted weight mirror with fallback plus a durable, size-verified cache that makes 0.5–1.5 GB models load on iOS where the upstream CDN stalls; we demonstrate the before/after on our test devices and flag a quantified multi-device success-rate sweep as future work.

The closest peer with a paper, WebLLM [Ruan et al., 2024], is a positioning preprint that compares one compiler-generated engine against its own native backend. We position differently (hand-written WGSL versus compiler-generated kernels, and an on-device engine that owns its forward pass) and we report the kind of systems detail (a dispatch-count characterization, a regression-to-win ablation, bit-exactness gates) that the genre usually omits.

2 Background and Motivation

WebGPU and WGSL. WebGPU [W3C, 2024] exposes the GPU through command buffers of *dispatches*: each compute pass binds a pipeline and a bind group, then dispatches a workgroup grid. WGSL is its shading language. Unlike CUDA, there is no mature library of fused transformer kernels; an engine either generates shaders through a compiler or writes them by hand. We write them by hand.

The motivating failure. Our prior stack used transformers.js over ONNX Runtime Web. On desktop it worked; on iOS it did not. Three forces compounded. First, a WebKit/WKWebView heap cap near 300–400 MB collided with the resident model plus the runtime's own buffers. Second, the runtime bundle was tens of megabytes before any model. Third, a consumer bundler could rewrite the runtime's dynamic imports into an on-demand chunk loader that hung at cold load on iOS WebKit. The measured throughput gap was the final argument: on an iPad, the ONNX path reached only ∼6.9–11.6 tok/s against ∼51 tok/s for native compute on the same device, a ∼5× penalty for the translation layer. Owning the stack removes all four problems at once, at the cost of writing the kernels ourselves. The shipped engine bundle is ∼90 KB gzipped (∼418 KB minified), against the multi-megabyte download a general-purpose ONNX or compiler runtime adds before any model (Figure 1).

3 System Architecture

Gerbil is organized as a small pipeline: **loader** → **IR** → **executor** → **WGSL kernels** → **tokens**.

Runtime IR. A model is a `ModelGraph` of `OpNode`s over `TensorDesc`s, with a small `OpType` taxonomy (embedding, matmul, INT4 matmul, RMSNorm, RoPE, attention, SwiGLU, elementwise) and a canonical tensor-naming scheme. The IR is built *at runtime* from the HuggingFace config by an architecture registry, not by an offline converter, so adding a model family is a graph generator, not a rebuild of an ONNX export.

Selective safetensors loader. The loader parses the safetensors header and creates zero-copy typed views, fetching *only* the tensors the graph references and streaming weights through a low-peak store so the JS heap never holds a second copy on top of the GPU buffers

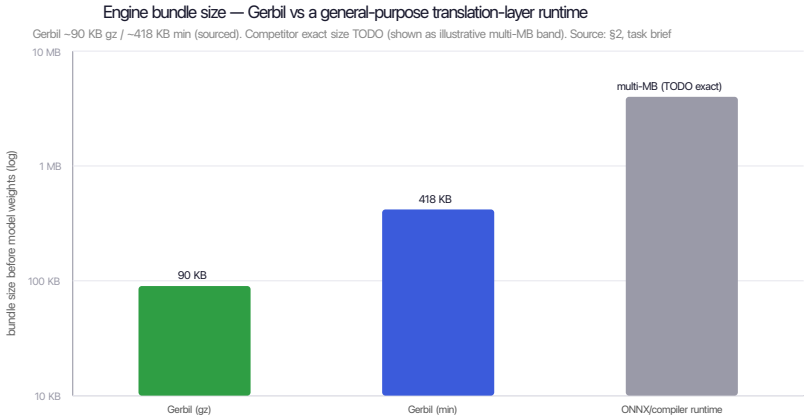

Figure 1: Engine bundle size. Gerbil ships ∼90 KB gzipped (∼418 KB minified) of inference code; the ONNX-web and compiler-runtime stacks it replaces add multiple megabytes before any model weights. Competitor exact sizes are [**TODO:**] pending a measured build cross-check. Source: §2, task brief; `paper/results/measurements.csv`.

(the property that, on iOS, is the difference between loading and OOM). INT4 weights are dequantized inside the matmul kernel.

Executor and the same compute everywhere. The executor allocates GPU buffers with liveness pooling (intermediates share buffers once their producers retire), then for each op binds the registered WGSL pipeline and records a dispatch. The KV cache is GPU-resident in an LHSd layout. Because the executor and every shader are plain TypeScript and WGSL, the identical code path runs in a browser and under Node via Dawn; we use this for a strong correctness invariant: greedy (temperature-0) decode is **byte-identical** across Dawn and WebKit on the same model, which we rely on throughout the evaluation as a parity oracle.

Op fusion in the executor. Because the executor owns the whole graph, it pattern-matches op sequences and substitutes single fused kernels during lowering: the gate_proj/up_proj/SwiGLU MLP chain collapses into one dispatch, the per-layer residual-add and RMSNorm fuse into a single `ResidualRMSNorm`, the two KV-cache appends fuse into one, and the depthwise conv's rolling-state update folds into the conv dispatch itself (at $T{=}1$ decode, the conv read and state write for a channel live in the same thread, so the fusion is race-free; prefill keeps the two-dispatch path). Column-block inputs fuse further still: LFM2.5's projection emits one $B|C|x$ tensor, and its conv kernel consumes the blocks in place—computing the $B{\cdot}x$ pre-gate per tap and applying the C gate in-kernel—which deletes the slice and multiply dispatches *and* their intermediate tensors. Together these cut Qwen3.5-0.8B decode from ∼342 to ∼269 dispatches per token (Figure 3) and LFM2.5 from 164 to 121. One constraint governs *where* a fusion must be applied: the pooled activation allocator reuses buffer slots across dispatch boundaries, and WebGPU rejects a bind group that binds one buffer both read-only and writable—so a fusion whose combined lifetime spans a pooled reuse must be expressed at the *graph* level (the architecture emits the fused node and the pool allocates around it), not patched into the executor's dispatch list afterward. Every removed dispatch matters more on mobile than on desktop, where each dispatch is its own command-buffer submit and drain (§4).

A hand-written-kernel lesson: `num_workgroups`. Writing the shaders by hand exposes pitfalls a compiler would hide. One we hit: reading the WGSL `num_workgroups` builtin inside a kernel forces Dawn on Metal onto a slow per-dispatch uniform path, because the driver must materialize the grid dimensions into an implicit uniform buffer on every dispatch. Passing the grid stride as an ordinary pipeline parameter instead of reading the builtin avoids that path. This is the kind of low-level lesson that owning the kernels surfaces and a black-box runtime would absorb silently.

Flavors: adapters over a shared base. Owning the load pipeline also makes model *adaptation* cheap at inference. A Gerbil *flavor* is a small PEFT LoRA adapter [Hu et al., 2022] (a few megabytes: `adapter_config.json` plus `adapter_model.safetensors`) trained on one of the base models. It is loaded on the fly—either at model load (`loadModel(id, {adapter})`) or hot-swapped onto an already-warm engine and base (`loadAdapter(repo)`; `loadAdapter(null)` drops back to the plain base and `getAdapter()` reports the active flavor)—resolving from a HuggingFace repo, a URL, or a local `file:` directory. Rather than ship a full fine-tuned checkpoint per tune (~ 1.6 GB), the base is cached once (and, being megabytes, a flavor never re-downloads it on a swap); the adapter is then folded into the base weights *at load time*,

$$W' = W + \frac{\alpha}{r} BA,$$

with rank-stabilized scaling ($\alpha/\sqrt{r}$), per-module `rank_pattern`/`alpha_pattern`, and `fan_in_fan_out` all honored from the PEFT config. The systems point is *where* the merge runs: on the raw HuggingFace-layout canonical weight tensors—after the selective loader materializes them but *before* any architecture-specific transform (the Qwen3.5 q/gate deinterleave, the MLX/GPTQ repack above) and before the executor's INT4 quantization. Because the delta is applied in the base tensor's original layout, every downstream transform and the quantizer treat a fused checkpoint identically to any other, so decode carries **zero** adapter overhead: no auxiliary low-rank matmul, no separate adapter matrices, no branch in the hot path. This is the opposite trade from server-side multi-LoRA serving (as in vLLM), which keeps adapters *separate* precisely so one base can serve many tunes at once; the on-device case serves a single flavor at a time, so merging carries no serving penalty—the adapter's runtime cost is zero, and hot-swapping a flavor merely re-merges from the cached base (adapters are megabytes; the ∼GB base is never re-downloaded). Adapter target keys are canonicalized through the *same* key mapper the base load uses, so Qwen, Gemma, and LFM2 layouts all resolve to their base tensors; merging targets float bases (every Tune output is 16-bit safetensors, in contrast to QLoRA-style finetuning over an already-quantized base [Dettmers et al., 2023]) and skips—with a warning, not silent corruption—any target whose base is a pre-quantized pack. The path is validated in two directions (`scripts/engine/test-lora.mjs`): a synthetic adapter reproduces $W + (\alpha/r)\,BA$ to within the committed 10^{-5} gate (the merge is exact float arithmetic, so the residual is rounding-level), and a real Qwen3.5-0.8B loaded through the adapter path with a zero adapter (B=0) generates *byte-identically* to the plain base, with a swap back via `loadAdapter(null)` reproducing the same output—so the adapter code path never perturbs a normal load, and because it merges pre-quantization it leaves every decode and bit-exactness result in this paper unchanged.

