SparseQJL: Efficient KV Cache Compression For Accelerating Large Language Models

Kent Jialei Liu, Hao Zou, Shivan Mukherjee
Computer Science Department
Columbia University
New York, NY, USA
kjl2186@columbia.edu, hz2999@columbia.edu, sm5155@columbia.edu

***Abstract*—The demand for enhanced performance in large language models (LLMs) has surged in recent years, particularly for tasks requiring longer context windows. One straightforward approach to addressing this demand is to scale model size by increasing the number of parameters and layers. However, as LLMs grow, they encounter significant memory bottlenecks that limit the full utilization of state-of-the-art computational resources. While hardware advancements have improved the speed of operations such as matrix multiplications, memory constraints remain a critical barrier, hindering the efficient deployment of these models at scale. A well-explored solution to this challenge is low-bit quantization of the key-value (KV) cache in the attention mechanism, which can drastically reduce memory usage without compromising model performance. In this work, we propose SparseQJL, an approach that combines Johnson-Lindenstrauss-based KV cache quantization with weight pruning. This hybrid strategy significantly improves memory efficiency and reduces the number of model parameters, enabling the scalable deployment of large-scale transformer models with extended context windows while retaining reasonable accuracy.**

***Index Terms*—machine learning, foundation models**

I. Introduction

With the rise of large language models (LLMs), the daily lives of people have been profoundly affected in the way they access and process information. As the capabilities of these models grow and their domains of expertise expand, the general population has become increasingly interested and consumption has grown.

Increased consumption leads to externalities, both positive and negative. On the positive side, many users are harnessing LLM abilities to innovate and create positive impact. For instance, researchers are using LLMs to accelerate scientific discoveries, entrepreneurs are developing new products and services that democratize access to information, and educators are integrating these models to personalize learning experiences. On the less optimistic side, one instance is the negative environmental impact. For instance, training a large model can emit as much carbon as up to 5 cars in their lifetimes. [Strubell et al.(2019)] With this, there is a call to use these models more effectively and with more awareness.

As people push the limits of the capabilities of LLMs, there rises a need to accommodate longer token sequence lengths. To account for this, a few challenges arise, specifically pertaining to the key-value (KV) cache which grows proportionally with increased sequence length and batch size. This leads to a memory bottleneck, which can limit the model's ability to fully take advantage of the GPUs which they typically run on.

Modern GPUs, such as the NVIDIA A100, excel in computational speed, with certain operations (e.g., INT8 or FP16) significantly outperforming traditional FP32 calculations by factors of 2 to 8. Despite their remarkable compute capabilities, GPUs are often constrained by memory bandwidth and capacity, which limits their ability to fully leverage their computational power for large-scale models. To overcome this bottleneck and harness the full potential of these GPUs, we must apply techniques that reduce model size without sacrificing performance. One effective approach is quantization, which involves representing floating-point numbers as integers through a mapping process. This not only compresses the model, reducing its memory footprint, but also converts the values into integer format, enabling faster and more efficient computations, thus allowing GPUs to perform at their peak efficiency.

In our approach, we propose a two-stage method, SparseQJL. In the first stage, we apply quantization. Inspired by the findings of [Zandieh et al.(2024)], we take advantage of the properties of the Johnson-Lindenstrauss (JL) lemma, which allows for minimal memory overhead due to the lack of needing to store quantization parameters. In the second stage, leveraging the techniques proposed by SparseGPT [Frantar and Alistarh(2023)], we prune unimportant weights in the model, reducing the size of projection and feedforward layers while maintaining output accuracy.

In Section II, we highlight some notable related work on the problem of KV cache compression followed by a brief background in Section III. Section IV highlights the details of our proposed approach, breaking down each component. We present our results in Section V followed by a discussion in Section VI. We end with closing remarks and directions of potential future work in Section Section VII.

II. Related Work

Extensive work has been done in recent years to address KV cache compression. In this section, we briefly outline some notable existing methods.

