TURBOATTENTION: EFFICIENT ATTENTION APPROXIMATION FOR HIGH THROUGHPUTS LLMS

Hao Kang [1 2] **Srikant Bharadwaj** [1] **James Hensman** [1] **Tushar Krishna** [2] **Victor Rühle** [1] **Saravan Rajmohan** [1]

ABSTRACT

Large language model (LLM) inference demands significant amount of computation and memory, especially in the key attention mechanism. While techniques, such as quantization and acceleration algorithms, like FlashAttention, have improved efficiency of the overall inference, they address different aspects of the problem: quantization focuses on weight-activation operations, while FlashAttention improves execution but requires high-precision formats. Recent Key-value (KV) cache quantization reduces memory bandwidth but still needs floating-point dequantization for attention operation.

We present *TurboAttention*, a comprehensive approach to enable quantized execution of attention that simultaneously addresses both memory and computational efficiency. Our solution introduces two key innovations: *FlashQ*, a headwise attention quantization technique that enables both compression of KV cache and quantized execution of activation-activation multiplication, and Sparsity-based Softmax Approximation (*SAS*), which eliminates the need for dequantization to FP32 during exponentiation operation in attention. Experimental results demonstrate that TurboAttention achieves 1.2-1.8x speedup in attention, reduces the KV cache size by over 4.4x, and enables up to 2.37x maximum throughput over the FP16 baseline while outperforming state-of-the-art quantization and compression techniques across various datasets and models.

1 INTRODUCTION

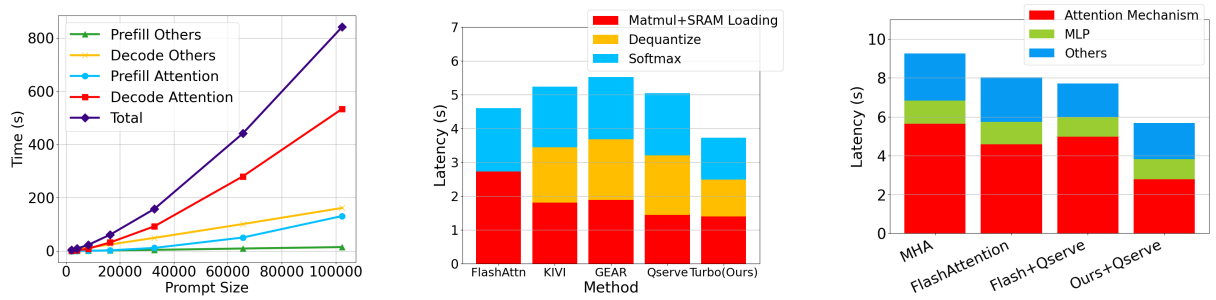
Large language models (LLMs) (Touvron et al., 2023; Gunasekar et al., 2023; Brown et al., 2020) have excelled in tasks like natural language understanding (Joshi et al., 2017; Dodge et al., 2021) and generative text production (Hendrycks et al., 2021; Zhong et al., 2017). However, as model size increases, computational and memory demands scale correspondingly, particularly during inference. This necessitates efficient strategies to reduce memory utilization and computational complexity, thereby reducing inference latency and improving throughput — critical requirements for real-time applications that demand faster user experiences.

Quantization is among the most commonly used optimization techniques that simultaneously addresses both computational and memory constraints in LLM inference. By reducing the numerical precision of model parameters, KV cache states, and activation values, this technique enables low-precision forward pass computations while significantly reducing memory footprint. This unified approach to precision reduction offers a systematic method for optimizing inference efficiency when applied to the computational and memory intensive parts of the LLM inference.

The bottlenecks during LLM inference can be split into three major sections: the linear projection operations (QKV projection and FFN), the memory-intensive Key/Value (KV) cache operations, and the attention mechanism's square computational and memory complexity with respect to context length. As both context lengths and model sizes scale up, the KV cache footprint expands substantially, while the attention computations become increasingly resource-intensive, leading to reduced throughput and elevated latencies, which ultimately results in degraded user experiences as well as increased costs. Figure 1a shows the time spent in the attention operation compared to others during inference as prompt sizes increases. The attention mechanism contributes significantly to the overall generation time, contributing up to 80% of the overall latency at large context lengths (>80k).

Previous quantization works, such as Atom (Zhao et al., 2024), QuaRot(Ashkboos et al., 2024), and Qserve (Lin et al., 2024), have primarily applied quantization techniques to linear network components (such as QKV and FFN projections), converting floating-point parameters and activations to low-bit integers to enhance memory efficiency and computational performance. On the other hand, approaches such as KIVI (Zirui Liu et al., 2023), GEAR (Kang et al., 2024), and Qserve (Lin et al., 2024) focus on compressing

[*]Equal contribution [1]Microsoft [2]Georgia Institute of Technology. Correspondence to: Srikant Bharadwaj <srikant.bharadwaj@microsoft.com>.



(a) Prompt:Output Tokens = 8:1 (b) Timeshare in GPU attention kernel (c) End-to-end inference timeshare

Figure 1. Latency profile of Phi3-Medium on Nvidia A100 GPU. (a) KV cache compression techniques impose a dequantization overhead in attention kernel latency. (b) *TurboAttention* significantly improves attention kernel latency compared to FlashAttention(FP16) baseline while other work mainly focuses on reducing KV-cache memory footprint and bandwidth only.(c) *TurboAttention* reduces latency of Matmul+KV-cache load by enabling quantized integer inference, dequantization by applying block progressive quantization, and faster softmax by introducing sparse activated softmax.

the KV cache to 4-bit or even 2-bit formats but rely on time-intensive floating-point decompression before executing the attention mechanism. This incurs decompression overhead during the execution of attention leading to increased overall latencies, as shown in Figure 1b. In general, these approaches largely neglect the latency overhead of the attention operation, which is a key bottleneck in terms of latency overhead during inference.

On the other hand, attention execution acceleration methods, such as Flash Attention-1,2,3 (Dao et al., 2022; Dao, 2023; Shah et al., 2024), along with related optimizations (Dao et al., 2023; Hong et al., 2024), have improved efficiency of the attention operation and have become the commercial standard. However, they operate exclusively on high-precision formats (FP16/32) which leads to high attention latencies at longer context lengths, as shown in Figure 1c. This leaves considerable scope for further optimizations using quantization, which could yield substantial benefits in memory usage and computational speed.

In this paper, we present, *TurboAttention*, a novel unified technique for enabling quantized execution of attention along with a cooperative KV cache compression mechanism which reduces latency, memory footprint, incurs negligible accuracy loss. First, to accelerate the matrix multiplications, minimize decompression time, and ensure compatibility with FlashAttention, we introduce *FlashQ* (section 3). Second, for efficient GPU tensor core-friendly, softmax computation, we develop *SAS* (section 4). Third, to alleviate the memory bottleneck posed by the KV cache, we design a CUDA kernel-friendly headwise mixed precision quantization method(section 3). Together, *FlashQ* and *SAS* form our comprehensive solution, *TurboAttention*. To our knowledge, this is the first work to bridge the state-of-the-art attention acceleration method (FlashAttention) with a quantization algorithm, and the first to apply lossy techniques to accelerate the entire attention mechanism, encompassing both matrix multiplication and softmax operations. A comparison of our approach with concurrent works is provided in Table 1.

Our evaluations show that *TurboAttention* outperforms state-of-the-art FlashAttention-3 on delivering up to 1.8x improvement in latency as well as up to $2.37\times$ maximum throughput. Further, our method delivers near-lossless accuracy across various tasks including mathematical reasoning (GSM8k and AQuA), symbolic reasoning (BigBench-Hard) and models (Llama3, Qwen2 and Phi-3), further validating its effectiveness.

2 Background and Motivation

In this section, we first introduce the key aspects of the attention mechanism and state-of-the-art quantization techniques before discussing the motivation and challenges in enabling a quantized execution of attention mechanism.

2.1 Attention Mechanism

Multi-head attention (MHA) is a core component of the transformer architecture, with each transformer model consisting of L stacked layers. Each layer comprises two submodules: a multi-head attention mechanism (MHA) and a feed-forward network (FFN). Given input token embeddings $\boldsymbol{X} \in \mathbb{R}^{n\times d}$, the MHA computes attention in parallel across H attention heads, defined as:

$$\begin{aligned} \text{MHA}\left(\boldsymbol{X}\right) &= \text{Concat}\left(\boldsymbol{H}^{(1)}, ..., \boldsymbol{H}^{(H)}\right)\boldsymbol{W}_o, \\ \boldsymbol{H}^{(h)} &= \text{Softmax}\left(\frac{\boldsymbol{Q}^{(h)}\boldsymbol{K}^{(h)\top}}{\sqrt{d_H}}\right)\boldsymbol{V}^{(h)} \end{aligned} \tag{1}$$

where $\boldsymbol{Q}^{(h)} = \boldsymbol{X}\boldsymbol{W}_{q_h}, \boldsymbol{K}^{(h)} = \boldsymbol{X}\boldsymbol{W}_{k_h}, \boldsymbol{V}^{(h)} = \boldsymbol{X}\boldsymbol{W}_{v_h}$ are the Query, Key, and Value matrices for head h. The projection matrices $\boldsymbol{W}_{q_h}, \boldsymbol{W}_{k_h}, \boldsymbol{W}_{v_h} \in \mathbb{R}^{d\times d_H}$ map the

Table 1. TurboAttention enables quantized execution of attention operation as well as KV cache compression. Note that techniques such as ATOM(Zhao et al., 2024), QuaRot(Ashkboos et al., 2024) are orthogonal to TurboAttention and can easily be applied in conjunction.

Target	Technique	QKV Projection	KV Cache Compression	Attention Execution	MLP	Memory Overhead	Inference Latency
Linear Operation (QKV, MLP, etc.)	ATOM (Zhao et al., 2024)	Quantized	✓	-	Quantized	↓	↓
	QuaRot (Ashkboos et al., 2024)	Quantized	✓	-	Quantized	↓	↓
	Qserve (Lin et al., 2024)	Quantized	✓	-	Quantized	↓↓	↓↓
Attention Operation	KIVI (Zirui Liu et al., 2023)	-	✓	-	-	↓	↓
	GEAR (Kang et al., 2024)	-	✓	-	-	↓	↓
	FlashAttention (Dao et al., 2022)	-	-	Flash	-	×	↓
	TurboAttention (This Work)	-	✓	Flash + Quantized	-	↓↓	↓↓

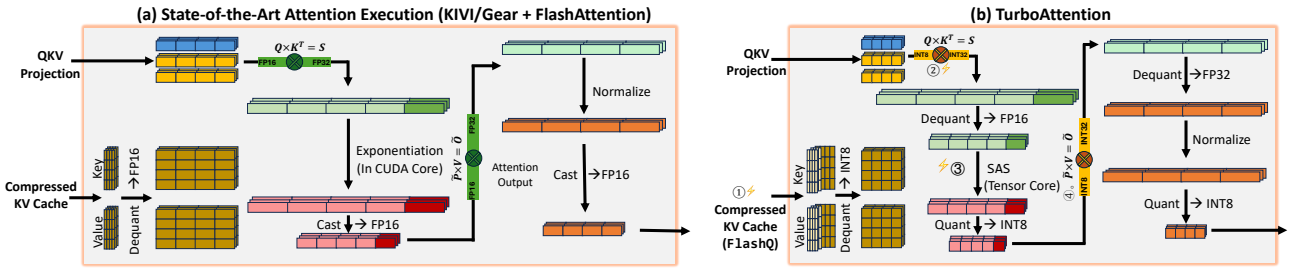


Figure 2. High-Level comparison of *TurboAttention* compared to state-of-the-art KV-cache compression technique combined with FlashAttention. *TurboAttention* accelerates the attention mechanism by adapting (1) FlashQ which enables KV cache compression and accelerated Matmuls ((2) and (3)) and (4) SAS which enables techniques which allow faster execution of attention by utilizing the tensor cores of GPUs efficiently.

input embedding $\boldsymbol{X}$ to the respective attention components, and d_H is typically set to d/H to balance the dimensionality across heads.

