Approximate Attention for Large Language Models using Post-Training KV Cache Compression

Candidate No. 1093059

Trinity Term, 2025

Submitted in partial completion of the

Master of Science in Advanced Computer Science

Word Count : 15549

Contents

List of abbreviations

- KV Cache – Key-Value Cache
- MLP – Multi-Layer Perceptron
- GPU – Graphics Processing Unit
- OOD – Out-Of-Distribution
- HBM – High Bandwidth Memory
- RAM – Random Access Memory
- SRAM – Static Random Access Memory
- LN – Layer Normalization
- GQA – Grouped Query Attention
- PTQ – Post-Training Quantization
- LLM – Large Language Model
- XB – X Billion (parameters)

List of Figures

List of Tables

Abstract

In recent years, Large Language Models (LLMs) have developed the ability to accurately process long corpora of text, for some going up to hundreds of thousands of tokens. Such long contexts stress today's hardware : for the generation of each token, attention layers must load the cached intermediate representations of all past tokens, also known as the KV (Key-Value) Cache. This translates to high memory traffic between GPU RAM and on-chip memory, making attention and LLM inference **memory-bandwidth bound**. To address this bottleneck, a recent line of work ("KV Cache Quantization") aims to compress the KV Cache, usually stored in 16 bit, to sub-4-bit precision. Our thesis contributes to this direction by introducing the low-precision compression framework "TemporalKV". Through an information-theoretic study and gradient-based error analysis, we empirically found that the representations of adjacent tokens in attention layers are highly correlated ("Temporal correlations"), opening an opportunity to compress them **jointly**. Based on these findings, and drawing upon previous research, we designed an optimized compression procedure efficiently integrating in modern LLM pipelines. Extensive experiments on the three latest LLaMA models reveal that TemporalKV outperforms state-of-the-art compression baselines across a wide range of settings, including perplexity evaluation and the long-context RULER benchmark. Our code is available at https://anonymous.4open.science/r/TemporalKV-7F6D.

Acknowledgements

I would like to express my deepest gratitude to my supervisors—Dr. Federico Barbero, Prof. Mike Giles, and Prof. Michael Bronstein—for their continuous support, guidance, and encouragement throughout my research and the writing of this thesis. Finding the right research direction and understanding the research posture to work on LLMs has been challenging, hence I want to thank Federico for setting me on the right track and making me understand the obstacles I needed to overcome in order to design an efficient compression framework. I would like to thank Prof. Mike Giles for his support and insights on the memory bottlenecks of attention, which guided the development of TemporalKV. Finally, I am deeply grateful to my friends in Oxford who supported me at times when my research did not yield the expected results. The great environment I was fortunate to be in inspired me to push forward and remain determined throughout the thesis.

Preface

In this thesis, we focus on a research field that has recently received considerable attention: Large Language Models. Ever since the landmark release of ChatGPT, the amount of work and funding dedicated to this topic has increased dramatically. Frontier-scale training is largely concentrated in industry due to compute costs, while a lot of academic work is focused on post-training experimental analysis or theoretical studies. Consequently, advancing the state-of-the-art in LLM research is a challenge.

However, LLM compression is an important line of work in which I wanted to be involved. Aiming for a career in AI research, I am also particularly concerned about the increasing energy footprint of AI systems. From a geopolitical perspective, as states and firms race to develop frontier models, the environmental impact of these technologies risks being sidelined. Therefore, I aspired to contribute at my scale to the energy efficiency of LLMs.

It took us time to find a precise topic where we could have a meaningful contribution at the start of the project. I was initially working on Sparse Attention Methods, and had adopted a theory-based approach to tackle this subject. While I managed to obtain some theoretical results, the assumptions made were often too optimistic, and did not align with the experimental reality of Large Language Models. In practice, my initial efforts did not lead to a competitive LLM compression method. For indicative purpose, we provide a brief description of this initial research direction in Appendix F, but we highlight that it is unrelated to the core subject of this thesis, and hence largely optional for the reader.

Following the advice of my co-supervisor Federico Barbero, we decided to shift towards a more empirical posture, relying on experimental observations. While LLMs can be analyzed theoretically to some extent, the literature on compression is particularly experimental, which motivated our new research orientation.

I analyzed the inner workings of LLMs for some time and ended up finding surprisingly

important correlations between adjacent tokens in attention layers. After reviewing the literature, we did not find any work studying this phenomenon or using it for compression purpose. We thus thought about exploiting it in a compression framework which would compress adjacent tokens jointly. This led us to the design of TemporalKV, a low-bit compression method which we present in this thesis.

1 Introduction

In recent years, Large language models (LLMs) have reshaped the field of natural language processing, powering applications such as chat bots, code generation, document retrieval, and scientific discovery. Their success is based on the Transformer architecture, and more precisely the self-attention mechanism which allows for powerful context modeling. However, this expressiveness comes with substantial computational costs: self-attention scales quadratically with input length, and requires substantial memory traffic from GPU RAM making inference over long sequences both time and energy-consuming. As LLMs are increasingly used for long-context tasks - summarizing day-long meetings, answering questions about entire codebases or reasoning over multi-document folders - the global energy cost of running AI devices around the world will increase sharply in the next few years. According to projections published by the Lawrence Berkeley National Laboratory in December [40], by 2028 AI in the US could consume as much electricity as 22 % of all households. Much of this consumption will likely stem from self-attention layers, which today represent the main bottleneck in common AI architectures.

To address the quadratic complexity of attention, modern LLMs use a key-value (KV) cache, storing intermediate attention representations from past tokens. By reusing these cached keys and values during text generation, models avoid recomputing attention over the full initial context at every step. While this greatly reduces computation, there remains an important bottleneck: the KV cache itself grows linearly with the sequence length, becoming prohibitively large in long-context scenarios (See Figure 1). In practice, the memory footprint of the KV cache can exceed that of the model's parameters, and at every token generation step the whole KV Cache must be loaded from GPU RAM (HBM) to on-chip memory (SRAM). These frequent reads make self-attention memory-bandwidth bound in long context scenarios, leaving compute units idle and degrading GPU utilization. This situation will get worse over time, as computational speeds progress faster than memory speeds. [15].

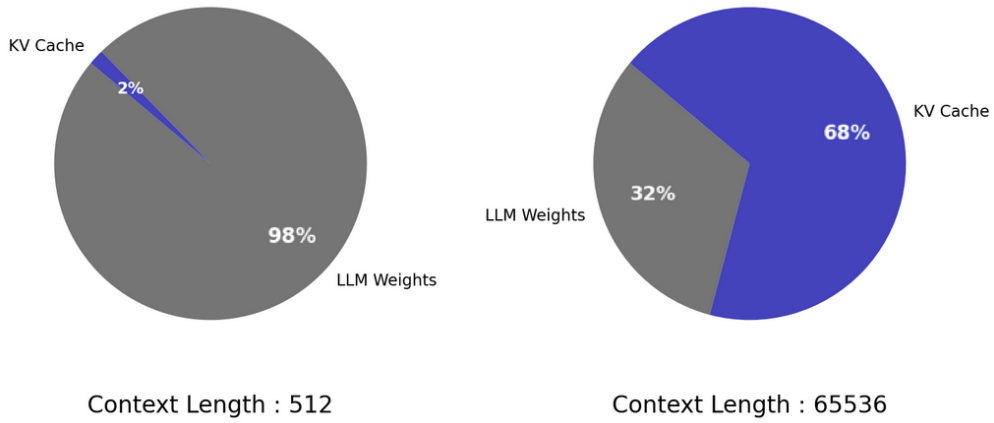


Figure 1: Comparison of memory bottlenecks depending on the context length, for a LLaMA 3.1 8B model with a batch size of 4. For long context (65536 tokens), the KV-Cache becomes the main memory bottleneck.

A promising way to alleviate this bottleneck is quantization—representing tensors with fewer bits to reduce storage and bandwidth requirements. While weight and activation quantization have been extensively studied in the literature, the quantization of the KV cache remains surprisingly underexplored considering the major lever it represents to improve the energy and memory efficiency of modern LLMs. Compressing the KV-Cache can allow to process much longer text sequences for a same RAM capacity. However, because it is dynamic, frequently accessed, and highly sensitive to precision loss, its quantization is non-trivial. In our thesis, we focus on post-training quantization, compressing the KV-Cache of models once they are already trained, without modifying the training pipeline. We make this choice because of the prohibitive cost of training LLMs with academic resources.

First works on post-training KV-Cache compression used uniform quantization (ATOM [52], KIVI [29]), normalizing tensors to the range $[0, 2^{n_{bits}} - 1]$ and then encoding their integer part using n_{bits}. While they have improved upon traditional datatypes using specific normalization schemes, these works have been greatly outperformed recently by non-uniform quantization schemes. Notably, KVQuant [17] made important progress when it used K-Means clustering on float values to determine a non-uniform datatype optimized for the data distribution of

floats in the KV Cache. Elaborating on KVQuant, ChannelKV [50] used coupled quantization, compressing multiple floats together instead of one by one like KVQuant. ChannelKV's framework slightly improved on KVQuant's performances by exploiting the correlations between adjacent channels in the KV-Cache. However, while ChannelKV is especially efficient on a few early layers of LLMs which have low-rank structures, it falls short of truly improving KVQuant for most layers of LLMs.

To address this limitation, in this work we introduce TemporalKV, a KV-Cache compression framework which improves on previous non-uniform KV-Cache quantization methods by leveraging "temporal" correlations, i.e. correlations between adjacent tokens. TemporalKV is based on the observation that adjacent tokens in an LLM often have very correlated Key/Value vectors, however these are compressed independently in current compression methods. We believe that this is an important inefficiency, as a joint compression would leverage these correlations to avoid encoding the same information multiple times. Therefore, in this thesis we aim to answer the following question :

How can correlations between adjacent tokens be leveraged to perform efficient KV-Cache compression and reduce the energy footprint of modern LLM architectures in long-context settings?

1.1 Contributions

Our thesis makes the following contributions to the field of KV Cache compression :

- We perform an in-depth study of current non-uniform KV Cache compression methods, analyzing their efficiency using information-theoretic tools as well as gradient-based layer-wise error analysis. We compare correlations of adjacent tokens and channels to identify compression opportunities, and find in practice that correlations between adjacent tokens are stronger than correlations between adjacent channels on most layers. By analyzing LLMs' projection matrixes, we provide an analysis of why ChannelKV

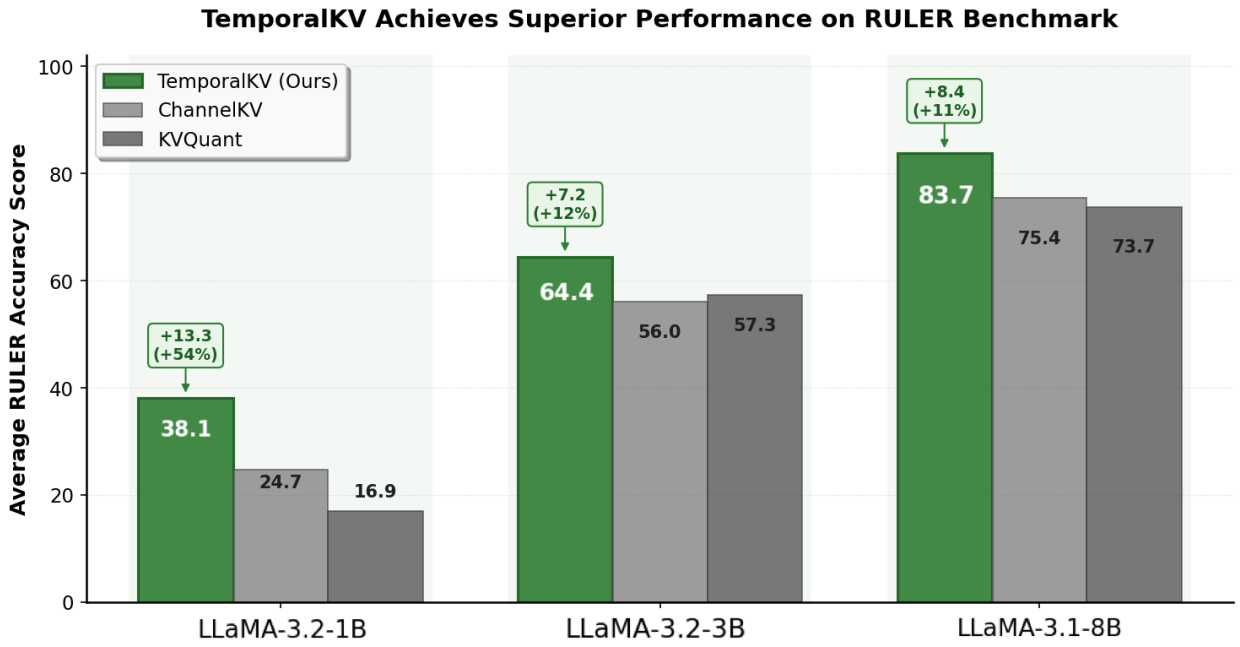


Figure 2: Comparison of TemporalKV to the baselines ChannelKV and KVQuant on the RULER benchmark (average over 11 tasks), for 2 bits per float precision. TemporalKV consistently outperforms existing KV cache compression methods across all model sizes. Numbers show absolute improvements and percentage gains over the best baseline.

has limited efficiency across the majority of layers.

- We introduce a new robust compression framework, TemporalKV, which uses an efficient KV-Cache chunking method in order to compress it while leveraging correlations between adjacent tokens ("temporal correlations"). Building upon previous research, we include and implement several options such as outlier removal, gradient-weighted compression, hybrid early layers compression. We validate their efficiency through an array of ablation experiments.

- We run a wide benchmark of compression methods on 3 new LLaMA models and several different datasets, measuring perplexity on classic datasets and accuracy in challenging long-context settings. We find that our method TemporalKV outperforms the baselines on most tasks (see Figure 2), validating our framework and further pushing low-precision KV cache compression efficiency.

1.2 Outline

We present our thesis using the following structure :

- In the **Background** section, we introduce fundamental notions guiding the design of our method. We detail the Large Language Models architecture as well as the equations of self-attention in order to precisely describe the role of the KV Cache. We also introduce compression tools based on the K-Means clustering algorithm, as well as information-theoretic notions that will later support our preliminary experiments.
- In the **Related Work** section, we broadly review the literature around attention optimization and compression. We describe current research trends and situate our framework TemporalKV in relation to prior work.
- In the **Preliminary Experiments** section, we conduct different experiments guiding the design of our method TemporalKV. Among these, we use information-theoretic tools to analyze the compression efficiency of methods in the literature.
- We precisely describe our method TemporalKV in the **Proposed Method** section. We detail the compression scheme and include ablation experiments justifying our choices.
- We finally compare TemporalKV to state-of-the-art baselines in the **Experiments** section, measuring perplexity and accuracy on a wide array of tasks.

Our code is available at : https://anonymous.4open.science/r/TemporalKV-7F6D

2 Background

In this part, we examine the tools and concepts underpinning the research developed in this thesis. On the one side, we delve into the foundations of modern LLM architectures, focusing on the attention mechanism. On the other side, we introduce clustering algorithms as the basis for our compression method.

2.1 Large Language Models architecture

Natural Language Processing, which focuses on the computational analysis and generation of human language, has been revolutionized by LLMs in the past years. Many prior methods (RNNs, n-gram models, CNNs) have been replaced by these centralized general-purpose models capable of few-shot or zero-shot learning across a wide range of tasks. In this subsection, we decompose the different blocks of an LLM, choosing to focus on inference rather than training because our thesis primarily studies **post-training** quantisation.

2.1.1 Tokenization and Embedding

The first step of an LLM pipeline consists in converting a sentence in natural language to a sequence of token embeddings. This process, known as "tokenization", converts natural language sentences into "tokens" of generally 3-7 letters, a granularity half-way between letters and words. This middle-ground is more computationally efficient, as it allows for a great flexibility and adaptation to different languages while limiting the length of sequences compared to letters.

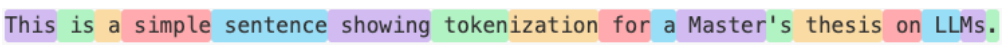


Figure 3: Tokenization example using the text-davinci-001 model. This sentence is broken down into tokens which have lengths from 1 to 9 characters. Note that some common words are directly encoded as tokens, which is often the case for tokenizers with high vocabulary size.

The set of used tokens $\mathcal{V}$ is chosen on a calibration set using algorithms such as Byte Pair Encoding, which determines optimal tokens based on letter adjacency frequencies. The number of tokens is limited to $|\mathcal{V}|$ called "vocabulary size", of 128,000 in the case of a LLaMA 3.1 8B model. Each of these tokens are then mapped to a trained embedding using a lookup table[1], resulting in a sequence of embeddings $X_0 \in \mathbb{R}^{n \times d}$.

2.1.2 Transformer architecture

The sequence of embeddings is then fed to a sequence of **transformer blocks**, as shown in Figure 4. Each transformer block consists of a Multi-Head self-attention layer [43] followed by an MLP, with two residual connections and a normalization in between. The position of the normalization component may however vary depending on the architecture (Pre-LN, Post-LN).

Algorithm 1 Transformer Block as in the original Attention paper [43]

Input: $X_i \in \mathbb{R}^{n \times d}$
Output: $X_{i+1} \in \mathbb{R}^{n \times d}$
$Y_i \leftarrow \text{MultiHeadSelfAtt}(X_i)$
$U_i \leftarrow \text{LayerNorm}(X_i + Y_i)$
$M_i \leftarrow \text{MLP}(U_i)$
$X_{i+1} \leftarrow \text{LayerNorm}(U_i + M_i)$

In decoder-only models[2], these transformer blocks are stacked across many layers (32 for a LLaMA 3.1 8B model), mapping an initial $X_0 \in \mathbb{R}^{n \times d}$ matrix to a final $X_f \in \mathbb{R}^{n \times d}$. These final embeddings are projected to logits $Z \in \mathbb{R}^{n \times |\mathcal{V}|}$, where $|\mathcal{V}|$ is the number of tokens. Each logit corresponds to a token of the vocabulary, and models (after a softmax) the probability of this token being next in the sequence knowing the preceding tokens.

[1] To encode positional information, previous models added "positional embeddings" to token embeddings, however this approach has been replaced by RoPE [41] which we'll cover when introducing attention layers.

[2] We focus on Decoder-only architectures, because they are the most widely used for general-purpose LLMs. However, note that encoder-decoder only are still used for specific tasks (translation, summarization) for which they remain state-of-the-art.

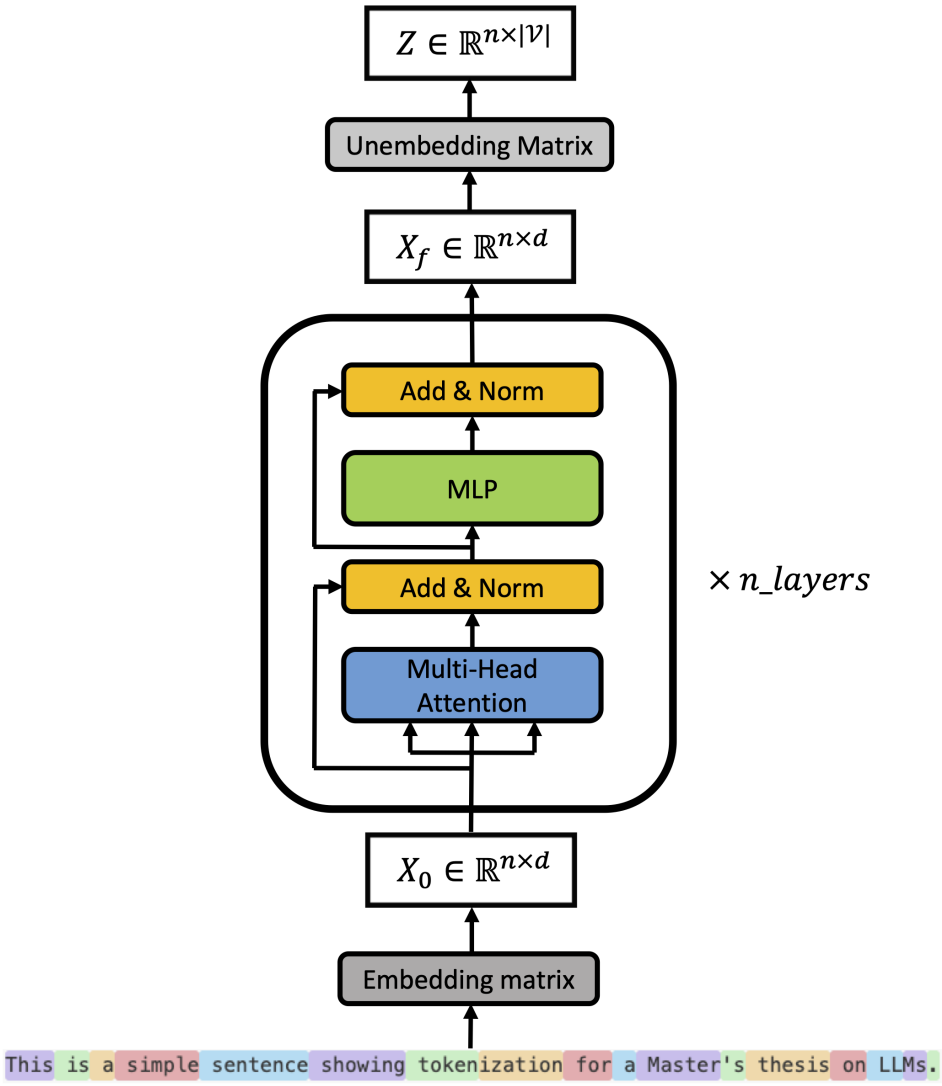


Figure 4: Representation of a Post-Norm Decoder-only architecture. We represent only one Transformer Block, however they are in practice stacked across the number of layers. The LLM maps an initial tokenized sentence to a final logit tensor Z modeling the conditional probability of every next token knowing the tokens before it.