A. Eviction

There has been work done on evicting tokens that are not deemed important. Eviction methods primarily focus on discard tokens that are less important. They vary in the policies on how to select tokens to discard. Naive approaches use simple sliding-window approaches, but this falls short when the sequence length exceeds cache size. [Beltagy et al.(2020)] [Xiao et al.(2024b)] Zhang et al. proposes H2 Eviction Algorithm which uses accumulative normalized attention scores to decide which tokens to keep while keeping recent tokens as they show strong correlation with current tokens. [Zhang et al.(2024)] Instead of solely focusing on memory usage, SparQ Attention takes into consideration the amount of data being transferred. [Ribar et al.(2024)] However, there are some drawbacks as this can lead to hallucinations and degraded performance. For instance, important context information including safety measures can be disregarded, leading to malicious responses. [Yang et al.(2024)]

B. Quantization

The motivation of quantization is to represent full precision values using lower precision and thus less bits. The idea is to map tensor values to values at reduced precision resulting in less memory usage and faster computation. When it comes to KV Cache quantization, there are two primary categories. Full quantization quantizes both weights and the KV cache while KV Cache-only quantization focuses on quantizing KV Cache activations leaving the model weights as they are. [Shi et al.(2024)] WKVQuant proposes an approach which focuses on quantizing the weights and KV cache using Post-Training Quantization (PTQ) techniques, not the activations. [Yue et al.(2024)] KIVI observes that the key and value cache should be quantized along different dimension. The key cache should be quantized per channel and the value cache should be quantized per token. [Liu et al.(2024)] In KVQuant, the keys are also quantized per-channel before Rotary Positional Embeddings (RoPE) is applied. KVQuant proposes nuqX, which quantizes due to sensitivity of different layers. [Hooper et al.(2024)] [Yang et al.(2024)] applies Aware Mixed-Precision, observing that the important KV pairs must be kept at a relatively higher precision to safeguard the generation quality. MIT's QServe developed the SmoothAttention kernel to effectively mitigate the accuracy degradation incurred by 4-bit KV quantization outliers. [Lin et al.(2024)]

III. Background

At inference time, transformer-based models leverage the KV cache to enhance computational efficiency and reduce latency. The mechanism involves storing the key and value representations computed during previous time steps, which can then be reused for subsequent token predictions without recomputing them.

A. Attention Mechanism

[Vaswani et al.(2023)] introduced the widely-adopted Scaled Dot-Product Attention, a key component in the decoder layers of many transformer models .

Given an input sequence $X = \{x_1, x_2, \ldots, x_t\}$, the self-attention mechanism computes attention scores by forming keys (K), queries (Q), and values (V) through linear transformations:

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V, \tag{1}$$

where W_Q, W_K, and W_V are learned projection matrices.

The attention output for a given token is calculated as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V, \tag{2}$$

where d_k is the dimensionality of the keys. At each inference step t, only the new query (q_t) and its interaction with the cached keys and values are computed. Specifically, the attention mechanism reuses the cached K and V matrices from previous steps, appending the new key (k_t) and value (v_t) to update the cache:

$$K' = [K; k_t], \quad V' = [V; v_t]. \tag{3}$$

Here, $[;]$ denotes concatenation along the sequence dimension. This caching reduces the computational complexity from $O(T^2)$ to $O(T)$ per time step, where T is the sequence length.

B. Integer Quantization

Integer quantization maps high-precision numbers to discrete values. Let $X_{\text{min}} = \min(X)$, $X_{\text{max}} = \max(X)$, and $q_{\text{max}} - q_{\text{min}} = 2^n - 2$ We can formulate this as follows:

$$Q_X = \left\lfloor \frac{X}{s} + z \right\rceil \tag{4}$$

where X is a floating point tensor and Q_X is the quantized result of X in n bits. s is the scaling factor and z is the zero point.

$$s = \frac{X_{\text{max}} - X_{\text{min}}}{q_{\text{max}} - q_{\text{min}}}, \quad z = \left\lfloor q_{\text{min}} - \frac{X_{\text{min}}}{s} \right\rceil \tag{5}$$

To dequantize Q_X and get $\hat{X}$, we can do:

$$\hat{X} = Q(X) = (Q_X - z) \cdot s \tag{6}$$

This is known as asymmetric quantization, where the quantization process involves a scaling and then adding an offset to map them to the target range, as opposed to symmetric quantization, which scales values around zero without an offset, typically resulting in simpler computations but less flexibility in representing distributions with large asymmetry. For symmetric quantization, we let $X_{\text{min}} = X_{\text{max}} = \max|X|$, $z = 0$, and $q_{\text{max}} - q_{\text{min}} = 2^n - 2$

C. KV Cache Quantization

Our main objective is to reduce the memory consumption of the KV cache. KV caching is particularly beneficial in autoregressive decoding, where tokens are processed sequentially. By avoiding redundant computation, it enables efficient scaling to long sequences and supports real-time applications like language modeling and machine translation.

