AutoQuant: Automatic Mixed-Precision Quantization of LLMs under User-Specified Memory Budgets

Telukuntla Vinay Koushik[1], Ramabathina Yashaswini[2], Mohanapriya V[3] and M.K Gowtham[4,]
Asis Marceline V[5]
[1,2,3,4,5]*Department of Computational Intelligence, SRM Institute of Science and Technology, Kattankulathur, Chengalpattu, Tamilnadu, India 603203*

***Abstract*—Uniform-precision quantization forces a losing trade-off between compression depth and generation quality — AutoQuant eliminates it. AutoQuant calculates a composite per-layer sensitivity score from 99th-percentile-clipped squared weight gradients fused with activation magnitudes. It then drives a two-phase greedy allocator over (INT4, INT8, INT16) — locking insensitive layers at INT4 via a cliff heuristic and blocking unnecessary INT16 upgrades via a gated threshold — all without fine-tuning or labeled data. Quantized layers use weight-only symmetric quantization with grouped column scales (group size 128) and a per-output-row FP16 residual to make up for rounding errors. They work with Hugging Face causal LMs through a REST API and an interactive dashboard. Evaluated on GPT-2 and open Llama-family checkpoints, AutoQuant achieves up to 3.8× compression with less than 2% perplexity degradation relative to the FP16 reference — at identical compressed sizes, consistently outperforming uniform INT8 baselines.**

***Index Terms*—Large Language Models, Mixed-Precision Quantization, Post-Training Quantization, Sensitivity Analysis, Model Compression, Weight-Only Quantization.**

I. INTRODUCTION

The rise of transformer-based large language models (LLMs) has changed natural language processing in a big way, but they can't be used in the real world because a 7-billion-parameter model takes up about 14 GB in half-precision, which is more than most consumer and edge-grade GPUs can handle. The gap between research capability and practical deployability gets bigger as model size grows. This means that compression is no longer just a nice thing to have; it's a must-have. Model quantization solves this problem directly by using less precise numbers to represent weights, which saves memory and bandwidth without changing the network topology. In this case, post-training quantization (PTQ) is very appealing because it only needs a small, unlabeled calibration corpus and doesn't require any extra processing power to retrain. The main problem is that transformer layers don't all respond the same way when precision is lost. Empirical evidence shows that embedding layers, early and final transformer blocks, and attention projection heads are much more fragile than intermediate feed-forward layers. Uniform quantization, which gives every layer the same bit-width, doesn't take this difference into account, which means that quality is lost where compression isn't needed and precision is lost where it matters most. Mixed-precision quantization, which chooses bit-widths based on how sensitive each layer is, is a principled way to solve this problem.

Existing approaches occupy different points in the design space. GPTQ [1] minimizes layer-wise reconstruction error via approximate second-order weight correction; AWQ [2] identifies salient weight channels through activation magnitude statistics; IMPQ [3] models cross-layer interactions through Shapley-style attribution. Each method achieves strong results but carries meaningful overhead — specialized CUDA kernels, expensive sensitivity searches, or architecture-specific assumptions — that limits accessibility for resource-constrained or CPU-bound deployment scenarios.

AutoQuant fills this gap with a PTQ pipeline that is lightweight, works with any hardware, and is easy to understand. This work specifically contributes: (i) a composite per-layer sensitivity estimator that combines squared weight gradients clipped at the

99th percentile with forward-pass activation magnitudes; (ii) a two-phase greedy knapsack allocator with a sensitivity cliff heuristic and a gated INT16 upgrade threshold that assigns bit-widths from (INT4, INT8, INT16) under a user-specified GB budget; and (iii) quantized linear layers with grouped column scales (group size 128) and a per-output-row FP16 residual, requiring no custom kernels and serializing directly to the Hugging Face checkpoint format. The whole pipeline can be accessed through both a REST API and an interactive web dashboard. This makes it easier for people who don't have specialized infrastructure to use mixed-precision deployment.

