HARDWARE-AWARE LANGUAGE MODEL QUANTIZATION FOR A MANY-CORE AI INFERENCE ACCELERATOR

Anonymous authors
Paper under double-blind review

ABSTRACT

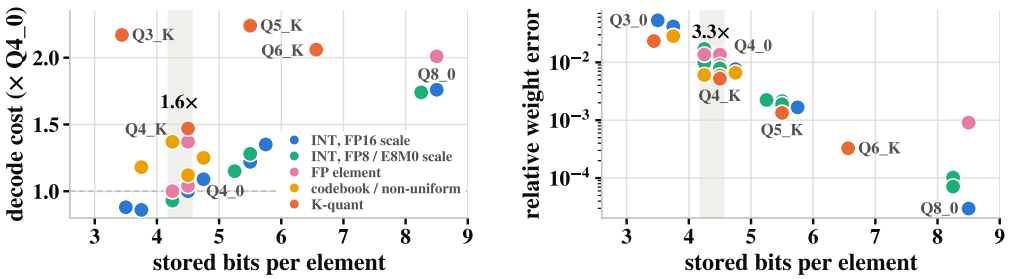
Edge accelerators for ML/AI workloads are designed around trade-offs between peak performance, memory capacity and bandwidth, power, and cost, with vendor economics driven by volume rather than per-unit margin. Because deployed silicon cannot be refreshed on the cadence of model releases, gains must increasingly come from the algorithmic side: pushing out the Pareto frontier of accuracy, memory footprint, and throughput within the constraints of hardware already in the field. Given the extensive work on post-training quantization, we choose it as the primary compression lever and treat it as a full inference-stack optimization problem, jointly optimizing accuracy, storage, and throughput rather than minimizing quantization error alone. We account for hardware features such as supported data formats, computational units, and the instruction set architecture, alongside structural constraints imposed by the runtime and compiler. Beyond the formats natively supported by the hardware, we treat the format design space itself — element and scale datatypes, group size, and hierarchical scales — as part of the optimization process, covering standardized block formats such as MXFP and NVFP as well as non-standard constructions. Algorithmically, we evaluate mixed-precision allocation and codebook-based quantization across dense and mixture-of-experts models from 4B to 35B parameters. We share with the wider community insights on what works well on paper and what actually improves the Pareto frontier across accuracy, memory footprint, and throughput for models deployed on edge silicon. We demonstrate our hardware-aware quantization methodology on a many-core edge accelerator with over a thousand RISC-V cores, achieving 34%, 67%, and 24% decode speedups over the llama.cpp K-quant of equal storage at 4.3×, 3.7× and 3.3× weight-memory reduction over BF16, within one point of its mean accuracy, on Qwen3.5-35B-A3B, Qwen3.5-4B, and Llama-3.1-8B respectively.

1 INTRODUCTION

Generative and multimodal models in the 0.1-35B parameters range are becoming the dominant workload for edge AI accelerators. Their use cases span from automotive cockpit assistants, industrial and retail smart vision to agentic workflows in mobile and on-premise setups Wang et al. (2025a;b), Robert Bosch GmbH (2026); Mercedes-Benz Group AG (2026), Sharshar et al. (2025), Belcak et al. (2025). To address these diverse use-cases, edge accelerators are designed around trade-offs between throughput, memory capacity and bandwidth, power consumption, and cost. Different designs diverge sharply in types of instructions and the programming model they support. They differ in compute throughput, memory bandwidth, the numeric formats their datapaths support, and whether their memory is unified with the host or private to the device Podbucki & Szalwach (2026). NVIDIA Jetson Orin, a GPU based accelerator, runs a single instruction multiple thread programming model featuring 64 GB at 204.8 GB/s within 15-60 W power budget at the high-end. Hailo-10H is a dataflow architecture with 4-bit weights native to its datapath, delivering 40 TOPS at INT4, with 4-8 GB of private LPDDR4 DRAM inside a 2.5 W power envelope. Our target accelerator is the Esperanto Technologies ET-SoC-1 Ditzel (2022). It is a spatial many-core accelerator for ML workloads comprising 1088 RISC-V cores, each with its own scalar, vector, and tensor units. Each compute unit within a core comes with its own independent program counter but can coordinate across multiple sets of cores through a network-on-chip and custom coordination instructions.

Quantization is one of the most effective compression techniques on edge hardware. Usually posed as an algorithmic problem i.e., find a smaller representation of the weights that preserves accuracy,

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.



(a) Decode cost

(b) Weight error

Figure 1: Near equal bit-width formats differ by 1.6× in (a) throughput and 3.3× in (b) weight error on ET-SoC-1, the most accurate among the slowest. Bit-width alone cannot choose among them.

its effect on inference speed is settled several layers below: it is a function of the kernel that decodes the format, the compiler that schedules it, and the memory hierarchy data moves through. Stored bit-width is routinely treated as a proxy for both footprint and speed but as we show, this does not always yield the best results when both footprint and inference speeds matter. A quantization format one device consumes directly may, on another, be unpacked through costly table lookups or a multi-instruction conversion sequence, degrading inference speed. Which format is preferable is therefore a property of the format, datapath, and runtime triple, and an ordering established on one platform carries no guarantee on another. Choosing a format in isolation and leaving the kernel, compiler, and memory hierarchy to absorb the consequences forfeits throughput that no downstream tuning can recover. For ET-SoC-1, as shown in Fig. 1, the decode speed and weight error vary by 1.6× and 3.3× when bits per element (BPE) changes by only 0.25 over a vanilla 4-bit block quantization. The recent block-scaled formats — MXFP4 and NVFP4 — illustrate this, and Section 3.2 characterizes their behaviour on a datapath that supports neither natively.

Any quantization scheme maps onto a hardware through its compilation and runtime stack. Edge inference is predominantly single-user, where decoding is bandwidth-bound and the quality of the low-level kernels dominates inference speed. An effective compiler should enable new workloads and quantization formats quickly and runtime should add minimal overhead so that kernel execution dominates initialization or setup. We therefore build on open-source llama.cpp, which operates directly on quantized blocks behind a flat C interface with minimal dependencies. It supports the latest open-source models across a variety of hardware Gerganov (2023). Its quantization families span much of the design space explored in the literature. ET-SoC-1 is integrated in llama.cpp. Its upstream version supports only a limited set of quantized matrix-multiplications so evaluating a broader quantization format space on it requires supplying new kernels rather than selecting existing ones. To do so at scale, we employ an in-house hardware-in-the-loop agentic kernel generation framework. Using a detailed specification of the target architecture, it generates and iteratively refines C kernels for candidate formats with expert approval. This continuous improvement with hardware in the loop enables hardware-aware quantization.

We analyze the quantization format design space under the constraints imposed by ET-SoC-1 and similar accelerators with no specialized sub-8-bit datapaths. Our methodology of hardware-aware quantization comprising of kernel generation for different quantization formats, profiling them on the target hardware and choosing between them under throughput/storage constraints, however, is repeatable for any edge accelerator. We also respect the limitations set by the open-source llama.cpp runtime like all experts of a layer sharing one quantization format and a single-user memory configuration among others. Avoiding customization over the open-source version is purely a choice made by the authors for wider compatibility of ET-SoC-1 with the llama.cpp ecosystem. Under these constraints, we jointly optimize the throughput, memory footprint and accuracy axes through a mixed-precision quantization format allocation strategy driven by hardware profiled data and derive insights into what works best for the stack under consideration. Our contributions are fourfold: (1) a characterization of the quantization format design space along four dimensions, namely element datatype, scale datatype, group size and zero point, identifying for each the hardware capability it relies on and its degradation when that support is absent (Section 3.1); (2) measured decode throughput per format on the dense and Mixture-of-Experts (MoE) matrix-multiply paths, which eliminates several accuracy-competitive formats on performance grounds and leaves a **viable set** of 14 (Section 3.4); (3) a **hardware-aware mixed-precision allocator** that assigns each tensor a format from

the viable set, minimizing a sensitivity objective derived from the model output subject to separate storage and decode-time budgets priced from profiled kernels (Section 4); and (4) an evaluation on the ET-SoC-1 across two dense and one MoE model from 4B to 35B parameters (Section 5), where, at equal storage and within one point of mean accuracy, our allocations decode **34%**, **67%** and **24%** faster than the llama.cpp K-quants on Qwen3.5 35B A3B, Qwen3.5 4B and Llama 3.1 8B.

2 Related Work

Post-training quantization techniques for transformer models — GPTQ, AWQ, SmoothQuant, and QuaRot — reduce error at a fixed representation. The representation itself trades accuracy against decode cost. It is well established at the kernel level that reduced precision need not reduce latency: QServe attributes the gap to dequantization overhead inside the GEMM main loop Lin et al. (2025), and FP6-LLM co-designs a kernel to make a non-native six-bit format execute efficiently on tensor cores Xia et al. (2024). The HAWQ family Dong et al. (2019; 2020); Yao et al. (2021) scores each layer by Hessian curvature and solves a constrained integer program. APTQ allocates across blocks by Hessian trace Guan et al. (2024), SliM-LLM by a group-wise salience score selected to minimize layer-output divergence Huang et al. (2025), and MixLLM by a global criterion at output-feature granularity Zheng et al. (2026). The sensitivity signal in most of these methods is a layer-local curvature term. The storage budget, expressed as a footprint or an average bit-width, is used as a proxy for throughput that may not replicate across hardware. A third assumption is implicit: that formats are interchangeable at equal stored width. We show they are not. Element encoding, scale datatype, group size, and grid uniformity each meet the datapath differently. Section 3.4 shows that formats within 0.25 BPE of one another differ significantly in decode cost on our target. We therefore allocate over format identity rather than bit-width alone, draw candidates from kernels profiled on the device, and weight each by an importance term measured at the model output.

MoE models raise a distinct question, since expert activation is sparse and uneven. One group of prior work protects the routing distribution: MoEQuant Chen et al. (2025b) and EAQuant Fu et al. (2025) construct expert-balanced calibration sets so that rarely selected experts are not starved of data, while EAC-MoE Chen et al. (2025a) conditions compression on expert selection and realigns router behaviour after quantization.The other treats the expert as the unit of precision assignment with MxMoE Duanmu et al. (2025) using an integer program over MoE linear blocks minimizing the weighted geometric mean of an accuracy and throughput cost under a memory budget and generating mixed-precision kernels for the result. Under sparse activation, methods budgeting for storage alone optimize the wrong metric: an expert block with E experts occupies its full footprint but contributes only k/E of it to per-token weight traffic during the decode phase, so the two costs rank tensors differently. We therefore constrain storage and measured execution time separately, and allocate over bit-widths from viable format families based on profiled kernel data.

3 The Quantization Format Design Space

We characterize the space of block-quantized weight formats, spanning standard and custom realizations. We first define the space and locate existing formats within it, then examine what each region demands of the decoding hardware. We then determine the viable set under ET-SoC-1's constraints.

3.1 Design dimensions

A block-quantized format is specified by four choices: the group size over which a scale is shared, the datatype of that scale, whether a zero-point accompanies it, and the datatype of the elements which can either belong to a uniform/non-uniform grid or be defined by a codebook. Table 1 captures the range of formats considered in this paper. A per-tensor outer scale is required only with FP8 block scales, where scale magnitudes fall below the FP8 normal range. BPE follows from these choices directly. Two formats can agree on stored width while differing entirely in how a stored value becomes an operand.

Existing formats cluster into four regions in this space, distinguished by the work reconstruction requires: (1) *flat-block integer* formats, the GGUF legacy family Q4_0 through Q8_0 and the asymmetric variants used here, pair a uniform grid with one floating-point (FP) scale and, for asymmetric

Table 1: Candidate formats in the four design dimensions, with effective BPE and the capability each requires. Last two columns report the decode time (averaged over tensor shapes across models) for each format normalized to the Q4_0 time, separately for the dense and MoE paths.

Scale datatype	Format label	Element datatype	Group size	BPE	Requires	Dense decode (×)	MoE decode (×)
FP16	Q3_0	INT3	32	3.50	unpack	0.88	1.00
	Q3_1	INT3 + ZP8	32	3.75	unpack	0.86	0.94
	Q4_0	INT4	32	4.50	unpack	1.00	1.00
	Q4_1	INT4 + ZP8	32	4.75	unpack	1.09	0.95
	Q5_0	INT5	32	5.50	unpack, hi bit	1.22	1.14
	Q5_1	INT5 + ZP8	32	5.75	unpack, hi bit	1.35	1.04
	Q8_0	INT8	32	8.50	native	1.76	1.38
	QFP4_0	FP4 E2M1	32	4.50	FP4 or LUT	1.04	0.97
	QFP8_0	FP8 E4M3	32	8.50	FP8 or LUT	2.01	1.86
	Q3_CB8	3-bit index	32	3.75	LUT, 8 centroids	1.18	1.35
	Q4_CB16	4-bit index	32	4.75	LUT, 16 centroids	1.25	1.06
	IQ4_NL	4-bit index	32	4.50	LUT, 16 centroids	1.12	1.05
FP8 E4M3 (+ FP32)	Q4_0_FP8	INT4	32	4.25	FP8→FP scale	0.93	0.91
	Q4_1_FP8	INT4 + ZP8	32	4.50	FP8→FP scale	1.05	1.02
	Q5_0_FP8	INT5	32	5.25	FP8→FP scale	1.15	1.15
	Q5_1_FP8	INT5 + ZP8	32	5.50	FP8→FP scale	1.28	1.09
	Q8_0_FP8	INT8	32	8.25	FP8→FP scale	1.75	1.33
	NVFP4	FP4 E2M1	16	4.50	FP8→FP, FP4	1.37	1.31
E8M0	MXFP4	FP4 E2M1	32	4.25	exponent add, FP4	1.00	0.91
	BFP4	INT4	32	4.25	exponent add	0.93	0.92
	BFP8	INT8	32	8.25	exponent add	1.74	1.32
6-bit+FP16	IQ4_XS	4-bit index	32 in 256	4.25	LUT, 2-level scale	1.37	1.20
	Q3_K	INT3	16 in 256	3.44	2-level scale	2.17	1.74
	Q4_K	INT4	32 in 256	4.50	2-level scale	1.47	1.55
	Q5_K	INT5	32 in 256	5.50	2-level scale	2.24	2.05

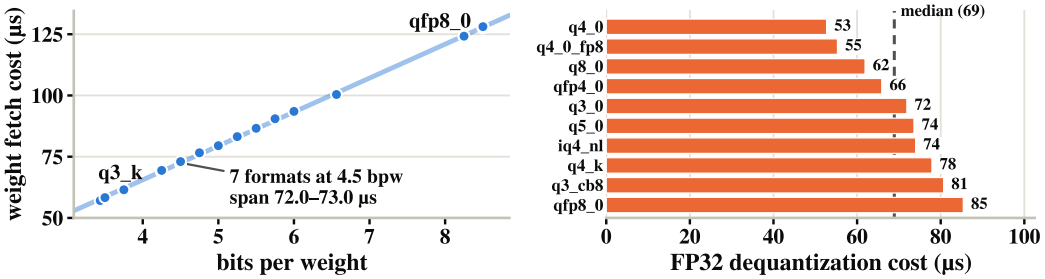
formats, a zero point per block, and were designed for commodity CPU SIMD; (2) *hierarchical-scale* formats quantize the scale in turn, as Q4_K stores an FP16 super-block (group size 256) scale and minimum alongside eight sub-block (group size 32) scales and minima, each at six bits and packed across byte boundaries; (3) *low-precision-element* formats, the Open Compute Project's MXFP Rouhani et al. (2023), NVIDIA's NVFP4 and block floating point (BFP), assume sub-8-bit element decode together with power-of-two (PoT) or FP8 scale arithmetic; and (4) *lookup* formats require a per-element lookup from a predefined or learnt codebook, covering the GGUF i-quants (IQ4_NL and IQ4_XS), and the formats of Ahmad et al. (2025) denoted QN_CB.

3.2 Dequantization cost structure

Reconstructing a quantized weight decomposes into work performed once per block and work performed once per element. The division is not fixed by the format: a scale shared across a block factors out of the inner product, so whether it is applied once per block or once per element depends on the datapath rather than the encoding. Depending on what the multiply-accumulate (MAC) unit will accept as an operand, three regimes can be identified:

- **Regime 1**: Packed elements consumed together with their scales, as recent block-scaling hardware (such as Blackwell GPUs) does, with reconstruction absorbed into the pipeline.
- **Regime 2**: MAC units accept the element's arithmetic class but not its width, elements are widened by shift and mask or through a sequence of bit manipulation steps, accumulation proceeds in that class, and the shared scale is applied once to the accumulator.
- **Regime 3**: The MAC unit requires a converted value, so every element passes through an arithmetic conversion sequence or a table lookup before it can be consumed.

The regime which applies is a property of the unit executing the multiply-accumulate. On Blackwell GPUs, where conversion is absorbed into the matrix pipeline, NVFP4 outruns INT4 despite storing more BPE; on hardware lacking an FP4 execution path the ordering inverts and the format is expanded to FP16 ahead of every matrix multiply degrading throughput. In the second and third regimes the kernel must convert the scale into a usable operand, and its datatype sets that cost. An FP16 scale already is one. An E8M0 scale is executed by a single exponent-field manipulation by



(a) Illustration of fetch cost vs BPE.

(b) Dequant cost of select formats from the viable set.

Figure 2: Measured weight fetch and dequantization cost per format on a dense 4096 × 2048 matvec.

a dedicated instruction, or otherwise by a shift and mask. An E4M3 scale requires an FP8-to-FP16 conversion, native where the datapath supports FP8 and otherwise a bit-manipulation sequence or a 256-entry lookup. In both regimes this is paid once per block and amortizes with group size.

Non-uniform grids add a per-element indirection that cannot be amortized. If the multiplier accepts native numerics like small integers or floating point formats such as INT8 or FP8, the main cost is incurred in the lookup. The lookup cost depends on where the table resides. A small codebook can reside in registers or nearby on-chip memory while a big one may have to be stored in a distant but larger memory incurring a higher data movement penalty. Where dequantization executes matters as much as what it costs. Conversion fused into the accumulation loop adds arithmetic to the inner loop but leaves the block scale hoisted and the compressed tensor intact in memory. Conversion that must be materialized first — because the consuming unit accepts only a dense tile of a wider type, or because a separate unit performs the conversion and communicates through memory — writes out the widened weights and reads them back, so the scale is applied per element and the compressed representation never reaches the multiplier.