At inference time, the operation of auto-regressive LLMs involves two main steps: prompt encoding and token generation. During prompt encoding, the LLM processes the input sequence to establish the contextual information required for generating subsequent tokens. This contextual information is represented as KV pairs, which are stored in the KV cache. As each new token is generated, its corresponding KV vectors are computed and appended to the KV cache. Consequently, the size of the KV cache grows linearly with the length of the token sequence. Therefore, as the context length grows, the KV cache becomes a performance bottleneck.

Here, we formulate the problem of quantizing the KV cache, with a focus on minimizing its memory footprint while maintaining accuracy. Let $\mathcal{M}$ denote the set of all quantization methods. For a quantization method m, we denote the quantized KV cache as the output of the following function $f(K, V, m) = \tilde{K}, \tilde{V}$. With this, we need a method m which minimizes the memory footprint while maintaining a reasonable accuracy.

$$m^* = \text{argmin}_{m \in \mathcal{M}} \quad \text{Memory(m)} \tag{7}$$

subject to:

$$\left\| \text{Softmax}\left(\frac{1}{\sqrt{d_k}} Q K^\top\right) - \text{Softmax}\left(\frac{1}{\sqrt{d_k}} Q \tilde{K}^\top\right) \right\|_2^2 \leq \alpha,$$

$$\left\| \text{Softmax}\left(\frac{1}{\sqrt{d_k}} Q K^\top\right) V - \text{Softmax}\left(\frac{1}{\sqrt{d_k}} Q \tilde{K}^\top\right) \tilde{V} \right\|_2^2 \leq \beta$$

where $\mathcal{M}$ is the set of quantization methods, Memory(m) is the memory cost of method m, and α, β are hyperparameters controlling the quantization loss for the softmax operation and the multiplication of the result with V respectively.

IV. SparseQJL

In order to address 7, we propose a two-stage method, **SparseQJL**, to address the memory bottlenecks arising from KV cache growth and model size during long-sequence inference.

A. Stage 1: Quantized Johnson-Lindenstrauss (QJL) Transform for KV Cache Compression

The **Johnson-Lindenstrauss Lemma** states that any set of n points in a high-dimensional space can be embedded in a $m = O(\log n/\epsilon^2)$ dimensions with relatively high distance preservation, where ϵ is some small value. [Dasgupta and Gupta(2003)] This result has proved useful in many computer science applications including unsupervised learning and database dimensionality reduction. Inspired by [Zandieh et al.(2024)], we reduce the dimensionality of the key vectors in the KV cache apply a Johnson-Lindenstrauss (JL) transform, followed by sign quantization in the first stage. Zandieh et al. that these transformations can give rise to unbiased estimator for the inner product of key and query vectors.

a) Dimensionality Reduction: The JL transform projects the original key vectors $k \in \mathbb{R}^d$ and query vectors $q \in \mathbb{R}^d$ into a lower-dimensional space $m \ll d$ while approximately preserving pairwise distances:

$$\tilde{k} = Sk, \quad S \in \mathbb{R}^{m \times d}, \quad S_{ij} \sim \mathcal{N}(0, 1), \tag{8}$$

where S is a Gaussian random projection matrix as seen in the proof by [Dasgupta and Gupta(2003)].

b) Sign Quantization: After the initial random projection, single-bit sign quantization is applied to the reduced keys. In this case, the sign function maps as $\mathbb{R}^d \to \{-1, +1\}^m$.

$$\hat{k} = \text{sign}(\tilde{k}), \quad \nu = \|k\|_2. \tag{9}$$

where $\hat{k}$ is the quantized key, and ν represents the norm of the original key.

c) Approximate Inner Product: The proposed unbiased inner product estimator is defined as follows:

$$\widehat{\text{Prod}}(q, k) = \frac{\sqrt{\pi/2}}{m} \cdot \nu \cdot \langle Sq, \hat{k} \rangle, \tag{10}$$

where q_i is the query vector and $\langle \cdot, \cdot \rangle$ denotes the inner product.

