1 — Motivation, background, and the hard bits of KV-cache quantization (≈4 min)

Thanks to HaoDong for introducing quantization methods in large language model training. Now, I will talk about the quantization on the KV cache during LLM inference.

During LLM inference, we need to store the keys and values which have already been calculated into the KV cache, to avoid recalculation. However, as batch size and context length grow, the Kv cache increases rapidly and becomes the dominant consumer of memory. And the increasing KV cache also limit the inference speed. Because LLM is auto-regressive, which means every time a new token is generated, the entire Kv cache must be pulled back from GPU main memory into on-chip SRAM, meanwhile leaving the computational cores idle. So the increasing KV cache is becoming the new memory and speed bottleneck.

Therefore, reducing the KV cache size in LLMs while preserving model performance is crucial.

Next i am going to introduce two papers focusing on addressing this problem using quantization for Kv cache.

The first paper is KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache, published on ICML 2024. This paper tries to quantize the kv cache aggressively to a extremely low-bit precision -- only 2 bits,to reduce the memory footprint of kv cache and improve the inference speed.

2 — K I V I: the problem and why earlier tricks failed (≈3 min)

First, let't talk about two different quantization granularity for kv cache. The left one of the plot is per-token quantization, each row of the tensor is a token, per-token quantization quantize each token sequentially along the token dimension. The right is the per-channel quantization, each column of the tensor is a channel, per-channel quantization quantize each channel individually.

The previous works on quantization for kv cache usually use per-token quantization for both key and value caches because of the streaming nature of kv cache. What's the streaming nature of kv cache? KV cache is unlike the weights of large language model. The weights are static, which means the dimension of weights is fixed. In contrast the KV cache is a streaming

data structure. Each time we input a new token to LLM, the corresponding key and value will be appended to the kv cache, keeping the kv cache consistently grow, which means the number of the rows in this tensor will keep growing.

If we use the per-token quantization, every time a new key and value arrives, we can directly quantize them per-token and append the quantized kv entry into the kv cache, without the need to modify the previous quantized kv cache. But if we use the per-channel quantization for kv cache, every time a new key and value arrives, we have to modify previous quantized KV cache along the channel dimension, which introduces a new overhead. So previous works on quantization for kv cache usually use per-token quantization for both the keys and values .

And we can see from the table: With per-token quantization, using INT4 precision generally maintains model accuracy. However, when the precision is further reduced to INT2, there is a noticeable drop in accuracy, which means the previous per-token quantization on keys and values can't be applied to the low-bit precision.

So to address this problem, the paper explores different quantization granularity for keys and values :

From the table,we can observe that:

1. When the value cache is quantized per-channel, accuracy drops significantly, no matter how the key cache is quantized.
2. At the lower precisions INT2, the best accuracy is achieved by quantizing the key cache per-channel and the value cache per-token.

So, why does quantizing the key cache per-channel and the value cache per-token yield the best results at low precision?

And next we will find out the reasons behind it.

3 — Why *keys* get per-channel scales and *values* get per-token scales (≈5 min)

Let's look at the visualization of key and value distribution across different models and different layers.

For the key cache, we notice that certain channels exhibit very large magnitudes which are also called outliers. In the case, if we use the per-token quantization for key cache, the large

outliers dominate the quantization range, leading to large quantization error for other non-outlier channels. So this channel-specific pattern means that if we use per-channel quantization for the key cache, the quantization error is confine to each channel and does not affect the other channels. So using per-channel quantization for keys helps maintain overall accuracy, especially when some channels contain extreme outliers.

Unlike the key cache, the value cache doesn’t have strong outliers in specific channels. So what's the reason it’s better to quantize the value cache per token. Because the attention output is the weighted sum of value cache across various tokens, with the weights being the attention scores. Since the attention score is highly sparse, the output is just the combination of value caches of a few important tokens. The per-token quantization can confine the error to each individual token. Thus, quantizing other tokens does not affect the accuracy of important tokens.

---

4 — K I V I System Design: How Groups and Residuals Work

So based on the previous analysis, the optimal approach is to quantize the key cache per channel and the value cache per token.

However, per-channel quantization for key cache introduces a new challenge. As we mentioned before, when a new kv entry arrives, the per-channel quantization requires modifying the previous quantized kv cache, which does not align well with the streaming nature of the kv cache and introduces a new overhead.

To address this, KIVI proposes a system design to splits the key cache into two parts. The first part is the grouped key cache, which consists of several groups, each group containing a fixed number of keys. We use per-channel quantization to quantize each group to 2 bits individually. The second part is the residual key cache, which includes remaining keys whose number is not enough form a complete group. We keep residual key cache in full precision. And during the decoding phase, we add the newly keys to the residual key cache. Once the number of residual key cache can form a complete group, we quantize the residual key cache and move them to the grouped key cache.