The remainder of this paper is organized as follows. Section II surveys related work. Section III details the AutoQuant methodology. Section IV describes implementation. Section V presents experimental results. Section VI discusses limitations and future directions, and Section VII concludes.

II. LITERATURE SURVEY

Quantization of large language models has evolved rapidly across three interconnected dimensions: the granularity of the sensitivity signal, the flexibility of precision assignment, and the practicality of end-to-end deployment — each generation of methods advancing one dimension while leaving others unresolved.

The first PTQ frameworks for LLMs set the groundwork for later work. ZeroQuant [1] showed that token-wise dynamic activation quantization and group-wise weight scaling could work together to achieve near-lossless INT8 compression on GPT-J-6B and GPT-NeoX-20B. This made grouped quantization a practical standard. LLM.int8() [2] solved the outlier problem by using a decomposed matrix multiplication. It sent outlier activation dimensions through an FP16 path while compressing the rest through INT8. This made it possible to do lossless inference for models with more than 100B parameters without any calibration data. Then, GPTQ [3] formalized weight-only compression by changing the Optimal Brain Quantization framework to work with transformers. It quantized weights column-by-column with approximate Hessian correction, getting 3–4-bit compression of the OPT and BLOOM families with almost no loss of perplexity. This set the standard for engineering that AutoGPTQ, HuggingFace Transformers, and TensorRT-LLM would later use.

The next wave was defined by the improvement of the sensitivity signal and the precision granularity. AWQ [4] discovered that roughly 1% of weight channels, characterized by significantly large activation magnitudes, are responsible for the majority of quality degradation during low-bit compression; safeguarding prominent channels through learned per-channel scaling results in robust INT4 quality without the need for backpropagation or layer-wise reconstruction. SmoothQuant [5] tackled the same outlier issue from the activation side by using a mathematically equivalent per-channel smoothing transformation that moves quantization difficulty from activations to weights. This makes W8A8 inference possible with throughput gains of up to 1.56× on standard INT8 GEMM hardware. QLoRA [6] showed that 4-bit NF4 quantization of a frozen backbone, along with LoRA adapters trained at FP16, can match the quality of full-precision fine-tuning on MMLU. This proves that aggressive compression can be used as a base for parameter-efficient adaptation. FlexGen [7] extended the compression-deployment relationship further, framing LLM inference under extreme memory constraints as a linear programming problem over GPU, CPU, and NVMe tiers, with fixed group-wise INT4/INT8 quantization integrated as a static preprocessing step to reduce data transfer costs.

The next wave was defined by the improvement of the sensitivity signal and the precision granularity. AWQ [4] discovered that roughly 1% of weight channels, characterized by significantly large activation magnitudes, are responsible for the majority of quality degradation during low-bit compression; safeguarding prominent channels through learned per-channel scaling results in robust INT4 quality without the need for backpropagation or layer-wise reconstruction. SmoothQuant [5] tackled the same outlier issue from the activation side by using a mathematically equivalent per-channel smoothing transformation that moves quantization difficulty from activations to weights. This makes W8A8 inference possible with throughput gains of up to 1.56× on standard INT8 GEMM hardware. QLoRA [6] showed that 4-bit NF4 quantization of a frozen backbone, along with LoRA adapters trained

at FP16, can match the quality of full-precision fine-tuning on MMLU. This proves that aggressive compression can be used as a base for parameter-efficient adaptation.

Even with this progress, five gaps that won't go away make it hard to put the ideas into practice. Uniform precision assignment is common in ZeroQuant [1], LLM.int8() [2], GPTQ [3], SmoothQuant [5], and FlexGen [7]. It doesn't take into account how sensitive different transformer layers are, wasting precision on layers that are tolerant and not protecting critical ones enough. There isn't a system that takes a memory target in gigabytes and automatically comes up with an optimized bit-width plan. Instead, each method requires the practitioner to directly specify the average bits per weight. Gradient-based sensitivity signals in Fisher methods [8][9] are still open to outlier spikes and single-domain calibration bias. Mixed-precision search is either too expensive to compute—Shapley computation in IMPQ [10] and block-level backward passes in FGMP [9]—or not available at all in the systems that are most commonly used [3][4][5]. Lastly, every method that was looked at needs a GPU to be available for at least one stage of the pipeline, and none of them makes a fully self-contained, reproducible Hugging Face-format artifact with all the tools needed to use it. AutoQuant is driven by all five gaps at once. It combines a clipped gradient-activation sensitivity estimator with a two-phase greedy knapsack under a user-defined GB budget. It is built entirely in standard PyTorch and delivered through a full pipeline with a REST API, an interactive dashboard, and standard HF checkpoint serialization.