Prefill and decoding. During generation, let the model generate n_g tokens. In the initial step, the input tokens $\boldsymbol{X}_0 \in \mathbb{R}^{n \times d}$ are prefilled, and the Keys and Values $(\boldsymbol{K}, \boldsymbol{V})$ for each head and each layer are cached for future use. This prefill stage results in the KV caches: $\boldsymbol{K}_0 = \text{Concat}(\boldsymbol{K}^{(1)}, \ldots, \boldsymbol{K}^{(H)})$ and $\boldsymbol{V}_0 = \text{Concat}(\boldsymbol{V}^{(1)}, \ldots, \boldsymbol{V}^{(H)})$, where $\boldsymbol{K}_0, \boldsymbol{V}_0 \in \mathbb{R}^{n \times d}$.

At each step t $(1 \leq t \leq n_g)$ of autoregressive decoding, the model generates a new token $\boldsymbol{x}_t$, conditioned on the input and previously generated tokens. For the subsequent steps, multi-head attention (MHA) only computes the Query/Key/Value vectors $(\boldsymbol{q}_t, \boldsymbol{k}_t, \boldsymbol{v}_t \in \mathbb{R}^d)$ for the newly generated token $\boldsymbol{x}_t$. These vectors are then appended to the KV cache: $\boldsymbol{K}_t = \boldsymbol{K}_{t-1} | \boldsymbol{k}_t$ and $\boldsymbol{V}_t = \boldsymbol{V}_{t-1} | \boldsymbol{v}_t$. The attention mechanism is then performed between $\boldsymbol{q}_t$ and the updated KV cache, i.e., $\boldsymbol{K}_t$ and $\boldsymbol{V}_t$.

2.2 Attention Acceleration Techniques

The query matrix $Q \in R^{N_q \times d}$ and key and value matrices $K, V \in R^{N_k \times d}$ are inputs to the following equation which is computed independently for each head and batch instance. The output matrix $H \in R^{N_q \times d}$ is obtained in essentially three steps as shown in Equation 2.

$$S = QK^\top, \quad P = \text{softmax}\left(\frac{S}{\sqrt{d}}\right), \quad H = PV. \quad (2)$$

These intermediate activations S and P are large and thus place a significant demand on memory bandwidth. FlashAttention (Dao et al., 2022) addresses this by chunking the Query, Key, and Value matrices along the token dimension and tiling the attention mechanism to accelerate the process as described below.

Softmax approximation. Softmax approximation poses a significant challenge for large language model (LLM) inference due to its computational complexity, particularly involving exponential calculations and normalization. Several approximation methods (Kim et al., 2021; Xia & Zhang, 2024) have been proposed and applied to vision-language models (VLMs) and convolutional neural networks (CNNs). However, these approaches experience substantial performance degradation when directly applied to LLMs due to pure integer approximation and lack of coherent algorithm design.

Flash Attention. FlashAttention integrates the online softmax algorithm (Milakov & Gimelshein, 2018) to fuse the three operations illustrated in Equation 2. It requires only a single pass over an entire row of tokens to compute attention. This method partitions the attention output matrix H into independent output tiles, with each tile's computation being independent of the others. This approach eliminates the need for intermediate global memory reads and writes of S and P tensors. FlashAttention employs a tiling strategy, where small chunks of $query$, key, and $value$ are transferred to the compute cores to first perform "partial attention" for each pair of chunks. This is followed by normalization at

the end with the help of additional values (sum and max).

Additionally, optimization techniques such as Ring Attention (Liu et al., 2023a), Striped Attention (Brandon et al., 2023), and Lean Attention (Sanovar et al., 2024) aim to balance computation across GPU resources while using FlashAttention's core algorithm. These techniques significantly reduce the memory bandwidth overhead during inference, especially for long-context generations. However, the tiled dataflow of FlashAttention's core algorithm, as shown in Figure 2, involving three activation tensors, makes it unsuitable for direct application of state-of-the-art quantization techniques. The quantization techniques do not address the tiled nature of $query$, key, and $value$ tensors, thus, techniques which utilize token-wise or channel-wise quantization scale and min cannot be implemented directly without loss of efficiency. On the other hand, these quantization techniques can be applied ot vanilla attention algorithm, but they have the overhead of storing the large intermediate tensors resulting in high memory bandwidth demand.

Further, to avoid overflow, the exponentiation (in partial-attention calculation) in FlashAttention is performed in FP32 data format (exponentiation in Figure 2(a)), while the matrix multiplications are performed in FP16 using GPU tensor cores. Current generation GPUs are inefficient in handling FP32 and do not make use of tensor cores, resulting in higher latency and inefficiency in the attention operation (FP32 CUDA cores deliver only ∼3% of the performance of FP16 Tensor Cores). The usage of FP16 and FP32 also adds pressure to the scratchpad and register file on the compute units of GPU.

2.3 Progressive Quantization

Progressive quantization (PQ) uses a combination of INT8 and INT4 representations progressively, combining the computational efficiency of symmetric quantization with the ability of asymmetric quantization to represent values accurately at INT4. An overview of PQ follows.

Quantization maps high-precision floating-point values (x) into discrete levels ($Q(x)$), expressed as:

$$Q(x) = \left\lceil \frac{x - z}{s} \right\rfloor, \quad \hat{x} = Q(x) \cdot s + z\,, \tag{3}$$

where s is a scaling factor and z is a zero-point. These depend on the quantization type, symmetric (sym.) or asymmetric (asym.):

$$s = \begin{cases} \dfrac{x_{\max} - x_{\min}}{2^{\text{bit}} - 1}, & \text{sym.} \\ \dfrac{\max|x|}{2^{\text{bit}-1} - 2}, & \text{asym.} \end{cases}, \quad z = \begin{cases} 0, & \text{sym.} \\ x_{\min}, & \text{asym.} \end{cases} \tag{4}$$

For two quantized matrices $\hat{A}$ and $\hat{B}$, the low-bit integer matrix multiplication is:

$$\begin{aligned} O_{ij} \approx \sum_k \hat{A_{ik}}\hat{B_{kj}} &= s_a s_b \sum_k Q(A_{ik})Q(B_{kj}) \\ &+ s_a z_b \sum_k Q(A_{ik}) + s_b z_a \sum_k Q(B_{kj}) + z_a z_b\,. \end{aligned} \tag{5}$$

We see that asymmetric quantization introduces computation (the last three terms) that are not necessary in symmetric computation (where $z = 0$). In general, asymetric quantization leads to lower error rates but results in higher computational overhead.

Progressive quantization proposes two stage of quantization to solve this problem. First, symmetric quantization is used to build INT8 representations for computational efficiency, avoiding the extra overhead of asymmetric quantization (the three non-zero terms above). Then, for better memory efficiency, 8-bit integers are further compressed to INT4 using asymmetric quantization.

The integer inference formula for PQ is:

$$O_{ij} = s_a s_b \sum_k \hat{Q}(A_{ik})\hat{Q}(B_{kj})\,, \tag{6}$$

where $\hat{Q}(A)$ and $\hat{Q}(B)$ denote:

$$\hat{Q}(A) = Q(Q(x)) \cdot s_a^{\text{int}} + z_a^{\text{int}}\,, \tag{7}$$

$$\hat{Q}(B) = Q(Q(x)) \cdot s_b^{\text{int}} + z_b^{\text{int}}\,. \tag{8}$$

Here, s_a and s_b are the (FP16) scales from the first INT8 quantization step. Equations 7 and 8 represent dequantization from *asymmetric* INT4 quantization into INT8. The scales s^{int} and zero points z^{int} are stored in INT8, but the majority of the data $Q(Q(x))$ are stored in INT4.

This approach, proposed in Qserve (Lin et al., 2024) for weight quantization, is more hardware-friendly compared to decompression or other low-bit integer inference methods, such as those in Atom (Zhao et al., 2024), LLM.int8 (Dettmers et al., 2022), or TensorRT-LLM . In our work, we extend the design of PQ to represent KV cache, with the objective of making the dequantization into the attention computations faster while keeping it compatible to attention acceleration mechanisms (such as FlashAttention).

2.4 Challenges in Efficient Attention Mechanism

As we saw in Figure 1a, the attention operation can constitute up to 80% of the overall inference execution time when combined with state-of-the-art quantization techniques for other parts of transformer. This is majorly due to the high-precision execution of attention operation in state-of-the-art techniques, such as FlashAttention. As we saw in subsection 2.2, the usage of FP16 and FP32 not only adds pressure to the scratchpad and register file on the compute units of

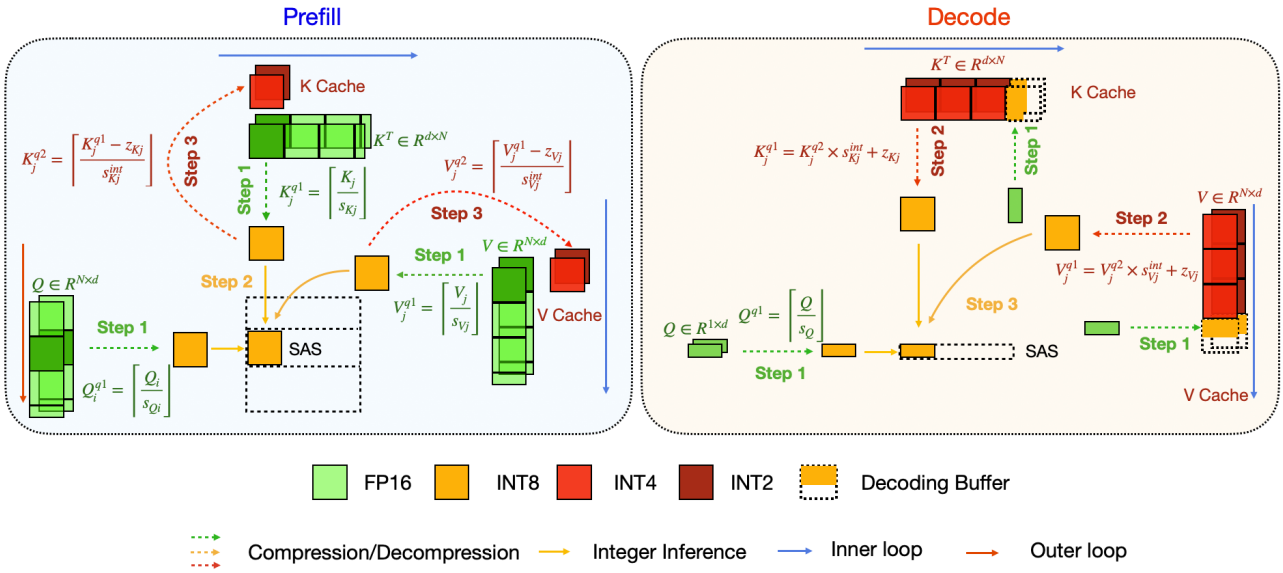


Figure 3. Dataflow of *TurboAttention* in pre-fill and decode. At pre-fill (left) we first compress QKV block-wise into INT8(Step1), and compute the attention matrix (on-line) using *SAS* (see section 4, Step2). Next, we compress the INT8 KV blocks into asymmetric INT4/INT2, channel-wise, in integer arithmetic: these are stored in the cache(Step3). At decoding (right), we first compress generated qkv to INT8(Step1) and decompress the KV cache to INT8 for integer inference(Step2). Again, we use *SAS* to compute attention(Step3).

GPU, but slows down the matrix multiplications and softmax operation inside the attention operation because of the higher precision. For example, the peak performance of FP32 CUDA cores used in FlashAttention's exponentiation operation is only ∼3% of the FP16 tensor performance delivered by Nvidia's A100-SXM. Thus, there is a need to enable quantized execution of the attention operation to leverage the faster low-precision tensor cores in modern GPUs. However, a naive quantization of the attention operation could lead to a loss of the model's generative performance (accuracy).

