RaanA: A Fast, Flexible, and Data-Efficient Post-Training Quantization Algorithm

Anonymous Author(s)
Affiliation
Address
email

Abstract

Post-training Quantization (PTQ) has become a widely used technique for improving inference efficiency of large language models (LLMs). However, existing PTQ methods generally suffer from crucial limitations such as heavy calibration data requirements and inflexible choice of target number of bits. In this paper, we propose RaanA, a unified PTQ framework that overcomes these challenges by introducing two novel components: 1) RaBitQ-H, a variant of a randomized vector quantization method RaBitQ, designed for fast, accurate, and highly efficient quantization; and 2) AllocateBits, an algorithm that optimally allocates bit-widths across layers based on their quantization sensitivity. RaanA achieves competitive performance with state-of-the-art quantization methods while being extremely fast, requiring minimal calibration data, and enabling flexible bit allocation. Extensive experiments demonstrate RaanA's efficacy in balancing efficiency and accuracy.

1 Introduction

The rapid advancement in deep learning has demonstrated that increasing the size of neural networks (NNs) often leads to significant improvements in performance across various tasks [20, 7, 1, 15]. Large language models (LLMs), such as GPT [8, 1] and LLaMA [32, 15], have exhibited remarkable capabilities, but their deployment remains a challenge due to substantial computational and memory requirements. Among the various strategies developed to improve LLM inference, Post-Training Quantization (PTQ) has emerged as a widely adopted approach for optimizing the memory usage and inference speed of LLMs, balancing between accuracy and efficiency [10, 11, 25, 16, 23, 9]. The essential idea of PTQ is to reduce the numerical precision of the parameters of a trained model, replacing high precision floating-point representations with lower precision alternatives, which in turn reduces memory consumption and alleviates memory bandwidth constraints.

Optimal Brain Quantizer (OBQ) is one of the earlier influential methods in PTQ. It formulates the quantization problem as an optimization task, where the core objective is to minimize the layer-wise quantization error $\widehat{\boldsymbol{W}}^{(\ell)*} = \arg\min_{\widehat{\boldsymbol{W}}\in\mathcal{C}} \left\| \boldsymbol{X}^{(\ell)}\boldsymbol{W}^{(\ell)} - \boldsymbol{X}^{(\ell)}\widehat{\boldsymbol{W}} \right\|_{\mathcal{F}}^2$, where $\boldsymbol{X}^{(\ell)}$, $\boldsymbol{W}^{(\ell)}$ and $\widehat{\boldsymbol{W}}$ are the input feature, original weight matrix and quantized weight matrix of layer ℓ (a linear transformation layer) respectively. This approach has led to significant improvements in quantization accuracy by optimizing the quantization parameters (rescale, zero-point etc.) according to layer-wise behavior approximation. OBQ has become a foundation of PTQ research, with many recent methods building upon its framework [11, 23, 25, 9]. Despite their effectiveness, existing approaches based on the OBQ framework typically suffer from notable limitations:

- First, the OBQ framework only controls the error of the output of each layer and does not account for the overall error in the model output; more importantly, it treats all layers identically, whereas in practice, their importance can vary: the error in the first layer propagates through all subsequent layers, while that in the last layer affects only itself. Omitting this hierarchy in error sensitivity can lead to suboptimal quantization strategies.

- Second, the optimal solution of the OBQ framework highly depends on the input $\boldsymbol{X}^{(\ell)}$ which, however, is not constant across batches, and as a result, all the quantization methods based on this framework require a lot of calibration data to accurately estimate the input features (more precisely, the so-called layer-wise Hessian $\boldsymbol{X}^{(\ell)\top}\boldsymbol{X}^{(\ell)}$) [17]. This high requirement of data not only limits the efficiency of quantization, but also the generalization performance of the method, as it can perform worse if the data distribution is far from its calibration data distribution.

- Moreover, the OBQ framework directly replaces each weight matrix with a quantized weight matrix. Since we are only concerned with the output of a layer rather than its internal structure, there is potential for greater flexibility by allowing additional operations within a layer.

The above-mentioned limitations are not exclusive to OBQ-based methods, but also shared by many other PTQ methods. For example, Norm-Tweaking [24], although only a plugin for adjusting LayerNorm parameters, also highly relies on calibration data, and EasyQuant [31], although it does not require calibration, only controls layer-wise error and is limited to entry-wise float-point rounding. In this paper, we propose a novel PTQ framework that overcomes the above issue, based on the following ideas.

RaBitQ-H: Adopting Efficient Vector Quantization Methods to LLM Quantization If we don't stick to preserving the structure of the original matrix multiplication and embrace a broader class of operations, it is possible to bring in much stronger tools from vector quantization. Specifically, in this paper we adopt RaBitQ [13, 12], a state-of-the-art vector quantization algorithm that enjoys very fast computation and well-controlled error-bound. However, RaBitQ is originally designed to solve approximate nearest neighbor search (ANNS) and can not be directly adopted to LLM quantization (see the detailed explanation in Appendix C). To address this issue, in this paper we propose RaBitQ-H, a variant of RaBitQ, to address the challenges of adopting RaBitQ to LLM quantization.

AllocateBits: Allocating Bits Unevenly Across Layers As mentioned before, it is suboptimal to uniformly assign computational resources across all layers, as some layers can be more important than others. There has been a research direction called Mixed Precision Quantization (MPQ), which allows different numbers of bits used in each layer [28, 16, 6]. However, existing MPQ methods are mostly based on heuristics and are typically limited to binary choices of bit-widths (e.g. 4-bits v.s. 8-bits), making them not sufficiently flexible. In this paper, we propose a simple and effective approach to estimate the importance of each layer and formulate the bit allocation task as an integer programming problem that allows arbitrary bit-width choices. By solving this integer program, we obtain an optimal bit allocation strategy across layers.

In this paper, we propose **RaanA**: a novel PTQ framework combining **RaBitQ-H and AllocateBits**. RaanA enjoys the following superiority:

1. Extremely Fast and Device-Independent: Unlike other state-of-the-art PTQ methods that generally require hours or even days to perform on large models, RaanA generally only requires tens of minutes on 70b models. Additionally, RaanA is largely device-independent, with most of its computation can be efficiently performed on CPUs, reducing its reliance on GPU devices.

2. Minimal Calibration Required: RaanA requires only a few or even zero samples for calibration, as opposed to most existing methods that typically require a huge amount of data.

3. Strong Performance: RaanA performs comparably to state-of-the-art PTQ methods. Notably, it remains effective even at extremely low bit-widths (e.g., < 3 average bits), a regime where many other methods struggle.

4. High Flexibility: RaanA allows any choice of target average number of bits, enabling more flexible and and fine-grained quantization choices.

Paper Structure Due to space limitation, here in the main paper we only outline the overall framework and briefly introduce the AllocateBits and RabitQ-H algorithm, deferring a more formal introduction to appendix. In Appendix A we introduce some background on the methods we use and related work. Section 2 and Appendices B and C describe RaanA in detail: Section 2.1 presents the preliminaries and overall framework; Appendix B explains how to estimate the sensitivity of each layer and solve the bit allocation problem; and Appendix C introduces the RaBitQ-H algorithm. In Appendix D, we present our experimental results. Finally, in Appendix E, we conclude the paper and discuss potential future directions.

2 Preliminaries

Throughout this paper, we use bold lowercase letters to represent vectors (e.g. $\boldsymbol{x}$) and bold uppercase letters to represent matrices (e.g. $\boldsymbol{A}$), and we use the corresponding unbold letters with subscript to represent the entries of vectors or matrices (e.g. x_i is the i-th entry of vector $\boldsymbol{x}$).

In this paper, we fix a trained neural network $f : \mathbb{R}^{n\times d_0} \times \mathbb{R}^m \to \mathbb{R}$ as the object of study, where d_0 is the dimensionality of the input and m is the number of learnable parameters[1]. We use $\boldsymbol{x} \in \mathbb{R}^{n\times d_0}$ and $\boldsymbol{\theta}$ to denote the input data and the collection of all parameters of f, respectively, where n is the token number[2].

