
# DEVELOPERWEEK 2025 HACKATHON

## Pliops Challenge # 2

# GPU Memory Enhancer

By

Rini Susan V S

# Hackathon Challenge

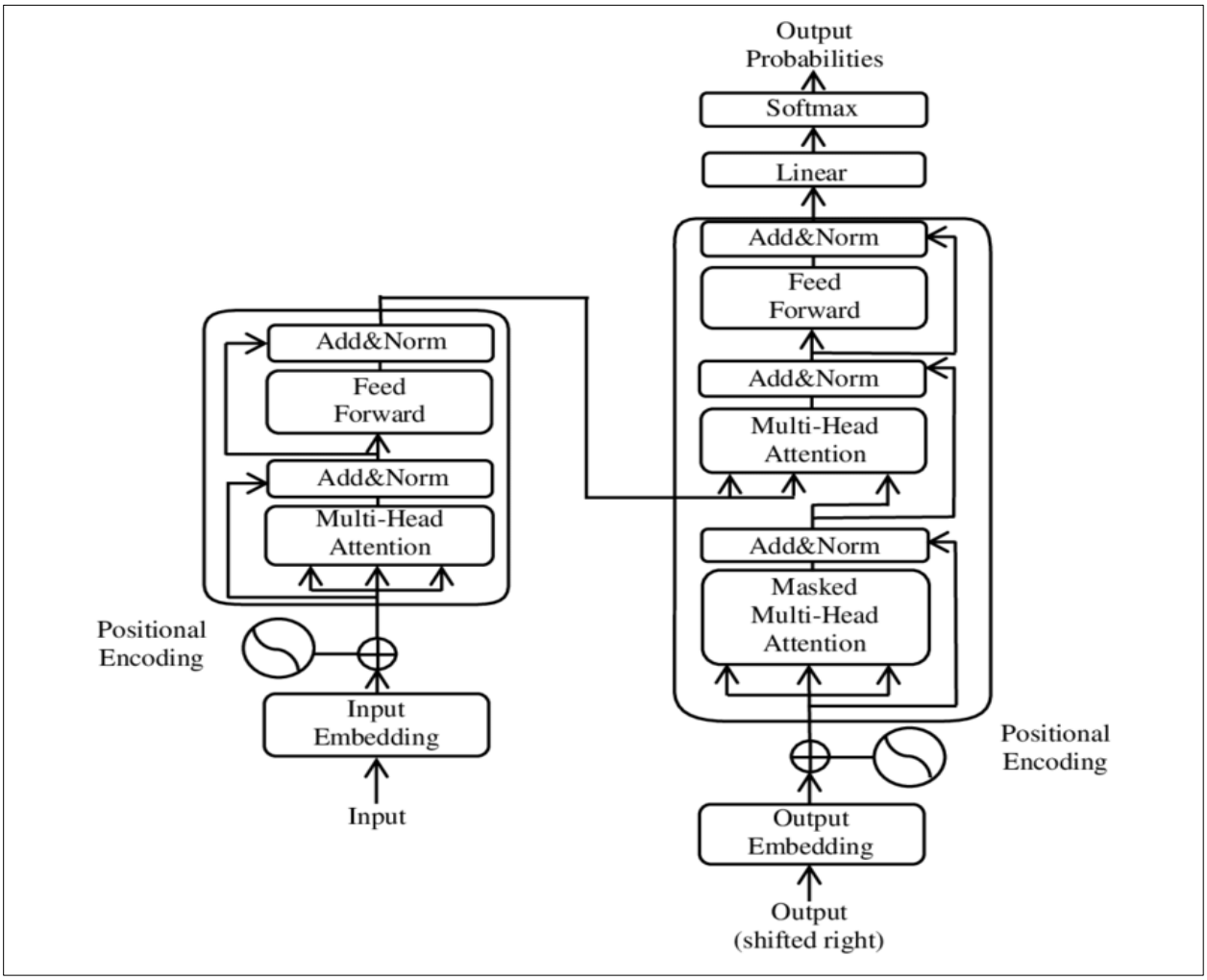
**Pliops Challenge # 2 - A better vLLM framework for Inferencing**

Look for and identify solutions within the vLLM community (source) to address the ever-expanding problem of the very high cost of GPU compute for having to recompute the previous prompts.