New tokens are generated one by one by sampling from the output probabilities derived from the logits. Note that everytime a new token is generated, a new forward pass must be done from the beginning to compute the next token's logits. However, as we later explain in Subsection 2.2.2, KV Caches allow to significantly reduce the compute cost by avoiding redundant calculations of Keys (K) and Values (V).

2.2 Attention layers

In this subsection, we delve into the main focus of our study by defining self-attention.

2.2.1 Causal scaled dot-product attention

Below, we provide definitions of Rotary Positional Encoding (RoPE) and causal scaled dot-product self-attention, the type of attention used in every layer of LLMs. Note that the equations we provide here are for a single attention head, outputting X_{out} of dimension $n \times d_{head}$. In LLMs, multi-head attention is usually used, and concatenates the output of h independent attention heads in a final output $X_{out,tot}$ of dimension $n \times hd_{head}$. To match the initial dimension, h is typically chosen such that $hd_{head} = d$.

The RoPE function encodes positional information in every key and query of the self-attention module and is defined as :

Definition : Rotary Positional Encoding. Let d_{head} be the dimension of keys and queries in a given head, and let $x \in \mathbb{R}^{d_{head}}$ be a key or query at position $p \in \mathbb{N}$ in the initial sequence.

Define frequencies:

$$\theta_k = 10000^{-2k/d_{head}}, \quad k = 0, 1, \ldots, \tfrac{d_{head}}{2} - 1,$$

and angles:

$$\phi_k = \theta_k \cdot p.$$

Define the 2×2 rotation:

$$R(\phi_k) = \begin{bmatrix} \cos\phi_k & -\sin\phi_k \\ \sin\phi_k & \cos\phi_k \end{bmatrix},$$

and the block-diagonal matrix:

$$R_p = \mathrm{diag}(R(\phi_0), R(\phi_1), \ldots, R(\phi_{d_{head}/2-1})) \in \mathbb{R}^{d_{head} \times d_{head}}.$$

Then:

$$\boxed{\text{RoPE}(x, p) = R_p x}$$

is the Rotary Positional Encoding of x at position p.

Now that we have introduced the required notations and concepts, we can define self-attention, which we represent in Figure 5 :

Definition : Causal scaled dot-product Attention. Given a tokenized sequence of length n, let the embedding matrix be $X \in \mathbb{R}^{n \times d}$ where each row corresponds to the embedding of a token in the sequence.

Within each attention head, let :

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V,$$

where $W_Q, W_K \in \mathbb{R}^{d \times d_{head}}$ and $W_V \in \mathbb{R}^{d \times d_{head}}$ are respectively the query, key and value learnable projection matrices.

We project Keys $K \in \mathbb{R}^{n \times d_{head}}$ and Queries $Q \in \mathbb{R}^{n \times d_{head}}$ using Rotary Positional Encodings (RoPE) :

$$\hat{Q} = \text{RoPE}(Q), \quad \hat{K} = \text{RoPE}(K),$$

The causal scaled dot-product attention at position t is then defined as[3]:

$$\text{Attention}(\hat{q}_t, \hat{K}, V) = \sum_{i=1}^{t} a_i v_i,$$

where

$$a_i = \frac{\exp\left(\hat{q}_t^\top \hat{k}_i / \sqrt{d_{head}}\right)}{\sum_{j=1}^{t} \exp\left(\hat{q}_t^\top \hat{k}_j / \sqrt{d_{head}}\right)}$$

[3]We note $\hat{k}_i = \hat{K}_{i,.}$, $\hat{q}_i = \hat{Q}_{i,.}$, $v_i = V_{i,.}$

Simply put, for a given query and a set of keys (each with their corresponding value), self-attention computes dot-products between the query and each key. These dot-products then determine an "attention weight" for each key, which is used to perform a weighted sum of the value vectors.

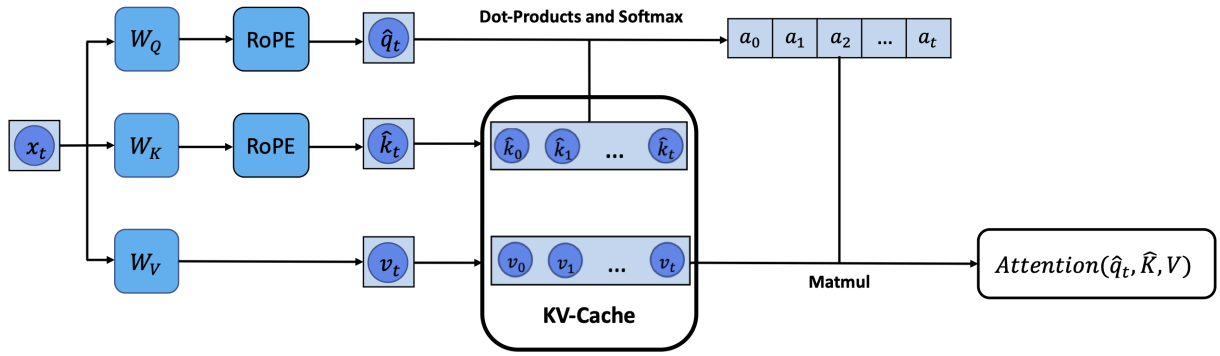


Figure 5: Representation of causal self-attention in the decoding phase (inspired by [25]) Tokens are processed one by one, with the preceding keys and values cached in the KV Cache. The dot-products between the current query and previous keys allow to derive attention weights, which lead to the attention output after a matrix multiplication.

2.2.2 The auto-regressive characteristic and the KV Cache

Note that in the causal self-attention definition, each query $\hat{q}_t$ only attends to its key and keys of previous tokens $\hat{k}_1, \hat{k}_2, ..., \hat{k}_t$. **Therefore, at every layer, any token's embedding only depends on previous tokens, and has no interaction with future tokens.** This is referred to as the **"auto-regressive"** characteristic of LLMs. This essential property allows to encode the order of the text sequence in the architecture. Without it, self-attention would be permutation-invariant[4], i.e. self-attention's output would not depend on the order of the initial text. This is a property we naturally want to avoid when modeling natural language, where order is extremely important.

Due to this auto-regressive nature, for a given layer and head, once they have been first computed, **the key and value of a given token remain the same throughout the**

[4]To be more precise, we would also need to exclude RoPE to make self-attention permutation-invariant

inference process. This allows for what we previously referred to as the **KV Cache**, where we cache every key and value computed (see Figure 5). Thus, everytime we generate a new token, we don't need to recompute the keys and values of previous tokens for every layer and head : we only load them from the KV Cache. However, because keys and values differ between heads and layers, there are $h * n_{layers}$ different KV Caches in a single LLM. For a typical LLaMA 3.1 8B model, this corresponds to $32 * 16 = 512$ KV Caches, which can quickly take up much memory on long sequences (Figure 1 in the Introduction).

2.2.3 The two-phase Inference Process of LLMs

Large Language Models are generative models which, given an input text referred to as *context*, generate new text conditioned on this input. Given the auto-regressive nature we just discussed, the generation process can be divided into two distinct phases:

- Prefilling : In this phase, the LLM processes the whole *context* in one forward pass, and caches the keys and values for all context tokens in every layer and head's KV Cache. Since all context tokens are known in advance, they can be batched together, allowing the GPU to run at high utilization. This phase is compute-bound, with the bottleneck lying either in the MLP computations or in the self-attention operations.

- Decoding : After the LLM has processed the context during the prefilling phase, it generates new tokens sequentially, **one at a time**. Unlike prefilling, each token generation requires a separate forward pass through the model. This is because the forward pass for a future token cannot begin until the preceding token is generated — requiring its output logits and a sampling step. This leads to high under-utilization of the GPU, as it is bottlenecked by the loading of the KV Cache rather than compute [9]. Indeed, we only need to perform computations for the single new token thanks to KV-Caching of previous tokens (see Figure 5). However, self-attention is still highly memory-intensive, due to the need to load full KV Caches from GPU VRAM at each forward pass. This makes this phase memory-bandwidth bound.

In Figure 5, we represented attention during the decoding phase. We highlight that the whole KV Cache has to be loaded from GPU VRAM at every token generation, making attention memory-bound. This paves the way for compression of the KV Cache to reduce this memory overhead.

2.2.4 Attention variants

There are various forms of self-attention, and providing a comprehensive definition of all of them lies beyond the scope of this thesis. Nevertheless, we highlight Grouped Query Attention (GQA) in Figure 6, a variation of multi-head attention which is used in most recent LLMs, including those evaluated in our experiments. It uses less key and value heads while keeping the same number of query heads. As a result, it alleviates the memory bottleneck of the KV Cache while preserving the overall number of heads. This variation requires minimal adaptation for our method compared to standard multi-head attention, and we therefore apply our framework to GQA in the same way.

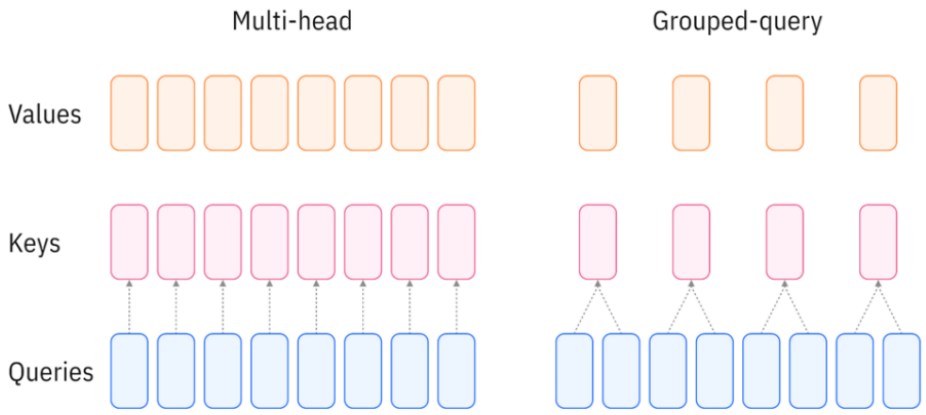


Figure 6: Side-by-side comparison of multi-head attention and grouped query attention (Source : IBM). Rather than having independent heads each with their own queries, keys and values, Grouped Query Attention groups multiple query heads together for one value and key head.

2.3 Vector compression

Due to the memory bottleneck of LLMs, data compression has been employed to reduce the size of the KV Cache. For a given layer and head, the KV Cache (Key-Value Cache) consists of real matrixes K and V of shape $n \times d_{head}$. An encoding function *encode* can be applied to compress these matrixes, in order to use less bits to store them in GPU VRAM and reduce the memory overhead of loading them during self-attention. This compression comes at the cost of accuracy, with a decompressing function *decompress* yielding an approximation of the original matrix. This approximate attention using a compressed KV Cache is depicted in Figure 7. Note that in this figure compression is applied to keys after RoPE, however it is also possible to compress keys before RoPE and apply RoPE on the fly during decompression [17] [50], which we will discuss in Subsection 4.2.

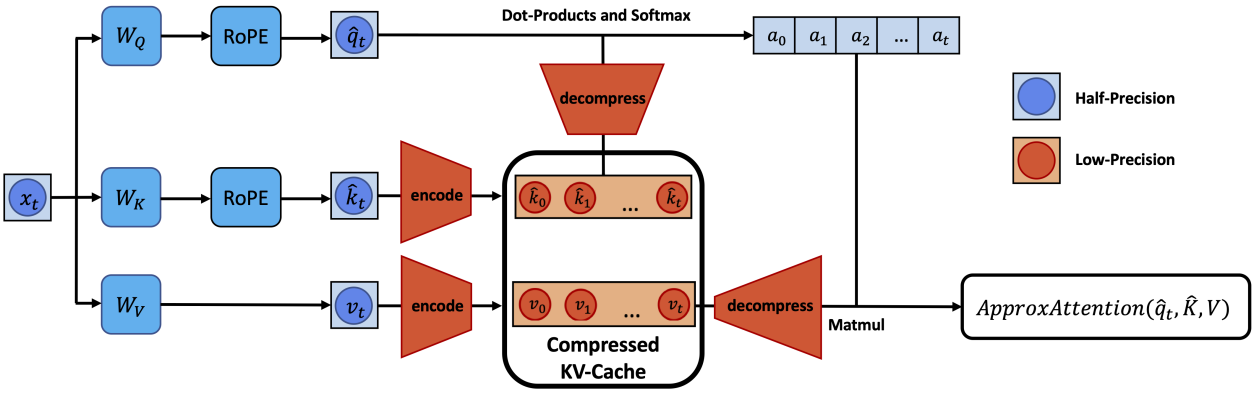


Figure 7: Representation of an attention layer using KV Cache Compression (inspired by [25]). The *encode* component compresses the activations to a low-bit representation, while the *decompress* component decompresses it back to half-precision.

2.3.1 The Compression framework

In this subsection, we introduce generic notations and a simple framework for compression.

Given N vectors $(z_i)_{i=1,\dots,N}$ in $\mathbb{R}^s$ with importance weights $(w_i)_{i=1,\dots,N}$, we wish to compress them into a low-precision code $(encode(z_i))_{i=1,\dots,N}$ with a fixed number of bits n_{bits} per float (hence $s * n_{bits}$ bits to compress each vector), minimizing a loss function :

$$\min_{encode,\, decompress} \sum_{i=1}^{N} w_i \ell \left(z_i, decompress(encode(z_i))\right)$$

This is equivalent to assigning each vector to one of $k = 2^{s*n_{bits}}$ optimized centroids $(c_j)_{j=1,\dots,k}$ in $\mathbb{R}^s$ by minimizing the following loss function :

$$\min_{(c_j)_{j=1,\dots,k},\, assign} \sum_{i=1}^{N} w_i \ell \left(z_i, c_{assign(z_i)}\right)$$

If we use the squared error as the loss function, we then obtain the MSE loss function used in the K-Means algorithm :

$$\min_{(c_j)_{j=1,\dots,k},\, \text{assign}} \sum_{i=1}^{N} w_i \left\| z_i - c_{\text{assign}(z_i)} \right\|_2^2 \tag{1}$$

Thus, compression is reformulated as a basic clustering problem, which can be solved with the weighted K-Means algorithm that we recall in the following (next page) :

Algorithm 2 K-Means Algorithm

Input: Data points $z_1, \dots, z_N \in \mathbb{R}^s$, weights $w_1, \dots, w_N \in \mathbb{R}^+$, number of clusters k
Output: Cluster centres $c_1, \dots, c_k$ and assignments
Randomly initialise cluster centres $c_1, \dots, c_k$ (or use k-means++)
Define $S = \sum_{i=1}^{N} w_i$
repeat
  for $i = 1$ **to** N **do**
    Assign each point to its nearest centre:

$$\text{assign}(z_i) \leftarrow \text{argmin}_{j=1,\dots,k} \|z_i - c_j\|_2^2$$

  for $j = 1$ **to** k **do**
    Update each centre to the weighted mean of its assigned points:

$$c_j \leftarrow \frac{1}{\sum_{i:\text{assign}(z_i)=j} w_i} \sum_{i:\text{assign}(z_i)=j} w_i z_i$$

until *assignments do not change*;

Once a set of centroids is determined using K-Means, each new vector can be encoded by assigning it to the centroid with lowest norm-2 distance. Instead of storing the vector in half precision, we then only need to store the indice of this centroid in $\{1, \dots, k\}$. The decompressing phase then simply consists in retrieving the centroid corresponding to this indice.

Algorithm 3 Encoding step

Input: Data point $z \in \mathbb{R}^s$, centroids $c_1, \dots, c_k \in \mathbb{R}^s$
Output: Indice $i^* \in \{1, \dots, k\}$ of closest centroid to z

$$i^* = argmin_{j \in \{1,\dots,k\}} \|z - c_j\|_2^2$$

Algorithm 3 illustrates the encoding step consisting of an argmin on distances to centroids. The reconstruction of z is then obtained with c_{i^*}.

Note that the assignment step in encoding can be prohibitively long for large k, as the distance must be computed with each of the k vectors. Since lowering k (i.e. lowering

n_{bits} as $k = 2^{s*n_{bits}}$) leads to poorer reconstruction accuracy, a tradeoff is needed between reconstruction accuracy and encoding speed. Hence, literature on compression has found several tricks to optimize this tradeoff, which we'll cover in the next subsection.

2.3.2 Product Quantization

In LLM-related applications, compression usually only goes as low as 1 or 2 bits per float [17] [50] ($n_{bits} \geq 1$). For a vector of dimension 128, this corresponds to a minimum of 128 bits per vector. Using $k = 2^{128} \approx 10^{38}$ centroids is impossible, therefore the basic K-Means algorithm is impossible to use directly on these high-dimensional vectors.

This is where Product Quantization [19] allows to leverage the strength of K-Means while remaining computationally tractable. Instead of directly applying K-Means on the whole vector of dimension s, the vector is divided in m chunks of equal sizes (assuming m divides s) to which we independently apply K-Means. For a budget of 1 bit per float ($n_{bits} = 1$), a chunk of dimension 8 will correspond to 8 bits, i.e. $2^8 = 256$ centroids per chunk, which is reasonable in terms of encoding time and centroid storage.

More formally, instead of one centroid set $\mathcal{C}$ of cardinal $2^{s*n_{bits}}$, we rely on a Cartesian product of centroids $\mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_m$, each with cardinal $|\mathcal{C}_i| = 2^{\frac{s*n_{bits}}{m}}$. Each of these $\mathcal{C}_i$ is determined independently by running K-Means on the corresponding chunk. Then, for a vector z of dimension s, we encode it by dividing it into m chunks of size s/m and encoding each of these chunks separately with its corresponding centroid set.

$$\underbrace{z_1, \ldots, z_{s/m}}_{u_1(z)}, \ldots, \underbrace{z_{s-s/m+1}, \ldots, z_s}_{u_m(z)} \rightarrow \underbrace{encode_1\big(u_1(z)\big)}_{i_1^*}, \ldots, \underbrace{encode_m\big(u_m(z)\big)}_{i_m^*}$$

In the decompressing step, we only have to select the centroids corresponding to the codes of each chunk and concatenate them to reconstruct the decompressed vector.

$$i_1^*, \ldots, i_m^* \rightarrow \left[c_{1,i_1^*}, c_{2,i_2^*}, \ldots, c_{m,i_m^*}\right]$$

Note that $|\mathcal{C}_1 \times \mathcal{C}_2 \times \cdots \times \mathcal{C}_m| = 2^{m \times \frac{sn_{bits}}{m}} = 2^{sn_{bits}} = |\mathcal{C}|$, while the encoding time is drastically lower for Product Quantization. The complexity of encoding the whole vector is $\mathcal{O}(m \times \frac{s}{m} \times 2^{s \times n_{bits}/m}) = \mathcal{O}(s2^{s \times n_{bits}/m})$ for product quantization, against $\mathcal{O}(s2^{s \times n_{bits}})$ for full vector quantization. Product Quantization is thus a computationally tractable alternative to full vector quantization.

2.3.3 Residual Quantization

Another alternative compression scheme, which is a generalization of Product Quantization, is Residual Quantization [14]. In this quantization type, we also use a cartesian product of centroid sets but the vector is not divided into chunks. Instead, we use classic compression on the full vector with a reduced number of centroids then recursively compress the compression error of the previous step as shown in Algorithm 4.

Algorithm 4 Residual Quantization: Assignment Step

Input: Input vector $z \in \mathbb{R}^s$, number of compression steps m, codebooks $\mathcal{C}_1, \ldots, \mathcal{C}_m$, each $\mathcal{C}_l = \{c_{l,1}, \ldots, c_{l,k}\} \subset \mathbb{R}^h$

Output: Assignments $(j_1^*, \ldots, j_m^*)$

Set $r_0 \leftarrow z$

for $l = 1$ **to** m **do**

Find

$$j_l^* \leftarrow \text{argmin}_{j \in \{1,\ldots,k\}} \|r_{l-1} - c_{l,j}\|_2^2$$

Update the residual:

$$r_l \leftarrow r_{l-1} - c_{l,j_l^*}$$

Using the assignments $(j_1^*, \ldots, j_m^*)$, the decompressing step consists in summing corresponding centroids :

$$decompress(j_1^*, \ldots, j_m^*) = \sum_{l=1}^{m} c_{l,j_l^*}$$

For a precision of n_{bits} per float, the encoding complexity of Residual Quantization is higher than Product Quantization but still much lower than full vector quantization, at $\mathcal{O}(sm2^{s \times n_{bits}/m})$. Like Product Quantization, the total number of centroids remains the same.