A NN model typically consists of multiple linear transformation blocks where the input is multiplied by a weight matrix. These include, for example, feed-forward layers as well as the Q, K, and V transformation layers in Transformer architectures [35]. We assume the model contains a total of L linear layers and denote the input features, parameter matrix, and output features of the k-th linear layer by $\boldsymbol{X}^{(k)} \in \mathbb{R}^{n\times d_k}$, $\boldsymbol{W}^{(k)} \in \mathbb{R}^{d_k\times c_k}$, and $\boldsymbol{H}^{(k)} = \boldsymbol{X}^{(k)}\boldsymbol{W}^{(k)} \in \mathbb{R}^{n\times c_k}$, respectively, where d_k and c_k are the input and output dimensions of the k-th linear layer. Note that $\boldsymbol{X}^{(k)}$ is principally a function of $\boldsymbol{x}$, and $\boldsymbol{W}^{(k)}$ is a function of $\boldsymbol{\theta}$, but we omit the arguments $\boldsymbol{x}$ and $\boldsymbol{\theta}$ when they are clear from context.

We use $\boldsymbol{x}_i^{(k)} \in \mathbb{R}^d$ to represent the i-th row of $\boldsymbol{X}^{(k)}$ and $\boldsymbol{w}^{(k)}i \in \mathbb{R}^d$ to represent the i-th column of $\boldsymbol{W}^{(k)}$. The (i,j)-th entry of $\boldsymbol{H}^{(k)}$ is given by $h^{(k)}i,j = \left\langle \boldsymbol{x}_i^{(k)}, \boldsymbol{w}_j^{(k)} \right\rangle$.

2.1 Overall Framework

We first present the overall framework of RaanA in Algorithm 1. It takes a trained model, calibration data and desired quantization parameters as input and output quantized weight matrix together with other information used for de-quantization (see Algorithm 3 for the de-quantization algorithm). The algorithm consists of two parts: determining each layer's target bit-width, and performing quantization. In Algorithm 1 we use the error estimator Err_k and vector quantization algorithm $\mathsf{RaBitQ\text{-}H}$ as black boxes, and they will be specified in the subsequent sections.

3 A Brief Introduction to the Main Algorithms

This section summarizes the two main components of our approach: (1) the bits-allocation strategy (`AllocateBits`) and (2) the quantization algorithm RaBitQ-H. We provide an intuitive description while omitting most technical details for brevity. See Appendices B and C for the details of the algorithms.

3.1 AllocateBits: Error Estimation and Bits Allocation

When quantizing the k-th layer with b bits, the induced error on model output is denoted by $\mathsf{Err}_k(b, \boldsymbol{x})$. Under mild assumptions (see [12]), this error is bounded by$\mathsf{Err}_k(b, \boldsymbol{x}) \lesssim \alpha_k 2^{-b}$, where α_k depends on the Jacobian of f w.r.t. $\boldsymbol{H}^{(k)}$, as well as norms of input and weight matrices. Thus,

[1] A neural network with multi-dimensional output can be viewed as several neural networks with scalar output, and therefore we only consider NNs with scalar output.

[2] n can also be understood as the batch size, since we only consider linear layers that do not involve any cross-token interaction.

Algorithm 1: Overall Framework of RaanA

Input: trained model parameters $\boldsymbol{\theta}_0$, bit-width candidate set $\mathscr{B}$, overall bits budget R and calibration data $\{\boldsymbol{x}_i\}_{i=1}^{n_c}$;

`/* AllocateBits */`

Solve the bits-allocation problem

$$\begin{aligned} \{b_k^*\}_{k=1}^{L} = \arg\min_{\{b_k\}_{k=1}^{L}} \frac{1}{n_c} \sum_{k=1}^{L} \sum_{i=1}^{n_c} \mathsf{Err}_k\left(b_k; \boldsymbol{x}_i\right) \\ \text{s.t.} \quad \sum_{k=1}^{L} b_k m_k \le R \text{ and } b_k \in \mathscr{B}, \forall k \in [L]; \end{aligned} \tag{1}$$

`/* Quantization */`

Calculate $\left(\widehat{\boldsymbol{W}}^{(k)}, \boldsymbol{r}^{(k)}, \boldsymbol{D}^{(k)}\right) = \mathsf{RaBitQ\text{-}H}\left(\boldsymbol{W}^{(k)}, b_k^*\right)$ for $k = 1, 2, \cdots L$;

Return: $\left\{\left(\widehat{\boldsymbol{W}}^{(k)}, \boldsymbol{r}^{(k)}, \boldsymbol{D}^{(k)}\right)\right\}_{k=1}^{L}$.

Algorithm 1: The overall framework of RaanA. $\mathsf{Err}_k(b; \boldsymbol{x})$ is an estimation of the overall error brought by the k-th linear layer with b-bit quantization, estimated at point $\boldsymbol{x}$, and $\mathsf{RaBitQ\text{-}H}$ is the RaBitQ-H quantization algorithm. $m_k = d_k c_k$ is the number of parameters at layer k; n_c is the number of calibration data; $\mathscr{B}$ is a set containing all desired choice of layer-wise bit-width (e.g. $\mathscr{B} = \{1, 2, \cdots 8\}$), and R is the desired total number of bits used (i.e. bits per parameter times total number of weight parameters).

minimizing the overall quantization error reduces to

$$\min_{\{b_k\}} \sum_{k=1}^{L} \alpha_k 2^{-b_k}, \quad \text{s.t.} \sum_{k=1}^{L} b_k m_k \le R,\ b_k \in \mathscr{B}. \tag{2}$$

Although this is an NP-hard integer program, the problem size is moderate in practice. A divide-by-GCD trick greatly reduces the budget size, making dynamic programming feasible. The resulting algorithm runs in $O(L|\mathscr{B}|R/g)$ time, completing within seconds even on CPU.

Calibration of α_k is efficient: unlike Hessian-based approaches, it only depends on input norms and Jacobians, which are stable and require very few samples. In practice, **few-shot calibration** uses as few as 5 training samples, while **zero-shot calibration** relies on a single synthetic sentence, yet still yields accurate estimates.

3.2 RaBitQ-H: Quantization with Randomized Hadamard Transformation

RaBitQ [12] is a universal multi-bit quantization algorithm that preserves inner products with controlled error. While directly applicable to large models, its original design requires costly random rotations of $O(d^2)$ per vector. To address this, RaBitQ-H replaces random rotation with a **Randomized Hadamard Transformation (RHT)**, which (i) needs only d_k random bits, (ii) runs in $O((c_k + n) \log d_k)$ time via fast Hadamard kernels, and (iii) retains the same error guarantees.

The quantization pipeline is as follows:

- **Preprocessing:** Apply RHT to weight matrix and quantize using RaBitQ, storing scaling factors.
- **Inference:** Apply RHT to input, perform matrix multiplication with quantized weights, and rescale the result.

Implementation requires handling non-power-of-two dimensions and applying simple preprocessing (e.g., input centralization), which further stabilizes error without altering theoretical guarantees.

Summary. `AllocateBits` provides a principled way to assign bits across layers under a global budget, while RaBitQ-H ensures efficient and accurate quantization with RHT. Together, they enable fast and memory-efficient quantization for large language models with minimal calibration data.

References

[1] Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. *arXiv preprint arXiv:2303.08774*, 2023.

[2] Krish Agarwal, Rishi Astra, Adnan Hoque, Mudhakar Srivatsa, Raghu Ganti, Less Wright, and Sijia Chen. Hadacore: Tensor core accelerated hadamard transform kernel. *arXiv preprint arXiv:2412.08832*, 2024.

[3] Cecilia Aguerrebere, Ishwar Singh Bhati, Mark Hildebrand, Mariano Tepper, and Theodore Willke. Similarity search in the blink of an eye with compressed indices. *Proc. VLDB Endow.*, 16(11):3433–3446, July 2023.

[4] Noga Alon and Bo'az Klartag. Optimal compression of approximate inner products and dimension reduction. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 639–650, 2017.

