UC Irvine
UC Irvine Electronic Theses and Dissertations

Title
APQ: Toward Arbitrary-dimensional Quantization of Large Language Models with Extended Product Quantization

Permalink
https://escholarship.org/uc/item/06w5841q

Author
Wang, Yian

Publication Date
2025

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, available at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA,
IRVINE

APQ: Toward Arbitrary-dimensional Quantization of Large Language Models with Extended Product Quantization

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Electrical and Computer Engineering

by

Yian Wang

Thesis Committee:
Asistant Professor Hyoukjun Kwon, Chair
Assistant Professor Sitao Huang
Associate Professor Salma Elmalaki

2025

© 2025 Yian Wang

DEDICATION

To my family, mentors, and friends.

TABLE OF CONTENTS

Page

LIST OF FIGURES **iv**

LIST OF TABLES **v**

LIST OF ALGORITHMS **vi**

ACKNOWLEDGMENTS **vii**

ABSTRACT OF THE THESIS **viii**

1 Introduction **1**

2 Background and Motivation **3**
2.1 Scalar Quantization 3
2.2 Vector Quantization and Product Quantization 5

3 APQ **8**
3.1 APQ Methodology 8
3.1.1 APQ Quantization 8
3.1.2 APQ Inference 10
3.2 APQ Design Space 12

4 Experiments **17**
4.1 Design Space Exploration 17
4.2 Experimental Settings 20
4.2.1 Baselines 20
4.2.2 LLM Workloads 21
4.2.3 Configurations 21
4.3 Methodology 22
4.4 Results 23
4.4.1 Model Performance with Compression Rate Tradeoff 23
4.4.2 Scalability 25

5 Conclusion and Future Work **26**

Bibliography **28**

LIST OF FIGURES

Page

3.1 Example APQ Quantization Process. In this example, the number of subspaces (N_{ss}) is 4. ①: The weight matrix is split into N_{ss} subspaces; ②: Run K-means clustering algorithm on single subspace; ③: Apply ② to all subspaces independently. 9

3.2 Example APQ Weight Reconstrcution Process. In this example, the number of subspaces (N_{ss}) is 4. Lookups in different subspaces are independant and can be performed in parallel. ss' is the reconstructed subspace; W' stands for the reconstructed weight matrix. 11

3.3 APQ Inference Process with hardware acceleration opportunities. Upperleft: The weight is split into N_{ss}=4 subspaces, and input corresponding to the subspaces is highlighted. Lowerleft: Vector-matrix multiplication is applied between one input row and cluster table to generate partial sum buffer. Upperright: Each column in index table is used to index into the partial sum buffer and the results are then accumulated to generate one entry in the output matrix. Lowerright: Repeat the lookup-and-accumulate process in upperright for all input rows and index table columns to generate full output matrix. 13

4.1 Design Space Exploration results for LLaMA2-7B, Qwen2.5-7B, and Mistral-7B models on compression rate and perplexity score on wikitext2 testset. 19

LIST OF TABLES

Page

4.1 APQ result comparision between dim=0 and dim=1 with LLaMA2-7B model. 17
4.2 Prefill and Decode latency of LLaMA2-7B on RTX 3090. The sample sequence length is 128. 18
4.3 Prefill and Decode latency of Qwen2.5-7B on RTX 3090. The sample sequence length is 128. 18
4.4 Prefill and Decode latency of Mistral-7B on RTX 3090. The sample sequence length is 128. 20
4.5 Selected APQ configurations. 21
4.6 Compress ratio (CR) and perplexity results on Wikitext2 testset (Wiki). . . 23
4.7 Results of 8 0-shot tasks on LLaMA2-7B model. 24
4.8 Results of 8 0-shot tasks on Qwen-2.5-7B model. 24
4.9 Results of 8 0-shot tasks on Mistral-7B-v0.3 model. 25
4.10 Scalability of APQ with LLaMA models 25

LIST OF ALGORITHMS

Page

1 APQ Quantization Process 10
2 APQ Weight Reconstruction Process 12
3 APQ Inference Process 14

ACKNOWLEDGMENTS

I would like to sincerely thank Professor Hyoukjun Kwon, my advisor and thesis committee chair, for his guidance, thoughtful feedback, and consistent support throughout this research. His expertise and encouragement have played a key role in both my academic and personal growth.

I am also grateful to my committee members, Professor Sitao Huang and Professor Salma Elmalaki, for their time, helpful feedback, and suggestions, which have improved the quality of this thesis.

Thank you to the members of the ISA Lab at the University of California, Irvine, for their collaboration and insightful discussions that enriched my research. I also appreciate the University of California, Irvine, for providing the resources and environment that made this work possible.

Finally, I am truly thankful to my friends and family for their constant encouragement and support. Their belief in me has been a steady source of motivation throughout this journey.

ABSTRACT OF THE THESIS

APQ: Toward Arbitrary-dimensional Quantization of Large Language Models with Extended Product Quantization

By

Yian Wang

Master of Science in Electrical and Computer Engineering

University of California, Irvine, 2025

Asistant Professor Hyoukjun Kwon, Chair

Multi-Codebook Quantization (MCQ) has emerged as a promising technique for compressing large language models, offering significant storage and computational savings. Product Quantization, a widely used variant of MCQ, partitions weight tensors into smaller subspaces and quantizes each subspace independently, enabling efficient compression and fast approximate computations. However, existing Product Quantization methods often rely on fixed partitions and do not fully explore the broader design space available for large language model deployments.

In this work, we propose APQ, a flexible and extensible product quantization framework that supports arbitrary-dimensional quantization of weight tensors in large language models. Through extensive empirical analysis, we demonstrate that APQ matches conventional scalar quantization methods—including 4-bit integer (INT4) and 8-bit integer (INT8) baselines—by better capturing the structural redundancy of model weights, while preserving model performance across multiple benchmarks. We also explore a wide design space for Product Quantization, such as subvector dimensionality and codebook size, and study their impact on model performance, compression rate, and hardware performance. Our results reveal key trade-offs between compression rate, model performance, and system efficiency.

Furthermore, we show how APQ can be integrated into existing pipelines for large language models without the need for retraining or finetuning, making it suitable for efficient post-training quantization.

This work provides a comprehensive analysis of the Product Quantization design space and highlights the importance of flexible, hardware-friendly quantization strategies for scalable deployment of large language models.

Chapter 1

Introduction

The rapid growth of large language models (LLMs) has created an urgent need for efficient compression techniques that reduce storage, memory, and inference latency, without compromising model performance. Among various approaches, post-training quantization (PTQ) has emerged as the de facto standard due to its simplicity and wide hardware compatibility. PTQ enables compression of pretrained models without requiring retraining, making it highly practical for real-world deployment. A commonly used PTQ technique is scalar quantization (SQ), which quantizes each weight or activation element individually, typically converting high-precision formats such as FP32 or FP16 into lower-precision formats like INT8 or INT4. While effective, scalar quantization often introduces limited trade-offs in compression and performance, demands careful calibration, and incurs runtime overhead due to explicit dequantization during inference.

In this paper, we present APQ (**A**rbitrary-dimensional **P**roduct **Q**uantiza-tion), a novel post-training quantization framework designed as a drop-in replacement for scalar quantization. APQ replaces traditional matrix multiplication in LLMs with massive parallel table lookups, providing Multi-Codebook Quantization (MCQ) as a solution in addition to scalar

quantization. APQ achieves compression rates comparable to state-of-the-art INT4-based schemes, while maintaining competitive performance on language understanding, generation, and reasoning tasks. Unlike integer-based methods, APQ operates entirely in floating-point, eliminating the need for runtime dequantization. This design simplifies the inference pipeline, reduces latency, and improves hardware efficiency. Additionally, APQ requires no calibration or fine-tuning, facilitating straightforward application to existing models.

APQ also introduces flexibility in compression granularity by varying the design parameters, providing a continuum of compression-performance trade-offs. This enables finer control over model size and accuracy, making APQ suitable for diverse deployment targets ranging from edge devices to large-scale inference servers.