# Background

## Large Language Models

Large language models, also known as LLMs, are very large deep learning models that are pre-trained on vast amounts of data. The underlying transformer is a set of neural networks that consist of an encoder and a decoder with self-attention capabilities. The encoder and decoder extract meanings from a sequence of text and understand the relationships between words and phrases in it.



- vLLM

vLLM is a fast and easy-to-use library for LLM inference and serving. vLLM utilizes PagedAttention, a new attention algorithm that effectively manages attention keys and values. This algorithm is inspired by the classic idea of virtual memory and paging in operating systems.

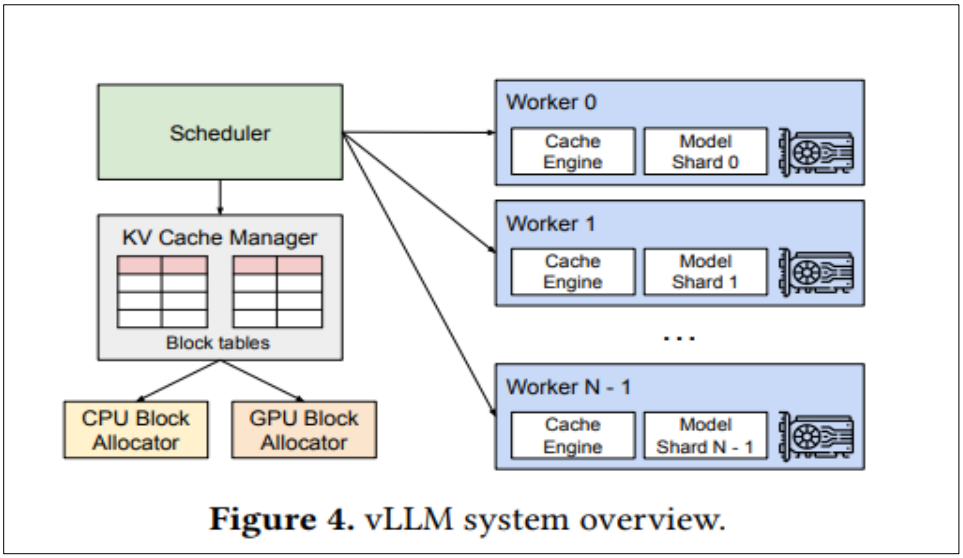



**Figure 4.** vLLM system overview.

Unlike the traditional attention algorithms, PagedAttention stores continuous keys and values in non-contiguous memory space. The contiguous logical blocks of a sequence are mapped to non-contiguous physical blocks via a block table. The physical blocks are allocated on demand as new tokens are generated.