By using an unbiased estimator between the queries and keys, we eradicate the need to store quantization constants such as scaling factor and zero point as with existing methods. The storing of these parameters, which is typically done in 1 or 2 bits per parameter, significantly reduces memory overhead in addition to the sign quantization.

B. Stage 2: Sparsification of the model using SparseGPT

SparseGPT [Frantar and Alistarh(2023)] is a structured pruning method specifically designed for large-scale transformer models. It takes advantage of overparameterization to remove redundant weights in a model. In Stage 2 of SparseQJL, we use this method to sparsify the weight matrices in the projection and feed-forward layers.

a) Hessian Approximation: The pruning process begins with an approximation of the inverse Hessian of the weights of a layer. This is also known as the curvature of the loss function and is used to capture the sensitivity of the model's loss to changes in each weight. Given the input activations $\mathbf{X}$, the inverse Hessian is approximated as:

$$\mathbf{H}^{-1} = (\mathbf{X}\mathbf{X}^\top + \lambda \mathbf{I})^{-1}, \tag{11}$$

where $\mathbf{X}$ represents the input activations, λ is a regularization parameter to stabilize the computation, and $\mathbf{I}$ is the identity matrix.

b) Mask Selection and Pruning: Sparsification is then performed iteratively across blocks of weights $\mathbf{W}_{:,j:(j+B)}$ processed at each step. All weights are ranked based on curvature impact and based on a target sparsity, a subset of weights with highest curvature impact are selected. The mask retains these high-impact weights, pruning all low-impact weights in the process. It is applied column by column across the weight matrix to introduce sparsity.

$$\text{argmax}_c \left(\frac{w_c^2}{[\mathbf{H}^{-1}]_{cc}} \right), \tag{12}$$

where $[\mathbf{H}^{-1}]_{cc}$ are the diagonal elements of the inverse Hessian corresponding to each weight.

c) Error Correction and Weight Updates: After the mask is generated, the remaining weights are updated using the computed error and the inverse Hessian. This error correction step ensures that pruning-induced error is minimized and layer-by-layer accuracy is maintained.

$$\mathbf{E}_{:,j-i} \leftarrow \mathbf{W}_{:,j:(j+B)} / [\mathbf{H}^{-1}]_{jj}, \tag{13}$$

$$\mathbf{W}_{:,j:(j+B)} \leftarrow \mathbf{W}_{:,j:(j+B)} - \mathbf{E}_{:,j-i} \cdot \mathbf{H}^{-1}_{j,j:(j+B)}, \tag{14}$$

where $\mathbf{E}$ is the block quantization error matrix and B is the block size. This step is crucial for mitigating the loss in model performance due to pruning.

d) Mask Application: After the updates are performed, the pruning mask $\mathbf{M}$ is applied once more to finalize the sparsified weight matrix, ensuring that pruned weights remain zero:

$$\mathbf{W} \leftarrow \mathbf{W} \cdot \mathbf{M}. \tag{15}$$

By combining Hessian-aware pruning, blockwise updates, and error correction, this stage achieves sparsity levels of up to 50% or higher in one shot with no retraining and minimal loss of accuracy. Leveraging the overparameterization of these models, Stage 2 achieves a comprehensive reduction in memory overhead and computational requirements while minimally affecting accuracy.

C. Advantages of SparseQJL

- **Memory Efficiency:** Memory consumption of the KV is significantly reduced through quantization and pruning.
- **Computational Efficiency:** Quantization and reduced model size enables faster inference on memory-constrained hardware.
- **Accuracy Preservation:** Norm preservation and Hessian-aware pruning ensure minimal loss in accuracy.

V. Experiments

A. Setup

Our experiments are conducted on the WikiText-2, C4, and Penn Treebank (PTB) datasets to evaluate the effectiveness of KV-Quantization with sparsity for large language models. Perplexity is used as the evaluation metric, consistent with prior works. We utilize the HuggingFace Transformers library [Wolf et al.(2020)] for model and dataset management. Experiments are performed on an NVIDIA A100 GPU with 40GB of memory, leveraging SparseGPT [Frantar and Alistarh(2023)] and GPTQ [Frantar et al.(2023)] implementations for efficient CLI-based experimentation. Transformer layers are sparsified sequentially, following the methodology outlined by [Yao et al.(2022)], [Frantar et al.(2023)], [Frantar and Alistarh(2023)], to minimize memory overhead. All sparsification experiments are conducted in a one-shot manner without fine-tuning, in line with recent advancements in post-training quantization [Yao et al.(2022)], [Frantar et al.(2023)], [Dettmers et al.(2022)].