In summary, APQ reduces model size while offering multiple configuration options to suit different hardware and latency requirements. Its combined software and hardware advantages make it a practical and scalable quantization solution for next-generation LLMs. Our contributions are as follows:

- We introduce APQ, a flexible post-training quantization framework that generalizes product quantization to arbitrary-dimensional partitions, enabling compression without loss in model performance;
- We provide a comprehensive analysis of the APQ design space, quantifying trade-offs between memory usage, model performance, and inference latency across multiple LLMs and benchmarks;
- We demonstrate that APQ achieves compression rates and model performance comparable to state-of-the-art INT4 quantization schemes, while maintaining floating-point compatibility.

Chapter 2

Background and Motivation

2.1 Scalar Quantization

Scalar quantization is a foundational technique for compressing machine learning models by reducing the precision of individual weights, activations, or key-value caches to a smaller number of bits, typically 8-bit integers (INT8) or 4-bit integers (INT4). In the context of LLMs, scalar quantization has received significant attention due to its simplicity, hardware compatibility, and relatively low computational overhead. Scalar quantization typically maps floating-point weights to lower-precision integer representations using a linear transformation. This process involves determining a scaling factor and an offset for each weight, which can be computed using the minimum and maximum values of the weight tensor. The quantization process can be expressed mathematically as follows:

$$Q(w) = \text{round}\left(\frac{x - \min(w)}{\max(w) - \min(w)} \cdot (2^b - 1)\right) \tag{2.1}$$

However, applying scalar quantization directly to pretrained LLMs often leads to severe

accuracy degradation, particularly in high-sensitivity components such as attention and feed-forward layers, as Equation 2.1 suffers from large quantization errors when extremely large or small values are present in the weight tensors as outliers.

To address these challenges, several recent methods have introduced advanced quantization-aware preprocessing and calibration strategies. GPTQ (Gradient-based Post-Training Quantization) [7] is a widely adopted method for compressing LLMs using INT4 or INT3 weights without requiring any retraining. GPTQ performs layer-wise quantization by minimizing the quantization error with respect to a local approximation of the loss, typically using second-order information. Specifically, it approximates the Hessian of the loss with respect to the weights using a small calibration dataset and greedily determines optimal rounding decisions to minimize the quantization-induced loss. This Hessian-aware strategy allows GPTQ to preserve accuracy more effectively than uniform or naive rounding methods, especially in large transformer models.

SmoothQuant [23] shifts the focus to activation quantization by proposing a transformation that reduces the activation range before quantization. It introduces a simple affine transformation based on a per-channel scaling factor that "smooths" the activation distribution. When applied before static quantization, this method significantly improves post-training quantization accuracy, especially when activations are also quantized to low-bit formats.

AWQ (Activation-aware Weight Quantization) [13] is one method that focuses on identifying and handling outlier weights instead of focusing on activations. AWQ selectively rescales columns of weight matrices to reduce the dynamic range, allowing better alignment with INT4 representation. It further avoids quantizing the most sensitive channels, preserving them in higher precision (e.g., FP16), which improves accuracy without retraining.

Despite their effectiveness, scalar quantization methods operate independently on each element and may fail to capture higher-order correlations within or across tensors.

Group quantization methods, which quantize small groups of elements together, offer some improvement by capturing local dependencies, but they still fall short in modeling the complex, global correlations present in large neural network weights. This motivates the exploration of more structured quantization methods, such as Product Quantization, which partition tensors into subvectors and quantize them jointly. Our work builds on this motivation and proposes a generalization of PQ to arbitrary dimensions, aiming to bridge the gap between fine-grained quantization and hardware-efficient deployment.

2.2 Vector Quantization and Product Quantization

Vector quantization (VQ) [9] is a powerful technique that extends scalar quantization by treating vectors of elements as quantization unit and quantizing them jointly. This approach allows for better representation of the underlying data distribution and can significantly reduce the quantization error compared to scalar methods. VQ is particularly effective in scenarios where the data exhibits strong correlations, such as in neural network weights.

Product quantization (PQ) [10] is a specific form of vector quantization that partitions the entire space into smaller orthogonal subspaces and quantizes subvectors within each subspace independently. This method leverages the idea that the joint distribution of the subvectors can be approximated by the product of their marginal distributions, leading to a more compact representation.

VQ and PQ relied on clustering algorithms, such as k-means, to learn the quantization codebooks. K-Means works by iteratively assigning all vectors (data points) to a fixed number of groups (number of clusters) based on given distance methics, and update the group centers by the average of each group. Integrating K-Means with PQ and VQ involves clustering the input data into K_s clusters and representing each cluster by its center. The quantization

process then maps each input vector to the nearest center in the codebook, eliminating the need to store the entire input. This approach effectively reduces the dimensionality of the data while preserving its structure.

Vector Quantization and Product Quantization on input have been adopted as the basis for several recent works in the field of neural network compression. Maddness [3] proposes using learnable hash function to avoid the multiply-add operations during mapping. PECAN [17] relied on Content Addressable Memory (CAM) for simple mapping. Following Differentiable Product Quantization (DPQ) [4], PECAN, LUT-NN [20], and PQA [1] make the learning process compatible with gradient-based learning framework, and propose custom hardware architectures to facilitate the efficient implementation of PQ-based models. Additionally, LUT-DLA [12] includes a co-design space search engine to accommodate various applications and many sub-optimal design points. These methods leverage the advantages of VQ and PQ to reduce the number of parameters and computational requirements, making them suitable for deployment on resource-constrained devices.

However, current dominating approaches apply VQ and PQ on input matrices. As indicated in the above methods, the PQ and VQ quantization parameters tend to be small, indicating a hard compression-performance trade-off in these input-targetted methods. Additionally, their implications for LLMs remain underexplored, especially given that matrix multiplications in LLMs are 10-100$\times$ larger in scale compared with fully-connected layer in neural networks. As a result, important questions remain regarding the effectiveness and practicality of these methods for LLM deployment. In particular, the applicability of product quantization to LLMs has not been thoroughly investigated, and it remains unclear how PQ can be generalized to arbitrary dimensions or what trade-offs exist between compression rate, model performance, and hardware efficiency.

In this thesis, we aim to answer the following research question: *Can Product Quantization be effectively applied to large language models? Specifically, can arbitrary-*

dimensional product quantization provide an effective and hardware-efficient alternative to traditional scalar quantization for compressing large language models, while maintaining competitive model performance and inference efficiency?

Chapter 3

APQ

In this section, we introduce APQ, a framework that enables product quantization in LLMs.

3.1 APQ Methodology

3.1.1 APQ Quantization

Product Quantization operates by partitioning the weight matrix $W \in \mathbb{R}^{d \times d'}$ into N_{ss} subspaces. The partitioning dimension, denoted as dim_{ss}, can be selected along either the row (d) or column (d') dimension. We use a binary variable dim to indicate this choice: if $dim = 0$, we split along the d (row) dimension; if $dim = 1$, we split along the d' (column) dimension. The remaining dimension is then treated as the datapoint dimension, dim_{dp}. The weight matrix is divided into N_{ss} subspaces along the selected subspace dimension. Thus,