2.4 Information Theory

In this final subsection, we introduce information theoretic tools which we'll use in our experiments to compare quantization methods.

2.4.1 Entropy

We first provide a generic definition of entropy in the context of Information Theory, introduced by Claude Shannon in his 1948 book "A Mathematical Theory of Communication".

Definition : Entropy. Given a random variable X with values in $\mathcal{X}$, distributed following $p : \mathcal{X} \to \mathbb{R}^+$, the *entropy* of X is defined as :

$$H(X) = -\sum_{x \in \mathcal{X}} p(x) \log_2 p(x)$$

Entropy is the amount of information (in bits) needed to describe the state of X, quantifying the level of uncertainty in X.

Now, if two variables X and Y are correlated, it is possible to encode them together at a lower bit budget than encoding them separately for a same precision. Joint Entropy measures the amount of information needed to encode two random variables together and is defined in the following :

Definition : Joint Entropy. Given two random variables X and Y defined over $\mathcal{X}$ and

$\mathcal{Y}$, with joint distribution $p : \mathcal{X} \times \mathcal{Y} \to \mathbb{R}^+$, the *joint entropy* of (X, Y) is defined as:

$$H(X, Y) = -\sum_{(x,y)\in\mathcal{X}\times\mathcal{Y}} p(x, y) \log_2 p(x, y)$$

The joint entropy measures the total uncertainty associated with the pair (X, Y), i.e. the number of bits required on average to describe the outcome of both X and Y simultaneously.

Thus, the difference $H(X) + H(Y) - H(X, Y)$ encodes the added bit budget of encoding X and Y separately rather than jointly.

2.4.2 Perplexity

In this subsection, we introduce perplexity, a common metric to evaluate LLMs' prediction capabilities on any text. To do so, we first define the Cross-Entropy of two distributions below [5]:

Definition : Cross Entropy. Given two distributions p and q defined over the same $\mathcal{X}$, the *cross entropy from p to q* is

$$H(p, q) = -\sum_{x\in\mathcal{X}} p(x) \log_e q(x).$$

The cross-entropy corresponds to the amount of information needed to encode the state of a random variable following p with a coding scheme optimized for the distribution q.

As represented before in Figure 4, for a given sequence of token ids $K = (k_1, k_2, ...k_n)$, an LLM outputs logits $Z \in \mathbb{R}^{n\times|\mathcal{V}|}$, which are then converted to probabilities $P = softmax(Z) \in \mathbb{R}^{n\times|\mathcal{V}|}$.

P can be seen as a **predicted** probability distribution $\hat{p}$ of the next token knowing the tokens that precede it :

[5] We use the natural logarithm here because it is the one commonly used for LLMs' perplexity measurements

$$\forall (i,j) \in [\![1,n]\!] \times [\![1,|\mathcal{V}|]\!], P_{i,j} = \hat{p}(j|k_1, k_2, ..., k_{i-1})$$

.

Definition : Perplexity. Using the previous formalism, we define perplexity as :

$$\mathrm{PPL}(K) = \exp\left\{ -\frac{1}{n} \sum_{i=1}^{n} \log \hat{p}(k_i \mid k_1, k_2, ..., k_{i-1}) \right\}$$

This perplexity can be interpreted as an "exponentiated averaged negative log-likelihood". It measures how well the LLM predicts each next token over a whole sequence, and can be averaged across whole datasets.

We can also view the negative log-likelihood $-\hat{p}(k_i \mid k_1, k_2, ..., k_{i-1})$ as a cross-entropy between the predicted distribution $\hat{p}(.|k_1, k_2, ..., k_{i-1})$ and the ground truth distribution p_{i-GT} of the constant variable equal to k_i:

$$p_{i-GT}(j) = \begin{cases} 1, & \text{if } j = k_i, \\ 0, & \text{if } j \neq k_i. \end{cases}$$

We can then reformulate perplexity as :

$$\mathrm{PPL}(K) = \exp\left\{ \frac{1}{n} \sum_{i=0}^{n-1} H\big(p_{i\text{-}GT}, \hat{p}(\cdot \mid k_{<i})\big) \right\}$$

This interpretation of perplexity reflects the amount of information needed to encode the ground-truth variable using a coding scheme optimized for the LLM's conditional probabilities.

2.4.3 Effective Rank

In linear algebra, matrix rank is a discrete quantity. Because of this, all floating-point projection matrices in LLMs have full (maximum) rank when measured in the traditional sense. To obtain a more informative measure, we use the effective rank [38] in our experiments. This continuous metric, based on Singular Value Decomposition, captures the effective dimensionality of a matrix.

Definition : Effective Rank. Let A be a matrix with singular values $\{\sigma_1, \sigma_2, \ldots, \sigma_Q\}$. Define the normalized spectrum $p_k = \frac{\sigma_k}{\sum_{j=1}^{Q} \sigma_j}$. The *effective rank* of A, denoted $\text{erank}(A)$, is defined as:

$$\text{erank}(A) = \exp\left(H(p_1, p_2, \ldots, p_Q)\right)$$

where $H(p_1, \ldots, p_Q)$ is the Shannon entropy of the distribution $\{p_k\}_{k=1}^{Q}$, given by:

$$H(p_1, \ldots, p_Q) = -\sum_{k=1}^{Q} p_k \log p_k$$

3 Related Work

Addressing the memory bottleneck of attention has become a central component of LLM work in the past 5 years, in order to find a better tradeoff between computational cost and model accuracy in modern language models. This literature can be divided in three categories : the first consists in deriving new LLM architectures with alternative attention mechanisms, which we cover in Subsection 3.1. Their aim is to outperform vanilla attention by introducing innovative structural changes, training experimental models with these novel mechanisms. We are not able to contribute to this line of work with academic computational resources, due to the prohibitive cost of training. However, we review these methods in our first subsection as they provide important ideas and concepts on attention. The second category of the literature is Post-Training Approximate Attention. This line of work uses approximation methods to compute attention outputs on **already trained** models with minimal computational cost while maintaining reasonable precision. It can be divided into Post-Training Sparse Approximate Attention, which is not the subject of our thesis, and **Post-Training Quantization**, the body of literature our thesis contributes to. We briefly mention Post-Training Sparse Approximate Attention in Subsection 3.2 and provide a detailed review of Post-Training Quantization in Subsection 3.3.

3.1 Alternative attention mechanisms

Alternative attention mechanisms have been developed since the first Attention paper [43] was published in 2017. The aim is to reduce the complexity of attention, which is quadratic in the sequence length due to the multiple key-query dot-products to compute.

First applications of LLMs only trained models on fixed size segments of a few hundred characters processed independently [1]. This limitation, due to the computational cost of attention, leads to a problem known as *context segmentation*, meaning that LLMs could only model dependencies up to the segment size. To tackle this issue, "windowed attention"

emerged as a solid alternative, borrowing ideas from Truncated Back-Propagation Through Time [44] with an attention mechanism that only attends to a fixed-size window of local tokens. In this method, while the dependencies are also limited by the window size on one layer, the maximum dependency across the whole LLM corresponds to the window size multiplied by the number of layers [8]. Figure 8 represents the size of the extended context allowed by windowed attention across 3 layers. Longformer [4] uses this windowed attention idea, and defines in addition to that important old "global tokens" which every token attends to.

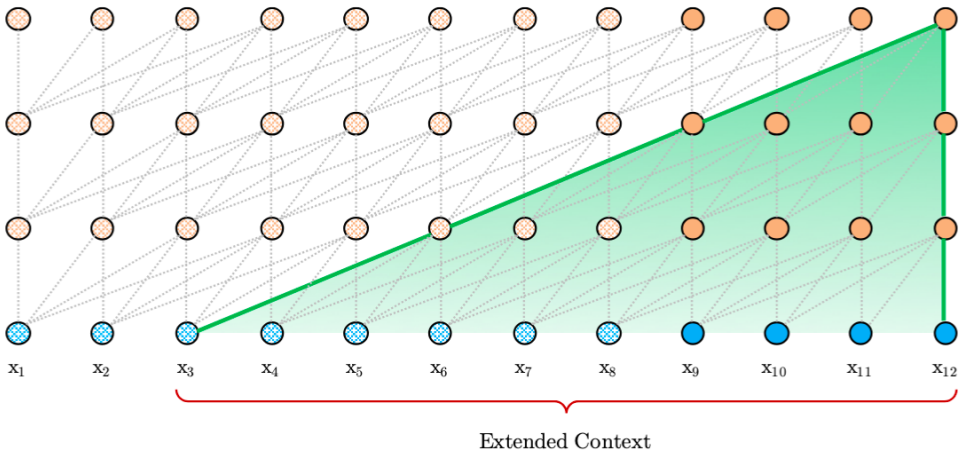


Figure 8: Representation of a windowed attention layer with a window size of 3 across 3 layers (taken from [8]). In every layer, each token is able to attend to its 3 preceding tokens. Over 3 layers, the extended context thus has size 9.

Alternatively, Sparse Transformers [6] only keeps a fraction of old strided tokens for long context in addition to a local window where all tokens are attended to. This idea of keeping only a fraction of important old tokens is rooted in the typical sparse behavior of attention : empirically, attention is proven to be very sparse, with only a few keys having high dot-product with the query. Figure 9 shows how dynamically selecting only 1% of most similar keys to the query allows to recover an average 89% of attention weights in a LLaMA 3 8B model. These keys take up most of the attention weight while the others have almost nonexistent weights [34]. Interestingly, these ideas were reused in recent progress by DeepSeek in

their paper on Natively Trainable Sparse Attention [48].

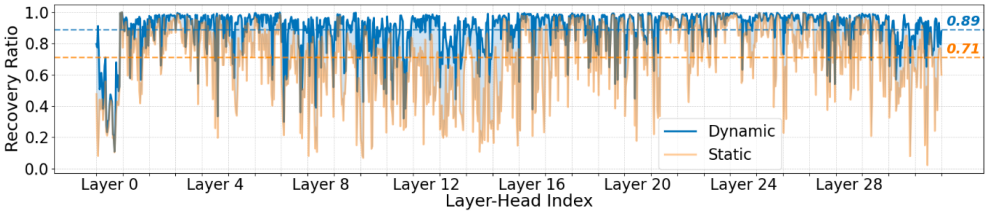


Figure 9: Recovery ratio of total attention weights using 1% of most important tokens on a LLaMA 3-8B model. This figure is taken from [28]. Using dynamically selected keys which depend on the query, an average recovery ratio of 89 % is achieved (blue curve). The orange curve represents the recovery ratio for a fixed 1% tokens that are independently chosen from the query, achieving a lower 73% ratio.

A more recent line of work, rather than keeping a fraction of past old tokens, uses compressive memory components to selectively retain important information about past tokens. These compressive memory components, often of fixed size, are generally combined with a windowed local attention pattern similar to the one previously mentioned. Memformer [46] and more recently LM2 [22] use cross-attention to make a fixed-size memory component interact with a local attention mechanism. Similarily, InfiniAttention [42] uses a hybrid between attention and Recurrent Neural Networks : it uses attention on local tokens while representing old tokens inside an iteratively updated hidden state, combined with local information using a gating mechanism. Alternatively, Compressive Transformers [35] try various compression functions (pooling, convolutions, retaining most used tokens) to compress the KV Cache of old tokens and then incorporate it back into the attention output.

Finally, some works achieve linear complexity attention by using low-rank matrixes or alternative kernels methods to replace the attention output [7] [47].

3.2 Post-Training Sparse Approximate Attention

We now delve into Post-Training Sparse Approximate Attention. In contrast to the sparse methods discussed earlier, where sparsity is built into the training process, this approach

operates on fully trained models that are originally dense, attending to all context tokens. Sparse attention approximates them by introducing sparsity post-hoc, selectively loading a fraction of all tokens' keys and values during inference.

Post-training sparse attention methods leverage the sparsity of attention (as mentioned in the previous subsection) to obtain accurate approximations. First methods like StreamingLLM [51] only kept the first tokens of the input text, known as "attention sinks" [24], as well as local window tokens. These few tokens already proved to yield a good approximation of the attention score, taking up a significant part of the attention weight. Building on StreamingLLM, H2O [27] proposed a token eviction policy that statically cached a fraction of "heavy-hitter" tokens based on statistics computed during prefilling. Unlike H2O, recent works cache all tokens but focus on efficient ways to dynamically retrieve most relevant keys from the KV Cache having high dot-product with a given query. They draw on approximate nearest-neighbor search, a decades-old body of literature researching how to find most similar vectors to a given query with a minimal number of distance computations. [28] used a custom nearest-neighbor graph index, while [5] leveraged Locality-Sensitive Hashing to introduce a sparse statistical estimator of attention outputs.

However, these works suffer from important limitations in the context of attention. First, keys and queries intrinsically have different distributions in self-attention, which makes nearest-neighbor search much less efficient in what is referred to as the "Out-Of-Distribution (OOD) Problem". On top of that, the application of RoPE (see subsection 2.2.1) leads to distribution shifts adding up to the OOD problem. Moreover, these methods are typically applied only at the decoding stage, while prefilling still requires half-precision computation without approximation. This is a major limitation, as prefilling remains computationally demanding and requires substantial resources for long sequences. Using approximation solely during decoding falls short of truly addressing the computational bottleneck inherent to attention. **Altogether, these reasons explain why we chose to focus our**

thesis on Post-Training Quantization. As we mentioned in the Preface, we still provide a brief description of our initial research efforts on Sparse Attention in Appendix F, but it is largely optional for the reader as it is unrelated to the core subject of this thesis.

3.3 Post-Training Quantization

> **Disclaimer.** *To avoid confusion for this subsection and the remainder of the thesis, we make the following clarification on dimension and representation conventions. When mentioning matrixes of shape $n \times d_{head}$, the first dimension (corresponding to n) will be referred to as the "token dimension" or "temporal dimension" while the second dimension (corresponding to d_{head}) is the "channel dimension". In all Figures representing 2D matrixes, each column always corresponds to a given token, while each row always corresponds to a given channel. When we refer to the "token dimension", we refer to the horizontal dimension of the matrix in the Figures, while the "channel dimension" refers to the vertical dimension.*

Post-Training Quantization (PTQ) differs from Sparse Attention methods in that all tokens are kept in the KV Cache and used in the attention computation. In PTQ, all these tokens are compressed in low-precision to limit the memory overhead of loading them from GPU RAM (HBM). More formally, in a given attention head, the KV Cache originally consists of two matrixes of shape $n \times d_{head}$, where each float is kept in half or high precision (16-bit or 32-bit). Post-Training Quantization aims to reduce this precision while keeping reasonable performance of the LLM.

Most straightforward methods compress every float number independently using low precision datatypes (2-bit, 4-bit or 8-bit), after a normalization step [21]. Typically, each float of

a matrix X is encoded uniformly in a B-int and decompressed following :

$$encode(X) = \left\lfloor \frac{X - z_X}{s_X} \right\rceil, \quad decompress(Y) = Y \cdot s_X + z_X$$

where $z_X = \min X$ is the zero-point, and $s_X = \frac{\max X - \min X}{2^B - 1}$ is the scale factor.

The zero-point and scale statistics are typically not computed on the whole $n \times d_{head}$ matrix, but rather along the token or channel dimension, as depicted in Figure 10. The quantization method KIVI [29] showed that channel-wise scaling is more efficient to quantize keys, while token-wise scaling works better for values. ATOM [52] uses a hybrid block-wise scaling also depicted in Figure 10.

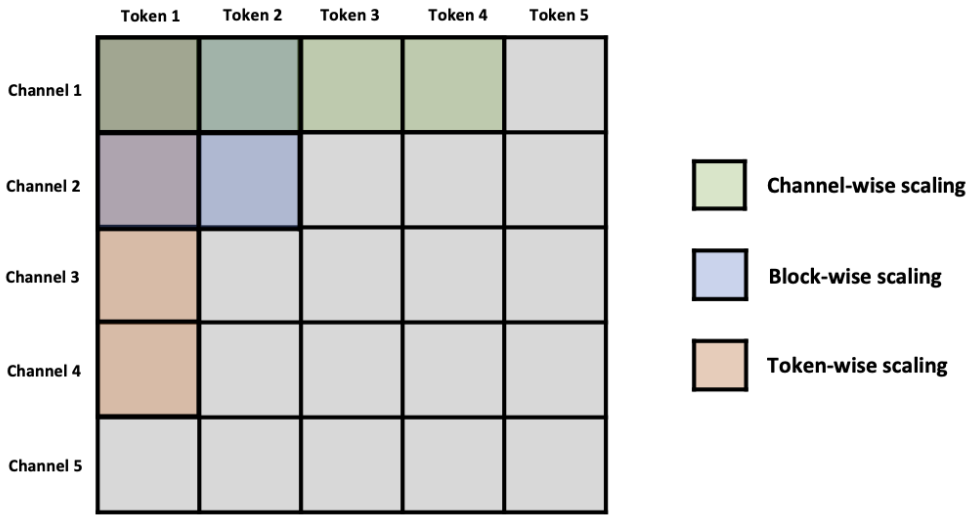


Figure 10: Different scaling strategies. Channel-wise scaling shares statistics for several different tokens on a unique channel. Token-wise scaling shares statistics for several different channels for a unique token. Block-wise scaling is a middle-ground used by ATOM which shares statistics both for some tokens and some channels.

Previously described quantization schemes are referred to as "uniform quantization" because they map values to a uniform range of integers. While they yield good results, efforts have also been dedicated to find non-uniform quantization methods better suited to the distribution of floats. In particular, instead of rounding floats to a uniform range of integers like

KIVI or ATOM, **KVQuant** [17] uses K-Means to determine a set of non-uniform centroids, as described in 2.3. This method optimizes the following loss.

$$\min_{(c_j)_{j=1,\dots,k},\,\text{assign}} \sum_{i=1}^{N} w_i \left\| a_i - c_{\text{assign}(a_i)} \right\|_2^2$$

Instead of integers between 0 and $2^B - 1$, floats are mapped to their closest value among the optimized centroids $(c_1, ..., c_k)$ following Algorithm 3. Note that unlike the compression equations in the Background section, where we quantized vectors z_i of dimension s, the a_i and c_j are floats here. Thus, KVQuant corresponds to an extreme case of Product Quantization where the chunks have a length of 1, i.e. the z_i are divided into s chunks of dimension 1. KIVI and ATOM also fall in this category compressing float by float, as represented in Figure 11.

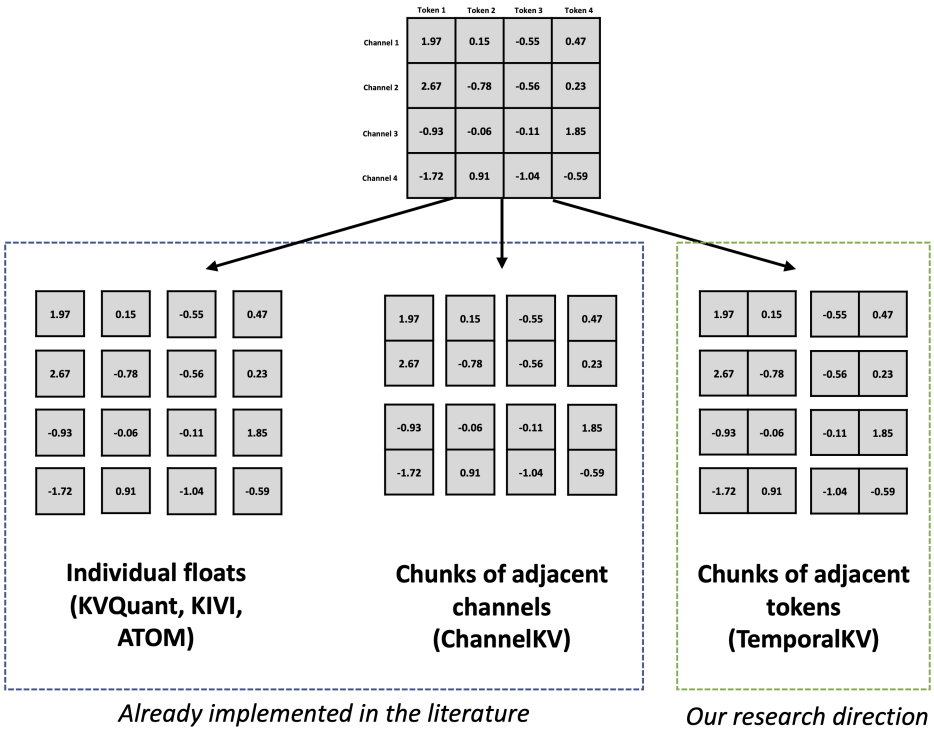


Figure 11: Comparison of quantization strategies of KVQuant [17], ChannelKV [50] and our research direction exploiting correlations of adjacent tokens. Here we represent chunks of size 2 for simplicity, but chunks of size 4 and 8 can also be used.