[5] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. *Advances in Neural Information Processing Systems*, 37:100213–100240, 2024.

[6] Alireza Behtash, Marijan Fofonjka, Ethan Baird, Tyler Mauer, Hossein Moghimifam, David Stout, and Joel Dennison. Universality of layer-level entropy-weighted quantization beyond model architecture and size. *arXiv preprint arXiv:2503.04704*, 2025.

[7] James Betker, Gabriel Goh, Li Jing, Tim Brooks, Jianfeng Wang, Linjie Li, Long Ouyang, Juntang Zhuang, Joyce Lee, Yufei Guo, et al. Improving image generation with better captions. *Computer Science. https://cdn. openai. com/papers/dall-e-3. pdf*, 2(3):8, 2023.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

[9] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher M De Sa. Quip: 2-bit quantization of large language models with guarantees. *Advances in Neural Information Processing Systems*, 36:4396–4429, 2023.

[10] Elias Frantar and Dan Alistarh. Optimal brain compression: A framework for accurate post-training quantization and pruning. *Advances in Neural Information Processing Systems*, 35:4475–4488, 2022.

[11] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan-Adrian Alistarh. Optq: Accurate post-training quantization for generative pre-trained transformers. In *11th International Conference on Learning Representations*, 2023.

[12] Jianyang Gao, Yutong Gou, Yuexuan Xu, Yongyi Yang, Cheng Long, and Raymond Chi-Wing Wong. Practical and asymptotically optimal quantization of high-dimensional vectors in euclidean space for approximate nearest neighbor search. *arXiv preprint arXiv:2409.09913*, 2024.

[13] Jianyang Gao and Cheng Long. Rabitq: quantizing high-dimensional vectors with a theoretical error bound for approximate nearest neighbor search. *Proceedings of the ACM on Management of Data*, 2(3):1–27, 2024.

[14] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. Optimized product quantization for approximate nearest neighbor search. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2946–2953, 2013.

[15] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.

[16] Ziyi Guan, Hantao Huang, Yupeng Su, Hong Huang, Ngai Wong, and Hao Yu. Aptq: Attention-aware post-training mixed-precision quantization for large language models. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*, pages 1–6, 2024.

[17] Babak Hassibi, David G Stork, and Gregory J Wolff. Optimal brain surgeon and general network pruning. In *IEEE international conference on neural networks*, pages 293–299. IEEE, 1993.

[18] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, January 2011.

[19] William B Johnson and Joram Lindenstrauss. Extensions of lipschitz mappings into a hilbert space 26. *Contemporary mathematics*, 26:28, 1984.

[20] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

[21] Richard M Karp. *Reducibility among combinatorial problems*. Springer, 2010.

[22] Grigory Khromov and Sidak Pal Singh. Some fundamental aspects about lipschitz continuity of neural networks. *arXiv preprint arXiv:2302.10886*, 2023.

[23] Changhun Lee, Jungyu Jin, Taesu Kim, Hyungjun Kim, and Eunhyeok Park. Owq: Outlier-aware weight quantization for efficient fine-tuning and inference of large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 13355–13364, 2024.

[24] Liang Li, Qingyuan Li, Bo Zhang, and Xiangxiang Chu. Norm tweaking: High-performance low-bit quantization of large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 18536–18544, 2024.

[25] Ji Lin, Jiaming Tang, Haotian Tang, Shang Yang, Wei-Ming Chen, Wei-Chen Wang, Guangxuan Xiao, Xingyu Dang, Chuang Gan, and Song Han. Awq: Activation-aware weight quantization for on-device llm compression and acceleration. *Proceedings of Machine Learning and Systems*, 6:87–100, 2024.

[26] Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant: Llm quantization with learned rotations. *arXiv preprint arXiv:2405.16406*, 2024.

[27] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *arXiv preprint arXiv:1609.07843*, 2016.

[28] Nilesh Prasad Pandey, Markus Nagel, Mart van Baalen, Yin Huang, Chirag Patel, and Tijmen Blankevoort. A practical mixed precision algorithm for post-training quantization. *arXiv preprint arXiv:2302.05397*, 2023.

[29] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.

[30] Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. *arXiv preprint arXiv:2308.13137*, 2023.

[31] Hanlin Tang, Yifu Sun, Decheng Wu, Kai Liu, Jianchen Zhu, and Zhanhui Kang. Easyquant: An efficient data-free quantization algorithm for llms. *arXiv preprint arXiv:2403.02775*, 2024.

[32] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[33] Joel A Tropp. Improved analysis of the subsampled randomized hadamard transform. *Advances in Adaptive Data Analysis*, 3(01n02):115–126, 2011.

[34] Albert Tseng, Jerry Chee, Qingyao Sun, Volodymyr Kuleshov, and Christopher De Sa. Quip#: Even better llm quantization with hadamard incoherence and lattice codebooks. *arXiv preprint arXiv:2402.04396*, 2024.

[35] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

A Background and Related Work

In this section, we introduce some of the related works as well as the background of methods we use in this paper.

Weight Quantization in Post-training Quantization Among many methods that aims to make large language models more efficient, PTQ targets at improving the efficiency in inference-time. In this paper we especially focus on weight quantization. The problem of weight quantization in PTQ can be described as follows: given a pre-trained large language model, find another model who requires much less bits to store such that it approximates the behavior of the original model. OBQ [10] is one of the early works for PTQ. As mentioned in Section 1, many existing PTQ methods follows the framework of OBQ and is devoted to better or faster solve the layer-wise OBQ problem [11, 23, 25, 30, 9, 34]. As we mentioned in Section 1, most of these methods highly relies on performing float-point rounding and estimating the layer-wise Hessian.

Vector Quantization for Approximate Nearest Neighbor Search There have been a vast literature of vector quantization in ANNS. The classical scalar quantization (SQ) and its variant LVQ [3] take finite uniform grids as codebooks and independently round every coordinate to an unsigned integer. These methods support to compute inner product based on arithmetics of unsigned integers without decompression. However, they failed to achieve reasonable accuracy using low-bit quantization (< 4 bits). PQ [18] and OPQ [14] construct a codebook via KMeans clustering and quantizes vectors by mapping them to the nearest vector in the codebook. With clustering, these methods better capture the distributions of a dataset and provide better accuracy using low-bit quantization. However, their computation of inner product replies on looking up tables, which is costly in modern systems. RaBitQ [13] and its multi-bit extension [12] construct a codebook by translating, normalizing, and randomly rotating finite uniform grids. They achieve superior accuracy while enabling efficient computation. Theoretically, they provide an unbiased estimator with asymptotically optimal error bounds for the space-accuracy trade-off in inner product estimation between unit vectors. See Appendix F.2 for more details.

(Randomized) Hadamard Transformation In this paper, one critical technique we used is Randomized Hadamard Transformation (RHT), which is a randomized version of Hadamard Transformation and can be viewed as an approximation of random rotation [33]. Hadamard Transformation, in our context, is a specific class of orthonormal matrix whose matrix multiplication that can be fast computed, and is used in many quantization works [34, 5, 26, 2] and other areas. Due to space limitation, we defer a more detailed introduction of RHT to Appendix F.1.

B AllocateBits: Error Estimation and Bits Allocation

In this section, we discuss how to calculate $\mathsf{Err}_k(b, \boldsymbol{x})$ and how to solve the bits-allocation problem eq. (1). We begin by defining this quantity. In the model f, if we replace the parameters of the k-th layer, $\boldsymbol{W}^{(k)}$, with a b-bit quantized version, this leads to an error in the output $\boldsymbol{H}^{(k)}$. We denote this error by $\boldsymbol{\epsilon}^{(k)}(b) \in \mathbb{R}^{n\times c}$. Consequently, $\boldsymbol{\epsilon}^{(k)}(b)$ leads to an error in the overall model output f. We define $\mathsf{Err}_k\left(b, \boldsymbol{x}\right)$ as the absolute difference between the model outputs before and after quantization at layer k with b bits.