III. PROPOSED METHODOLOGY

AutoQuant decomposes the mixed-precision quantization problem into four sequential stages: sensitivity estimation, bit-width allocation, layer quantization, and checkpoint serialization. Each stage is described below with its governing formulation.

A. Per-Layer Sensitivity Estimation

Not all transformer layers respond equally to precision loss. AutoQuant quantifies this by computing a composite sensitivity score for each linear layer l, combining two signals: how critically the loss depends on the layer's weights, and how large the layer's activations are during inference.

For each calibration sample x_i, a standard forward-backward pass collects the weight gradient. To suppress outlier spikes, gradients are clipped at the 99th percentile before squaring. The mean clipped squared gradient across N samples gives the gradient sensitivity term G_l — a lightweight diagonal Fisher Information approximation requiring no second-order computation:

$$\mathrm{Gl} = \mathrm{N1i} = 1\sum\mathrm{N} \parallel \mathrm{clip}(\mid \mathrm{gl,i} \mid, \mathrm{P99}) \parallel \mathrm{F2}$$

Simultaneously, the mean absolute activation magnitude A_l is recorded during the same forward passes. The two terms are multiplied and normalised to [0, 1] to produce the final composite score:

$$gl, i = \frac{\partial L(xi)}{\partial Wl}$$

A score is elevated only when both gradient criticality and activation load are simultaneously high — layers with large gradients but negligible activations, or vice versa, are correctly ranked as less sensitive.

B. Sensitivity Cliff Detection

Before allocation begins, AutoQuant automatically partitions layers into a sensitive group and an insensitive tail using a cliff heuristic — avoiding any manual threshold. Sensitivity scores are sorted in descending order, and the cliff index k* is defined as the position of the largest consecutive gap:

$$g'l, i = clip(\mid gl, i \mid,\ 0,\ P99(gl, i \mid))$$

All layers ranked below k* are unconditionally locked at INT4 and excluded from further budget consideration, preventing the allocator from wasting precision on layers that do not require it.

C. Two-Phase Greedy Knapsack Allocation

Bit-width assignment is formulated as a constrained optimisation over {INT4, INT8, INT16}. Given a user-specified budget B in gigabytes, the allocator maximises total weighted precision subject to a 95% effective memory constraint (reserving 5% for metadata):

$$Gl = 1/N\ N \sum i = 1 \parallel g'l, i \parallel^2\ F$$

where $M_l(b_l) = |W_l| \cdot b_l/8$ is the byte cost of layer l at bit-width b_l. Non-linear layers (Embedding,

LayerNorm) are always retained at FP16 and their fixed cost is pre-deducted from B.

The optimisation is solved greedily in two passes over layers sorted by $\hat{S}_l$ in descending order. Phase 1 attempts INT4 → INT8 upgrades for every candidate layer; each upgrade is committed if the cumulative budget is not exceeded. Phase 2 then attempts INT8 → INT16 upgrades, but only for layers whose sensitivity exceeds a configurable gate threshold τ (default 0.75). This two-pass structure ensures broad INT8 coverage before any INT16 upgrades are committed — preventing a small number of highly sensitive layers from consuming the entire budget at full precision.