4 Anatomy of a Decode Millisecond

The load-bearing systems fact about this engine is *where* a decode token's time actually goes, because the answer is different on desktop and on mobile—and both differ from the folk model that GPU inference is bandwidth-bound. Each decode token executes a command buffer of ~ 269 dependent GPU dispatches (after fusion; ~ 342 before). On an M4 Max

under Node/Dawn, direct whole-pass GPU timestamps decompose the 3.34 ms wall-clock token ($\approx$ 300 tok/s) into 2.87 ms of GPU pass execution ($\sim$ 92% busy) and $\sim$ 0.48 ms of per-submit machinery (queue submission, driver scheduling, and the 4-byte token readback map).

The kernel side is occupancy- and shape-bound, not dispatch-bound. Microbenchmarks of the production INT4 matvec kernel at production shapes give the per-dispatch cost model: fixed cost is $\sim 1.5\,\mu$s for tiny elementwise ops and $\sim$ 4–7 μs for matvec-sized ones; *dependent* dispatches carry no measurable drain penalty over independent ones, and pipeline switches between dispatches are free. Consequently, removing 18 small dispatches per token moves desktop wall-clock by zero—on Dawn, dispatch *count* is not the desktop constraint. What dominates instead is per-kernel efficiency, which scales with total dispatch size: at N=1024/K=1024 the matvec streams weights at only $\sim$16% of DRAM peak (ramp-bound), rising to $\sim$60% at N=6144 and for the 248k-row LM head. Workgroup *occupancy* pathologies are the largest single class of loss: the linear-attention state-recurrence kernel originally dispatched one workgroup per head —16 workgroups on a 40-core GPU—leaving most of the chip idle for a quarter of every token; fanning it out to one workgroup per (head, state-column chunk), exploiting that each state column has exactly one owner (no cross-workgroup synchronization, prefill included), was alone worth +27% end-to-end. With that repair and two fused-matvec workgroup-geometry corrections, decode sits $\sim 4\times$ above its weight-bandwidth floor ($\sim$1230 tok/s for 444 MB/token at 546 GB/s), with the residual concentrated in small-shape kernel ramp and the unpruned LM head.

The per-submit overhead is measurable and reclaimable. The 0.48 ms/token of non-GPU time attaches to the *submit*, not the work: encoding the entire 287-dispatch pass *twice* in one command buffer raises wall time from 3.34 ms to 6.21 ms—exactly one extra 2.87 ms pass, with the overhead appearing once. Batching K chained greedy steps per command buffer therefore amortizes it K-fold (a $\sim$+12% ceiling at K=4); realizing the ceiling requires the per-step position-dependent uniforms to ride pre-built bind-group variants rather than mid-buffer copy commands, since each compute→blit→compute encoder switch on Metal costs $\sim 10\,\mu$s and $\sim$36 of them cancel the saving.

Recording cost and the GPU-driven decode loop. A naive decode loop also pays a per-*call* recording cost that differs by runtime: in the browser, `setPipeline`/`setBindGroup`/`dispatchWorkgro` calls are in-process (nanoseconds each), while under Node each crosses the `webgpu` N-API boundary (microseconds each)—enough to hold Node decode to $\sim$145 tok/s while the same hardware reached $\sim$225–350 tok/s in Chrome. The engine closes this gap structurally rather than per-call: the pipelined greedy path chains the argmax result into `input_ids` *on the GPU*, keeps up to `PIPELINE_DEPTH` steps in flight, and reads tokens back one step behind, so command-buffer recording overlaps GPU execution and Node/Dawn matches in-browser throughput.

Ruled-out levers. We measured and rejected the obvious knobs, each within noise on desktop: the Dawn `skip_validation` toggle (+4%), raising `PIPELINE_DEPTH` from 2 to 8 (+7%, so readback latency is not dominant), software prefetch in the INT4 matvec K-loop ($-$1.2%: the Metal compiler already pipelines the loads), manual instruction-level parallelism in the same loop (dual accumulators $-$4.5%, 2$\times$ unroll $-$1.1%: the compiler owns this level), split-K fan-out of small matvecs (worse: per-workgroup ramp shrinks below usefulness), packing scale/zero metadata as f16 across all eight INT4 kernels (+1.2%, flat),

and subgroup reductions (large regression under Dawn). A recurring methodological result: isolated-kernel microbenchmark wins repeatedly failed to transfer to the production pass, whose dispatch overlap already supplies the memory-level parallelism the isolated wins exploit—every keep/revert verdict therefore comes from interleaved in-engine A/B runs.

Mobile is the dispatch-bound regime; the levers split accordingly. On WebKit, every dispatch is a command-buffer submit and drain, so dispatch count *is* the binding constraint there—op fusion is worth 2–3× more on mobile than on desktop, and the fusion waterfall of Figure 3 is primarily a mobile investment. We validate this asymmetry directly on-device, first with a single controlled fusion—folding the depthwise-conv state update into the conv dispatch removes 8 of 164 dispatches (4.9%) for LFM2.5-230M, measures *exactly neutral* on desktop Dawn, and improves iPhone greedy decode +6 to +10% (37.5 → 39.6/41.1 tok/s, same page and weights, minutes apart), agreeing with the 1/dispatches prediction of +5.1%—and then with a fusion campaign that takes LFM2.5 from 164 to 121 dispatches per token (−26%; sampled decode 66 → ∼100 tok/s sustained on the same iPhone). Across both model families the phone obeys a simple law once command-buffer grouping is saturated: tok/s ≈ 12,000/dispatches per token (LFM2.5: 121 ⇒∼ 100; Qwen3.5-0.8B: 263 ⇒∼ 44), which converts any proposed fusion directly into a predicted mobile speedup. Grouping itself is model-dependent: an on-device ladder sweep shows LFM2.5 (121 dispatches) saturating at ∼32 dispatches per command buffer while Qwen3.5-0.8B (263) keeps gaining to ∼1024 (19 → 45 tok/s from group 32 to 1024, $\sigma < 0.5$ across trials)—so the engine's crash-surviving batching probe targets 1024, and a phone reaches its optimum with zero configuration. On desktop the levers that remain are occupancy and geometry at production shapes, submit amortization as above, and *speculative decoding* [Leviathan et al., 2023, Chen et al., 2023], which amortizes all per-token fixed costs across several accepted tokens by verifying many drafts in one batched forward (a batched-M matvec is also far more bandwidth-efficient than M=1). Self-speculative drafting in the MEDUSA [Cai et al., 2024]/EAGLE [Li et al., 2024] family is an unusually good fit because Gerbil owns the forward pass and the hidden states, and a tree-verified scheme [Miao et al., 2024] maps onto the same batched-verify shape. We explicitly reject a CUDA serving path: it is a different product that dilutes the on-device, no-server identity. Figure 2 shows the cross-runtime decode rates this characterization explains.

5 Vocab-Pruned Decoding with a GPU Argmax Remap

The single largest decode operation is the LM-head matmul plus argmax over a ∼248K-row vocabulary. **Lever B** prunes that head to a kept prefix so the decode GEMV and the argmax run over a much smaller vocab: a pure decode-throughput lever, opt-in and off by default.