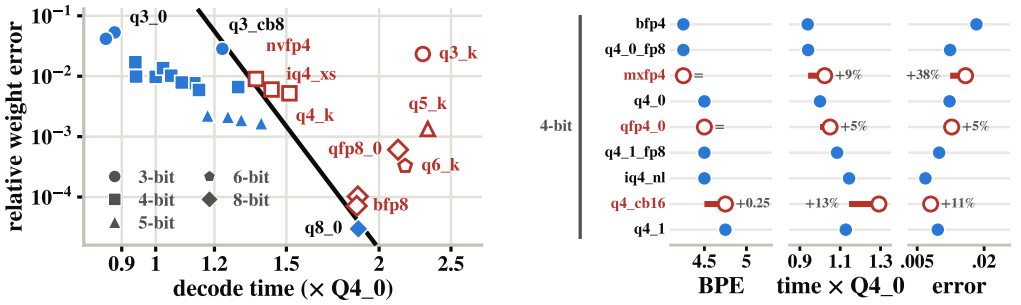
3.3 Target hardware architecture and features

ET-SoC-1 is a spatial many-core accelerator for ML workloads comprising 1088 RISC-V cores, each with its own scalar, vector, and tensor units. Like major accelerators Codrescu et al. (2014); Firoozshahian et al. (2023); Tenstorrent (2026), ET-SoC-1 features a software-programmable on-chip fast memory paired with 32 GB of LPDDR4x main memory, and compute units that can deliver up to 80 TOPS (INT8). Placing tensors and their attributes in the fastest memory tier boosts speed, but its limited capacity relative to main memory demands spatio-temporally optimized kernels that orchestrate data movement and compute across the fabric. More details about the ET-SoC-1 are given in Appendix A.

On ET-SoC-1, prefill and decode phases are served by different units. Prefill's wide matrix multiplications run on the tensor units, which accept FP32, FP16, and INT8; single-stream decode, producing narrow matrix-vector shapes, runs on the vector units, which provide only FP32 and INT32. Neither offers a type narrower than INT8, so no format reaches the first regime on either path. We quantize the model weights, accumulating against unquantized activations. Dynamically quantizing activations costs more as explained by profiled kernel measurements in Appendix B.4. Minimal variation of the prefill phase as a function of the quantization format is observed as explained in Table 6 in Appendix B.1. During the decode phase, every candidate format must therefore be converted element-by-element to FP32 before the vector unit can consume it, placing all of them in the third regime where they separate on conversion cost rather than on stored width.

3.4 ET-SoC-1 profiled data and trends

The kernels for the formats in Table 1 were produced by an expert-guided agentic co-design stack combining automated generation with expert intervention similar to Zhang et al. (2026); its details are not a part of this manuscript. All formats were developed under the same process and comparable optimization effort, with some formats like Q4_0 undergoing additional finetuning. Absolute figures are specific to these kernels and this target; the methodology, however, transfers to other kernel realizations on this accelerator as well as other accelerators given their own kernels.



(a) Viability (b) Pruning within the 4-bit range

Figure 3: Selection of the viable formats: (a) Weight error against decode time (relative to Q4_0): formats below the line are viable (solids), leaving 17 of 26. (b) In the 4-bit range, a format dominated by another on storage, time and error is removed, leaving 14 viable formats.

The end-to-end kernel cost can be broken down into the cost of fetching the weights and the dequantization cost. Having established that during the decode phase all quantization formats considered in Table 1 undergo a per-element casting operation prior to the FP32 vector unit and that activations are unquantized, it comes as no surprise that the cost for fetching activations (22.9 μs) and the computational cost of the vector unit (≈30 - 31 μs) remain stable across formats. Fig. 2 illustrates the fetch and dequant cost for a subset of the formats in Table 1 for a dense 4096 × 2048 matvec. It can be seen from Fig. 2a that the fetching cost scales linearly with the BPE for each format since this depends entirely on the DRAM to L2 bandwidth. The dequantization cost on the other hand depends on the dequantization complexity, including differences in bit-unpacking, scale reconstruction, or table lookup, as explained in Section 3.2. The two costs are not directly additive to the end-to-end kernel time due to scheduling and overlap between the phases in the actual end-to-end compute.

The end-to-end kernel time normalized by the Q4_0 kernel time, geometrically averaged across the tensor shapes and further pooled across all models, is separately reported for the dense and MoE matrix paths in Table 1. The relative orderings established therein closely follow the qualitative discussion in Section 3.2 with the fallback mechanism that allows for a quantization format's execution in the absence of native hardware support and demonstrate the following key findings:

- Formats with elements of the low-precision FP datatype result in a slightly higher cost compared to equivalent formats with INT elements. As expected, FP to FP translations are costlier compared to INT to FP.
- Similarly, all codebook formats which require an explicit look-up operation cost relatively more on this hardware, with further dependency on the size of the codebook. Memory lookup transaction is slightly costlier than casting compute.
- Hierarchical formats incur multiple memory transactions for scales and cast instructions for format conversion. This complicated dequant process results in a significantly higher cost despite being at a similar or lower BPE. In addition to having two-level scales, each of which requires its own dequant, the cost is further exacerbated for 3-bit and 5-bit formats due to nibble alignment. Further, the large group size of these formats requires multiple cache lines (64 bytes on the ET-SoC-1) for the complete group to be loaded.
- On the MoE path, formats differ relatively less from one another than on the dense path. The routed-expert kernel has a larger format-independent cost than its dense counterpart.
- Three sources contribute to low single digit measurement variations: Changes in instruction ordering due to small changes in the C code, runtime chip conditions (temperature, voltage levels) and sampling noise induced by the on-chip performance monitoring units.

Cost alone does not determine which formats are worth carrying forward. Fig. 3a plots each format's decode time against median relative weight error, with the non-dominated set marking the frontier available on this target. Formats above and to the right are dominated: another format reconstructs more accurately at equal or lower cost, so no allocation would prefer them. The frontier runs from Q3_0, cheapest and least accurate, to Q8_0, and the exclusions are not marginal. Q3_K and Q5_K cost roughly twice Q4_0 while several flat-block formats reach comparable error below it. Within

each bit-width band the same argument applies more finely. Fig. 3b compares formats of comparable stored width on the three quantities the allocator sees: BPE, decode time, and weight error. A format is retained only if it is not dominated on all three. At 4 bits this removes MXFP4, which Q4_0_FP8 matches in width while running 9% faster and 38% more accurately, and QFP4_0, which trails Q4_0 by 5% in time and in error at equal width. Applying this test across bands reduces Table 1 to 14 formats (Q3_0, Q3_1, Q3_CB8, BFP4, Q4_0, Q4_0_FP8, Q4_1, Q4_1_FP8, IQ4_NL, Q5_0, Q5_0_FP8, Q5_1, Q5_1_FP8, Q8_0), which constitute the candidate set for the allocator of Section 4.

4 Mixed-Precision Allocation

We formulate mixed-precision allocation as an assignment of formats and bit-widths from the viable set (Section 3.4) to tensors across the model. The objective measures each candidate's effect on the model output rather than on its own layer's reconstruction, and the constraints are the quantities that bind on the target: storage from the format's block layout, and execution time from profiled kernels.

Loss quantification: With $\mathcal{T}$ denoting the set of quantizable tensors and $\mathcal{F}$ the viable format set from Section 3.4, an assignment $z_{t,f} \in \{0,1\}$ assigns each tensor t exactly one format f, enforced by the constraint $\sum_f z_{t,f} = 1$. Let $p_\theta(y \mid x)$ be the model's predictive distribution over the next token y given a calibration context x, and θ_0 the vectorized full-precision weights. A tensor $W_t \in \mathbb{R}^{R\times K}$ is applied at every token position s of x, mapping the layer input $h_s \in \mathbb{R}^K$ to the output $o_s = W_t h_s$, and the sensitivity of the log-likelihood to o_s is captured by the gradient $g_s = \partial \log p_\theta(y \mid x)/\partial o_s$, evaluated at θ_0 with $y \sim p_{\theta_0}(\cdot \mid x)$. Quantizing W_t to a format f, captured by the quantization function $Q_f(\cdot)$, introduces an error $\mathcal{E}_{t,f} = Q_f(W_t) - W_t$ in the weights. Generalizing this across the model, the effect of quantization is captured by a perturbation in the weights given by $\theta_0 + \Delta$ where the contribution of tensor W_t to Δ is given by $\Delta_t = \sum_f z_{t,f} \operatorname{vec}(\mathcal{E}_{t,f})$, and Δ is zero for all parameters outside $\mathcal{T}$. We quantify the effect of the perturbation Δ by the KL divergence between the full-precision model and the perturbed model:

$$L(\theta) \;=\; \mathrm{E}_x[D_{\mathrm{KL}}(p_{\theta_0}(\cdot \mid x) \,\|\, p_\theta(\cdot \mid x))]\,, \tag{1}$$

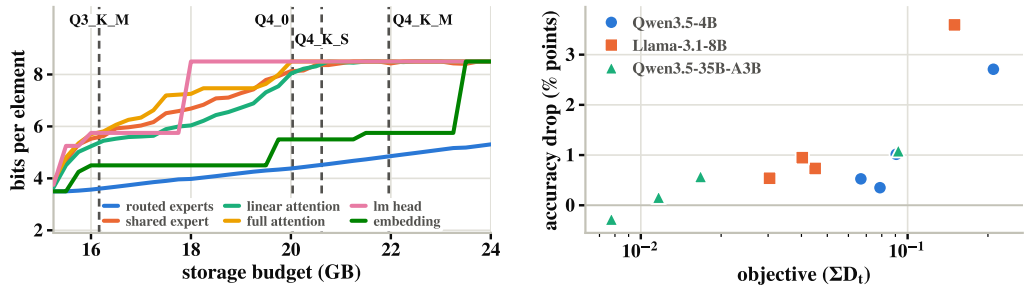
where the full-precision model supplies the reference distribution, so only calibration contexts, and no corpus labels, are required.

Numerical tractability: The value and gradient of (1) both vanish at θ_0, so its Taylor expansion is led by the Hessian, which equals the Fisher information (Proposition 1):

$$L(\theta_0 + \Delta) = \Delta^\top F \Delta + O(\|\Delta\|^3), \qquad F = \mathrm{E}_x \mathrm{E}_{y\sim p_{\theta_0}}\big[\nabla_\theta \log p_{\theta_0}(y \mid x)\, \nabla_\theta \log p_{\theta_0}(y \mid x)^\top\big].$$

We make three standard approximations, stated and discussed in Appendix C. We drop the cubic remainder (local quadraticity) and the cross-tensor blocks $F_{tt'}$, $t \neq t'$, so that the objective is additive over tensors (Dong et al., 2019; Yao et al., 2021). For the remaining blocks, the identity is exact: because W_t is applied at every token position, its score is a sum over positions, and for the format f assigned to t, $\Delta_t^\top F_{tt} \Delta_t = \mathrm{E}_x \mathrm{E}_{y\sim p_{\theta_0}}\big[\big(\sum_s g_s^\top \mathcal{E}_{t,f} h_s\big)^2\big]$ (Proposition 2). Dropping the products between different token positions, as is standard for weight-shared layers (Bolshim & Kugaevskikh, 2026), gives the per-tensor, per-format cost $D_{t,f} = \mathrm{E}_x \mathrm{E}_{y\sim p_{\theta_0}}\big[\sum_s (g_s^\top \mathcal{E}_{t,f}\, h_s)^2\big]$, and the objective $\sum_t \sum_f D_{t,f}\, z_{t,f}$, which is linear in the assignment.

Sampling: We estimate $D_{t,f}$ under teacher forcing on a calibration set constructed as a mixture of generations from the unquantized model and a general text corpus (WikiText). For a calibration sequence $w_{1:n}$, each prefix $w_{<r}$ acts as a context x in (1), and at each position r we draw a single label $\hat{y}_r \sim p_{\theta_0}(\cdot \mid w_{<r})$ at temperature one and without truncation, so that the labels follow the model distribution and the estimate targets the true rather than the empirical Fisher Martens & Grosse (2015); Kunstner et al. (2019). A single backward pass of $\ell = \sum_r \log p_{\theta_0}(\hat{y}_r \mid w_{<r})$ delivers $g_s = \partial \ell / \partial o_s$ at every position of every tensor, and over the N positions of the calibration set we estimate $\widehat{D}_{t,f} = \frac{1}{N}\sum_s (g_s^\top \mathcal{E}_{t,f}\, h_s)^2$. Since g_s accumulates the scores of all label positions, each squared term contains products between different labels; these vanish in expectation because the draws $\hat{y}_r$ are independent given the sequence and each score has zero mean by the first Bartlett identity, so $\widehat{D}_{t,f}$ is an unbiased estimate of $D_{t,f}$. Because g_s and h_s are evaluated at θ_0, they are shared by every format, and each tensor's backward hook evaluates all $|\mathcal{F}|$ candidates from them. The format axis therefore costs one matrix product per candidate rather than an additional pass through the model. Site-specific considerations are detailed in Appendix C.4.



(a) Bit allocation for Qwen3.5 35B A3B: varied σ (b) Objective against measured accuracy

Figure 4: (a) Average BPE of each tensor type of Qwen3.5 35B A3B over a storage sweep at its Q4_K_M decode budget; dashed lines mark the four matched storage budgets. (b) Objective against the six-benchmark accuracy drop from BF16 (Table 2) for every evaluated allocation.

Constraints and resultant program: Each tensor-format pair (t, f) carries a storage cost $\sigma_{t,f}$ and a latency cost $\tau_{t,f}$. Storage is exact: BPE of the realized block layout, including scales, zero points and any auxiliary table, times the parameter count, so a format is charged what it occupies. Latency comes from the kernel measurements of Section 3.4 at the tensor's own shape, multiplied by the number of calls to that tensor per decode step, and is assumed additive across tensors. Both enter as ε-constraints, so sweeping the budgets traces a frontier in which each point is optimal for its budget pair. The program is a binary integer linear program with one variable per pair,

$$\min_{z} \sum_{t,f} \widehat{D}_{t,f}\, z_{t,f} \quad \text{s.t.} \quad \sum_{t,f} \sigma_{t,f}\, z_{t,f} \le \bar{\sigma}, \quad \sum_{t,f} \tau_{t,f}\, z_{t,f} \le \bar{\tau}, \quad \sum_{f} z_{t,f} = 1 \;\; \forall t. \tag{2}$$

For these models this is a few thousand binaries, solved by branch-and-cut (HiGHS, zero final MIP gap) in seconds. Measurement prior to the solver requires one forward and one backward pass per-batch over the calibration set, with the per-format contractions evaluated within that pass, so the number of model passes is independent of $|\mathcal{F}|$. Algorithm 1 in the appendix states the procedure.

Fig. 4a sweeps $\bar{\sigma}$ for Qwen3.5 35B A3B at $\bar{\tau}$ of its Q4_K_M counterpart. The routed experts, which hold most parameters but are sparsely active, absorb nearly all of the reduction (3.5-5.3 bpe), while the output head, attention and shared-expert projections are raised in that order; the embedding is held low until the end. Since every viable candidate already decodes faster than the K-quants (Section 3.4), the decode constraint is inactive at K-quant budgets and binds only against Q4_0, at an 8-12% higher objective. At every budget the best uniform format carries a 2-9$\times$ higher objective (Appendix C.6.2). Fig. 4b shows that the objective orders the evaluated configurations by their accuracy drop, though not in proportion, which suffices for comparing candidates.

5 Evaluation

Models and tasks: We evaluate on two dense models: Qwen3.5 4B and Llama 3.1 8B (Instruct), and on one MoE model: Qwen3.5 35B A3B, spanning 4B to 35B parameters. Quantization quality is measured by three types of evaluation: (1) perplexity on WikiText-2 and C4; (2) non-reasoning accuracy on ARC-Challenge and HellaSwag; and (3) reasoning accuracy on MMLU-Redux, GSM8K, MATH-500, GPQA Diamond, and MBPP+. Efficiency is reported on the ET-SoC-1 in terms of storage required for model weights and decode throughput in tokens per second.

Implementation: The per-tensor objective $\Delta_t^\top F_t \Delta_t$ is estimated per-model from a calibration dataset comprising of self-generated and WikiText-2 sequences, and (2) solved under the constraints of the llama.cpp runtime (Appendix D.1). Allocation recipes paired with different community baselines in the storage-throughput-accuracy space are reported with the storage budget σ set by the baseline's size and the decode budget τ constrained by the baseline's decode time on the ET-SoC-1. The resulting recipes cost no more than the baseline in terms of storage and decode throughput by construction. Weights are quantized by round-to-nearest using the same importance matrix the community baseline was built with, so the calibration seen by the rounding step is identical between each recipe and its baseline. No other algorithmic preprocessing is applied to ensure fairness. We further ensure that all quantization recipes are evaluated through one pipeline with thinking disabled. Details of the metric and scoring mechanism used for each benchmark are reported in Appendix E.2.1.

Table 2: Storage, decode throughput (end to end on the ET-SoC-1, tok/s over 256 generated tokens), perplexity and mean accuracy. Each *Ours* row is solved within the storage and decode budgets of the baseline above it. Accuracy columns average Tables 20 and 21 (GSM8K flexible extract); the last column is the six-benchmark mean relative to BF16.

		Storage		Decode	Perplexity ↓		Accuracy (%) ↑		
Model	Configuration	GB	bpw	tok/s	Wiki2	C4	Non-reas.	Reas.	of BF16
Qwen3.5 4B	BF16	8.41	16.00	5.6	9.59	13.74	63.8	81.4	100
	Q3_K_M	2.29	4.36	9.0	**9.90**	**14.37**	62.0	**77.9**	96.1
	Ours	2.30	4.37	**15.0**	10.28	14.59	**63.6**	77.7	**96.5**
	Q4_K_M	2.74	5.21	9.4	9.77	**13.91**	**63.1**	79.4	97.8
	Ours	2.74	5.22	**13.4**	**9.76**	13.98	63.0	**81.0**	**99.3**
Llama 3.1 8B	BF16	16.06	16.00	3.6	7.22	11.39	67.5	60.1	100
	Q3_K_M	4.02	4.00	6.5	**7.69**	**12.28**	**66.3**	**56.9**	**95.8**
	Ours	4.02	4.00	**11.2**	8.01	12.78	65.7	55.8	94.2
	Q4_K_M	4.92	4.90	8.0	7.37	11.69	66.8	**59.7**	**99.2**
	Ours	4.92	4.90	**9.9**	**7.35**	**11.68**	**67.4**	59.4	99.1
Qwen3.5 35B A3B	BF16	69.32	16.00	–	6.59	10.68	72.4	89.5	100
	Q3_K_M	16.22	3.74	9.6	**6.74**	**10.92**	**71.2**	**89.1**	**99.2**
	Ours	16.22	3.74	**12.9**	6.77	11.05	71.1	88.5	98.7
	Q4_K_M	22.02	5.08	10.6	6.60	10.71	71.9	89.4	99.7
	Ours	22.01	5.08	**11.7**	**6.59**	10.71	**72.0**	**90.1**	**100.3**

Baselines: We compare against the two deployment options available to a llama.cpp user for this accelerator. The first is Q4_0, which with Q8_0 is the simplest and one of the fastest quantization formats the backend supports. It, however, is not the optimal baseline on this target as demonstrated in Table 19 in the Appendix. The second are the community K-quant recipes most widely deployed with llama.cpp, at three sizes (Q3_K_M, Q4_K_S, and Q4_K_M), taken from the importance-matrix-calibrated community releases of Han et al. (2023). K-quant presets assign formats by tensor role, through a hand-designed mixed-precision recipe, and are the natural reference for an allocator. Their storage is the published file size and their decode time is the kernel measurement of Section 3.4, validated through end-to-end quantized model runs on ET-SoC-1. The unquantized BF16 model is the accuracy reference. More details on the baseline evaluation are reported in Appendix D.