Because compressing float by float is an extreme and specific case of Product Quantization, the paper **ChannelKV** [50] improved on it by using Product Quantization on larger chunks (of size 2,4 or 8). These chunks group adjacent channels of keys/values as represented in Figure 11. This approach outperformed KVQuant because of the high correlations between adjacent channels.

Our method TemporalKV generalizes this approach by also leveraging "temporal" correlations, using chunks along the token dimension (see Figure 11 : TemporalKV groups adjacent tokens' values together). To the best of our knowledge, no previous work has used Product Quantization on chunks along the token dimension. However, we find in Subsection 4.1 that these correlations are high, outperforming correlations of adjacent channels on most layers of modern LLMs.

Summary of the Section: *In this section, we presented a broad review of the literature on optimizing the efficiency of attention. While our thesis is focused on Post-Training Quantization of standard attention, we also described alternative attention architectures and training-time strategies. We covered Post-Training Sparse Attention as well, an area we ultimately did not pursue in this thesis due to challenges in modern architectures, most notably the "Out-Of-Distribution" problem.*

4 Preliminary experiments

In this section, we perform preliminary experiments which will guide our experimental design, and compare our method to baselines KVQuant and ChannelKV.

4.1 Information Theoretic Study

Based on the tools introduced in Subsection 2.4, we perform an information theoretic study of our method TemporalKV in comparison to KVQuant and ChannelKV. As a reminder, KVQuant compresses matrixes float per float, ChannelKV compresses matrixes using chunks along the channel dimension, and TemporalKV uses chunks along the token dimension (See Figure 11). We will show that the chunking of our method TemporalKV requires less information to encode that KVQuant and ChannelKV. In information theoretic language, this is equivalent to saying that the Joint Entropy (see Definition 2.4.1) per float of TemporalKV is lower than that of ChannelKV and KVQuant.

As a first illustration, we plot on Figure 12 the 2D Histograms of chunks of dimension 2 from the methods TemporalKV (up) and ChannelKV (down), taken from key K and value V matrixes. To plot these histograms, we collected 64 sample texts of token length 1024 on Wikitext-2 [31] and processed them using a LLaMA 3.1 8B model. We then chunked these activations using ChannelKV and TemporalKV's methods (See Figure 11), with a chunk size of 2. The histograms represent the distributions of these 2-dimensional vectors. Note that the activations were taken before RoPE was applied (See Definition 2.2.1) because we later show it is the most efficient way to compress Keys and Queries (see next Subsection).

We find that TemporalKV's chunks exhibit a clear diagonal pattern, which means that floats from adjacent tokens are likely to be similar in value. Meanwhile, the correlations with ChannelKV's chunking are much less clear, with a symmetric pattern that is seemingly harder to encode as all directions seem equivalent.

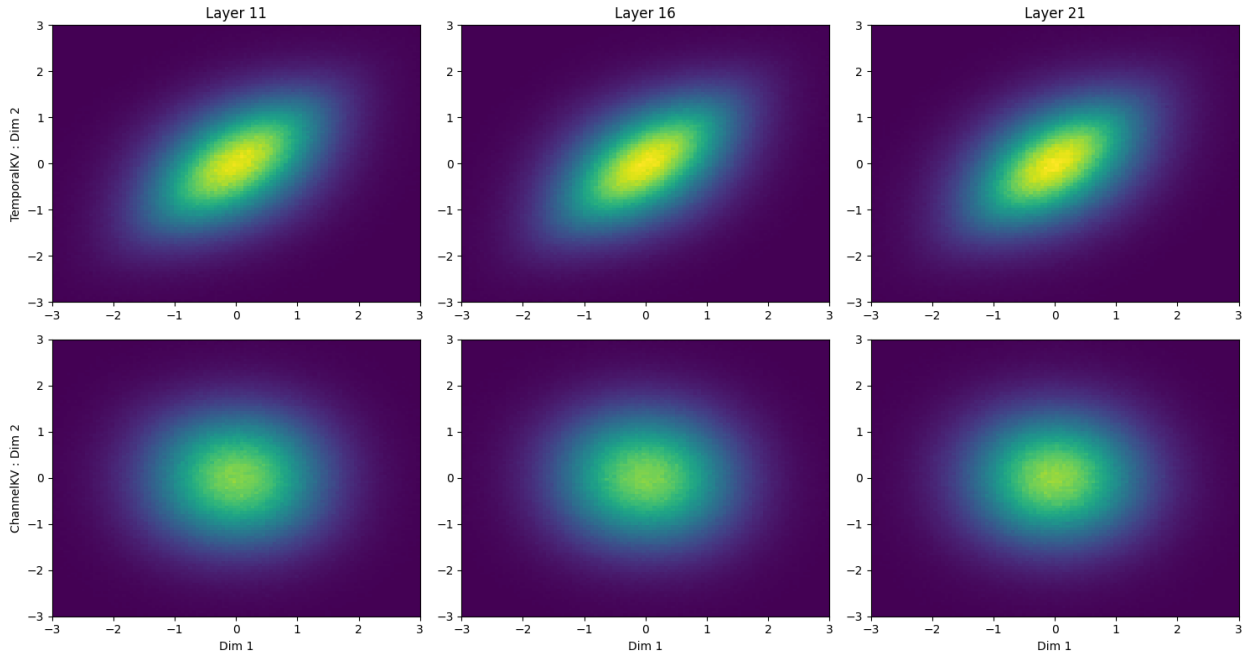


Figure 12: 2D Histograms of methods TemporalKV (first line) and ChannelKV (second line) on 3 different layers, on a LLaMA 3.1 8B model

To confirm this visual observation, we computed joint entropies per float for different numbers of coupled channels for ChannelKV and TemporalKV, also comparing these results to the entropies of KVQuant. However, the entropy definitions introduced in Subsection 2.4 are intractable on a continuous set like $\mathbb{R}$ given a finite number of samples. Therefore, we discretized the domains using the "binning trick" [23] and calculated entropies on the resulting discrete probability distribution. The results are displayed in Figure 13.

Important observations can be made from Figure 13 :

- Coupling channels clearly allows to reduce the entropy per float, thus showing that the two chunking methods of TemporalKV and ChannelKV are more information-efficient than KVQuant. The more channels coupled, the less entropy : therefore, we should aim to use the largest computationally tractable chunk length (see Subsection 2.3.2 for the discussion on computational cost).
- TemporalKV's chunking clearly has lower entropy than ChannelKV's chunking on all

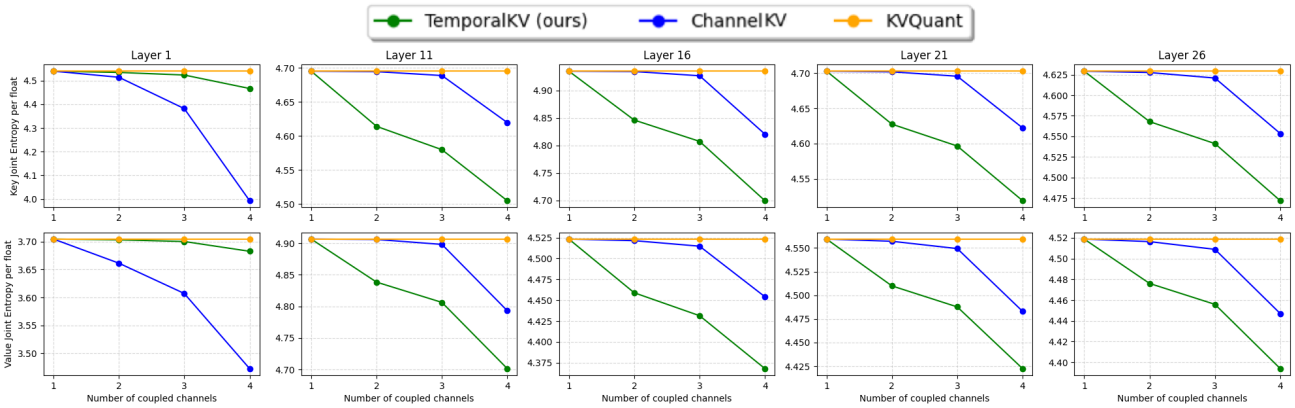


Figure 13: Plots of joint entropy per float as a function of the number of channels for TemporalKV (green), ChannelKV (blue) and KVQuant (yellow). Results for keys are on the first line, results for values are on the second line. On all layers except the first ones, TemporalKV clearly outperforms ChannelKV and KVQuant on all numbers of coupled channels.

layers except Layer 1. This shows than the chunking we introduce in our thesis is more information-efficient than ChannelKV's on most layers of the LLM. That being said, Layer 1 is a critical layer on most LLMs, and ChannelKV still has a significant edge on this layer. **Therefore, our study shows that the very first layer(s) should be compressed using ChannelKV's chunking whereas our method TemporalKV should be used for all other layers.**

Figure 14 further compares the methods' entropies across all layers, confirming that TemporalKV's chunking outperforms ChannelKV's on all layers apart from layer 1, on both Keys (left) and Values (right). Again, both methods outperform KVQuant for all layers.

It is worth noting that ChannelKV greatly outperforms other methods on Layer 1 only, particularly for Keys. This can be explained by the fact that the earlier layers of an LLM typically represent simple functions, with **low-rank** projection matrixes. Thus, activations' channels are highly correlated, resulting in high performance of ChannelKV. To confirm our intuition, we plot the effective rank of projection matrixes W_K and W_V (See Definition 2.2.1) on Figure 15. We find that Layer 1 and 2 indeed have particularly low-rank W_K compared to later layers, thus resulting in the great performance of ChannelKV for keys on those

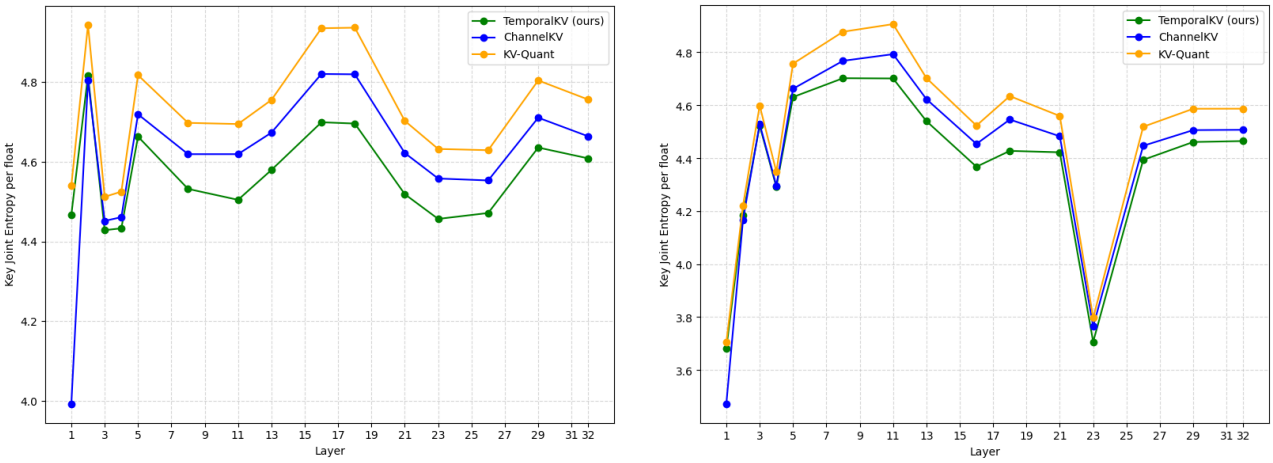


Figure 14: Joint entropy per float on all layers of a LLaMA 3.1 8B model, for Key vectors (left) and Value vectors (right). On Keys, our method TemporalKV (green) outperforms ChannelKV (blue) and KVQuant (yellow) on all layers except the first ones, ChannelKV having a clear edge on layer 0. On Values, TemporalKV outperforms the other methods for layers after 5, but ChannelKV still has an edge for layer 1.

particular layers. Explaining the performance of ChannelKV for values on Layer 1 is less immediate, as W_V remains high-rank. Most likely, the underlying data at Layer 1 is already low-rank before being projected.

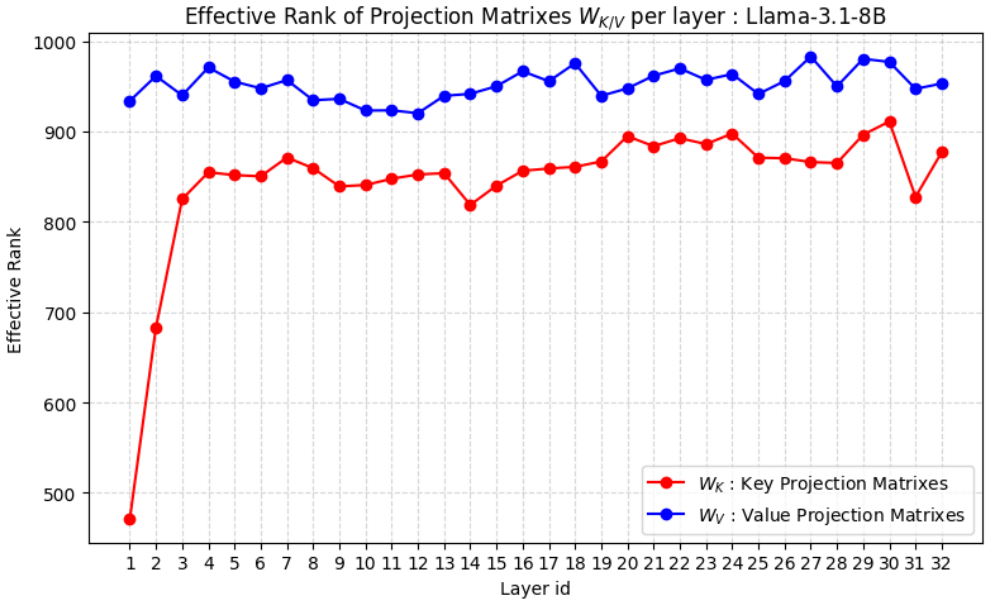


Figure 15: Effective Rank of projection matrixes W_K (red) and W_V (blue) per layer on a LLaMA 3.1 8B model. Key Projection matrixes have much lower rank for the first layers.

4.2 Analyzing the channels of keys and values

In this subsection, we analyze the different channels of Keys and Values to guide our compression method. In Figure 16, we represent channel magnitudes across the channel and token dimensions before and after RoPE is applied (See Definition 2.2.1 for RoPE) on a LLaMA 3.1 8B model.

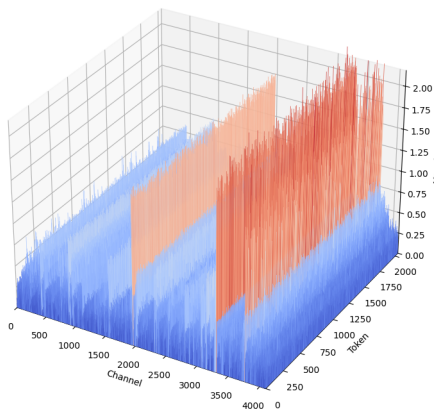
From Figure 16, we make two fundamental observations :

- Pre-RoPE activations exhibit a much more regular pattern than post-RoPE activations. The application of positional encodings in RoPE perturbs the magnitudes, creating sinusoidal patterns that are detrimental to normalization.
- For Pre-RoPe activations, magnitudes clearly remain stable for a same channel, but definitely not for a same token.

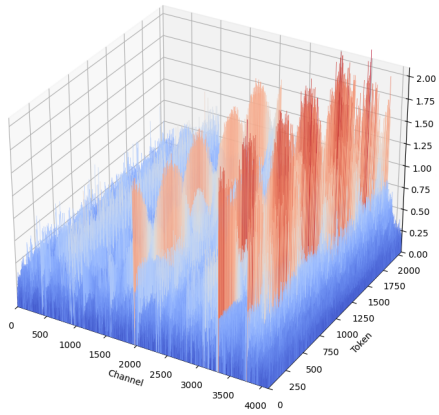
From those two observations, we draw the following conclusions :

- Because pre-RoPE keys exhibit a much more regular pattern, Keys must be compressed before RoPE is applied, not after. Note that this requires applying RoPE on the fly everytime we load Keys from the KV Cache (see Definition 2.2.1), which is doable as it does not represent a significant bottleneck [17].
- Because activation magnitudes only remain stable for a same channel, we need to use channel-wise normalization rather than block/token-wise normalization, as represented in Figure 10.[6]

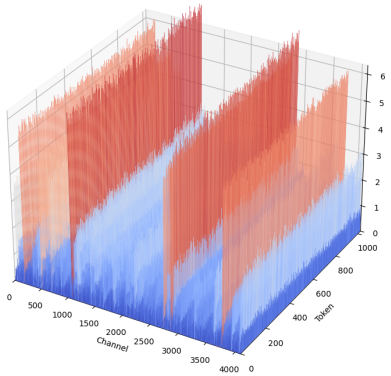
[6]For Values, in practice we found that the normalization strategy had little impact on the results, and we therefore used channel-wise normalization as well.



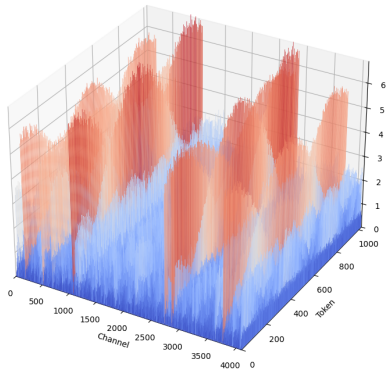
(a) Layer 1 before RoPE



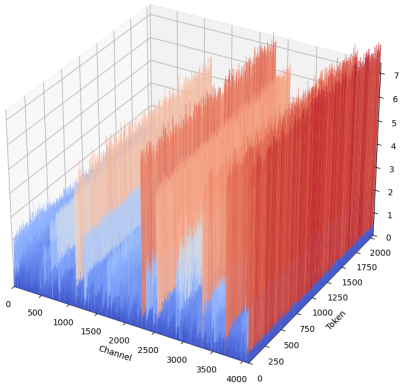
(b) Layer 1 after RoPE



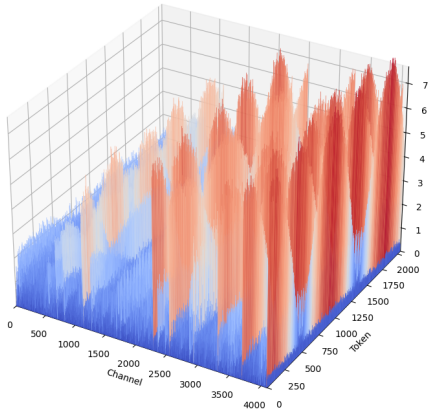
(c) Layer 16 before RoPE



(d) Layer 16 after RoPE



(e) Layer 32 before RoPE



(f) Layer 32 after RoPE

Figure 16: Representation of Keys' activation magnitudes across two dimensions : channel and token dimensions. Each row shows a pair of activations: before and after RoPE is applied. Colors from blue to white to red represent the increasing magnitude of activations. This figure clearly shows how regular pre-RoPE activations are compared to post-RoPE.

4.3 Choosing between Product Quantization and Residual Quantization

In this final subsection, we compare the two forms of compression introduced in Subsection 2.3 : Product Quantization and Residual Quantization[7]. To do so, we run these compression methods (with $m = 2$) on activations collected from the Wikitext dataset, on a LLaMA 3.1 8B model. We keep the number of bits per float constant across methods for fair comparison. As an example, for a vector of dimension 4, Product Quantization divides it in 2 vectors of dimension 2 compressed separately, while Residual Quantization compresses the 4-dimensional vector two times recursively (see Subsection 2.3).

Dim	**Layer 1**	**Layer 5**	**Layer 16**	**Layer 26**	**Layer 32**
4	+33%	+14%	+13%	+12%	+16%
8	+27%	+14%	+11%	+11%	+13%
16	+29%	+24%	+22%	+20%	+22%

Table 1: Relative MSE difference of Residual Quantization compared to Product Quantization

In Table 1, we find that Residual Quantization systematically exhibits higher Mean Squared Error than Product Quantization, with differences ranging between 11% and 33% on the layers studied. Thus, it shows that Product Quantization performs better than Residual Quantization in our use case at equal precision, justifying the use of Product Quantization for our TemporalKV method.

[7]For Residual Quantization, we use a greedy assignment strategy [11], as it is the only computationally tractable type of Residual Quantization in our case.