The tied-head reframe. The shipping default (Qwen3.5-0.8B MLX 4-bit) has a *tied* head: `lm_head` is absent and the loader shares the input embedding table for output projection. On a tied model, untying to prune *adds* resident memory while speeding decode, so Lever B is a decode-TPS lever, not an out-of-memory fix (the OOM was already solved by §7). It therefore ships device-gated: untie+prune on memory-rich devices, stay tied on memory-tight ones.

Mechanism. For a tied model the head is untied, re-quantized to INT4, and sliced to a kept prefix $[0, \text{keepN})$ plus a mandatory tail of every special/added token and all 256 byte-fallback tokens, so any string is still emittable (worst case byte-by-byte). The input embedding stays full; only the output projection shrinks. The kept set is the frequency-ranked prefix, and the quality cost of the +8–16% speed is a hard coherence floor: greedy output stays byte-identical to the unpruned reference down to a 64k kept prefix, then diverges at 32k. Speed saturates

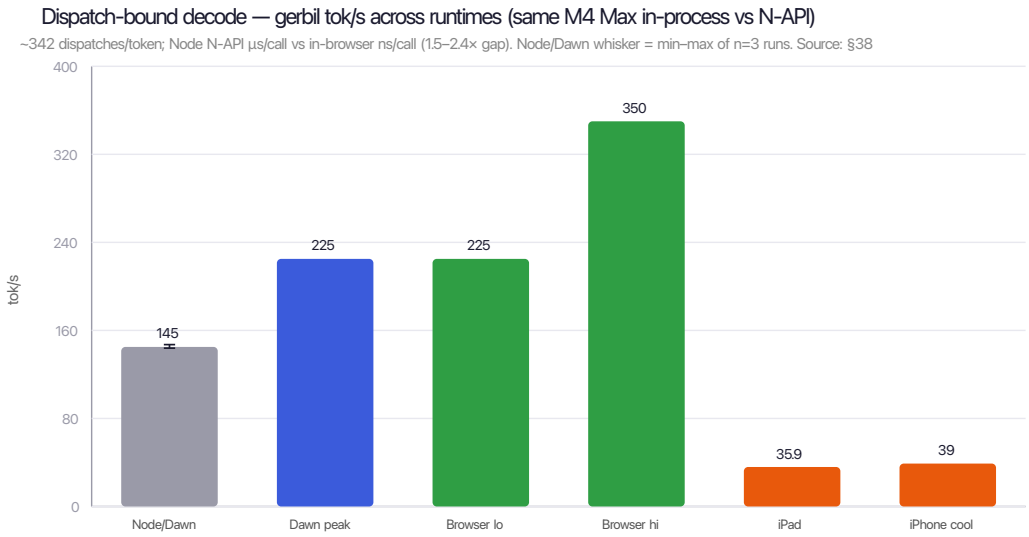

Figure 2: Decode throughput across runtimes on the same hardware. A naive Node loop pays per-call N-API recording cost (its $\sim$ 145 tok/s appears as the post-fix baseline in Table 2, whisker = min–max of n=3 runs); the GPU-driven pipelined decode closes that gap, and mobile rates reflect the WebKit submit-per-dispatch regime. Source: §4; `paper/results/measurements.csv`.

near 64k, so a general-assistant keep of 131072 clears the quality gates while roughly halving the head and sitting comfortably above the floor. This is a deliberately *static, prefix* pruning of the output projection, distinct from the classical, *dynamic, input-conditioned* vocabulary-selection line in neural machine translation [Mi et al., 2016, L'Hostis et al., 2016], whose pitfalls [Domhan et al., 2022] mirror our coherence-floor finding, and orthogonal to softmax-acceleration methods [Grave et al., 2017] that keep every token reachable. The tension with vocabulary scaling laws [Tao et al., 2024] is exactly what the device gate manages.

The GPU argmax remap (the result). The first cut (1.1.0) paired the prune with a CPU-side index remap, which forced the slower per-token `forwardArgmax` path: a pruned head no longer matched the GPU-chained pipelined decode loop, so off WebKit it was a *regression* on Dawn even as it sped up WebKit. The fix (1.1.1) makes the **GPU argmax kernel itself remap** its winning index back to the real token id (via `prefix_len`/`tail_start` parameters; identity when not pruning). A pruned head now yields real token ids on *every* decode path, including the no-readback pipelined path Dawn uses, so the forced fallback is dropped and prune helps Dawn instead of hurting it. The regression-became-a-win is the cleanest evidence that the bottleneck is the pipeline shape, not the arithmetic. Vision/multimodal loads skip pruning entirely, because image captioning reaches mid-vocab tokens the prefix drops. Table 1 and Figure 4 report the validated A/B, and Figure 5 the coherence-versus-speed trade with the 64k/32k floor marked.

6 On-Device Neural-Codec Text-to-Speech

Gerbil synthesizes speech with a codec language model that emits audio tokens, decoded to a waveform by a *convolutional neural codec* [Wang et al., 2023]. The codec decoders are deliberately **FFT-free**: they avoid an iSTFT/spectrogram front-end, which keeps them inside the pure-WGSL kernel set and off any external runtime.

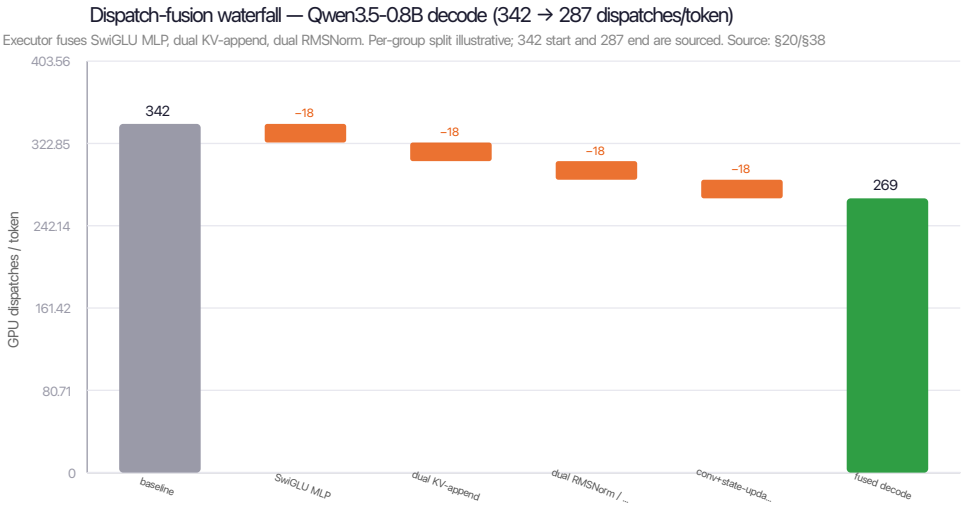

Figure 3: Dispatch-fusion waterfall for Qwen3.5-0.8B decode. Fusing the SwiGLU MLP, the dual KV-append, the residual-add+RMSNorm pair, and the conv state update removes ~ 73 dispatches per token ($\sim 342 \rightarrow \sim 269$); on mobile each removed dispatch also removes a command-buffer submit and drain. Source: §3, `docs/gpu-engine/paper.md` §20/§38; `paper/results/measurements.csv`.

Table 1: Lever B A/B, keep 131072, byte-identical greedy output. Source: §5.

Platform	Before	After	Δ
iPhone (iOS 18.7, WebKit), 1.1.0	21.1	24.5	+16%
iPhone WebKit (short A/B), 1.1.1	32.3	35.7	+10.5%
Node/Dawn (M4 Max), 1.1.1	133.8	144.8	+8%

Three engines, one kernel family. The first decoder is a NanoCodec [NVIDIA, 2025]: finite scalar quantization [Mentzer et al., 2024] (4 groups × 4 dims, mixed-radix dequant) feeding a causal HiFi-GAN [Kong et al., 2020] vocoder (a causal conv, five upsample stages, snake activations [Ziyin et al., 2020]), implemented as three new kernels (`FSQDequant`, `HalfSnake1d`, `ConvTranspose1dDepthwise`). The second is the IBM DAC decoder [Kumar et al., 2023] for an OuteTTS [OuteAI, 2025] backbone (Qwen3-0.6B): a residual vector-quantizer dequant plus a `ConvTranspose1d`+snake stack that *reuses* the same NanoCodec kernel family. The third, and the first encoder-decoder model in the engine, is Parler-TTS [Lyth et al., 2024]: a Flan-T5 [Raffel et al., 2020] text encoder whose hidden states are cross-attended into a delay-pattern audio decoder that emits DAC codec tokens, so a natural-language description ("a calm, low-pitched voice") conditions the synthesized voice. All three reach the GPU through the engine's own forward pass, like every other model, and share the convolutional codec kernel family.