D. Layer Quantization

Each layer assigned $b_l < 16$ is replaced by a DynamicQuantizedLinear module applying weight-only symmetric quantization with grouped column scales (default group size G = 128). For each output row r and column group g, the per-group scale and quantized weight are:

$$Al = N1i = 1\sum NTi \cdot dl1t, j\sum hl, i(t, j)$$

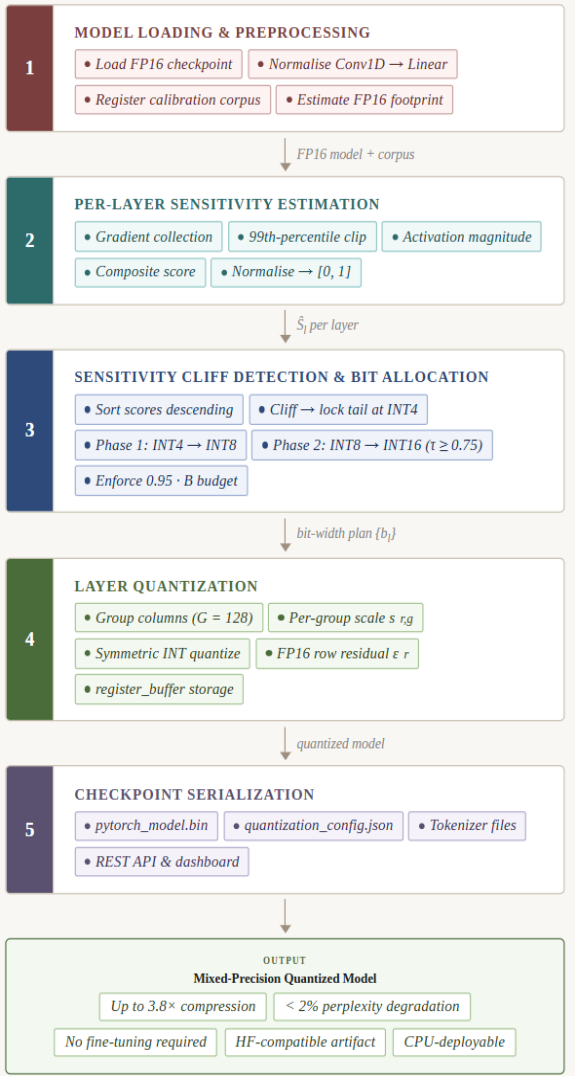
A per-output-row FP16 residual ε_r is stored to absorb the asymmetric rounding error that accumulates across the full row — recovering quality without retaining any full-precision weights. During inference, weights are dequantized on the fly:

$$Sl = Gl \cdot Al$$

Quantized weights and scales are stored as register_buffer entries rather than parameters, ensuring correct serialization, device transfer, and exclusion from any optimizer during downstream use.

E. Checkpoint Serialization

The quantized model is saved as a self-contained Hugging Face-compatible checkpoint: pytorch_model.bin containing all quantized buffers, scales, and residuals; quantization_config.json recording the per-layer bit-width plan, sensitivity scores, compression ratio, and all hyperparameters for full reproducibility; and the original tokenizer files copied unchanged. The resulting artifact requires no custom CUDA extensions and is directly loadable by any standard Hugging Face inference environment.



IV. RESULTS

A. Layer Sensitivity Profiling and Quantization Thresholding

This plot details the layer-wise sensitivity distribution of the model, which is a critical metric for determining optimal quantization bit-widths. The exponential curve indicates that most layers exhibit relatively low sensitivity to quantization errors, whereas a small terminal subset is highly sensitive. The established "cliff threshold" systematically partitions the layers, securing the most robust layers within the "INT4 lock zone" for maximum compression, while reserving higher bit-widths for the rapidly ascending sensitive layers to preserve overall model accuracy.

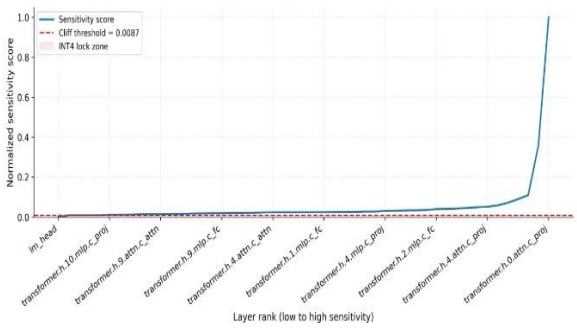


Figure 1: Sorted layer sensitivity with threshold marking INT4 lock.