B. Models, Datasets and Evaluation

We primarily focus on the OPT model family [Zhang et al.(2022)] and the LLaMA family [Touvron et al.(2023)], examining their scaling and quantization behavior. While our emphasis is on the largest variants within these families, we also include results on smaller models to provide a comprehensive view of their performance under different compression strategies, potentially useful for deployment of such models on resource-constrained devices.

For evaluation, we use perplexity as the primary metric. Perplexity is a robust and widely accepted measure for assessing the accuracy of compression methods, particularly in language modeling tasks [Yao et al.(2022)], [Frantar et al.(2023)], [Dettmers et al.(2022)]. Experiments are conducted on the test sets of raw WikiText-2 [Merity et al.(2016)] and Penn Treebank (PTB) [Marcus et al.(1994)], along with a subset of the C4 validation data—benchmarks that are extensively used in large language model compression literature [Yao et al.(2022)], [Park et al.(2024)], [Frantar et al.(2023)], [Xiao et al.(2024a)].

Perplexity calculations follow the exact procedure outlined by HuggingFace, using a full-stride evaluation. Both dense and sparse results are obtained using the same implementation, provided as supplementary material, to ensure a fair and reproducible comparison across all experiments.

C. Baselines

We compare our method against several established baselines to evaluate its effectiveness. First, we include **Baseline Models** without any modifications, evaluated in fp16 precision using the HuggingFace Transformers library as a standard reference point. Second, we consider **4-Bit Weight Quantization**, leveraging the GPTQ method to compress model weights into 4-bit precision. Third, we evaluate **SparseGPT** with 50% Uniform Sparsity, achieved using SparseGPT, which is one of the most efficient and accurate post-training pruning techniques currently available. Finally, we include a combination baseline, **Pruning + Weight Quantization (WQ)**, where models are pruned to 50% uniform sparsity using SparseGPT and further quantized to 4-bit weights with the GPTQ method, before evaluating against our hybrid method. These baselines provide a comprehensive framework for comparison, spanning both pruning and quantization approaches.

D. Baseline Results

The baseline results I across WikiText-2, PTB, and C4 datasets demonstrate consistent trends in perplexity and memory trade-offs for different compression techniques.

- **4-Bit Weight Quantization** shows minimal degradation in perplexity while significantly reducing memory usage (e.g., OPT-1.3B maintains baseline perplexity with a 4.40GB memory footprint on C4).
- **SparseGPT** introduces higher perplexity scores compared to the baseline, particularly in smaller models like OPT-125M, but remains efficient in terms of sparsity and memory savings.

TABLE I
EVALUATION RESULTS FOR BASELINE MODELS AND VARIOUS PRUNING AND QUANTIZATION METHODS.

Model	Wikitext-2	C4	PTB
Baseline Models			
OPT-125m	27.65	26.56	38.99
OPT-350m	22.00	22.59	31.08
OPT-1.3b	14.62	16.07	20.29
LLaMA-2-7b	8.71	23.52	133.24
LLaMA-2-13b	7.68	16.90	180.15
4-Bit Quantization (GPTQ Method)			
OPT-125m	27.65	26.56	38.99
OPT-350m	22.00	22.59	31.08
OPT-1.3b	14.62	16.07	20.29
LLaMA-2-7b	5.89	-	35.34
LLaMA-2-13b	5.16	-	52.22
SparseGPT (50% Uniform Sparsity)			
OPT-125m	36.93	33.50	55.51
OPT-350m	31.88	29.25	43.71
OPT-1.3b	17.48	19.23	25.58
LLaMA-2-7b	7.02	9.24	186.68
LLaMA-2-13b	6.02	8.22	76.81
SparseGPT + INT4 Weight Quantization			
OPT-125m	41.33	35.97	63.16
OPT-350m	34.84	31.45	48.18
OPT-1.3b	23.49	24.02	30.05
LLaMA-2-7b	7.64	9.74	61.56
LLaMA-2-13b	6.28	8.50	83.94

- **Pruning + Weight Quantization (WQ)** achieves a balance between perplexity and compression, offering moderate perplexity increases while maintaining compact memory usage, particularly for larger models like LLaMA-2-7B and 13B.