On the other hand, compression techniques such as KIVI (Zirui Liu et al., 2023), GEAR (Kang et al., 2024), and Qserve (Lin et al., 2024) have proposed techniques to compress the KV cache to 4-bit or even 2-bit formats. However, these techniques aim to reduce the memory footprint of KV cache, but rely on time-intensive decompression to FP16 before executing the attention mechanism using acceleration mechanisms such as FlashAttention.

Additionally, these methods frequently employ dynamic reordering of outlier channels or smooth attention (Lin et al., 2024) techniques to adjust key-query interactions, aiming to mitigate outliers (Dettmers et al., 2022). However, these strategies introduce additional latency, hindering the adoption of integer inference within the attention mechanism. Specifically, the smoothing factor(Xiao et al., 2024b) that transfers outliers from key to query is not applicable in cases where both tensors are quantized activations. Similarly, dynamic reordering requires synchronized reordering of query, key, and value components, reducing its overall effectiveness. Thus, there is a huge need to develop a quantization technique that will enable a quantized execution of the attention operation in a FlashAttention compatible manner and reduce the memory footprint of KV cache while not losing the accuracy of LLMs.

TurboAttention thus presents a novel unified technique for enabling quantized execution of attention along with a cooperative KV cache compression mechanism which reduces latency, memory footprint, incurs negligible accuracy loss. We present the in two parts; *FlashQ* (section 3), which enables quantization of Key, Value, and Query in a FlashAttention compatible manner and *SAS*, which enables efficient GPU tensor core-friendly, softmax computation (section 4).

3 FlashQ: Headwise Mix-precision progressive quantization

FlashQ involves three key components that enable and significantly accelerate the quantized attention mechanism:

1. **Blockwise Progressive Quantization.** FlashAttention-compatible quantization of Key and Value for quantized execution of MatMuls.
2. **Head-wise Mixed Precision.** Compress different heads to different formats (2-bit and 4-bit) for maximum compression.
3. **Enhanced KV cache Buffer.** For dynamic management of the quantized KV cache during the decode phase, eliminating the need for recompression of KV cache.

Figure 3 illustrates the overall flow of *FlashQ* along with

SAS (discussed later in section 4), integrated with the flash attention mechanism. Our method can be deployed to any hardware that has int8 accelerators from RTX4090 to H100.

3.1 Blockwise Progressive Quantization (BPQ)

As we saw in subsection 2.1, multi-head attention generates large intermediate activations, which place a significant demand on memory bandwidth. FlashAttention (Dao et al., 2022) addresses this by chunking the Query, Key, and Value matrices along the token dimension and tiling the attention mechanism to accelerate the process. FlashAttention divides the h-th head's query, key, and value matrices ($\boldsymbol{Q}^{(h)}$, $\boldsymbol{K}^{(h)}$, and $\boldsymbol{V}^{(h)}$) into sub-blocks of size T_c and T_r, denoted as $\boldsymbol{Q}_c^{(h)}$, $\boldsymbol{K}_r^{(h)}$, and $\boldsymbol{V}_r^{(h)}$, for efficient tiled computation.

To enable a FlashAttention-compatible quantized execution of attention we propose Blockwise Progressive Quantization (BPQ). In BPQ, each of the sub-block is first compressed using 8-bit symmetric quantization. Unlike Qserve, which applies per-channel PQ to weights, we apply BPQ at a sub-block granularity via the function:

$$X^{q1} = Quant^{8}_{sym}(X) \tag{9}$$

where $\mathbf{X} \in \boldsymbol{Q}_c^{(h)}, \boldsymbol{K}_r^{(h)}, \boldsymbol{V}_r^{(h)}$ represents a sub-block, and $\text{Quant}^{8}_{\text{sym}}$ denotes symmetric 8-bit quantization. Given the critical memory bottleneck posed by the KV cache, as described in subsection 2.4, we apply progressive quantization after computation to further compress the Key and Value tensors for storage.

Further, building on the techniques from KIVI (Liu et al., 2024) and informed by our own analysis (Figure 4), we further reduce quantization errors by compressing the Key and Value 8-bit tensors in a channel-wise manner using asymmetric quantization:

$$K_g^{q2} = Quant^{4/2}_{asym}(K_g^{q1}), V_g^{q2} = Quant^{4/2}_{asym}(V_g^{q1}) \tag{10}$$

where $\hat{K}_g^{q1}$ and $\hat{V}_g^{q1}$ are groups in each channel of the key and value 8-bit tensors. This minimizes the errors caused by outliers in certain channels while keeping the tiled nature needed to make it compatible with FlashAttention's core algorithm. Since a new query vector is generated at each decoding step, compressing it further beyond the initial 8-bit quantization is unnecessary.

3.2 Head-wise Mixed Precision

To further enhance progressive quantization and achieve substantial memory savings in the KV cache, we explore reducing the bit-width to 2-bit. Although 4-bit KV cache quantization has shown near-lossless performance across various models, as demonstrated by Atom (Zhao et al., 2024), QuaRot (Ashkboos et al., 2024), and Qserve (Lin et al., 2024), uniformly applying 2-bit compression to all attention

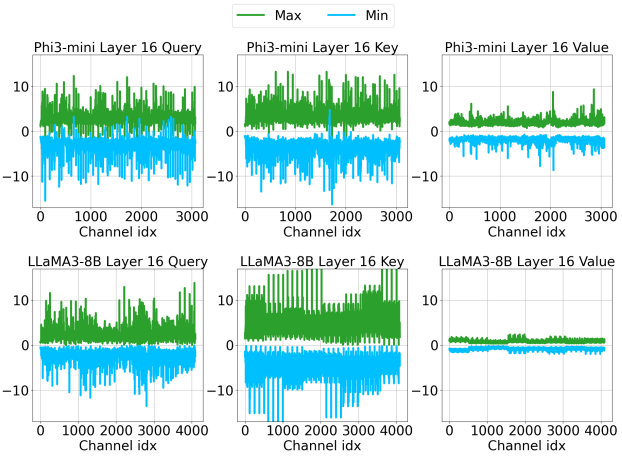


Figure 4. Query, key and value channels min-max distribution of Phi3-mini and LLaMA3-8B Models. We observe that certain heads in query and key have a number of large-magnitude channels. For value, there is no obvious outlier pattern.

heads often leads to significant model performance degradation. Approaches like smooth factors (Lin et al., 2024; Xiao et al., 2024a) and offline reordering (e.g., Qserve and Atom) can improve compression accuracy but introduce additional latency overhead, making them challenging to integrate with dynamically expanding KV caches. While token-wise mixed precision could further enhance accuracy, it incurs dynamic execution overhead in CUDA kernels and is not FlashAttention-compatible, thus severely reducing hardware efficiency.

Inspired by recent work on head pruning in multi-head attention (Ge et al., 2024; Liu et al., 2023b; Rajput et al., 2024), we propose a headwise mixed-precision approach. This approach combines 2-bit and 4-bit compression, and thus achieves a favorable trade-off between accuracy and memory savings. It applies 2-bit and 4-bit BPQ for different heads based on their priority. The priority is calculated by:

$$\text{priority}^{(h)} = \text{gap}^{(h)} \times \text{std}^{(h)} \tag{11}$$

Here, $\text{gap}^{(h)}$ represents the difference between the maximum and minimum values across all channels for head h, which captures the range of values in the attention mechanism. A larger gap indicates that compressing this head may introduce a larger quantization error, thus making it more important to preserve precision. $\text{std}^{(h)}$ is the standard deviation of the channel-wise gaps within each head. This measures the variability of the gaps, where higher variability (i.e., a larger standard deviation) implies that the head's distribution is more uneven. Heads with a higher $\text{std}^{(h)}$ are more sensitive to quantization, and thus require higher bit precision to minimize performance degradation.

Using this metric, we rank all heads in terms of their priority scores. Instead of using a fixed threshold, we compress the lowest-priority n_h heads in each layer to 2-bit, while the remaining heads are quantized to 4-bit. The quantization

strategy is thus defined as:

$$\text{Quantization strategy} = \begin{cases} \text{2-bit}, & \text{if rank(priority}^{(h)}) \leq n_h \\ \text{4-bit}, & \text{if rank(priority}^{(h)}) > n_h \end{cases} \tag{12}$$

Here, n_h is the number of heads to be compressed to 2-bit, determined per layer. By ranking the heads based on their priority and applying lower-bit quantization to the least critical heads (i.e., those with the smallest priority), we achieve a balanced trade-off between memory savings and model performance. Figure 3 illustrates the head-wise quantization dataflow of *FlashQ, where the second head is block-wise progressively quantized to INT2*. As we will see later, this technique presents a robust way to compress the KV-cache and can be employed in a majority of models easily.

3.3 Enhanced KV cache Buffer

FlashQ introduces an efficient decoding buffer to accelerate inference during long-context generation. To support integer inference in the attention mechanism, newly generated tokens are temporarily stored in a buffer $\mathcal{B}$ of size n_b (e.g., $n_b = 64$) and quantized using 8-bit symmetric quantization (Equation 9). This avoids the need for compression at every decoding iteration. Once the buffer reaches its capacity, *FlashQ* applies progressive quantization (Equation 10) to further compress the keys and values from 8 bits to lower bit-widths for improved memory efficiency.

To ensure stability, we employ a universal scale for 8-bit quantization and clamp outliers that exceed this range, preventing the need to re-compress previously stored tokens in the buffer when new tokens with larger values are added. This approach contrasts with previous methods like KIVI (Liu et al., 2024) and GEAR (Kang et al., 2024), which keep buffers in full precision, introducing significant memory overhead and preventing the use of integer inference in the attention mechanism. We summarize the detailed algorithm of TurboAttention in Appendix A.

4 SAS: Sparse Activated Softmax

Softmax is a key bottleneck in the attention mechanism, particularly in flash attention workflows, where tiling operations introduce additional overhead. From our experiments, we observe that softmax computation costs over 30% of the attention execution time both in prefill and decoding stages. The primary performance issue stems from frequent data type conversions between FP16 and FP32 for performing the exponentiation operation. Current GPUs are limited to single-precision floating-point (FP32) operations for exponentiation, which exacerbates this bottleneck. Previous methods, such as (Vasyltsov & Chang, 2021) and (Shim et al., 2017), either introduce significant errors that degrade performance or are unsuitable for modern large language models.

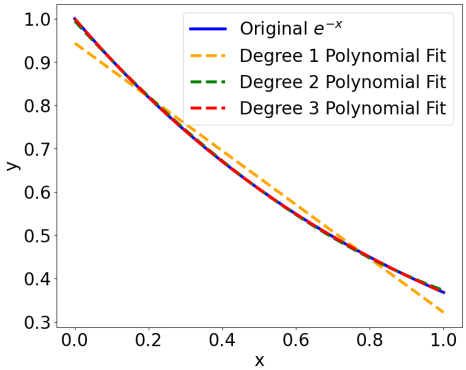


Figure 5. Polynomial-fit for the decimal part of value in exponentiation operation.

To mitigate these issues, we propose a softmax approximation technique based on polynomial approximation(POLY) and lookup tables(LUT), called Sparse Activated Softmax (SAS). This method divides the exponential computation into two parts: the integer and decimal components. For example, an exponent computation can be separated into integer x_{int} and decimal x_{dec} parts which lie in [0,1) (Detailed algorithm in Appendix B). For the integer part, we use a lookup table, which remains compact because large values in the exponential function decay quickly, allowing us to skip larger integers. For the decimal part, we employ a simple polynomial approximation.

$$e^{-x} = e^{-x_{int}} \times e^{-x_{dec}} \approx LUT(-x_{int}) \times POLY(-x_{dec}) \tag{13}$$

As such, the approximation algorithm of SAS goes:

$$SAS(x) = \begin{cases} 0, & x < n_r \\ LUT(x_{int}) \times POLY(x_{dec}), & x \geq n_r \end{cases} \tag{14}$$

To optimize the computation of the exponential function e^{-x}, we limit the approximation range to $[0, 1]$ and reduce the polynomial degree to less than 3 to enhance computational efficiency. Using the least squares method to determine the coefficients, our 3-degree polynomial approximation of e^{-x} within this range is as follows:

$$POLY(x) = -0.1025\,x^3 + 0.4626\,x^2 - 0.9922\,x + 0.9996 \tag{15}$$

This polynomial captures the essential behavior of e^{-x} over $[0, 1]$ with minimal computational overhead, as shown in Figure 5. This hybrid approach reduces computational and data type conversion overhead while preserving accuracy, effectively alleviating the exponentiation bottleneck in flash attention algorithm. The overall exponentiation operation is accelerated, as the polynomial operation can be done solely in GPU Tensor Cores in FP16 dataformat.