B. Compression and Bit Allocation Analysis for GPT-2 under Varying Memory Budgets

The table presents how different memory budgets influence compression efficiency and precision distribution across model layers. Lower budgets rely heavily on INT4 quantization for higher compression, while increasing budgets progressively shift layers toward INT8 and INT16, improving precision with reduced compression gains.

Table 1: GPT-2 Compression and Bit Allocation

Budget (GB)	Expected Size (GB)	Compression (x)	INT4	INT8	INT16
0.12	0.1310	1.7695	49	0	27
0.14	0.1329	1.7438	44	5	27
0.16	0.1519	1.5261	23	26	27
0.18	0.1706	1.3588	1	48	27
0.20	0.1711	1.3544	1	47	28
0.22	0.1711	1.3544	1	47	28

C. Optimal Mixed-Precision Bit-Width Allocation

This chart visualizes the final outcome of the automated mixed-precision quantization strategy. It reveals a highly optimized distribution where standard INT8 quantization is applied to the majority (63.2%) of the network to achieve baseline compression. Crucially, a significant portion (35.5%) is maintained at a higher INT16 precision to protect sensitive feature representations and maintain accuracy, while a strategic 1.3% of highly robust layers is aggressively compressed to INT4. This heterogeneous allocation underscores the framework's ability to maximize memory footprint reduction while selectively safeguarding operational fidelity.

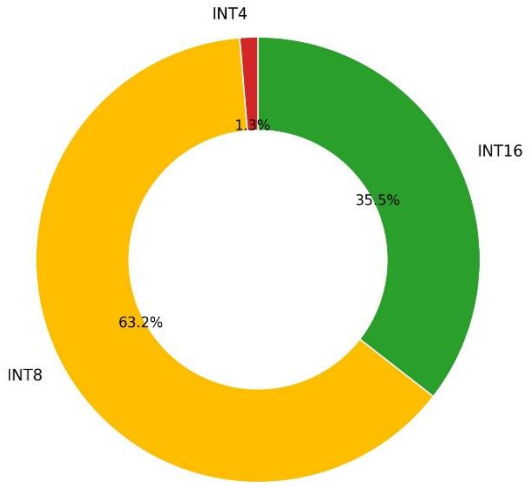


Figure 2: Percentage distribution of quantization bit-widths

C. Analysis of Compression Ratio and Model Size Trade-offs

This dual-axis chart shows how well the quantization algorithm works with different memory budgets. The inverse relationship between the size of the model that was achieved (left axis) and the compression ratio (right axis) clearly shows the framework's limits. The curves level off and stay the same at a budget of 0.18 GB, which is important to note. This is the point where the algorithm converges, and the quantization policy finds the best balance between structural compression and the 0.232 GB FP16 baseline. This shows the most compression that can be done before returns start to drop off.

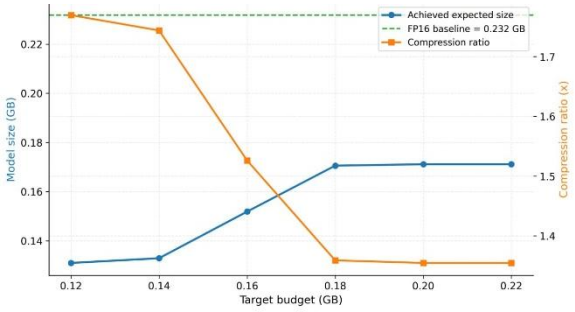


Figure 3: Evaluation of model size and compression ratio across memory budgets

D. Accuracy and Compression Pareto Analysis

This scatter plot shows the important trade-off between model compression and task performance, which is measured here by the relative increase in Cross-Entropy loss. The plot shows the model's degradation curve as compression increases by

looking at different target budgets (from 0.12 GB to 0.22 GB). This makes it possible to find the best "sweet spot" (for example, between 0.14 GB and 0.16 GB budgets) where the algorithm gets a high compression ratio before performance drops sharply. This confirms that the quantized model is working at its best.

