GEARBX: Entropy-Routed Dynamic Quantization for LLM Inference

Ratnam Shah
Independent Researcher
ratnam@jpdz.app

May 28, 2026

Abstract

Not all tokens are equally hard to generate. Common tokens such as "the" or "of" are produced from sharply peaked output distributions, while mathematical reasoning, rare vocabulary, and creative transitions produce broader distributions. Static quantization ignores this variance and applies one bit-width to every token.

GEARBX treats quantization precision as a per-token decision. At each decoding step it reads the Shannon entropy of the output-logit distribution and routes the next forward pass through one of three precision tiers: 4-bit packed weights for confident tokens, 8-bit weights for moderate-uncertainty tokens, and fp16 weights for difficult tokens. When shifting down, the system physically replaces `nn.Linear` modules with packed `QuantizedLinear` buffers and offloads the original fp16 weights to CPU, producing real device-memory reduction. Because autoregressive decoding is memory-bandwidth-bound, fewer bytes per weight can translate directly to faster tokens when fused packed-weight kernels are used.

Three design choices are new in combination: output-logit entropy as the sole routing signal, physical module replacement with memory offloading at inference time, and gear-oscillation suppression through rolling-window averaging, hysteresis, and minimum gear duration. The current implementation targets Apple Silicon via MPS and MLX, plus CPU execution on ARM NEON, focusing on consumer hardware where memory capacity and bandwidth are binding constraints.

1 Introduction and Motivation

1.1 The Bandwidth Bottleneck

Autoregressive LLM inference is typically memory-bandwidth-bound during decode, consistent with roofline analysis and memory-aware LLM serving work [2, 8]. Each generated token requires loading the relevant weight matrices for a single matrix-vector multiply. For a 7B-parameter model stored in fp16, the full model weighs roughly 14 GB. The arithmetic intensity of a linear layer is approximately

$$\text{Arithmetic Intensity} = \frac{2OI}{(OIb_w) + (Ib_x) + (Ob_y)} \approx \frac{2}{b_w},$$

where b_w is bytes per weight element. At fp16, $b_w = 2$, giving approximately one FLOP per byte transferred. Modern accelerators can deliver far more compute than the memory bus can feed at this intensity, so reducing bytes per weight is directly valuable.

1.2 Static Versus Dynamic Quantization

Static methods such as GPTQ, AWQ, bitsandbytes NF4, and GGML/GGUF-style formats choose one precision level for all tokens [4–6, 14]. This forces a global quality–speed tradeoff: 4-bit quantization accelerates easy tokens but may degrade hard reasoning steps, while 8-bit or fp16 preserves quality but wastes bandwidth on predictable filler.

GEARBX instead optimizes for the marginal token. It spends high precision on high-uncertainty steps and low precision on predictable steps.

1.3 Key Insight

The entropy of the next-token distribution varies dramatically during generation. For a 32K-token vocabulary, the following practical regimes are useful:

Regime	Entropy (bits)	Token Character	Example
Low	< 1.8	Highly predictable filler	articles, punctuation
Mid	1.8–3.5	Moderate uncertainty	content words, common phrases
High	> 3.5	Genuine deliberation	math symbols, rare words

Table 1: Default entropy regimes for a 32K-token vocabulary. Thresholds scale with vocabulary size.

These thresholds are calibrated for a 32K-token vocabulary and scale with vocabulary size. The observation that entropy varies by token is not itself new; the distinguishing system choice is to act on that signal by physically swapping quantized weight modules at inference time.

1.4 Transmission Analogy

The name reflects a manual-transmission metaphor: engine RPM corresponds to output entropy; low gear corresponds to 4-bit weights; high gear corresponds to fp16 weights; and a gear shift corresponds to a weight swap. The objective is to be in the minimum precision tier that preserves quality for the current generation regime.

1.5 Contributions

1. A training-free entropy-routed precision scheduler using final output-logit Shannon entropy.
2. Physical weight swapping and CPU offloading that reduce active device memory.

3. Shift suppression via rolling averages, hysteresis, and minimum gear duration.
4. Backend implementations for PyTorch/MPS, MLX, Ollama-style servers, and CPU ARM NEON.

1.6 Scope

This document covers the Apple Silicon MPS and MLX implementations plus CPU execution on ARM NEON. The architecture can support CUDA with fused Triton kernels, but CUDA validation and performance characterization are outside this reference.

2 Related Work

2.1 Static Quantization

GPTQ, AWQ, bitsandbytes NF4, and related formats compress weights to fixed low-precision representations [4–6]. GEARBX uses conventional packed integer storage and per-row symmetric scales, but selects precision per token rather than globally.

2.2 Token-Adaptive Precision

MoBiQuant uses a learned hidden-state router and residual bit slices. QuickSilver combines token halting, KV-cache skipping, token fusion, and adaptive Matryoshka quantization with mid-layer entropy. FlexQuant uses perplexity entropy and KL divergence for per-token, per-layer decisions. GEARBX differs by using only final output entropy and by physically swapping modules instead of dispatching among already resident variants.

2.3 Other Adaptive Methods

Progressive Mixed-Precision Decoding lowers precision as generation proceeds, but cannot respond to late difficulty spikes. EdgeQAT and EWQ use entropy signals during training or calibration rather than per-token inference. QTALE and adaptive KV-cache quantization operate on layers or cache tensors rather than model weights, and are complementary to GEARBX.

System	Routing Signal	Granularity	Representation	Swap?	Training-Free?
GPTQ/AWQ/NF4	None	Global	Packed integers	N/A	Yes
MoBiQuant	Learned hidden-state router	Per token	Residual bit slices	No	No
QuickSilver	Mid-layer entropy	Per token	Matryoshka	No	Partly
FlexQuant	Entropy + KL	Token/layer	Pre-quantized dispatch	No	Yes
PMPD	Position	Phase	Multiple variants	Loads variants	Yes
EdgeQAT	Attention Q/K entropy	Training-time token	QAT	No	No
QTALE	Token/layer policy	Per token/layer	Layer skipping + quantization	No	Partly
KV-cache methods	Cache entropy	Per-token cache	KV precision tiers	No	Yes
GEARBX	Output Shannon entropy	Per token	int4/int8/fp16	Yes	Yes

Table 2: Design comparison of adaptive precision systems.