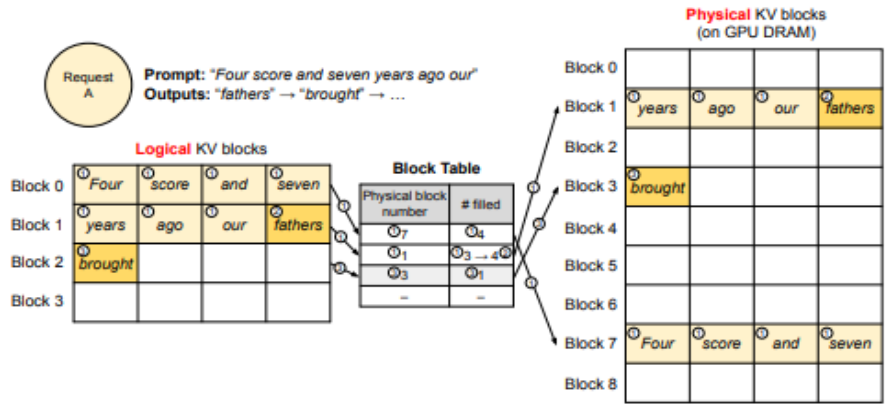
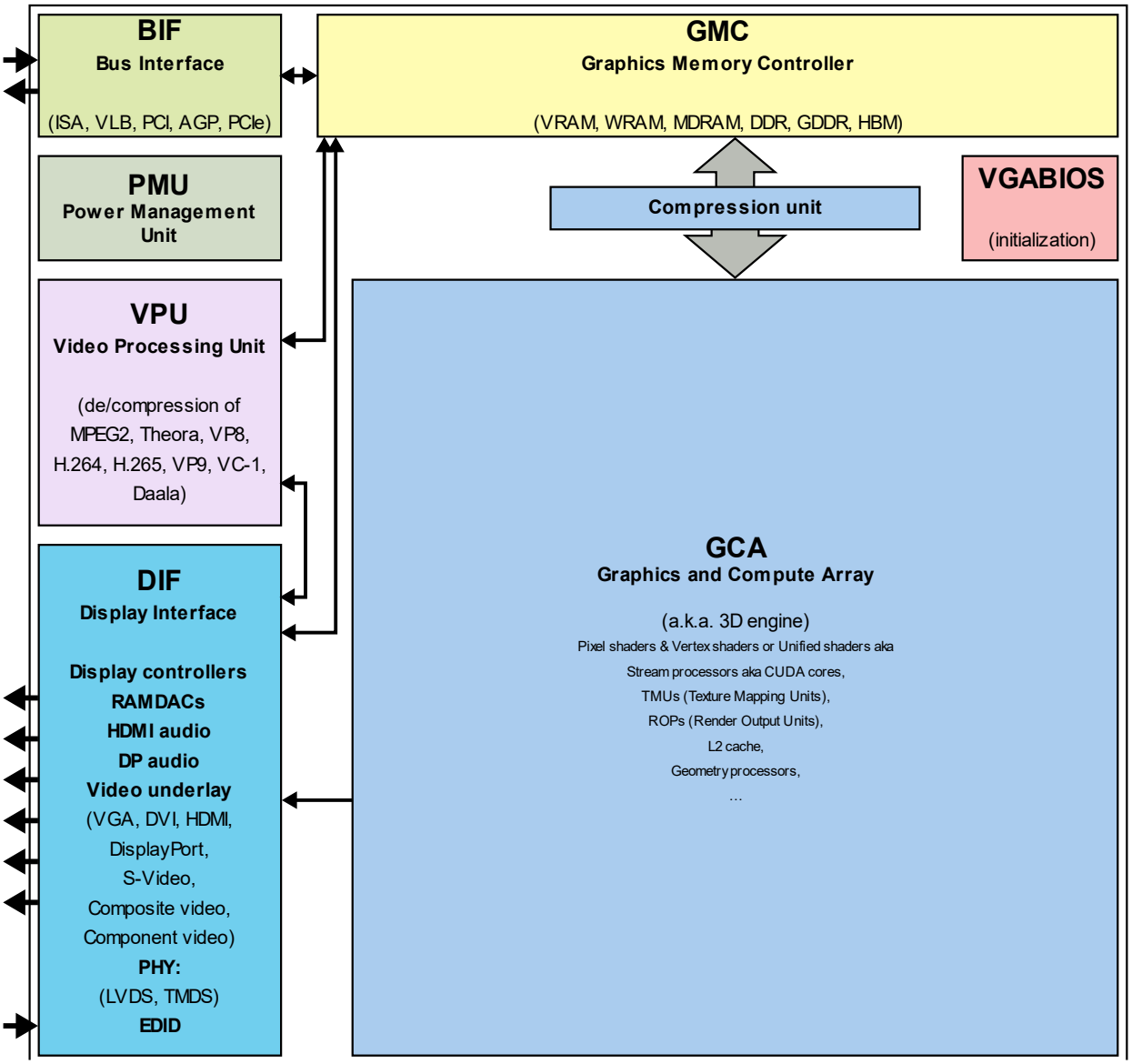