Summary of the Section: *In 4.1, our study on entropy showed that on almost all layers, TemporalKV's encoding is more information-efficient than ChannelKV and KVQuant. Only a few first layers act as outliers, due to their low-rank nature. In 4.2, we analyzed activations' magnitudes and showed it is more appropriate to use channel-wise normalization and perform compression on pre-RoPE keys. In 4.3, we compared Product Quantization and Residual Quantization and concluded on the superiority of Product Quantization in our usecase.*

5 Proposed method

In this section, we specifically detail and explain the design of our method TemporalKV.

5.1 General framework

Our method follows the previously described compression framework in Figure 7. We now detail the *encode* and *decompress* phases of our method TemporalKV in Figure 17. Note that the normalization phase is not included in this figure for simplicity.

- In the *encode* phase, we chunk the K/V matrix by grouping, for a same channel, adjacent tokens' values together. Then, we separately assign each of these chunks to its closest centroid among a list of pre-determined 256 centroids[8]. The corresponding int8 indice of the centroid is then stored in the KV Cache.
- In the *decompress* phase, we look up in our centroid table to reconstruct each chunk with its corresponding centroid, which then allows to reconstruct the K/V matrix.

The normalization is done before encoding, and the denormalization is done after decompressing, using per-channel scaling as explained in the Preliminary Experiments Section. The means and standard deviations are computed offline for every channel in each head and layer on a calibration dataset.

We highlight that our approach slightly differs from classic Product Quantization introduced in the Background Section. Indeed, Product Quantization uses a different centroid set for every chunk of adjacent tokens, and shares it across all channels. In our case, this is untractable as the number of tokens can be huge. **Thus, we instead share centroids across all tokens and use different centroid sets for different channels.**

[8]We systematically use 256 centroids per chunk because it's the maximum number where the indices can efficiently be stored with PyTorch's int8. The next int type would be int16 corresponding to $2^8 = 65536$ centroids, which becomes computationally intractable in an LLM pipeline.

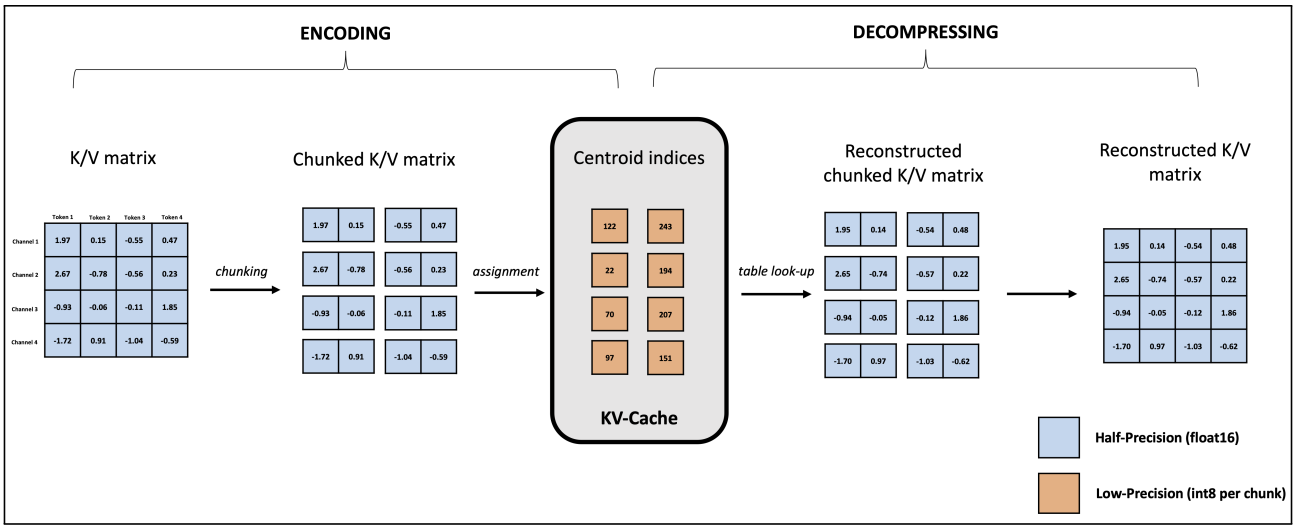


Figure 17: Temporal KV's compression framework, representing the *encode* and *decompress* phases (following Figure 7's framework) for a chunk size of 2

We use 3 configurations with chunk sizes of 2, 4, and 8, respectively corresponding to 4-bit, 2-bit and 1-bit per float precision (because the precision for each chunk is 8-bit using an int8 indice). In the experiments, they will be respectively referred to as TempKV-2-4bit, TempKV-4-2bit and TempKV-8-1bit. The configuration represented on Figure 17 is TempKV-2-4bit (chunks of size 2).

5.2 Compression details

In this subsection, we detail how the centroids are computed using weighted K-Means. We consider N chunks $u_1, u_2, ..., u_n$ of size l_{chunk} taken from one channel in one attention head.

5.2.1 Fisher-weighted K-Means

In our case, the basic K-Means compression objective would be :

$$\min_{(c_j)_{j=1,...,k},\,\text{assign}} \sum_{i=1}^{N} \left\| u_i - c_{\text{assign}(u_i)} \right\|_2^2 \tag{2}$$

However, as remarked in other works [50] [17], LLMs are more sensitive to the quantization of certain activations than others. This is related to the fact that some tokens concentrate

more attention weight than others, as we explained in the Related Work Section. This sensitivity can be quantified for a given activation u using the gradient $\frac{\partial \mathcal{L}}{\partial u}$, where $\mathcal{L}$ is the cross-entropy loss commonly used for LLMs. Following the sensitivity framework of [32] [17], we thus use an objective with Fisher weights $\left(\frac{\partial \mathcal{L}}{\partial u_i}\right)^T \frac{\partial \mathcal{L}}{\partial u_i}$, which correspond to a sum of diagonal Fisher matrix coefficients. These Fisher weights capture how much compressing a given chunk affects the loss $\mathcal{L}$. They are incorporated by formulating the problem as a weighted K-Means :

$$\min_{(c_j)_{j=1}^{k}, \text{assign}} \sum_{i=1}^{N} \left(\frac{\partial \mathcal{L}}{\partial u_i}\right)^T \frac{\partial \mathcal{L}}{\partial u_i} \left\| u_i - c_{\text{assign}(u_i)} \right\|_2^2 \tag{3}$$

In Table 2, we perform an ablation experiment on LLaMA 7B 32K where we measure perplexity of the quantized model with and without Fisher weighting :

Chunk size	**w/o Fisher weightiing**	**w/ Fisher weighting**
4	6,77	6,56
8	10,96	7,83

Table 2: Perplexity comparison with and without Fisher Weighting on a LLaMA 2 7B 32K model.

We find that the Fisher-weighted K-Means allows to better optimize model perplexity by shifting the centroid distribution towards more critical activations.

5.2.2 Centroid Sharing and Normalization

Instead of strictly using a different centroid set for each channel, in practice we relax this rule by sharing centroids for groups of a few adjacent channels. The benefit of this practice is that we reduce the memory overhead of our method by storing less centroids. However, the drawback is that different channels have different magnitudes and zero-points, as previously shown in Figure 16, thus one single centroid set cannot directly adapt to different channels.

This is why we use normalization to have a uniform distribution across different channels

and allow centroid sharing between them[9].

In the experiments, the size of channel groups for centroid sharing is set so that the memory overhead of our method is exactly the same as that of ChannelKV, thus allowing for fair comparison. In Appendix B, we detail the memory footprint of centroid storage and show that it systematically only represents a small percentage of the model's weights.

5.3 Outlier removal

The KVQuant paper [17] introduced outlier removal and showed it can greatly improve the performance of quantized LLMs with limited memory footprint.

It consists in determining, for a given selectivity (commonly of 1%), the highest and lowest activations in the KV Cache (referred to as "outliers") and store them in full/half precision in a sparse matrix format. Following KVQuant's framework, we implemented outlier removal for our method and also for ChannelKV (which was not done in their original paper). To do so, we calculated 0.005 and 0.995 quantiles for every channel in every head and layer on a calibration set (Wikitext-2, as in KVQuant), and used these quantiles to determine outliers during evaluation.

We made two key observations :

- The first observation, also made in KVQuant's paper, is that outlier removal indeed greatly improves the quality of models, clearly reducing perplexity and improving accuracy. This observation can be made across all our experiments (see the Experiments Section).

- However, computing outlier thresholds offline can be misleading, especially in cases of distribution shifts between the calibration set and the test set. Indeed, the quantiles calculated on a calibration set do not necessarily generalize. We followed the threshold

[9] In practice, we only need normalization for keys as value channels already have similar scales and zero-points

calibration method of KVQuant on the same dataset Wikitext [31] and found that the proportion of outliers on the test set can be very variable, with percentages ranging from 1% to 4,56%, as depicted in Table 3. We thus chose to manually cap the proportion of outliers to 1% in all our experiments for all methods.

Dataset	**Test outlier rate**
Wikitext-2	1,05 %
C4	1,32 %
PTB	1,22 %
RULER-NIAH1	4,56 %
RULER-NIAH2	1,44 %

Table 3: Test outlier removal rate using 0.005 and 0.995 quantiles from Wikitext-2 on a LLaMA 3.1 8B model.

5.4 Keeping Attention Sinks in full precision

Attention sinks are known to have extreme importance in self-attention layers [3], concentrating a high percentage of attention weights. Thus, LLMs are highly sensitive to compression errors on attention sinks. While Fisher-weighted K-Means should already take this sensitivity into account, we find in practice that attention sink's Fisher weights are much higher than other tokens. Figure 18 depicts the average Fisher weight per token on a LLaMA 3.1 8B model : it shows that the first token's Fisher magnitude is higher by 3 orders of magnitude than most tokens. Thus, in a Fisher-weighted K-Means, the attention sink risks highly biasing the centroid computation and reducing precision for other tokens. Hence, we make the choice of systematically excluding the first 8 tokens' activations from K-Means clustering and we always keep them in half-precision in our experiments.

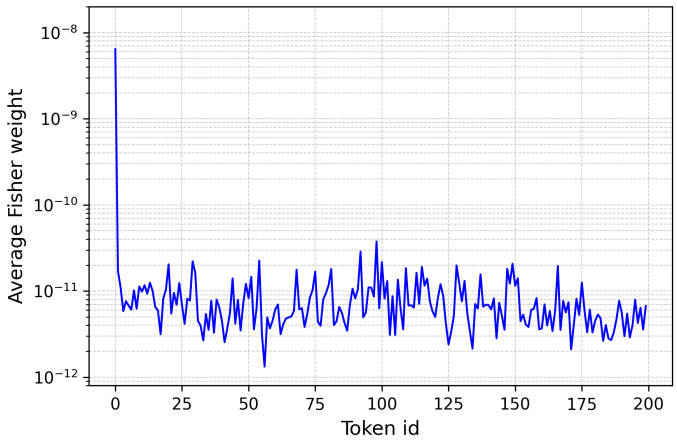


Figure 18: Fisher weight from LLaMA 3.1 8B's layer 15 averaged on 8 text samples of 2048 tokens from Wikitext-2 dataset. The x-axis represents the token's position in the sequence.

To confirm our intuition, we run an ablation experiment where we compress attention sinks with our centroids instead of keeping them in half-precision. Results are available in Table 4 and clearly show the benefit of keeping them in half-precision : we observe a huge perplexity increase when not keeping the attention sinks in half-precision.

Model	**w/ sink compression**	**w/o sink compression**
LLaMA 3.1 8B	10,82	145,77
LLaMA 3.2 1B	15,71	92,32
LLaMA 3.2 3B	13,42	104,21

Table 4: Perplexity comparison with and without attention sink compression.

5.5 Hybrid method for early layers

In our preliminary experiments, we showed that the chunking of TemporalKV is more information-efficient than ChannelKV on almost all layers of the LLM, except a few early ones. To solidify this observation, we also compared Mean Squared Errors (weighted with gradients) of both methods to precisely determine which is better.

Figure 19 represents a per layer breakdown, indicating which method has a lower MSE for every layer in 3 different LLaMA models. This Figure is for a chunk size of 4, but we also

compute it for chunk sizes 2 and 8 along with the per head breakdown in Appendix D. On average, TemporalKV outperforms ChannelKV on 87% of all layers, showing the comparative efficiency of TemporalKV's chunking. These layers are systematically among the first ones, which is consistent with our information-theoretic study.

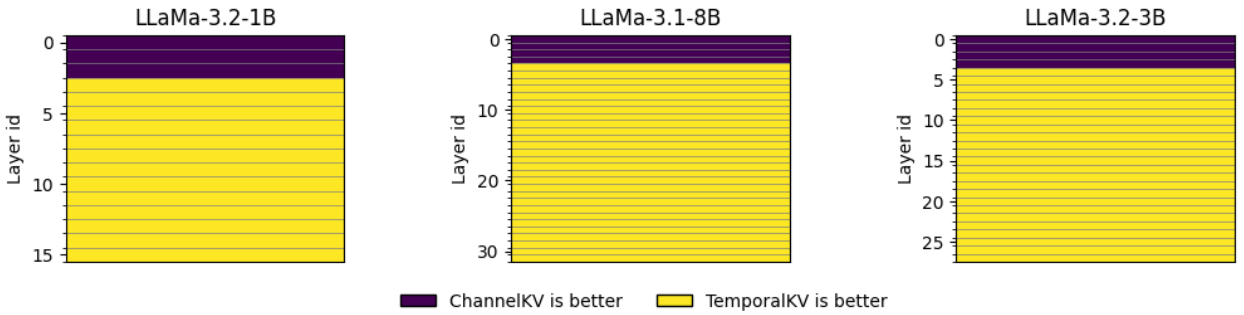


Figure 19: Breakdown of the quantization method with lowest Fisher-weighted Mean Squared Error on 3 LLaMA models. Results are represented for a chunk size of 4.

To leverage the complementarity of the two chunking methods, we apply our temporal chunking to all layers except the early ones identified in our Mean Squared Error comparison, for which we use ChannelKV's chunking.

6 Experiments

In this section, we detail our experimental setup and evaluate our compression method TemporalKV on a wide variety of datasets, tasks and models. In our experiments, we compare our method to the previously described state-of-the-art compression methods : KVQuant and ChannelKV.

6.1 Datasets

The datasets used in our experimental study span a wide range of text and tasks in order to provide a comprehensive evaluation of TemporalKV relatively to the baselines. We experiment on Wikitext [31], C4 [36], the Penn Tree Bank dataset [30] and multiple tasks from the RULER benchmark [18], all in English language.

- Wikitext (2016, Salesforce): Like previous works KVQuant and ChannelKV, we use this dataset to calibrate our centroids with K-Means. This choice was made because Wikitext is one of the "classic" benchmarking datasets for Large Language Models, with clean and well-curated English text taken from the set of verified "Good and Featured articles" on Wikipedia. It contains long coherent paragraphs useful for learning/evaluating on long context tasks, and has large vocabulary.

- Colossal Cleaned Common Crawl/C4 (2020) : Used for evaluation in our experiments, this dataset is another "classic" dataset also commonly used for pretraining of LLMs. It was developed by Meta and Google as a smaller curated version of the "Common Crawl" dataset, the most widely used open web crawl for LLMs with billions of Internet pages. It is thus a representative subset of English Internet, very suitable for pretraining or challenging evaluation. Note that this dataset was also chosen for evaluation in the original papers of KVQuant and ChannelKV.

- Penn Tree Bank dataset (1990, UPenn): This older NLP dataset was released in the

1990s by the University of Pennsylvania and consists of articles from the Wall Street Journal. It is a tiny dataset that cannot be used for training but represents a quality well-curated evaluation baseline.

To measure the accuracy of our models in challenging long-context scenarios, we evaluate them on the RULER benchmark [18]. This benchmark contains a wide range of tasks allowing to evaluate LLMs' long-context capabilities in different scenarios :

- Needle-In-A-Haystack (NIAH) : In this task, the model is given a long context of several thousand words. This context mostly consists of "distracting text", and a "needle" is hidden at a random position in the text. This needle is the sentence "One of the special magic numbers for long-context is: 12345", indicating a "magic number". At the end of the context, a query sentence asks the model to retrieve this "needle" with the text : "What is the special magic number for long-context mentioned in the provided text?". The model succeeds if it manages to generate the exact number. This task allows to measure the ability of the model to extract localized information from a long context.

- Needle-In-A-Haystack variants : Many variants to the NIAH problem exist. In the Multi-Key NIAH (MKey) task, several potential needles are inserted in the text but the model is prompted to only retrieve a specific one. In Multi-Values NIAH (MVValue) and Multi-Queries NIAH (MQuery), several keys are also inserted but the model must retrieve all of them.

- Variable Tracking (VT) : This task emulates a "minimal coreference chain resolution task", as described in RULER's paper. More precisely, a variable X is initialized with a value V in the text input ("X = V"), and multiple other assignment equations are written at various positions throughout the whole context (e.g. "Y = X", "Z = Y"). The model is then prompted to return all variables names pointing to the value V. This task measures the model's ability to track and reason about logical statements across a large context.

- Common Word Extraction (CWE, FWE) : In Common Word Extraction, the model is given a long context and asked to retrieve the words appearing most frequently in this text. The model is typically prompted using the sentence "What are the 10 most common words in the above text?". This task allows to measure the ability of the model to precisely synthesize information from a broad context where every part is important.

- Question Answering (QA) : In Question Answering, the model is given a long context and prompted to answer a multiple-choice question about this text. The context mostly consists of "distracting text", and a "golden paragraph" randomly inserted in the text containing the answer to the question. This task measures the ability of the context to reason about a localized paragraph among a long context.

6.2 Models

To perform our experiments, we use the latest long context models released by Meta. These models are open-weight and announce supporting contexts up to 128k tokens. Due to our limited academic computational resources, we restrict ourselves to models under 8B parameters : LLaMA 3.1 8B, LLaMA 3.2 1B and LLaMA 3.2 3B.

We represent the configuration parameters of the models used in Table 5. "#Q Heads" represents the number of query attention heads, whereas "#KV Heads" represent the number of key-value attention heads. Note that these numbers are different because all models use Grouped-Query Attention, sharing multiple query heads for each key-value head.

Model	**Hidden Size**	**# Layers**	**# KV Heads**	**# Q Heads**	**Vocab Size**
LLaMA-3.1-8B	4096	32	8	32	128,256
LLaMA-3.2-3B	3072	28	8	32	128,256
LLaMA-3.2-1B	2048	16	8	32	128,256

Table 5: Architectural parameters for LLaMA models. Head dimension is computed as hidden size divided by the number of Query heads. Parameters were taken from the Hugging Face model pages.

The models use RoPE scaling, SiLU [16] activation functions, and the normalization function is RMSNorm [49]. They all use the datatype torch.bfloat16 from PyTorch.

6.3 Experimental details

In this subsection, we detail our experimental setup. We were able to perform our experiments thanks to Oxford University's Advanced Research Computing service and the help of Prof. Mike Giles, who gave us access to compute units.

6.3.1 K-Means Clustering

For the K-Means clustering, we used an available fast GPU implementation [10] that we edited to support the weighted version of K-Means. We were able to batch multiple clustering operations on small GPUs (e.g. T4), such that running K-Means across the whole model took at most a few hours. We deem this time as acceptable, because the centroid computation only needs to be done once and can then be used for all datasets. We performed 50 iterations of K-Means for each run, using K-Means ++ [2] to initialize the centroids.

Training samples came from 64 text excerpts of 2048 tokens all from the Wikitext-2 dataset. We chose this dataset because it proved to allow good generalization in previous works. After customizing the Hugging Face library so that we could collect both activations and gradients (similarily to previous works KVQuant and ChannelKV), we stored the corresponding tensors for all heads and layers.

6.3.2 Implementation

To evaluate the models, experiments were performed using PyTorch [33] and Hugging Face's Tranformers library [45]. We ran the experiments on a single A100 GPU with 40GB of RAM.

To run our modified compressed forward pass, we implemented a custom "LlamaAttention" Hugging Face class to insert the *encode* and *decompress* operations, as previously depicted

in Figure 7. As represented in Figure 17, we normalize each new token's activations (step 1), reshape its Key/Value matrix into chunks (step 2), and assign each chunk to the closest centroid (step 3). The corresponding indices are then stored in the KV Cache. During the decompressing phase, we recover each chunk using a table look-up (step 4), reshape the chunks to recover the normalized K/V matrix that we denormalize (step 5). Finally, we apply RoPE to the denormalized K/V matrix (step 6)

Due to limitations in time, we chose to focus on optimizing the accuracy and perplexity of our method, and we did not dedicate as much time to a low-level latency optimization of our attention framework. Our current latencies (reported in Appendix C) are obtained with high-level PyTorch code which constrains us to copy the decompressed matrix back to GPU RAM, thus preventing TemporalKV from speeding up standard attention. However, we want to emphasize that a custom CUDA attention kernel would provide substantial speedup. Indeed, the previously described steps can be performed locally on GPU SRAM and limit the number of reads and writes to GPU HBM. We provide a detailed framework in Appendix A to obtain such a speedup.