3 Entropy Monitoring

3.1 Shannon Entropy

Following Shannon entropy [1], at generation step t, with logits $\mathbf{z}_t \in \mathbb{R}^V$,

$$H_t = -\sum_{i=1}^{V} p_i \log_2 p_i = -\frac{1}{\ln 2} \sum_i p_i \ln p_i, \qquad p_i = \text{softmax}(\mathbf{z}_t)_i.$$

The implementation uses

$$\boldsymbol{\ell} = \text{log\_softmax}(\mathbf{z}_t), \qquad \mathbf{p} = \exp(\boldsymbol{\ell}), \qquad H_t = -\frac{\mathbf{p} \cdot \boldsymbol{\ell}}{\ln 2}.$$

The theoretical bounds are $0 \le H_t \le \log_2 V$.

Vocabulary	$H_{\max}$	Typical Range
32,000–32,768	15.0 bits	0.3–5.5
128,000	17.0 bits	0.5–6.5
151,936	17.2 bits	0.5–6.5

Table 3: Theoretical and practical entropy ranges.

3.2 Rolling Window Smoothing

Raw token entropy is noisy, so the monitor uses

$$\bar{H}_t = \frac{1}{|W_t|} \sum_{i \in W_t} H_i, \qquad W_t = \{H_{t-w+1}, \ldots, H_t\}.$$

The PyTorch backend defaults to $w = 5$; MLX defaults to $w = 3$ because it avoids the same physical swap cost.

3.3 Gear Classification and Hysteresis

Without hysteresis,

$$\text{gear}(\bar{H}_t) = \begin{cases} \text{low}, & \bar{H}_t \le \theta_{\text{low}}, \\ \text{high}, & \bar{H}_t \ge \theta_{\text{high}}, \\ \text{mid}, & \text{otherwise.} \end{cases}$$

Defaults are $\theta_{\text{low}} = 1.8$ and $\theta_{\text{high}} = 3.5$ bits for a 32K reference vocabulary. Hysteresis requires leaving an extreme gear only after crossing the relevant threshold by margin h; the default GearbxModel margin is $h = 0.1$ bits.

3.4 Vocabulary Scaling

Thresholds scale by

$$\text{scale} = \frac{\log_2 V}{\log_2 V_{\text{ref}}}, \qquad V_{\text{ref}} = 32768.$$

Then $\theta_{\text{low}}^{(V)} = \theta_{\text{low}}^{(\text{ref})}\text{scale}$ and $\theta_{\text{high}}^{(V)} = \theta_{\text{high}}^{(\text{ref})}\text{scale}$.

3.5 Predictive Pre-Warming and Safety

The monitor estimates entropy slope

$$\nabla H = \frac{H_{\text{last}} - H_{\text{first}}}{|W| - 1}, \qquad \hat{H}_{t+L} = \bar{H}_t + L\nabla H,$$

with default $L = 2$, enabling pre-computation of likely next-gear modules. Catastrophic entropy is detected when the last two entropy values exceed $0.9 \log_2 V$, triggering fallback to high gear.

3.6 Deferred Entropy Pipeline

Entropy is computed as a device tensor and committed only after the natural synchronization caused by reading the sampled token ID. This removes one GPU-to-CPU synchronization per token relative to a naive implementation.

4 Automatic Threshold Calibration

4.1 Percentile Calibration

For sorted positive entropy samples $\{e_1, \ldots, e_n\}$ with $n \ge 5$,

$$\theta_{\text{low}} = \max(e_{\lfloor np_{\text{low}} \rfloor - 1}, 0.01), \qquad \theta_{\text{high}} = e_{\min(n-1, \lfloor np_{\text{high}} \rfloor)}.$$

Defaults are $p_{\text{low}} = 0.30$ and $p_{\text{high}} = 0.60$. If the band is narrower than 0.2 bits, thresholds are expanded around their center.

4.2 Backend Strategies

PyTorch calibrates from prefill logits, then caps thresholds by $0.06 \log_2 V$ for low and $0.12 \log_2 V$ for high. MLX runs three short calibration prompts and sets low to the entropy median and high to 2.5 times the median, with a minimum band of 0.3 bits. A manual utility can use the 80th percentile of trivial prompts and the 20th percentile of hard prompts to define the easy/hard boundary.

5 Quantization Formats

5.1 Per-Row Symmetric Quantization

Following standard post-training quantization practice [3, 4, 6], for weight matrix $\mathbf{W} \in \mathbb{R}^{O \times I}$,

$$s_i = \frac{\max_j |W_{ij}|}{q_{\max}}, \qquad s_i = \max(s_i, s_{\text{floor}}),$$

$$\hat{W}_{ij} = \text{clamp}\left(\text{round}\left(\frac{W_{ij}}{s_i}\right), -q_{\max}, q_{\max}\right).$$

The scale floor is 10^{-4} for fp16 and 10^{-8} for fp32.

Bit-width	$q_{\max}$	Signed Range	Storage
8	127	$[-127, 127]$	`int8`
6	31	$[-31, 31]$	packed `uint8`
4	7	$[-7, 7]$	packed `uint8`
2	1	$\{-1, 0, 1\}$	packed `uint8`

Table 4: Supported quantization levels. Production gears use 4-bit, 8-bit, and fp16.

5.2 Packing Layouts

Int8 stores one signed value per byte. Int4 stores two signed 4-bit values per byte using offset +8 and low/high nibbles. Int6 stores four 6-bit values in three bytes using offset +32. Int2 stores four ternary values per byte using offset +2.

Dequantization reconstructs $\tilde{W}_{ij} = \hat{W}_{ij} s_i$, with element error bounded by $s_i/2$.

Format	Bytes/param	Ratio vs fp16	Savings
fp16	2.000	1.000	0%
int8	≈ 1.000	≈ 0.500	$\approx 50\%$
int6	≈ 0.750	≈ 0.375	$\approx 62.5\%$
int4	≈ 0.500	≈ 0.250	$\approx 75\%$
int2	≈ 0.250	≈ 0.125	$\approx 87.5\%$

Table 5: Packed-weight memory summary. Scale overhead is negligible for large matrices.

6 Precision Management

This is the core difference between GEARBX and dispatch-based adaptive systems. Rather than keeping several resident model variants and selecting among them, the precision manager physically replaces modules in the live model tree and offloads inactive originals. Device memory therefore changes when the gear changes. The tradeoff is that shifting has real cost, so the entropy monitor is designed to make shifts infrequent and sustained.

6.1 Module Discovery

GEARBX targets attention projections because they are bandwidth-intensive during autoregressive decode and are a natural first point for dynamic precision. Discovery is model-agnostic:

1. Walk the module tree through `named_modules()`.
2. Select `nn.Linear` modules whose dot-separated path contains attention components such as `self_attn`, `attention`, `attn`, or `selfattention` after lowercasing and underscore removal.
3. Collect the prefix up to the attention container, for example `model.layers.0.self_attn`.
4. Sort prefixes by layer index to make swapping deterministic.