Through this design, we can address the conflict between per-channel quantization and the streaming nature of key cache, avoiding modifying the previous kv cache in every iteration.

The value cache uses a similar two-part approach. Most of the value cache is divided into groups and quantized per token to 2 bits, saving memory. However, the most recent R tokens are kept in full precision as a "residual window." ,which is like a sliding windows. As new tokens are generated, this window slides forward: older tokens in the window are quantized and moved to the grouped cache, while the newest tokens stay in full precision. This way, KIVI keeps the most important, recent value information accurate , while still compressing the rest of the cache to save memory and speed up inference.

---

5 — K I V I Results: Key Findings

The left experimental results show that the “2-bit (key per-channel, value per-token)” configuration consistently delivers the best performance among all tested quantization schemes, confirming our earlier analysis. However, for challenging generation tasks such as the math-reasoning task GSM8K, simple “2-bit (key per-channel, value per-token)” quantization leads to a significant drop in accuracy compared to full precision.

KIVI address this problem and maintain the model accuracy, which means the residual part is essential for preserving accuracy on difficult tasks like mathematical reasoning.

The right experimental results, KIVI compresses the size of kv cache and achieves up to a 4× increase in batch size and a 2.35× to 3.47× improvement in throughput, all while maintaining similar maximum memory usage.

So KIVI achieves improving the inference speed while maintaining the model performance through well-designed quantization for kv cache.

This is the end of the first paper.

And the second paper we are going to introduce is: **Oaken: Fast and Efficient LLM Serving with Online-Offline Hybrid KV Cache Quantization**, published at ISCA 2025. This work tried to achieves both high accuracy and high performance by co-designing kv cache quantization algorithms and hardware for batched-inference scenarios.

---

6 — Oaken: Problem Statement & Motivation

In real-world deployment scenarios, large language models (LLMs) are typically served using batch inference. This means that instead of processing one request at a time, the system handles multiple requests simultaneously in a batch.

Let’s start by why we need batch inference and what challenges it will brings.

The figure is the utilization of compute core in different inference situations.

The figure(a) is the Single-instance inference,which means processing one request at a time.

- **Single-instance inference:** During prefill, multiple tokens are processed in parallel, keeping computational cores busy and utilized well. But in the generation phase, tokens are generated one by one each time, leaving most computational cores idle whose utilization rate is very low.

To improve core utilization, we increase the batch size which is also called batched inference. The figure(b) is the batched inference meaning LLM serve multiple requests simultaneously.

- **Batched inference:**
  However, during the generation phase, even with batched requests, core utilization improves a little bit but still remains under-utilization and latency increases due to bandwidth contention.

Figure (c) shows GPU core utilization during the generation phase , highlighting that under-utilization mainly results from multi-head attention operations because of the kv cache.

This happens because kv cache from attention layers cannot be shared across different requests, so each request in the attention layer uses only a limited number of compute cores.

While increasing batch size improves hardware utilization, it also worsens the KV cache problem. Each request in a batch maintains its own KV cache, causing memory usage to grow rapidly. This leads to higher latency and greater pressure for memory bandwidth.

Why don’t existing solutions suffice?

So reducing the size of kv cache with quantization is crucial.

But most prior quantization work has concentrated on weight quantization to accelerate LLM inference. However, as shown in Figure a, while the memory usage for model weights remains constant as batch size grows, but the KV cache size increases rapidly and soon dominates total device memory. Figure b further demonstrates that 4-bit weight quantization gains only minimal performance improvement in batched scenarios, whereas 4-bit KV cache quantization delivers much larger gains.

So the KV cache quantization is much more important than weight quantization, especially in batched inference

7 — Three Empirical Observations That Shape Oaken

To further explore the quantization for kv cache, Oaken examines distribution of the KV cache across several LLMs and datasets, which will guide Oaken's quantization approach.

First,figure(a) shows that when we compare KV cache values across different models and decoder layers, we find that the minimum and maximum values can vary a lot—not only just between models, but also among different layers within the same model.

Second,figure(b) shows that when we compare the KV cache value ranges for the Llama2-7B model across different datasets, the value range stays consistent no matter which dataset we use.

Based on the first two observations, Oaken can set quantization factors separately for each model and decoder layer, and these factors are insensitive to the input content. In other words, by analyzing the distribution of each layer offline, we can determine the appropriate thresholds and scaling factors for kv cache quantization without the need for online adjustment during inference. This approach reduces online computation overhead while ensuring quantization accuracy.

Third, as shown in figure (c), there are some outliers in some specific channel, which is similar to the first paper. However, outliers are not all in the specific channels can also be scattered throughout other channels. If we use just one quantization scale for the whole vector or group, these scattered outliers can cause significant accuracy loss.

---

8 — Oaken’s Quantization in Three Concrete Steps

After previous observation, this is the overview of Oaken’s Quantization for kv cache ,trying to maximize KV cache compression while preserving accuracy and minimizing runtime overhead.