Results: Table 2 compares each recipe with the K-quant of equal storage at a 3-bit (Q3_K_M) and a 4-bit (Q4_K_M) budget; Appendix E.2.3 adds the Q4_0 and Q4_K_S budgets and Appendix E.2.2 each benchmark. Every recipe decodes faster than its 3-bit and 4-bit K-quant equivalent: $1.67\times$ and $1.43\times$ on Qwen3.5 4B, $1.72\times$ and $1.24\times$ on Llama 3.1 8B, and $1.34\times$ and $1.10\times$ on Qwen3.5 35B A3B respectively. The gain is largest at 3 bits, where the K-quant's two-level scales cost most to decode, and smallest on Qwen3.5 35B A3B, where a third of the decode step is non-matmul time (Appendix B.6). Accuracy stays within one point of the K-quant in five of the six pairs (-0.5 to $+1.5$ points of BF16); the exception is Llama 3.1 8B at 3 bits (-1.6), where the K-quant spends its extra decode time on a format with lower error per bit than any viable candidate (Section 3.4). Differences below about half a point are within the item-sampling noise of the benchmarks (Appendix E.2.1).

6 Conclusion

We treated post-training quantization as a full-stack problem on an accelerator without native sub-INT8 arithmetic, asking not which formats preserve accuracy but which of those also decode fast enough to raise throughput. Memory footprint follows from the representation and throughput from how it maps to the hardware datapath. On the ET-SoC-1, formats within a fraction of a bit of one another differ by $1.6\times$ in decode cost, several accuracy-competitive formats are not viable on throughput, and the ordering we measure differs from that reported on hardware with native low-precision units. Within the surviving set we allocate formats across tensors under separate storage and measured execution-time budgets, with timing coefficients taken from profiled kernels, and apply the algorithmic layer conditionally on the format fixed upstream. This yields on average more than 50% increase in throughput across dense and Mixture-of-Experts models from 4B to 35B parameters at similar accuracy and storage costs. The specific rankings hold for accelerators with comparable numerics; the methodology does not depend on that, since the mapping from format to achieved throughput is a property of the target and must be measured rather than assumed.

AI USE STATEMENT

In this work, we have used generative AI tools to support implementation of scripts to speed up our experiments and interpretation of data. We have not used generative AI tools to generate data or to produce measurements. Additionally, we used generative AI tools to draft and edit sections of the paper, to create figures, and to find and summarize related work. We have reviewed all AI-assisted work and take responsibility for the final content of this work, including the text, claims or artifacts produced with the aid of generative AI.

ETHICS STATEMENT

To the best of our knowledge, this work presents no notable ethical or societal-impact concerns.

REPRODUCIBILITY STATEMENT

The accelerator and the measured kernels are described in Section 3.4 and Appendix A; Appendices B.1–B.6 give the per-shape kernel timings, their decomposition, and the agreement between kernel-level and model-level decode time from which the solver's cost tables are built. The allocation program is stated in Eq. 2 and Algorithm 1, and Appendix C gives its assumptions and proofs, the treatment of the head, routed experts and embedding (Appendix C.4), and the implementation and solver configuration (Appendix D.1). The baselines are given in Appendix D, and the benchmarks, harness settings and per-benchmark results in Appendix E.2.1. A worked example of our kernel optimization is given in Appendix F.

REFERENCES

Muhammad Ahmad, Khurram Mazher, Saqib Akram, Ahmad Tameem, and Saad Bin Nasir. QuantX: A framework for hardware-aware quantization of generative AI workloads, 2025. URL https://arxiv.org/abs/2505.07531.

AI Foundry. ET Platform: ET accelerator firmware and runtime. https://github.com/aifoundry-org/et-platform, 2026. GitHub repository. Accessed September 14, 2026.

Bartowski. Llama.cpp imatrix GGUF quantizations. https://huggingface.co/bartowski, 2026. Hugging Face model repositories. Accessed September 26, 2026.

Peter Belcak, Greg Heinrich, Shizhe Diao, Yonggan Fu, Xin Dong, Saurav Muralidharan, Yingyan Celine Lin, and Pavlo Molchanov. Small language models are the future of agentic AI, 2025. URL https://arxiv.org/abs/2506.02153.

Stella Biderman, Hailey Schoelkopf, Lintang Sutawika, Leo Gao, Jonathan Tow, Baber Abbasi, Alham Fikri Aji, Pawan Sasanka Ammanamanchi, Sidney Black, Jordan Clive, Anthony DiPofi, Julen Etxaniz, Benjamin Fattori, Jessica Zosa Forde, Charles Foster, Jeffrey Hsu, Mimansa Jaiswal, Wilson Y. Lee, Haonan Li, Charles Lovering, Niklas Muennighoff, Ellie Pavlick, Jason Phang, Aviya Skowron, Samson Tan, Xiangru Tang, Kevin A. Wang, Genta Indra Winata, François Yvon, and Andy Zou. Lessons from the trenches on reproducible evaluation of language models, 2024. URL https://arxiv.org/abs/2405.14782.

Maxim Bolshim and Alexander Kugaevskikh. Stochastic estimation of the layer-wise Hessian trace for monitoring neural-network training, 2026. URL https://arxiv.org/abs/2605.25674.

Yuanteng Chen, Yuantian Shao, Peisong Wang, and Jian Cheng. EAC-MoE: Expert-selection aware compressor for mixture-of-experts large language models. In *Proceedings of the 63rd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 12942–12963, Vienna, Austria, July 2025a. Association for Computational Linguistics. doi: 10.18653/v1/2025.acl-long.633. URL https://aclanthology.org/2025.acl-long.633/.

Zhixuan Chen, Xing Hu, Dawei Yang, Zukang Xu, Chen Xu, Zhihang Yuan, Sifan Zhou, and Jiangyong Yu. MoEQuant: Enhancing quantization for mixture-of-experts large language models via

expert-balanced sampling and affinity guidance. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*. PMLR, 2025b. URL https://openreview.net/forum?id=0epuNvt5Dj.

Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zeng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. Hexagon DSP: An architecture optimized for mobile multimedia and communications. *IEEE Micro*, 34(2):34–43, 2014. doi: 10.1109/MM.2014.12.

David R. Ditzel. Accelerating ML recommendation with over 1,000 RISC-V/tensor processors on Esperanto's ET-SoC-1 chip. *IEEE Micro*, 42(3):31–38, 2022. doi: 10.1109/MM.2022.3140674.

Zhen Dong, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. HAWQ: Hessian aware quantization of neural networks with mixed-precision. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 293–302, 2019. URL https://openaccess.thecvf.com/content_ICCV_2019/html/Dong_HAWQ_Hessian_AWare_Quantization_of_Neural_Networks_With_Mixed-Precision_ICCV_2019_paper.html.

Zhen Dong, Zhewei Yao, Daiyaan Arfeen, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. HAWQ-V2: Hessian aware trace-weighted quantization of neural networks. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 33, pp. 18518–18529, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/d77c703536718b95308130ff2e5cf9ee-Abstract.html.

Haojie Duanmu, Xiuhong Li, Zhihang Yuan, Size Zheng, Jiangfei Duan, Xingcheng Zhang, and Dahua Lin. MxMoE: Mixed-precision quantization for MoE with accuracy and performance co-design. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*, pp. 14793–14806. PMLR, 2025. URL https://proceedings.mlr.press/v267/duanmu25a.html.

Runa Eschenhagen, Alexander Immer, Richard E. Turner, Frank Schneider, and Philipp Hennig. Kronecker-factored approximate curvature for modern neural network architectures. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 36, pp. 33624–33655, 2023. URL https://proceedings.neurips.cc/paper_files/paper/2023/file/6a6679e3d5b9f7d5f09cdb79a5fc3fd8-Paper-Conference.pdf.

Amin Firoozshahian, Joel Coburn, Roman Levenstein, Rakesh Nattoji, Ashwin Kamath, et al. MTIA: First generation silicon targeting Meta's recommendation systems. In *Proceedings of the 50th Annual International Symposium on Computer Architecture (ISCA)*. ACM, 2023. doi: 10.1145/3579371.3589348.

Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. OPTQ: Accurate quantization for generative pre-trained transformers. In *International Conference on Learning Representations (ICLR)*, 2023. URL https://openreview.net/forum?id=tcbBPnfwxS.

Zhongqian Fu, Ning Ding, Kai Han, Xianzhi Yu, Xiaosong Li, Xinghao Chen, Yehui Tang, and Yunhe Wang. EAQuant: Enhancing post-training quantization for MoE models via expert-aware optimization, 2025. URL https://arxiv.org/abs/2506.13329.

Georgi Gerganov. Llama.cpp: LLM inference in C/C++. https://github.com/ggml-org/llama.cpp, 2023. GitHub repository.

Roger Grosse and James Martens. A Kronecker-factored approximate Fisher matrix for convolution layers. In *Proceedings of the 33rd International Conference on Machine Learning (ICML)*, volume 48 of *Proceedings of Machine Learning Research*, pp. 573–582. PMLR, 2016. URL https://proceedings.mlr.press/v48/grosse16.html.

Ziyi Guan, Hantao Huang, Yupeng Su, Hong Huang, Ngai Wong, and Hao Yu. APTQ: Attention-aware post-training mixed-precision quantization for large language models. In *Proceedings of the 61st ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6. ACM, 2024. doi: 10.1145/3649329.3658498.

Daniel Han, Michael Han, and Unsloth team. Unsloth. https://github.com/unslothai/unsloth, 2023. GitHub repository.

Babak Hassibi, David G. Stork, and Gregory J. Wolff. Optimal brain surgeon and general network pruning. In *Proceedings of the IEEE International Conference on Neural Networks*, volume 1, pp. 293–299, 1993. doi: 10.1109/ICNN.1993.298572.

Wei Huang, Haotong Qin, Yangdong Liu, Yawei Li, Qinshuo Liu, Xianglong Liu, Luca Benini, Michele Magno, Shiming Zhang, and Xiaojuan Qi. SliM-LLM: Salience-driven mixed-precision quantization for large language models. In *Proceedings of the 42nd International Conference on Machine Learning (ICML)*, volume 267 of *Proceedings of Machine Learning Research*. PMLR, 2025. URL https://proceedings.mlr.press/v267/huang25aa.html.

Frederik Kunstner, Philipp Hennig, and Lukas Balles. Limitations of the empirical Fisher approximation for natural gradient descent. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 32, pp. 4158–4169, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/46a558d97954d0692411c861cf78ef79-Abstract.html.

Yann LeCun, John S. Denker, and Sara A. Solla. Optimal brain damage. In David S. Touretzky (ed.), *Advances in Neural Information Processing Systems (NIPS)*, volume 2, pp. 598–605. Morgan-Kaufmann, 1989. URL https://proceedings.neurips.cc/paper_files/paper/1989/file/6c9882bbac1c7093bd25041881277658-Paper.pdf.

Shiyao Li, Xuefei Ning, Ke Hong, Tengxuan Liu, Luning Wang, Xiuhong Li, Kai Zhong, Guohao Dai, Huazhong Yang, and Yu Wang. LLM-MQ: Mixed-precision quantization for efficient LLM deployment. In *NeurIPS Workshop on Efficient Natural Language and Speech Processing (ENLSP)*, 2023. URL https://neurips.cc/virtual/2023/81141.

Yuhang Li, Ruihao Gong, Xu Tan, Yang Yang, Peng Hu, Qi Zhang, Fengwei Yu, Wei Wang, and Shi Gu. BRECQ: Pushing the limit of post-training quantization by block reconstruction. In *International Conference on Learning Representations (ICLR)*, 2021. URL https://openreview.net/forum?id=POWv6hDd9XH.

Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. QServe: W4A8KV4 quantization and system co-design for efficient LLM serving. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 7, 2025. URL https://proceedings.mlsys.org/paper_files/paper/2025/hash/fbe2b2f74a2ece8070d8fb073717bda6-Abstract-Conference.html.

James Martens and Roger Grosse. Optimizing neural networks with Kronecker-factored approximate curvature. In *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, volume 37 of *Proceedings of Machine Learning Research*, pp. 2408–2417. PMLR, 2015. URL https://proceedings.mlr.press/v37/martens15.html.

Mercedes-Benz Group AG. Next generation in-car intelligence: Mercedes-Benz and Liquid AI. https://group.mercedes-benz.com/technology/innovation/collaboration/liquid-ai.html, 2026. Accessed September 16, 2026.

Markus Nagel, Rana Ali Amjad, Mart Van Baalen, Christos Louizos, and Tijmen Blankevoort. Up or down? Adaptive rounding for post-training quantization. In *Proceedings of the 37th International Conference on Machine Learning (ICML)*, volume 119 of *Proceedings of Machine Learning Research*, pp. 7197–7206. PMLR, 2020.

Kacper Podbucki and Bartłomiej Szalwach. Cost-effective edge AI: Hailo-8 powered Raspberry Pi vs. NVIDIA Jetson AGX Orin in airport infrastructure monitoring. *Electronics*, 15(17):3774, 2026. doi: 10.3390/electronics15173774.

Robert Bosch GmbH. AI-powered cockpit. https://www.bosch-mobility.com/en/solutions/vehicle-computer/ai-powered-cockpit/, 2026. Accessed September 16, 2026.

Bita Darvish Rouhani, Nitin Garegrat, Tom Savell, Ankit More, Kyung-Nam Han, Ritchie Zhao, Mathew Hall, Jasmine Klar, Eric Chung, Yuan Yu, Michael Schulte, Ralph Wittig, Ian Bratt, Nigel Stephens, Jelena Milanovic, John Brothers, Pradeep Dubey, Marius Cornea, Alexander Heinecke, Andres Rodriguez, Martin Langhammer, Summer Deng, Maxim Naumov, Paulius Micikevicius, and Michael Siu. OCP microscaling formats (MX) specification version 1.0. Technical report, Open Compute Project Foundation, September 2023. URL https://www.opencompute.org/documents/ocp-microscaling-formats-mx-v1-0-spec-final-pdf.

Ahmed Sharshar, Latif U. Khan, Waseem Ullah, and Mohsen Guizani. Vision-language models for edge networks: A comprehensive survey. *IEEE Internet of Things Journal*, 12(16):32701–32724, August 2025. doi: 10.1109/JIOT.2025.3579032.

Tenstorrent. Wormhole PCIe cards: Hardware specifications. https://docs.tenstorrent.com/aibs/wormhole/index.html, 2026. Product documentation. Accessed September 20, 2026.

Xubin Wang, Qing Li, and Weijia Jia. Cognitive edge computing: A comprehensive survey on optimizing large models and AI agents for pervasive deployment, 2025a. URL https://arxiv.org/abs/2501.03265.

Xubin Wang, Zhiqing Tang, Jianxiong Guo, Tianhui Meng, Chenhao Wang, Tian Wang, and Weijia Jia. Empowering edge intelligence: A comprehensive survey on on-device AI models. *ACM Computing Surveys*, 57(9), April 2025b. doi: 10.1145/3724420.

Haojun Xia, Zhen Zheng, Xiaoxia Wu, Shiyang Chen, Zhewei Yao, Stephen Youn, Arash Bakhtiari, Michael Wyatt, Donglin Zhuang, Zhongzhu Zhou, Olatunji Ruwase, Yuxiong He, and Shuaiwen Leon Song. Quant-LLM: Accelerating the serving of large language models via FP6-centric algorithm-system co-design on modern GPUs. In *Proceedings of the 2024 USENIX Annual Technical Conference (USENIX ATC)*, pp. 699–713. USENIX Association, 2024. URL https://www.usenix.org/conference/atc24/presentation/xia.

Zhewei Yao, Zhen Dong, Zhangcheng Zheng, Amir Gholami, Jiali Yu, Eric Tan, Leyuan Wang, Qijing Huang, Yida Wang, Michael W. Mahoney, and Kurt Keutzer. HAWQ-V3: Dyadic neural network quantization. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*, volume 139 of *Proceedings of Machine Learning Research*, pp. 11875–11886. PMLR, 2021. URL https://proceedings.mlr.press/v139/yao21a.html.

Genghan Zhang, Shaowei Zhu, Anjiang Wei, Zhenyu Song, Allen Nie, Zhen Jia, Nandita Vijaykumar, Yida Wang, and Kunle Olukotun. AccelOpt: A self-improving LLM agentic system for AI accelerator kernel optimization. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 8, 2026. URL https://proceedings.mlsys.org/paper_files/paper/2026/hash/0f8426558905746fc38da5e335700aec-Abstract-Conference.html.

Zhen Zheng, Xiaonan Song, and Chuanjie Liu. MixLLM: LLM quantization with global mixed-precision between output-features and highly-efficient system design. In *Proceedings of Machine Learning and Systems (MLSys)*, volume 8, 2026. URL https://openreview.net/forum?id=VBbMRQ4VOc.

APPENDIX CONTENTS

A Hardware Details **15**

A.1 ET-SoC-1 15

B Kernel Level Measurements **16**

B.1 Prefill & decode 16

B.2 Per-shape kernel measurements 16

B.3 Dequantization cost decomposition 17

B.4 Activation quantization 18

B.5 Sweeps over quantization format dimensions 19

B.6 Kernel-to-model level measurement agreement 23

C Mixed Precision Allocator assumptions and derivations **24**

C.1 Notation 24

C.2 Assumptions 24

C.3 Derivations 26

C.4 Non-standard sites 27

C.5 Procedure and implementation 29

C.6 Allocations 29

D Baselines **30**

D.1 Implementation details 30

E Evaluation Benchmarks **32**

E.1 Performance Measurements 32

E.2 Accuracy Measurements 33

F Worked example: optimizing the Q4_0 decode kernel **33**

F.1 Scalar device kernel 34

F.2 Structure of the optimized kernel 35

F.3 Optimizations 36

A HARDWARE DETAILS

A.1 ET-SOC-1

The Esperanto Technologies ET-SoC-1 is a high performance and low power RISC-V accelerator developed by Esperanto Technologies. It is supported by AI Foundry's open-source ET platform publicly available under Apache License 2.0 (AI Foundry, 2026). It is a power-efficient accelerator which has 1088 minion RISC-V cores each with its own scalar, vector and tensor unit. Each minion core is an in-order core supporting two hardware threads and a local data cache. Two threads allow overlapping compute and communication while part of the cache can be software controlled allowing more programmer control. Each vector processing unit is 8 lanes wide with 32 bits per lane and supports both floating and integer operations. There are many custom instructions that programmers need to understand to unlock the full potential of the hardware. Each minion comes with an asynchronous tensor core that is software controlled. It allows fused multiply add operations over FP32, FP16 and INT8 numeric types.