Further, from our observations, the large negative values

of the attention scores (resulting from the query-key multiplication) become extremely small after softmax due to the nature of the exponential function. As such, we apply a sparsification strategy, retaining only the larger attention scores within a certain range n_r(e.g., 0 to -5) and setting the rest to zero after the sparse activated softmax computation to keep the look-up table small enough. This further reduces the computational cost, making SAS a practical and efficient solution for large-scale language models.

5 EVALUATIONS

5.1 Experiment Setup

Models and dataset We conducted an extensive evaluation of *TurboAttention* for generative inference using a variety of open-source pre-trained and instruction-tuned models, including LLaMA3-8B-inst (Dubey et al., 2024), Phi3-mini-inst (Abdin et al., 2024), and Qwen2.5-7B (Yang et al., 2024), on multiple generative tasks. These tasks included mathematical reasoning datasets such as GSM8k (Cobbe et al., 2021) and AQuA (Ling et al., 2017), symbolic reasoning tasks from BigBench Hard (BBH) (Suzgun et al., 2022). Given the complexity of these tasks, we use the chain-of-thought prompts created by (Fu et al., 2023) to improve reasoning, which contains 8-shot demonstrations of multi-step reasoning. With the CoT demonstrations, we have the average prefill length of GSM8k, AQuA, and BBH as 900, 1304, 1021 respectively. We then prompt the model to generate 256 tokens and extract answers from them.

5.2 TurboAttention Implementation

To minimize overheads, we implemented and optimized *TurboAttention* with OpenAI Triton(Tillet et al., 2019) kernel support as follows. First, to address the memory bottlenecks caused by frequent quantization and dequantization, we fused the QKV projection with quantization (Equation 9). Additionally, efficient dequantization from progressive quantization was integrated directly into the attention mechanism to reduce decoding latency (Equation 10).

Second, we implement a KV cache buffer (subsection 3.3 for the newly generated Key and Value vectors during decode. These Key/Value vectors are further compressed to INT4 and INT2 (depending on the head) every n_b steps. To avoid recompression when newly generated KV tokens exceed the previous cache value range maximum, we clamp outliers to ensure only newly generated tokens are compressed to INT8. This further improves decoding efficiency by reducing unnecessary recompression.

Third, we leveraged the Triton framework to implement INT8 matrix multiplications for the attention mechanism, thereby reducing both memory and computation bottlenecks. Finally, we implemented an efficient polynomial approximation and lookup table for *SAS*, significantly enhancing the overall performance of the softmax operation. For *TurboAttention*, we fix block size B_c, B_r, and n_b to 64, *SAS* threshold n_r to -6, and set half of the heads' KV cache to 2-bit quantization for head-wise mixed precision. Further optimization can be achieved by implementing CUDA kernels, as they can deliver higher efficiency and substantial speedup.

Framework. For our experiments, we applied *TurboAttention* and the baseline methods to models available on *Huggingface* (Wolf et al., 2019), with our inference framework implemented in *PyTorch* (Paszke et al., 2019). Since our focus is on attention acceleration, all other parts of the model are maintained in FP16 unless otherwise stated.

Precision. In our experiments, we explore ultra-low precision quantization techniques, reporting performance results for 4-bit, 3-bit, and mixed headwise 2/4-bit quantization of the KV cache. Our findings indicate that *TurboAttention* achieves near-lossless performance with 4-bit KV cache compression across various tasks and models. However, consistent with findings in other KV cache compression methods, compressing to 2-bit can led to accuracy degradation. To balance accuracy and memory savings, we applied 2-bit precision to 50% of model heads, achieving a practical trade-off. Consequently, we benchmark *TurboAttention* with mixed precision quantization against other 3-bit baseline methods for a comprehensive comparison in 2.

Additional ablation studies examining the impact of the number of 2-bit heads and block size variations are provided in subsection 5.6, while the results for pure 2-bit KV cache quantization are included in the Appendix.

5.3 Baseline Techniques

We compare *TurboAttention* against the following state-of-the-art techniques:

• *FP16* A dense model where both activations and weights are represented as 16-bit floating-point numbers. This serves as our base dense model without any attention optimizations.

• *FlashAttention* An attention algorithm applying FlashAttention (Shah et al., 2024) with FP16 computation (and FP32 for exponentiation). This method is "exact" and does not alter accuracy but improves system efficiency by optimizing the attention mechanism.

• *KIVI* (Liu et al., 2024) A near-lossless KV cache quantization method that compresses the key cache per-channel and the value cache per-token using fine-grained grouping. It stores residual tokens of length n_b in full precision, and achieving 4-bit KV cache compression at its best accuracy mode.

• *GEAR*(Kang et al., 2024) A concurrent KV cache quanti-

Model		LLaMA3-8B-inst			Qwen2-7B-inst			Phi3-3.8B-inst			All
Method	**Bit** b	**GSM8k** Acc	**AQuA** Acc	**BBH** Acc	**GSM8k** Acc	**AQuA** Acc	**BBH** Acc	**GSM8k** Acc	**AQuA** Acc	**BBH** Acc	**Ave.** Acc
FP16	16	78.24	50.79	58.71	71.87	45.67	50.71	84.53	58.66	57.83	61.89
$\text{KIVI}_{g=64,\, n_b=64}$	4	61.18	46.46	53.20	52.16	45.28	45.33	57.09	54.72	51.24	51.85
$\text{GEAR-L}^{(\text{KCVT})}_{r=4}$	4	64.94	47.09	57.14	54.46	46.06	46.41	79.86	54.36	53.40	55.97
TurboAttention$_{n_b=64}$	4	**78.31**	**48.03**	**58.63**	**66.19**	**46.06**	**46.86**	**84.00**	**55.91**	**58.43**	**60.27**
$\text{KIVI}_{g=64,\, n_b=64}$	3	59.43	44.88	51.40	52.38	41.73	46.47	55.57	47.24	50.98	50.01
$\text{GEAR-L}^{(\text{KIVI})}_{r=4}$	3	62.06	46.12	52.72	53.15	**42.03**	**46.65**	56.18	**49.60**	**51.39**	51.10
TurboAttention$^{(\text{mixed})}_{n_b=64}$	2/4	**77.53**	**47.94**	**54.36**	**66.56**	41.91	45.13	**63.53**	31.49	51.32	**53.31**

Table 2. Results on CoT reasoning tasks, which are hard generative tasks. Here, Bit represents the average compressed bit of KV cache for different methods. The best results are shown in **bold**. Mixed precision results have same KV cache size with 3-bit simulated benchmark.

zation method that achieves 2-bit near-lossless compression by leveraging low-rank approximations to reduce quantization errors. Residual tokens of length n_b are stored in full precision for error compensation. Here we use the efficiency version GEAR-L, that deploy quantiations and low rank approximation together to compress KV cache.

It is important to note that TurboAttention is not just a KV cache compression algorithm but rather a unified approximation algorithm for the attention mechanism. However, we compare it with KV cache compression algorithms to highlight its ability to preserve both performance and efficiency in large language models (LLMs).

5.4 Evaluation Results

Generative performance on reasoning tasks. Table 2 demonstrate that *TurboAttention* achieves comparable or superior performance to recent KV cache compression algorithms across different models and the challenging reasoning tasks, in both 4-bit and lower-bit compression settings. Despite further compressing the query and approximating the softmax operation, *TurboAttention* maintains near lossless accuracy, achieving an average accuracy of **60.27%** across three models and datasets, which is closely aligned with the FP16 baseline average accuracy of **61.89%**. Notably, *TurboAttention* also delivers outstanding 3-bit performance, surpassing baseline methods while providing reduced KV cache size, and enhanced inference efficiency.

5.5 Efficiency Comparison

In this section, we evaluate latency(wall clock time) and maximum throughputs of *TurboAttention* on a single A100 GPU(80GB). We measure the attention mechanism of the model under varying batch sizes with a fixed context length of 1k, as well as under different context lengths (ranging from 4k to 32k) with a fixed batch size of 4. We separately assess the prefill and decoding phases and report the speedup ratio compared to the FlashAttention FP16/32 baseline. Figure 6 shows the latency comparison across various input settings and methods, all using the same attention mechanism configuration.

Our results indicate that, across batch sizes from 1 to 64 and context lengths from 4k to 32k, *TurboAttention* consistently outperforms both FlashAttention FP16 and other KV cache compression techniques combined with Flash Attention baselines. Specifically, *TurboAttention* achieves a speedup of up to **1.8×** for prefill and up to **1.7×** for decoding, highlighting significant improvements in softmax, matrix multiplication, and dequantization efficiency. Additionally, by compressing the KV cache to int-4/2, *TurboAttention* enables long-context generation up to 32k with a batch size of 4, whereas the FP16 baseline encounters out-of-memory (OOM) issues beyond a 4k context length. We also observe that due to the dequantization overhead in other KV cache compression methods, such as KIVI, these methods exhibit worse latency relative to the FP16 baseline.

Figure 7a shows the throughputs comparison across various batch sizes with fixed context length 1k and generation length 125 tokens. It demonstrats that, compared to the Flash-FP16 baseline, *TurboAttention* significantly improves maximum throughputs by up to $2.37\times$. Meanwhile, *TurboAttention* achieves better throughputs compared to KIVI or GEAR due to our efficient progressive dequantization, compressed KV cache, and *SAS* which improves the latency.

5.6 Ablation Study

Ablation study on head wise selection methodology In this study, we aim to validate the effectiveness of our priority-based head selection strategy as detailed in subsection 3.2. We compare our approach against several baseline methods that use simpler, less adaptive metrics to drive head selection. As explored in subsection 3.2 and illustrated in Figure 4, channelwise outliers critically impact quantization

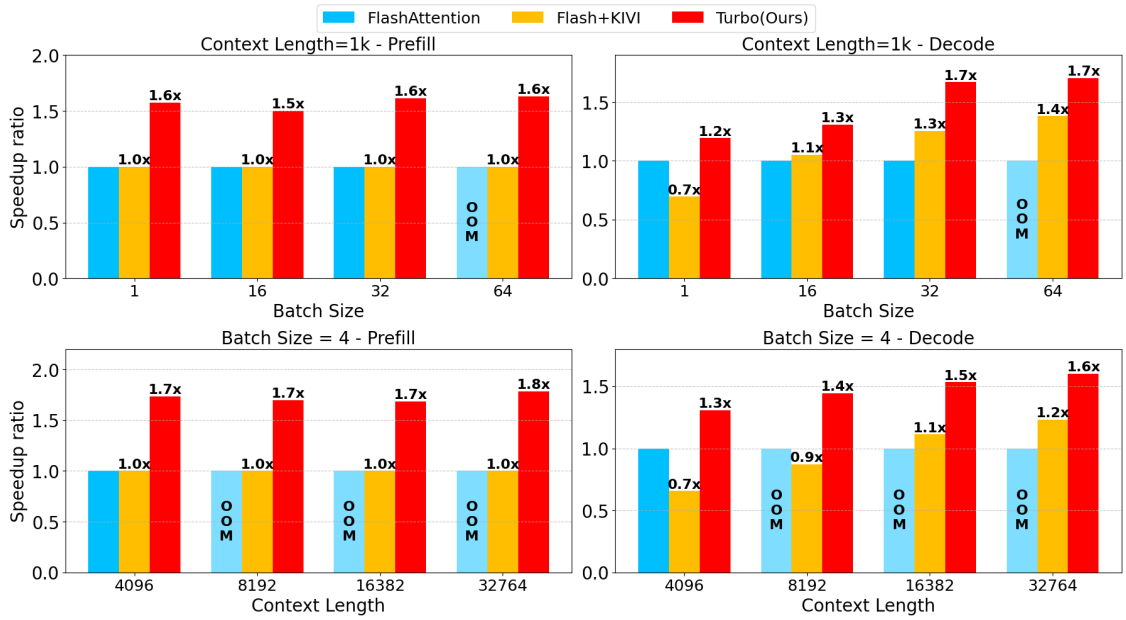
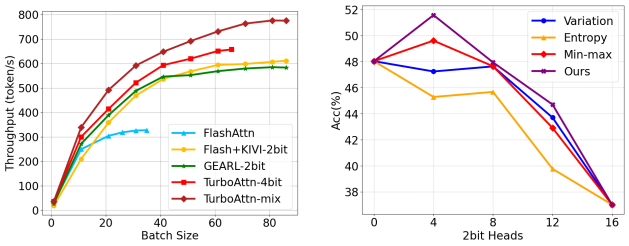