Let’s break down Oaken’s quantization method into three key steps.

Step 1: Threshold-Based Online-Offline Hybrid Quantization

LLM quantization methods highlight that both large values and small values near zero can significantly affect model accuracy. Large values, if not handled properly, can dominate the quantization range and introduce substantial errors, while small values are prone to vanishing due to underflow, also leading to increased quantization error.

Oaken addresses these challenges by splitting each per-token KV vector into three groups: the outer group (large outliers), the inner group (small outliers whose value is near zero), and the middle group (inliers, where most values are concentrated). The boundaries for these groups are determined by four thresholds, which are profiled offline by collecting statistics from each decoder layer. This offline profiling eliminates the need for expensive online operations like sorting or top-K selection during inference.

During inference, Oaken uses these pre-computed thresholds to quickly assign each value to its corresponding group. For each group, it calculates the minimum and maximum values and computes the quantization scaling factor for each group online, making the process efficient Quantization is performed per token, focusing only on the newly generated key-value vectors at each step, ensuring both accuracy and speed.

Step 2: Group-Shift Quantization

The first step mitigates quantization accuracy loss by splitting values into three groups: outer, middle,and inner group. However, this approach poses a new challenge. While grouping helps isolate outliers, quantizing them remains challenging due to their wide value range. Previous methods often addressed this by using mixed precision—such as FP16 for outliers and INT4 for inliers—but this approach introduces significant storage overhead (up to 23 bits per outlier) and adds hardware complexity, especially as the proportion of outliers increases. These extra costs make it difficult to efficiently balance accuracy and performance.

Oaken overcomes this with a hardware-friendly group-shift quantization technique. The core

idea is to shift the value range of each group using thresholds obtained from offline profiling, effectively narrowing the spread of values before quantization. For the outer group (large outliers), values above the upper threshold are shifted down, and those below the lower threshold are shifted up. The same approach can be applied to the middle group (inliers) if needed. This shifting concentrates values within a smaller range, making it possible to quantize even outliers to a lower bitwidth (e.g., 5 bits) without significant information loss.

With group-shift quantization, Oaken quantizes the middle group into 4 bits, and both the inner and outer groups into 5 bits. This approach minimizes quantization loss, avoids mixed-precision complexity, and enables efficient, low-bitwidth storage for all groups.

Step 3: Fused Dense-and-Sparse Encoding

To store quantized KV cache values efficiently, Oaken adopts a fused dense-and-sparse encoding scheme. The majority of values—the inliers from the middle group—are stored densely in a matrix, while the outliers (outer and inner groups) are stored sparsely using a coordinate list (COO) format. In this scheme, the positions in the dense matrix that correspond to outliers are zeroed out.

To further reduce storage overhead, Oaken reuses these zeroed elements in the dense matrix. Specifically, four bits of each 5-bit quantized outlier are embedded directly into these zeroed dense entries. The remaining bits—6 index bits (to indicate location), 1 group bit (to distinguish group membership), and 1 sign bit—are stored in the sparse COO format.

This fused approach reduces the storage required for each outlier from 23 bits (in prior methods) to just 8 bits, significantly improving KV cache compression while maintaining memory alignment and hardware efficiency. The memory management unit can efficiently handle both dense and sparse matrices on a page basis, further streamlining access and storage.

In summary, by combining group-shift quantization with fused dense-and-sparse encoding, Oaken achieves aggressive KV cache compression with minimal overhead, enabling fast and scalable LLM inference.

9 — Oaken System Design

Now, let me explain how Oaken’s memory management unit, or MMU, is designed to efficiently handle the quantized KV cache.

Figure 10 shows the MMU's operations. Its main job is to manage both dense and sparse matrices in a page-based manner, which helps optimize bandwidth utilization.

Without this specialized MMU, handling variable-sized sparse matrices would require extra overhead for indexing, reshaping, and other operations, which would slow down the system.

MMU uses two separate management tables—one for dense data and another for sparse data—to track virtual-to-physical memory mappings and transfer sizes at page granularity. Dense inlier values, which are fixed-size and aligned, are stored contiguously to support burst-mode memory access, while sparse outlier values, which vary in size, are stored using a compact COO format along with indexing metadata. The MMU organizes KV cache per token, layer, and attention head, enabling efficient sequential writes and burst-mode reads across tokens. This architecture maximizes bandwidth utilization, reduces fragmentation, and supports low-latency memory access crucial for high-throughput batched inference.

---

10 — Oaken Results

- **Throughput**: Oaken achieves up to **1.58× improvement** over A100 GPUs and outperforms other quantized accelerators like QServe and Tender.
- **Accuracy**: Only **0.87% average loss**, outperforming QServe and matching or exceeding Tender and Atom—while using a lower effective bitwidth.

Oaken strikes a new balance between capacity and bandwidth, enabling fast, accurate, and scalable LLM serving under real-world constraints.