We first state a property of $\boldsymbol{\epsilon}^{(k)}(b)$ under RaBitQ-H, using Assumption B.1, which is justified by the analysis of RaBitQ (see Appendix F.2 for more details).

Assumption B.1. There exists a constant $K > 0$, such that for any $k \in [L], i \in [n], j \in [c_k]$, $\boldsymbol{\epsilon}^{(k)}(b) \in \mathbb{R}^{d_k}$ is a random vector such that

$$\boldsymbol{\epsilon}_{i,j}^{(k)}(b) \lesssim 2^{-b} \left\|\boldsymbol{x}_i^{(k)}\right\| \left\|\boldsymbol{w}_j^{(k)}\right\| \tag{3}$$

with high probability, where we use $\lesssim$ to hide constant terms.

When Assumption B.1 holds, Corollary B.2 is a direct corollary of Theorem G.2, which we prove in the appendix.

Corollary B.2 (Informal). *Fix the model input $\boldsymbol{x}$ and parameter $\boldsymbol{\theta}$, and suppose $\boldsymbol{\epsilon} = \boldsymbol{\epsilon}^{(k)}(b)$ satisfies Assumption B.1, then the following statement holds with probability at least 0.99:*

$$\mathsf{Err}_k\left(b, \boldsymbol{x}\right) \lesssim 2^{-b}\sqrt{\frac{\log c_k}{d_k}} \left\| \left.\frac{\partial f(\boldsymbol{\theta}, \boldsymbol{x})}{\partial \boldsymbol{H}^{(k)}}\right|_{\substack{\boldsymbol{x}=\boldsymbol{x}\\ \boldsymbol{\theta}=\boldsymbol{\theta}}} \right\|_{\mathcal{F}} \left\|\boldsymbol{X}^{(k)}\right\|_{\mathcal{F}} \left\|\boldsymbol{W}^{(k)}\right\|_{\mathcal{F}}, \tag{4}$$

where we use $\lesssim$ to hide constant coefficients and small $O(1/d)$ terms.

Denote $\alpha_k = \sqrt{\frac{\log c_k}{d_k}} \left\| \frac{\partial f(\boldsymbol{\theta}, \boldsymbol{x})}{\partial \boldsymbol{H}^{(k)}} \Big|_{\substack{\boldsymbol{x}=\boldsymbol{x} \\ \boldsymbol{\theta}=\boldsymbol{\theta}}} \right\|_{\mathcal{F}} \left\| \boldsymbol{X}^{(k)} \right\|_{\mathcal{F}} \left\| \boldsymbol{W}^{(k)} \right\|_{\mathcal{F}}$ be the coefficient in Corollary B.2, we estimate the error introduced by b-bit quantization at the k-th layer by $\alpha_k 2^{-b_k}$. Then the bits-allocation problem eq. (1) can be rewritten as

$$\begin{aligned} \{b_k^*\}_{k=1}^L = \arg \min_{\{b_k\}_{k=1}^L} \sum_{k=1}^L \alpha_k 2^{-b_k} \\ \text{s.t.} \quad \sum_{k=1}^L b_k m_k \leq R, \\ b_k \in \mathscr{B}, \forall k \in [L]. \end{aligned} \tag{5}$$

B.1 Solving the Bits-Allocation Problem

Now we consider solving problem eq. (5). At first glance, this is a nonlinear integer programming problem which is known to be NP-complete [21]. Nevertheless, we note that the problem size is manageable in practice so that it can be efficiently solved via dynamic programming. Let $g = \gcd(m_1, \cdots m_L, R)$ be the greatest common divisor (GCD) of all m_k's and R, then we have

$$\sum_{k=1}^L b_k m_k \leq R \iff \sum_{k=1}^L b_k \frac{m_k}{g} \leq \frac{R}{g}, \tag{6}$$

and thus the total bits budget can be reduced to R/g. Thanks to the design choice of most large language models, whose hidden sizes are usually powers of 2, in practice g is typically very large. As a result, the size of this problem can be prominently reduced to a scale where finding the global optimum via a dynamic programming algorithm becomes feasible. We provide the algorithm for solving this problem in Appendix H.1.

Algorithm 4 from Appendix H.1 runs in $O(L|\mathscr{B}|R/g)$ time. In practice, both L and $|\mathscr{B}|$ are less than 100, $\frac{R}{g}$ is on the order of 10^5 and the entire process can be completed in a few seconds on CPU. For LLaMA models, the value of g is on the order of 10^6, making the divide-by-GCD trick – although seemingly simple – crucial for efficiency; without it, the algorithm would be millions of times slower.

B.2 Few-shot and Zero-shot Calibration

In Algorithm 1, a calibration set $\{\boldsymbol{x}_i\}_{i=1}^{n_c}$ is used to estimate the error (i.e. α_k in eq. (5)). In principle, we can certainly use a large amount of data to obtain a better estimator of α_k. However, in RaanA, α_k only depends on the norm of input and the Jacobian of the overall output w.r.t. the output of layer k. Unlike the calibration in OBQ-based methods that requires estimating the layer-wise Hessian, these values here are practically very stable and can be estimated using a very small amount of data. This has also been observed in some previous work – for example, [22] found that the empirical mean of Jacobian quickly converges to the Lipschitz constant of the model.

In our implementation, we consider two settings: **few-shot calibration** and **zero-shot calibration**. In few-shot setting, we use 5 samples from the training set of the corresponding dataset, which is significantly less than what mainstream methods use (e.g. Quip# uses more than 6000 samples [34]). In the zero-shot calibration setting, we only use one synthetic sentence to estimate α_k, without resorting to any actual training data. Specifically, we repeat the following sentence 100 times to form our single calibration data point in the zero-shot setting [3]:

> "The curious fox leaped over the quiet stream, its reflection rippling in the golden afternoon light."

C RaBitQ-H: A variant of RaBitQ with Randomized Hadamard Transformation

In [12], the authors introduced RaBitQ, a universal multi-bit vector quantization algorithm that preserves the results of vector inner products and achieves an asymptotically optimal error rate. In large

[3] This sentence was suggested by ChatGPT.

language models, the basic operation and bottleneck is matrix multiplication (MM), which can also be viewed as performing multiple inner products. Therefore, it is possible to adopt RaBitQ as our quantization method. RaBitQ has the appealing property that the error rate is controlled in all cases with high probability and does not require handling outliers (see details in Appendix F.2). This ensures the desired property stated in Assumption B.1.

However, directly applying RaBitQ in the MM scenario can be inefficient. In RaBitQ, it requires applying a random rotation to each vector during the pre-processing stage. For a d-dimensional vector, applying a random rotation takes $O(d^2)$ time. This is acceptable in the original ANNS scenario, where the data dimension is much smaller than the number of data points, making the rotation cost negligible. In contrast, in the context of large language models and MM, the number and dimensionality of the vectors are comparable[4], making the cost of performing or storing a random rotation almost the same as that of the original MM, negating the potential benefits of quantization.

To address this issue, we replace the random rotation in the original paper with a Randomized Hadamard Transformation (RHT), which is known to approximate random rotation in many cases [33]. RHT has the following desired properties: 1) For each layer k, it only requires d_k random bits, making storing the transformation not a bottleneck; 2) It can be efficiently computed in time $O((c_k + n)\log d_k)$ using existing Hadamard kernels [2]. We adopt the analysis from the original RaBitQ paper [12] and confirm that the output of RaBitQ-H satisfies Assumption B.1. The complete quantization and de-quantization algorithms are provided in Algorithms 2 and 3.

Algorithm 2: Preprocess: Quantization

Input: weight matrix $\boldsymbol{W}^{(k)} \in \mathbb{R}^{d_k \times c_k}$, desired number of bits $B \in \mathscr{B}$;

$\boldsymbol{\xi} \leftarrow \{\xi_i\}_{i=1}^{d_k}$ where ξ_i is sampled i.i.d. from the Rademacher distribution;

$\boldsymbol{D}^{(k)} \leftarrow \operatorname{diag}(\boldsymbol{\xi})$;

$\boldsymbol{W}' \leftarrow \mathsf{Hadamard}\left(\boldsymbol{D}^{(k)}\boldsymbol{W}\right)$;

$\widehat{\boldsymbol{W}}^{(k)}, \boldsymbol{r}^{(k)} \leftarrow \mathsf{RaBitQ}\left(\boldsymbol{W}', B\right)$;

Return: $\widehat{\boldsymbol{W}}^{(k)}, \boldsymbol{r}^{(k)}, \boldsymbol{D}^{(k)}$.