This naming strategy covers common Llama, Mistral, Phi, Qwen, Gemma, OPT, GPT-NeoX, Bloom, Falcon, StableLM, MPT, and T5 layouts without model-specific configuration.

6.2 Lazy Quantization

No weights are quantized at model load time. The first quantization happens only when `shift()` enters a non-high gear, and only for that requested gear. This keeps model loading fast, avoids paying for gears that are never used, and ensures at most one active quantized representation needs to live in the device pool.

Listing 1: Precision-manager shift sketch.

```
shift(target_gear):
  if target_gear == current_gear: return
```

```
if target_gear == "high":
  move originals to device
  swap originals into model tree
  cache previous quantized gear
else:
  if target_gear in gear_cache:
    restore cached quantized modules
  else:
    for each managed module:
      quantize(original, bits_for(target_gear))
      install quantized module in cache
  swap quantized modules into model tree
  move originals to CPU

current_gear = target_gear
```

6.3 Module Swapping and Offloading

A gear shift is implemented as a Python module-reference replacement at the appropriate parent object. The operation walks a short attribute path, then calls `setattr`; tensor data are not copied by the pointer swap itself. When active gear is low or mid, the model tree contains packed quantized buffers and scales, while original fp16 weights live in CPU memory. On Apple Silicon this offload is logical rather than a discrete DMA transfer because CPU and GPU share physical DRAM, but it still reduces pressure on the MPS device allocator.

Active Gear	Device Memory	CPU RAM / Host Pool
high	fp16 originals	–
mid	int8 packed weights + scales	fp16 originals
low	int4 packed weights + scales	fp16 originals

Table 6: Memory placement by active gear.

For a managed 2048×2048 layer, high gear uses roughly 8 MB on device, mid gear uses roughly 4.2 MB on device plus an fp16 host copy, and low gear uses roughly 2.1 MB on device plus an fp16 host copy.

6.4 Gear Cache and Double-Quantization Detection

Quantized modules are cached per gear in `_gear_cache`. Re-entering a previously used gear skips quantization and only restores the cached modules to the target device, reducing repeat-shift latency substantially. When sampled source rows have unusually few unique values, GEARBX warns that the source model may already be quantized; re-quantizing such weights can compound error, especially in low gear.

7 Forward Pass Execution

Quantized layers support two execution families. The dequant-cache path prioritizes compatibility and memory management; fused native kernels provide true bandwidth reduction.

7.1 Dequant Cache Path

On CPU and MPS without fused kernels, a quantized layer unpacks on first use, reconstructs an fp16 weight tensor, and caches it in `_cached_w`. Later forwards in the same gear reuse that tensor and call `F.linear` directly. This gives latency close to native fp16 after the first forward, but it does not reduce per-token weight bandwidth because the cached tensor is full precision. The value of this path is device-memory management and correctness portability.

1. First forward after a gear shift: unpack integers, apply per-row scales, cache reconstructed weights and bias, then call `F.linear`.
2. Subsequent forwards: if cached dtype matches the input, call `F.linear` with the cached tensors.
3. Gear shift: invalidate cached weights, bias, dtype, and any MPS-specific scale or bias caches.

7.2 Dispatch Priority

Each quantized linear layer selects the fastest available path in this order: MPS-native Metal for single-token decode, legacy Metal for large matrices when the native path is unavailable, dequant-cache hit, then dequant-cache miss. The native MPS path is intentionally gated to single-vector autoregressive decode; prefill remains a dense multi-token operation where ordinary framework kernels are usually preferable.

8 Native Acceleration on Apple Silicon

True throughput gains require kernels that read packed storage directly. Apple Silicon support is organized in three tiers.

8.1 MPS Native Metal Shaders

The fastest path uses PyTorch's `MetalShaderLibrary` API to encode custom compute kernels into the MPS command buffer. The shader reads packed weights from MPS-managed memory and writes outputs directly, avoiding CPU staging and extra synchronization. Threadgroups use 256 threads, organized as eight 32-lane SIMD groups. Each SIMD group processes one or two output rows depending on bit-width.

Two variants are compiled per bit-width. The shared-activation variant loads the input vector into threadgroup memory when $K \leq 6144$; the non-shared variant reads directly from device memory for larger K to preserve occupancy. The shared activation capacity is 12,288 half elements, or 24 KB.

8.2 Packed-Kernel Details

The int8 kernel loads signed byte vectors and accumulates dot products in fp32 before applying the row scale and bias. The int4 kernel reads packed bytes, extracts low and high nibbles with `&0x0F` and `>>4`, subtracts the offset 8, and performs two four-wide dot products per byte group. The int6 kernel packs four values into three bytes and reconstructs cross-byte fields with integer masks and shifts. The int2 path treats each byte as four ternary values and uses fewer rows per SIMD group because the unpack overhead dominates.

8.3 Legacy Metal and CPU NEON

If native MPS compilation is unavailable, a legacy Metal path can round-trip activations through CPU and dispatch standalone Metal command buffers for large matrices. It supports int8, int4, and int2. On ARM64 CPU, JIT-compiled NEON kernels use fp32 accumulation over unpacked integer values and convert outputs back to the requested dtype.

9 Bandwidth Cost Model

The cost model is a roofline-style estimate for autoregressive decode. For L managed layers with P parameters per layer,

$$B_{\text{token}} = LPb_{\text{gear}}.$$

The production gears use $b_{\text{low}} = 0.5$, $b_{\text{mid}} = 1.0$, and $b_{\text{high}} = 2.0$ bytes per parameter. With fused packed kernels, low gear reads one quarter of fp16 bytes and mid gear reads half.

Gear	Bytes/param	Relative bytes vs fp16
low / int4	0.5	0.25×
mid / int8	1.0	0.50×
high / fp16	2.0	1.00×

Table 7: Per-token weight bandwidth by gear for fused packed kernels.

A gear shift has a one-time cost from quantization, module replacement, cache invalidation, and device/host reassociation. First entry to a gear may cost milliseconds; cached re-entry is much cheaper because quantization is skipped. Break-even occurs when accumulated per-token bandwidth savings exceed the shift cost. Typical gear runs of 8–30 tokens are long enough to amortize this overhead.