Figure 5 shows how the cores are organised. 8 minions make a neighbourhood which boasts an instruction cache and cooperative load instruction serving all minions. 4 neighbourhoods make a shire. Each shire has a 4 MB fast memory which can be programmed as a cache or a software controlled local memory. It also has the message passing infrastructure to communicate and cooperate with other shires over a network-on-chip. There are a total of 34 shires on one chip giving a total of 1088 minions and 136 MB of fast local memory. 32 shires serve as the compute array while others serve as firmware and service processors. There are eight separate memory shires that interface with a total of 32GB LPDDR4X DRAM. The accelerator is deployed as a single PCIe card (Figure 6), which runs every measurement in this paper.

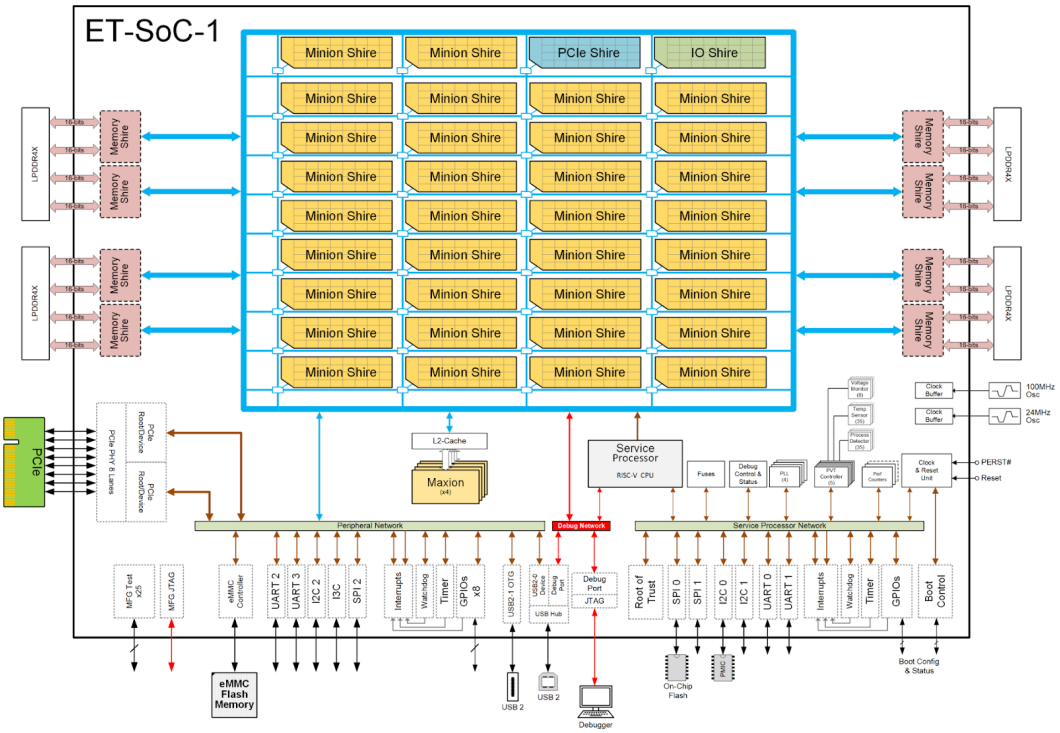


Figure 5: ET-SoC-1 architecture: 1088 minion cores grouped into neighbourhoods and shires, connected by a network-on-chip.

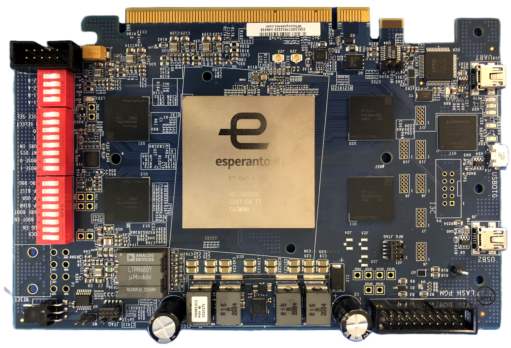


Figure 6: The ET-SoC-1 PCIe card used for all measurements.

Table 3: Measured Q4_0 cost per tensor for Qwen3.5 35B A3B, with the call count each tensor carries in one step and its contribution to the weight path. The router is held at FP32 and is not part of the weight path.

			Prefill		Decode	
Tensor	Shape	Calls	ms/call	Total	ms/call	Total
Dense path						
qkv	8192×2048	40	4.420	176.8	0.169	6.76
gate	4096×2048	30	2.501	75.0	0.106	3.18
attn_out, lin_out	2048×4096	40	2.562	102.5	0.107	4.28
gate, up shared expert	512×2048	100	0.500	50.0	0.041	4.10
shared expert down	2048×512	40	0.460	18.4	0.041	1.64
SSM α, β	32×2048	60	0.337	20.2	0.040	2.40
lm_head	248320×2048	1	172.838	172.8	4.224	4.22
MoE path (top-8 of 256 experts)						
expert gate, up	512×2048	80	18.902	1512.2	0.125	10.00
expert down	2048×512	40	16.042	641.7	0.132	5.28
Weight path				2769.6		41.86

B Kernel Level Measurements

B.1 Prefill & decode

The allocation of Section 4 constrains decode only. Tables 3, 4, and 5 give the Q4_0 cost of every tensor in both phases, and Table 6 gives each format's cost relative to Q4_0 in both. The two phases differ in how much the format matters. Prefill runs on the tensor units, which consume a dequantized tile, and its dequantization is paid once per weight tile and amortized over the 512 tokens of the prompt: on the dense path every format lies within 0.96–$1.25\times$ of Q4_0, and all but Q6_K lie within 0.8–$1.2\times$ on the representative shapes of Table 7. Decode runs on the vector units, pays dequantization once per token, and separates the same formats by 2.7–$3.1\times$ on a single shape. The format choice is therefore a decode decision, and in the single-user regime decode is also where the time goes: a response of a few hundred tokens spends far longer in decode than in its one prefill. The routed-expert prefill is the exception, varying more with the format than the dense path does: the 5-bit formats cost 1.38–$1.47\times$ Q4_0 there, against 1.04–$1.15\times$ in expert decode. A prefill-constrained allocation would therefore lean further toward 4-bit experts than the decode-constrained one does, and is left to future work.

B.2 Per-shape kernel measurements

Tables 7 and 8 give the per-call kernel time on a representative subset of the captured shapes: one large body projection of each dense model, the largest output head, and the two routed-expert projections of Qwen3.5 35B A3B. The rows are the formats of Table 6, which aggregates every captured

Table 4: Measured Q4_0 cost per tensor for Qwen3.5 4B, with the call count each tensor carries in one step and its contribution to the weight path.

			Prefill		Decode	
Tensor	Shape	Calls	ms/call	Total	ms/call	Total
Dense path						
qkv	8192×2560	32	5.230	167.4	0.206	6.59
gate	4096×2560	24	2.946	70.7	0.120	2.88
attn_out, lin_out	2560×4096	32	4.913	157.2	0.147	4.70
kv	1024×2560	16	0.946	15.1	0.052	0.83
MLP gate, up	9216×2560	64	11.511	736.7	0.222	14.21
MLP down	2560×9216	32	11.273	360.7	0.227	7.26
lm_head	248320×2560	1	198.949	198.9	5.079	5.08
Weight path				1706.8		41.56

Table 5: Measured Q4_0 cost per tensor for Llama 3.1 8B, with the call count each tensor carries in one step and its contribution to the weight path.

			Prefill		Decode	
Tensor	Shape	Calls	ms/call	Total	ms/call	Total
Dense path						
q, attn_out	4096×4096	64	4.497	287.8	0.185	11.84
kv	1024×4096	64	1.535	98.2	0.065	4.16
MLP gate, up	14336×4096	64	22.295	1426.9	0.500	32.00
MLP down	4096×14336	32	15.031	481.0	0.493	15.78
lm_head	128256×4096	1	192.820	192.8	4.328	4.33
Weight path				2486.7		68.10

shape of every model by call count. Three regularities carry over from these shapes to the aggregate. Prefill varies little with the format: every format but Q6_K lies within 0.8–$1.2\times$ of Q4_0 on these shapes, since the tensor units consume a dequantized tile and the dequantization is amortized over the 512 tokens of the prompt. Decode separates the formats by 2.7–$3.1\times$ on the same shape, and the order is stable across models and shapes: the integer formats with an FP16, FP8 or E8M0 block scale are cheapest at every width, and the K-quants, the 8-bit formats and QFP8_0 are the most expensive.

B.3 Dequantization cost decomposition

Tables 9 and 10 report the decode kernel phase by phase, which is what allows a difference between formats to be attributed to a phase rather than reported only as a total (Fig. 2 plots the dense fetch and dequantization columns). Activation staging ($22.9\,\mu$s) and the FP32 compute (30–31 μs) are the same for every format, so the two format-dependent phases are fetch and dequantization, and they behave in opposite ways. Fetch is a pure bandwidth transaction: it is linear in bits per element ($13.8\,\mu$s per bit, $R^2 > 0.999$) and formats of equal width fetch in equal time whatever their layout. Dequantization does not follow width at all: it spans $52.7\,\mu$s (Q4_0) to $141.7\,\mu$s (Q3_K) and is set by the per-block work. Two-level scales are the most expensive since each sub-block scale must itself be decoded before it can scale its elements: Q2_K, Q3_K, Q5_K and Q6_K are the four costliest formats to dequantize (97–142 μs) at widths from 2.6 to 6.6 bits. The lookup formats follow (QFP8_0, IQ4_XS and Q3_CB8 at 81–85 μs), because neither FP8 nor a non-uniform grid has a native path and each element is reconstructed through a table.

The overlap column models the kernel as staging plus the larger of fetch and dequantization plus compute, i.e. it assumes the hardware hides the shorter of the two. The model is within 16% of the measurement for the 4- and 5-bit integer formats. It overestimates the cheapest low-bit formats (Q3_0, Q3_1 and MXFP4, by 29–38%), and underestimates the 8-bit formats and Q4_K–Q6_K (ratios 0.77–0.93), where fetch and dequantization are both long. The discrepancies can be attributed

Table 6: Measured kernel cost per format, relative to Q4_0 on the same path, averaged over models. Each model's figure weights its per-shape kernel times by the call counts of Tables 4–3, giving the per-step weight-path cost of assigning that format to every tensor on that path; the average is geometric, since the figures are ratios. Per-model Q4_0 references are given in those tables. The MoE columns are Qwen3.5 35B A3B's own figure. Below $1.00\times$ is faster than Q4_0.

Format	bpe	Dense prefill (×)	Dense decode (×)	MoE prefill (×)	MoE decode (×)
Q3_0	3.5	1.05	**0.88**	1.00	1.00
Q4_0	4.5	1.00	1.00	1.00	1.00
Q5_0	5.5	**0.98**	1.22	1.47	1.14
Q8_0	8.5	1.03	1.76	1.10	1.38
Q3_1	3.75	**0.99**	**0.86**	1.01	**0.94**
Q4_1	4.75	1.00	1.09	1.04	**0.95**
Q5_1	5.75	**0.98**	1.35	1.40	1.04
Q4_0_FP8	4.25	**0.99**	**0.93**	1.02	**0.91**
Q4_1_FP8	4.5	1.00	1.05	1.03	1.02
Q5_0_FP8	5.25	**0.98**	1.15	1.44	1.15
Q5_1_FP8	5.5	**0.98**	1.28	1.38	1.09
Q8_0_FP8	8.25	**0.98**	1.75	1.09	1.33
QFP4_0	4.5	1.07	1.04	1.45	**0.97**
QFP8_0	8.5	1.16	2.01	1.69	1.86
MXINT4	4.25	**0.99**	**0.93**	1.01	**0.92**
MXFP4	4.25	1.06	1.00	1.38	**0.91**
MXINT8	8.25	**0.98**	1.74	1.08	1.32
NVFP4	4.5	1.08	1.37	1.52	1.31
Q2_K	2.625	1.19	1.47	1.95	1.37
Q3_K	3.4375	1.21	2.17	1.84	1.74
Q4_K	4.5	**0.97**	1.47	1.17	1.55
Q5_K	5.5	1.10	2.24	1.56	2.05
Q6_K	6.5625	1.25	2.06	2.05	1.69
Q3_CB8	3.75	**0.98**	1.18	1.08	1.35
Q4_CB16	4.75	**0.99**	1.25	1.07	1.06
IQ4_XS	4.25	**0.96**	1.37	1.13	1.20
IQ4_NL	4.5	**0.97**	1.12	1.10	1.05

to the underlying hardware scheduling mode and the nature of profiling individual components separately.

On the routed-expert kernel the same split holds with a compressed spread: per-expert calls move little data, so fixed costs weigh more, and dequantization ranges over 47–103 μs against a fetch that again follows width. One asymmetry there is a kernel artefact rather than a property of the format: the asymmetric Q5_1 expert kernel carries an optimization (each thread reuses the activation row across two weight rows) that the symmetric Q5_0 kernel does not yet have, which is why Q5_1 decodes faster than Q5_0 on the experts and not on the dense path.

B.4 Activation quantization

Every recipe in this paper quantizes weights only. Table 11 gives the rationale for this choice: with Q4_0 weights held fixed, it varies the activation operand of the decode matrix-vector kernel and reports the per-step weight-path time of each model relative to FP32 activations.

FP16 activations are slower on every model, by 3–32%. The vector units have no FP16×FP16 multiply, so the FP16 activation is promoted to FP32 in the inner loop while the Q4_0 weight is still dequantized to FP32; the narrower operand removes no work and adds a conversion. INT8 activations reach the integer path, and a Q8_0 activation that is already quantized saves up to 7% of the weight path on Qwen3.5 35B A3B. It is not a saving in practice: the activation is produced in

Table 7: Per-call kernel time (ms) on representative dense shapes, prefill at $N = 512$ and single-token decode. One large body projection per model and the largest output head.

	Prefill				Decode			
Model	Qwen3.5 4B	Llama 3.1 8B	Qwen3.5 35B A3B	Qwen3.5 35B A3B	Qwen3.5 4B	Llama 3.1 8B	Qwen3.5 35B A3B	Qwen3.5 35B A3B
Shape	9216×2560	14336×4096	8192×2048	248320×2048	9216×2560	14336×4096	8192×2048	248320×2048
Role	MLP gate, up	MLP gate, up	qkv	lm_head	MLP gate, up	MLP gate, up	qkv	lm_head
Q3_0	13.120	20.804	4.530	154.9	0.203	0.426	0.148	3.524
Q4_0	11.511	22.295	4.420	172.8	0.222	0.500	0.169	4.224
Q5_0	11.554	20.919	4.426	145.4	0.277	0.660	0.201	5.098
Q8_0	12.001	22.939	4.425	187.2	0.390	1.047	0.303	8.031
Q3_1	11.777	21.783	4.484	162.0	0.198	0.414	0.144	3.447
Q4_1	11.617	21.647	4.418	173.5	0.245	0.613	0.179	4.412
Q5_1	11.644	20.986	4.497	146.1	0.297	0.781	0.216	5.505
Q4_0_FP8	11.511	21.864	4.449	166.8	0.210	0.477	0.156	3.837
Q4_1_FP8	11.453	22.305	4.411	172.4	0.224	0.628	0.170	4.091
Q5_0_FP8	11.636	20.933	4.414	144.9	0.255	0.638	0.187	4.897
Q5_1_FP8	11.434	21.117	4.420	146.5	0.280	0.732	0.203	5.143
Q8_0_FP8	11.179	21.270	4.432	156.1	0.411	1.026	0.292	7.798
QFP4_0	13.863	22.348	4.417	157.5	0.233	0.564	0.165	4.121
QFP8_0	15.467	24.289	4.671	181.0	0.462	1.211	0.320	8.915
BFP4	11.549	21.854	4.458	166.9	0.206	0.483	0.156	3.817
MXFP4	13.534	22.743	4.414	152.9	0.221	0.561	0.162	4.115
BFP8	11.207	21.279	4.442	156.4	0.415	1.027	0.286	7.757
NVFP4	13.962	22.565	4.462	159.5	0.310	0.660	0.233	5.785
Q2_K	15.495	24.228	4.917	176.3	0.361	0.780	0.242	6.423
Q3_K	15.905	24.981	4.967	183.2	0.542	1.140	0.365	9.858
Q4_K	11.243	21.083	4.361	145.5	0.358	0.684	0.266	6.562
Q5_K	14.452	22.284	4.632	162.5	0.571	1.047	0.395	10.639
Q6_K	16.477	26.192	5.036	192.6	0.508	1.095	0.344	9.180
Q3_CB8	11.151	21.420	4.397	160.7	0.287	0.614	0.198	5.231
Q4_CB16	11.421	21.121	4.446	164.6	0.301	0.649	0.207	5.581
IQ4_XS	11.232	21.171	4.431	136.6	0.327	0.743	0.227	5.985
IQ4_NL	11.265	21.201	4.435	142.9	0.254	0.614	0.180	4.543

FP32, and once its conversion to Q8_0 is counted the kernel is 3–12% slower than FP32 on every model. Part of the reason is the multiply-accumulate itself: FP32 has a fused multiply-add (one operation, 32.4 μs per 4096×2048 chain), while the INT32 path issues a separate multiply and add (two operations, 37.0 μs), so the integer datapath loses on compute and also pays the FP32-to-Q8_0 conversion.

Prefill behaves differently. The wide matrix multiplications run on the tensor units, which accept FP16, and FP16 activations roughly halve the prefill weight path (0.51–$0.53\times$ FP32 across the three models, conversion excluded). This paper constrains decode only (Section B.1), so the result is recorded here rather than used; a prefill-constrained allocation would take FP16 activations for its wide shapes. The Q8_0 prefill kernel is not yet optimized and is not reported. Consequently, Q8_0 activation quantization during prefill only is not considered.

B.5 Sweeps over quantization format dimensions

This section reports the effect of each design dimension with the others held fixed, so that a performance difference can be attributed to a dimension rather than to a format taken as a whole. Table 14 varies element dtype at fixed scale dtype and group size; Table 13 varies scale dtype at fixed element dtype and group size, including the power-of-two case; Table 12 varies group size at a fixed format, and reports storage alongside throughput because the two move together along this dimension. Where an entry is not available the corresponding kernel does not yet exist, which is recorded rather than omitted.

B.5.1 Group Size

Error grows at a nearly constant rate, rising 18–31% with each doubling of g. This is expected: the block scale is set by the largest magnitude in the group, and larger groups tend to contain larger outliers. Decode time behaves differently. Going from $g = 16$ to $g = 32$ halves the number of scales and removes 21% of the routed-expert weight path (where $g = 16$ was captured) and an estimated 6–9% of the dense path. Past $g = 32$, however, the scales are already a small share of both the bytes fetched and the per-block work, so each further doubling buys only 0–7%, and the entire range from 32 to 256 is worth just 4–10%.