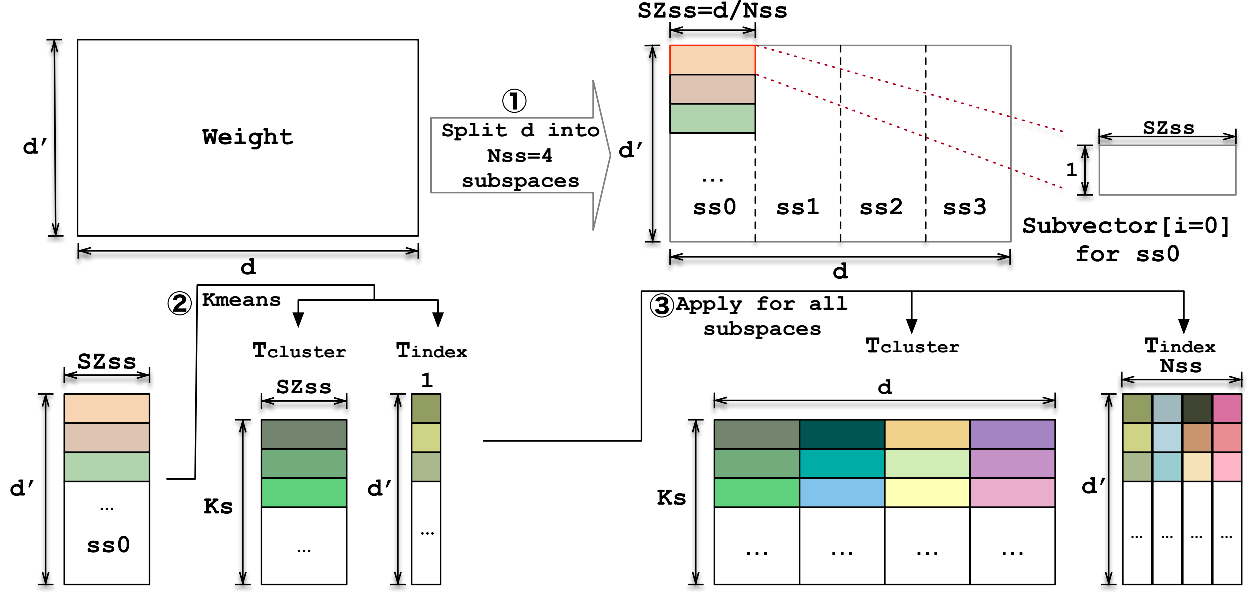

Figure 3.1: Example APQ Quantization Process. In this example, the number of subspaces (N_{ss}) is 4. ①: The weight matrix is split into N_{ss} subspaces; ②: Run K-means clustering algorithm on single subspace; ③: Apply ② to all subspaces independently.

we have each weight subspace (W_{ss}) as:

$$W_{ss} \in \begin{cases} \mathbb{R}^{\frac{d}{N_{ss}} \times d'}, & \text{if dim} = 0 \\ \mathbb{R}^{d \times \frac{d'}{N_{ss}}}, & \text{if dim} = 1 \end{cases} \tag{3.1}$$

The size of each subspace is thus $SZ_{ss} = \frac{dim_{ss}}{N_{ss}}$.

After selecting the subspace dimension, we next determine the number of clusters K_s assigned to each subspace. This value is set by the number of bits b allocated for quantization, with the relationship $K_s = 2^b$. As a result, each subspace contains K_s clusters, and across all N_{ss} subspaces, the total number of clusters is $K_s \times N_{ss}$.

Within each subspace, we independently apply the K-Means clustering algorithm with K_s clusters along the datapoint dimension dim_{dp}. This approach ensures that the clustering process is specifically adapted to the structure of each subspace. For every subspace, K-Means produces a table of cluster centers $T^{ss}_{\text{cluster}} \in \mathbb{R}^{K_s \times SZ_{ss}}$ and an index table $T^{ss}_{\text{index}} \in$

$\mathbb{R}^{dim_{dp}}$. These tables are then aggregated across all subspaces to obtain the final cluster center table $T_{\text{cluster}} \in \mathbb{R}^{N_{ss} \times K_s \times SZ_{ss}}$ and the index table $T_{\text{index}} \in \mathbb{R}^{N_{ss} \times dim_{dp}}$. The complete quantization process is detailed in Algorithm 1 and illustrated in Figure 3.1.

Algorithm 1 APQ Quantization Process

Require: Weight matrix W with shape (d, d'), number of clusters K_s, number of subspaces N_{ss}, dimension to perform APQ dim.

Ensure: Cluster table T_{cluster}, index table T_{index}

1: **let** dim_{dp} be the data point dimension.
2: $dim_{ss} \leftarrow d$ if $dim \leftarrow 0$ else d', dim_{dp} be the number of data points
3: $SZ_{ss} \leftarrow \frac{dim_{ss}}{N_{ss}}$
4: Initialize $T_{\text{cluster}} \in \mathbb{R}^{N_{ss}, K_s, SZ_{ss}}$, $T_{\text{index}} \in \mathbb{R}^{N_{ss}, n_{dp}}$
5: **parallel for** $ss_i \leftarrow 1$ to N_{ss} **do**
6: $\quad (C, I) \leftarrow \text{KMeans}(W_{ss_i}, K_s)$
7: $\quad T_{\text{cluster}}[ss_i] \leftarrow C$
8: $\quad T_{\text{index}}[ss_i] \leftarrow I$
9: **end parallel for**
10: **return** $T_{\text{cluster}}, T_{\text{index}}$

3.1.2 APQ Inference

A straightforward approach to perform inference with APQ is to reconstruct the original weight matrix by utilizing the cluster centers as lookup tables indexed by indices from corresponding index tables. Since the subspaces are independent by construction, this reconstruction can be fully parallelized as indicated in the nested parallel for loop in Algorithm 2.

However, this reconstruction approach is inefficient, as it temporarily requires the entire weight matrix to be materialized in memory in addition to storing the cluster centers and index tables, which undermines the purpose of quantization. To address this overhead, we propose a reconstruction-free inference method. Instead of recovering the full weight matrix, we compute a series of small lookup tables—one for each subspace—by performing vector-matrix multiplications between the input and the relatively small cluster centers.

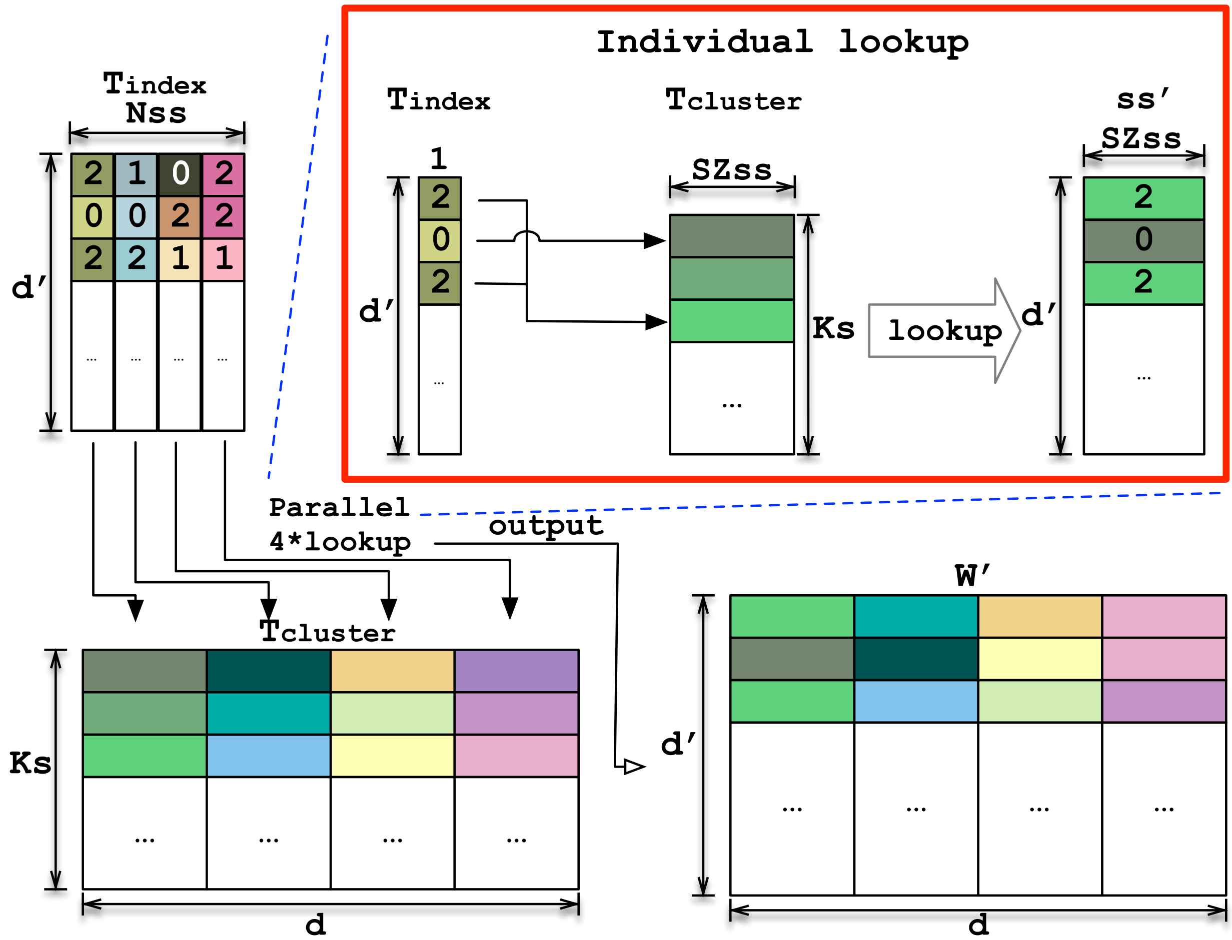

Figure 3.2: Example APQ Weight Reconstrcution Process. In this example, the number of subspaces (N_{ss}) is 4. Lookups in different subspaces are independant and can be performed in parallel. ss' is the reconstructed subspace; W' stands for the reconstructed weight matrix.