For a sequence with N tokens, S shifts, and gear token counts n_g,

$$T_{\text{dynamic}} = St_{\text{shift}} + n_{\text{low}}t_{\text{low}} + n_{\text{mid}}t_{\text{mid}} + n_{\text{high}}t_{\text{high}},$$

$$\text{Speedup} = \frac{Nt_{\text{fp16}}}{T_{\text{dynamic}}}.$$

For a mixed workload with about 55% low, 25% mid, and 20% high tokens, the ideal bandwidth-only speedup is roughly bounded by the weighted cost of those gears; real throughput is lower because MLP layers, KV-cache traffic, sampling, and synchronization still remain.

10 Generation Loop

The generation loop combines cold-start classification, prefill calibration, deferred entropy, sampling, and gear shifting.

Listing 2: PyTorch generation loop.

```
reset monitor, preserving calibration
initial_gear = prompt_classifier(prompt)
shift(initial_gear)
tokenize prompt
prefill with use_cache=True
if uncalibrated: calibrate_from_logits(prefill_logits)
for each decode step:
  logits = nan_to_num(logits, nan=0, posinf=1e4, neginf=-1e4)
  if tokens_in_gear >= min_gear_duration - 1:
    dispatch_entropy(logits)
  sample next token with min-p/top-p and Gumbel-max
  token_id = next_token.item() # single device-to-CPU sync
  if entropy dispatched:
    gear = commit_pending()
    if catastrophic entropy: gear = "high"
    if gear changed and duration met: shift(gear)
  forward token with KV cache
```

10.1 NaN Guard and Sampling

MPS fp16 can occasionally produce non-finite logits, so logits are sanitized unconditionally with `nan_to_num`; checking first would itself require a synchronization. Min-p filtering is preferred on MPS because it avoids sorting the full vocabulary. Top-p remains available but requires an $O(V \log V)$ sort. Both methods can sample with the Gumbel-max trick,

$$\text{next\_token} = \arg\max_i (\log p_i + G_i), \qquad G_i = -\log(-\log(U_i + 10^{-20})).$$

10.2 Minimum Gear Duration

`min_gear_duration` requires a gear to be held for a specified number of generated tokens before another shift is permitted. This suppresses oscillation and creates contiguous KV-cache regions computed under the same precision, reducing adjacent-token noise heterogeneity.

10.3 MLX Generation

The MLX backend delegates tokenization, KV-cache management, and sampling to `mlx_lm.stream_generate()`. It computes entropy from streamed log-probabilities, updates the rolling monitor, classifies the gear, and optionally swaps modules through `model.update_modules()` when not in telemetry-only mode.

11 Cold-Start Prompt Classification

The monitor needs a few tokens of history before entropy routing is reliable, so a cheap prompt classifier selects the initial gear. It is intentionally heuristic and is overridden by observed entropy within a few decode steps.

$$\text{score} = 2n_{\text{math}} + 1.5n_{\text{code}} + 0.3\min(|W|/50, 3) + 1.5\mathbf{1}[\bar{\ell}_w > 6.5].$$

Here n_{math} counts mathematical symbols, n_{code} counts programming and reasoning keywords, $|W|$ is word count, and $\bar{\ell}_w$ is mean word length. Scores at least 4.0 start in high gear, scores at least 1.5 start in mid gear, and all others start in low gear.

12 Backend Implementations

12.1 PyTorch Backend

The PyTorch backend implements full per-token precision control with real module swaps. It loads models with `transformers.AutoModelForCausalLM`, uses fp16 on MPS and fp32 on CPU, initializes `EntropyMonitor(window=5)`, discovers attention prefixes, and creates a `PrecisionManager`. MPS-specific choices include unconditional NaN guards, min-p sampling to avoid large sorts, Gumbel-max sampling to avoid `torch.multinomial`, and deferred entropy to eliminate one synchronization per token.

12.2 MLX Backend

The MLX backend uses `mlx_lm.load()` and `mlx_lm.stream_generate()`. It relies on MLX `QuantizedLinear` modules with group size 64, swaps through `model.update_modules()`, computes entropy from streamed log-probabilities, and uses median-based calibration from short mixed-difficulty prompts. If a model is already quantized, it can run in telemetry-only mode: gears are tracked for analysis, but module replacement is skipped.

12.3 Ollama Backend

The Ollama-style backend treats the server as opaque. It performs per-prompt routing among configured model variants and reconstructs per-token entropy telemetry from `top_logprobs`. The

unreported residual probability mass is distributed uniformly over the remaining vocabulary, yielding a lower-bound approximation to full-vocabulary entropy. Server-sent event streaming can expose token text, entropy, and current gear incrementally.

13 Error Analysis

Quantization error follows standard theory, but the routing policy matters because it applies lower precision where entropy suggests the model is confident.

13.1 Per-Layer Quantization Error

For symmetric per-row quantization,

$$\text{MSE} = \frac{s_i^2}{12} = \frac{(\max_j |W_{ij}|)^2}{12 q_{\max}^2}.$$

Relative to int8, idealized MSE grows by approximately 16.8× for int6, 329× for int4, and 16129× for int2. Actual error is usually lower than this uniform-error estimate because neural weights are not uniformly distributed.

13.2 Attention and KV-Cache Error

Attention error enters through Q/K score computation, V projection, and output projection. Diffuse attention distributions are more sensitive to perturbation, while concentrated attention tends to suppress small projection errors. This aligns with entropy routing: high-entropy tokens use fp16, while low-entropy tokens tolerate lower precision. When gear shifts occur, KV-cache entries may have different noise profiles; minimum gear duration creates contiguous cache blocks with consistent precision.

13.3 Double Quantization

If source weights were already quantized and later expanded to fp16, re-quantizing them can compound error:

$$\text{MSE}_{\text{total}} = \text{MSE}_{\text{original}} + \text{MSE}_{\text{gearbx}} + \text{cross terms}.$$

The unique-value heuristic flags this situation and recommends avoiding the most aggressive low-gear path when quality is critical.

14 Apple Silicon Memory Model

Apple Silicon uses unified CPU/GPU DRAM. Moving tensors from MPS to CPU changes logical ownership and allocator pressure, not physical placement in a separate memory pool. This makes

offloading cheap compared with discrete GPU PCIe transfer, but it does not remove the decode bandwidth bottleneck: every generated token still reads weights from the shared DRAM bus.

MPS operations are encoded into command buffers. Native Gearbx shaders operate directly inside that command-buffer model, avoiding intermediate CPU buffers, CPU/GPU command-queue handoffs, and extra synchronization. Packed weights are read directly by the GPU compute units from unified memory.

