1. Why does quantization not decrease the accuracy of LLM?

It is a common misconception that quantization *never* decreases accuracy. In reality, it **does** introduce noise, but modern techniques minimize this degradation to negligible levels (often <1% perplexity gap).

The reason LLMs are remarkably robust to quantization lies in their **Over-Parameterization** and **Distributional Properties**:

- **Over-Parameterization (The Lottery Ticket Hypothesis):** LLMs have billions of parameters, many of which are redundant.[1] The model acts as a massive ensemble; small precision errors in individual weights often cancel out across layers, or the network finds alternative paths to process information effectively.
- **Outlier Management:** Research (like *LLM.int8()* or *SpQR*) found that accuracy degradation usually comes from a tiny percentage (<0.1%) of "outlier" weights or activations that have massive magnitudes.
    - *Technique:* If you quantize 99.9% of the weights to 4-bit/8-bit but keep the **outlier features in FP16**, accuracy is almost perfectly preserved.
- **Calibration (AWQ / GPTQ):** Modern quantization isn't just "rounding to the nearest integer."[2] Algorithms like **AWQ (Activation-aware Weight Quantization)** analyze which weights are most critical for activation and preserve their precision (or scale them effectively), sacrificing precision only on "unimportant" weights.[3]
- +1

---

2. What are the techniques by which you can optimize the inference of LLM for higher throughput?

Throughput optimization is about maximizing the number of **Tokens Per Second (TPS)** the system can process, usually by saturating the GPU compute.[4]

- **Continuous Batching (Iteration-level Batching):**
    - *Standard Batching* waits for all sequences in a batch to finish, wasting GPU cycles on padding for shorter sequences.
    - *Continuous Batching* (pioneered by **Orca** and **vLLM**) ejects completed sequences immediately and inserts new requests into the running batch at the iteration level.[5] This drastically increases GPU utilization.
- **PagedAttention (KV Cache Optimization):**
    - The KV Cache grows dynamically and causes memory fragmentation.[6] **vLLM's PagedAttention** manages KV cache like an OS manages virtual memory (pages/blocks).[7] This reduces memory waste, allowing you to fit **larger batches** into the same GPU memory, directly boosting throughput.
    - +1
- **Pipeline Parallelism (PP):**

    - Splits the model layers across different GPUs (e.g., Layers 1-10 on GPU 0, 11-20 on GPU 1). This allows you to process multiple batches simultaneously in a "pipeline" bubble, increasing total system throughput.

---

3. How to accelerate response time of model without attention approximation like group query attention?

If you cannot use approximations (like GQA, Sliding Window, or Sparse Attention) and must perform **exact** inference, you focus on **Memory Bandwidth** and **Algorithmic Efficiency**:

- **Speculative Decoding (The "Free Lunch"):**
    - *Concept:* Use a tiny "draft model" (fast but inaccurate) to generate a sequence of 5 tokens.[8] The main "target model" (slow but accurate) verifies all 5 tokens in a single parallel pass.
    - *Why it accelerates:* If the draft is correct, you generated 5 tokens for the cost of 1 verification step. If incorrect, you discard and correct. This preserves the **exact** output distribution of the main model while reducing latency.
- **FlashAttention (IO-Awareness):**
    - This is **not** an approximation. It is a mathematically exact calculation of Attention.
    - It speeds up inference by optimizing the read/write operations between the GPU's slow HBM (High Bandwidth Memory) and fast SRAM. By fusing operations (tiling), it reduces memory access overhead, which is the main bottleneck in attention.
- **Tensor Parallelism (TP):**
    - Unlike Pipeline Parallelism (which helps throughput), TP splits the *calculation* of a single layer across multiple GPUs (e.g., splitting query/key/value projections).
    - This allows the cluster to calculate one token's embedding faster by combining the compute power of multiple cards, reducing the latency for a single user.
- **Operator Fusion:**
    - Fusing multiple kernels (like `LayerNorm` + `activation`) into a single kernel launch to reduce the overhead of launching operations on the GPU.[9]