Algorithm 2 APQ Weight Reconstruction Process

Require: Cluster table T_{cluster}, index table T_{index}, number of subspaces N_{ss}, size of each subspace SZ_{ss}, dimension to perform APQ dim

Ensure: Reconstructed weight matrix W'

1: **parallel for** $ss_i \leftarrow 1$ to N_{ss} **do**

2: **parallel for** $i \leftarrow 1$ to dim_{dp} **do**

3: $v \leftarrow T_{\text{cluster}}[ss_i][T_{\text{index}}[ss_i][i]]$

4: Assign v to

$$\begin{cases} W'[ss_i \times SZ_{ss} : (ss_i + 1) \times SZ_{ss}][i], & \text{if } dim = 0 \\ W'[i][ss_i \times SZ_{ss} : (ss_i + 1) \times SZ_{ss}], & \text{if } dim = 1 \end{cases}$$

5: **end parallel for**

6: **end parallel for**

7: **return** W'

This approach leverages the high memory and network-on-chip (NoC) bandwidth to reduce the number of computations. Since each subspace typically contains far fewer cluster centers than the number of original weight vectors, this method reduces the number of floating-point operations (FLOPs) by an average of 7×, as suggested by our experimental results.

Once the lookup tables are computed, the final outputs are generated by performing index-based lookups and accumulating results across subspaces. This reconstruction-free approach eliminates redundant computations inherent in traditional dense matrix multiplications, where many rows of the weight matrix may perform nearly identical operations due to sparsity and redundancy. By leveraging the clustered structure of quantized weights, APQ reduces repeated multiply-accumulate operations with lightweight, parallel table lookups, significantly reducing FLOPs without increasing memory pressure. The complete inference procedure is detailed in Algorithm 3 and in Figure 3.3.

3.2 APQ Design Space

APQ targets linear layers in models, which are typically the most memory-intensive components. Suppose the baseline model is in half-precision (FP16, 2 bytes).

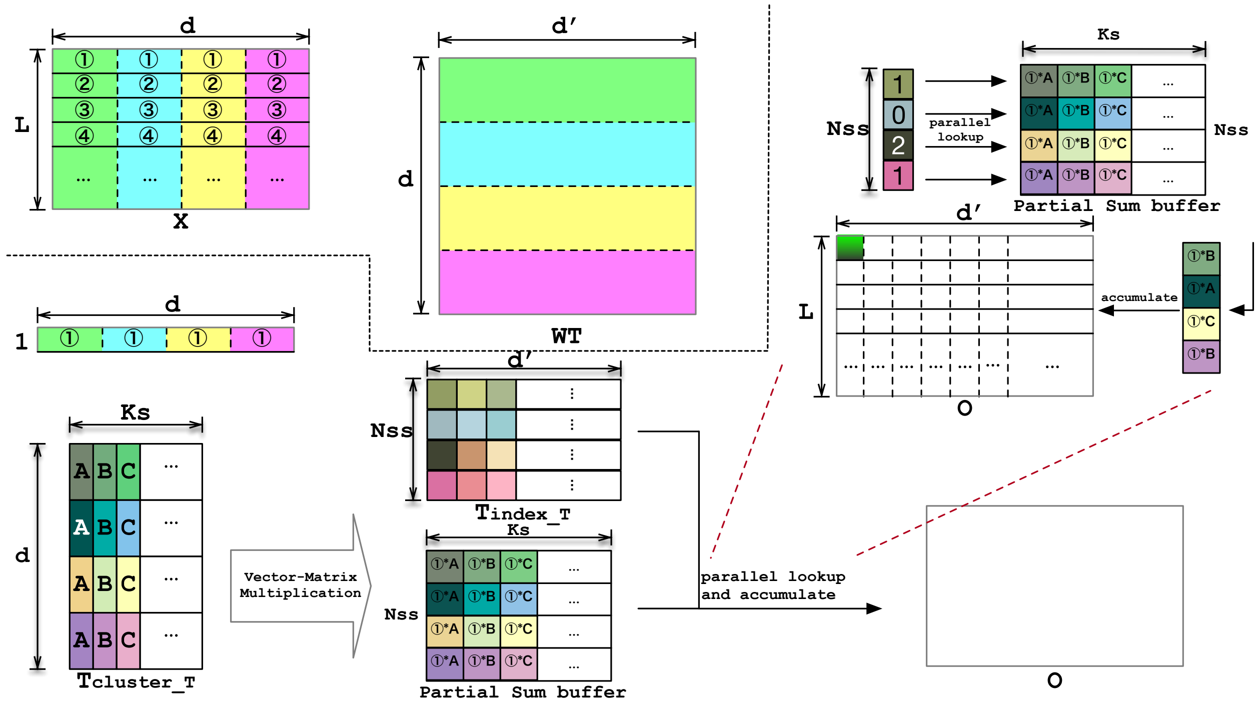

Figure 3.3: APQ Inference Process with hardware acceleration opportunities. Upperleft: The weight is split into N_{ss}=4 subspaces, and input corresponding to the subspaces is highlighted. Lowerleft: Vector-matrix multiplication is applied between one input row and cluster table to generate partial sum buffer. Upperright: Each column in index table is used to index into the partial sum buffer and the results are then accumulated to generate one entry in the output matrix. Lowerright: Repeat the lookup-and-accumulate process in upperright for all input rows and index table columns to generate full output matrix.

Algorithm 3 APQ Inference Process

Require: Input activation A with shape (L, d), cluster table T_{cluster}, index table T_{index}, number of subspaces N_{ss}, size of each subspace SZ_{ss}, dimension to perform APQ dim

Ensure: Output of current layer O

1: Initialize $O \in \mathbb{R}^{L,d}$ with zeros.
2: **parallel for** $ss_i \leftarrow 1$ to N_{ss} **do**
3:    **parallel for** $i \leftarrow 1$ to dim_{dp} **do**
4:       **let** lut be a lookup table of size K_s
5:       $lut \leftarrow T_{\text{cluster}}[ss_i][i] \times A[i][ss_i * SZ_{ss} : (ss_i + 1) * SZ_{ss}]$ in parallel
6:       $O += lut[T_{\text{index}}[ss_i][d_i]]$
7:    **end parallel for**
8: **end parallel for**
9: **return** O

Since there are $(N_{ss} \times K_s \times SZ_{ss})$ elements in the cluster table T_{cluster} and each element takes 16 bits in memory, the total amount of memory T_{cluster} takes in bits is:

$$S^{\text{layer}}_{\text{cluster}} = N_{ss} \times K_s \times SZ_{ss} \times 16 \tag{3.2}$$

Similarly, the index table T_{index} contains $(N_{ss} \times dim_{dp})$ elements. Each index specifies which cluster center is selected from a subspace-specific codebook and only needs enough bits to represent values in the range $[0, K_s - 1]$. Therefore, the bitwidth required per index is $\lceil \log_2(K_s) \rceil$. The total memory for the index table in bits is thus:

$$S^{\text{layer}}_{\text{index}} = N_{ss} \times dim_{dp} \times \lceil \log_2(K_s) \rceil \tag{3.3}$$

Since we replace the original weight matrix with the cluster centers and indices, the total memory size of the APQ-quantized linear layer, S_{apq}, becomes:

$$S^{\text{layer}}_{\text{apq}} = S^{\text{layer}}_{\text{cluster}} + S^{\text{layer}}_{\text{index}} \tag{3.4}$$

$$= 16 \times K_s \times dim_{ss} + \lceil \log_2(K_s) \rceil \times N_{ss} \times dim_{dp} \tag{3.5}$$

Conclusions can be drawn from Equation 3.5 that for a given model, the size of the APQ-quantized linear layer is only determined by dim, K_s and N_{ss}, since model precision, dim_{ss} and dim_{dp} are fixed given dim as discussed in Section 3.1.1.