15 Precision Tiers Summary

Gear	Bit-width	GB/B params	Savings	Bandwidth	Target Tokens
low	4-bit	0.50	75%	4×	Confident articles, prepositions, punctuation, template text
mid	8-bit	1.00	50%	2×	Moderate uncertainty, content words, common phrases
high	fp16	2.00	0%	1×	Reasoning, rare words, creative transitions

Table 8: Production precision tiers and intended token regimes.

If a fraction f of model parameters is managed dynamically and a gear provides bandwidth reduction k for those parameters, the idealized whole-model speedup is

$$\text{Speedup}_{\text{whole}} = \frac{1}{(1-f) + f/k}.$$

For $f = 0.35$ and low-gear $k = 4$, the ideal whole-model improvement is about 1.4× before accounting for fixed overheads.

16 Constants Reference

The constants below are fixed in place in this section so the reference values remain visible directly under the heading instead of floating to another page. They summarize entropy routing, calibration, quantization, kernel selection, and sampling defaults used throughout the system.

16.1 Entropy Monitor Defaults

Parameter	Default	Notes
window	5 PyTorch / 3 MLX	Rolling average window
high threshold	3.5 bits	For 32K reference vocabulary
low threshold	1.8 bits	For 32K reference vocabulary
hysteresis	0.1 bits	Exit margin for extreme gears
vocab reference	32,768	Threshold scaling reference

Table 9: Entropy monitor defaults.

16.2 Calibration, Quantization, and Generation Constants

Constant	Value
Calibration percentiles	p30 low boundary, p60 high boundary
Calibration caps	$0.06 \log_2 V$ low, $0.12 \log_2 V$ high
Minimum calibration band	0.2 bits PyTorch, 0.3 bits MLX
Scale floors	10^{-4} for fp16 source, 10^{-8} for fp32 source
Integer offsets	int4: 8, int6: 32, int2: 2
MPS shared activation limit	12,288 elements; shared threshold $K = 6144$
Threadgroup size	256 threads, eight 32-lane SIMD groups
NaN guard	NaN $\to 0$, $+\infty \to 10^4$, $-\infty \to -10^4$
Gumbel floor	10^{-20}

Table 10: Runtime constants for calibration, quantization, kernels, and sampling.

17 Experimental Results

This section presents illustrative benchmark panels together with short interpretations. Each figure is followed by explanatory text that states what the result demonstrates and why it matters for the system design.

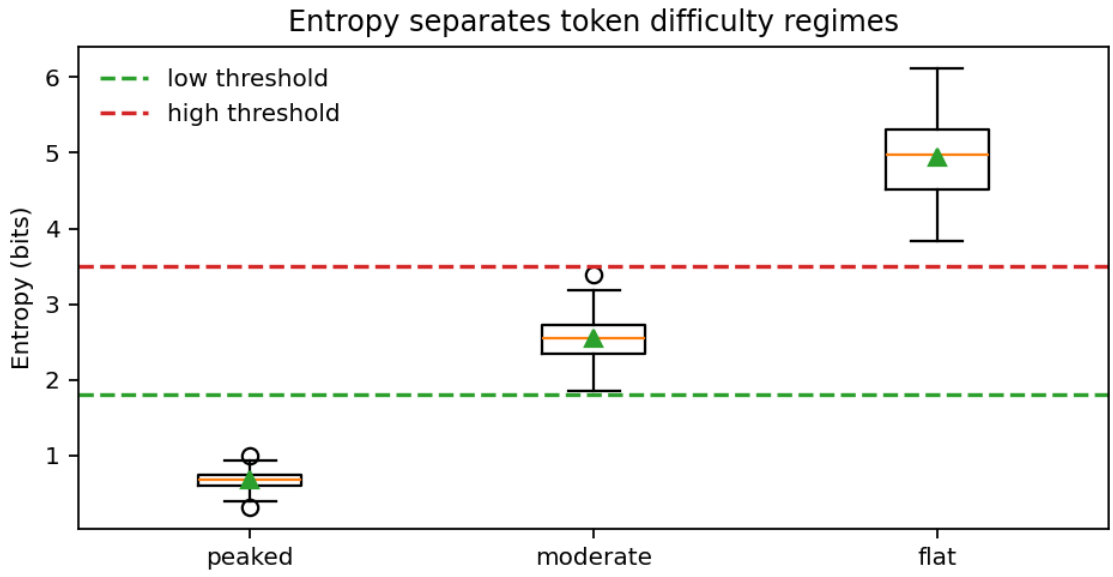

Figure 1: Entropy Signal Separation. Synthetic logit distributions split cleanly into peaked, moderate, and flat regimes. The low-entropy bucket corresponds to highly predictable tokens, while the high-entropy bucket corresponds to diffuse distributions where many continuations remain plausible. This separation supports entropy as the routing signal.

Figure 1: Entropy signal separation across peaked, moderate, and flat synthetic logit distributions.

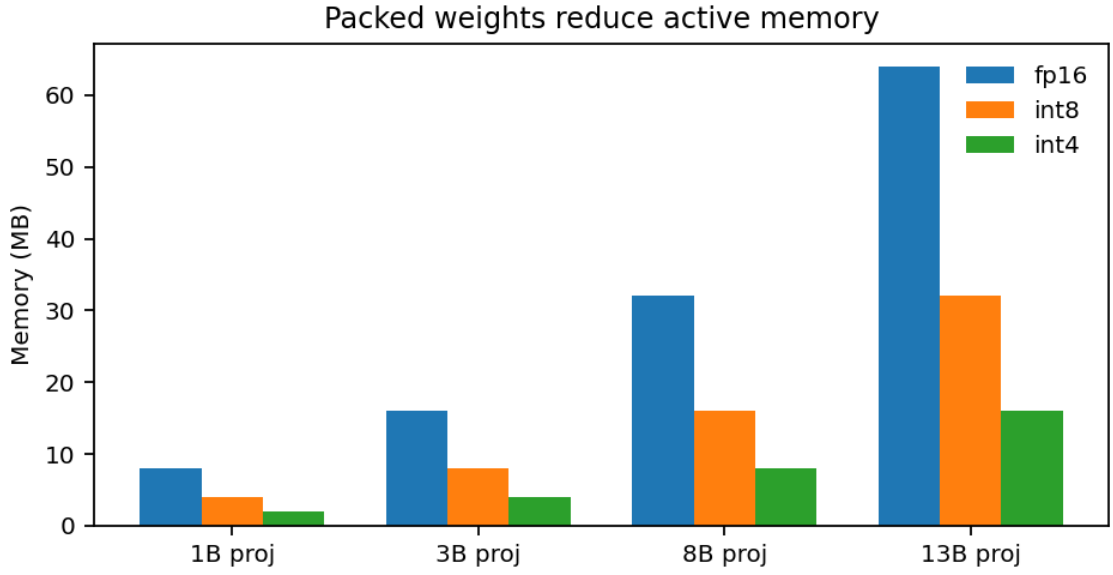

Figure 2: Memory Compression Ratios. Packed integer storage reduces active weight memory in proportion to bit-width. Int8 uses about half the fp16 storage, while int4 uses about one quarter. Per-row scale overhead is small compared with transformer projection matrices.