Figure 6. Speedup of the attention mechanism of Phi3-medium on 1xA100-80GB-SXM GPU. *TurboAttention* achieves up to 1.8× latency improvement for prefill under various scenarios. For decoding, *TurboAttention* delivers up to 1.7× improvement over FlashAttention in the FP16 data format. Other methods, such as KIVI, may exhibit higher latency than the FP16 baseline due to the time-consuming dequantization process required before computation. OOM denotes out of memory for Phi3-medium model.



(a) Delivered throughput (b) Priority technique vs. Acc.

Figure 7. (a) Throughput comparison running Phi3-Medium on a 1xA100-SXM-80GB GPU. (b) Comparison of different priority methods with different number of 2-bit heads in LLaMA3-8B-inst on AQua dataset

error. To address this, we examine head selection strategies that incorporate channelwise value ranges as the basis for selection.

Baseline methods include: Entropy: Selecting 2-bit heads based on the entropy of each head. Min-Max: Using the range between minimum and maximum values within heads. Variation: Evaluating variation in channelwise value ranges. The benchmarks are run on the LLaMA-3-8B-inst model with 8-shot COT prompts on the AQua dataset. Given LLaMA-3-8B-inst's 8-head configuration per key/value cache, we perform head selection across a range of 0 to 16 heads. Results, as shown in Figure 7b, indicate that our prioritized strategy significantly surpasses other selection approaches in minimizing quantization error, supporting its efficiency and robustness in managing outliers.

Ablation study on block size We evaluate the sensitivity of *TurboAttention* to varying block sizes (B_c and B_r) for the query, key, and value matrices. These block sizes are closely related to the device's SRAM capacity and have a substantial impact on system efficiency. Consequently, assessing *TurboAttention*'s performance across different block sizes is essential to confirm its robustness and adaptability across configurations. As shown in Table 3 , we tested Phi3-mini with various block sizes on the GSM8K-COT dataset using an 8-shot setup. The results demonstrate that *TurboAttention* maintains robustness across different block sizes, underscoring its ability to integrate seamlessly with Flash Attention across various systems while preserving model accuracy.

Table 3. TurboAttention on Phi3-mini with different block size of query, key, and value.

Block size(B_r,B_c)	Dataset	Acc
(32,32)	GSM8K	78.01
(32,64)	GSM8K	78.01
(64,32)	GSM8K	78.31
(64,64)	GSM8K	78.31
(64,128)	GSM8K	78.16
(128,64)	GSM8K	78.01
(128,128)	GSM8K	77.83

Ablation study of SAS and FlashQ Separately To further demonstrate what are accruacy degradation caused by SAS and FlashQ separately, we have provided more experiments in Table 5. It demonstrated that both FlashQ and SAS can

achieve near lossless performance on complex dataset. More ablation results about SAS are in appendix.

6 Conclusions

This paper introduces *TurboAttention*, a highly efficient compression algorithm for attention mechanisms that integrates seamlessly with attention acceleration methods like FlashAttention, achieving near loss-less performance. *TurboAttention* demonstrates state-of-the-art results in handling complex generative tasks, including mathematical and reasoning challenges, using 4-bit and 2-bit KV cache compression. Additionally, it provides significant performance gains, with up to 1.8x latency speedup and a 2.37x increase in maximum throughput over the FP16 baseline, marking a substantial advancement in efficient attention mechanism execution.

7 Acknowledgement

We are deeply grateful to Qingyang Chen (a Ph.D. alumnus from Georgia Tech) for his guidance on Triton and CUDA kernels. We also thank Akshat Ramachandran(PhD student at Georgia Tech), Shang Yang, and Han Guo(both PhD student at MIT) for their valuable suggestions on refining the paper.

References

Abdin, M., Aneja, J., Awadalla, H., Awadallah, A., Awan, A. A., Bach, N., Bahree, A., Bakhtiari, A., Bao, J., Behl, H., Benhaim, A., Bilenko, M., Bjorck, J., Bubeck, S., Cai, M., Cai, Q., Chaudhary, V., Chen, D., Chen, D., Chen, W., Chen, Y.-C., Chen, Y.-L., Cheng, H., Chopra, P., Dai, X., Dixon, M., Eldan, R., Fragoso, V., Gao, J., Gao, M., Gao, M., Garg, A., Giorno, A. D., Goswami, A., Gunasekar, S., Haider, E., Hao, J., Hewett, R. J., Hu, W., Huynh, J., Iter, D., Jacobs, S. A., Javaheripi, M., Jin, X., Karampatziakis, N., Kauffmann, P., Khademi, M., Kim, D., Kim, Y. J., Kurilenko, L., Lee, J. R., Lee, Y. T., Li, Y., Li, Y., Liang, C., Liden, L., Lin, X., Lin, Z., Liu, C., Liu, L., Liu, M., Liu, W., Liu, X., Luo, C., Madan, P., Mahmoudzadeh, A., Majercak, D., Mazzola, M., Mendes, C. C. T., Mitra, A., Modi, H., Nguyen, A., Norick, B., Patra, B., Perez-Becker, D., Portet, T., Pryzant, R., Qin, H., Radmilac, M., Ren, L., de Rosa, G., Rosset, C., Roy, S., Ruwase, O., Saarikivi, O., Saied, A., Salim, A., Santacroce, M., Shah, S., Shang, N., Sharma, H., Shen, Y., Shukla, S., Song, X., Tanaka, M., Tupini, A., Vaddamanu, P., Wang, C., Wang, G., Wang, L., Wang, S., Wang, X., Wang, Y., Ward, R., Wen, W., Witte, P., Wu, H., Wu, X., Wyatt, M., Xiao, B., Xu, C., Xu, J., Xu, W., Xue, J., Yadav, S., Yang, F., Yang, J., Yang, Y., Yang, Z., Yu, D., Yuan, L., Zhang, C., Zhang, C., Zhang, J., Zhang, L. L., Zhang, Y., Zhang, Y., Zhang, Y., and Zhou, X. Phi-3 technical report: A highly capable language model locally on your phone, 2024. URL https://arxiv.org/abs/2404.14219.

Ashkboos, S., Mohtashami, A., Croci, M. L., Li, B., Jaggi, M., Alistarh, D., Hoefler, T., and Hensman, J. Quarot: Outlier-free 4-bit inference in rotated llms, 2024. URL https://arxiv.org/abs/2404.00456.

Brandon, W., Nrusimha, A., Qian, K., Ankner, Z., Jin, T., Song, Z., and Ragan-Kelley, J. Striped attention: Faster ring attention for causal transformers. *arXiv preprint arXiv:2311.09431*, 2023.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I., and Amodei, D. Language models are few-shot learners, 2020. URL https://arxiv.org/abs/2005.14165.

Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., Hesse, C., and Schulman, J. Training verifiers to solve math word problems, 2021.

Dao, T. Flashattention-2: Faster attention with better parallelism and work partitioning, 2023. URL https://arxiv.org/abs/2307.08691.

Dao, T., Fu, D. Y., Ermon, S., Rudra, A., and Ré, C. Flashattention: Fast and memory-efficient exact attention with io-awareness, 2022. URL https://arxiv.org/abs/2205.14135.

Dao, T., Haziza, D., Massa, F., and Sizov, G. Flashdecoding: Stanford CRFM — crfm.stanford.edu. https://crfm.stanford.edu/2023/10/12/flashdecoding.html, 2023. [Accessed 22-04-2024].

Dettmers, T., Lewis, M., Belkada, Y., and Zettlemoyer, L. Llm.int8(): 8-bit matrix multiplication for transformers at scale, 2022. URL https://arxiv.org/abs/2208.07339.

Dodge, J., Sap, M., Marasović, A., Agnew, W., Ilharco, G., Groeneveld, D., Mitchell, M., and Gardner, M. Documenting large webtext corpora: A case study on the colossal clean crawled corpus, 2021. URL https://arxiv.org/abs/2104.08758.

Dubey, A., Jauhri, A., Pandey, A., Kadian, A., Al-Dahle, A., Letman, A., Mathur, A., Schelten, A., Yang, A., Fan, A.,