**Figure 6.** Block table translation in vLLM.

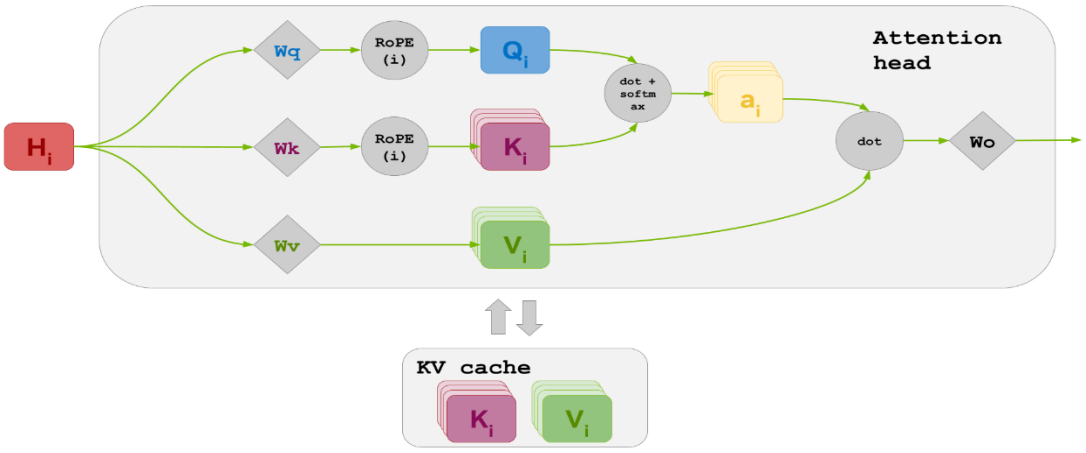
PagedAttention also enables memory sharing through its block table; different sequences in PagedAttention can share the blocks by mapping their logical blocks to the same physical block.

## GPU

A graphics processing unit (GPU) is an electronic circuit that can perform mathematical calculations at high speed. Computing tasks like graphics rendering, machine learning (ML), and video editing require the application of similar mathematical operations on a large dataset. A GPU's design allows it to perform the same operation on multiple data values in parallel. This increases its processing efficiency for many compute-intensive tasks.



- KV Cache




In autoregressive models, text generation happens **token by token**, with each prediction relying on all preceding tokens for context. This repetitive computation becomes inefficient as the sequence grows, especially for large models. KV Cache optimizes this process by storing the intermediate results—keys (K) and values (V)—from the attention layers, so the model can reuse them for future tokens instead of recalculating them.

The LLM attention inference-time workflow involves two phases:

i) the **prefill phase**, where the input prompt is used to generate KV cache for each transformer layer of LLMs;

ii) the **decoding phase**, where the model uses and updates KV cache to generate the next token, one at a time.

## KV Cache Scalability Problem

As powerful as the KV Cache is, it comes with a major drawback - it scales linearly with the size of the context window. There are times when KV cache space is insufficient to handle all batched requests. The values stored in the KV Cache come from all the attention blocks used by the model. The memory consumed by the KV Cache is determined by the following equation:

Size(KV)=2×precision×nlayers×nheads×d×ntokensSize(KV)=2×precision×*nlayers*×*nheads*×*d*×*ntokens*

Each of these factors contributes to the explosion in memory usage. Thus KV Cache is both a critical enabler and a significant bottleneck for deploying large language models (LLMs) with long context windows.

The vLLM can preempt requests to free up KV cache space for other requests. Preempted requests are recomputed when sufficient KV cache space becomes available again. While this mechanism ensures system robustness, preemption and recomputation can adversely affect end-to-end latency.

## Possible Solution

**KV Cache Quantization** reduces memory usage for long-context text generation in LLMs with minimal impact on quality, offering customizable trade-offs between memory efficiency and generation speed. By compressing the KV cache into a more compact form, a considerable amount of memory can be saved and used to run longer context generation on consumer GPUs. Key-value cache quantization in Transformers was largely inspired by the KIVI: A Tuning-Free Asymmetric 2bit Quantization for kv Cache paper.

### 1. KIVI – A Tuning-Free Asymmetric 2bit Quantization for KV Cache

#### ➢ Introduction

Quantization is a straightforward and effective solution to reduce KV cache size, which decreases the total bytes taken by KV cache. A tuning-free 2-bit KV cache quantization algorithm named KIVI quantizes the key cache per channel and the value cache, per token.