Table 8: Per-call kernel time (ms) on the routed-expert shapes of Qwen3.5 35B A3B (top-8 of 256 experts per call), prefill at $N = 512$ and single-token decode.

Format	Prefill		Decode	
	512×2048 expert gate, up	2048×512 expert down	512×2048 expert gate, up	2048×512 expert down
Q3_0	18.982	16.013	0.124	0.135
Q4_0	18.902	16.042	0.125	0.132
Q5_0	27.435	24.395	0.150	0.135
Q8_0	20.628	17.817	0.168	0.191
Q3_1	19.049	16.083	0.116	0.126
Q4_1	19.727	16.789	0.122	0.120
Q5_1	26.001	23.122	0.133	0.131
Q4_0_FP8	19.313	16.444	0.114	0.119
Q4_1_FP8	19.362	16.516	0.125	0.138
Q5_0_FP8	26.724	23.959	0.150	0.139
Q5_1_FP8	25.728	22.926	0.138	0.141
Q8_0_FP8	20.414	17.618	0.164	0.179
QFP4_0	26.823	24.186	0.118	0.136
QFP8_0	31.227	28.474	0.237	0.237
BFP4	19.153	16.287	0.116	0.118
MXFP4	25.882	22.608	0.113	0.122
BFP8	20.234	17.465	0.162	0.179
NVFP4	28.193	25.406	0.166	0.169
Q2_K	35.866	33.160	0.185	0.154
Q3_K	34.043	31.095	0.232	0.200
Q4_K	21.940	19.137	0.212	0.167
Q5_K	28.976	26.217	0.270	0.243
Q6_K	38.164	34.087	0.226	0.192
Q3_CB8	20.031	17.949	0.187	0.143
Q4_CB16	19.844	17.757	0.132	0.142
IQ4_XS	21.201	18.331	0.156	0.148
IQ4_NL	20.609	17.745	0.132	0.139

The natural stopping point is where one more doubling costs more in relative error than it returns in storage and speed combined, and here that point is $g = 32$. Stepping into it saves 0.5 bits per element (a tenth of the $g = 16$ file), 6–9% of dense decode time, and 21% of the expert path, in exchange for a 31% increase in error. Every step beyond it saves at most 0.25 bits per element (5.6%) and at most 7% of decode time, while error rises another 21–25%. Taken together, the whole range from $g = 32$ to $g = 256$ recovers only 0.44 bits per element. On the dense models, storage tips the balance, since the time saved by moving from $g = 16$ to $g = 32$ is no larger than what the later steps save, but the storage savings are immense. Conveniently, 32 is also the block size of most llama.cpp native types in the candidate set, so the choice is natural from a kernel support perspective as well.

B.5.2 Scale Datatype and Hierarchy

Table 13 holds the element fixed and changes only the block scale, at three element types. With INT4 elements the three scales run at the same speed within 7%: an FP8 or E8M0 scale is decoded once per block of 32, so its conversion cost is amortized, and both narrower scales save 0.25 bits per element over FP16 and are 7–9% faster on decode. They differ sharply in error. The FP8 scale with its FP32 outer scale reproduces the FP16-scale error to within 1%, because E4M3 keeps three mantissa bits of the block's range and the outer scale restores the dynamic range the narrow exponent lacks. The E8M0 scale is a power of two, so the block maximum is rounded to the next power of two and up to half of the INT4 grid goes unused; its error is 75% above Q4_0. At 4 bits the FP8

Table 9: Decode kernel phases on a dense 4096×2048 matrix-vector product (μs). Fetch streams the quantized weight from DRAM; dequantization reconstructs FP32 weights from the staged block; compute is the FP32 multiply-accumulate (30–31 μs, identical for every format) and activation staging adds 22.9 μs. Each phase is profiled in isolation. *Overlap* models the kernel as staging $+ \max(\text{fetch},\ \text{dequant} + \text{compute})$; *measured* is the end-to-end kernel.

Format	bpe	Fetch	Dequant	Raw sum	Overlap	Measured	Overlap / meas.
Q3_0	3.5	58.3	71.9	184	125	91	1.38
Q4_0	4.5	73.0	52.7	179	106	106	1.00
Q5_0	5.5	86.0	73.6	213	127	125	1.02
Q8_0	8.5	128.1	61.9	243	151	176	0.86
Q3_1	3.75	61.5	62.7	178	116	88	1.32
Q4_1	4.75	76.5	66.1	196	120	107	1.12
Q5_1	5.75	90.5	78.5	222	132	134	0.98
Q4_0_FP8	4.25	68.8	55.3	178	109	95	1.14
Q4_1_FP8	4.5	72.8	54.8	181	108	99	1.09
Q5_0_FP8	5.25	83.2	65.7	202	119	111	1.07
Q5_1_FP8	5.5	85.8	66.8	206	120	125	0.96
Q8_0_FP8	8.25	124.2	63.6	241	147	175	0.84
QFP4_0	4.5	72.7	65.9	192	119	98	1.22
QFP8_0	8.5	128.1	85.4	267	151	179	0.84
BFP4	4.25	69.0	53.9	176	107	96	1.12
MXFP4	4.25	68.8	66.2	188	120	93	1.29
BFP8	8.25	124.0	63.4	241	147	166	0.88
NVFP4	4.5	72.7	73.6	200	127	136	0.93
Q2_K	2.625	47.2	97.0	198	150	137	1.10
Q3_K	3.4375	57.1	141.7	252	195	203	0.96
Q4_K	4.5	72.8	77.9	204	131	153	0.86
Q5_K	5.5	85.9	120.1	259	174	224	0.77
Q6_K	6.5625	100.4	117.6	271	171	194	0.88
Q3_CB8	3.75	61.5	80.7	196	134	114	1.18
Q4_CB16	4.75	76.6	71.5	202	125	118	1.06
IQ4_XS	4.25	69.4	82.7	206	136	129	1.06
IQ4_NL	4.5	72.9	74.0	200	127	106	1.20

scale is therefore a strict improvement on FP16 (smaller, faster, as accurate), and E8M0 buys the same saving at a large accuracy cost.

At 8 bits the order of the scales reverses in error. The element grid is fine enough that the scale's own quantization dominates: E8M0 and FP8 scales raise the error $2.4\times$ and $3.4\times$ over Q8_0, at an unchanged decode cost, since all three are bound by fetch at this width. The scale datatype is therefore a width-dependent choice, and no single narrow scale is best across the range.

With FP4 elements the scale has a larger effect on time than on the integer path. MXFP4 is as fast as Q4_0 on dense decode and faster on the experts, at 33% higher error than the FP16-scaled QFP4_0. NVFP4, which pairs an FP8 scale with a group of 16 rather than 32, has the lowest FP4 error but is 31–37% slower on decode than Q4_0. Hierarchy matters here only through the outer scale: the FP8 rows need the FP32 per-tensor scale to reach the errors shown, and it is applied outside the inner loop, so it adds no significant time. Other factors including the group size and the FP4 elements are the major reasons for the slowdown.

B.5.3 Element Datatype

Table 14 holds the scale at FP16 and the group at 32 and changes the element. Each additional integer bit divides the error by roughly 4.6 (INT3 to INT4 by 5.4, INT4 to INT5 by 4.6) for one more bit of fetch per element; on decode INT5 is 22% slower than INT4, while INT3 is 12% faster, the extra unpacking of a 3-bit field being cheaper than the bytes it saves. A zero point (the Qn_1

Table 10: Decode kernel phases on the routed-expert kernel of Qwen3.5 35B A3B (μs); dequantization is the per-block scale decode. Phases are profiled in isolation, *measured* is the end-to-end kernel.

Format	Fetch	Dequant	Compute	Measured
Q3_0	101	62	49	124
Q4_0	108	52	49	125
Q5_0	137	63	46	150
Q8_0	164	48	48	168
Q3_1	98	61	49	116
Q4_1	109	53	49	122
Q5_1	125	67	47	133
Q4_0_FP8	102	56	50	114
Q4_1_FP8	110	56	49	125
Q5_0_FP8	130	67	48	150
Q5_1_FP8	124	61	47	138
Q8_0_FP8	158	47	48	164
QFP4_0	111	58	42	118
QFP8_0	167	73	43	237
BFP4	100	55	50	116
MXFP4	101	59	42	113
BFP8	158	49	51	162
NVFP4	129	75	45	166
Q2_K	141	84	48	185
Q3_K	153	103	47	232
Q4_K	172	69	48	212
Q5_K	192	85	48	270
Q6_K	172	94	47	226
Q3_CB8	112	85	46	187
Q4_CB16	130	77	46	132

Table 11: Decode weight path with Q4_0 weights, relative to FP32 activations, weighted by each shape's call count. *Q8_0* assumes activations already arrive quantized; *Q8_0 + conv* includes the per-token FP32→Q8_0 conversion, which is what a deployment pays.

Model	FP32 (ms)	FP16	Q8_0	Q8_0 + conv
Qwen3.5 4B	42.1	1.19×	0.98×	1.03×
Llama 3.1 8B	67.5	1.32×	1.03×	1.06×
Qwen3.5 35B A3B, dense path	26.0	1.10×	0.93×	1.12×
Qwen3.5 35B A3B, routed experts	15.3	1.03×	0.96×	1.12×

rows) lowers the error by 22% at every width for 0.25 bits per element, the gain of an asymmetric grid on weight distributions that are not centred at zero; its decode cost is within the noise of the symmetric kernel at 3 bits and 9–11% higher at 4 and 5 bits on the dense path.

On the expert path Q5_1 is faster than Q5_0, which is a kernel artefact (the symmetric expert kernel lacks an activation-reuse optimization), not a property of the zero point.

Floating-point elements lose to integers of the same width on both axes. FP4 E2M1 at FP16 scale (QFP4_0) has 5% more error than INT4 and costs a lookup per element: 4% slower on dense decode and 45% slower on the expert prefill. FP8 E4M3 has $20\times$ the error of INT8 at the same 8.5 bits, since its non-uniform grid spends codes on a dynamic range the block scale already provides, and it is the slowest format of the set on decode ($2.01\times$ Q4_0, against $1.76\times$ for Q8_0) because neither the vector nor the tensor units take FP8 operands. On this capability class the element therefore should be an integer.

Table 12: Q4_0 at group size g, relative to $g = 32$. Weight error is the median relative error over every linear tensor of Qwen3.5 4B and Llama 3.1 8B (the two agree to within 0.2%). Decode is the per-step weight path over the captured shapes, weighted by call count.

			Decode weight path			
g	bpe	Weight error	Qwen3.5 4B	Llama 3.1 8B	Qwen3.5 35B A3B dense	Qwen3.5 35B A3B MoE
16	5.00	0.76×	1.06×	1.10×	1.06×	1.27×
32	4.50	1.00×	1.00×	1.00×	1.00×	1.00×
64	4.25	1.25×	1.00×	0.98×	0.95×	0.93×
128	4.125	1.51×	0.95×	0.92×	0.94×	0.93×
256	4.0625	1.78×	0.96×	0.91×	0.93×	0.90×

Table 13: Scale datatype varied at a fixed element datatype, group size 32 unless noted. Weight error is the median relative error over every linear tensor of Qwen3.5 4B (Llama 3.1 8B agrees to within 1%). Kernel columns are relative to Q4_0 (Table 6); dequantization is the isolated phase at 4096×2048 (μs, Table 9). The FP8 rows carry the per-tensor FP32 outer scale. ‡NVFP4 is defined at group size 16.

Element	Scale	Format	bpe	Weight error	Dense prefill	Dense decode	MoE prefill	MoE decode	Dequant
INT4	FP16	Q4_0	4.5	9.78×10^{-3}	1.00	1.00	1.00	1.00	52.7
	FP8 E4M3	Q4_0_FP8	4.25	9.89×10^{-3}	0.99	0.93	1.02	0.91	55.3
	E8M0	BFP4	4.25	1.71×10^{-2}	0.99	0.93	1.01	0.92	53.9
INT8	FP16	Q8_0	8.5	2.98×10^{-5}	1.03	1.76	1.10	1.38	61.9
	FP8 E4M3	Q8_0_FP8	8.25	1.02×10^{-4}	0.98	1.75	1.09	1.33	63.6
	E8M0	BFP8	8.25	7.16×10^{-5}	0.98	1.74	1.08	1.32	63.4
FP4 (E2M1)	FP16	QFP4_0	4.5	1.02×10^{-2}	1.07	1.04	1.45	0.97	65.9
	FP8 E4M3	NVFP4‡	4.5	9.01×10^{-3}	1.08	1.37	1.52	1.31	73.6
	E8M0	MXFP4	4.25	1.36×10^{-2}	1.06	1.00	1.38	0.91	66.2

B.6 KERNEL-TO-MODEL LEVEL MEASUREMENT AGREEMENT

The cost model used throughout is built from kernels measured in isolation: each tensor's per-call time at its shape and format, multiplied by its call count per step, plus a term for everything that is not a weight matrix product. Table 15 checks that model against the running model on Qwen3.5 35B A3B for nine GGUF configurations, from uniform Q4_0 through expert-only low-bit mixes and two of our earlier allocations to the community K-quants. Decode is measured over 256 generated tokens and averaged; prefill is a single 512-token prompt, matching the $N = 1$ and $N = 512$ kernel captures.

Residual terms The non-matmul time (normalization, attention score computation, the recurrent state update of the linear-attention layers, activation functions, and per-operator synchronization) is *measured*: per-operator timers placed in the model graph record it directly, at 26.4–28.0 ms per decode step and 2.23–2.39 s per prefill. It is nearly independent of the weight format, and it is large, 27–40% of the decode step and 38–48% of the prefill. What remains unmeasured is the synchronization and cache eviction between operators that the timers do not fully capture, which appears as the gap between the profiled sum and llama.cpp's end-to-end figure. That gap is within $\pm 2\%$ on decode (0.985–1.021) and between -2% and $+5\%$ on prefill.

Agreement The kernel-built estimate overpredicts the measured decode time by 1–6% on every configuration whose weight path is made of the QN_0, QN_1 and codebook formats, and by 13% on Q4_K_M, whose isolated K-quant kernel times run above their cost inside the model. On prefill it overpredicts by 1–8%. The ordering it implies is consistent and used by the allocator, and the rare exceptions are pairs whose measured times differ by less than 2%. The bias is one-signed and near-uniform across configurations, so comparisons between configurations priced from the same kernel captures are affected far less than absolute times.

Table 14: Element datatype varied at a fixed FP16 block scale and group size 32. Columns as in Table 13.

Element	Format	bpe	Weight error	Dense prefill	Dense decode	MoE prefill	MoE decode	Dequant
INT3	Q3_0	3.5	5.32×10^{-2}	1.05	0.88	1.00	1.00	71.9
INT4	Q4_0	4.5	9.78×10^{-3}	1.00	1.00	1.00	1.00	52.7
INT5	Q5_0	5.5	2.13×10^{-3}	0.98	1.22	1.47	1.14	73.6
INT8	Q8_0	8.5	2.98×10^{-5}	1.03	1.76	1.10	1.38	61.9
INT3 + ZP8	Q3_1	3.75	4.17×10^{-2}	0.99	0.86	1.01	0.94	62.7
INT4 + ZP8	Q4_1	4.75	7.66×10^{-3}	1.00	1.09	1.04	0.95	66.1
INT5 + ZP8	Q5_1	5.75	1.67×10^{-3}	0.98	1.35	1.40	1.04	78.5
FP4 E2M1	QFP4_0	4.5	1.02×10^{-2}	1.07	1.04	1.45	0.97	65.9
FP8 E4M3	QFP8_0	8.5	6.10×10^{-4}	1.16	2.01	1.69	1.86	85.4

Table 15: Kernel-to-model agreement on Qwen3.5 35B A3B, per GGUF configuration. *Kernel* is the step built from the isolated kernel captures (weight path by call count, plus the measured non-matmul time and router); *profiled* is the sum of per-operator times measured inside the running model; *measured* is llama.cpp's own end-to-end figure (decode averaged over 256 generated tokens, prefill for a 512-token prompt). All in ms. *Rest* is every non-expert tensor of a linear layer.

	Decode (ms per token)				Prefill (ms, 512 tokens)			
Configuration	Kernel	Profiled	Measured	Kernel / meas.	Kernel	Profiled	Measured	Kernel / meas.
Q4_0	69.2	66.6	65.4	1.058	4930	4873	4642	1.062
Q3_0 experts, Q8_0 rest	86.1	81.9	81.3	1.059	4952	4776	4592	1.078
Q3_1 experts, Q8_0 rest	84.8	81.4	80.6	1.052	4954	4789	4604	1.076
Q4_0 experts, Q8_0 rest	85.9	82.8	82.0	1.048	4935	4875	4895	1.008
Q4_1 experts, Q8_0 rest, F16 head	91.1	88.8	87.0	1.047	5006	4852	4885	1.025
Q4_CB16 experts, Q8_0 rest	91.2	88.9	88.5	1.031	5178	5052	5029	1.030
Q3_CB8 experts, Q8_0 rest	91.5	87.8	87.0	1.052	5190	5046	4956	1.047
Q4_K_M	103.3	93.0	91.7	1.126	5616	5340	5373	1.045
Q3_K_M	103.5	102.4	102.0	1.015	6541	5983	6110	1.071

C Mixed Precision Allocator assumptions and derivations

This appendix states the assumptions behind the objective of Section 4, proves the identities used there, compares the label and context choices available for estimating it, and specializes it to sites that are not ordinary linear layers. The procedure itself is Algorithm 1 in Appendix C.5.

C.1 Notation

We follow the notation in Section 4 here. A calibration sequence $w_{1:n}$ supplies one context $x = w_{<r}$ per position r, whose next token is distributed as $p_{\theta_0}(\cdot \mid x)$. Within a context, tensor t is applied at every position s, with input h_s, output $o_s = W_t h_s$, and score $g_s = \partial \log p_{\theta_0}(y \mid x)/\partial o_s$. For a format f we write

$$a_s = g_s^\top \mathcal{E}_{t,f}\, h_s, \qquad \Phi(p) = \mathrm{diag}(p) - pp^\top,$$

where $\Phi(p)$ is the covariance of a one-hot label drawn from p. The expectation E_x is the average over the N contexts of the calibration set, so every per-tensor quantity is a per-token average.

The objective omits the factor $\frac{1}{2}$ of the expansion: $D_{t,f}$ is twice the predicted contribution to (1). The factor is common to every (t, f) and leaves the solution of Eq. (2) unchanged; wherever a predicted divergence is quoted, it is $D/2$.