Goyal, A., Hartshorn, A., Yang, A., Mitra, A., Sravankumar, A., Korenev, A., Hinsvark, A., Rao, A., Zhang, A., Rodriguez, A., Gregerson, A., Spataru, A., Roziere, B., Biron, B., Tang, B., Chern, B., Caucheteux, C., Nayak, C., Bi, C., Marra, C., McConnell, C., Keller, C., Touret, C., Wu, C., Wong, C., Ferrer, C. C., Nikolaidis, C., Allonsius, D., Song, D., Pintz, D., Livshits, D., Esiobu, D., Choudhary, D., Mahajan, D., Garcia-Olano, D., Perino, D., Hupkes, D., Lakomkin, E., AlBadawy, E., Lobanova, E., Dinan, E., Smith, E. M., Radenovic, F., Zhang, F., Synnaeve, G., Lee, G., Anderson, G. L., Nail, G., Mialon, G., Pang, G., Cucurell, G., Nguyen, H., Korevaar, H., Xu, H., Touvron, H., Zarov, I., Ibarra, I. A., Kloumann, I., Misra, I., Evtimov, I., Copet, J., Lee, J., Geffert, J., Vranes, J., Park, J., Mahadeokar, J., Shah, J., van der Linde, J., Billock, J., Hong, J., Lee, J., Fu, J., Chi, J., Huang, J., Liu, J., Wang, J., Yu, J., Bitton, J., Spisak, J., Park, J., Rocca, J., Johnstun, J., Saxe, J., Jia, J., Alwala, K. V., Upasani, K., Plawiak, K., Li, K., Heafield, K., Stone, K., El-Arini, K., Iyer, K., Malik, K., Chiu, K., Bhalla, K., Rantala-Yeary, L., van der Maaten, L., Chen, L., Tan, L., Jenkins, L., Martin, L., Madaan, L., Malo, L., Blecher, L., Landzaat, L., de Oliveira, L., Muzzi, M., Pasupuleti, M., Singh, M., Paluri, M., Kardas, M., Oldham, M., Rita, M., Pavlova, M., Kambadur, M., Lewis, M., Si, M., Singh, M. K., Hassan, M., Goyal, N., Torabi, N., Bashlykov, N., Bogoychev, N., Chatterji, N., Duchenne, O., Çelebi, O., Alrassy, P., Zhang, P., Li, P., Vasic, P., Weng, P., Bhargava, P., Dubal, P., Krishnan, P., Koura, P. S., Xu, P., He, Q., Dong, Q., Srinivasan, R., Ganapathy, R., Calderer, R., Cabral, R. S., Stojnic, R., Raileanu, R., Girdhar, R., Patel, R., Sauvestre, R., Polidoro, R., Sumbaly, R., Taylor, R., Silva, R., Hou, R., Wang, R., Hosseini, S., Chennabasappa, S., Singh, S., Bell, S., Kim, S. S., Edunov, S., Nie, S., Narang, S., Raparthy, S., Shen, S., Wan, S., Bhosale, S., Zhang, S., Vandenhende, S., Batra, S., Whitman, S., Sootla, S., Collot, S., Gururangan, S., Borodinsky, S., Herman, T., Fowler, T., Sheasha, T., Georgiou, T., Scialom, T., Speckbacher, T., Mihaylov, T., Xiao, T., Karn, U., Goswami, V., Gupta, V., Ramanathan, V., Kerkez, V., Gonguet, V., Do, V., Vogeti, V., Petrovic, V., Chu, W., Xiong, W., Fu, W., Meers, W., Martinet, X., Wang, X., Tan, X. E., Xie, X., Jia, X., Wang, X., Goldschlag, Y., Gaur, Y., Babaei, Y., Wen, Y., Song, Y., Zhang, Y., Li, Y., Mao, Y., Coudert, Z. D., Yan, Z., Chen, Z., Papakipos, Z., Singh, A., Grattafiori, A., Jain, A., Kelsey, A., Shajnfeld, A., Gangidi, A., Victoria, A., Goldstand, A., Menon, A., Sharma, A., Boesenberg, A., Vaughan, A., Baevski, A., Feinstein, A., Kallet, A., Sangani, A., Yunus, A., Lupu, A., Alvarado, A., Caples, A., Gu, A., Ho, A., Poulton, A., Ryan, A., Ramchandani, A., Franco, A., Saraf, A., Chowdhury, A., Gabriel, A., Bharambe, A., Eisenman, A., Yazdan, A., James, B., Maurer, B., Leonhardi, B., Huang, B., Loyd, B., Paola, B. D., Paranjape, B., Liu, B., Wu, B., Ni, B., Hancock, B., Wasti, B., Spence, B., Stojkovic, B., Gamido, B., Montalvo, B., Parker, C., Burton, C., Mejia, C., Wang, C., Kim, C., Zhou, C., Hu, C., Chu, C.-H., Cai, C., Tindal, C., Feichtenhofer, C., Civin, D., Beaty, D., Kreymer, D., Li, D., Wyatt, D., Adkins, D., Xu, D., Testuggine, D., David, D., Parikh, D., Liskovich, D., Foss, D., Wang, D., Le, D., Holland, D., Dowling, E., Jamil, E., Montgomery, E., Presani, E., Hahn, E., Wood, E., Brinkman, E., Arcaute, E., Dunbar, E., Smothers, E., Sun, F., Kreuk, F., Tian, F., Ozgenel, F., Caggioni, F., Guzmán, F., Kanayet, F., Seide, F., Florez, G. M., Schwarz, G., Badeer, G., Swee, G., Halpern, G., Thattai, G., Herman, G., Sizov, G., Guangyi, Zhang, Lakshminarayanan, G., Shojanazeri, H., Zou, H., Wang, H., Zha, H., Habeeb, H., Rudolph, H., Suk, H., Aspegren, H., Goldman, H., Damlaj, I., Molybog, I., Tufanov, I., Veliche, I.-E., Gat, I., Weissman, J., Geboski, J., Kohli, J., Asher, J., Gaya, J.-B., Marcus, J., Tang, J., Chan, J., Zhen, J., Reizenstein, J., Teboul, J., Zhong, J., Jin, J., Yang, J., Cummings, J., Carvill, J., Shepard, J., McPhie, J., Torres, J., Ginsburg, J., Wang, J., Wu, K., U, K. H., Saxena, K., Prasad, K., Khandelwal, K., Zand, K., Matosich, K., Veeraraghavan, K., Michelena, K., Li, K., Huang, K., Chawla, K., Lakhotia, K., Huang, K., Chen, L., Garg, L., A, L., Silva, L., Bell, L., Zhang, L., Guo, L., Yu, L., Moshkovich, L., Wehrstedt, L., Khabsa, M., Avalani, M., Bhatt, M., Tsimpoukelli, M., Mankus, M., Hasson, M., Lennie, M., Reso, M., Groshev, M., Naumov, M., Lathi, M., Keneally, M., Seltzer, M. L., Valko, M., Restrepo, M., Patel, M., Vyatskov, M., Samvelyan, M., Clark, M., Macey, M., Wang, M., Hermoso, M. J., Metanat, M., Rastegari, M., Bansal, M., Santhanam, N., Parks, N., White, N., Bawa, N., Singhal, N., Egebo, N., Usunier, N., Laptev, N. P., Dong, N., Zhang, N., Cheng, N., Chernoguz, O., Hart, O., Salpekar, O., Kalinli, O., Kent, P., Parekh, P., Saab, P., Balaji, P., Rittner, P., Bontrager, P., Roux, P., Dollar, P., Zvyagina, P., Ratanchandani, P., Yuvraj, P., Liang, Q., Alao, R., Rodriguez, R., Ayub, R., Murthy, R., Nayani, R., Mitra, R., Li, R., Hogan, R., Battey, R., Wang, R., Maheswari, R., Howes, R., Rinott, R., Bondu, S. J., Datta, S., Chugh, S., Hunt, S., Dhillon, S., Sidorov, S., Pan, S., Verma, S., Yamamoto, S., Ramaswamy, S., Lindsay, S., Lindsay, S., Feng, S., Lin, S., Zha, S. C., Shankar, S., Zhang, S., Zhang, S., Wang, S., Agarwal, S., Sajuyigbe, S., Chintala, S., Max, S., Chen, S., Kehoe, S., Satterfield, S., Govindaprasad, S., Gupta, S., Cho, S., Virk, S., Subramanian, S., Choudhury, S., Goldman, S., Remez, T., Glaser, T., Best, T., Kohler, T., Robinson, T., Li, T., Zhang, T., Matthews, T., Chou, T., Shaked, T., Vontimitta, V., Ajayi, V., Montanez, V., Mohan, V., Kumar, V. S., Mangla, V., Albiero, V., Ionescu, V., Poenaru, V., Mihailescu, V. T., Ivanov, V., Li, W., Wang, W., Jiang, W., Bouaziz, W., Constable, W., Tang, X., Wang, X., Wu, X., Wang, X., Xia, X., Wu, X., Gao, X., Chen,

Y., Hu, Y., Jia, Y., Qi, Y., Li, Y., Zhang, Y., Zhang, Y., Adi, Y., Nam, Y., Yu, Wang, Hao, Y., Qian, Y., He, Y., Rait, Z., DeVito, Z., Rosnbrick, Z., Wen, Z., Yang, Z., and Zhao, Z. The llama 3 herd of models, 2024. URL https://arxiv.org/abs/2407.21783.

Fu, Y., Ou, L., Chen, M., Wan, Y., Peng, H., and Khot, T. Chain-of-thought hub: A continuous effort to measure large language models' reasoning performance, 2023.

Ge, S., Zhang, Y., Liu, L., Zhang, M., Han, J., and Gao, J. Model tells you what to discard: Adaptive kv cache compression for llms, 2024. URL https://arxiv.org/abs/2310.01801.

Gunasekar, S., Zhang, Y., Aneja, J., Mendes, C. C. T., Giorno, A. D., Gopi, S., Javaheripi, M., Kauffmann, P., de Rosa, G., Saarikivi, O., Salim, A., Shah, S., Behl, H. S., Wang, X., Bubeck, S., Eldan, R., Kalai, A. T., Lee, Y. T., and Li, Y. Textbooks are all you need, 2023. URL https://arxiv.org/abs/2306.11644.

Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., and Steinhardt, J. Measuring mathematical problem solving with the math dataset, 2021. URL https://arxiv.org/abs/2103.03874.

Hong, K., Dai, G., Xu, J., Mao, Q., Li, X., Liu, J., Dong, Y., Wang, Y., et al. Flashdecoding++: Faster large language model inference with asynchronization, flat gemm optimization, and heuristics. *Proceedings of Machine Learning and Systems*, 6:148–161, 2024.

Joshi, M., Choi, E., Weld, D. S., and Zettlemoyer, L. Triviaqa: A large scale distantly supervised challenge dataset for reading comprehension, 2017. URL https://arxiv.org/abs/1705.03551.

Kang, H., Zhang, Q., Kundu, S., Jeong, G., Liu, Z., Krishna, T., and Zhao, T. Gear: An efficient kv cache compression recipe for near-lossless generative inference of llm, 2024.

Kim, S., Gholami, A., Yao, Z., Mahoney, M. W., and Keutzer, K. I-bert: Integer-only bert quantization, 2021. URL https://arxiv.org/abs/2101.01321.

Langley, P. Crafting papers on machine learning. In Langley, P. (ed.), *Proceedings of the 17th International Conference on Machine Learning (ICML 2000)*, pp. 1207–1216, Stanford, CA, 2000. Morgan Kaufmann.

Lin, Y., Tang, H., Yang, S., Zhang, Z., Xiao, G., Gan, C., and Han, S. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving, 2024. URL https://arxiv.org/abs/2405.04532.

Ling, W., Yogatama, D., Dyer, C., and Blunsom, P. Program induction by rationale generation: Learning to solve and explain algebraic word problems. *ACL*, 2017.

Liu, H., Zaharia, M., and Abbeel, P. Ring attention with blockwise transformers for near-infinite context. *arXiv preprint arXiv:2310.01889*, 2023a.

Liu, Z., Wang, J., Dao, T., Zhou, T., Yuan, B., Song, Z., Shrivastava, A., Zhang, C., Tian, Y., Re, C., and Chen, B. Deja vu: Contextual sparsity for efficient llms at inference time, 2023b. URL https://arxiv.org/abs/2310.17157.

Liu, Z., Yuan, J., Jin, H., Zhong, S., Xu, Z., Braverman, V., Chen, B., and Hu, X. Kivi: A tuning-free asymmetric 2bit quantization for kv cache. *arXiv preprint arXiv:2402.02750*, 2024.

Milakov, M. and Gimelshein, N. Online normalizer calculation for softmax. *arXiv preprint arXiv:1805.02867*, 2018.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. Pytorch: An imperative style, high-performance deep learning library. In Wallach, H. M., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E. B., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 8024–8035, 2019.

Rajput, S., Sheng, Y., Owen, S., and Chiley, V. Inference-friendly models with mixattention, 2024. URL https://arxiv.org/abs/2409.15012.

Sanovar, R., Bharadwaj, S., Amant, R. S., Rühle, V., and Rajmohan, S. Lean attention: Hardware-aware scalable attention mechanism for the decode-phase of transformers, 2024. URL https://arxiv.org/abs/2405.10480.

Shah, J., Bikshandi, G., Zhang, Y., Thakkar, V., Ramani, P., and Dao, T. Flashattention-3: Fast and accurate attention with asynchrony and low-precision, 2024. URL https://arxiv.org/abs/2407.08608.

Shim, K., Lee, M., Choi, I., Boo, Y., and Sung, W. Svd-softmax: Fast softmax approximation on large vocabulary neural networks. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc., 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/4e2a6330465c8ffcaa696a5a16639176-Paper.pdf.

Suzgun, M., Scales, N., Schärli, N., Gehrmann, S., Tay, Y., Chung, H. W., Chowdhery, A., Le, Q. V., Chi, E. H., Zhou, D., and Wei, J. Challenging big-bench tasks and whether chain-of-thought can solve them, 2022.

Tillet, P., Kung, H.-T., and Cox, D. D. Triton: an intermediate language and compiler for tiled neural network computations. *Proceedings of the 3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, 2019. URL https://api.semanticscholar.org/CorpusID:184488182.

Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., Rodriguez, A., Joulin, A., Grave, E., and Lample, G. Llama: Open and efficient foundation language models, 2023. URL https://arxiv.org/abs/2302.13971.