The results I show that 4-Bit Weight Quantization maintains baseline perplexity with significant memory savings, while SparseGPT introduces higher perplexity but improves efficiency. Pruning + WQ achieves a balance, offering moderate (perplexity increases) with compact memory usage.

In summary, the results highlight that LLaMA models tend to maintain lower perplexity across all methods compared to the OPT family, particularly on PTB and C4 datasets. SparseGPT shows promise for efficiency but requires tuning to minimize perplexity increases.

E. SparseQJL Results

The results II show that **SparseQJL**, our proposed method combining SparseGPT and QJL, achieves competitive perplexity across datasets (WikiText2, PTB, and C4). Compared to SparseGPT, **SparseQJL** maintains lower perplexity scores while incurring minimal performance degradation compared to QJL alone. This highlights **SparseQJL**'s advantage of achieving balanced trade-offs between performance and computational overhead, making it an efficient solution for model compression.

VI. DISCUSSION

The results presented in this work underscore the effectiveness of various compression techniques for large language models, while also highlighting the nuanced trade-offs between perplexity and memory efficiency.

TABLE II
EVALUATION RESULTS FOR SPARSEQJL METHOD. BASELINE MODELS REFER TO FULL-PRECISION WITH NO MODIFICATION.

Model	Wikitext-2	C4	PTB
Baseline Models			
LLaMA-2-7b	8.71	23.52	133.24
LLaMA-2-13b	7.68	16.90	180.15
SparseGPT (50% Uniform Sparsity)			
LLaMA-2-7b	7.02	9.24	186.68
LLaMA-2-13b	6.02	8.22	76.81
QJL			
LLaMA-2-7b	**5.12**	**7.10**	**40.58**
LLaMA-2-13b	**4.58**	**6.57**	**52.16**
SparseQJL			
LLaMA-2-7b	6.09	9.43	491.04
LLaMA-2-13b	5.33	8.33	85.23

a) SparseQJL Analysis: Our proposed SparseQJL method combines the strengths of SparseGPT and QJL, achieving a balance between perplexity and computational efficiency. SparseQJL maintains lower perplexity scores compared to SparseGPT, with minimal performance degradation relative to QJL alone. This demonstrates its ability to effectively compress models while preserving accuracy, making it a compelling choice for scenarios requiring both performance and efficiency. Furthermore, SparseQJL's ability to handle Key matrix compression through QJL contributes significantly to reducing KV cache memory overhead, an area that remains a bottleneck in long-sequence processing.

b) Trade-Offs and Limitations: While SparseQJL offers balanced trade-offs, its performance still depends on careful tuning of sparsity and quantization parameters to adapt to different model architectures and datasets. Additionally, SparseGPT's higher perplexity on smaller models highlights the need for dataset-specific adjustments to mitigate performance losses.

c) Future Directions: The results suggest several promising avenues for future research. One direction is exploring dynamic or adaptive sparsity levels to better align with model size and dataset characteristics. Additionally, extending SparseQJL to incorporate Value matrix compression could further reduce memory overhead without sacrificing performance. To further push the boundary between compression and accuracy trade-off, exploration of sparse JL projections is also encouraged. Investigating the interaction of compression techniques with fine-tuning on downstream tasks may also reveal more insights into their practical applicability.

Overall, our findings demonstrate that SparseQJL effectively balances perplexity, memory efficiency, and computational overhead, making it a promising and robust solution for large-scale language model compression.

VII. Conclusion

In this work, we have made significant progress by integrating the QJL framework with SparseGPT for LLaMA models. Ongoing modifications aim to extend this integration to the OPT architecture among others, enabling SparseGPT-based pruning for robust applicability. Our results highlight the potential of combining a novel quantization technique and pruning to achieve substantial memory savings with reasonable trade-offs in accuracy. To further enhance accuracy retention, we plan to refine our methodology, including exploring quantization-aware pruning techniques and variants of JL transformations.