C.2 Assumptions

Assumption 1 (Local quadraticity)**.** *$L(\theta_0 + \Delta)$ is represented by its second-order expansion about θ_0 (Proposition 1); the $O(\|\Delta\|^3)$ remainder is discarded.*

The second-order expansion of a loss about trained weights is the standard basis for curvature-based compression, from the saliency criteria of Optimal Brain Damage and Optimal Brain Surgeon (Le-

Cun et al., 1989; Hassibi et al., 1993) to mixed-precision allocation by Hessian sensitivity (Dong et al., 2019; 2020; Yao et al., 2021) and second-order rounding and reconstruction (Nagel et al., 2020; Li et al., 2021; Frantar et al., 2023). Those works assume the first-order term is negligible because the weights sit near a minimum of the training loss. Here it vanishes identically, since θ_0 is the reference of its own divergence (Proposition 1), so the expansion discards only the cubic and higher terms. These terms shrink relative to the quadratic one as the perturbation shrinks, so the approximation is tightest at the higher bit-widths and weakest at the lowest candidates in $\mathcal{F}$.

Assumption 2 (Block-diagonal Fisher)**.** *The cross-tensor terms of $\Delta^\top F \Delta$ are discarded, so the objective is additive over tensors.*

Partitioning Δ by tensor,

$$\tfrac{1}{2}\Delta^\top F \Delta \;=\; \tfrac{1}{2}\sum_t \Delta_t^\top F_{tt} \Delta_t \;+\; \tfrac{1}{2}\sum_{t \neq t'} \Delta_t^\top F_{tt'} \Delta_{t'}, \tag{3}$$

and the objective retains the first sum only. The reason is structural. A per-tensor cost that depended on the rest of the assignment would make the objective quadratic in z; the program would cease to be linear, and one coefficient table could no longer serve every budget on the frontier with optimality guarantees. The same block-diagonal treatment underlies curvature approximations for neural networks generally (Martens & Grosse, 2015) and every integer-programming formulation of mixed-precision allocation we are aware of, which score each layer or block in isolation and sum the scores (Dong et al., 2019; Yao et al., 2021; Guan et al., 2024; Li et al., 2023; Duanmu et al., 2025). The discarded term has indefinite sign, and we do not bound it here.

Assumption 3 (Cross-position independence)**.** *Within a context, the products $a_s a_{s'}$ with $s \neq s'$ are discarded.*

Because W_t is shared across positions, the exact quadratic form is the second moment of the sum $\sum_s a_s$ (Proposition 2). The discarded products are nonzero wherever a sequence-mixing layer (attention, or the linear-attention layers of the Qwen3.5 models) lies downstream of t: sites s and s' both perturb later outputs, and the divergence there is quadratic in the sum of those perturbations. The products vanish for the language-model head (Proposition 4) and for any tensor downstream of the last sequence-mixing layer.

Treating the positions of a weight-shared layer as independent samples is the standard simplification in curvature estimation for such layers. It is the spatially uncorrelated derivatives assumption of Kronecker-factored curvature for convolutions (Grosse & Martens, 2016), similar to the setting in which the weight-sharing dimension is treated like the batch dimension (Eschenhagen et al., 2023), and the per-token treatment of weight-shared layers in (Bolshim & Kugaevskikh, 2026). Two features limit its effect here. It is exact at the head, which alone carries a large share of the objective, and at every tensor past the last sequence-mixing layer. Elsewhere, only the part of the discarded mass that varies across tensors changes the allocation, since a factor common to all tensors cancels in Eq. (2). We do not bound that variation.

Assumption 4 (Fixed routing)**.** *In Mixture-of-Experts layers, top-k membership is held at its full-precision value.*

The gate scores are differentiable and enter through the gradient (Proposition 5), but membership is piecewise constant in the router input: a change in which experts are selected is a discontinuity the expansion does not represent. The router itself is held unquantized (Appendix D.1), so such changes arise only through perturbations of its input.

In summary, the objective rests on Assumptions 1–4: the first three are standard in curvature-based allocation, the head is exact under the first two alone, and the fourth is inherent to any perturbation analysis around fixed routing.

Definitional choices. Two further choices define the objective. The contexts of (1) are calibration prefixes under teacher forcing, so the objective is the divergence on that context distribution; a deployed model conditions on its own generations instead, and the self-generated half of the calibration set moves the context distribution toward that regime. The labels are drawn from p_{θ_0}, which introduces variance but no bias (Proposition 3).

C.3 DERIVATIONS

Proposition 1 (Second-order form). *$L(\theta_0) = 0$, $\nabla_\theta L(\theta_0) = 0$, and $\nabla^2_\theta L(\theta_0) = F$ with $F = \mathrm{E}_x \mathrm{E}_{y\sim p_{\theta_0}}\big[\nabla_\theta \log p_{\theta_0}(y \mid x)\, \nabla_\theta \log p_{\theta_0}(y \mid x)^\top\big]$. Under Assumption 1, therefore, $L(\theta_0 + \Delta) = \frac{1}{2}\Delta^\top F \Delta$.*

Proof. Write $L(\theta) = \mathrm{E}_x \sum_y p_{\theta_0}(y \mid x)\big[\log p_{\theta_0}(y \mid x) - \log p_\theta(y \mid x)\big]$, which is zero at θ_0. Differentiating, $\nabla_\theta L = -\mathrm{E}_x \sum_y p_{\theta_0} \nabla_\theta \log p_\theta$; at θ_0 this is $-\mathrm{E}_x \nabla_\theta \sum_y p_\theta\big|_{\theta_0} = -\mathrm{E}_x \nabla_\theta 1 = 0$, the first Bartlett identity. Differentiating again, $\nabla^2_\theta L = -\mathrm{E}_x \sum_y p_{\theta_0} \nabla^2_\theta \log p_\theta$. Using $\nabla^2 \log p = \nabla^2 p / p - \nabla \log p \, \nabla \log p^\top$ and $\sum_y \nabla^2 p_\theta = 0$, at θ_0 this equals F, the second Bartlett identity. □

Proposition 2 (Block form at a linear site). *For a tensor t applied at positions s of a context x,*

(i) $F_{tt} = \mathrm{E}_x \mathrm{E}_{y\sim p_{\theta_0}}\big[\sum_{s,s'} (h_s h_{s'}^\top) \otimes (g_s g_{s'}^\top)\big]$;

(ii) $\mathrm{vec}(\mathcal{E}_{t,f})^\top F_{tt}\, \mathrm{vec}(\mathcal{E}_{t,f}) = \mathrm{E}_x \mathrm{E}_{y\sim p_{\theta_0}}\big[\big(\sum_s a_s\big)^2\big]$;

(iii) under Assumption 3, the per-tensor cost is $D_{t,f} = \mathrm{E}_x \mathrm{E}_{y\sim p_{\theta_0}}\big[\sum_s a_s^2\big]$.

Proof. Since $o_s = W_t h_s$ at every position, $\partial \log p / \partial (W_t)_{ij} = \sum_s g_{s,i} h_{s,j}$, so $\nabla_{W_t} \log p = \sum_s g_s h_s^\top$ and $\mathrm{vec}(\nabla_{W_t} \log p) = \sum_s h_s \otimes g_s$. Its outer product is $\sum_{s,s'} (h_s \otimes g_s)(h_{s'} \otimes g_{s'})^\top = \sum_{s,s'} (h_s h_{s'}^\top) \otimes (g_s g_{s'}^\top)$, which gives (i). For (ii), $(B \otimes C)\,\mathrm{vec}(A) = \mathrm{vec}(CAB^\top)$, so $\mathrm{vec}(A)^\top (B \otimes C)\,\mathrm{vec}(A) = \mathrm{tr}(A^\top C A B^\top)$. With $A = \mathcal{E}_{t,f}$, $B = h_s h_{s'}^\top$ and $C = g_s g_{s'}^\top$ this is $(g_s^\top \mathcal{E}_{t,f} h_s)(g_{s'}^\top \mathcal{E}_{t,f} h_{s'}) = a_s a_{s'}$, and summing over s, s' gives $(\sum_s a_s)^2$. Item (iii) is (ii) with the terms $s \neq s'$ removed. □

Proposition 3 (Unbiasedness of the sampled estimator). *Let $\hat{y}_r \sim p_{\theta_0}(\cdot \mid w_{<r})$ be drawn independently at every position of a calibration sequence, let $\ell = \sum_r \log p_{\theta_0}(\hat{y}_r \mid w_{<r})$, and let $g_s = \partial \ell / \partial o_s$. Then the estimator $\widehat{D}_{t,f}$ of Section 4 satisfies $\mathrm{E}[\widehat{D}_{t,f}] = D_{t,f}$.*

Proof. By linearity, $g_s = \sum_r g_s^{(r)}$ with $g_s^{(r)} = \partial \log p_{\theta_0}(\hat{y}_r \mid w_{<r}) / \partial o_s$, which is the score of Proposition 2 for the context $x = w_{<r}$ and label $\hat{y}_r$, and is zero at sites outside that context. We write accordingly $a_s = \sum_r a_s^{(r)}$. Given the sequence, $a_s^{(r)}$ depends on $\hat{y}_r$ alone and has zero mean, since $\mathrm{E}_{\hat{y}_r}[\partial \log p / \partial o_s] = \partial\big(\sum_y p\big)/\partial o_s = 0$. The labels are independent, so $\mathrm{E}[a_s^{(r)} a_s^{(q)}] = 0$ for $r \neq q$ and

$$\mathrm{E}\Big[\sum_s a_s^2\Big] = \sum_r \mathrm{E}_{\hat{y}_r}\Big[\sum_s \big(a_s^{(r)}\big)^2\Big],$$

which is the sum over the contexts $w_{<r}$ of the per-context cost of Proposition 2(iii). Dividing by the number of contexts N gives $D_{t,f}$. □

The argument removes products between different *label* positions only, which vanish exactly. Products between different *site* positions within one label's term do not vanish; they are the terms discarded by Assumption 3. Explicitly, with $u_r = e_{\hat{y}_r} - p_r$ the score at the logits of position r and $\zeta_{r,s}$ the perturbation of those logits caused by $\mathcal{E}_{t,f} h_s$, $a_s^{(r)} = u_r^\top \zeta_{r,s}$ and $\mathrm{E}\big[(a_s^{(r)})^2\big] = \zeta_{r,s}^\top \Phi(p_r) \zeta_{r,s}$ (Proposition 4).

Proposition 4 (Closed form at the language-model head). *At the head, o_s is the logit vector at position s. With $\zeta_s = \mathcal{E}_{t,f} h_s$,*

$$\mathrm{E}_{\hat{y}_s\sim p_s}\big[g_s g_s^\top\big] = \Phi(p_s), \qquad \mathrm{E}_{\hat{y}_s\sim p_s}\big[a_s^2\big] = \sum_i p_{s,i} \zeta_{s,i}^2 - \Big(\sum_i p_{s,i} \zeta_{s,i}\Big)^2, \tag{4}$$

and $\mathrm{E}[a_s a_{s'}] = 0$ for $s \neq s'$. The head's cost is therefore exact under Assumptions 1 and 2 alone.

Proof. The head at position s affects only the prediction at s, so $g_s = e_{\hat{y}_s} - p_s$. Using $\sum_y p_y e_y = p$ and $\sum_y p_y e_y e_y^\top = \mathrm{diag}(p)$, $\mathrm{E}[g_s g_s^\top] = \sum_y p_y (e_y - p)(e_y - p)^\top = \mathrm{diag}(p) - pp^\top$. Then $\mathrm{E}[(g_s^\top \zeta_s)^2] = \zeta_s^\top \Phi(p_s) \zeta_s$, which expands to the right-hand side. For $s \neq s'$, a_s and $a_{s'}$ depend on distinct, independent labels and each has zero mean. □

Proposition 5 (Routed experts)**.** *Let a layer compute $\sum_{e \in \mathrm{topk}(s)} \pi_e(s)\, V_e\, \phi(U_e h_s)$, with router score π_e, input projection U_e (gate and up), activation ϕ and output projection V_e, and let g_s^{L} be the score at the layer output. Under Assumption 4:*

(i) the score at V_e's output is $\pi_e(s)\, g_s^{\mathrm{L}}$ on tokens routed to e and zero otherwise;

(ii) the score at U_e's output is $J_\phi(U_e h_s)^\top V_e^\top \pi_e(s)\, g_s^{\mathrm{L}}$ on routed tokens and zero otherwise, with J_ϕ the Jacobian of ϕ;

(iii) for the tensor t stacking one matrix per expert, Proposition 2 applies with $a_s = \sum_{e \in \mathrm{topk}(s)} g_{e,s}^\top \mathcal{E}_{t,f,e}\, h_{e,s}$, where $g_{e,s}$ and $h_{e,s}$ are the score and input of expert e's matrix on token s.

Proof. Items (i) and (ii) are the chain rule: expert e's output enters the layer output only through the scalar $\pi_e(s)$ and only when $e \in \mathrm{topk}(s)$. For (iii), the stacked tensor's parameter vector is the concatenation of the experts', so the inner product of $\mathrm{vec}(\mathcal{E}_{t,f})$ with its gradient is $\sum_s \sum_e g_{e,s}^\top \mathcal{E}_{t,f,e} h_{e,s}$, and the proof of Proposition 2 applies term by term. □

Routing frequency and gate strength therefore enter through the gradient rather than through a separately constructed importance weight: an expert selected on few tokens, or with small π_e, contributes proportionally less. Unrouted tokens contribute nothing, so dividing by N keeps expert and dense costs commensurate.

Proposition 6 (Output MSE as the isotropic-score case)**.** *Conditional on the context, $\mathrm{E}_y[a_s^2 \mid x] = \zeta_s^\top \mathrm{E}_y[g_s g_s^\top \mid x]\, \zeta_s$ with $\zeta_s = \mathcal{E}_{t,f} h_s$. Replacing $\mathrm{E}_y[g_s g_s^\top \mid x]$ by I in $D_{t,f}$ gives the layer output error $\mathrm{E}_x \sum_s \|\mathcal{E}_{t,f} h_s\|^2$.*

Proof. Given x, ζ_s is fixed and $a_s = g_s^\top \zeta_s$, so $\mathrm{E}_y[a_s^2 \mid x] = \zeta_s^\top \mathrm{E}_y[g_s g_s^\top \mid x] \zeta_s$. With the identity in place of the conditional second moment this is $\|\zeta_s\|^2$. □

Output MSE is thus the objective obtained by asserting that every output direction of every tensor is propagated to the model output equally. The assertion fails across tensors, whose scores differ in norm by orders of magnitude, and within a tensor, whose score is anisotropic across output channels.

C.4 Non-standard sites

Three sites are not ordinary matrix products over a calibration activation. Each is handled by specializing Proposition 2 rather than by a separate construction.

Language-model head Proposition 4 evaluates the head in closed form from p_s and h_s, with no sampling variance and no discarded cross-position terms; it needs no backward pass.

Routed experts Proposition 5 gives the expert scores from the layer-output score g_s^{L}, the routing, and the unquantized expert weights. The pass therefore retains the layer input, g_s^{L} and the routing for each MoE layer, and the expert contributions are evaluated from them after the pass, once per expert and format. This avoids rebuilding every expert's $\mathcal{E}_{t,f,e}$ in each calibration batch, which would dominate the cost for tensors holding most of the parameters.

Embedding Writing the embedding as $o_s = W^\top e_{j_s}$, with $W \in \mathbb{R}^{V \times H}$ and e_{j_s} the one-hot of the token at position s, places it in the form of Proposition 2 with input dimension V. Under Assumption 3,

$$D_{\mathrm{embed},f} \;=\; \mathrm{E}_x \mathrm{E}_y \Big[\sum_s \big(g_s^\top \varepsilon_f[j_s] \big)^2 \Big], \tag{5}$$

Algorithm 1 Mixed-precision allocation over the viable format set.

Require: Quantizable tensors $\mathcal{T}$; viable formats $\mathcal{F}$ (Section 3.4); storage budget $\bar{\sigma}$; decode budget $\bar{\tau}$; calibration set of N positions

for all $t \in \mathcal{T}$ and $f \in \mathcal{F}$ **do**
  $\sigma_{t,f} \leftarrow$ BPE of the realized block layout $\times$ parameter count
  $\tau_{t,f} \leftarrow$ kernel time at t's shape $\times$ calls per decode step
  $\mathcal{E}_{t,f} \leftarrow Q_f(W_t) - W_t$ ▷ RTN, fixed importance matrix
end for
for all batches of calibration sequences **do**
  Forward at θ_0, retaining the input h_s at every site
  Draw $\hat{y}_r \sim p_{\theta_0}(\cdot \mid w_{<r})$ at each non-pad position ▷ temperature 1, untruncated
  Backward on $\ell = \sum_r \log p_{\theta_0}(\hat{y}_r \mid w_{<r})$ to obtain g_s at every site
  for all linear $t \in \mathcal{T}$ other than the head and experts, and $f \in \mathcal{F}$ **do**
    $A_{t,f}$ += $\sum_s \left(g_s^\top \mathcal{E}_{t,f} h_s\right)^2$
  end for
  Head: $A_{t,f}$ += $\sum_s \left[\sum_i p_{s,i}\zeta_{s,i}^2 - (\sum_i p_{s,i}\zeta_{s,i})^2\right]$ ▷ Prop. 4
  Embedding: $A_{t,f}$ += $\sum_s \left(g_s^\top \varepsilon_f[j_s]\right)^2$ on observed rows ▷ Eq. (5)
  Retain layer input, layer-output score g_s^{L} and routing for each MoE layer
end for
for all expert tensors t and $f \in \mathcal{F}$ **do**
  Reconstruct $g_{e,s}$ and $h_{e,s}$ from the retained quantities ▷ Prop. 5
  $A_{t,f}$ += $\sum_s \left(\sum_{e \in \mathrm{topk}(s)} g_{e,s}^\top \mathcal{E}_{t,f,e} h_{e,s}\right)^2$
end for
$\widehat{D}_{t,f} \leftarrow A_{t,f}/N$
Merge a tied embedding and head into one decision; fix tensors excluded from the search
Solve Eq. (2) for z by branch-and-cut; record the optimality gap
return the assignment $\{(t,f) : z_{t,f} = 1\}$

with $\varepsilon_f[j]$ the quantization error on row j. At this site Assumption 3 also discards the products between positions that carry the same token, which share one row error. Quantization groups run along the hidden axis, so rows are quantized independently and evaluating ε_f on the observed rows alone reproduces the contraction exactly. The embedding is a gather rather than a matrix product at decode, so $\tau_{t,f} = 0$ for an untied embedding and it is constrained by storage alone.

Tied Weights Where the embedding and head share one tensor, a single perturbation acts at two sites with different inputs, a one-hot at the gather and a hidden state at the head. The objective sums the two contributions, $D = D_{\mathrm{embed}} + D_{\mathrm{head}}$, under one decision variable, discarding the products between the two sites as Assumption 3 discards those between positions. Storage is charged once and decode cost at the head only. Tying is detected by tensor identity rather than by a configuration flag, since the two can disagree.