To compute the memory footprint of the entire APQ-quantized model, we ned to aggregate the memory of all APQ-quantized linear layers along with auxiliary components such as LayerNorm and embedding layers. Suppose S_{apq} represents the total memory footprint of the APQ-quantized model, and S_{base} represents the memory footprint of the original model. We can calculate the compression rate (CR) as follows:

$$\text{CR} = \left(1 - \frac{S_{\text{apq}}}{S_{\text{base}}}\right) \times 100\% \tag{3.6}$$

To further improve storage efficiency, we apply deduplication by eliminating repeated cluster centers within each subspace after clustering. Redundant centers are removed independently within each subspace, since identical centers across subspaces are treated as distinct. Consequently, the size derived from Equation 3.5 represents a theoretical upper bound of the memory footprint given a (dim, K_s, N_{ss}) configuration.

We also observed that for integer models, APQ does not yield any favorable compression-performance trade-offs. As indicated by Equation 3.5, compression is only achieved if $S_{\text{apq}}^{\text{layer}} < N \times dim_{ss} \times dim_{dp}$, where the latter represents the original layer size in bits for an INT-N format. Furthermore, the number of clusters must satisfy $K_s < 2^N$; otherwise, the index table alone would exceed the original layer size, as shown in Equation 3.3. Since most integer models use low precisions (3-4 bits), the number of unique values in the weight matrix is already limited. In such cases, even the maximum feasible number of clusters, 2^{N-1}, is only half the number of unique values, resulting in overly aggressive quantization that significantly degrades model performance. **Therefore, we define the design space of APQ as determined by the following parameters in the floating-point domain:**

$$\text{Design Space} = \{K_s, N_{ss}, dim\} \tag{3.7}$$

A design space exploration of APQ is then conducted by varying both the number of clusters K_s and the subspace size SZ_{ss}, using common power-of-two values for both $dim = 0$ and $dim = 1$. Specifically, SZ_{ss} is chosen from 1 up to dim_{ss}, spanning the weight matrix. When $SZ_{ss} = 1$, the method reduces to simple element-wise K-Means quantization; when $SZ_{ss} = dim_{ss}$, it becomes equivalent to vector quantization. Similarly, K_s is varied from 1 to dim_{dp}. The case $K_s = 1$ represents an extreme setting where the K-means cluster table contains only the average of all samples, while $K_s = dim_{dp}$ corresponds to no quantization, as each sample is uniquely represented.

We also consider model performance and end-to-end latency in the design space as the quality of the method. Together with compression rate, we can evaluate the trade-off between model performance, latency and compression rate. Detailed design space exploration results are presented in section 4.1.

Chapter 4

Experiments

4.1 Design Space Exploration

We ran experiments on both $dim = 0$ and $dim = 1$ configurations to explore the design space of APQ. The comparison of model performance on selected tasks between these two configurations is shown in Table 4.1. We observe that the performance gap between these two configurations is significant, and we find that $dim = 0$ consistently yields slightly better performance than $dim = 1$ across all evaluated tasks on all configurations. Therefore, for the remainder of this paper, we focus on the $dim = 0$ configuration as the default setting for APQ.

Table 4.1: APQ result comparision between dim=0 and dim=1 with LLaMA2-7B model.

Method	dim	CR	Wiki	BQ	PIQA	SIQA	HS	WG	ARC-e	ARC-c	OBQA	Average
Baseline	-	0%	5.47	77.70%	79.05%	46.11%	76.00%	68.90%	74.49%	46.16%	44.20%	64.07%
APQ Flex	0	58.85%	5.65	76.51%	78.51%	45.04%	74.72%	69.53%	73.44%	44.45%	43.80%	63.25.%
	1	59.06%	NaN	37.83%	49.51%	32.91%	25.05%	49.57%	25.08%	22.70%	27.60%	33.78%
APQ Edge	0	75.94%	6.04	75.75%	77.42%	44.11%	72.81%	67.88%	69.82%	42.92%	41.60%	61.54%
	1	75.94%	NaN	37.83%	49.51%	32.91%	25.04%	49.57%	25.08%	22.69%	27.60%	33.78%
APQ Server	0	63.95%	5.55	78.38%	79.00%	46.21%	75.19%	68.35%	74.32%	45.05%	43.60%	63.76%
	1	74.79%	NaN	37.83%	49.51%	32.91%	25.04%	49.57%	25.08%	22.70%	27.60%	33.78%

As discussed in section 3.2, Equation 3.5 defines the memory footprint of a single APQ-quantized linear layer. For each configuration, we calculate the resulting compression rate relative to the baseline FP16 model. The results of this analysis are summarized in Figure 4.1, where we show the compression design space for selected models.

Method	**# Bits**	**LLaMA2-7B**		
		CR	**Prefill**	**Decode**
Baseline	16-16-16	0%	37.09 ms	21.37 ms
SmoothQuant [23]	8-8-16	48.03%	47.77 ms	42.93 ms
GPTQ-W3 [7]	3-16-16	77.16%	162.89 ms	148.08 ms
GPTQ-W4 [7]	4-16-16	71.11%	34.10 ms	21.59 ms
AWQ-W3 [13]	3-16-16	77.93%	2091.97 ms	2176.54 ms
AWQ-W4 [13]	4-16-16	70.57%	32.76 ms	17.68 ms
APQ Flex	0-1-64[a]	58.85%	10700.53 ms	10864.57 ms
APQ Edge	0-4-1024[a]	75.94%	10700.53 ms	10864.57 ms
APQ Server	0-2-512[a]	63.95%	10700.53 ms	10864.57 ms

Table 4.2: Prefill and Decode latency of LLaMA2-7B on RTX 3090. The sample sequence length is 128.

Method	**# Bits**	**LLaMA2-7B**		
		CR	**Prefill**	**Decode**
Baseline	16-16-16	0%	36.12 ms	21.85 ms
SmoothQuant [23]	8-8-16	42.82%	45.38 ms	40.11 ms
GPTQ-W3 [7]	3-16-16	68.79%	169.22 ms	155.44 ms
GPTQ-W4 [7]	4-16-16	63.40%	36.25 ms	21.41 ms
AWQ-W3 [13]	3-16-16	72.08%	4529.59 ms	4559.34 ms
AWQ-W4 [13]	4-16-16	62.78%	35.07 ms	18.61 ms
APQ Flex	0-1-64[a]	52.44%	10700.53 ms	10864.57 ms
APQ Edge	0-4-1024[a]	67.63%	10700.53 ms	10864.57 ms
APQ Server	0-2-512[a]	61.95%	10700.53 ms	10864.57 ms

Table 4.3: Prefill and Decode latency of Qwen2.5-7B on RTX 3090. The sample sequence length is 128.