Parler: encoder-decoder, cross-attention, and a second tokenizer family. Parler exercises three capabilities the engine had not shown before. It is the first *encoder-decoder* model: the IR generator emits a Flan-T5 encoder graph whose output feeds cross-attention layers in the decoder, on top of the decoder's own self-attention and the delay-pattern interleaving of nine DAC codebooks. It also needs a second tokenizer family: Flan-T5 uses a

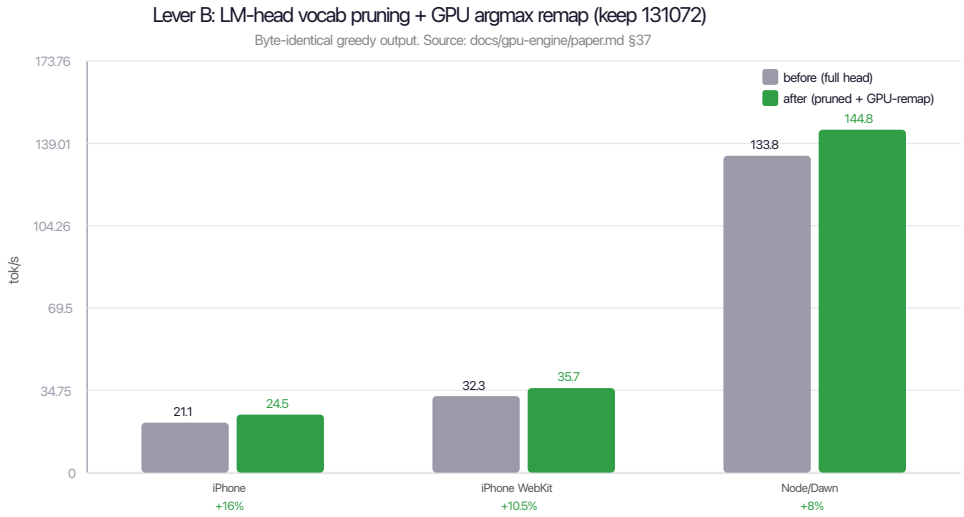

Figure 4: Lever B before/after by platform. The Dawn column is the 1.1.0 → 1.1.1 regression-to-win: the GPU argmax remap restores the pipelined decode path. Source: `paper/results/measurements.csv`.

Unigram/SentencePiece vocabulary, so we added a Viterbi Unigram tokenizer alongside the existing byte-level BPE, and it reproduces HuggingFace ids *exactly* (1110/1110). End-to-end against PyTorch `parler_tts`, the rendered waveform reaches a normalized cross-correlation of 1.000: of 648 emitted codes, 627 (96.8%) match, frames 0–66 are bit-exact across all nine codebooks, and the late-frame drift renders to silence rather than audible artifacts. Parler is the strongest evidence that the hand-written WGSL approach generalizes beyond decoder-only LMs.

OuteTTS: encode/decode symmetry for raw-audio cloning. OuteTTS clones a voice from a raw audio clip, which requires the *encode* direction of the codec as well as decode. We build the DAC `speech` encoder graph, an `argmin` RVQ-encode kernel that quantizes the encoder latent to codebook indices, and an energy-based forced alignment that maps the reference transcript onto the clip, all on-device. The encoder latent matches torch within $\max|\text{err}| = 7.9 \times 10^{-5}$ at cosine 1.0, RVQ-encode codes match exactly (450/450), and a clip round-trips coherently (encode then decode). Encode and decode share the same convolutional kernel family in opposite directions, so the cloning path adds the new ops without a new runtime.

Validation. Each codec is checked against a reference (MLX for NanoCodec, torch for DAC and Parler) with a committed gate of $\max|\text{err}| < 10^{-3}$; the measured errors clear it by orders of magnitude (Figure 6). The NanoCodec decoder reaches $\max|\text{err}| \approx 4.2 \times 10^{-6}$ vs. MLX; the DAC decoder reaches 2.86×10^{-6} (decode) and 1.28×10^{-6} (latent) vs. torch; the DAC encoder latent reaches 7.9×10^{-5} with cosine 1.0, and RVQ-encode codes match the reference exactly (450/450). On the language-model side, the OuteTTS backbone reproduces the reference greedy decode *exactly* (402/402 tokens), the new Unigram/SentencePiece tokenizer matches HuggingFace ids exactly (1110/1110), and the Parler encoder-decoder path reaches a rendered-waveform normalized cross-correlation of 1.000 against PyTorch (627/648 codes, frames 0–66 bit-exact across nine codebooks). INT4 weights shrink the NanoCodec backbone (LFM2-350M [Liquid AI, 2025]) from ~ 1479 MB to ~ 492 MB (3×), and a pre-quantized

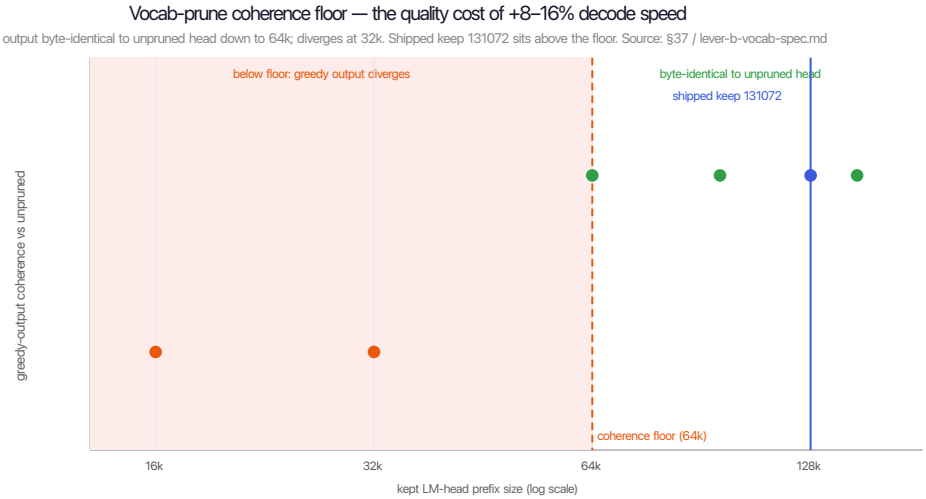

Figure 5: Vocab-prune coherence floor. Greedy output stays byte-identical to the unpruned head down to a 64k kept prefix, then diverges at 32k; the shipped keep of 131072 sits well above the floor. The quality cost of the +8–16% speed is this floor, which the device gate manages. Source: §5, `docs/gpu-engine/paper.md` §37; `paper/results/measurements.csv`.

mirror format lets the model download small. **Status (honest):** all three TTS engines ship in the engine (v1.4.x) and produce audio end-to-end. The codec *decoders* are validated bit-exact, the OuteTTS backbone greedy decode is exact (402/402), and the Parler waveform NCC is 1.000 with a small tail drift that renders to silence. The Kani-TTS (NanoCodec) backbone is the most recent to land: its autoregressive codec-LM driver (host-side, full-logit readback, per-codebook sampling) now runs end-to-end into the bit-exact NanoCodec decoder and is validated on Dawn (`scripts/engine/test-kani-speak.mjs`), so the earlier scaffolded-driver caveat no longer applies. Validation gates are deterministic and contention-independent; the one open quality item is the Parler tail-frame drift, and a clean re-run of the perf-sensitive paths is the remaining final check.

7 Reliable Mobile Deployment

A fast engine is useless if the model never loads. On a cold (uncached) load, iOS Safari stalled at "reading model header" and the tab died. The root cause was *not* in the engine: HuggingFace migrated weights to Xet storage, where a cold range request redirects to a signed URL, and that redirect-plus-range *stalls indefinitely* on iOS Safari while returning a `206` in well under a second from a server or desktop browser.