Algorithm 3: Inference: Matrix Multiplication Estimation

Input: input features $\boldsymbol{X}^{(k)} \in \mathbb{R}^{n \times d_k}$, quantized weight matrix $\widehat{\boldsymbol{W}}$, rescale factor $\boldsymbol{r}^{(k)} \in \mathbb{R}^{c_k}$, Rademacher samples $\boldsymbol{D}^{(k)} \in \mathbb{R}^{d_k}$, desired number of bits $B \in \mathscr{B}$;

$\boldsymbol{X} \leftarrow \mathsf{Hadamard}\left(\boldsymbol{D}^{(k)}\boldsymbol{X}^{(k)\top}\right)^{\top}$;

$\boldsymbol{z} \leftarrow \frac{2^B-1}{2}\boldsymbol{X}\boldsymbol{1}$;

$\boldsymbol{Y} \leftarrow \boldsymbol{X}\widehat{\boldsymbol{W}}^{(k)}\boldsymbol{1}\boldsymbol{r}^{\top} - \boldsymbol{z}\boldsymbol{r}^{\top}$;

Return: $\boldsymbol{Y}$.

Algorithms 2 and 3: **Quantization and Inference algorithms of RaBitQ-H**. Here Hadamard refers to Hadamard transformation (see Appendix F.1 for a formal definition) and RaBitQ refers to the original RaBitQ algorithm *without random rotation*. Principally they are both vector operations, and here by applying them to matrices we mean applying column-wise. $\boldsymbol{r}^{(k)} \in \mathbb{R}^{c_k}$ in the algorithms is the rescale factor output by Extended RaBitQ, and $\boldsymbol{1}$ means a c_k-dimensional all-one vector.

Remarks on Implementation. The practical implementation of RaBitQ-H is slightly more complicated than described here, involving two additional components. First, since the fast Hadamard transformation only applies to vector sizes that are powers of 2, we need to handle vectors whose sizes are not powers of 2. Second, we apply some small tricks in the implementation (such as input centralization) that preprocess the inputs and weights before running RaBitQ-H; these tricks do not affect the output of the quantization / de-quantization algorithm but in practice reduce the error rate. Due to space limitations, we defer these details to Appendices H.2 and H.3.

D Experiment Results

In this section, we present our experimental results. We focus on the performance loss after quantization, compared to the full-precision (fp16) model. Following previous work [25, 30], we evaluate our method on the LLaMA model family and language modeling tasks.

Datasets. As in previous studies [25, 30], we evaluate our method on the wikitext2 [27] and c4 [29] datasets. We split the test / validation sets into sequences of length 2048 as test samples and

[4] c_k is the number of vectors and d_k is the vector dimension.

measure the average perplexity of the quantized model on each test sample. For c4, since the full validation set is too large and model performance converges quickly, we use 500 samples as the test set.

Baseline Methods. We compare RaanA with GPTQ [11], AWQ [25], OmniQuant [30], and Quip#$_{\text{no FT \& no } E_8}$ [34][5]. Additionally, for 4-bit quantization, we also compare with EasyQuant [31], a lightweight quantization method that does not require calibration data and targets 4-bit quantization. Note that most baseline models use tricks such as grouping and keeping full-precision outliers that introduce extra bit costs[6], generally ranging from 0.1 to 0.3 bits. To make a fair comparison, we report RaanA performance with $x + 0.1$ bits and $x + 0.3$ bits for $x \in 2, 3, 4$.

D.1 Performance

In this sub-section, we compare the performance of RaanA under few-shot calibration with baseline methods. The results on wikitext2 are reported in Table 1 and we defer the results on c4 to Appendix I due to limited space. It is clear from the table that RaanA is comparable with or better than baseline methods, especially in the extreme regime of 2+ bits quantization.

Method	Avg. bits	llama-7b	llama-13b	llama2-7b	llama2-13b	llama2-70b
fp16	16	5.68	5.09	5.47	4.88	3.31
GPTQ	2+	44.01	15.60	36.77	28.14	-
AWQ	2+	2.6e5	2.8e5	2.2e5	1.2e5	-
OmniQuant	2+	9.72	7.93	11.06	8.26	6.55
Quip#*	2	9.95	7.18	12.3	**7.60**	4.87
RaanA	2.1	13.70	8.28	18.31	51.05	4.81
RaanA	2.3	**8.53**	**6.63**	**10.63**	8.96	**4.49**
GPTQ	3+	8.81	5.66	6.43	5.48	3.85
AWQ	3+	6.35	5.52	6.24	5.32	-
OmniQuant	3+	6.15	5.44	6.03	5.28	3.78
Quip#*	3	6.29	5.52	6.19	5.34	3.71
RaanA	3.1	6.33	5.53	6.20	5.48	3.66
RaanA	3.3	**6.10**	**5.38**	**6.00**	**5.27**	**3.59**
EasyQuant	4+	6.01	5.29	-	-	-
GPTQ	4+	6.22	5.23	5.69	4.98	3.42
AWQ	4+	5.78	5.19	5.60	4.97	-
OmniQuant	4+	**5.77**	**5.17**	**5.58**	**4.95**	**3.40**
Quip#*	4	5.83	5.20	5.66	5.00	3.42
RaanA	4.1	5.86	5.20	5.69	5.02	3.42
RaanA	4.3	5.83	**5.17**	5.65	4.98	**3.40**

Table 1: **Perplexity results on wikitext2**. Methods labeled '+" in the number of bits use tricks such as grouping and keeping outliers full-precision that bring $0.1 \sim 0.3$ extra bit cost. Quip#* refers to Quip#$_{\text{no FT \& no } E_8}$. For OmniQuant, GPTQ and AWQ, we compare with the version with grouping size 128. Best performance of each category is labeled bold.

D.2 Zero-shot Calibration vs Few Shot Calibration

In this subsection, we compare the performance between few-shot calibration and zero-shot calibration. Results for wikitext2 are displayed in Table 2 for results on c4 are deferred to Appendix I. We also add results from EasyQuant, which also does not require calibration data, as a comparison. It is clear from the resutls that, although the performance does generally go down a little bit with zero-shot calibration, they are generally comparable with few-shot calibration results, validating the effectiveness of zero-shot calibration for RaanA.

[5]The full version of Quip# [34] fine-tunes the model after quantization. In this paper, we compare RaanA with Quip#$_{\text{no FT \& no } E_8}$, the version without fine-tuning, since fine-tuning is a universal plug-in component in quantization and is orthogonal to our contribution.

[6]The only exception in our baselines is Quip#$_{\text{no FT \& no } E_8}$, which has a precise control over the average number of bits. We admittedly require slightly more bits to match its performance. However, RaanA is significantly more lightweight than Quip#$_{\text{no FT \& no } E_8}$, which uses over 6000 calibration samples to estimate the layer-wise Hessian.

Method	Avg. bits	llama-7b	llama-13b	llama2-7b	llama2-13b	llama2-70b
fp16	16	5.68	5.09	5.47	4.88	3.31
RaanA-few	2.1	13.70	8.28	18.31	51.05	4.81
RaanA-zero	2.1	15.50	10.12	26.13	13.37	7.89
RaanA-few	3.1	6.33	5.53	6.20	5.48	3.66
RaanA-zero	3.1	6.45	5.63	6.41	5.55	4.01
EasyQuant	4+	6.01	5.29	-	-	-
RaanA-few	4.1	5.86	5.20	5.69	5.02	3.42
RaanA-zero	4.1	5.86	5.23	5.73	5.04	3.50

Table 2: **Perplexity Comparison Between Zero-shot Calibration and Few-shot Calibration on wikitext2**. RaanA-zero refers to RaanA with zero-shot calibration and RaanA-few refers to RaanA with few-shot calibration.