While compression rate is a key metric, it is equally important to consider the impact of APQ on model performance and hardware efficiency. To this end, we evaluate the perplexity of APQ across different configurations using the WikiText2 test set, providing insight into how various design choices affect language modeling quality, as shown in Figure 4.1. In addition to accuracy, we analyze the hardware latency of APQ, measuring inference speed

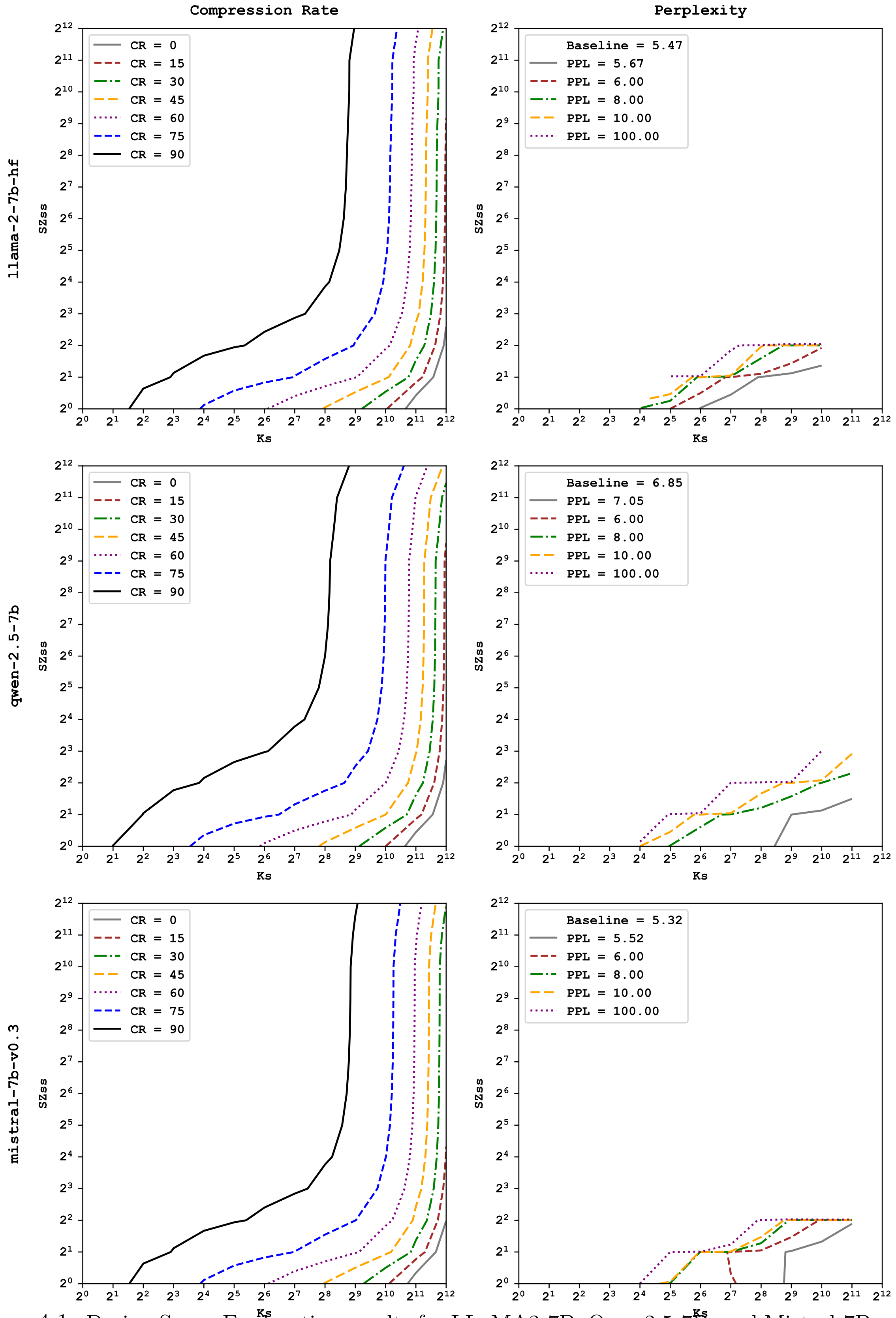

Figure 4.1: Design Space Exploration results for LLaMA2-7B, Qwen2.5-7B, and Mistral-7B models on compression rate and perplexity score on wikitext2 testset.

Method	# Bits	LLaMA2-7B		
		CR	Prefill	Decode
Baseline	16-16-16	0%	40.58 ms	22.37 ms
SmoothQuant [23]	8-8-16	4633	50.83 ms	46.13 ms
GPTQ-W3 [7]	3-16-16	7442	181.19 ms	161.66 ms
GPTQ-W4 [7]	4-16-16	6858	36.50 ms	22.34 ms
AWQ-W3 [13]	3-16-16	7602	3170.51 ms	3334.66 ms
AWQ-W4 [13]	4-16-16	6802	34.46 ms	19.49 ms
APQ Flex	0-1-64[a]	56.78%	10700.53 ms	10864.57 ms
APQ Edge	0-4-1024[a]	73.32%	10700.53 ms	10864.57 ms
APQ Server	0-2-512[a]	63.43%	10700.53 ms	10864.57 ms

Table 4.4: Prefill and Decode latency of Mistral-7B on RTX 3090. The sample sequence length is 128.

under a range of cluster sizes and subspace dimensions with PyTorch backend. We found the latency is consistent for different configurations since PyTorch invokes the same kernel. Thus, we report the reconstruct version of APQ in Table 4.2, Table 4.3, and Table 4.4. As can be seen from the tables, INT8 and INT4 models utilize builtin INT4 hardware support which matches the latency compared with baseline. INT3 variants, however, do not have this advantage so their latency degrade significantly. This is more evident in AWQ-v3 as there is no custom kernel for INT3. This comprehensive evaluation enables us to assess the trade-offs between compression, accuracy, and efficiency, guiding the selection of optimal configurations for practical deployment.

4.2 Experimental Settings

4.2.1 Baselines

Before presenting our experimental setup and results, we briefly recap the quantization methods evaluated in this work. GPTQ [7] is a post-training quantization technique that minimizes quantization error using second-order information, enabling accurate INT4/INT3

quantization without retraining. AWQ [13] focuses on activation-aware weight quantization, selectively preserving sensitive weights in higher precision to improve accuracy. SmoothQuant [23] introduces a smoothing transformation to activations prior to quantization, enhancing post-training quantization performance. SpinQuant [14] applies an orthogonal rotation to weight matrices before quantization, redistributing variance to reduce quantization error. These methods represent the current state-of-the-art in LLM quantization and serve as strong baselines for comparison with our proposed approach.

4.2.2 LLM Workloads

We experiment on the LLaMA-2 [22] models (7B, 13B), Qwen-2.5 [21, 24, 25] models (7B), and Mistral [11] 7B model. We evaluate these models on the WikiText2 test set [15] and eight zero-shot commonsense reasoning tasks. The selected benchmarks include BoolQ [5], PIQA [2], SIQA [19], OBQA [16], HellaSwag [26], WinoGrande [18], ARC-easy, and ARC-challenge [6].

4.2.3 Configurations

Based on the design space exploration results reported in section 4.1, we select two representative APQ configurations, namely APQ Edge and APQ Server, that strike a balance between compression rate, model performance, and end-to-end latency. We also picked APQ Flex as one APQ configuration to demonstrate APQ's flexibility over the design space. The full specification of these configurations is provided in Table 4.5.

Table 4.5: Selected APQ configurations.

Configuration	Edge	Server	Flex
Number of clusters	1024	512	64
Size of clusters	4	2	1

4.3 Methodology

We reproduced the quantized models of GPTQ[1], AWQ[2], and SmoothQuant[3] using their official implementations. We obtained model performance on all selected tasks using the quantizated models with lm-evaluation-harness [8].

To measure compression rates, we focus exclusively on the memory required to store model weights, excluding input activations and key-value caches.

For GPTQ, both INT4 and INT3 weights are packed by default into INT32 tensors, and we report their actual model sizes as stored.

AWQ also packed INT4 weights into INT32 tensors using its kernels. Since AWQ does not provide official INT3 kernel support, we estimate the memory footprint of AWQ INT3 models by applying a 25% reduction to the memory used by the corresponding INT4 weights. This reduction is applied only to the linear layers, while other components such as LayerNorm and embedding layers are left unchanged in the AWQ INT4 version.

For SmoothQuant, we hardcoded element size (1 byte for 8-bit quantization and 2 bytes for scaling factors in FP16) since there is no data packing provided.

To measure the end-to-end latency, we used dummy inputs with dummy key-value cache to measure both the prefill and decode latency.

We include scripts in our codebase that we use to run above experiments.

Table 4.6: Compress ratio (CR) and perplexity results on Wikitext2 testset (Wiki).