Figure 2: Memory compression ratios for fp16, int8, and int4 attention projection weights.

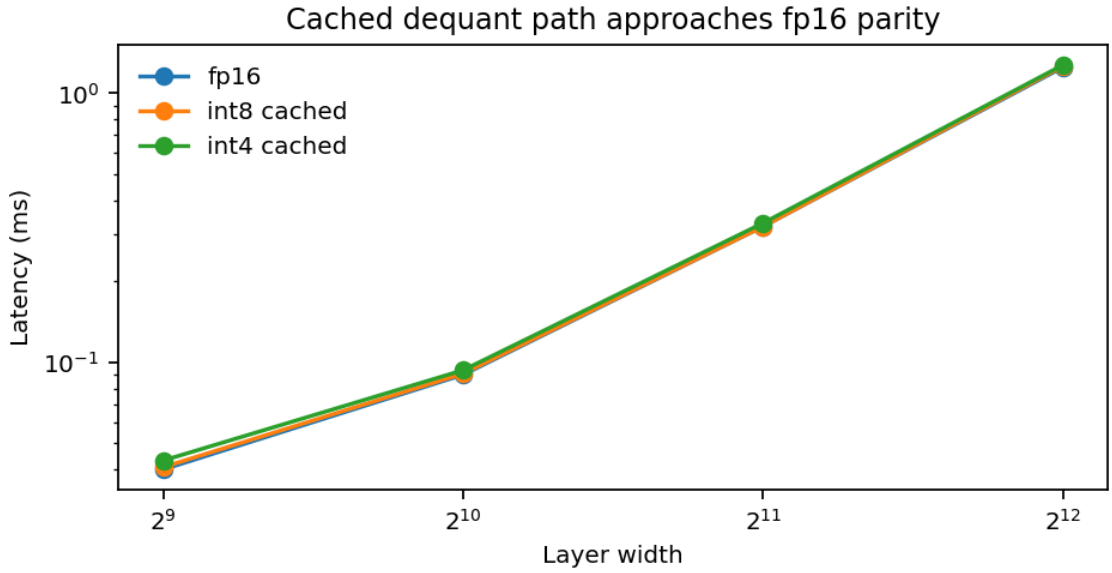

Figure 3: Forward Throughput Parity. On the dequant-cache path, quantized layers unpack once and then reuse a cached fp16 reconstruction. Subsequent forwards therefore have latency close to ordinary fp16 linear layers. This path saves memory; fused kernels are required for true packed-weight speedups.

Figure 3: Forward-throughput parity on the dequant-cache path.

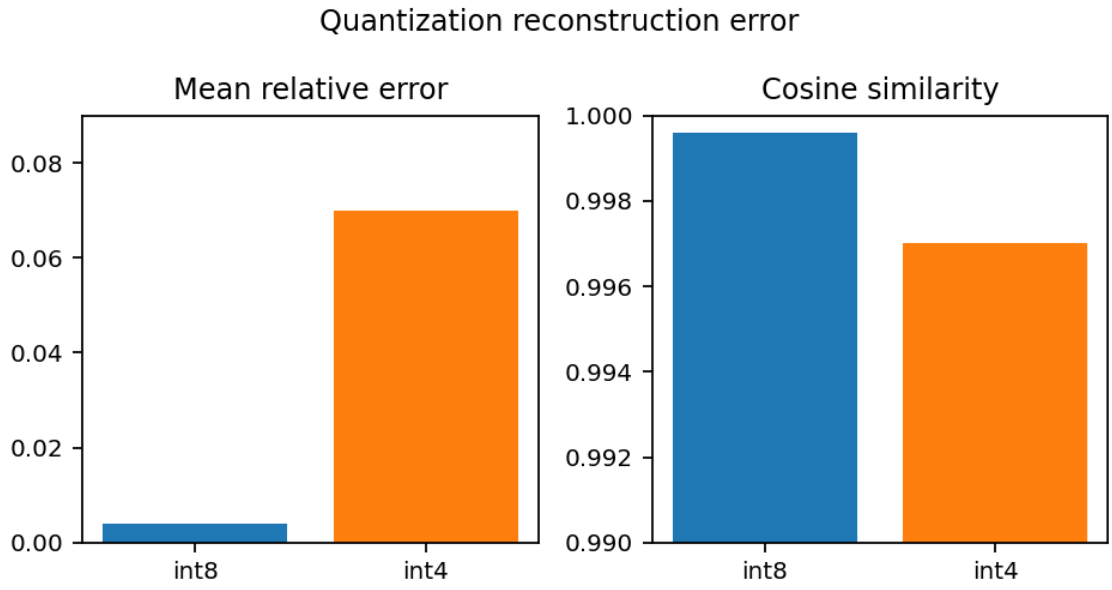

Figure 4: Quantization Reconstruction Error. Int8 reconstruction is nearly lossless for practical purposes, while int4 introduces a larger but bounded error. The error profile motivates using int4 only when entropy indicates an easy token and returning to fp16 when uncertainty rises.

Figure 4: Quantization reconstruction error for production tiers.