Coverage Only rows that appear in the calibration set receive gradient. The number of distinct tokens grows far more slowly than the calibration length, so even at 10^5 positions the covered fraction of a 10^5–10^6-entry vocabulary is small: $D_{\mathrm{embed},f}$ is an average over covered rows extended to the remainder, while storage is charged on all V rows. Three observations bound the concern. The covered rows are the frequent ones, so they carry most of the probability mass that (1) integrates against, and an uncovered row contributes to L only on sequences containing it. The per-row contraction is exact where it is measured, so the error is one of extrapolation across rows rather than estimation within them. And the extrapolation is shared across formats, so it shifts $D_{\mathrm{embed},f}$ by a factor common to all f and affects the embedding's ranking against other tensors more than the choice of format for it. We nonetheless regard this as the least-supported element of the objective.

C.5 Procedure and implementation

Calibration The objective is estimated once per model from 250 calibration sequences (about 2×10^5 tokens), half generated by the full-precision model and half drawn from the WikiText-2 training split. One label is sampled per position from p_{θ_0} at unit temperature without truncation, with the sampling seed recorded, and the head is evaluated in closed form (Proposition 4). The errors $\mathcal{E}_{t,f}$ are produced by the same round-to-nearest quantizer and importance matrix used to build the deployed recipe (Section 5).

Runtime constraints The constraints of the llama.cpp runtime are respected throughout: all experts of a layer share one format, and in the Mixture-of-Experts model the router and shared-expert gate are held at FP32 and excluded from the search.

C.6 Allocations

This appendix expands Section 4. It analyzes the per-tensor objective and cost tables the program solves over, and the resultant allocations achieved.

C.6.1 Objective frontier

Figure 7 traces the objective the program reaches against the storage budget at three decode budgets. The frontier falls by a factor of 4 to 8 per bit per element over the swept range. At the Q4_K_M, Q4_K_S and Q3_K_M decode budgets it coincides with the unconstrained frontier, so those budgets are free: the candidate formats are fast enough that a storage-optimal assignment already meets them. The Q4_0 decode budget separates from it, and on the dense models its frontier goes flat beyond about 2.7 GB (Qwen3.5 4B) and 4.8 GB (Llama 3.1 8B): the remaining ways to spend storage are all formats that decode more slowly, so extra bytes buy nothing at that speed. On Qwen3.5 35B A3B it keeps falling, more slowly, because the routed experts still have fast formats to move to.

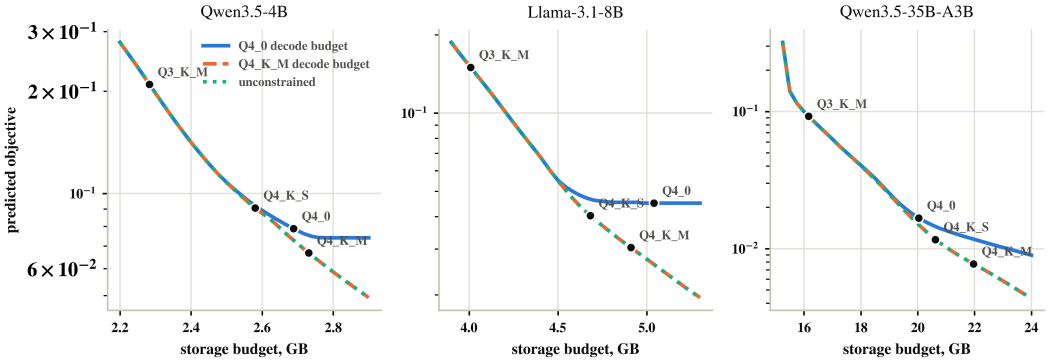


Figure 7: Predicted objective against storage budget at the decode budgets of the Q4_0 and Q4_K_M matches and without a decode constraint. The Q4_K_S and Q3_K_M decode budgets coincide with the unconstrained line and are not drawn. Points are the stored allocations.

C.6.2 Uniform, storage-only and decode-constrained allocation

Table 16 compares three different approaches to allocation: uniform allocation, mixed precision based on storage alone, and decode constrained allocation. Allocating rather than choosing one format has the predicted cost: the best uniform format that fits the storage budget carries a 2 to 9 times higher objective. The effect of conditioning on decode cost is the smallest, and it acts only where the baseline is fast. Against the K-quants the storage-only program already returns an assignment inside the decode budget, identical to the full program, so the decode advantage over those files comes from the viable set of Section 3.4 rather than from the constraint. Against Q4_0 the constraint binds on all three models and trades objective for speed: 8% on Qwen3.5 4B, 72% on Llama 3.1 8B and 12% on Qwen3.5 35B A3B, for a predicted decode 6–17% shorter than the

storage-only assignment. The decode figures are the kernel level estimates; the measurements of Table 2 are the accurate comparisons across configurations.

Table 16: Different allocations at every matched budget, from the solver alone (no quantization or evaluation): predicted decode (ms per token) and objective for the best uniform candidate format that fits the storage budget, the program with the storage constraint only, and the full program. The full program reproduces each stored allocation exactly. Bold marks where the decode constraint binds.

			Best uniform			Storage only		Storage + decode (ours)	
Model	Budget	Decode budget	Format	Decode	Objective	Decode	Objective	Decode	Objective
Qwen3.5 4B	Q4_0	48.2	Q4_1_FP8	42.8	0.5891	51.4	0.0728	**48.2**	**0.0788**
	Q3_K_M	86.8	BFP4	40.4	0.9533	45.5	0.2096	45.5	0.2096
	Q4_K_S	75.1	Q4_1_FP8	42.8	0.5891	49.4	0.0906	49.4	0.0906
	Q4_K_M	81.2	Q4_1_FP8	42.8	0.5891	51.4	0.0668	51.5	0.0668
Llama 3.1 8B	Q4_0	71.6	IQ4_NL	81.2	0.0890	86.3	0.0262	**71.6**	**0.0450**
	Q3_K_M	129.7	Q3_CB8	80.2	0.6209	72.2	0.1499	72.7	0.1499
	Q4_K_S	97.3	IQ4_NL	81.2	0.0890	81.5	0.0403	81.5	0.0403
	Q4_K_M	103.3	IQ4_NL	81.2	0.0890	85.3	0.0304	85.3	0.0304
Qwen3.5 35B A3B	Q4_0	73.9	IQ4_NL	67.0	0.0611	79.4	0.0150	**73.9**	**0.0167**
	Q3_K_M	104.0	Q3_0	62.3	0.5615	70.7	0.0923	70.7	0.0923
	Q4_K_S	96.1	IQ4_NL	67.0	0.0611	80.5	0.0117	80.5	0.0117
	Q4_K_M	98.8	IQ4_NL	67.0	0.0611	81.5	0.0078	81.5	0.0078

C.6.3 Per-role assignment

Table 17 lists, for every stored allocation, the bits each tensor role receives and the formats it uses. Three patterns hold across models. The output head and the attention projections are held near 8 bits once the budget allows it, the MLP or routed-expert weights absorb the remaining reduction, and the program uses most of the candidate set: a single role mixes up to nine formats, the FP8-scale and IQ4_NL variants most often, because formats of nearly equal width differ in per-tensor cost and error and the program picks the cheaper one tensor by tensor.

C.6.4 Sensitivity across tensors

Figure 8 sets each tensor's sensitivity (its objective term at Q4_0) beside the bits the Q4_K_M-matched allocation gives it. Sensitivity is concentrated rather than uniform: on Qwen3.5 4B the linear-attention projections of the first two layers alone account for half of the uniform-Q4_0 objective, and on Llama 3.1 8B the first layers' value projections and the first and last MLP down projections stand out. The allocation follows sensitivity per byte rather than sensitivity alone. Those few sensitive tensors receive 8 bits, as do the small linear-attention gates and shared-expert projections of Qwen3.5 35B A3B, which cost little storage even where their sensitivity is modest; the routed experts, sensitive but holding most of the parameters, stay near 4.5 bits.

D Baselines

D.1 Implementation details

Calibration and probe The objective of Equation 2 is estimated once per model from 250 calibration sequences (about 2×10^5 tokens), half generated by the full-precision model and half drawn from the WikiText-2 training split. One label is sampled per position from p_{θ_0} at unit temperature, and the language-model head is evaluated in closed form (Appendix C.4).

Runtime constraints The constraints of the llama.cpp runtime are respected throughout: all experts of a layer share one format, and in the Mixture-of-Experts model the router and shared-expert gate are held at FP32.

Community K-quants Each K-quant baseline is the published GGUF file of Han et al. (2023), apart from the Q3_K_M for Qwen3.5 35B A3B which is taken from Bartowski (2026) to maintain

Table 17: Per-role assignment of every stored allocation: parameter-weighted bits per element of each tensor role and the formats the role uses (most parameters first). Under each budget are the whole-model bits per element and the predicted objective D.

Model	Budget	Role	bpe	Formats
Qwen3.5 4B	Q4_0	full attn.	5.48	IQ4_NL, Q5_0_FP8, Q8_0, Q5_1_FP8, Q5_1, Q4_0_FP8
	5.11 bpe	linear attn.	5.31	Q5_0_FP8, Q4_0_FP8, IQ4_NL, Q8_0, Q4_1_FP8, Q5_1_FP8, Q5_1, Q5_0, BFP4
	D = 0.0788	MLP	5.15	Q5_0_FP8, IQ4_NL, Q4_1_FP8, Q4_0_FP8, Q5_1, Q5_1_FP8, Q5_0, Q8_0
		embedding (tied)	4.50	Q4_1_FP8
	Q3_K_M	full attn.	4.49	IQ4_NL, Q4_0_FP8, Q3_CB8, Q4_1_FP8, Q5_1_FP8, Q5_0_FP8
	4.34 bpe	linear attn.	4.30	Q3_CB8, Q4_0_FP8, IQ4_NL, Q3_0, Q5_0_FP8, Q5_1_FP8, Q8_0, Q5_0, Q5_1
	D = 0.2096	MLP	4.29	Q4_0_FP8, IQ4_NL, Q3_CB8, Q4_1_FP8, Q5_0_FP8, Q5_1_FP8
		embedding (tied)	4.50	Q4_1_FP8
	Q4_K_S	full attn.	5.40	IQ4_NL, Q5_1_FP8, Q5_0_FP8, Q8_0, Q5_1, Q4_0_FP8
	4.91 bpe	linear attn.	5.17	IQ4_NL, Q4_0_FP8, Q5_0_FP8, Q8_0, Q5_1, Q5_0, Q5_1_FP8, Q4_1_FP8
	D = 0.0906	MLP	4.84	IQ4_NL, Q5_0_FP8, Q5_1_FP8, Q5_1, Q4_1_FP8
		embedding (tied)	4.50	Q4_1_FP8
	Q4_K_M	full attn.	5.54	IQ4_NL, Q5_0_FP8, Q5_1_FP8, Q8_0, Q5_1
	5.19 bpe	linear attn.	5.30	IQ4_NL, Q5_0_FP8, Q8_0, Q4_0_FP8, Q5_1, Q5_1_FP8, Q4_1_FP8
	D = 0.0668	MLP	5.01	IQ4_NL, Q5_0_FP8, Q5_1_FP8, Q5_1, Q8_0
		embedding (tied)	5.50	Q5_0
Llama 3.1 8B	Q4_0	attention	5.14	Q5_0_FP8, Q4_0_FP8, Q5_0, Q8_0, Q5_1_FP8, Q5_1
	4.97 bpe	MLP	4.58	Q4_0_FP8, Q5_0_FP8, Q8_0, Q5_1, Q5_0
	D = 0.0450	lm head	5.25	Q5_0_FP8
		embedding	8.50	Q8_0
	Q3_K_M	attention	4.11	Q4_0_FP8, IQ4_NL, Q3_0, Q3_CB8, Q5_1_FP8, Q5_0_FP8
	3.99 bpe	MLP	3.96	Q3_CB8, Q4_0_FP8, Q3_0, IQ4_NL, Q8_0, Q5_0_FP8
	D = 0.1499	lm head	4.50	IQ4_NL
		embedding	3.50	Q3_0
	Q4_K_S	attention	4.86	IQ4_NL, Q5_0_FP8, Q4_0_FP8, Q5_1_FP8, Q5_0, Q5_1, Q8_0, Q3_0, Q4_0
	4.66 bpe	MLP	4.60	IQ4_NL, Q4_0_FP8, Q5_0_FP8, Q8_0
	D = 0.0403	lm head	5.25	Q5_0_FP8
		embedding	4.25	Q4_0_FP8
	Q4_K_M	attention	5.07	IQ4_NL, Q5_0_FP8, Q5_0, Q5_1_FP8, Q5_1, Q4_0_FP8, Q8_0
	4.89 bpe	MLP	4.80	IQ4_NL, Q5_0_FP8, Q5_1, Q8_0, Q5_1_FP8
	D = 0.0304	lm head	5.75	Q5_1
		embedding	4.50	IQ4_NL
Qwen3.5 35B A3B	Q4_0	full attn.	7.33	Q8_0, Q5_0_FP8, Q5_1
	4.62 bpe	linear attn.	6.09	Q5_0_FP8, Q8_0, Q5_1_FP8, Q5_1, Q5_0, BFP4
	D = 0.0167	shared expert	7.20	Q8_0, Q5_0, Q5_1_FP8, Q5_1
		routed experts	4.46	IQ4_NL, Q4_0_FP8, Q5_0_FP8
		lm head	8.50	Q8_0
		embedding	5.50	Q5_1_FP8
	Q3_K_M (b.)	full attn.	5.82	Q5_0_FP8, Q5_1, Q8_0, Q5_1_FP8, Q5_0
	3.73 bpe	linear attn.	5.42	Q5_0_FP8, Q5_1_FP8, IQ4_NL, Q5_1, Q8_0, Q5_0
	D = 0.0923	shared expert	5.65	Q5_1_FP8, Q5_0_FP8, Q5_1, IQ4_NL, Q8_0, Q5_0
		routed experts	3.60	Q3_0, Q3_CB8, Q4_0_FP8
		lm head	5.75	Q5_1
		embedding	4.50	IQ4_NL
	Q4_K_S	full attn.	8.50	Q8_0
	4.76 bpe	linear attn.	8.36	Q8_0, Q5_1, Q5_1_FP8, Q5_0
	D = 0.0117	shared expert	8.43	Q8_0, Q5_1
		routed experts	4.52	IQ4_NL, Q5_0_FP8, Q4_0_FP8
		lm head	8.50	Q8_0
		embedding	5.50	Q5_1_FP8
	Q4_K_M	full attn.	8.50	Q8_0
	5.07 bpe	linear attn.	8.50	Q8_0, Q5_1, Q5_0
	D = 0.0078	shared expert	8.43	Q8_0, Q5_1, Q5_1_FP8
		routed experts	4.85	IQ4_NL, Q5_0_FP8, Q5_0
		lm head	8.50	Q8_0
		embedding	5.75	Q5_1

consistent use of K-quants across all baselines. For accuracy it is dequantized to a BF16 checkpoint and evaluated with the same pipeline, settings, and batch size as every other configuration, so no difference in harness separates a baseline from its paired recipe. Llama 3.1 8B is read by the GGUF loader of the transformers library; the Qwen3.5 architectures are not yet supported and are converted by a dedicated script. Storage is the published file size. Decode time is approximated for allocation using the kernel level measurements. The header also records the importance matrix each file was built with, and the paired recipe uses that same matrix.

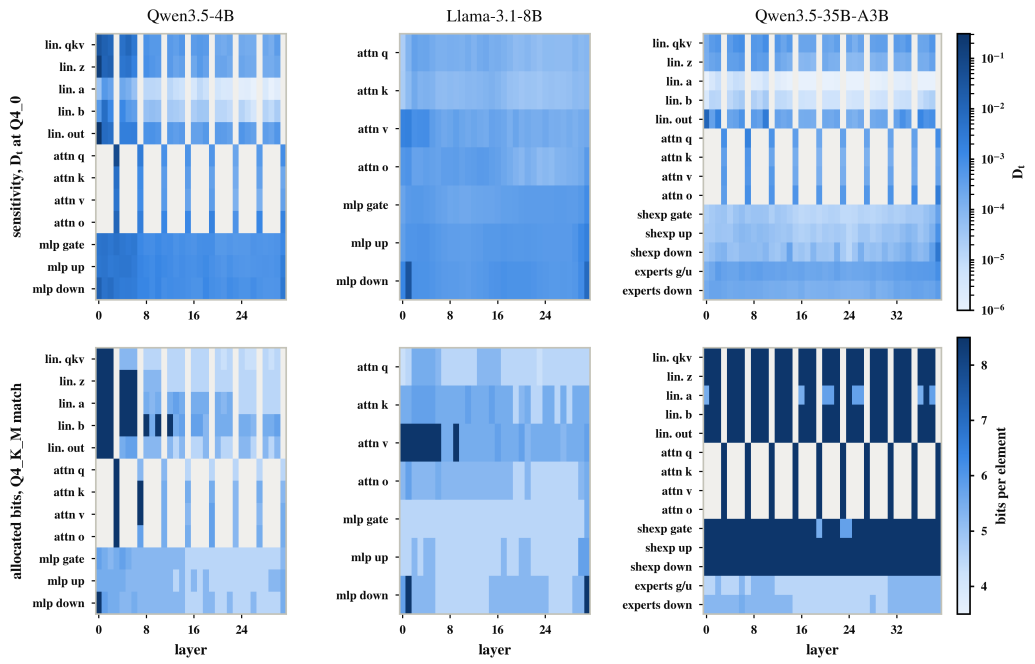


Figure 8: Top: per-tensor objective at uniform Q4_0, by layer and role (log scale). Bottom: bits per element of the Q4_K_M-matched allocation. Grey cells are roles absent from that layer (the Qwen3.5 models alternate linear- and full-attention layers).

Q4_0 This format quantizes every linear tensor to Q4_0 at group size 32 by round-to-nearest, keeps the token embedding and the output layer at Q8_0, and keeps the norms at FP32. No importance matrix is used, so this is the file llama.cpp produces by default. On Qwen3.5 35B A3B the router is held at FP32, as in every other row. The dense files are built directly with llama.cpp, and storage is the file size, as for the K-quant rows. The Qwen3.5 4B file omits the multi-token-prediction block that llama.cpp emits as an extra layer, which the published K-quant files also omit and which ordinary decoding does not use.

E Evaluation Benchmarks

E.1 Performance Measurements

E.1.1 Kernel-level measurement on the ET-SoC-1

Every kernel time in the paper (Tables 1, 3–15) is measured on a single ET-SoC-1 card with a standalone benchmark harness that launches one kernel at a time on the device. For each launch, the harness records the in-kernel cycle count through the ET-SoC-1's performance monitoring unit, takes the median over the harts of the neighbourhoods, and converts it to time at 600 MHz; host launch overhead is excluded. Each kernel is launched at least 32 times, with each run starting from cold caches. All kernels consume the weights in their native llama.cpp block layout, excluding any memory layout repacking. Decode kernels run at $N = 1$ on the vector units and prefill kernels at $N = 512$ on the tensor units. Shapes are the weight-matrix shapes captured from each model; routed-expert kernels are measured with 256 experts and 8 active per token, drawn as distinct experts from a fixed-seed routing table. Weights are generated with each format's own quantizer, and every kernel is verified against a scalar reference on its full shape sweep before its timings are used. The phase decomposition of Appendix B.3 isolates each phase by removing the other components from the kernel and timing the remainder in the same harness.