Method	**# Bits**	**LLaMA2-7B**		**Qwen2.5-7B**		**Mistral-7B**		**Average**	
	W-A-KV	CR (↑)	Wiki (↓)	CR (↑)	Wiki (↓)	CR (↑)	Wiki (↓)	CR (↑)	Wiki (↓)
Baseline	16-16-16	0%	5.47	0%	6.85	0%	5.32	0%	5.88
SmoothQuant [23]	8-8-16	48.03%	5.52	42.82%	7.26	48.13%	5.35	46.33%	6.04
GPTQ-W3 [7]	3-16-16	77.16%	6.77	68.79%	8.45	77.31%	10.48	74.42%	8.57
GPTQ-W4 [7]	4-16-16	71.11%	5.77	63.40%	7.17	71.24%	8.18	68.58%	7.04
AWQ-W3 [13]	3-16-16	77.93%	6.24	72.08%	8.80	78.04%	6.06	76.02%	7.03
AWQ-W4 [13]	4-16-16	70.57%	5.60	62.78%	7.09	70.71%	5.44	68.02%	6.04
APQ Flex	0-1-64[a]	58.85%	5.65	52.44%	7.27	59.05%	6.62	56.78%	6.51
APQ Edge	0-4-1024[a]	75.94%	6.04	67.63%	7.94	76.40%	5.88	73.32	6.62
APQ Server	0-2-512[a]	63.95%	5.55	61.95%	7.05	64.38%	5.48	63.43	6.02

[a]dim-SZss-Ks

4.4 Results

We highlight model performance, compression rate, and inference latency results to discuss the benefits of APQ.

4.4.1 Model Performance with Compression Rate Tradeoff

APQ model performance on WikiText2 testset with compression rate is shown in Table 4.6. APQ produces similar perplexity scores to SmoothQuant INT8 version while maintaining a compression rate close to INT4 methods.

Similar results in 0-shot commonsense tasks, where we observe that APQ achieves comparable performance to the plain FP16 models and state-of-the-art quantization schemes. This indicates that APQ effectively maintains the model's language modeling capabilities while significantly reducing its size. Results on LLaMA2-7B is shown in Table 4.7, results on Qwen-2.5-7B is shown in Table 4.8, and results on Mistral-7B is shown in Table 4.9. We find consistency in both perplexity-based language modeling tasks and accuracy-based tasks across these models.

[1]https://github.com/ist-daslab/gptq
[2]https://github.com/mit-han-lab/llm-awq
[3]https://github.com/mit-han-lab/smoothquant

Table 4.7: Results of 8 0-shot tasks on LLaMA2-7B model.

Method	**# Bits** W-A-KV	**CR**	**BoolQ**	**PIQA**	**SIQA**	**HS**	**WG**	**ARC-e**	**ARC-c**	**OBQA**	**Average**
Baseline	16-16-16	0%	77.70%	79.05%	46.11%	76.00%	68.90%	74.49%	46.16%	44.20%	64.07%
SmoothQuant [23]	8-8-16	48.03%	76.88%	79.10%	45.85%	75.95%	68.11%	73.82%	45.13%	44.40%	63.65%
GPTQ-W3 [7]	3-16-16	77.16%	70.40%	76.55%	43.04%	72.89%	64.48%	67.00%	39.68%	40.80%	59.35%
GPTQ-W4 [7]	4-16-16	71.11%	77.24%	78.02%	46.93%	75.49%	69.22%	72.85%	44.79%	43.60%	63.51%
AWQ-W3 [13]	3-16-16	77.93%	68.37%	78.07%	44.06%	73.43%	67.48%	70.83%	43.08%	42.00%	60.91%
AWQ-W4 [13]	4-16-16	70.57%	78.68%	79.05%	46.11%	75.16%	68.27%	73.94%	45.90%	43.60%	63.83%
APQ Flex	0-8-2048[a]	58.85%	76.51%	78.51%	45.04%	74.72%	69.53%	73.44%	44.45%	43.80%	63.25.%
APQ Edge	0-4-1024[a]	75.94%	75.75%	77.42%	44.11%	72.81%	67.88%	69.82%	42.92%	41.60%	61.54%
APQ Server	0-2-512[a]	63.95%	78.38%	79.00%	46.21%	75.19%	68.35%	74.32%	45.05%	43.60%	63.76%

[a]dim-SZss-Ks

Table 4.8: Results of 8 0-shot tasks on Qwen-2.5-7B model.

Method	**# Bits** W-A-KV	**CR**	**BoolQ**	**PIQA**	**SIQA**	**HS**	**WG**	**ARC-e**	**ARC-c**	**OBQA**	**Average**
Baseline	16-16-16	0%	84.74%	79.81%	54.70%	78.88%	73.24%	77.56%	51.02%	47.20%	68.39%
SmoothQuant [23]	8-8-16	48.03%	84.37%	79.92%	54.96%	78.48%	70.48%	75.67%	50.08%	45.40%	67.42%
GPTQ-W3 [7]	3-16-16	68.79%	78.86%	78.67%	48.77%	75.00%	66.85%	77.44%	50.51%	43.00%	64.88%
GPTQ-W4 [7]	4-16-16	63.40%	83.39%	79.97%	54.55%	78.22%	72.37%	74.78%	51.36%	45.00%	67.45%
AWQ-W3 [13]	3-16-16	72.08%	80.91%	78.94%	47.95%	74.97%	67.48%	72.01%	48.37%	44.00%	64.32%
AWQ-W4 [13]	4-16-16	62.78%	83.54%	79.16%	53.73%	78.34%	72.13%	76.68%	51.27%	45.80%	67.58%
APQ Flex	0-1-64[a]	52.44%	82.08%	79.87%	52.66%	78.66%	71.67%	77.36%	52.47%	46.20%	67.62%
APQ Edge	0-4-1024[a]	67.63%	80.89%	78.40%	51.23%	76.09%	69.22%	73.70%	48.04%	44.20%	65.22%
APQ Server	0-2-512[a]	61.95%	84.92%	80.14%	53.22%	78.37%	72.06%	77.36%	51.62%	46.40%	68.01%

[a]dim-SZss-Ks

Moreover, APQ offers finer-grained control over the quantization design space compared to conventional methods. By varying its configuration parameters, users can achieve a wide range of compression rates, enabling more precise trade-offs between model size and performance. This flexibility allows APQ to reach intermediate compression points that are not accessible with standard 8-bit, 4-bit or 3-bit quantization, providing better options for deployment scenarios that require a balance between efficiency and accuracy. For example, APQ Flex achieves comparable performance for targeted models while providing a larger compression rate compared with INT8, demonstrating its ability to fulfill diverse deployment requirements.

Table 4.9: Results of 8 0-shot tasks on Mistral-7B-v0.3 model.

Method	**# Bits** W-A-KV	**CR**	**BoolQ**	**PIQA**	**SIQA**	**HS**	**WG**	**ARC-e**	**ARC-c**	**OBQA**	**Average**
Baseline	16-16-16	0%	82.11%	82.26%	45.95%	80.38%	73.40%	78.36%	52.30%	44.00%	67.34%
SmoothQuant [23]	8-8-16	48.03%	81.83%	81.93%	45.64%	80.19%	74.42%	78.24%	52.13%	44.80%	67.39%
GPTQ-W3 [7]	3-16-16	77.31%	76.97%	79.37%	44.62%	75.84%	63.85%	70.11%	45.30%	34.80%	61.35%
GPTQ-W4 [7]	4-16-16	71.24%	78.96%	81.06%	46.11%	78.23%	71.03%	76.51%	49.48%	40.40%	65.22%
AWQ-W3 [13]	3-16-16	78.04%	76.63%	79.37%	44.77%	77.83%	69.37%	73.86%	47.78%	40.60%	63.77%
AWQ-W4 [13]	4-16-16	70.71%	80.03%	81.61%	45.18%	79.78%	71.58%	77.35%	50.51%	43.20%	66.15%
APQ Flex	0-1-64[a]	59.05%	78.29%	80.69%	44.01%	73.39%	70.64%	75.34%	48.63%	45.00%	64.50%
APQ Edge	0-4-1024[a]	76.40%	80.80%	80.58%	45.65%	78.06%	72.69%	76.05%	48.46%	43.00%	65.66%
APQ Server	0-2-512[a]	64.38%	80.92%	81.34%	45.60%	79.01%	73.48%	77.69%	51.70%	45.20%	66.87%

[a]dim-SZss-Ks

4.4.2 Scalability

Table 4.10 demonstrates APQ's strong scalability across different model sizes. We observe that the compression rate and model performance achieved by APQ remain consistent as we scale from smaller to larger LLMs, such as from LLaMA2-7B to LLaMA2-13B. This indicates that the benefits of APQ are not limited to a specific model size, but generalize well to larger architectures. Furthermore, the ability to maintain similar perplexity and accuracy metrics across models of varying scales highlights the robustness of APQ's quantization strategy. This scalability makes APQ a practical solution for both resource-constrained edge deployments and large-scale server environments, ensuring efficient compression and inference performance regardless of the underlying model size.