D.3 Quantization Time

We note that due to the lack of GPU implementation of RaBitQ, the main part of RaanA is run on CPU, which is the time bottleneck. In our current implementation, the only parts requiring GPUs are calibration (which requires one or a few backward passes of the model) and Hadamard Transformation. Despite the main part running on CPU, RaanA still runs much faster than many existing quantization methods, demonstrating its high efficiency and device independence.

We report the time RaanA used for quantization in Table 3 under a specific setting as an illustration of the efficiency of RaanA. The experiments are conducted with 4 NVIDIA A100 GPUs[7]. As shown, RaanA completes the quantization of a 70B model in under one hour, significantly faster than other heavyweight quantization methods such as Quip#$_{\text{no FT \& no } E_8}$, which can take up to 10 hours for the same model size, despite having comparable performance.

Model	Time (s)
llama2-7b	301.74
llama2-13b	567.61
llama2-70b	3293.26

Table 3: **Quantization Time**. The time required to complete the RaanA quantization process with few-shot calibration and average number of bits of 2.1.

E Discussion and Conclusion

In this work, we introduce RaanA: a new PTQ framework combining RaBitQ-H, a variant of RaBitQ that especially fits LLM quantization, and AllocateBits, an algorithm to allocate bit-widths across layers optimally. RaanA overcomes traditional challenges of PTQ methods such as high reliance on calibration and inflexible bits allocation. Extensive experiment results validate the performance of RaanA, especially highlighting the effectiveness of zero-shot calibration, eliminating the requirement of heavy calibration.

Limitations and Future Work Here we discuss current limitations of the RaanA framework and potential future directions to improve them. 1) More efficient implementation: As we mentioned in Appendix D.3, currently the implementation of RaanA is not optimal, as the computation is bottlenecked by the CPU-bound execution of RaBitQ. A more efficient implementation / approximation of RaBitQ (ideally on GPU) would vastly accelerates RaanA. 2) Finer-grained bits-allocation: currently RaanA allocates bit-widths layer-wisely, constraining the parameters in each layer to share the same bit-width, which can be sub-optimal. It is possible to consider a finer-grained bits-allocation, e.g. column-wisely or even entry-wisely, as a future direction.

Remark on LLM Design Last but not least, we would like to advocate future large language model designers to use a powers of 2 as the hidden size more often; as arbitrary as it may seem, this choice actually has the following advantages that make quantization easier and faster: 1) It maximizes the GCD between layers and thus improves the speed of the AllocateBits algorithm. 2) It makes it easier to use fast Hadamard Transformations since it is only defined for spaces whose dimension is a power of 2.

[7]Here the GPU configuration does not significantly impact the quantization time since the bottleneck is the CPU computation of RaBitQ. The machine we use has two AMD EPYC 7513 CPUs.

F Detailed Introduction to Algorithmic Tools

In this section, we introduce the algorithms used in RaanA.

F.1 Detailed Introduction to Randomized Hadamard Transformation

In this sub-section, we provide a formal definition of RHT. Let d be a positive integer number d that is a power of 2, the Hadamard Transformation of size d is a linear transformation recursively defined by the following matrix:

$$\boldsymbol{H}_d := \begin{bmatrix} \boldsymbol{H}_{d/2} & \boldsymbol{H}_{d/2} \\ \boldsymbol{H}_{d/2} & -\boldsymbol{H}_{d/2} \end{bmatrix} \tag{7}$$

and $\boldsymbol{H}_1 = 1$. For a vector $\boldsymbol{x} \in \mathbb{R}^d$, we define

$$\mathsf{Hadamard}(\boldsymbol{x}) := \frac{1}{\sqrt{d}} \boldsymbol{H}_d \boldsymbol{x}. \tag{8}$$

it has been shown that the Hadamard Transformation can be computed in a fast way, i.e. applying a Hadamard Transformation to each column of an $d \times n$ matrix only requires $O(n \log d)$ time.

Let $\boldsymbol{\xi} = \{\xi_i\}_{i=1}^n$ be n i.i.d. sampled Rademacher variable, and let $\boldsymbol{D} = \operatorname{diag}(\boldsymbol{\xi})$. For a matrix $\boldsymbol{x} \in \mathbb{R}^d$, we say

$$\boldsymbol{x} \mapsto \mathsf{Hadamard}(\boldsymbol{D}\boldsymbol{x}) \tag{9}$$

the Randomized Hadamard Transformation (RHT) of $\boldsymbol{x}$, which is exactly what we used in Algorithm 2. Since the Hadamard Transformation is orthonormal, it's not hard to restore the result of RHT, we only need to store the vector $\boldsymbol{\zeta}$, which has only d binary entries.

F.2 Detailed Introduction to RaBitQ

In this sub-section, we provide a brief introduction to RaBitQ [13] and its multi-bit extension [12]. The RaBitQ methods were proposed initially for vector quantization in database systems. They target to produce accurate estimation of inner product and Euclidean distances based on the quantized vectors while using the minimum space for storing quantization codes. Specifically, let $\boldsymbol{P}$ be a Johnson-Lindenstrauss Transformation (a.k.a., random rotation) [19]. For a vector $\boldsymbol{x} \in \mathbb{R}^d$, RaBitQ randomly rotates it into $\boldsymbol{P}\boldsymbol{x}$ and quantizes $\boldsymbol{P}\boldsymbol{x}$ to a vector of b-bit unsigned integers $\bar{\boldsymbol{x}} \in [2^b]^d$ with a rescaling factor $t \in \mathbb{R}$. Then for another vector $\boldsymbol{y} \in \mathbb{R}^d$, RaBitQ estimates the inner product $\langle \boldsymbol{x}, \boldsymbol{y} \rangle$ with

$$\langle \boldsymbol{x}, \boldsymbol{y} \rangle \approx \langle t \cdot (\bar{\boldsymbol{x}} - c_b \cdot \mathbf{1}_d), \boldsymbol{P}\boldsymbol{y} \rangle \tag{10}$$

where $\mathbf{1}_d$ denotes the d-dimensional vector whose coordinates are ones and $c_b = (2^b - 1)/2$. For the details of the quantization algorithms and the rescaling factors, we refer readers to the original paper [12].

As has been proven in the original RaBitQ papers, RaBitQ guarantees that the estimation is **unbiased** and **asymptotically optimal** in terms of the trade-off between the error bound and the space for storing the codes. Specifically, with probability as least $1 - \delta$, to guarantee that

$$\left| \langle \boldsymbol{x}, \boldsymbol{y} \rangle - \langle t(\bar{\boldsymbol{x}} - c_b \mathbf{1}_d), \boldsymbol{P}\boldsymbol{y} \rangle \right| < \epsilon \|\boldsymbol{x}\| \|\boldsymbol{y}\| \tag{11}$$

it suffices to let $b = \Theta\left(\log\left(\frac{1}{d} \cdot \frac{\log(1/\delta)}{\epsilon^2}\right)\right)$ when ϵ is sufficiently small, i.e., $\frac{\log(1/\delta)}{\epsilon^2} > d$. This result achieves the optimality established in a theoretical study [4].

Additionally, RaBitQ provides an empirical formula for the trade-off between errors and spaces. Specifically, with probability at least 99.9%, we have

$$\left| \langle \boldsymbol{x}, \boldsymbol{y} \rangle - \langle t(\bar{\boldsymbol{x}} - c_b \mathbf{1}_d), \boldsymbol{P}\boldsymbol{y} \rangle \right| < \frac{c_{\text{error}}}{\sqrt{d} 2^b} \|\boldsymbol{x}\| \|\boldsymbol{y}\| \tag{12}$$

where $c_{\text{error}} = 5.75$.

G Theoretical Results

In this section, we prove the main theorem of this paper, which directly leads to Corollary B.2. We first state a formal version of Assumption B.1, stating that the error from RaBitQ (and RaBitQ-H) quantization has sub-exponential error.