The fix. We self-host the weights on object storage (zero egress; a clean `206` with the right CORS header) and resolve the mirror at load time, falling back transparently to HuggingFace when a model is not mirrored. Three supporting changes harden the path so a flaky CDN degrades instead of hanging: a 30 s abort-timeout on range fetches (so a stall becomes a retriable error), resilient retries (six attempts, honoring `Retry-After`, with jitter, to ride out throttling within one load), and a durable, size-verified weight cache that detects and re-fetches truncated entries iOS produces under storage pressure. A related loader bug compounded this on large models: a single `fs` read or write above 2 GiB silently truncated, zeroing the weights of any model that spilled past that size with a cache directory configured, which we fixed by chunking the reads and writes into bounded file-descriptor loops. With the

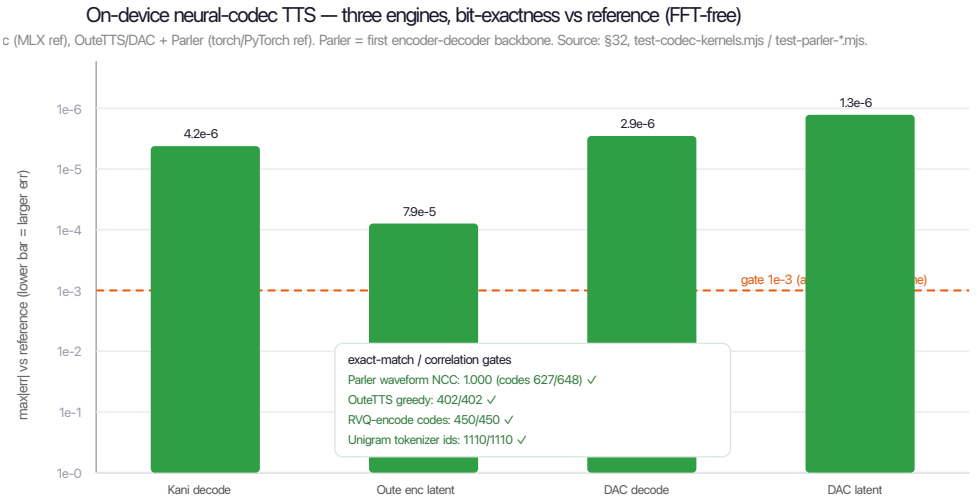

Figure 6: Validation across the three TTS engines. The codec decode/encode bars sit far above the committed 10^{-3} max |err| gate (log axis); the right-hand panel reports the exact-match and correlation gates: OuteTTS greedy 402/402, RVQ codes 450/450, Unigram tokenizer 1110/1110, and Parler waveform NCC 1.000. Source: §6, `scripts/engine/test-codec-kernels.mjs`, `test-parler-*.mjs`; `paper/results/measurements.csv`.

mirror plus this hardening, iOS loads and generates on the cold path for the first time: an iPhone downloads Qwen3.5-0.8B-4bit and runs at ∼38–40 tok/s cool, dropping toward ∼21 tok/s hot under thermal throttling (Figure 7). A quantified cold-load success-rate sweep across multiple devices is the remaining measurement (§10).

8 Evaluation

Methodology. All numbers below were recorded from existing runs; no benchmark was executed under the GPU contention present while this manuscript was prepared, and the deterministic validation gates (§6) are contention-independent. The testbed comprises an Apple M4 Max (Node+Dawn and desktop browser, Chrome 149, adapter `apple`/Metal-3, 40-core GPU, 128 GB, macOS 15.6.1 build 24G90), an iPad (iPadOS 26.5, WebKit via WKWebView over HTTPS), and an iPhone (iOS 18.7, WebKit). Throughput is greedy (temperature 0) decode tok/s; every mobile configuration produced output byte-identical to the desktop Dawn reference. Named third-party baselines (WebLLM, wllama) were measured in a *same-base-model, native-format* head-to-head on the same devices on 2026-06-25 ($n = 5$; Table 2, lower block); the desktop RAM and OS point version are as listed here, while the exact iPad/iPhone model identifiers and RAM remain flagged as [**TODO:**] in `paper/results/measurements.csv`.

Benchmark protocol. The head-to-head harness drove all three engines through one identical protocol on each device. Each engine generated 100 tokens from the same fixed prompt ("Write a detailed multi-paragraph explanation of how a CPU executes a program, covering fetch, decode, execute, registers, the ALU, cache hierarchy, and pipelining…") under greedy decoding (temperature 0), repeated for $n = 5$ runs per engine. Decode tok/s is measured uniformly by the page as generated_tokens/wall_clock_ decode_seconds; where an engine self-reports a rate it is captured only as a cross-check, and the page's own wall-clock figure

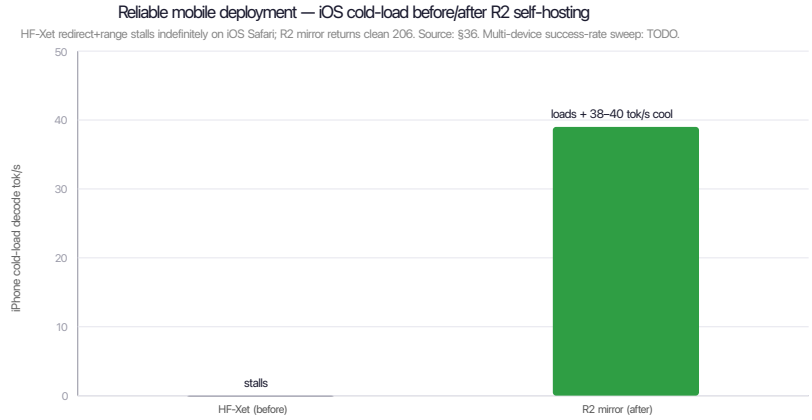

Figure 7: iOS cold-load, before/after. The upstream CDN redirect stalls indefinitely on iOS Safari; the self-hosted mirror returns a clean `206` and the device loads and decodes. Source: §7.

is the headline so the three are compared apples-to-apples. Cold-load (weight download and model construction) is timed separately and *excluded* from decode tok/s. All engines ran with their default settings; no per-engine tuning was applied. We report the per-engine median over the three runs (and, where available, the min–max range); the raw harness output is in `bench/results.jsonl` (schema `gerbil-head-to-head/v2`). Mobile figures carry an additional thermal caveat: phone throughput drifts down as the device heats within a run, so a single median understates the spread on the iPhone in particular. The on-device measurements in §4 (fusion A/Bs, group-ladder sweeps) were collected with an autonomous variant of the same harness: the phone keeps a bench page open in a daemon mode that polls a job queue over a tunnel, executes each queued configuration (model, sampling, batching group), and posts results back—so per-configuration mobile experiments run unattended and are replayed identically (back-to-back replications of the same configuration reproduced sampled throughput within $\sigma < 0.5$ tok/s).

Throughput. Table 2 summarizes decode rates. Its upper block reports Gerbil's own deployment configurations and its lower block the same-base-model head-to-head; the two blocks are measured under different conditions and should not be read against each other (see below). Figure 8 shows three *autoresearch* optimization campaigns. Autoresearch is an automated hill-climbing search over kernel and dispatch parameters: each step applies one focused engine edit, rebuilds, re-benchmarks Node/Dawn decode, and keeps or reverts the change, logging every step—kept *and* reverted—to `scripts/engine/results.jsonl`. Together the campaigns drove Node/Dawn Qwen decode from a 126.2 tok/s baseline to a **302.0 tok/s** *kept peak* (the best configuration retained by the search, not the same-session head-to-head number). The largest single kept step is the +27% state-recurrence occupancy repair of §4; the same search also produced the measured cost model reported there, and its reverted steps supply that section's ruled-out levers. Figure 9 shows the iPad submit-granularity curve that exposes the same dispatch-bound regime on mobile (round-trip-bound until $\sim$32 dispatches per command buffer, dispatch-bound beyond).

Reconciling the two desktop in-browser figures. The table reports two desktop in-browser Gerbil numbers, and they answer different questions. The $\sim$ 225–350 tok/s figure

is the *uncontended deployment/peak envelope*: the rate the site playground reaches on an otherwise-idle M4 Max, consistent with the autoresearch kept peak. The 178.8 tok/s figure is the *same-session* desktop number from the same-base-model head-to-head, recorded back-to-back with the WebLLM 84.5 tok/s run on 2026-06-25 under whatever load that session carried. The **2.1**× ratio is computed from that same-session pair (178.8/84.5), so it is internally consistent even though the absolute 178.8 sits below the uncontended peak; the two figures differ because of session, contention, and configuration, not because of any change to the engine.