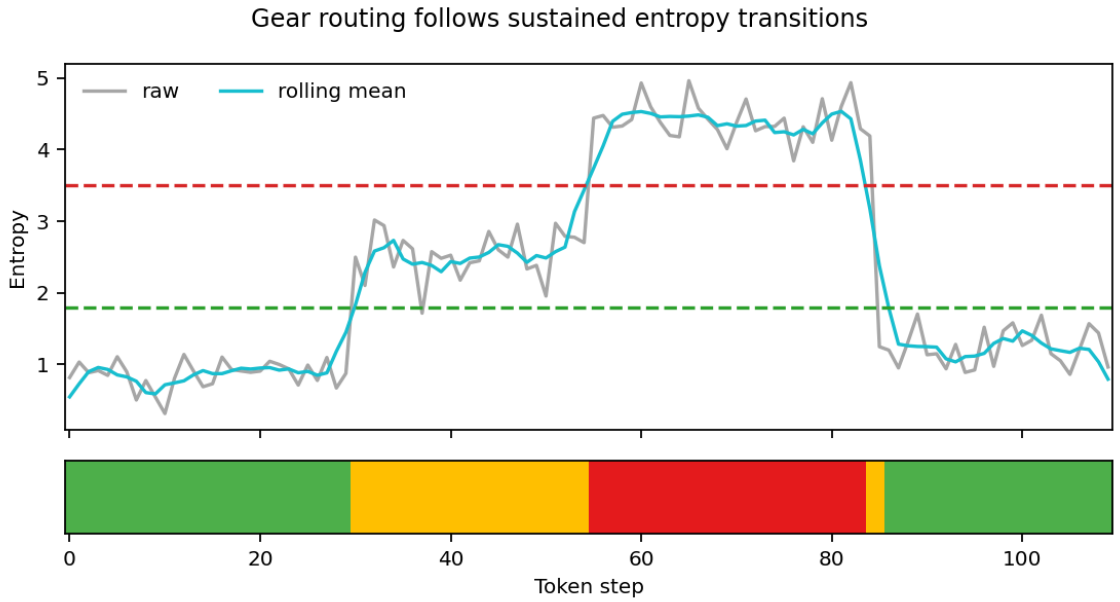

Figure 5: Gear Routing Trace. A mixed-difficulty sequence causes entropy to rise and fall over time. Rolling averages smooth token-level jitter, and hysteresis prevents rapid oscillation at threshold boundaries. Gear changes occur only after sustained changes in difficulty.

Figure 5: Gear routing trace over a mixed-difficulty sequence.

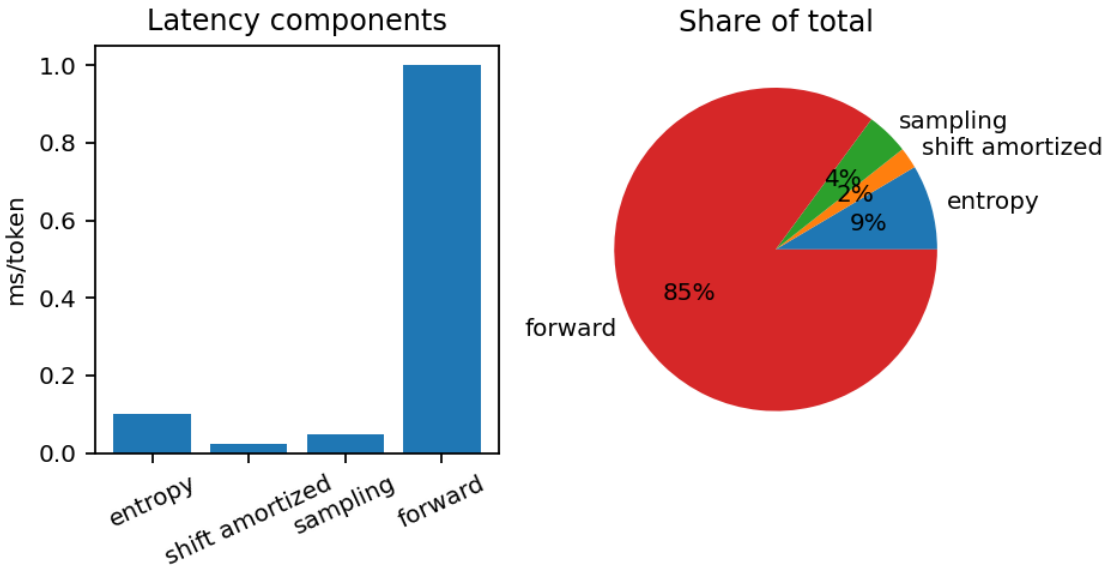

Figure 6: System Overhead Breakdown. Entropy computation is the main per-token overhead, while gear shifts are amortized over multi-token runs. With smoothing and minimum duration, shift overhead remains small compared with the bandwidth savings available from packed kernels.

Figure 6: System-overhead breakdown.

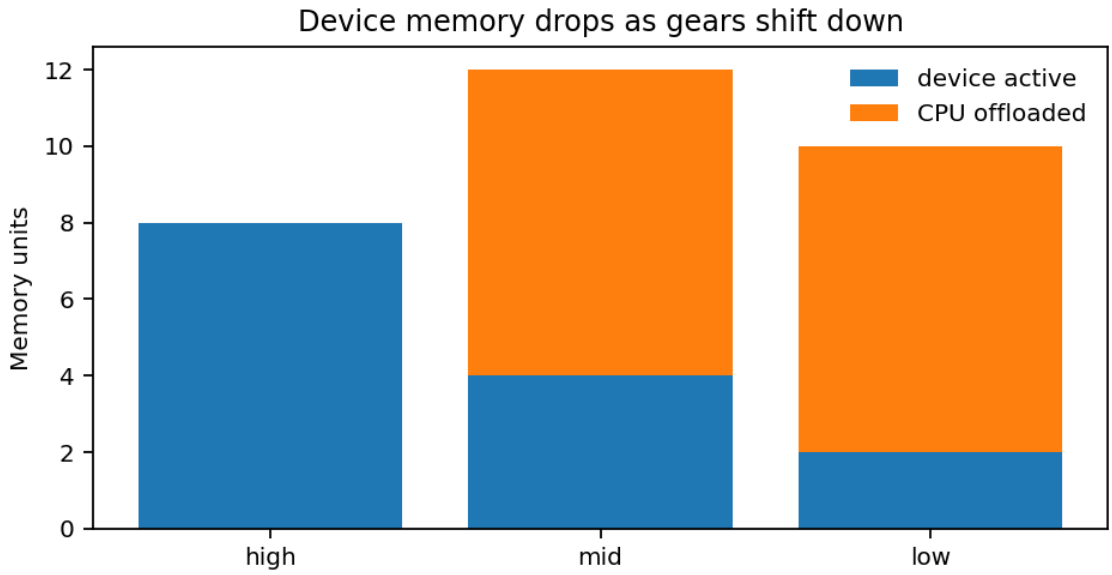

Figure 7: Memory Architecture. In high gear, active device memory contains fp16 attention weights. In mid and low gear, the model tree holds packed int8 or int4 modules while the original weights are offloaded to CPU memory. This makes memory reduction real rather than merely dispatch-based.