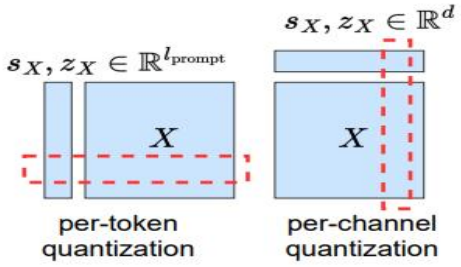



Figure 1: Definition of per-token and per-channel quantization. $\boldsymbol{X} \in \mathbb{R}^{l_{\text{prompt}} \times d}$ is key/value cache where $l_{\text{prompt}}$ is the number of tokens and $d$ is the number of channels. $z_X$ is the zero-point, $s_X$ is the scaling factor.

It was observed that quantizing the key cache per-channel and the value cache per-token to 2bit results in a very small accuracy drop. Also, in the key cache, some fixed channels exhibited very large magnitudes, whereas in the value cache, there was no significant pattern for outliers. The persistence of outliers within each channel means that per-channel quantization can confine the quantization error to each channel without impacting the other normal channels.

#### ➢ Algorithm

The key and value cache of newly generated tokens arrive sequentially. From the implementation perspective, per-token quantization aligns well with streaming settings, allowing newly quantized tensors to be directly appended to the existing quantized value cache by token dimension. For per-channel quantization, the quantization process spans across different tokens, which cannot be directly implemented in the streaming setting.

As shown in Figure 3, the key idea to solve this problem is to group the key cache, every G tokens, and quantize them separately. Because the number of tokens in XK can be arbitrary, we split XK into two parts, namely, the grouped part XKg = XK[: l − r] and residual part XKr = XK[l − r :], where l is the number of tokens inside the current key cache XK, r is the number of residual tokens, where l − r can be divisible by G.

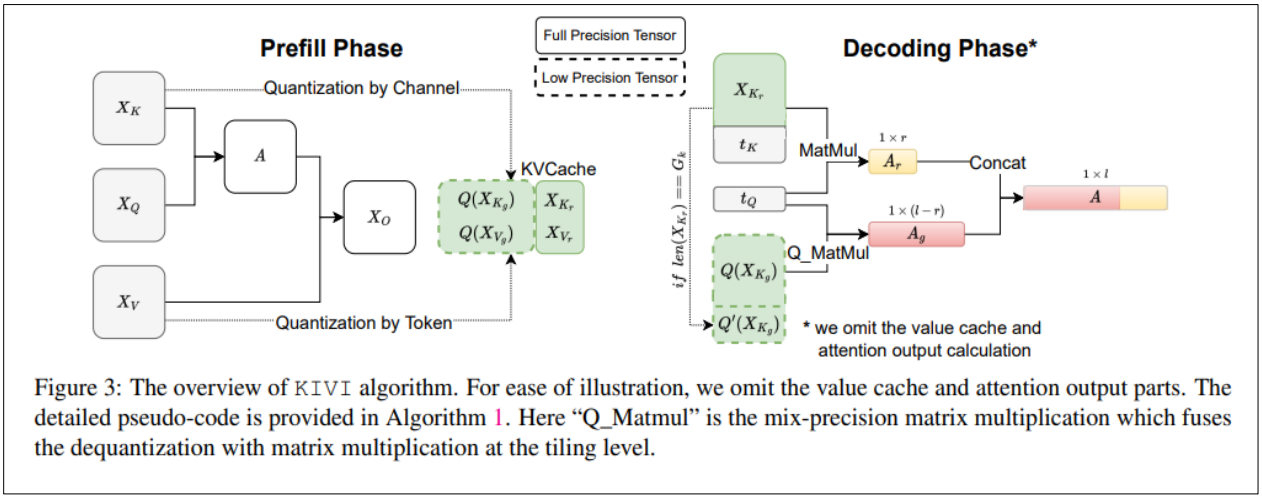
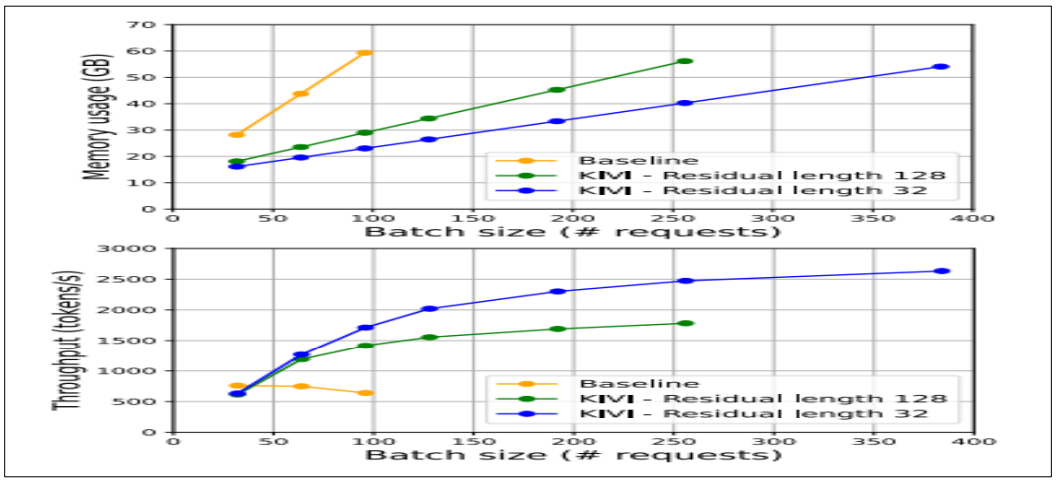