E.1.2 END-TO-END MEASUREMENT ON THE ET-SOC-1

The decode and TTFT columns of Table 2 are measured by running each GGUF file end to end with the llama.cpp ET backend on a single ET-SoC-1 card. Each run processes a fixed 512-token prompt and then generates 256 tokens at a fixed seed. Each benchmark uses llama.cpp's official benchmark tooling; TTFT is llama.cpp's reported processing time for the prompt, while decode throughput is the mean generation rate over the output tokens. Each configuration is run 10 times with the average reported.

E.2 ACCURACY MEASUREMENTS

E.2.1 EVALUATION CONFIGURATION

All models, community baselines included, are dequantized to BF16 and evaluated with the language-model evaluation harness (Biderman et al., 2024) under greedy decoding. C4 perplexity is computed on the first validation shard of the English split. Documents are concatenated in file order with a blank-line separator, tokenized with each model's own tokenizer, and cut into non-overlapping windows of 2048 tokens; the first 256 windows (524,288 tokens) are scored, and perplexity is the exponent of the mean token negative log-likelihood. WikiText-2 uses the same windowing on the full raw test split (wikitext-2-raw-v1), all windows scored. Quantization is weight-only (fake quantization); activations are not quantized. Generative tasks use the model's chat template with thinking disabled across all models for fairness. Table 18 gives the protocol for each benchmark.

Table 18: Evaluation protocol per benchmark.

Benchmark	Items	Num-shot	Scoring	Metric
WikiText-2	test split	–	log-likelihood	perplexity
C4	256 windows	–	log-likelihood	perplexity
ARC-Easy	2376	0	log-likelihood	length-normalized accuracy
ARC-Challenge	1172	0	log-likelihood	length-normalized accuracy
HellaSwag	10042	0	log-likelihood	length-normalized accuracy
MMLU-Redux	5330	0	generative, chat	exact match
GSM8K	1319	5	generative, chat	exact match, flexible extraction
MATH-500	500	0	generative, chat	exact match
GPQA Diamond	198	0	generative, chat	accuracy
MBPP+	378	0	generative, chat	pass@1

E.2.2 PER-BENCHMARK RESULTS

Tables 20 and 21 give accuracy on each benchmark for every configuration of Table 19, whose accuracy columns are the mean of these.

E.2.3 RESULTS AT ALL FOUR BUDGETS

Table 19 extends Table 2 with the Q4_0 and Q4_K_S budgets. At Q4_K_S the recipe decodes $1.41\times$, $1.21\times$ and $1.07\times$ faster than the K-quant on Qwen3.5 4B, Llama 3.1 8B and Qwen3.5 35B A3B, within one point of its accuracy (0.0, -0.8, $+0.4$). Uniform Q4_0 is the only baseline the recipes do not outpace: at its decode budget they are 3%, 9% and 13% slower on the device, as the solver's kernel-level estimate under-prices the other viable formats relative to Q4_0 within the full model. In exchange they are more accurate at equal storage (99.5 against 95.3% of BF16 on Qwen3.5 4B, 98.8 against 97.7% on Llama 3.1 8B, level at 99.3% on Qwen3.5 35B A3B), with lower perplexity on all three models.

F WORKED EXAMPLE: OPTIMIZING THE Q4_0 DECODE KERNEL

To make the kernel measurements concrete, this appendix walks through one kernel: the Q4_0 matrix-vector implementation used for dense decode, which is also the reference against which every format in Table 1 is normalized. We first give a plain-C device kernel, then the structure of the

Table 19: Results at all four budgets (Table 2 shows Q3_K_M and Q4_K_M). Storage, decode throughput (end to end on the ET-SoC-1, tok/s over 256 generated tokens), perplexity and mean accuracy. Each *Ours* row is solved within the storage and decode budgets of the baseline above it. Accuracy columns average Tables 20 and 21 (GSM8K flexible extract); the last column is the seven-benchmark mean (ARC-E excluded) relative to BF16.

		Storage		Decode	Perplexity ↓		Accuracy (%) ↑		
Model	Configuration	GB	bpw	tok/s	Wiki2	C4	Non-reas.	Reas.	of BF16
Qwen3.5 4B	BF16	8.41	16.00	5.6	9.59	13.74	63.8	81.4	100
	Q4_0	2.70	5.13	14.1	9.88	14.24	62.8	76.8	95.3
	Ours	2.70	5.14	13.7	9.83	14.11	63.2	81.2	99.5
	Q3_K_M	2.29	4.36	9.0	9.90	14.37	62.0	77.9	96.1
	Ours	2.30	4.37	15.0	10.28	14.59	63.6	77.7	96.5
	Q4_K_S	2.59	4.93	9.9	9.73	13.95	63.0	80.3	98.7
	Ours	2.59	4.93	14.0	9.95	14.14	63.0	80.3	98.7
	Q4_K_M	2.74	5.21	9.4	9.77	13.91	63.1	79.4	97.8
	Ours	2.74	5.22	13.4	9.76	13.98	63.0	81.0	99.3
Llama 3.1 8B	BF16	16.06	16.00	3.6	7.22	11.39	67.5	60.1	100
	Q4_0	5.05	5.03	11.6	7.56	12.09	67.0	58.3	97.7
	Ours	5.00	4.98	10.5	7.42	11.77	66.7	59.4	98.8
	Q3_K_M	4.02	4.00	6.5	7.69	12.28	66.3	56.9	95.8
	Ours	4.02	4.00	11.2	8.01	12.78	65.7	55.8	94.2
	Q4_K_S	4.69	4.67	8.5	7.37	11.70	67.2	59.6	99.3
	Ours	4.69	4.67	10.3	7.39	11.76	67.7	58.7	98.5
	Q4_K_M	4.92	4.90	8.0	7.37	11.69	66.8	59.7	99.2
	Ours	4.92	4.90	9.9	7.35	11.68	67.4	59.4	99.1
Qwen3.5 35B A3B	BF16	69.32	16.00		6.59	10.68	72.4	89.5	100
	Q4_0	20.08	4.64	13.9	6.74	10.97	71.4	89.1	99.3
	Ours	20.09	4.64	12.1	6.62	10.74	72.1	88.9	99.3
	Q3_K_M	16.22	3.74	9.6	6.74	10.92	71.2	89.1	99.2
	Ours	16.22	3.74	12.9	6.77	11.05	71.1	88.5	98.7
	Q4_K_S	20.67	4.77	11.0	6.62	10.73	72.0	89.0	99.4
	Ours	20.67	4.77	11.8	6.62	10.72	72.0	89.5	99.8
	Q4_K_M	22.02	5.08	10.6	6.60	10.71	71.9	89.4	99.7
	Ours	22.01	5.08	11.7	6.59	10.71	72.0	90.1	100.3

optimized kernel, then each optimization with the code that implements it. The listings in §F.3 are excerpts of the optimized kernel's source.

F.1 Scalar device kernel

A Q4_0 block stores 32 weights as sixteen bytes of packed 4-bit codes and one FP16 scale. The kernel below runs on the device as written. Each of the 2,048 hardware threads (harts) takes every 2,048th row and computes its dot product with scalar instructions only. The result is written with an atomic store, which bypasses the non-coherent L1 so the host sees it.

```
typedef struct { uint16_t d; uint8_t qs[16]; } block_q4_0;  // 32 weights, 18 B

int entry_point(struct MatvecParams *p, void *env) {
  int64_t hart = get_hart_id();                                  // 0 .. 2047
  for (int64_t n = 0; n < p->N; n++) {
    const float *x = p->B + n * p->nb11;                         // activation row n
    float       *y = p->C + n * p->nbd1;                         // output row n
    for (int64_t m = hart; m < p->M; m += 2048) {
      const block_q4_0 *row = p->A + m * p->nb01;                // weight row m
      float acc = 0.0f;
      for (int64_t b = 0; b < p->K / 32; b++) {
        float d = fp16_to_fp32(row[b].d);
        for (int j = 0; j < 16; j++) {
          int lo = (row[b].qs[j] & 0x0F) - 8;                    // weight j
          int hi = (row[b].qs[j] >> 4) - 8;                      // weight j + 16
          acc += d * lo * x[32*b + j] + d * hi * x[32*b + j + 16];
        }
      }
      atomic_store_f32(&y[m], acc);                              // bypass L1
    }
```

Table 20: Non-reasoning accuracy (%), zero-shot, length-normalized. Avg. is over ARC-C and HellaSwag; ARC-E is shown but not averaged.

Model	Configuration	ARC-E	ARC-C	HellaSwag	Avg.
Qwen3.5 4B	BF16	75.5	54.6	73.1	63.8
	Q4_0	75.9	53.7	71.9	62.8
	Ours	75.2	54.2	72.3	63.2
	Q3_K_M	75.3	52.7	71.2	62.0
	Ours	78.2	55.6	71.6	63.6
	Q4_K_S	74.8	53.5	72.4	63.0
	Ours	75.9	53.6	72.4	63.0
	Q4_K_M	74.1	53.3	72.8	63.1
	Ours	74.8	53.4	72.6	63.0
Llama 3.1 8B	BF16	79.8	55.5	79.5	67.5
	Q4_0	79.1	54.9	79.1	67.0
	Ours	79.1	54.4	78.9	66.7
	Q3_K_M	77.8	54.4	78.1	66.3
	Ours	79.7	53.6	77.8	65.7
	Q4_K_S	79.3	55.6	78.7	67.2
	Ours	79.4	56.5	79.0	67.7
	Q4_K_M	79.7	54.7	78.8	66.8
	Ours	79.6	55.6	79.2	67.4
Qwen3.5 35B A3B	BF16	79.3	62.3	82.5	72.4
	Q4_0	77.9	60.8	81.9	71.4
	Ours	80.1	62.0	82.2	72.1
	Q3_K_M	78.2	60.7	81.7	71.2
	Ours	80.4	61.4	80.8	71.1
	Q4_K_S	79.6	61.8	82.3	72.0
	Ours	79.7	62.0	82.1	72.0
	Q4_K_M	79.4	61.4	82.3	71.9
	Ours	79.8	61.5	82.5	72.0

```
  }
  return 0;
}
```

F.2 Structure of the optimized kernel

The optimized kernel keeps the arithmetic and reorganizes the work around it. Before any dot product, the activation is copied into each shire's on-chip L2 scratchpad. The kernel then chooses one of three work distributions from the shape (M, N, K). Every path computes its row dot products with the same vectorized routine, `q4_dot_compute`, which replaces the scalar inner loops. Numbers in parentheses refer to the optimizations in §F.3.

```
int entry_point(struct MatvecQ4_0xF32GemvParams *params, void *env) {
  uint64_t hart_id = get_hart_id();

  // (4) activation staging: copy x into the shire's L2 scratchpad once
  if (stage_b) {
    /* the shire's 64 harts each DMA a slice of x */
    et_barrier(ET_BARRIER_SHIRE);
  }

  // shape-dependent dispatch
  if (use_w_dma) {
    // (6) N == 1, M % 2048 == 0, M >= 4096, K >= 4096:
    //     each hart streams one full row through the scratchpad
  } else if (use_ksplit || use_ksplit_group) {
    // (5) few rows per hart, long K:
    //     the two harts of a core each take half of K
  } else {
```

Table 21: Reasoning accuracy (%), thinking disabled. GSM8K is strict-match / flexible-extract; Avg. uses flexible-extract. MBPP+ is pass@1.

Model	Configuration	MMLU-Redux	GSM8K	MATH-500	GPQA-D	MBPP+	Avg.
Qwen3.5 4B	BF16	78.0	85.9 / 88.5	92.8	65.7	82.0	81.4
	Q4_0	76.5	81.0 / 84.5	88.4	57.1	77.5	76.8
	Ours	77.7	83.9 / 87.1	91.6	69.2	80.2	81.2
	Q3_K_M	76.5	83.4 / 84.8	88.8	62.1	77.5	77.9
	Ours	76.0	82.0 / 85.0	90.4	58.1	79.1	77.7
	Q4_K_S	77.5	85.4 / 87.4	91.6	65.7	79.4	80.3
	Ours	77.5	82.9 / 86.7	93.4	64.1	79.9	80.3
	Q4_K_M	77.7	86.0 / 87.7	91.6	62.1	77.8	79.4
	Ours	78.0	85.1 / 87.0	92.4	67.2	80.4	81.0
Llama 3.1 8B	BF16	69.6	77.8 / 82.1	48.8	22.2	77.8	60.1
	Q4_0	68.1	77.8 / 80.0	47.2	21.2	75.1	58.3
	Ours	69.2	70.7 / 81.4	45.4	23.2	77.8	59.4
	Q3_K_M	66.4	71.8 / 77.8	45.6	21.2	73.5	56.9
	Ours	64.9	72.9 / 74.8	40.4	26.8	72.0	55.8
	Q4_K_S	68.7	75.6 / 80.8	46.0	27.3	75.1	59.6
	Ours	69.3	75.7 / 81.3	43.6	25.3	73.8	58.7
	Q4_K_M	68.3	77.0 / 80.9	46.0	25.3	77.8	59.7
	Ours	69.1	79.0 / 83.2	44.6	25.8	74.1	59.4
Qwen3.5 35B A3B	BF16	87.1	95.0 / 95.5	95.8	78.3	91.0	89.5
	Q4_0	87.0	94.7 / 95.1	95.6	77.3	90.5	89.1
	Ours	87.0	95.0 / 95.6	95.0	74.7	92.1	88.9
	Q3_K_M	86.5	94.1 / 94.6	95.2	77.8	91.5	89.1
	Ours	86.6	93.5 / 94.5	95.6	74.7	91.3	88.5
	Q4_K_S	87.2	94.8 / 95.5	95.6	74.2	92.3	89.0
	Ours	87.0	95.2 / 95.5	95.8	76.8	92.3	89.5
	Q4_K_M	87.1	95.2 / 95.5	96.2	77.3	91.0	89.4
	Ours	87.3	94.2 / 95.1	96.8	79.8	91.5	90.1

```
    // default: stripe rows over the harts
    for (int64_t m = hart_id; m < M; m += STRIDE_M)        // STRIDE_M = 2048
      /* q4_dot_compute(row m, x) */;
  }
  return 0;
}

// per-row dot product, shared by every path
float q4_dot_compute(const block_q4_0 *q_row, const float *b_col,
                     int64_t K_blocks, int pf_enabled) {
  q4_dot_reset();                                  // zero f20-f23
  q4_dot_tile(q_row, b_col, K_blocks, pf_enabled);  // (1)-(3) block loop
  return q4_dot_reduce();                          // (2) final reduction
}
```

F.3 Optimizations

(1) Vector block decode. Each block is decoded on the 8-lane vector unit: two byte gathers load its 16 packed bytes, and each 8-weight chunk is unpacked, converted and scaled in five instructions. The FP16 block scale is converted once per block.

```
fbcx.ps     f6, scale          # broadcast FP16 block scale, convert once
fcvt.ps.f16 f6, f6
fgb.ps      f8, f31(qs)        # gather qs[0..7]  into 8 lanes
fgb.ps      f9, f31(qs+8)      # gather qs[8..15] into 8 lanes
flw.ps      f12, x[0..7]       # ... f13-f15: x[16..23], x[8..15], x[24..31]
fandi.pi    f2, f8, 15         # weights 0..7: low nibbles
faddi.pi    f2, f2, -8         #   re-centre
fcvt.ps.pw  f2, f2             #   int32 -> FP32
```

```
fmul.ps     f2, f2, f6         #   apply scale
fmadd.ps    f20, f2, f12, f20  #   accumulate
...                            # high nibbles (fsrli.pi) and f9 -> f21, f22, f23
```

(2) Hoisted loads and independent accumulators. All loads of a block are issued before use, and its four 8-weight chunks accumulate into four separate registers, so consecutive multiply-adds never wait on each other. The four are combined once per row:

```
# per block: one accumulator per chunk
fmadd.ps  f20, f2, f12, f20    # weights  0..7
fmadd.ps  f21, f3, f13, f21    # weights 16..23
fmadd.ps  f22, f4, f14, f22    # weights  8..15
fmadd.ps  f23, f5, f15, f23    # weights 24..31

# per row: 4 accumulators -> 1, then 8 lanes -> 1
fadd.ps   f20, f20, f21 ;  fadd.ps f22, f22, f23 ;  fadd.ps f20, f20, f22
fswizz.ps f1, f20, 0xB1 ;  fadd.ps f2, f20, f1  ...
```

(3) Weight prefetch. The weight line `Q4_PF_DIST` (= 4) blocks ahead is requested into L2, once per 64-byte line.

```
uint64_t pf_line = (q_row + kb + Q4_PF_DIST) & ~63ULL;
if (pf_line != last_pf_line) {
  l2_prefetch(pf_line, 1, 64);
  last_pf_line = pf_line;
}
```

(4) Activation staging. The shire's 64 harts each DMA a slice of the activation into the L2 scratchpad; a barrier publishes it, and all dot products then read it on-chip.

```
while (remaining > 0) {                    // this hart's slice, 16 lines per DMA
  et_tensor_load_l2scp_conf_t conf = { .dst_start = dst_line, .addr = src,
                                       .num_lines = cl - 1, .stride = 64 };
  et_tensor_load_l2scp(&conf);
  WAIT_TENSOR_LOAD_L2_0;
  ...
}
et_barrier(ET_BARRIER_SHIRE);
```

(5) K-split for few rows. The two harts of a core each compute half of a row's dot product; the second publishes its partial sum through the scratchpad and the first adds and stores.

```
float partial = compute_row_dot_q4_0(q_row + k_start, b_col + k_start*32, k_len, pf);
if (is_hart1) {
  *l2scp_slot = partial;  FENCE;  flush_to_l2(l2scp_slot, 1, 64);  WAIT_CACHEOPS;
  et_sem_post(ET_BARRIER_MINION);  et_sem_wait(ET_BARRIER_MINION);
} else {
  et_sem_wait(ET_BARRIER_MINION);
  float other = *l2scp_slot;
  et_sem_post(ET_BARRIER_MINION);
  atomic_store_f32(dst, partial + other);
}
```

(6) Full-row DMA for large aligned matrices. Each hart streams its row into the scratchpad in 576-byte chunks, double-buffered: chunk $c{+}1$ is requested before chunk c is used.

```
for (int64_t c = 0; c < n_chunks; c++) {
  if ((c & 1) == 0) { WAIT_TENSOR_LOAD_L2_0; } else { WAIT_TENSOR_LOAD_L2_1; }
  et_barrier(ET_BARRIER_SHIRE);
  if (c + 1 < n_chunks) w_dma_issue(buf[(c+1) & 1], row + (c+1) * chunk_bytes, ...);
  acc += q4_dot_compute(buf[c & 1], b_col + c * W_KCHUNK_BLOCKS * 32,
                        W_KCHUNK_BLOCKS, 0);
}
```