Vasyltsov, I. and Chang, W. Efficient softmax approximation for deep neural networks with attention mechanism, 2021. URL https://arxiv.org/abs/2111.10770.

Wolf, T., Debut, L., Sanh, V., Chaumond, J., Delangue, C., Moi, A., Cistac, P., Rault, T., Louf, R., Funtowicz, M., et al. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

Xia, T. and Zhang, S. Q. Hyft: A reconfigurable softmax accelerator with hybrid numeric format for both training and inference, 2024. URL https://arxiv.org/abs/2311.13290.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models, 2024a. URL https://arxiv.org/abs/2211.10438.

Xiao, G., Lin, J., Seznec, M., Wu, H., Demouth, J., and Han, S. Smoothquant: Accurate and efficient post-training quantization for large language models, 2024b. URL https://arxiv.org/abs/2211.10438.

Yang, A., Yang, B., Hui, B., Zheng, B., Yu, B., Zhou, C., Li, C., Li, C., Liu, D., Huang, F., Dong, G., Wei, H., Lin, H., Tang, J., Wang, J., Yang, J., Tu, J., Zhang, J., Ma, J., Yang, J., Xu, J., Zhou, J., Bai, J., He, J., Lin, J., Dang, K., Lu, K., Chen, K., Yang, K., Li, M., Xue, M., Ni, N., Zhang, P., Wang, P., Peng, R., Men, R., Gao, R., Lin, R., Wang, S., Bai, S., Tan, S., Zhu, T., Li, T., Liu, T., Ge, W., Deng, X., Zhou, X., Ren, X., Zhang, X., Wei, X., Ren, X., Liu, X., Fan, Y., Yao, Y., Zhang, Y., Wan, Y., Chu, Y., Liu, Y., Cui, Z., Zhang, Z., Guo, Z., and Fan, Z. Qwen2 technical report, 2024. URL https://arxiv.org/abs/2407.10671.

Zhao, Y., Lin, C.-Y., Zhu, K., Ye, Z., Chen, L., Zheng, S., Ceze, L., Krishnamurthy, A., Chen, T., and Kasikci, B. Atom: Low-bit quantization for efficient and accurate llm serving, 2024. URL https://arxiv.org/abs/2310.19102.

Zhong, V., Xiong, C., and Socher, R. Seq2sql: Generating structured queries from natural language using reinforcement learning. *CoRR*, abs/1709.00103, 2017.

Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Braverman, V., Beidi Chen, and Hu, X. Kivi : Plug-and-play 2bit kv cache quantization with streaming asymmetric quantization. 2023. doi: 10.13140/RG.2.2.28167.37282. URL https://rgdoi.net/10.13140/RG.2.2.28167.37282.

A DETAILED ALGORITHM OF *TurboAttention*

Algorithm 1 *TurboAttention Prefill*

Require: Matrices $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{N \times d}$ in HBM, compressed KV cache $\mathbf{K}^{q2}, \mathbf{V}^{q2}$,integer scale of KV cache $s^{int}_{\mathbf{K}^{q2}}$, floating scale s_K, s_V, $s^{int}_{\mathbf{V}^{q2}}$,zero point of KV cache $z^{int}_{\mathbf{K}^{q2}}$, $z^{int}_{\mathbf{V}^{q2}}$, block sizes B_c, B_r, Softmax approximate algorithm *SAS*.

Divide $\mathbf{Q}$ into $T_r = \left\lceil \frac{N}{B_r} \right\rceil$ blocks $\mathbf{Q}_1, \ldots, \mathbf{Q}_{T_r}$ of size $B_r \times d$ each, and divide $\mathbf{K}, \mathbf{V}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}_1, \ldots, \mathbf{K}_{T_c}$ and $\mathbf{V}_1, \ldots, \mathbf{V}_{T_c}$, of size $B_c \times d$ each.

Divide the output $\mathbf{O} \in \mathbb{R}^{N \times d}$ into T_r blocks $\mathbf{O}_i, \ldots, \mathbf{O}_{T_r}$ of size $B_r \times d$ each, and divide the logsumexp L into T_r blocks $L_i, \ldots, L_{T_r}$ of size B_r each.

for $1 \leq i \leq T_r$ **do**

  Load $\mathbf{Q}_i$ from HBM to on-chip SRAM.

  On chip, initialize $\mathbf{O}_i^{(0)} = (0)_{B_r \times d} \in \mathbb{R}^{B_r \times d}, \ell_i^{(0)} = (0)_{B_r} \in \mathbb{R}^{B_r}, m_i^{(0)} = (-\infty)_{B_r} \in \mathbb{R}^{B_r}$.

  for $1 \leq j \leq T_c$ **do**

    Load $\mathbf{K}_j, \mathbf{V}_j$ from HBM to on-chip SRAM.

    On chip, compute $s_{\mathbf{Q}_i} = \frac{max(abs(\mathbf{Q}_i))}{119}, \mathbf{Q}_i^{q1} = \left\lceil \frac{\mathbf{Q}_i}{s_{\mathbf{Q}_i}} \right\rfloor$,$s_{\mathbf{K}_j} = \frac{max(abs(\mathbf{K}_j))}{119}, \mathbf{K}_j^{q1} = \left\lceil \frac{\mathbf{K}_j}{s_{\mathbf{K}_j}} \right\rfloor$,

    $s_{\mathbf{V}_j} = \frac{max(abs(\mathbf{V}_j))}{119}, \mathbf{V}_j^{q1} = \left\lceil \frac{\mathbf{V}_j}{s_{\mathbf{V}_j}} \right\rfloor$.

    On chip, compute $\mathbf{S}_i^{(j)} = s_{\mathbf{Q}_i} \cdot s_{\mathbf{K}_j} \cdot \mathbf{Q}_i^{q1} \mathbf{K}_j^{q1} \in \mathbb{R}^{B_r \times B_c}$.

    On chip, compute $m_i^{(j)} = \max(m_i^{(j-1)}, \mathrm{rowmax}(\mathbf{S}_i^{(j)})) \in \mathbb{R}^{B_r}$, $\tilde{\mathbf{P}}_i^{(j)} = SAS(\mathbf{S}_i^{(j)} - m_i^{(j)}) \in \mathbb{R}^{B_r \times B_c}$ (pointwise), $\ell_i^{(j)} = SAS(m_i^{j-1} - m_i^{(j)}) \ell_i^{(j-1)} + \mathrm{rowsum}(\tilde{\mathbf{P}}_i^{(j)}) \in \mathbb{R}^{B_r}$.

    On chip, compute $s_{\tilde{\mathbf{P}}_i^{(j)}} = \frac{max(abs(\tilde{\mathbf{P}}_i^{(j)})}{119}, Q(\tilde{\mathbf{P}}_i^{(j)}) = \left\lceil \frac{\tilde{\mathbf{P}}_i^{(j)}}{s_{\tilde{\mathbf{P}}_i^{(j)}}} \right\rfloor$,

    On chip, compute $\mathbf{O}_i^{(j)} = \mathrm{diag}(SAS(m_i^{(j-1)} - m_i^{(j)}))^{-1} \mathbf{O}_i^{(j-1)} + s_{\tilde{\mathbf{P}}_i^{(j)}} \cdot s_{\mathbf{V}_j} \cdot Q(\tilde{\mathbf{P}}_i^{(j)}) \mathbf{V}_j^{q1}$.

    On chip, compute $s^{int}_{\mathbf{K}_j^{q2}} = \left\lceil \frac{max(\mathbf{K}_j^{q2}) - min(\mathbf{K}_j^{q2})}{2^{bit}-1} \right\rfloor$, $z^{int}_{\mathbf{K}_j^{q2}} = \left\lfloor \frac{min(\mathbf{K}_j^{q2})}{s^{int}_{\mathbf{K}_j^{q2}}} \right\rfloor$, $\mathbf{K}_j^{q2} = \left\lceil \frac{\mathbf{K}_j^{q1}}{s^{int}_{\mathbf{K}_j^{q2}}} - z^{int}_{\mathbf{K}_j^{q2}} \right\rfloor$(channelwise)

    On chip, compute $s^{int}_{\mathbf{V}_j^{q2}} = \left\lceil \frac{max(\mathbf{V}_j^{q2}) - min(\mathbf{V}_j^{q2})}{2^{bit}-1} \right\rfloor$, $z^{int}_{\mathbf{V}_j^{q2}} = \left\lfloor \frac{min(\mathbf{V}_j^{q2})}{s^{int}_{\mathbf{V}_j^{q2}}} \right\rfloor$, $\mathbf{V}_j^{q2} = \left\lceil \frac{\mathbf{V}_j^{q1}}{s^{int}_{\mathbf{V}_j^{q2}}} - z^{int}_{\mathbf{V}_j^{q2}} \right\rfloor$(channelwise)

    Write $\mathbf{K}_j^{q2}$ to HBM as the j-th block of $\mathbf{K}^{q2}$.

    Write $\mathbf{V}_j^{q2}$ to HBM as the j-th block of $\mathbf{V}^{q2}$.

    Write $s^{int}_{\mathbf{K}_j^{q2}}$,$s^{int}_{\mathbf{V}_j^{q2}}$ to HBM as the j-th block of $s^{int}_{\mathbf{K}^{q2}}$, $s^{int}_{\mathbf{V}^{q2}}$.

    Write $s_{\mathbf{K}_j}$,$s_{\mathbf{V}_j}$ to HBM as the j-th block of $s_{\mathbf{K}}$, $s_{\mathbf{V}}$.

    Write $z^{int}_{\mathbf{K}_j^{q2}}$,$z^{int}_{\mathbf{V}_j^{q2}}$ to HBM as the j-th block of $z^{int}_{\mathbf{K}^{q2}}$, $z^{int}_{\mathbf{V}^{q2}}$.

  end for

  On chip, compute $\mathbf{O}_i = \mathrm{diag}(\ell_i^{(T_c)})^{-1} \mathbf{O}_i^{(T_c)}$.

  On chip, compute $L_i = m_i^{(T_c)} + \log(\ell_i^{(T_c)})$.

  Write $\mathbf{O}_i$ to HBM as the i-th block of $\mathbf{O}$.

  Write L_i to HBM as the i-th block of L.

end for

Return the output $\mathbf{O}$ and the logsumexp L.

Algorithm 2 *TurboAttention Decode*

Require: Matrices $\mathbf{Q}, \in \mathbb{R}^{1\times d}$ in HBM, compressed KV cache $\mathbf{K}^{q2}, \mathbf{V}^{q2} \in \mathbb{R}^{N\times d}$, integer scale of KV cache $s^{int}_{\mathbf{K}^{q2}}$,$s^{int}_{\mathbf{V}^{q2}}$, floating scale s_K, s_V, zero point of KV cache $z^{int}_{\mathbf{K}^{q2}}$, $z^{int}_{\mathbf{V}^{q2}}$, block size B_c, Softmax approximate algorithm *SAS*.

Divide $\mathbf{K}^{q2}, \mathbf{V}^{q2}$ in to $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks $\mathbf{K}^{q2}_1, \dots, \mathbf{K}^{q2}_{T_c}$ and $\mathbf{V}^{q2}_1, \dots, \mathbf{V}^{q2}_{T_c}$, of size $B_c \times d$ each.

Divide $s^{int}_{\mathbf{K}^{q2}}$,$s^{int}_{\mathbf{V}^{q2}}$,$z^{int}_{\mathbf{V}^{q2}}$, block size $B_c$into $T_c = \left\lceil \frac{N}{B_c} \right\rceil$ blocks accordingly .

Divide the output $\mathbf{O} \in \mathbb{R}^{1\times d}$ into T_c blocks $\mathbf{O}_i, \dots, \mathbf{O}_{T_c}$ of size $1 \times \frac{d}{T_c}$ each, and divide the logsumexp L into T_r blocks $L_i, \dots, L_{T_c}$ of size 1 each.