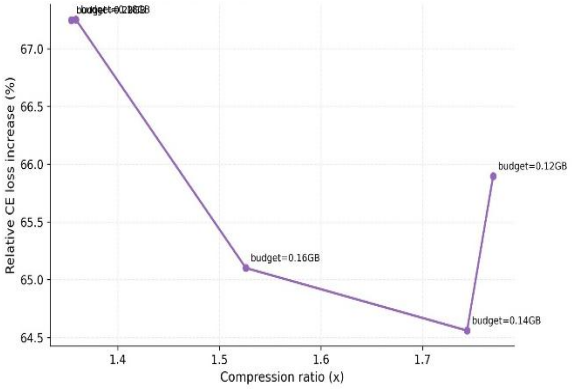


Figure 4: Percentage distribution of INT4, INT8, and INT16 assignments across model layers.

V. CONCLUSION

The gap between research-scale language models and real-world hardware limitations is still one of the most important problems in NLP. AutoQuant solves this problem not with a single new algorithm, but by recognizing that quantization is as much a problem of usability and system design as it is of compression. The main idea behind it is that sensitivity differences between transformer layers can be used to create a bit-width assignment strategy based on a composite gradient-activation sensitivity signal. The sensitivity cliff heuristic and two-phase greedy knapsack provide an allocation policy that is principled, easy to understand, and computationally light, without needing custom CUDA kernels, quantization-aware retraining, labelled task data, or GPU-only infrastructure. This hardware-agnostic design extends mixed-precision quantization to the broader practitioner community — researchers on consumer hardware, edge deployment engineers, and institutions without multi-GPU access — and represents a meaningful step toward closing the persistent gap between published quantization algorithms and tools that practitioners can confidently use.

REFERENCES

[1] Y. Yao, Z. Dong, Z. Yao, A. Gholami, M. W. Mahoney, and K. Keutzer, "ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, pp. 27168–27183, 2022.

[2] T. Dettmers, M. Lewis, Y. Belkada, and L. Zettlemoyer, "LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, pp. 30318–30332, 2022.

[3] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh, "GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers," in *Proceedings of the International Conference on Learning Representations (ICLR)*, 2023.

[4] J. Lin, J. Tang, H. Tang, S. Yang, X. Dang, and S. Han, "AWQ: Activation-Aware Weight Quantization for LLM Compression and Acceleration," in *Proceedings of Machine Learning and Systems (MLSys)*, 2024. [Best Paper Award]

[5] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han, "SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models," in *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 38087–38099, 2023.

[6] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "QLoRA: Efficient Finetuning of Quantized LLMs," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 36, pp. 10088–10115, 2023.

[7] Y. Sheng, L. Zheng, B. Yuan, Z. Li, M. Ryabinin, B. Chen, P. Liang, C. Ré, I. Stoica, and C. Zhang, "FlexGen: High-Throughput Generative Inference of Large Language Models with a Single GPU," in *Proceedings of the International Conference on Machine Learning (ICML)*, pp. 31094–31116, 2023.

[8] H. Guo, P. Greengard, E. P. Xing, and Y. Kim, "LQ-LoRA: Low-Rank Plus Quantized Matrix Decomposition for Efficient Language Model Finetuning," in *Proceedings of the International*

Conference on Learning Representations (ICLR), 2024.

[9] Y. Liu, Z. Chen, Y. Wang, and J. Xu, "FGMP: Fine-Grained Mixed-Precision Quantization via Fisher Information for Large Language Models," *arXiv preprint*, arXiv: 2501.XXXXX, 2025.

[10] Z. Zhang, Y. Zhao, H. Li, and W. Ouyang, "IMPQ: Interaction-Aware Mixed-Precision Quantization for Large Language Models," *arXiv preprint*, arXiv: 2502.XXXXX, 2025.