Figure 3: The overview of KIVI algorithm. For ease of illustration, we omit the value cache and attention output parts. The detailed pseudo-code is provided in Algorithm 1. Here "Q_Matmul" is the mix-precision matrix multiplication which fuses the dequantization with matrix multiplication at the tiling level.

For the value cache, similar to the key cache, it is split into two parts, and the most recent value cache is kept in full precision, namely, XVg and XVr. A queue is maintained, and each newly arrived value cache is pushed into the queue. Once the queue reaches the predefined residual length R, the most outdated value cache is poped. Then the poped value cache is quantized per-token and concatenated with the previously quantized value cache along the token dimension.

## ➢ Memory usage and throughput comparison




As shown in the above figure, with similar maximum memory usage, KIVI enables up to 4x larger batch size and gives 2.35x ~ 3.47x larger throughput. This throughput number can grow larger with longer context length and output length.

## Conclusion

By quantizing the KV cache into lower precision formats, memory usage is significantly reduced, allowing for longer text generations without encountering memory constraints. This can be achieved using the KIVI algorithm, where the key cache is quantized per channel and the value cache is quantized per token. In real LLM workload, KIVI allows up to 4x larger batch sizes and 3.47x throughput.

vLLM is a system-level work, which includes memory management through the use of PagedAttention or memory usage prediction. It can lower the memory requirements of the KV cache and simultaneously increase model throughput. The research direction of vLLM is orthogonal to KIVI research since system-level optimizations can also be applied to the KIVI algorithm.

Few more interesting research articles that focus on KV Cache optimization include:

- AsymKV: Enabling 1-Bit Quantization of KV Cache with Layer-Wise Asymmetric Quantization Configurations
- KVPress: A toolkit for KV Cache Compression

## References


LLM : https://aws.amazon.com/what-is/large-language-model/

https://en.wikipedia.org/wiki/Large_language_model

vLLM : https://arxiv.org/pdf/2309.06180

https://docs.vllm.ai/en/latest/performance/optimization.html

GPU : https://en.wikipedia.org/wiki/Graphics_processing_unit

https://aws.amazon.com/what-is/gpu/

KVCache : Mastering Long Contexts in LLMs with KVPress

https://huggingface.co/blog/kv-cache-quantization

KIVI : https://arxiv.org/pdf/2402.02750

KVPress: https://huggingface.co/blog/nvidia/kvpress