Table 4.10: Scalability of APQ with LLaMA models

Model		CR	Wiki	BQ	PIQA	SIQA	HS	WG	ARC-e	ARC-c	OBQA	Average
LLaMA2-7B	Baseline	0%	5.47	77.70%	79.05%	46.11%	76.00%	68.90%	74.49%	46.16%	44.20%	64.07%
	APQ Flex	58.85%	5.65	76.51%	78.51%	45.04%	74.72%	69.53%	73.44%	44.45%	43.80%	63.25.%
	APQ Edge	75.94%	6.04	75.75%	77.42%	44.11%	72.81%	67.88%	69.82%	42.92%	41.60%	61.54%
	APQ Server	63.95%	5.55	78.38%	79.00%	46.21%	75.19%	68.35%	74.32%	45.05%	43.60%	63.76%
LLaMA2-13B	Baseline	0%	4.88	80.55%	80.52%	47.38%	79.37%	72.37%	77.52%	49.23%	45.20%	66.51%
	APQ Flex	59.96%	5.05	79.32%	79.92%	45.49%	77.61%	71.27%	77.56%	50.68%	44.60%	65.61%
	APQ Edge	78.08%	5.36	78.87%	79.00%	45.45%	76.56%	71.98%	75.80%	48.12%	44.20%	65.00%
	APQ Server	65.91%	4.94	79.97%	80.25%	46.88%	78.91%	71.82%	77.90%	50.00%	45.00%	66.34%

Chapter 5

Conclusion and Future Work

This thesis introduces APQ, a novel quantization method that enables higher-dimensional quantization with flexibility in the design space. APQ achieves model performance and compression rates comparable to existing methods, while providing enhanced flexibility for diverse deployment scenarios. Through comprehensive design space exploration, we demonstrate the potential of APQ as an effective quantization approach for large language models.

The current design space exploration result, however, can benefit from further optimization. Due to time constraints, only the reconstruction-based APQ inference was implemented in PyTorch, which is suboptimal in terms of latency. We plan to include a custom GPU kernel for APQ inference as discussed in Section 3.1.2, enabling a thorough analysis of the latency design space, where we can achieve a better trade-off between latency and model performance. A custom GPU kernel is beneficial as GPU provides high bandwidth which favors massive parallel lookup table construction and lookups. Future work will also investigate custom hardware solutions to further accelerate inference, leveraging dynamic lookup table construction and buffer size optimization specific to APQ.

Building on custom GPU kernels and hardware, we also aim to further expand the design

space of APQ by introducing flexibility across layers. In the current APQ setting, all linear layers share identical quantization parameters, which may not be optimal since different layers can have varying weight distributions and tolerance to quantization. As future work, we plan to explore assigning different cluster numbers and subspace sizes to different layers, enabling more adaptive and effective quantization.

We also plan to include a post-training finetune step to further improve the performance under high compression rate settings.

Bibliography

[1] A. Abouelhamayed, A. Cui, J. Fernandez-Marques, N. Lane, and M. Abdelfattah. Pqa: Exploring the potential of product quantization in dnn hardware acceleration. *ACM Transactions on Reconfigurable Technology and Systems*, 18(1):1–29, 2024.

[2] Y. Bisk, R. Zellers, J. Gao, Y. Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 7432–7439, 2020.

[3] D. Blalock and J. Guttag. Multiplying matrices without multiplying. In *International Conference on Machine Learning*, pages 992–1004. PMLR, 2021.

[4] T. Chen, L. Li, and Y. Sun. Differentiable product quantization for end-to-end embedding compression. In H. D. III and A. Singh, editors, *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pages 1617–1626. PMLR, 13–18 Jul 2020.

[5] C. Clark, K. Lee, M.-W. Chang, T. Kwiatkowski, M. Collins, and K. Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.

[6] P. Clark, I. Cowhey, O. Etzioni, T. Khot, A. Sabharwal, C. Schoenick, and O. Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.

[7] E. Frantar, S. Ashkboos, T. Hoefler, and D. Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[8] L. Gao, J. Tow, B. Abbasi, S. Biderman, S. Black, A. DiPofi, C. Foster, L. Golding, J. Hsu, A. Le Noac'h, H. Li, K. McDonell, N. Muennighoff, C. Ociepa, J. Phang, L. Reynolds, H. Schoelkopf, A. Skowron, L. Sutawika, E. Tang, A. Thite, B. Wang, K. Wang, and A. Zou. The language model evaluation harness, 07 2024.

[9] R. Gray. Vector quantization. *IEEE Assp Magazine*, 1(2):4–29, 1984.

[10] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence*, 33(1):117–128, 2010.

[11] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, et al. Mistral 7b. arxiv. *arXiv preprint arXiv:2310.06825*, 10, 2023.

[12] G. Li, S. Ye, C. Chen, Y. Wang, F. Yang, T. Cao, C. Liu, M. M. S. Aly, and M. Yang. Lut-dla: Lookup table as efficient extreme low-bit deep learning accelerator. In *2025 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 671–684. IEEE, 2025.

[13] J. Lin, J. Tang, H. Tang, S. Yang, W.-M. Chen, W.-C. Wang, G. Xiao, X. Dang, C. Gan, and S. Han. Awq: Activation-aware weight quantization for llm compression and acceleration. In *MLSys*, 2024.

[14] Z. Liu, C. Zhao, I. Fedorov, B. Soran, D. Choudhary, R. Krishnamoorthi, V. Chandra, Y. Tian, and T. Blankevoort. Spinquant: Llm quantization with learned rotations. *arXiv preprint arXiv:2405.16406*, 2024.

[15] S. Merity, C. Xiong, J. Bradbury, and R. Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[16] T. Mihaylov, P. Clark, T. Khot, and A. Sabharwal. Can a suit of armor conduct electricity? a new dataset for open book question answering. *arXiv preprint arXiv:1809.02789*, 2018.

[17] J. Ran, R. Lin, J. C. L. Li, J. Zhou, and N. Wong. Pecan: A product-quantized content addressable memory network. In *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6. IEEE, 2023.

[18] K. Sakaguchi, R. L. Bras, C. Bhagavatula, and Y. Choi. Winogrande: An adversarial winograd schema challenge at scale. *Communications of the ACM*, 64(9):99–106, 2021.

[19] M. Sap, H. Rashkin, D. Chen, R. LeBras, and Y. Choi. Socialiqa: Commonsense reasoning about social interactions. *arXiv preprint arXiv:1904.09728*, 2019.

[20] X. Tang, Y. Wang, T. Cao, L. L. Zhang, Q. Chen, D. Cai, Y. Liu, and M. Yang. Lut-nn: Empower efficient neural network inference with centroid learning and table lookup. In *Proceedings of the 29th Annual International Conference on Mobile Computing and Networking*, pages 1–15, 2023.

[21] Q. Team. Qwen2.5: A party of foundation models, September 2024.

[22] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[23] G. Xiao, J. Lin, M. Seznec, H. Wu, J. Demouth, and S. Han. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*, pages 38087–38099. PMLR, 2023.

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

[25] A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang, H. Wei, et al. Qwen2. 5 technical report. *arXiv preprint arXiv:2412.15115*, 2024.

[26] R. Zellers, A. Holtzman, Y. Bisk, A. Farhadi, and Y. Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.