Same-base-model, native-format head-to-head against WebLLM and wllama. The lower block of Table 2 is a *same-base-model, each-in-its-native-format* comparison rather than a strictly controlled experiment: all three in-browser engines run the same base model (Qwen3.5-0.8B), but each in the 4-bit format it ships and consumes natively, on the same hardware on 2026-06-25 ($n=5$, medians, greedy). The three formats are not interchangeable, and they differ on several axes that bear directly on throughput in a regime dominated by per-token fixed costs. First, *bits per weight*: Gerbil's MLX INT4 is true 4-bit (affine, group size 64), WebLLM's MLC `q4f16_1` stores INT4 weights with fp16 activations at the finer group size 32, and wllama's GGUF `Q4_K_M` is a mixed-precision k-quant averaging $\sim$4.5 bits per weight ($\sim$12% larger than true 4-bit). Second, *scale/zero metadata bandwidth*: MLC's group size 32 carries roughly twice the per-dispatch scale/zero metadata of Gerbil's group size 64, a plausible contributor to the gap that is independent of shader authorship. Third, the *KV-cache layout, op-fusion schedule, INT4 dequant path, and memory allocator* differ wholesale between a hand-written WGSL engine and a TVM-compiled one. We therefore read the throughput ratio as the *combined* effect of all of these, not as an isolated measure of kernel authorship; cleanly decomposing the format contribution from the kernel contribution would require a matched-format ablation (running both engines on byte-identical quantized weights), which we do not perform here and flag as future work (§10).

With that framing, on the same base model the Gerbil stack is **2.1**× faster than WebLLM's TVM-compiled stack on the M4 Max desktop (178.8 vs. 84.5 tok/s, same session) and **1.5**× faster on the iPad (46.3 vs. 30.1 tok/s); an earlier impression that the two were comparable was an artifact of WebLLM defaulting to a smaller 0.5B model, which running both on the 0.8B base removes. The WASM/CPU path (wllama) *cannot load* the 0.8B model on any device: its $\sim$1.16 GB allocation exceeds the typed-array/WASM memory ceiling ("Invalid typed array length" on Chrome, "Length out of range of buffer" on WebKit). This is partly a format-size effect, not a pure engine limit: wllama's $\sim$4.5-bit `Q4_K_M` footprint is $\sim$12% larger than a true 4-bit copy of the same weights, so its larger byte size contributes to breaching the 2 GiB WASM typed-array ceiling; a tighter-quantized GGUF might fit, but no true-4-bit GGUF of this model is shipped, so at the as-shipped size CPU/WASM is not a viable in-browser substrate. The result on iPhone is about *reliability*, not speed: WebLLM **OOM-crashes the tab**, reproduced 3/3, when its working set collides with the iOS WebKit heap cap (the process dies before it can report a number), while Gerbil decodes at 25.8 tok/s. Gerbil is the only one of the three engines that runs the 0.8B model across desktop, iPad, *and* iPhone, which reinforces the mobile-deployment thesis of §7.

[2]Protocol (identical across engines and devices): same fixed prompt, 100 generated tokens, greedy (temperature 0), $n = 5$ runs/engine, default settings, no per-engine tuning; decode tok/s = generated tokens / wall-clock decode seconds (page measured); cold load timed separately and excluded. Per-engine native 4-bit format on the same base model, with approximate bits per weight (bpw) and quant group size: Gerbil `mlx-community/Qwen3.5-0.8B-4bit` (MLX INT4, true 4-bit, affine, group 64); WebLLM `Qwen3.5-0.8B-q4f16_1-MLC` (MLC `q4f16_1`: INT4 weights / fp16 activations, group 32, whose finer grouping roughly doubles per-dispatch scale/zero metadata bandwidth); wllama `bartowski Qwen_Qwen3.5-0.8B-GGUF Q4_K_M` (WASM/CPU, llama.cpp k-quant, mixed-precision $\sim$4.5 bpw, $\sim$12% larger than true 4-bit). The

Table 2: Decode throughput by device and runtime, Qwen3.5-0.8B (greedy, temperature 0, 100 generated tokens, default settings; per-row sample counts noted inline, head-to-head $n = 5$). The upper block reports Gerbil's own deployment configurations (measured under different sessions/conditions and not directly comparable to the lower block); the lower block is a *same-base-model, native-format* head-to-head (all three engines run Qwen3.5-0.8B, each in the 4-bit format it ships natively) measured 2026-06-25 on Chrome 149 / M4 Max (medians; min–max where available).[2] Source: `measurements.csv` (`head_to_head_controlled`), `bench/results.jsonl`.

Device	Runtime / engine	tok/s
Gerbil deployment configurations		
M4 Max	Gerbil, Node/Dawn (post-fix baseline)	145 (143.9–147.1, n=3)
M4 Max	Gerbil, Node/Dawn (autoresearch peak)	302.0 (300.5–302.9, n=6)
M4 Max	Gerbil, in-browser	225–350
iPad	Gerbil, WebKit (sustained)	35.9
iPhone	Gerbil, WebKit (cool / hot)	38–40 / 21
iPad	transformers.js (ONNX-web)	6.9–11.6
Same-base-model native-format head-to-head (Qwen3.5-0.8B, n=5)		
M4 Max (Chrome)	Gerbil (WGSL, MLX INT4)	**178.8** (176.7–181.0)
M4 Max (Chrome)	WebLLM (MLC/TVM, q4f16_1)	84.5
M4 Max (Chrome)	wllama (WASM/CPU)	✗ load fail
iPad (WebKit)	Gerbil (WGSL, MLX INT4)	**46.3** (45.0–46.8)
iPad (WebKit)	WebLLM (MLC/TVM, q4f16_1)	30.1
iPad (WebKit)	wllama (WASM/CPU)	✗ load fail
iPhone (WebKit)	Gerbil (WGSL, MLX INT4)	**25.8** (24.7–29.2)
iPhone (WebKit)	WebLLM (MLC/TVM, q4f16_1)	✗ OOM crash
iPhone (WebKit)	wllama (WASM/CPU)	✗ load fail

Per-contribution results. Lever B (§5, Table 1) gives +10–16% WebKit / +8% Dawn, byte-identical. The three TTS engines (§6, Figure 6) are bit-exact within 10^{-3} by orders of magnitude (Parler waveform NCC 1.000, OuteTTS greedy 402/402, RVQ codes 450/450, Unigram tokenizer 1110/1110), with a 3× INT4 size reduction. The deployment method (§7) converts a hard cold-load failure on iOS into a working ∼38–40 tok/s session.

9 Related Work

In-browser WebGPU LLMs. WebLLM [Ruan et al., 2024] over MLC/TVM [MLC team, 2023, Chen et al., 2018] is the closest peer; its kernels are compiler-generated, ours hand-written WGSL. transformers.js over ONNX Runtime Web [Hugging Face, 2023, ONNX Runtime authors, 2024] is the stack we replaced (and measured against). ratchet [Hugging Face, 2024] is the nearest architectural sibling (WebGPU-native) but Rust/WASM-driven; wllama [Nguyen, 2024] and picoLLM [Picovoice, 2024] are WASM/closed contrasts; Ten-

desktop Gerbil 178.8 (min–max 176.7–181.0, n=5) and WebLLM 84.5 (median of n=5; its first decode run, 31.1, is a cold/warmup outlier and the other four are 82–86) figures are from the *same* back-to-back session, so the 2.1× ratio is same-session and internally consistent. wllama ✗ *load fail*: the ∼1.16 GB model (inflated by the ∼4.5-bit `Q4_K_M` footprint) exceeds the typed-array/WASM memory ceiling and never loads ("Invalid typed array length" on Chrome, "Length out of range of buffer" on WebKit), partly a format-size effect, not a pure engine limit. WebLLM ✗ *OOM crash*: the iOS WebKit tab process died under the heap cap before reporting a number, reproduced 3/3. Gerbil is the only one of the three engines that runs the 0.8B model across desktop, iPad, and iPhone.