Coming back to our current PyTorch implementation, we bring attention to a few details which made our implementation considerably faster:

- To speed up the encoding step (see Algorithm 3), we compute squared distances using the standard expansion of the Euclidean norm. For a chunk $u \in \mathbb{R}^{l_{chunk}}$ and centroids $c_j \in \mathbb{R}^{l_{chunk}}$,

$$\|u - c_j\|_2^2 \;=\; \|u\|_2^2 \;-\; 2\,u^\top c_j \;+\; \|c_j\|_2^2. \tag{4}$$

  Rather than forming $(u - c_j)$ and summing its squared components—which can create a large $B \times k \times l_{chunk}$ temporary tensor when computing distances between B chunks and k centroids in batch—we precompute the norms $\|u\|_2^2$ (once per u) and $\|c_j\|_2^2$ (once per centroid), and obtain the cross terms via batched dot products.

Let $U \in \mathbb{R}^{B \times l_{chunk}}$ collect the chunks u_b row-wise and $C \in \mathbb{R}^{k \times l_{chunk}}$ collect the centroids c_j row-wise. We define $\mathbf{n}_u \in \mathbb{R}^B$ with $(\mathbf{n}_u)_b = \|u_b\|_2^2$ and $\mathbf{n}_c \in \mathbb{R}^k$ with $(\mathbf{n}_c)_j = \|c_j\|_2^2$, and let $\mathbf{1}_B, \mathbf{1}_k$ be all-ones column vectors. Then the full distance matrix $D \in \mathbb{R}^{B \times k}$ can be written as

$$D \;=\; \mathbf{n}_u \mathbf{1}_k^\top \;-\; 2\,UC^\top \;+\; \mathbf{1}_B \mathbf{n}_c^\top. \tag{5}$$

In practice, for long context we compute $UC^\top$ in tiles and stream the corresponding slices of $\mathbf{n}_u$ and $\mathbf{n}_c$ to fit in GPU memory. A further improvement, which would need to be implemented in CUDA, would fuse the distance computation and the argmin of Algorithm 3 on GPU SRAM to avoid copying distances back and forth between SRAM and HBM.

- In the decompressing step, to convert the stored int8 centroid indices from the KV Cache into their corresponding float16 vectors, we have to perform an efficiently batched table look-up. After experimenting, we decided to use a torch.gather operation, which we found to be much more efficient than PyTorch's advanced indexing.

6.3.3 Choice of the Baselines

In our experiments, we chose to compare our work to KVQuant and ChannelKV, which we presented in Subsection 3.3. The choice of these baselines was made because these methods represent the current state-of-the-art in low-bit quantization. When it was published in 2024, KVQuant beat previous uniform quantization methods such as ATOM or KIVI, and remains among the most competitive low-bit quantization methods to this day. ChannelKV is a variant of KVQuant which slightly improved on the experimental results, and has some similarities in spirit with our method. Moreover, as we detailed in Subsection 5.5, we use a hybrid method employing ChannelKV's chunking for the first layers of the LLM. Therefore, it was natural to also include this baseline in our experiments.

To ensure a fair comparison, we avoid making cross-paper metric comparisons. We make

this choice because backbones may differ, especially in terms of outlier removal rate, sink token compression and centroid sharing. What's more, the paper ChannelKV did not release any code, making it difficult to precisely know about the implementation details. Thus, we carefully re-implemented KVQuant and ChannelKV in our pipeline. The bit budgets, outlier removal rates, clustering, normalization as well as the train and test data are made rigorously uniform.

6.4 Results

Now that the experimental setup has been fully presented, we finally present our experimental results.

6.4.1 Perplexity comparisons

We measure the perplexity of our models on C4 and PTB. For all perplexity measurements, we used the same text preprocessing as KVQuant and used inputs of 2048 tokens. Table 6 shows our experimental results on C4 without outlier removal.

Configuration	Bits Per Float	LLaMA-3.1-8B	LLaMA-3.2-3B	LLaMA-3.2-1B
float16	16	9.17	10.75	11.21
KVQuant-4bit	4.00	9.37	11.21	13.63
ChanKV-2-4bit	4.00	9.74	12.26	13.62
TempKV-2-4bit	4.00	**9.31**	**11.04**	**13.47**
KVQuant-2bit	2.00	10.92	**13.39**	19.24
ChanKV-4-2bit	2.00	11.21	13.89	16.60
TempKV-4-2bit	2.00	**10.82**	13.42	**15.71**
KVQuant-1bit	1.00	33.24	51.47	197.14
ChanKV-8-1bit	1.00	19.95	26.67	36.65
TempKV-8-1bit	1.00	**16.16**	**22.51**	**28.04**

Table 6: Perplexity measured on C4 (the lower the better), on 256 text samples of 2048 tokens. No outlier removal is implemented for any method in this table.

From this table, it can first be observed that TemporalKV outperforms the baselines in 8

settings out of 9. In the only setting where it is outperformed (2 bit regime of the 3.2-3B model), TemporaKV remains very close to the best baseline KVQuant.

Also, the more channels are coupled, the better TemporalKV and ChannelKV perform compared to KVQuant. This is consistent with our entropy analysis in Subsection 4.1 : the longer the chunk size, the lower the entropy. Interestingly, while ChannelKV only outperforms KVQuant when many channels are coupled (chunk sizes of 4 or 8, respectively corresponding to the 2bit and 1bit regimes), TemporalKV also maintains superior performance for a chunk size of 2 (4 bit regime). This also aligns with our entropy analysis in Figure 13, where we could see that coupling 2 channels in ChannelKV practically had no impact on the entropy compared to KVQuant : conversely, coupling 2 channels in TemporalKV allowed for a clear entropy reduction on most layers.

In the next experiment, we also measure perplexity on the C4 dataset, but this time using outlier removal for all the methods : 1% of outliers are kept in half-precision. The results are available in Table 7. We find that TemporalKV outperforms both KVQuant and ChannelKV, this time for all 9 settings.

Configuration	Bits Per Float	LLaMA-3.1-8B	LLaMA-3.2-3B	LLaMA-3.2-1B
float16	16	9.17	10.75	11.21
KVQuant-4bit-1%	4.16	9.24	10.86	13.41
ChanKV-2-4bit-1%	4.16	9.29	10.94	13.54
TempKV-2-4bit-1%	4.16	**9.23**	**10.85**	**13.38**
KVQuant-2bit-1%	2.16	10.19	12.15	17.43
ChanKV-4-2bit-1%	2.16	10.20	12.12	16.23
TempKV-4-2bit-1%	2.16	**9.90**	**11.72**	**15.27**
KVQuant-1bit-1%	1.16	19.38	24.95	148.66
ChanKV-8-1bit-1%	1.16	16.27	19.32	32.56
TempKV-8-1bit-1%	1.16	**13.49**	**16.20**	**25.39**

Table 7: Perplexity measured on C4 (the lower the better) with 1% outlier removal, on 256 text samples of 2048 tokens. The added 0.16 bits per float is due to the outlier removal.

This table shows the effect of outlier removal : it allows for a decrease in perplexity for all

methods, bringing them closer to the float16 baseline. For example, in the LLaMA 3.1 8B model, Temp-2-4bit has a perplexity going from 9.31 to 9.23 thanks to outlier removal. This is a considerable performance leap considering that the float16 baseline has perplexity 9.17.

We run the same experiments as C4 on the Penn Tree Bank dataset : Table 8 and Table 9 respectively show the results without and with outlier removal. These experiments strengthen the observations made for the C4 experiments : TemporalKV outperforms the baselines on 8 settings out of 9, and the outlier removal brings an important perplexity reduction.

Configuration	Bits Per Float	LLaMA-3.1-8B	LLaMA-3.2-3B	LLaMA-3.2-1B
float16	16	9.63	11.60	15.27
KVQuant-4bit	4.00	9.72	11.76	**15.52**
ChanKV-2-4bit	4.00	9.75	11.78	15.69
TempKV-2-4bit	4.00	**9.71**	**11.73**	15.55
KVQuant-2bit	2.00	11.18	14.16	23.22
ChanKV-4-2bit	2.00	10.63	13.18	19.19
TempKV-4-2bit	2.00	**10.42**	**12.90**	**18.32**
KVQuant-1bit	1.00	43.67	60.88	229.10
ChanKV-8-1bit	1.00	19.14	24.41	47.05
TempKV-8-1bit	1.00	**16.07**	**21.32**	**39.09**

Table 8: Perplexity measured on Penn Tree Bank (the lower the better), on 256 text samples of 2048 tokens. No outlier removal is implemented for any method in this table.

Configuration	Bits Per Float	LLaMA-3.1-8B	LLaMA-3.2-3B	LLaMA-3.2-1B
float16	16	9.63	11.60	15.27
KVQuant-4bit-1%	4.16	**9.68**	11.69	**15.47**
ChanKV-2-4bit-1%	4.16	9.74	11.73	15.69
TempKV-2-4bit-1%	4.16	**9.68**	**11.68**	15.56
KVQuant-2bit-1%	2.16	10.58	13.08	20.47
ChanKV-4-2bit-1%	2.16	10.39	12.86	18.80
TempKV-4-2bit-1%	2.16	**10.22**	**12.56**	**17.94**
KVQuant-1bit-1%	1.16	23.77	38.80	177.51
ChanKV-8-1bit-1%	1.16	16.42	21.22	40.81
TempKV-8-1bit-1%	1.16	**14.23**	**18.63**	**33.79**

Table 9: Perplexity measured on Penn Tree Bank (the lower the better), on 256 text samples of 2048 tokens. The added 0.16 bits per float is due to the outlier removal.

6.4.2 Long-context accuracy comparisons

While perplexity is an important metric and a great indicator of how much a compression method affects LLMs' abilities, in this subsection we measure accuracy on the long-context RULER benchmark. Our aim is to get more informative and straightforward measures of our models' performances in challenging scenarios. We highlight that there is no benchmark on long-context in ChannelKV's paper comparing their method to KVQuant. We believe this is an important limitation of their work : the main motivation of low-bit compression is to serve long-context scenarios, because those are the scenarios where the KV-Cache becomes a substantial burden, as the problem is memory-bandwidth bound. In this subsection, we thus compare KVQuant, ChannelKV and TemporalKV on long-context tasks. We use outlier removal for all the experiments, as it has shown to significantly improve results for all methods.

To avoid cherry-picking datasets, we included every single task from RULER, except the "RULER QA Hotpot" task which was unavailable on LM-Eval [12] when we performed the experiments. In the tables, "QA" refers to the "RULER QA Squad" task.

Table 10 presents the results for LLaMA 3.1 8B, for bit widths of 4, 2 and 1 :

- In the 4-bit case (chunk size of 2), results are very close to the float16 baseline on most tasks, making it hard to designate a better method. On average across all tasks, TemporalKV has a slight edge over ChannelKV and KVQuant.
- In the 2-bit case, TemporalKV clearly outperforms the baselines on almost all tasks except Common/Frequent Word Extraction, where ChannelKV has a slight edge. On all 8 Needle In A Haystack tasks, TemporalKV clearly has better performance: for example, TemporalKV scores 48 % on the Multikey 3 task, while KVQuant and ChannelKV respectively score 5.6 % and 7.6%. On average across all tasks, TemporalKV is 8 accuracy points above ChannelKV, and 10 points above KVQuant.

- In the 1-bit case, TemporalKV outperforms the baselines by a large margin on all tasks except Common and Frequent Word Extraction where results are tight. On average, TemporalKV scores an accuracy more than two times higher than the baselines.

Config	Avg. bit	NIAH1	NIAH2	NIAH3	MKey1	MKey2	MKey3	MValue	MQuery	VT	CWE	FWE	QA	Avg.
fp16 Baseline	16	100	100	100	99.40	100	99.80	99.50	99.80	99.96	94.72	92	55.88	95.09
KVQuant-4bit-1%	4.16	100	100	99.40	98.40	99.80	95.40	**99.95**	**99.90**	99.16	94.02	85.60	51.78	93.62
ChanKV-2-4bit-1%	4.16	100	100	99.80	99	99.80	97.40	98.95	99.80	99.84	95	**90.47**	52.65	94.39
TempKV-2-4bit-1%	4.16	100	100	**100**	**99.40**	99.80	**98.40**	99.45	99.75	**99.84**	**95.20**	90.33	**52.68**	**94.57**
KVQuant-2bit-1%	2.16	100	**99.60**	89	83.40	46.40	5.60	93.55	92.15	89.72	83.16	60.93	40.23	73.65
ChanKV-4-2bit-1%	2.16	100	99.20	82.20	84.60	54.80	7.60	90.05	83.85	95.12	**91.30**	**79.60**	36.23	75.38
TempKV-4-2bit-1%	2.16	100	**99.60**	**98.40**	**87.20**	**70.20**	**48**	**97.95**	**96.50**	**98.40**	90.70	75.87	**41.92**	**83.73**
KVQuant-1bit-1%	1.16	41	5	0	1.20	0	0	0.05	0.05	2.96	13.20	19.80	11.28	7.88
ChanKV-8-1bit-1%	1.16	8	0.60	0	0.80	0	0	0.05	0.05	0.32	**23.66**	**38.73**	12.13	7.03
TempKV-8-1bit-1%	1.16	**43.80**	**22.80**	**12.80**	**7.80**	0	0	**5.85**	**3.80**	**16.32**	22.84	35.47	**21.42**	**16.07**

Table 10: Accuracy results for the LLaMA 3.1 8B model on the RULER benchmark (the higher the better). The context length is set to 8192 due to limits in our computational resources, and for every single task the accuracy is calculated over 500 samples. Bit widths 4 (up), 2 (middle) and 1 (down) are presented. The best method for a given bit width is highlighted in bold.

The results for the LLaMA 3.2 3B model, presented in Table 11, are similar to LLaMA 3.1 8B : accuracy performances are tight in the 4 bit case and close to the float16 baseline, while for the 2 bit case there is a clear edge for TemporalKV. Indeed, TemporalKV scores an average 64,43 % across all tasks, against 57,28 % for KVQuant and 56,00 % for ChannelKV.

Config	Avg. bit	NIAH1	NIAH2	NIAH3	MKey1	MKey2	MKey3	MValue	MQuery	VT	CWE	FWE	QA	Avg.
float16 Baseline	16	100	100	86.60	97.20	99.60	89.80	95.35	99.05	93	23.98	73.60	53.32	84.29
KVQuant-4bit-1%	4.16	99.80	100	82.60	95.40	99.40	83	93.20	96.40	90	26.12	71.80	51.25	82.41
ChanKV-2-4bit-1%	4.16	**100**	100	**88.40**	91.20	99.40	**85.60**	91.85	97.45	89.92	**32.10**	**72.33**	48.75	83.08
TempKV-2-4bit-1%	4.16	**100**	100	87	**96**	**99.80**	84	**93**	**98.05**	**93.16**	28.12	72.20	**51.98**	**83.46**
KVQuant-2bit-1%	2.16	96.80	95.40	77.20	69	36.60	1.20	61.25	64.70	82.68	27.18	41.53	**33.85**	57.28
ChanKV-4-2bit-1%	2.16	99.60	97.20	66.40	64	54.40	0.40	42.25	37.05	85.72	**47.22**	**50.87**	26.93	56.00
TempKV-4-2bit-1%	2.16	**99.80**	**98.20**	**78.40**	**79.40**	**74**	**12.80**	**66.70**	**70**	**88.40**	30.52	44.73	30.22	**64.43**

Table 11: Accuracy results for the LLaMA 3.2 3B model on the RULER benchmark (the higher the better). The context length is set to 8192 due to limits in our computational resources, and for every single task the accuracy is calculated over 500 samples. Bit widths 4 (up) and 2 (down) are presented. The best method for a given bit width is highlighted in bold.

Finally, we report results for the LLaMA 3.2 1B model in Table 12. TemporalKV clearly remains the best performing method on all tasks. On average, in the 4 bit case, the performance degradation of TemporalKV compared to float16 is 5.56 %, while it is 7.52 % for ChannelKV and 21.70 % for KVQuant. In the 2 bit case, TemporalKV outperforms ChannelKV and KVQuant by a large margin, especially on challenging Needle In A Haystack tasks. We bring the reader's attention to the results for NIAH3, where TemporalKV scores 46.60 % while ChannelKV scores 10.60% and KVQuant scores 1.60%. Similarily, TemporalKV has far better performance on the MValue, MQuery and VT tasks.

Config	Avg. bit	NIAH1	NIAH2	NIAH3	MKey1	MKey2	MKey3	MValue	MQuery	VT	CWE	FWE	QA	Avg.
float16 Baseline	16	100	100	83.80	96.60	61.60	37.40	76	94.90	83.04	20.58	58.20	39.58	70.98
KVQuant-4bit-1%	4.16	100	100	56.80	88	1.80	1.20	20.05	58.35	69.92	15.50	47.87	30.72	49.18
ChanKV-2-4bit-1%	4.16	100	100	82.20	88.20	34.80	13	69.30	87.25	82.68	17.60	51.87	34.58	63.46
TempKV-2-4bit-1%	4.16	100	100	**84.60**	**90.80**	**40.2**	**23.20**	**77.65**	**91.95**	**83.32**	**19.30**	**56.60**	**35.65**	**66.21**
KVQuant-2bit-1%	2.16	71	51.60	1.60	15.80	0	0	3.90	4.15	10.64	5.54	27.60	11	16.90
ChanKV-4-2bit-1%	2.16	95.80	75.80	10.60	18.80	0	0	4.25	5.85	33.08	9.32	29.60	13.83	24.74
TempKV-4-2bit-1%	2.16	**97.60**	**96.60**	**46.60**	**46.20**	**0.80**	0	**20.15**	**36.85**	**51.60**	**9.42**	**33**	**17.95**	**38.06**

Table 12: Accuracy results for the LLaMA 3.2 1B model on the RULER benchmark (the higher the better). The context length is set to 8192 due to limits in our computational resources, and for every single task the accuracy is calculated over 500 samples. Bit widths 4 (up) and 2 (middle) are presented. The best method for a given bit width is highlighted in bold.

6.5 Discussion

Experimental results have shown that TemporalKV outperforms the baselines over a wide variety of datasets, including classic perplexity benchmarks like C4 or PTB, and 10 out of 12 long context tasks in the RULER benchmark. Results are tight for the tasks Common and Frequent Word Extraction, where ChannelKV seems to retain a slight edge.

Progresses of TemporalKV are especially visible in the low-bit regime (1 bit and 2 bit), for chunk sizes of 8 and 4. In these settings, TemporalKV outperforms the baselines by a large margin.

Our results are consistent over 3 different state-of-the-art models, with numbers of parame-

ters 1B, 3B and 8B. They also align with our information-theoretic study in Subsection 4.1 : it confirms the intuition that temporal correlations in KV-Cache offer a great compression opportunity which, when carefully exploited, can yield important performance improvements compared to state-of-the-art compression methods. These temporal correlations, when coupled with ChannelKV's chunking on the first layers of LLMs, yield a powerful low-bit compression framework. With calibration on a small subset taken from Wikitext-2, our model learns generalizable compression patterns which outperform state-of-the-art baselines on a majority of tasks.

It is however important to precisely define the scope of our results :

- Our work, just like the baselines KVQuant and ChannelKV, is useful and efficient for bit widths between 1 and 4[10]. However, outside of these bit widths, these methods become either computationally intractable or have low accuracy. Indeed, for bit widths of 8 and above, the number of centroids (¿65536) is computationally intractable, with an encoding step that becomes a bottleneck in the pipeline. For bit widths of 0.5 and below, the precision of the model becomes very low, and has bad performance on long context benchmarks. To our knowledge, no existing work manages to maintain acceptable long-context performance at bit widths this low. This is therefore a limitation that is not specific to our quantization framework.

- Our results were only performed on Meta's LLaMA models. While we chose these models because they are a representative open-source reference in LLM research, more experiments on other LLMs have to be conducted to conclude on the systematic superiority of TemporalKV over the baselines. We were also limited in computational resources, only evaluating on LLMs with numbers of parameters below 8 Billions. More comprehensive evaluation on models with 13B or 70B parameters would make our conclusions more general.

[10]For TemporalKV and ChannelKV, this corresponds to chunk sizes between 2 and 8.

- TemporalKV showcases superiority over the baselines on current LLM architectures by relying on temporal correlations. However, the reach and scope of our work will depend on the evolution of LLMs in the next years : if LLM architectures undergo drastic changes, it is possible that our compression framework either loses (or gains) efficiency. Note that this claim is valid for many current LLM conference publications (including KVQuant and ChannelKV), because they often tend to have an important experimental component which is architecture-dependent. However, transformer architectures have been relatively stable over the past years, always comprising attention layers and the KV-Cache. For this reason, we are optimistic that TemporalKV will continue to provide benefits for low-bit LLM compression over a reasonably long horizon.