Assumption G.1. [[12]] There exists a constant $K > 0$, such that for any $k \in [L], i \in [n], j \in [c_k]$, $\boldsymbol{\epsilon}^{(k)}(b) \in \mathbb{R}^{d_k}$ is a random vector such that

$$\forall t > 0, \mathbb{P}\left\{\left|\epsilon_{i,j}^{(k)}(b)\right| > t\right\} \leq 2\exp\left[-K\left(\frac{t\sqrt{d_k}}{2^{-b_k}\left\|\boldsymbol{x}_i^{(k)}\right\|\left\|\boldsymbol{w}_j^{(k)}\right\|}\right)^2\right]. \tag{13}$$

Notice that Assumption G.1 does not assume $\epsilon_{i,j}^{(k)}(b)$-s are independent. We provide the following algorithm analyzing the behavior of a function under an $O(1/d)$ small error, which together with Assumption G.1 implies Corollary B.2.

Theorem G.2. *There exists constant $K > 0$ satisfies the following statement. Suppose $c \geq 2$, $d \gg c$ and $\boldsymbol{\epsilon} \in \mathbb{R}^{n\times c}$ is a random vector, such that $\forall i, j \in [n] \times [c]$,*

$$\mathbb{P}\left\{|\epsilon_{i,j}| > t\right\} \leq 2\exp\left(-\frac{Ct^2 d}{\lambda^2\gamma_{i,j}^2}\right) \tag{14}$$

for some constant K_1 (notice that $\epsilon_{i,j}$-s are not necessarily independent). Let $g : \mathbb{R}^{n\times c} \to \mathbb{R}$ be a smooth function, then for any $\boldsymbol{h} \in \mathbb{R}^{n\times c}$, the following statement holds with probability at least 0.99*:*

$$|g(\boldsymbol{h}) - g(\boldsymbol{h}+\boldsymbol{\epsilon})| \leq K\sqrt{\frac{\log c}{d}}\|\boldsymbol{\gamma}\|_{\mathcal{F}}\|\nabla g(\boldsymbol{h})\|_{\mathcal{F}} + O\left(\frac{\left\|\nabla^2 g(\boldsymbol{h})\right\|}{d}\right). \tag{15}$$

Proof. From the given condition, we have

$$\forall t > 0, \mathbb{P}\left\{|\epsilon_{i,j}| > t\gamma_{i,j}\right\} \leq 2\exp\left(-K_1 t^2 d/\lambda^2\right). \tag{16}$$

Thus we have

$$\forall t > 0, \mathbb{P}\left\{\exists i, j \in [n]\times[c], |\epsilon_{i,j}| > t\gamma_{i,j}\right\} \leq 2c\exp\left(-K_1 t^2 d/\lambda^2\right). \tag{17}$$

For constant $K_2 > 0$, let $t_0 = K_2\lambda\sqrt{\frac{\log c}{d}}$. It is evident that there exists a constant K_2 that only depends on K_1, such that

$$\mathbb{P}\left\{\exists i, j \in [n]\times[c], |\epsilon_{i,j}| > t_0\gamma_{i,j}\right\} \leq 0.01. \tag{18}$$

Thus, with probability ≥ 0.99, we have

$$\forall i, j \in [n]\times[c], |\epsilon_{i,j}| \leq K_2\lambda\gamma_{i,j}\sqrt{\frac{\log c}{d}}. \tag{19}$$

Therefore, with probability at least 0.99, we have

$$\|\boldsymbol{\epsilon}\|_{\mathcal{F}} \leq \sqrt{\sum_{i=1}^{n}\sum_{j=1}^{c}\epsilon_{i,j}^2} \leq K_2\lambda\sqrt{\frac{\log c}{d}}\|\boldsymbol{\gamma}\|_{\mathcal{F}}. \tag{20}$$

Using Taylor expansion, we have

$$\begin{aligned}
|g(\boldsymbol{h}) - g(\boldsymbol{h}+\boldsymbol{\epsilon})| &\leq \langle\nabla g(\boldsymbol{h}), \boldsymbol{\epsilon}\rangle + O\left(\left\|\nabla^2 g(\boldsymbol{h})\right\|\|\boldsymbol{\epsilon}\|^2\right) && (21)\\
&\leq \|\nabla g(\boldsymbol{h})\|_{\mathcal{F}}\|\boldsymbol{\epsilon}\|_{\mathcal{F}} + O\left(\frac{\left\|\nabla^2 g(\boldsymbol{h})\right\|}{d}\right) && (22)\\
&\leq K_2\lambda\sqrt{\frac{\log c}{d}}\|\boldsymbol{\gamma}\|_{\mathcal{F}}\|\nabla g(\boldsymbol{h})\|_{\mathcal{F}} + O\left(\frac{\left\|\nabla^2 g(\boldsymbol{h})\right\|}{d}\right). && (23)
\end{aligned}$$

$\square$

Corollary B.2 is thus a direct corollary of Assumption G.1 and Theorem G.2. Notice that in Corollary B.2 we view the second-order derivative of f w.r.t. $\boldsymbol{H}_k$ a constant (evaluated at a fixed point), and therefore omit the $\|\nabla^2 \cdot \|$ term.

H Implementation Details

In this section, we provide some implementation details that are not elaborated in the main text due to space limit.

H.1 Bits-Allocation Algorithm

In Appendix B, we mentioned that the bits-allocation problem can be solved efficiently by a dynamic programming algorithm after applying the divide-by-GCD trick. In Algorithm 4 we provide the detailed algorithm description.

In our implementation of Algorithm 4, we compute α_k as

$$\alpha_k = \frac{1}{\sqrt{d_k}} \left\| \left. \frac{\partial f(\boldsymbol{\theta}, \boldsymbol{x})}{\partial \boldsymbol{H}^{(k)}} \right|_{\substack{\boldsymbol{x}=\boldsymbol{x} \\ \boldsymbol{\theta}=\boldsymbol{\theta}}} \right\|_{\mathcal{F}} \left\| \boldsymbol{X}^{(k)} \right\|_{\mathcal{F}} \left\| \boldsymbol{W}^{(k)} \right\|_{\mathcal{F}}, \tag{24}$$

omitting the $\log c_k$ term in the , since it is almost constant across layers and therefore has negligible impact on the optimization.

Algorithm 4: Bits Allocation

Input: Coefficients $\{\alpha_k\}_{k=1}^L \in \mathbb{R}^L$, number of bits candidate $\mathscr{B}$, overall budget $R \in \mathbb{N}$
Initialize $f_{k,r} = +\infty$, where $k \in [L], r \in [R] \cup \{0\}$;
$g \leftarrow \text{gcd}\,(m_1, \cdots, m_L, R)$;
for $k = 1, 2, \cdots L$ **do**
  for $b \in \mathscr{B}$ **do**
    $r \leftarrow \left\lfloor \frac{m_k B}{g} + \frac{1}{2} \right\rfloor$;
    $c \leftarrow \alpha_k 2^{-b}$;
    if $k = 1$ **then**
      $f_{k,r} \leftarrow c$;
      $s_{k,r} \leftarrow \{b\}$;
    end
    else
      for $r' = 0, 1, \cdots \frac{R}{g} - r$ **do**
        if $f_{k,r'+r} > f_{k-1,r'} + c_k$ **then**
          $f_{k,r'+r} \leftarrow f_{k-1,r'} + c_k$;
          $s_{k,r'+r} \leftarrow s_{k,r'} \parallel \{b\}$
        end
      end
    end
  end
end
$r^* \leftarrow \arg\min_{r=0}^{R} f_{L,r}$;
Return: s_{L,r^*}.

Algorithm 4: The algorithm for bits allocation. In the algorithm $\alpha_k = \mathbb{E}_{x\sim\mathcal{D}}\lambda_k r_k$ is the coefficient for the k-th layer, $\{b\}$ represents a sequence with only one element b and $\parallel$ stands for sequence concatenation. The returned value is a sequence indicating the optimal bit-widths each layer, i.e. $s_{L,r^*} = \{b_k^*\}_{k=1}^L$.