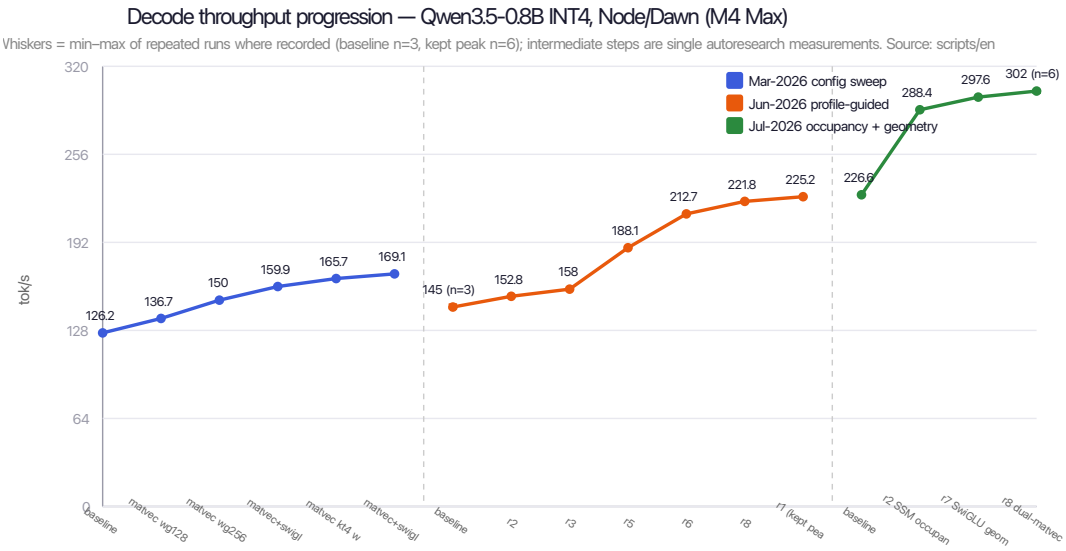

Figure 8: Autoresearch campaigns driving Node/Dawn Qwen decode (kept-improvement steps). Whiskers show the min–max of repeated runs where recorded (June baseline n=3, July kept peak n=6); the intermediate steps are single autoresearch measurements. Source: `scripts/engine/results.jsonl`.

sorFlow.js [Smilkov et al., 2019] is the historical web-inference anchor.

On-device/edge engines. llama.cpp [Gerganov and contributors, 2023] is the dominant local reference; PowerInfer [Song et al., 2024] and its mobile follow-up [Xue et al., 2024] motivate heterogeneous dispatch and the I/O-vs-compute framing; MNN [Jiang et al., 2020] is the pre-LLM mobile lineage; MLX [Apple Machine Learning Research, 2023] supplies our reference checkpoints and the Apple-silicon unified-memory context.

Speculative decoding and quantization. Our dispatch-bound framing makes speculative decoding [Leviathan et al., 2023, Chen et al., 2023, Cai et al., 2024, Li et al., 2024, Miao et al., 2024] a structural fit, discussed as future work. INT4 weights connect to the W4A16 line [Frantar et al., 2023, Lin et al., 2024, Dettmers et al., 2022].

Vocabulary reduction. The thin prior art is itself part of the contribution: static prefix pruning of a decoder-only LM head with an in-kernel argmax remap has no direct precedent. The nearest neighbors, dynamic NMT vocabulary selection [Mi et al., 2016, L'Hostis et al., 2016, Domhan et al., 2022] and softmax acceleration [Grave et al., 2017], either rebuild per input or keep every token reachable; vocabulary scaling laws [Tao et al., 2024] argue the opposite direction, which our device gate reconciles.

Neural-codec TTS. Our codecs descend from the conv-codec line [Zeghidour et al., 2021, Défossez et al., 2022, Kumar et al., 2023] with FSQ quantization [Mentzer et al., 2024], HiFi-GAN vocoding [Kong et al., 2020], and snake activations [Ziyin et al., 2020], in the codec-LM paradigm of VALL-E [Wang et al., 2023], with on-device backbones from LFM2 [Liquid AI, 2025] and the OuteTTS [OuteAI, 2025]/Kani-TTS [nineninesix, 2025] systems; Parler-TTS [Lyth et al., 2024] adds a text-prompted, Flan-T5 [Raffel et al., 2020] encoder-decoder

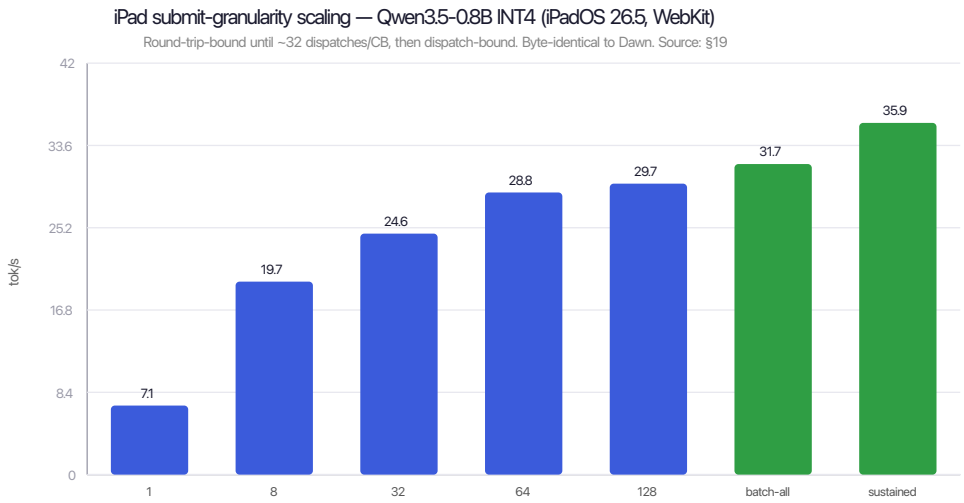

Figure 9: iPad submit-granularity scaling. Throughput is round-trip-bound until ~ 32 dispatches per command buffer, then dispatch-bound, the mobile face of §4. Source: §8.

design for voice control. The STT side reuses a raw-waveform front end [Useful Sensors, 2024].

10 Limitations and Future Work

The evaluation is single-author and single-device-class per form factor; throughput is thermally sensitive on phones and varies with browser version. The WebLLM / wllama head-to-head (§8, Table 2) carries its own caveats. It is a *same-base-model, native-format* comparison, not a strictly controlled experiment: the engines differ at once on quantization format and group size, KV-cache layout, op fusion, INT4 dequant path, and memory allocator, so the throughput ratio measures the *combined* effect of the whole stack, not kernel authorship in isolation. The single ablation that would decompose the dominant format-versus-kernel contribution, running both engines on byte-identical quantized weights (a matched-format ablation), is not performed here and is the most important missing measurement; without it we do not claim the kernel authorship alone accounts for the gap. The run is $n = 5$ per engine, with medians and Gerbil min–max ranges reported (WebLLM by median, as its first decode run is a cold/warmup outlier), on one device class per form factor, measured 2026-06-25; the WebLLM iPhone OOM crash was observed and reproduced 3/3 by hand but could *not* be auto-logged, because the WebKit tab process dies before the harness can write a result. We also do not report WebLLM's per-engine dispatch count or each engine's on-device peak-memory and model-size in bytes, which would further localize the gap; these are deferred to the matched-format ablation. Several clean sweeps remain before submission, each flagged as [**TODO:**] in the artifact: (a) the matched-format WebLLM-vs-Gerbil ablation above (byte-identical weights, plus per-engine dispatch counts and resident-memory measurements), at $n \geq 5$; (b) a controlled vocab-prune sweep with $N \geq 5$ runs and perplexity/MMLU-Pro/GSM8K quality metrics; (c) a quantified iOS cold-load success-rate and load-time sweep across multiple iPhones and iPads; (d) a re-measured dispatch breakdown with variance bars; (e) a broader objective and subjective TTS quality sweep across the three now-shipping engines (all codec decoders/encoder and the OuteTTS, Parler, and Kani-TTS backbones validated, with the Parler tail-frame drift the one open quality item); and (f) as a direction we are exploring—not an implemented, evaluated, or claimed contribution—

applying the LoRA correction at decode over a *static* INT4 base instead of merging it into the float base before quantization, which would let a flavor swap without re-quantizing. This runtime-on-INT4 path is not yet implemented, benchmarked, or shown to improve on the shipped merge-at-load flavors or on existing multi-adapter serving of quantized bases; the merge-at-load path (validated bit-exact to the base under a zero adapter) is the only adapter mechanism this paper claims. A precise runbook for (a), (b), and (c)—including how to construct the byte-identical matched-format weights and which per-engine dispatch/memory quantities to capture—is in `docs/paper-ablation-plan.md`. Speculative decoding (§4) is the next throughput lever and is deferred behind the mobile memory/submit work. Absolute perf numbers in this manuscript were recorded *before* the current GPU-contention window and should be re-confirmed once the device frees up; the bit-exactness and exact-match gates are contention-independent and stand.