Looking ahead, our immediate focus is on successfully running QJL on OPT models and implementing sparse pruning. Beyond this, we aim to experiment with alternative approaches, such as sparse random projection in place of QJL, and refine pruning methods to improve efficiency and accuracy. Despite the challenges, we are committed to advancing SparseQJL beyond the scope of this project. Our ultimate goal is to produce results that contribute meaningfully to the field of efficient transformer optimization, paving the way for a publication-worthy solution.

Acknowledgment

We would like to thank **Dr. Kaoutar El Maghraoui** and **Dr. Naigang Wang** for their continuous support and feedback throughout this project.

References

[Beltagy et al.(2020)] Iz Beltagy, Matthew E. Peters, and Arman Cohan. 2020. Longformer: The Long-Document Transformer. arXiv:2004.05150 [cs.CL] https://arxiv.org/abs/2004.05150

[Dasgupta and Gupta(2003)] Sanjoy Dasgupta and Anupam Gupta. 2003. An elementary proof of a theorem of Johnson and Lindenstrauss. *Random Structures & Algorithms* 22, 1 (2003), 60–65. https://doi.org/10.1002/rsa.10073 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/rsa.10073

[Dettmers et al.(2022)] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. LLM.int8(): 8-bit Matrix Multiplication for Transformers at Scale. arXiv:2208.07339 [cs.LG] https://arxiv.org/abs/2208.07339

[Frantar and Alistarh(2023)] Elias Frantar and Dan Alistarh. 2023. SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. arXiv:2301.00774 [cs.LG] https://arxiv.org/abs/2301.00774

[Frantar et al.(2023)] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2023. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. arXiv:2210.17323 [cs.LG] https://arxiv.org/abs/2210.17323

[Hooper et al.(2024)] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, Michael W. Mahoney, Yakun Sophia Shao, Kurt Keutzer, and Amir Gholami. 2024. KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization. arXiv:2401.18079 [cs.LG] https://arxiv.org/abs/2401.18079

[Lin et al.(2024)] Yujun Lin, Haotian Tang, Shang Yang, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. 2024. QServe: W4A8KV4 Quantization and System Co-design for Efficient LLM Serving. arXiv:2405.04532 [cs.CL] https://arxiv.org/abs/2405.04532

[Liu et al.(2024)] Zirui Liu, Jiayi Yuan, Hongye Jin, Shaochen Zhong, Zhaozhuo Xu, Vladimir Braverman, Beidi Chen, and Xia Hu. 2024. KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache. *arXiv preprint arXiv:2402.02750* (2024).

[Marcus et al.(1994)] Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The Penn Treebank: Annotating Predicate Argument Structure. In *Human Language Technology: Proceedings of a Workshop held at Plainsboro, New Jersey, March 8-11, 1994*. https://aclanthology.org/H94-1020

[Merity et al.(2016)] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. 2016. Pointer Sentinel Mixture Models. arXiv:1609.07843 [cs.CL] https://arxiv.org/abs/1609.07843

[Park et al.(2024)] Gunho Park, Baeseong Park, Minsub Kim, Sungjae Lee, Jeonghoon Kim, Beomseok Kwon, Se Jung Kwon, Byeongwook Kim, Youngjoo Lee, and Dongsoo Lee. 2024. LUT-GEMM: Quantized Matrix Multiplication based on LUTs for Efficient Inference in Large-Scale Generative Language Models. arXiv:2206.09557 [cs.DC] https://arxiv.org/abs/2206.09557

[Ribar et al.(2024)] Luka Ribar, Ivan Chelombiev, Luke Hudlass-Galley, Charlie Blake, Carlo Luschi, and Douglas Orr. 2024. SparQ Attention: Bandwidth-Efficient LLM Inference. arXiv:2312.04985 [cs.LG] https://arxiv.org/abs/2312.04985

[Shi et al.(2024)] Luohe Shi, Hongyi Zhang, Yao Yao, Zuchao Li, and Hai Zhao. 2024. Keep the Cost Down: A Review on Methods to Optimize LLM' s KV-Cache Consumption. arXiv:2407.18003 [cs.CL] https://arxiv.org/abs/2407.18003

[Strubell et al.(2019)] Emma Strubell, Ananya Ganesh, and Andrew McCallum. 2019. Energy and Policy Considerations for Deep Learning in NLP. arXiv:1906.02243 [cs.CL] https://arxiv.org/abs/1906.02243