7 Future Directions

This final section presents directions for future work that build upon the findings of this thesis.

7.1 Further improving compression

We believe the compression framework proposed in this thesis can be further improved.

First, optimizing the generalizability of compression could improve on our current results. We chose Wikitext-2 as a calibration set because it yielded good generalization on most tasks and was also used in previous works. However, we found that our method could perform worse than the baselines in some isolated cases, for example on RULER's "Common Word Extraction" task. It is likely that the temporal patterns learnt on Wikitext-2 have suffered a distribution shift for the "Common Word Extraction" task, causing our method to perform worse. To mitigate this issue, we could try to use a different calibration set with data from multiple different datasets, instead of Wikitext-2 which only consists of articles from Wikipedia.

Also, we believe that exploring other normalization techniques may improve TemporalKV. The objective is to find normalization constants striking a balance between accuracy and memory space/computational cost. Our current normalization is simple, with per-channel statistics computed offline on Wikitext-2. More accurate variants could include statistics computed online, or combining per-token normalization with per-channel normalization.

Regarding the quantization method used, Product Quantization, the literature provides variants which could achieve relatively important Mean Squared Error reduction. In particular, Optimized Product Quantization (OPQ) [13] is a well-known alternative. Instead of simply concatenating a vector into adjacent chunks, OPQ optimizes an orthogonal projection matrix applied to the vector in order to find the optimal chunking to minimize reconstruction error.

The downside is that the orthogonal projection would slow down our pipeline. Nevertheless, this remains an interesting direction for future work.

7.2 Centroids : computational and memory cost

7.2.1 Increasing the number of centroids

As previously stated, the choice of a number of 256 centroids was made because it is the maximum number such that centroid indices can be efficiently represented with 8-bit integers. The next type of int, int16, would correspond to 65536 centroids which appears computationally intractable in our current pipeline. Indeed, following Algorithm 3, it implies calculating 65536 distances for every chunk which would considerably slow down our forward passes. However, using a different assignment algorithm, we could potentially efficiently assign chunks to the closest centroid among 65536. This could involve using an approximate assignment algorithm. For example, for chunks of size 2 or 4, KD-Trees and Ball-Trees are usually very efficient and have much lesser complexity than the brute-force Algorithm 3. There exists efficient GPU implementations for KD-Trees [37] representing a promising research avenue for extending our framework. We could then compress longer chunks for a same bit width per float, which would further improve TemporalKV's performance.

7.2.2 Centroid sharing

In our experiments, we used centroid sharing between adjacent channels to maintain the same memory footprint as ChannelKV. This is possible because all channels share the same diagonal compression pattern that we showed in Subsection 4.1. Unlike TemporalKV, ChannelKV's correlations are channel-specific which makes centroid sharing impossible. Therefore, TemporalKV has a unique flexibility in choosing how many channels centroids are shared among. Thus, the memory footprint of TemporalKV can be modulated by sharing

centroids among more or less channels. This could be exploited for certain LLMs where the memory footprint of centroids becomes too high.

7.3 Leveraging heterogeneous compression to enhance efficiency

When performing our experiments, we observed that compression errors highly differed between different layers, different heads inside a same layer, but also different channels inside a same head. Thus, the compression errors could be highly reduced by accounting for this heterogeneity.

Currently, our hybrid method uses ChannelKV's chunking for some early layers of the LLM. The choice of these layers is made by evaluating the average Fisher-Weighted MSE per layer and then assigning each layer to a chunking method. We could use the same method at a higher granularity by using the per-head breakdown (see Figure 23) to assign each head to a different chunking method. However, the parallelization of the compression pipeline on the GPU would then be trickier because two different chunking methods would be mixed up, whereas we currently just batch all heads together.

Another direction of research would consist in using a mixed-precision KV-Cache, assigning different compression budgets to different layers (Similarily to MatryoshkaKV [26]) or channels (See [39]).

7.4 Understanding the cause of temporal correlations to better understand LLMs

More broadly, the existence of a diagonal pattern and temporal correlations in the KV-Cache raises questions about the architecture of current Large Language Models.

One could first ask why this diagonal pattern emerges, i.e. why adjacent tokens tend to have similar channel values. Interestingly, we found that temporal correlations are quite low in

the earlier layers of the LLMs, but become higher after 3 or 4 layers[11]. We see two potential causes for this emergence after a few layers. First, adjacent tokens have similar positional encodings (RoPE) in attention layers, which after a few layers might cause their embeddings to grow similar. Second, the attention sinking phenomenon might be related to this diagonal pattern. Indeed, attention sinks often concentrate a high percentage of the attention weight. This could lead to adjacent tokens' outputs depending disproportionately on the sinks' value vectors, thus growing similar after a few layers.

Verifying these hypotheses through theoretical or empirical analyses could provide important insights into how LLMs process adjacent tokens. A semantic study could also attempt to integrate this phenomenon in an interpretability analysis.

[11]This is proven in Figure 14. The entropy of TemporalKV is very similar to that of KVQuant in the earlier layers, showing that the temporal chunking has a little effect, i.e. that adjacent tokens have channel values that are dissimilar. But in later layers, the entropy of TemporalKV is clearly below KVQuant's, showing that the temporal correlations are higher.

Conclusion

In conclusion, our preliminary experiments and their consistency with strong experimental results establish TemporalKV as a competitive low-bit KV Cache compression method. For 3 different models, our experiments show that it outperforms the state-of-the-art baselines over a wide variety of settings, including perplexity measurements and long-context benchmarks.

TemporalKV further pushes the quality of low-bit quantization, showcasing high accuracy in long context settings while reducing the KV-Cache's size by a factor of 4 to 16. For a given RAM budget, TemporalKV thus allows to process sequences 4 to 16 times longer than with standard attention, while maintaining high accuracy. While wider experimentation on larger models is necessary to conclude on how general our findings are, we regard the exploitation of temporal correlations as a promising direction in KV-Cache quantization.

More broadly, TemporalKV is our contribution to the ongoing research effort to enhance the memory efficiency of modern AI systems. For low-precision KV-cache models to gain adoption, they must retain strong performance relative to their more energy-intensive counterparts. By increasing the accuracy of long-context models in the low-precision regime, we hope that TemporalKV can participate in making less energy-intensive models a more practical alternative.

A A potential substantial speedup with a custom Cuda Kernel

We used a high-level PyTorch implementation in our work, choosing to focus on optimizing the accuracy and perplexity of our method. A natural follow-up to our work is the implementation of a custom CUDA kernel allowing to fuse all the decompression and attention computation steps, as done in FlashAttention [9]. Attention being mostly memory-bandwidth bound for relatively small batch sizes [17], it is necessary to obtain an "IO-aware" algorithm, i.e. an algorithm that limits the number of reads and writes between different levels of GPU memory. To be more precise, we represent the typical memory hierarchy of a GPU on Figure 20.

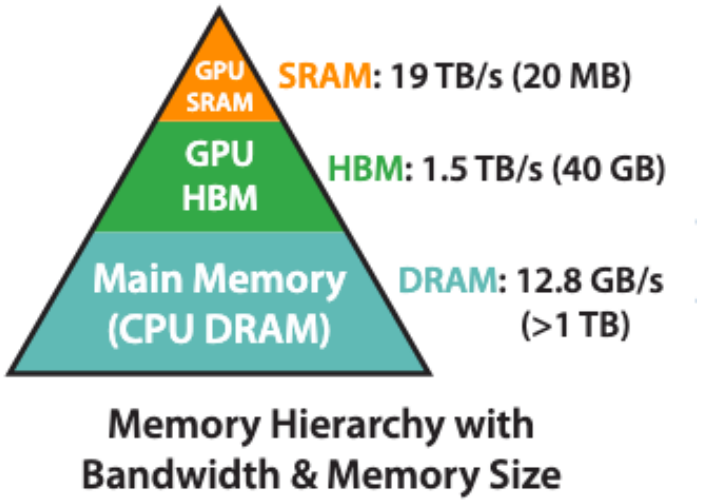


Figure 20: Typical Memory Hierarchy of a GPU (Taken from the Flash Attention paper). The memory bandwidth on the SRAM is typically 10 times higher than on HBM, but SRAM has much lower storage.

The bandwidth of the SRAM being 10 times higher than that of HBM, it is crucial to reduce the number of reads and writes between HBM and SRAM. The compression of incoming key and value states (*encode* step in Figure 7) is usually not the main bottleneck, because it only has to be done once and the compressed keys and values then stay in the KV Cache for

the remainder of the computation. However, decompression (*decompress* step in Figure 7) becomes a main bottleneck because the whole KV Cache must be decompressed for the generation of each token. To prevent this bottleneck, we present a framework for decompression and attention computation on Figure 21.

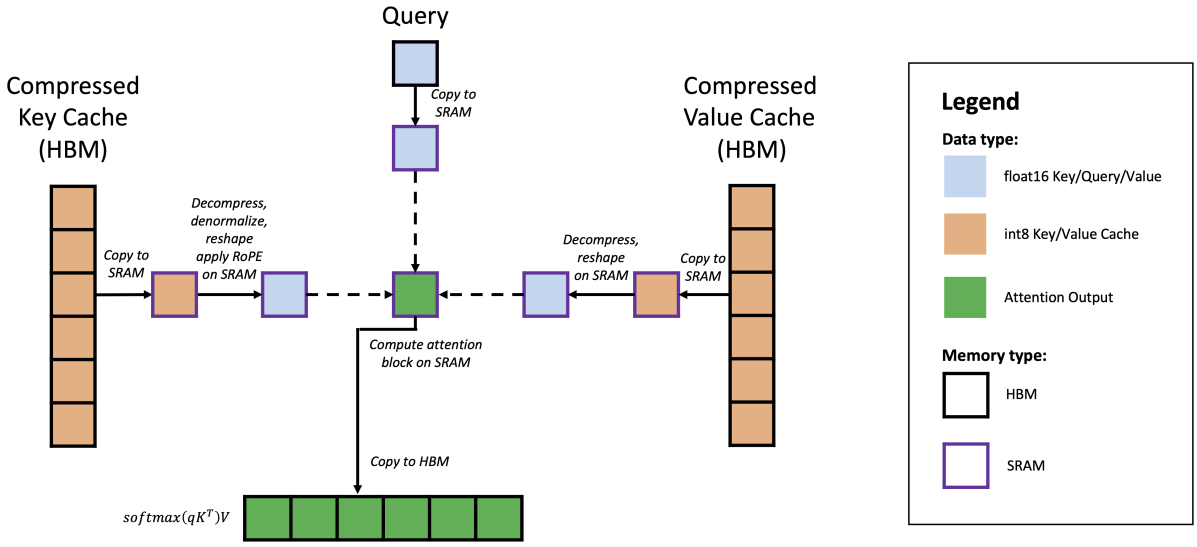


Figure 21: Framework to perform tiled attention using our KV Cache compression method. The Figure was inspired by an illustration in FlashAttention's paper.

In classic FlashAttention, the whole KV Cache in precision float16 must be copied from GPU HBM to GPU SRAM. This represents a considerable bottleneck because the problem is memory-bandwidth bound, and the KV Cache can take up significant memory. With an appropriate CUDA kernel, TemporalKV would be much faster because what it would copy from HBM to SRAM is the compressed KV Cache instead of the float16 one, the compressed KV Cache being 4x to 16x smaller thanks to our compression method (4 bits per float in the TempKV-2-4bit configuration, 1 bit per float in the TempKV-8-1bit configuration). We would then perform decompression, denormalization, reshaping and RoPE directly on the SRAM, before computing the attention block on SRAM as well.

Because SRAM has a capacity of only 20MB, it is impossible to decompress the KV Cache in SRAM all at once. Instead, like in FlashAttention, we can tile the computation and

compute attention block by block, each block being a subset of the compressed keys and their corresponding values. To do so, one needs to fuse the whole decompression pipeline with the attention kernel in a CUDA kernel, before copying the partial attention output to HBM.

Note that this would only be possible because the operations in our decompression pipeline are easily tiled across the whole Key and Value matrixes, like in FlashAttention. Indeed, denormalization is applied float by float as it only consists of a multiplication and an addition. The reshaping step can be applied to tiled data as well, thus being doable on SRAM. RoPE can be similarily tiled as it applies small matrix multiplications to chunks of size 2. Also, the decompression step can easily be tiled as it is performed chunk by chunk, and centroids can fit in the GPU's L2 cache : even in the worst case (LLaMA 2 7B with 32 heads and 32 layers), centroids take up approximately 4 MB per layer, which easily fits on the L2 Cache's 40MB for an A100 GPU.

Note that we cannot fully parallelize block-by-block because the softmax requires a division by a sum of exponentials where all dot-products have to be known. This challenge is however solved in FlashAttention's paper to which we refer for further detail [9].

B Centroid storage overhead

We represent the memory storage overhead of our method in Table 13. For a given model, all configurations have the same overhead because we use centroid sharing across different channels. Using less centroid sharing, we could trade some memory overhead for better accuracy.

We find that our method only adds 1% of memory overhead relatively to the model weights, which can be seen as negligible given the memory gain obtained from KV Cache quantization.

Config	**Model**			
	LLaMA 2 7B	*LLaMA 3.1 8B*	*LLaMA 3.2-1B*	*LLaMA 3.2 3B*
TempKV-8-1bit	132,2 MB (0,96 %)	33,6 MB (0,21 %)	8,4 MB (0,42 %)	29,4 MB (0,49 %)
TempKV-4-2bit	132,2 MB (0,96 %)	33,6 MB (0,21 %)	8,4 MB (0,42 %)	29,4 MB (0,49 %)
TempKV-2-4bit	132,2 MB (0,96 %)	33,6 MB (0,21 %)	8,4 MB (0,42 %)	29,4 MB (0,49 %)

Table 13: Centroid storage overhead. The percentage corresponds to the ratio of the centroid memory overhead by the memory of the LLM weights.

C Latency measurements

We report in Table 14 the latencies measured on a LLaMA-3.1-8B model, both in the prefilling and decoding stages. Note that these latencies are obtained with high-level PyTorch which requires to rewrite the decompressed matrix on GPU RAM. This is highly suboptimal and constrains TemporalKV to be slower than the float16 baseline. A CUDA kernel would allow TemporalKV to be faster than standard attention by alleviating the memory bottleneck.

	Prefill Time (s)	Per-token Decoding Time (s)
float16	0.14	0.034
TempKV-8-1bit	0.24	0.042
TempKV-4-2bit	0.31	0.042
TempKV-2-4bit	0.44	0.043

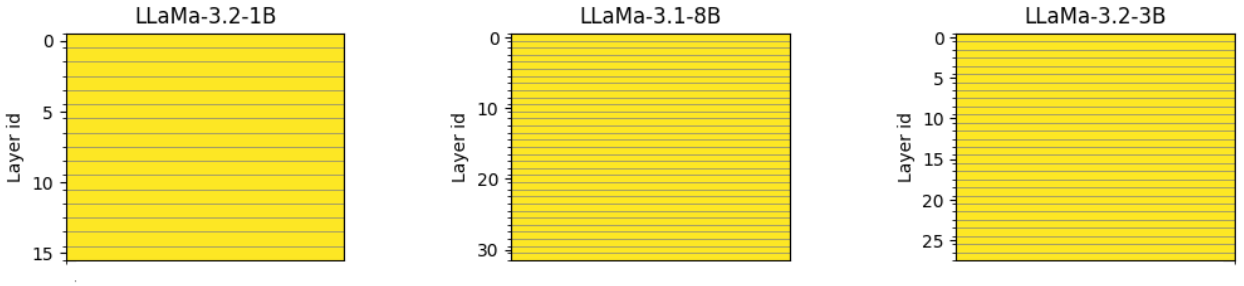
Table 14: Latency measurements on a LLaMA-3.1-8B model for a context length of 2048, on an H100 GPU. We averaged the values over 64 text samples and generated 50 tokens for each of them.

D Mean Squared Error comparison

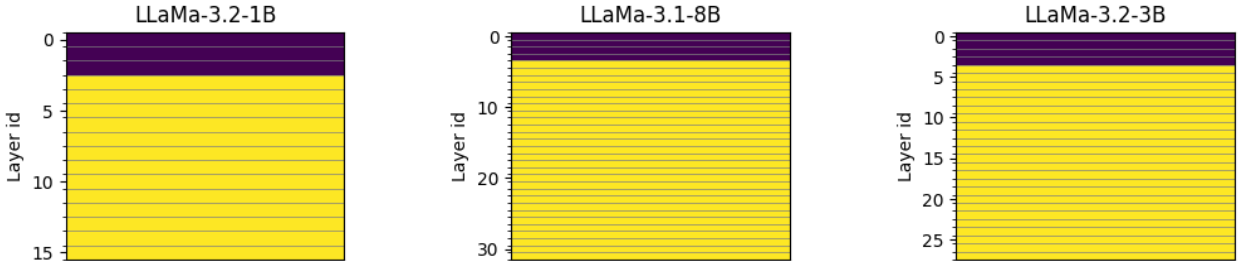
In this section of the appendix, we represent which of TemporalKV's and ChannelKV's chunking is most efficient. We represent both a per-layer (Figure 22) and a per-head breakdown in Figure 23, including both Values and Keys compression.

We find that for a chunk size of 2, temporal correlations outperform ChannelKV on exactly all layers across all models. This is consistent with our information-theoretic study in Figure 13, where it was clear that ChannelKV gained almost no more information efficiency compared to KVQuant when coupling 2 channels only. This is a very important result, because ChannelKV's paper offered very little progress on the setup with a bit width of 4 (chunk size of 2), whereas TemporalKV is clearly much more efficient.

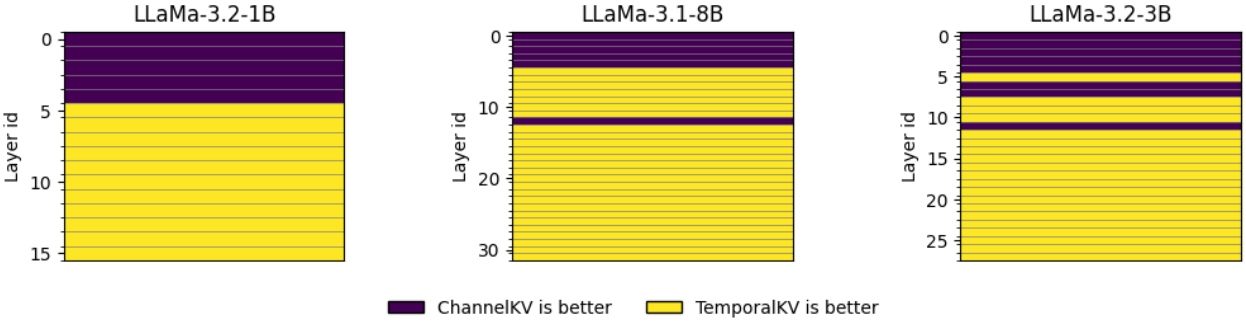
For a chunk size of 4 and 8, results are more balanced on the first layers which have low effective rank. Temporal correlations are usually outperformed by ChannelKV on the first 3 to 5 layers. This is due to the fact that for low-rank weight matrixes, correlations span across many dimensions and become especially meaningful as the chunk size gets bigger. ChannelKV would probably become particularly useful for chunk sizes like 16 or 32, but these correspond to extreme compression scenarios of respectively 0.5 and 0.25 bits per float, which are in practice never used because the LLM then has poor performance. This is why TemporalKV is so useful and efficient : it exploits the most efficient correlations (temporal ones) at the very precision levels (4 bits, 2 bits and 1 bit per float) which are most



(a) Chunk size of 2



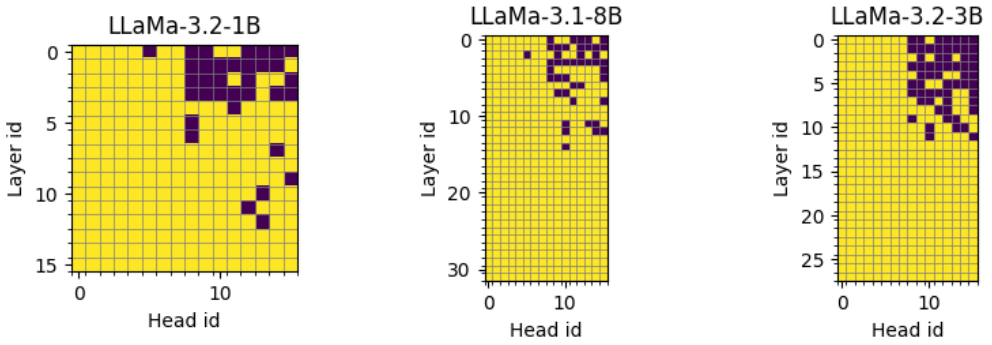
(b) Chunk size of 4



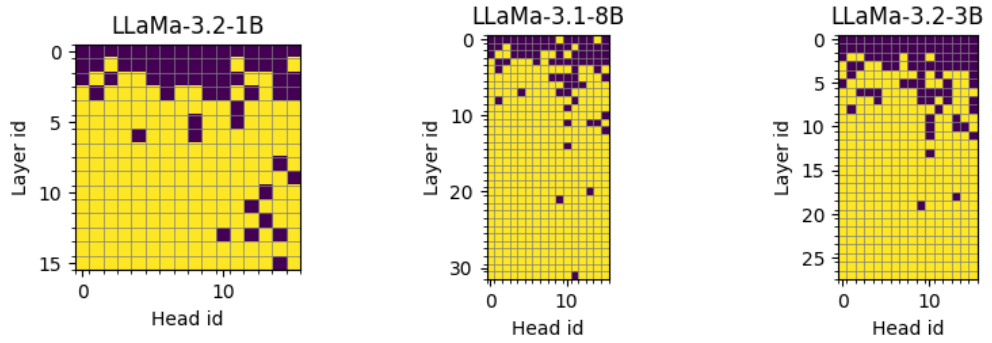
(c) Chunk size of 8

Figure 22: Per-layer breakdown of the quantization method with lowest Fisher-weighted Mean Squared Error on 3 LLaMA models, for chunk sizes 2 (up), 4 (middle) and 8 (down). Layers where ChannelKV's chunking has lower MSE are represented in purple, and layers where TemporalKV's chunking has lower MSE are represented in yellow.