Load $\mathbf{Q}$ from HBM to on-chip SRAM.

On chip, initialize $\mathbf{O}_j = (0)_{1\times \frac{d}{T_c}} \in \mathbb{R}^{1\times d}, \ell^{(0)}_i = (0)_{B_c} \in \nvDash^{B_c}, m^{(0)}_i = (-\infty)_{B_c} \in \mathbb{R}^{B_c}$.

for $1 \le j \le T_c$ **do**

  Load $\mathbf{K}^{q2}_j, \mathbf{V}^{q2}_j$ from HBM to on-chip SRAM.

  On chip, compute $s_{\mathbf{Q}} = \frac{max(abs(\mathbf{Q}))}{119}, \mathbf{Q}^{q1} = \left\lceil \frac{\mathbf{Q}}{s_{\mathbf{Q}}} \right\rfloor \mathbf{K}^{q1}_j = \mathbf{K}^{q2}_j \cdot s^{int}_{\mathbf{K}^{q2}_j} + z_{\mathbf{K}^{q2}_j}, \mathbf{V}^{q1}_j = \mathbf{V}^{q2}_j \cdot s^{int}_{\mathbf{V}^{q2}_j} + z_{\mathbf{V}^{q2}_j}$

  On chip, compute $\mathbf{S}_j = s_{\mathbf{Q}} \cdot s_{\mathbf{K}_j} \cdot \mathbf{Q}^{q1}\mathbf{K}^{q1}_j \in \mathbb{R}^{1\times B_c}$.

  On chip, compute $m_j = \max(m_{(j-1)}, \mathrm{rowmax}(\mathbf{S}_j)) \in \mathbb{R}^1$, $\tilde{\mathbf{P}}^{(j)} = SAS(\mathbf{S}_j - m_j) \in \mathbb{R}^{1\times B_c}$ (pointwise), $\ell^{(j)} = SAS(m^{j-1} - m^{(j)})\ell^{(j-1)} + \mathrm{rowsum}(\tilde{\mathbf{P}}^{(j)}) \in \mathbb{R}^1$.

  On chip, compute $s_{\tilde{\mathbf{P}}^{(j)}} = \frac{max(abs(\tilde{\mathbf{P}}^{(j)})}{119}, Q(\tilde{\mathbf{P}}^{(j)}) = \left\lceil \frac{\tilde{\mathbf{P}}^{(j)}}{s_{\tilde{\mathbf{P}}^{(j)}}} \right\rfloor$,

  On chip, compute $\mathbf{O}^{(j)} = \mathrm{diag}(SAS(m^{(j-1)} - m^{(j)}))^{-1}\mathbf{O}^{(j-1)} + s_{\tilde{\mathbf{P}}^{(j)}} \cdot s_{\mathbf{V}_j} \cdot Q(\tilde{\mathbf{P}}^{(j)})\mathbf{V}^{q1}_j$.

end for

On chip, compute $\mathbf{O} = \mathrm{diag}(\ell^{(T_c)})^{-1}\mathbf{O}^{(T_c)}$.

On chip, compute $L = m^{(T_c)} + \log(\ell^{(T_c)})$.

Write $\mathbf{O}$ to HBM.

Return the output $\mathbf{O}$ and the logsumexp L.

B Detailed Algorithm of *SAS*

Algorithm 3 Sparsity-based Softmax Approximation

Require: Input matrix $X \in \mathbb{R}^{N\times d}$, integer threshold n_r, look-up table T with $T[n_r + 1]$ set to 0, polynomial approximator function $POLY$.

Step 1: Normalize input values

$x_{max} = \mathrm{rowmax}(X)$ *// Max value per row*

$X = X - x_{max}$

Step 2: Apply sparsity threshold

$X[X < n_r] = n_r + 1$

Step 3: Separate integer and decimal parts

$X = X_{int} + X_{decimal}$

Step 4: Approximate exponential values

$X_{lut} = T[X_{int}]$ *// Lookup for integer part*

$X_{approx} = POLY(X_{decimal})$ *// Polynomial approx. for decimal part*

$X = X_{lut} \times X_{approx}$

Step 5: Normalize with row-wise sum

$X_{sum} = \mathrm{rowsum}(X)$

$X = \frac{X}{X_{sum}}$

Return X

C Detailed illustration of *FlashQ* and *TurboAttention*

In this section, we separately evaluate the accuracy degradation caused by *FlashQ* and *SAS*. Results are shown in Table 4. It demonstrated that both techniques are near loss-less.

Table 4. Separate discussion about accuracy degradation of FlashQ and SAS

Model	Dataset	Method	Acc
LLaMA3-8B-inst	AQuA	FP16	50.79
LLaMA3-8B-inst	AQuA	FlashQ-4bit	49.60
LLaMA3-8B-inst	AQuA	SAS	50.12
LLaMA3-8B-inst	AQuA	FlashQ-4bit + SAS(TurboAttn)	48.03

Table 5. Separate discussion about accuracy degradation of FlashQ and SAS

Model	Dataset	Method	Acc
LLaMA3-8B-inst	GSM8K	FP16	78.24
LLaMA3-8B-inst	GSM8K	LLM.int8()	77.94
LLaMA3-8B-inst	GSM8K	LLM.int8() + TurboAttention	77.48
LLaMA2-2-7B	PQ	FP16	79.05
LLaMA2-2-7B	PQ	Qserve	78.07
LLaMA2-2-7B	PQ	Qserve + TurboAttention	77.64

D DISTRIBUTION DIFFERENCE OF MODELS

Different classes of models exhibit distinct query, key, and value (QKV) distributions, which can significantly influence the effectiveness of compression algorithms. In Figure 4, we observe that the value cache of the Phi-3 model demonstrates a more pronounced outlier distribution compared to LLaMA and other models. Furthermore, among LLaMA-3, Qwen-2, and Phi-3, the Phi-3 value cache exhibits a stronger outlier distribution along the channel dimension than the token dimension.

Previous approaches, such as GEAR and KIVI, often suffer from performance degradation across these models due to their suboptimal value cache quantization strategies, which primarily rely on groupwise or tokenwise quantization. These methods fail to adequately account for the unique distributional characteristics of certain models, such as Phi-3.

As shown in Figure 8 and Figure 9, both LLaMA-3-8B-inst and Phi-3-mini-inst exhibit larger outlier distributions along the channel dimension, with Phi-3 being particularly extreme. These findings suggest that tokenwise compression is likely to perform worse than channelwise compression for models like Phi-3, where channel outliers dominate. Figure 10 shows a more direct result that channelwise group quantization has less quantization error compared with tokenwise group quantization.

E INTEGRATION OF WITH OTHER COMPRESSION ALGORITHMS

We further test TurboAttention with weight and activation quantization algorithms. Combining with these algorithms will exploit further memory and computation savings. Results are shown in Table 5.

F ADDITIONAL RESULT WITH LARGER MODEL AND HARDER DATASET.

Here we show ablation study with the larger model QWQ-32B and the Phi-4 14B model on GSM8K and Math500 dataset in Table 6.

Table 6. Accuracy and pass1 on GSM8K and Math500 dataset.

Model	Dataset	Method	Acc/Pass1
Phi-4	GSM8K	FP16	90.59
Phi-4	GSM8K	GEAR-L(KIVI)	87.29
Phi-4	GSM8K	TurboAttention-4bit	89.37
Phi-4	GSM8K	GEAR-L(KIVI)	78.73
Phi-4	GSM8K	TurboAttention-4bit	80.01
QWQ	MATH500	FP16	90.59
QWQ	MATH500	GEAR-L(KIVI)	84.00
QWQ	MATH500	TurboAttention-4bit	87.80
QWQ	MATH500	GEAR-L(KIVI)	74.40
QWQ	MATH500	TurboAttention-4bit	76.60

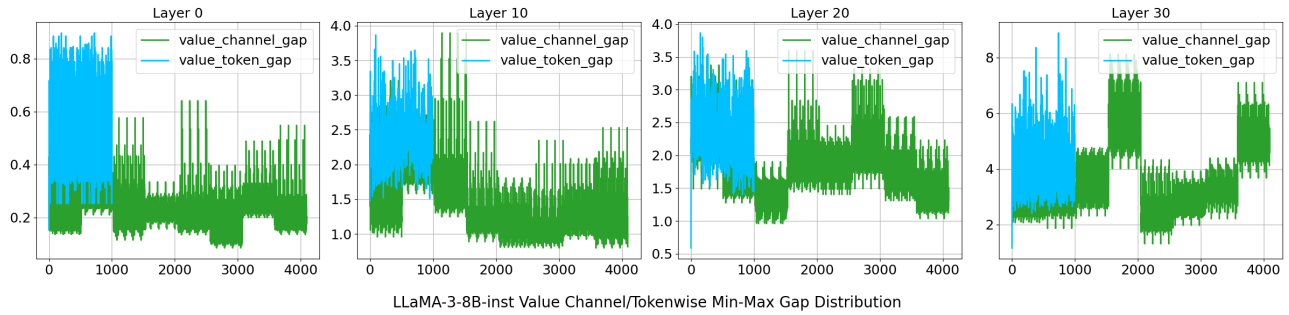


Figure 8. LLaMA-3-8B-inst Value Channel/Tokenwise Min-Max Gap Distribution. This illustrates the distribution of value cache min-max gaps across tokenwise and channelwise dimensions.

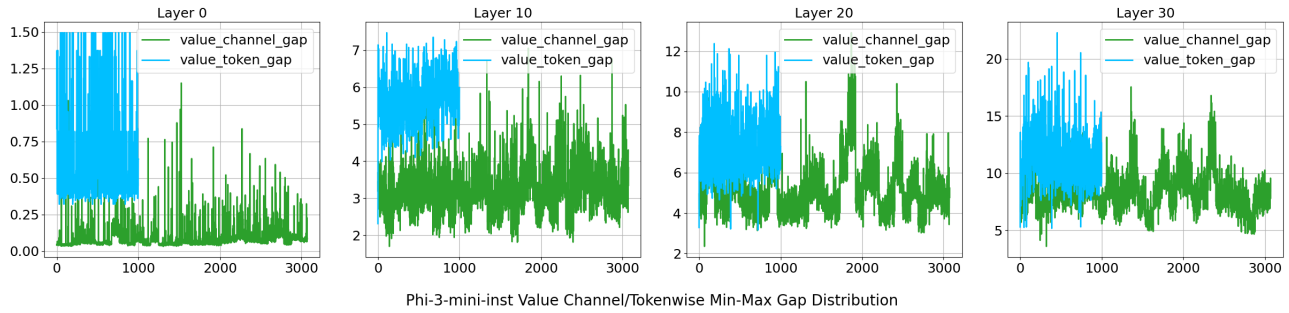


Figure 9. Phi-3-mini-inst Value Channel/Tokenwise Min-Max Gap Distribution. This figure highlights the more significant outlier distribution in the Phi-3 value cache, especially along the channel dimension.

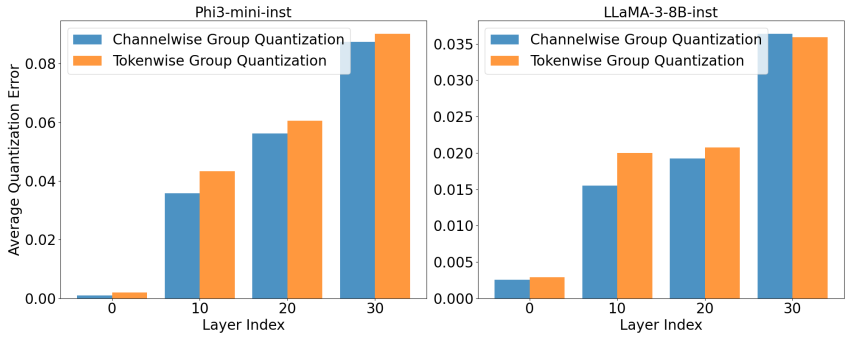


Figure 10. Quantization error of two method