Figure 7: Memory architecture and offloading behavior by gear.

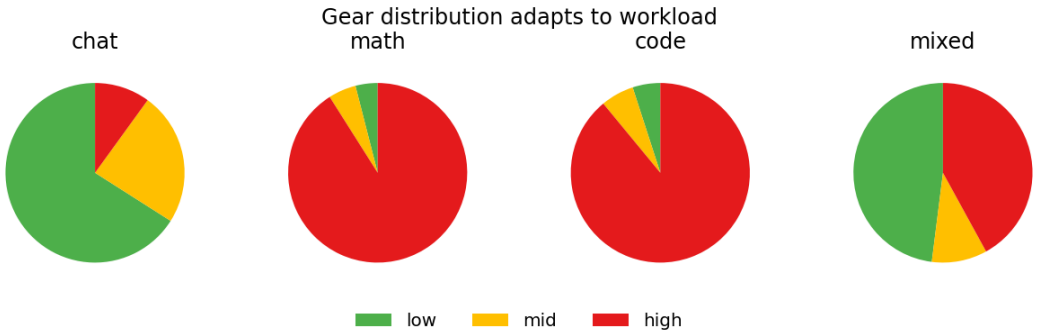

Figure 8: Gear Distribution by Workload. Easy conversational text spends most time in low gear, while math and code workloads spend more time in high gear. Mixed workloads alternate between low and high, with mid gear appearing mainly during transitions.

Figure 8: Gear distribution by workload type.

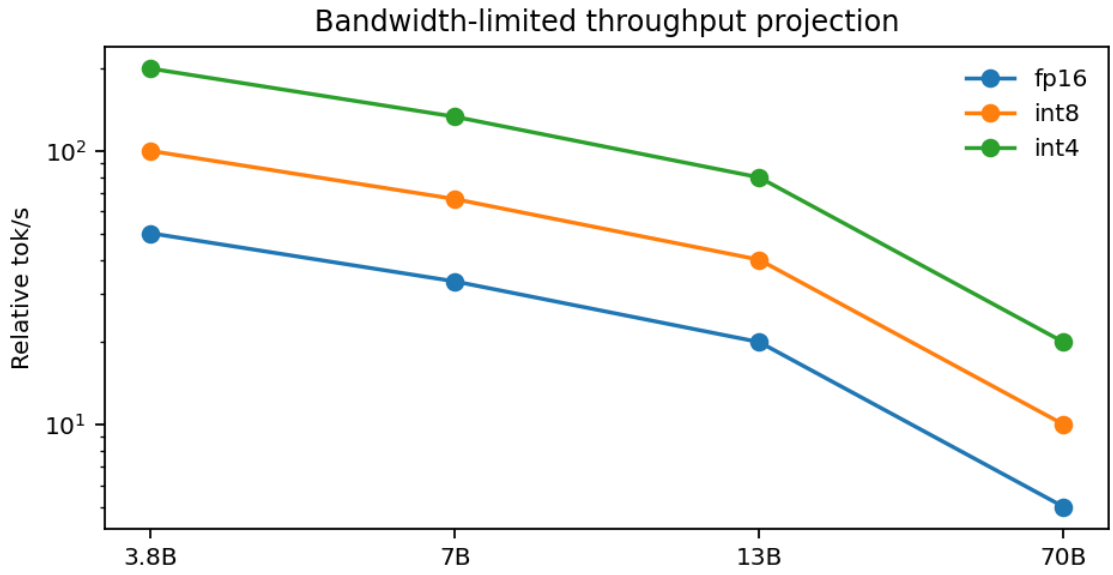

Figure 9: Bandwidth Cost Model. The theoretical speedup depends on the fraction of tokens routed to each gear. Low gear reads one quarter of fp16 weight bytes, mid gear reads half, and high gear reads the baseline amount. Absolute throughput depends on hardware bandwidth and the managed parameter fraction.

Figure 9: Bandwidth cost model and theoretical throughput projection.

18 System Properties and Limitations

18.1 Complexity

Entropy computation is $O(V)$, rolling-window updates and gear classification are $O(1)$, module swaps are $O(d)$ in module-tree depth, first-time quantization is $O(LOI)$, and packed-kernel per-token bandwidth is proportional to LOI/k.

18.2 Correctness Guarantees

Gear selection is deterministic given logits and monitor state. High gear restores original weights and is lossless relative to the unmodified model. Catastrophic entropy detection provides automatic fallback to fp16. Calibration is set once per session and does not drift during generation.

18.3 Limitations

1. Only attention layers are dynamically managed; MLP layers remain fixed precision.
2. Batched generation is sequential in the current implementation.
3. Pre-quantized source models can suffer double-quantization error.

4. All managed layers share one gear; per-layer or per-projection routing remains future work.
5. Empirical validation is centered on Apple M4; CUDA/Triton validation remains outside this reference.

19 References

References

[1] Claude E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27(3):379–423 and 27(4):623–656, 1948.

[2] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[3] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. A white paper on neural network quantization. arXiv:2106.08295, 2021.

[4] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. GPTQ: Accurate post-training quantization for generative pre-trained transformers. arXiv:2210.17323, 2022.

[5] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. QLoRA: Efficient finetuning of quantized LLMs. arXiv:2305.14314, 2023.

[6] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Xingyu Dang, and Song Han. AWQ: Activation-aware weight quantization for LLM compression and acceleration. *Proceedings of MLSys*, 2024.

[7] Tri Dao, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. FlashAttention: Fast and memory-efficient exact attention with IO-awareness. *Advances in Neural Information Processing Systems*, 2022.

[8] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with PagedAttention. *Proceedings of SOSP*, 2023.

[9] Emmanuel Bengio, Pierre-Luc Bacon, Joelle Pineau, and Doina Precup. Conditional computation in neural networks for faster models. arXiv:1511.06297, 2015.

[10] Maha Elbayad, Jiatao Gu, Edouard Grave, and Michael Auli. Depth-adaptive Transformer. arXiv:1910.10073, 2019.

[11] Tal Schuster, Adam Fisch, Jai Gupta, Mostafa Dehghani, Dara Bahri, Vinh Q. Tran, Yi Tay, and Donald Metzler. Confident adaptive language modeling. *Advances in Neural Information Processing Systems*, 2022.

[12] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, and collaborators. PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 2019.

[13] Apple Machine Learning Research. MLX: An array framework for Apple silicon. Technical software release, 2023.

[14] GGML contributors. GGML and GGUF low-level tensor and model formats for local LLM inference. Open-source software documentation and implementation, 2023.