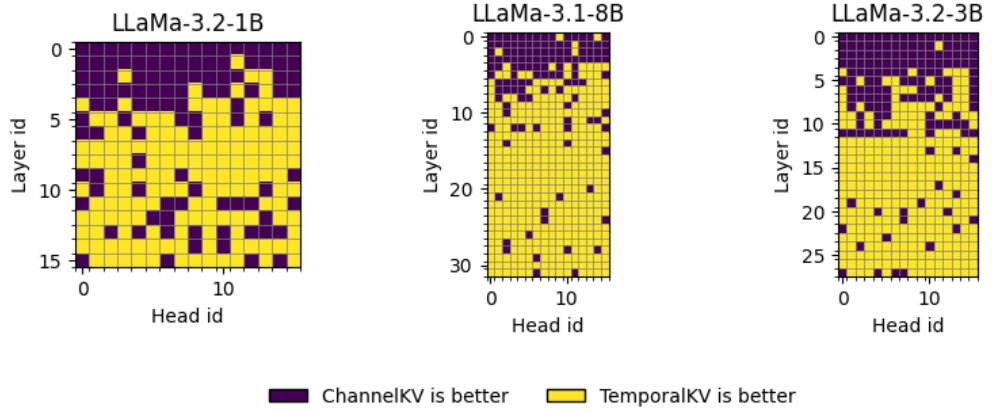
used in practice.



(a) Chunk size of 2



(b) Chunk size of 4



(c) Chunk size of 8

Figure 23: Per-head breakdown of the quantization method with lowest Fisher-weighted Mean Squared Error on 3 LLaMA models. The first 8 head ids correspond to keys, the last 8 head ids correspond to values. Heads where ChannelKV's chunking has lower MSE are represented in purple, and Heads where TemporalKV's chunking has lower MSE are represented in yellow.

E Handling newly generated tokens in TemporalKV

Because TemporalKV compresses multiple adjacent tokens together, newly generated tokens cannot be compressed one by one. More precisely, we need as many tokens as the chunk size to be generated before compressing these newly generated tokens together as a chunk. Thus, in practice TemporalKV uses a buffer that keeps newly generated tokens' keys and values in half-precision. This buffer has the size of the chunk size : for a chunk size of 4, the buffer keeps at most 4 tokens' keys and values. Once the buffer is full (i.e. once 4 tokens have been generated), the 4 tokens' keys and values are compressed together and sent to the KV-Cache in low-precision. The step is repeated for every newly generated 4 tokens.

To ensure fair comparison with the baselines in the experiments, for datasets requiring token generation we systematically keep a same local window in half-precision. Since this window is small compared to the context size (less than 10 for sequences of several thousands tokens), the impact on quantization precision is in practice negligible.

F An initial research direction we later abandoned : Sparse Attention

In this section of the appendix, we briefly present our initial research direction, which we later abandoned. Instead of using compression, we were exploring sparse attention : this line of work, rather than compressing the KV-Cache, keeps it in full precision and only retrieves a subset of keys (and their corresponding values) at every token generation. The main advantage of this approach is that the subset of keys is usually a small percentage of the whole KV-Cache, therefore the memory to copy from HBM to SRAM is greatly reduced. This allows to address the memory bottleneck of the KV-Cache.

These methods work because attention is typically sparse, with a few keys taking up the majority of attention weights [28]. However, the main challenge consists in determining, for a given query, the specific keys which concentrate attention weight. To do so, nearest-neighbor algorithms are often used in the literature. But as we described in Subsection 3.2, the Out-Of-Distribution problem is a major obstacle between keys and queries : when keys and queries follow different distributions, nearest-neighbor algorithms in the literature have poor performance.

In order to address this problem, we derived a theoretical mathematical method to align the first and second-order moments of keys and queries.

We assume keys are samples from a distribution $k \sim \mathcal{N}_k$ and queries are samples from $q \sim \mathcal{N}_q$. We denote (μ_k, Σ_k) and (μ_q, Σ_q) the means and variances of these distributions[12]. Because $\mathcal{N}_k$ and $\mathcal{N}_q$ are typically different, the previously described "Out-of-Distribution" problem arises : it is not possible to efficiently apply highest dot-product search between keys and queries with off-the-shelf nearest-neighbor algorithms [20].

[12]We assume that the variance matrixes are definite positive, as it is the case in LLM applications

Thus, we want to reformulate the expression of attention weights $w_i = \frac{\exp\left(q^\top k_i/\sqrt{d}\right)}{\sum_{j=1}^n \exp\left(q^\top k_j/\sqrt{d}\right)}$ with dot-products from similar distributions. More precisely, we want to avoid dot-products between q and k, and find functions $f_q : \mathbb{R}^d \to \mathbb{R}^d$ and $f_k : \mathbb{R}^d \to \mathbb{R}^d$ such that $w_i = \frac{\exp\left(f_q(q)^\top f_k(k_i)/\sqrt{d}\right)}{\sum_{j=1}^n \exp\left(f_q(q)^\top f_k(k_j)/\sqrt{d}\right)}$ and the distributions of $f_q(q)$ and $f_k(k)$ are similar enough.

Without further assumptions on the distributions $\mathcal{N}_k$ and $\mathcal{N}_q$, we target $f_q(q)$ and $f_k(k)$ to have same means and variances. Note that for any vector $v \in \mathbb{R}^d$, and any symmetric positive definite matrix $P \in \mathbb{R}^{d\times d}$:

$$\begin{aligned}
w_i &= \frac{\exp\left(q^\top k_i/\sqrt{d}\right)}{\sum_{j=1}^n \exp\left(q^\top k_j/\sqrt{d}\right)} \\
&= \frac{\exp\left(q^\top k_i/\sqrt{d}\right)}{\sum_{j=1}^n \exp\left(q^\top k_j/\sqrt{d}\right)} \\
&= \frac{\exp\left(q^\top P^{-1/2}P^{1/2}k_i/\sqrt{d}\right)}{\sum_{j=1}^n \exp\left(q^\top P^{-1/2}P^{1/2}k_j/\sqrt{d}\right)} \\
&= \frac{\exp\left((P^{-1/2}q)^\top P^{1/2}k_i/\sqrt{d}\right)}{\sum_{j=1}^n \exp\left((P^{-1/2}q)^\top P^{1/2}k_j/\sqrt{d}\right)} \\
&= \frac{\exp\left[(P^{-1/2}q)^\top \left(P^{1/2}k_i + v\right)/\sqrt{d}\right]}{\sum_{j=1}^n \exp\left[(P^{-1/2}q)^\top \left(P^{1/2}k_j + v\right)/\sqrt{d}\right]}
\end{aligned}$$

$P^{-1/2}q$ has mean and variance $(P^{-1/2}\mu_q, P^{-1/2}\Sigma_q P^{-1/2})$, while $P^{1/2}k + v$ has mean and variance $(P^{1/2}\mu_k + v, P^{1/2}\Sigma_k P^{1/2})$. We thus aim to find a symmetric positive definite matrix P and a vector v such that :

$$\begin{cases}
P^{-1/2}\mu_q = P^{1/2}\mu_k + v \\
P^{-1/2}\Sigma_q P^{-1/2} = P^{1/2}\Sigma_k P^{1/2}
\end{cases}$$

Which is equivalent to :

$$\begin{cases} P^{-1/2}\mu_q = P^{1/2}\mu_k + v \\ \Sigma_q = P\Sigma_k P \end{cases}$$

One simple solution to this problem is :

$$\begin{cases} P = \Sigma_k^{-1/2}\left(\Sigma_k^{1/2}\Sigma_q\Sigma_k^{1/2}\right)^{1/2}\Sigma_k^{-1/2} \\ v = P^{-1/2}\mu_q - P^{1/2}\mu_k \end{cases}$$

And P is indeed symmetric definite positive, because Σ_k and Σ_q are necessarily symmetric definite positive.

To conclude, we have found functions $f_q(q) = P^{-1/2}q$ and $f_k(k) = P^{1/2}k + v$ such that $f_q(q)$ and $f_k(k)$ follow distributions with same means and variances, and $w_i = \frac{\exp\left(f_q(q)^\top f_k(k_i)/\sqrt{d}\right)}{\sum_{j=1}^{n}\exp\left(f_q(q)^\top f_k(k_j)/\sqrt{d}\right)}$. At the time, we hypothesized that this reformulation could allow to mitigate the "Out-Of-Distribution" problem, since mapped keys $f_k(k)$ and queries $f_q(q)$ now follow distributions with same means and variances. We thought that applying nearest-neighbor search to $f_k(k)$ and $f_q(q)$ instead of k and q would yield much better results.

In practice, aligning the first and second-order moments was too superficial to completely mitigate the Out-Of-Distribution problem, because the distributions of Keys and Queries are too complex to be characterized by these statistics. On top of that, the application of RoPE caused distribution shifts making our theoretical framework fragile. Because we had more promising preliminary results on KV-Cache compression, we thus decided to shift our attention towards the design of TemporalKV.

References

[1] Rami Al-Rfou, Dokook Choe, Noah Constant, Mandy Guo, and Llion Jones. Character-level language modeling with deeper self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 3159–3166. AAAI Press, 2019.

[2] David Arthur and Sergei Vassilvitskii. k-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '07, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[3] Federico Barbero, Álvaro Arroyo, Xiangming Gu, Christos Perivolaropoulos, Michael Bronstein, Petar Veličković, and Razvan Pascanu. Why do llms attend to the first token? *arXiv preprint arXiv:2504.02732v4*, 2025. Submitted April 3, 2025; latest revision August 5, 2025.

[4] Iz Beltagy, Matthew E Peters, and Arman Cohan. Longformer: The long-document transformer. *arXiv preprint arXiv:2004.05150*, 2020.

[5] Zhuoming Chen, Ranajoy Sadhukhan, Zihao Ye, Yang Zhou, Jianyu Zhang, Niklas Nolte, Yuandong Tian, Matthijs Douze, Leon Bottou, Zhihao Jia, and Beidi Chen. Magicpig: Lsh sampling for efficient llm generation. *CSAI*, 2024.

[6] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. In *Proceedings of the 36th International Conference on Machine Learning*, pages 855–864. PMLR, 2019.

[7] Krzysztof Choromanski, Valerii Likhosherstov, David Dohan, Xingyou Song, Andreea Gane, Tamas Sarlos, Peter Hawkins, Jared Davis, Afroz Mohiuddin, Lukasz Kaiser, David Belanger, Lucy Colwell, and Adrian Weller. Rethinking attention with performers. In *International Conference on Learning Representations*, 2021.

[8] Zihang Dai, Zhilin Yang, Yiming Yang, Jaime Carbonell, Quoc V. Le, and Ruslan Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 2978–2988. Association for Computational Linguistics, 2019.

[9] Tri Dao, Daniel Y Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. Flashattention: Fast and memory-efficient exact attention with io-awareness. In *Advances in Neural Information Processing Systems*, 2023.

[10] DeMoriarty. fast_pytorch_kmeans: A pytorch implementation of k-means clustering. `https://github.com/DeMoriarty/fast_pytorch_kmeans`, 2025. Commit or release referenced, if applicable.

[11] Matthijs Douze, Alexandr Guzhva, Chengqi Deng, Jeff Johnson, Gergely Szilvasy, Pierre-Emmanuel Mazaré, Maria Lomeli, Lucas Hosseini, and Hervé Jégou. The faiss library, 2025.

[12] Leo Gao, Jonathan Tow, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Kyle McDonell, Niklas Muennighoff, Jason Phang, Hailey Schoelkopf, and Lintang Sutawika. Lm-evaluation-harness: A framework for evaluating autoregressive language models.

[13] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8, 2013.

[14] Yuval Gersho and Robert M. Gray. Residual vector quantization of speech parameters. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 30(4):770–780, 1982.

[15] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W. Mahoney, and Kurt Keutzer. AI and Memory Wall. RISELab Medium Blog Post, March 2021. Available online.

[16] Dan Hendrycks and Kevin Gimpel. Gaussian error linear units (gelus). *arXiv preprint*, arXiv:1606.08415, 2016. Introduces GELU and mentions the same x(x) function that later became known as SiLU.

[17] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. Kvquant: Towards 10 million context length llm inference with kv cache quantization. In *Advances in Neural Information Processing Systems (NeurIPS) 38*, January 2024. arXiv preprint arXiv:2401.18079.

[18] Cheng-Ping Hsieh, Simeng Sun, Samuel Kriman, Shantanu Acharya, Dima Rekesh, Fei Jia, Yang Zhang, and Boris Ginsburg. Ruler: What's the real context size of your long-context language models? *arXiv preprint arXiv:2404.06654*, 2024.

[19] Hervé Jégou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, 2011.

[20] J. Johnson, M. Douze, and H. Jégou. Faiss: A library for efficient similarity search and clustering of dense vectors. `https://github.com/facebookresearch/faiss`, 2017. Accessed: 2025-02-22.

[21] Hao Kang, Qingru Zhang, Souvik Kundu, Geonhwa Jeong, Zaoxing Liu, Tushar Krishna, and Tuo Zhao. GEAR: An efficient kv cache compression recipe for near-lossless generative inference of LLM. *arXiv preprint arXiv:2406.04529*, 2024.

[22] Jikun Kang, Wenqi Wu, Filippos Christianos, Alex J. Chan, Fraser Greenlee, George Thomas, Marvin Purtorab, and Andy Toulis. Lm2: Large memory models. *arXiv preprint arXiv:2502.06049*, 2025. Convergence Labs Ltd.

[23] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. *Physical Review E*, 69(6):066138, 2004.

[24] Xiaoyang Li, Shaojie Huang, Yuxuan Song, Xinyun Zhan, Junxian He, Caiming Xiong, and Graham Neubig. Efficient streaming language models with attention sinks. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

[25] Bokai Lin, Zihao Zeng, Zipeng Xiao, Siqi Kou, Tianqi Hou, Xiaofeng Gao, Hao Zhang, and Zhijie Deng. Matryoshkakv: Adaptive kv compression via trainable orthogonal projection. *arXiv*, abs/2410.14731, 2024. also accepted as a poster at ICLR 2025.

[26] Bokai Lin, Zihao Zeng, Zipeng Xiao, Siqi Kou, Tianqi Hou, Xiaofeng Gao, Hao Zhang, and Zhijie Deng. Matryoshkakv: Adaptive kv compression via trainable orthogonal projection, 2025.

[27] Felix Lindner, Aditya Grover, and Dan Alistarh. Heavy hitter oracle for efficient attention. In *Proceedings of the 41st International Conference on Machine Learning (ICML)*, 2024.

[28] Di Liu, Meng Chen, Baotong Lu, Huiqiang Jiang, Zhenhua Han, Qianxi Zhang, Qi Chen, Chengruidong Zhang, Bailu Ding, Kai Zhang, Chen Chen, Fan Yang, Yuqing Yang, and Lili Qiu. Retrievalattention: Accelerating long-context llm inference via vector retrieval. 2024.

[29] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. KIVI: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2406.07752*, 2024.

[30] Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. Building a large annotated corpus of english: The penn treebank. *Computational Linguistics*, 19(2):313–330, 1993.

[31] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models, 2016.

[32] Roman Novak, Yasaman Bahri, Daniel A. Abolafia, Jeffrey Pennington, and Jascha Sohl-Dickstein. Sensitivity and generalization in neural networks: an empirical study, 2018.

[33] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019.

[34] Guo Peng, Jian Ma, and Chao Zhang. Attention is naturally sparse with gaussian distributed input. *arXiv preprint arXiv:2109.03094*, 2021.

[35] Jack W. Rae, Anna Potapenko, Siddhant M. Jayakumar, and Timothy P. Lillicrap. Compressive transformers for long-range sequence modelling. In *International Conference on Learning Representations (ICLR)*, 2020.

[36] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *arXiv e-prints*, 2019.

[37] John Robinson. Kdtreegpu: A gpu-based implementation of a k-d tree builder. `https://github.com/johnarobinson77/KdTreeGPU`, 2025. Accessed: 2025-08-20.

[38] Olivier Roy and Martin Vetterli. The effective rank: A measure of effective dimensionality. In *Proceedings of the 15th European Signal Processing Conference (EUSIPCO)*, pages 606–610. IEEE, 2007.

[39] Utkarsh Saxena, Sayeh Sharify, Kaushik Roy, and Xin Wang. Resq: Mixed-precision quantization of large language models with low-rank residuals, 2025.

[40] Arman Shehabi, Sarah J. Smith, Alex Hubbard, Alexander Newkirk, Nuoa Lei, Md AbuBakar Siddik, Billie Holecek, Jonathan G. Koomey, Eric R. Masanet, and Dale A. Sartor. 2024 United States Data Center Energy Usage Report. Technical Report LBNL-2001637, Lawrence Berkeley National Laboratory, December 2024.

[41] Jianlin Su, Yu Lu, Shengfeng Pan, Bo Wen, and Yunfeng Liu. Roformer: Enhanced transformer with rotary position embedding. *arXiv preprint arXiv:2104.09864*, 2021.

[42] Yi Tay, Rami Al-Rfou, Vinh Q. Tran, Zhen Qin, Chung-Ching Lin, Sebastian Ruder, Zhenzhong Lan, William Fedus, Sharan Narang, and Quoc V. Le. Infini-attention: Long-sequence modeling without truncation. *arXiv preprint arXiv:2402.08716*, 2024.

[43] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems (NeurIPS)*, volume 30, 2017.

[44] Ronald J Williams and David Zipser. Gradient-based learning algorithms for recurrent networks and their computational complexity. *Back-propagation: Theory, Architectures, and Applications*, pages 433–486, 1995.

[45] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online, October 2020. Association for Computational Linguistics.

[46] Yuhuai Wu, Markus N. Rabe, DeLesley Hutchins, and Christian Szegedy. Memorizing transformers. *arXiv preprint arXiv:2204.05686*, 2022.

[47] Yunyang Xiong, Zhanpeng Zeng, Rohan Chakraborty, Mohamed Elhoseiny, Ankit B. Singh, and Steven C.H. Hoi. Nyströmformer: A nyström-based algorithm for approximating self-attention. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 14138–14147, 2021.

[48] Jingyang Yuan, Huazuo Gao, Damai Dai, Junyu Luo, Liang Zhao, Zhengyan Zhang, Zhenda Xie, Y. X. Wei, Lean Wang, Zhiping Xiao, Yuqing Wang, Chong Ruan, Ming Zhang, Wenfeng Liang, and Wangding Zeng. Native sparse attention: Hardware-aligned and natively trainable sparse attention. *arXiv preprint arXiv:2502.11089*, 2025.

[49] Biao Zhang and Rico Sennrich. Root mean square layer normalization. In *Advances in Neural Information Processing Systems*, volume 32 of *NeurIPS 2019*, pages 12360–12371. Curran Associates, Inc., 2019. 33rd Conference on Neural Information Processing Systems, Vancouver, Canada, December 2019.

[50] Tianyi Zhang, Jonah Yi, Zhaozhuo Xu, and Anshumali Shrivastava. Kv cache is 1bit-perchannel: Efficient large language model inference with coupled quantization. In *Advances in Neural Information Processing Systems 37 (NeurIPS 2024)*, 2024. NeurIPS main track.

[51] Yang Zhang, Yifan Chen, Ledell Wu, Wei Li, Ming Zhou, Jimmy Lin, Jiasheng Tang, Xiao Lin, and Liwei Wang. Streamingllm: Empowering language models with streaming capability. *arXiv preprint arXiv:2403.05530*, 2024.

[52] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. ATOM: Low-bit quantization for efficient and accurate LLM serving. *arXiv preprint arXiv:2406.07634*, 2024.