[Touvron et al.(2023)] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] https://arxiv.org/abs/2307.09288

[Vaswani et al.(2023)] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL] https://arxiv.org/abs/1706.03762

[Wolf et al.(2020)] 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. 2020. HuggingFace's Transformers: State-of-the-art Natural Language Processing. arXiv:1910.03771 [cs.CL] https://arxiv.org/abs/1910.03771

[Xiao et al.(2024a)] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2024a. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. arXiv:2211.10438 [cs.CL] https://arxiv.org/abs/2211.10438

[Xiao et al.(2024b)] Guangxuan Xiao, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024b. Efficient Streaming Language Models with Attention Sinks. arXiv:2309.17453 [cs.CL] https://arxiv.org/abs/2309.17453

[Yang et al.(2024)] June Yong Yang, Byeongwook Kim, Jeongin Bae, Beomseok Kwon, Gunho Park, Eunho Yang, Se Jung Kwon, and Dongsoo Lee. 2024. No Token Left Behind: Reliable KV Cache Compression via Importance-Aware Mixed Precision Quantization. arXiv:2402.18096 [cs.LG] https://arxiv.org/abs/2402.18096

[Yao et al.(2022)] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. 2022. ZeroQuant: Efficient and Affordable Post-Training Quantization for Large-Scale Transformers. arXiv:2206.01861 [cs.CL] https://arxiv.org/abs/2206.01861

[Yue et al.(2024)] Yuxuan Yue, Zhihang Yuan, Haojie Duanmu, Sifan Zhou, Jianlong Wu, and Liqiang Nie. 2024. WKVQuant: Quantizing

Weight and Key/Value Cache for Large Language Models Gains More. arXiv:2402.12065 [cs.LG] https://arxiv.org/abs/2402.12065

[Zandieh et al.(2024)] Amir Zandieh, Majid Daliri, and Insu Han. 2024. QJL: 1-Bit Quantized JL Transform for KV Cache Quantization with Zero Overhead. arXiv:2406.03482 [cs.LG] https://arxiv.org/abs/2406.03482

[Zhang et al.(2024)] Michael Zhang, Kush Bhatia, Hermann Kumbong, and Christopher Ré. 2024. The Hedgehog the Porcupine: Expressive Linear Attentions with Softmax Mimicry. arXiv:2402.04347 [cs.LG] https://arxiv.org/abs/2402.04347

[Zhang et al.(2022)] Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, Todor Mihaylov, Myle Ott, Sam Shleifer, Kurt Shuster, Daniel Simig, Punit Singh Koura, Anjali Sridhar, Tianlu Wang, and Luke Zettlemoyer. 2022. OPT: Open Pre-trained Transformer Language Models. arXiv:2205.01068 [cs.CL] https://arxiv.org/abs/2205.01068

APPENDIX

TABLE III
ADDITIONAL RESULTS

Model	**Method**	**WikiText2**	**PTB**	**C4**
OPT-2.7b	Baseline	12.471	17.974	13.343
	50% KV Prune	13.214	18.580	14.602
	50% KV Prune + W4	13.314	19.099	14.746
	50% Prune	13.527	20.377	15.807
	50% Prune + W4	14.143	21.758	16.362
	QJL			
	SparseQJL			
OPT-6.7b	Baseline	10.860	15.769	12.712
	50% KV Prune	11.384	16.108	12.855
	50% KV Prune + W4	12.048	16.948	14.426
	50% Prune	11.538	17.122	13.771
	50% Prune + W4	12.086	17.923	14.246
	QJL			
	SparseQJL			
LLAMA3-8B	Baseline	5.538	10.181	8.998
	50% KV Prune	5.953	10.686	9.757
	50% KV Prune + W4	5.782	10.426	9.466
	50% Prune	8.478	13.748	13.845
	50% Prune + W4	396.864	51.784	62.621
	QJL	5.538	10.181	8.999
	SparseQJL	8.545	13.851	13.882
LLAMA3-8B	Baseline	7.461	13.306	12.341
Instruct	50% KV Prune	7.461	13.306	12.341
	50% KV Prune + W4	7.683	13.763	12.800
	50% Prune	10.361	17.622	16.535
	50% Prune + W4	618.115	337.197	318.575
	QJL	7.461	13.304	12.341
	SparseQJL	10.417	17.506	16.576