H.2 Randomized Hadamard Transformation for Arbitrary Dimensionality

As we mentioned in Appendix F.1, the fast Hadamard Transformation is only defined with vector dimensionality d that is a power of 2. However, in practice, it is not always satisfied. In previous work such as [34], this issue is solved by finding the largest factor of d which is a power of 2, say $\tilde{d}$, and applying Hadamard Transformation block-wisely, with each block has size $\tilde{d}$. However, in our experiment, we found this method extremely inefficient. For example, for LLaMA models, there can be > 20 blocks.

Therefore, in this paper, we apply an easy and universal method to address this issue. We first find the largest power of 2 that is less or equal to d, i.e. $\hat{d} = 2^{\lfloor \log_2 d \rfloor}$, and apply RHT for the first and last $\hat{d}$ dimensions respectively. Our algorithm is described in Algorithm 5.

Algorithm 5: Practical RHT

Input: Vector $\boldsymbol{x} \in \mathbb{R}^d$

$\hat{d} = 2^{\lfloor \log_2 d \rfloor}$;

for *j = 1,2* **do**

  $\boldsymbol{\xi}^{(j)} \leftarrow \left\{\xi_i^{(j)}\right\}_{i=1}^{\hat{d}}$, where $\xi_i^{(j)}$ is sampled i.i.d. from the Rademacher distribution;

  $\boldsymbol{D}^{(j)} \leftarrow \operatorname{diag}\left(\boldsymbol{\xi}^{(j)}\right)$;

end

$\boldsymbol{x}_{1:\hat{d}} \leftarrow \mathsf{Hadamard}\left(\boldsymbol{D}^{(1)}\boldsymbol{x}_{1:d}\right)$;

$\boldsymbol{x}_{d-\hat{d}+1:d} \leftarrow \mathsf{Hadamard}\left(\boldsymbol{D}^{(2)}\boldsymbol{x}_{d-\hat{d}+1:d}\right)$;

Return: $\left(\boldsymbol{x}, \boldsymbol{D}^{(1)}, \boldsymbol{D}^{(2)}\right)$.

Algorithm 4: Practical Randomized Hadamard Transformation. $\boldsymbol{x}_{a:b}$ refers to the sub-vector of $\boldsymbol{x}$ consists of the a-th entry to the b-th entry of $\boldsymbol{x}$.

H.3 Tricks used in Quantization

In the actual implementation of RaanA, we optionally apply some transformations before performing quantization. Formally, for the $d \times c$ matrix, we define a **trick** to be a invertible **linear** transformation $T : \mathbb{R}^{n\times d} \to \mathbb{R}^{n\times d}$. Then for a linear layer where input matrix is $\boldsymbol{X} \in \mathbb{R}^{n\times d}$ and weight matrix is $\boldsymbol{W} \in \mathbb{R}^{d\times c}$, we have

$$\boldsymbol{X}\boldsymbol{W} = T^{-1}\left(T(\boldsymbol{X})\boldsymbol{W}\right). \tag{25}$$

Notice that T can be have a memory, i.e. it can return an auxiliary term to help T^{-1} to recover the computation result. After applying T, in the de-quantization stage we only need to estimate the matrix multiplication results for $T(\boldsymbol{X})\boldsymbol{W}$.

In practice, we the following heuristic tricks are optionally used.

- **Centralization**: $T(\boldsymbol{X}) = \left[\boldsymbol{X} - \boldsymbol{1}\boldsymbol{s}(\boldsymbol{X})^\top, \boldsymbol{s}(\boldsymbol{X})\right]$, where $\boldsymbol{s}(\boldsymbol{X}) \in \mathbb{R}^d$ is the average of all rows of $\boldsymbol{X}$;
- **Row Outlier Excluding**: $T(\boldsymbol{X}) = [\boldsymbol{X}_{\neg \boldsymbol{M}_r}, \boldsymbol{X}_{\boldsymbol{M}_r}]$, where $\boldsymbol{M}_r$ is a mask vector selecting the top 0.3% rows of $\boldsymbol{X}$ with largest norm, and $\neg\boldsymbol{M}_r$ selects the opposite. $\boldsymbol{X}_{\boldsymbol{M}_r}$ indicates selecting rows of $\boldsymbol{X}$ according to the mask vector $\boldsymbol{M}_r$;
- **Column Outlier Excluding**: $T(\boldsymbol{X}) = [\boldsymbol{X}_{:,\boldsymbol{M}_c}, \boldsymbol{X}_{:,\neg\boldsymbol{M}_c}]$, where $\boldsymbol{M}_c$ is a mask vector selecting the top 0.3% columns of $\boldsymbol{X}$ with largest norm, and $\neg\boldsymbol{M}_c$ selects the opposite. $\boldsymbol{X}_{:,\boldsymbol{M}_c}$ indicates selecting columns of $\boldsymbol{X}$ according to the mask vector $\boldsymbol{M}_c$.

For each of the trick functions T described above, it returns two values. The first return value is the matrix that joins the subsequent computation, and the second return value is to be memorized in order to recover the original computational result.

Method	Avg. bits	llama-7b	llama-13b	llama-7b	llama-13b
fp16	16	7.08	6.61	6.97	6.47
GPTQ	2+	27.71	15.29	33.70	NAN
AWQ	2+	11.9e5	2.3e5	1.7e5	9.4e4
Quip#*	2	**11.7**	**8.67**	14.8	**9.57**
OmniQuant	2+	12.97	10.36	15.02	11.05
RaanA-t	2.1	20.88	13.08	23.17	-
RaanA-t	2.3	12.96	9.18	**12.66**	10.37
GPTQ	3+	7.85	7.10	7.89	7.00
AWQ	3+	7.92	7.07	7.84	**6.94**
OmniQuant	3+	**7.75**	7.05	**7.75**	6.98
Quip#*	3	7.82	**6.98**	7.85	6.98
RaanA-t	3.1	8.25	7.34	8.38	7.52
RaanA-t	3.3	7.92	7.15	7.99	-
EasyQuant	4+	7.71	6.97	-	-
OmniQuant	4+	**7.21**	**6.69**	**7.12**	**6.56**
GPTQ	4+	7.21	6.69	7.12	6.56
AWQ	4+	7.21	6.70	7.13	6.56
Quip#*	4	7.25	6.70	7.17	6.59
RaanA-t	4.1	7.51	6.91	7.53	6.88
RaanA-t	4.3	7.52	6.87	7.45	6.83

Table 4: **Perplexity results on c4**. The setting and format are the same as Table 1, except dataset.

Notice that for Row Outlier Excluding and Column Outlier Excluding, the trick needs to store a few rows / columns of $\boldsymbol{X}$. We intentionally restrict the outlier ratios less than 0.3% in order to keep the extra bits used to store the extra information negligible.

Practically we find these tricks perform differently across different settings. To keep the configuration consistent and avoid heavy hyper-parameter tuning, in all the experiments presented in this paper, we use Centralization and Column Outlier Excluding.

I Additional Experiment Results

Method	Avg. bits	llama-7b	llama-13b	llama-7b	llama-13b
fp16	16	7.08	6.61	6.97	6.47
RaanA-zero	2.1	19.37	13.14	31.31	18.03
RaanA-few	2.1	20.88	13.08	23.17	-
RaanA-zero	3.1	8.44	7.55	8.61	7.69
RaanA-few	3.1	8.25	7.34	8.38	7.52
RaanA-zero	4.1	7.56	6.96	7.59	6.93
RaanA-few	4.1	7.51	6.91	7.53	6.88

Table 5: **Perplexity Comparison Between Zero-shot Calibration and Few-shot Calibration on c4**. The setting and format are the same as Table 2, except dataset.

In this section we display additional experiment results. Table 4 contains perplexity comparison between RaanA with baseline methods on c4, where RaanA uses few-shot calibration. Table 5 contains the few-shot vs zero-shot comparison of RaanA on c4. These additional experimental results support our claim in main paper.