11 Conclusion

Gerbil shows that a WebGPU inference engine written directly in WGSL, with no ONNX runtime and no deep-learning compiler, is a practical foundation for private, key-free, on-device language and speech models that runs the same compute in the browser and under Node. The decisive systems insight is that on-device decode is dispatch-bound, which reframes optimization around fusion and speculative decoding; the vocab-prune GPU-remap result, three bit-exact FFT-free TTS engines (including the first encoder-decoder, text-prompted Parler backbone), and the iOS deployment method each follow from owning the whole stack. A same-base-model, native-format head-to-head makes the case concrete: on one base model, the Gerbil stack decodes 2.1× faster than the compiler-generated peer on desktop and 1.5× on iPad, the WASM/CPU engine cannot load the model at all, and the compiler-generated peer OOM-crashes the iPhone, leaving Gerbil the only one of the three that runs the 0.8B across desktop, iPad, and iPhone. That throughput ratio reflects the combined effect of kernel authorship, the INT4 format, KV-cache layout, fusion, and memory management together; isolating the kernel-versus-format contribution would require a matched-format ablation we leave to future work, and the reliability result (running where the peers crash or fail to load) stands independent of it. The engine clears the recognizable in-browser-LLM bar while reporting the rigor the genre usually skips: a dispatch characterization, a regression-to-win ablation, a same-base-model baseline sweep, and bit-exactness gates.

Artifact

The engineering reference is `docs/gpu-engine/paper.md`; consolidated, provenance-tagged measurements are in `paper/results/` (regenerated by `paper/results/consolidate.mjs`); figures are generated by `paper/figures/make-figures.mjs` from `paper/results/figures.json`. Every number in the paper maps to a row in `paper/results/measurements.csv` with a source; [**TODO:**] rows mark numbers awaiting a clean sweep.

References

Apple Machine Learning Research. MLX: An array framework for apple silicon. https://github.com/ml-explore/mlx, 2023.

Tianle Cai, Yuhong Li, Zhengyang Geng, et al. Medusa: Simple LLM inference acceleration framework with multiple decoding heads. In *International Conference on Machine Learning (ICML)*, 2024.

Charlie Chen, Sebastian Borgeaud, Geoffrey Irving, et al. Accelerating large language model decoding with speculative sampling. *arXiv preprint arXiv:2302.01318*, 2023.

Tianqi Chen, Thierry Moreau, Ziheng Jiang, et al. TVM: An automated end-to-end optimizing compiler for deep learning. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2018.

Alexandre Défossez, Jade Copet, Gabriel Synnaeve, and Yossi Adi. High fidelity neural audio compression. *Transactions on Machine Learning Research (TMLR); arXiv:2210.13438*, 2022.

Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. LLM.int8(): 8-bit matrix multiplication for transformers at scale. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2022.

Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Tobias Domhan, Felix Hieber, Stephan Schamoni, et al. The devil is in the details: On the pitfalls of vocabulary selection in neural machine translation. In *NAACL*, 2022.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. In *International Conference on Learning Representations (ICLR)*, 2023.

Georgi Gerganov and contributors. llama.cpp. https://github.com/ggml-org/llama.cpp, 2023.

Edouard Grave, Armand Joulin, Moustapha Cissé, et al. Efficient softmax approximation for GPUs. In *International Conference on Machine Learning (ICML)*, 2017.

Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. LoRA: Low-rank adaptation of large language models. In *International Conference on Learning Representations (ICLR)*, 2022.

Hugging Face. Transformers.js: State-of-the-art machine learning for the web. https://github.com/huggingface/transformers.js, 2023.

Hugging Face. ratchet: A cross-platform browser ML framework. https://github.com/huggingface/ratchet, 2024.

Xiaotang Jiang, Huan Wang, Yiliu Chen, et al. MNN: A universal and efficient inference engine. *Proceedings of Machine Learning and Systems (MLSys)*, 2020.

Jungil Kong, Jaehyeon Kim, and Jaekyoung Bae. HiFi-GAN: Generative adversarial networks for efficient and high fidelity speech synthesis. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

Rithesh Kumar, Prem Seetharaman, Alejandro Luebs, et al. High-fidelity audio compression with improved RVQGAN. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2023.

Yaniv Leviathan, Matan Kalman, and Yossi Matias. Fast inference from transformers via speculative decoding. In *International Conference on Machine Learning (ICML)*, 2023.

Gurvan L'Hostis, David Grangier, and Michael Auli. Vocabulary selection strategies for neural machine translation. *arXiv preprint arXiv:1610.00072*, 2016.

Yuhui Li, Fangyun Wei, Chao Zhang, and Hongyang Zhang. EAGLE: Speculative sampling requires rethinking feature uncertainty. In *International Conference on Machine Learning (ICML)*, 2024.

Ji Lin, Jiaming Tang, Haotian Tang, et al. AWQ: Activation-aware weight quantization for LLM compression and acceleration. In *Proceedings of Machine Learning and Systems (MLSys)*, 2024.

Liquid AI. LFM2 technical report. *arXiv preprint arXiv:2511.23404*, 2025.

Dan Lyth, Simon King, and Hugging Face. Parler-TTS: A lightweight, high-quality text-to-speech model with natural-language voice prompts. https://github.com/huggingface/parler-tts, 2024.

Fabian Mentzer, David Minnen, Eirikur Agustsson, and Michael Tschannen. Finite scalar quantization: VQ-VAE made simple. In *International Conference on Learning Representations (ICLR)*, 2024.

Haitao Mi, Zhiguo Wang, and Abe Ittycheriah. Vocabulary manipulation for neural machine translation. In *Annual Meeting of the Association for Computational Linguistics (ACL)*, 2016.

Xupeng Miao, Gabriele Oliaro, Zhihao Zhang, et al. SpecInfer: Accelerating generative LLM serving with tree-based speculative inference and verification. In *ASPLOS*, 2024.

MLC team. MLC-LLM: Universal LLM deployment engine with machine learning compilation. https://github.com/mlc-ai/mlc-llm, 2023.

Xuan-Son Nguyen. wllama: Webassembly binding for llama.cpp. https://github.com/ngxson/wllama, 2024.

nineninesix. Kani-TTS. https://huggingface.co/nineninesix, 2025.

NVIDIA. NanoCodec: A low-frame-rate speech codec. In *Interspeech; arXiv:2508.05835*, 2025.

ONNX Runtime authors. ONNX Runtime Web. https://onnxruntime.ai, 2024.

OuteAI. OuteTTS: Interface for OuteAI text-to-speech models. https://github.com/edwko/OuteTTS, 2025.

Picovoice. picoLLM: On-device, cross-platform LLM inference. https://github.com/Picovoice/picollm, 2024.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020.

Charlie Ruan, Yucheng Qin, Xun Zhou, Ruihang Lai, Sicheng Feng, Yong Wang, et al. WebLLM: A high-performance in-browser LLM inference engine. *arXiv preprint arXiv:2412.15803*, 2024.

Daniel Smilkov, Nikhil Thorat, Yannick Assogba, et al. TensorFlow.js: Machine learning for the web and beyond. *Proceedings of Machine Learning and Systems (MLSys)*, 2019.

Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. PowerInfer: Fast large language model serving with a consumer-grade GPU. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2024.

Chaofan Tao, Qian Liu, Longxu Dou, et al. Scaling laws with vocabulary: Larger models deserve larger vocabularies. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2024.

Useful Sensors. Moonshine: Speech recognition for live transcription and voice commands. https://github.com/usefulsensors/moonshine, 2024.

W3C. WebGPU. https://www.w3.org/TR/webgpu/, 2024.

Chengyi Wang, Sanyuan Chen, Yu Wu, et al. Neural codec language models are zero-shot text to speech synthesizers. *arXiv preprint arXiv:2301.02111*, 2023.

Zhenliang Xue, Yixin Song, Zeyu Mi, et al. PowerInfer-2: Fast large language model inference on a smartphone. *arXiv preprint arXiv:2406.06282*, 2024.

Neil Zeghidour, Alejandro Luebs, Ahmed Omran, et al. SoundStream: An end-to-end neural audio codec. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 2021.

Liu Ziyin, Tilman Hartwig, and Masahito Ueda. Neural networks fail to learn periodic functions and how to fix it. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.