QStore: Quantization-Aware Compressed Model Storage

Raunak Shah
Univ. of Illinois Urbana-Champaign
raunaks3@illinois.edu

Zhaoheng Li
Univ. of Illinois Urbana-Champaign
zl20@illinois.edu

Yongjoo Park
Univ. of Illinois Urbana-Champaign
yongjoo@illinois.edu

ABSTRACT

Modern applications commonly leverage large, multi-modal foundation models, in complex workflows that demand the storage and usage of similar models in multiple precisions. A straightforward approach is to maintain a separate file for each model precision (e.g., INT8, BF16), which is indeed taken by model providers such as HuggingFace and Ollama. However, this approach incurs excessive storage costs as a higher precision model (e.g., BF16) is a superset of a lower precision model (e.g., INT8) in terms of information. Unfortunately, simply maintaining only the higher-precision model and requiring every user to dynamically convert the model precision is not desirable because every user of lower precision models must pay the cost for model download and precision conversion.

In this paper, we present QStore, a unified, lossless compression format for simultaneously storing a model in two (high and low) precisions efficiently. Instead of storing low and high-precision models separately, QStore stores low-precision model and only *residual information* needed to reconstruct high-precision models. The residual information size is significantly smaller than the original high-precision models, thus, achieving high storage cost savings. Moreover, QStore does not compromise model loading speed: The low-precision models can still be loaded quickly, while the high-precision models can also be reconstructed efficiently by merging low-precision data and the residual with QStore's lightweight decoding. We evaluate QStore for compressing multiple precisions of popular foundation models, and show that QStore reduces overall storage cost by up to 2.2× while enabling up to 1.7× and 1.8× faster model saving and loading versus existing approaches.

PVLDB Reference Format:
Raunak Shah, Zhaoheng Li, and Yongjoo Park. QStore: Quantization-Aware Compressed Model Storage . PVLDB, 19(3): 388 - 398, 2025.
doi:10.14778/3778092.3778100

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/illinoisdata/qstore.

1 INTRODUCTION

Foundation models have become highly accessible to users thanks to the availability of model hosting platforms such as HuggingFace [84], Ollama [12], and ModelScope [79]. Developers download pre-trained models hosted on these platforms (e.g., from cloud storage), then apply them to tasks such as fine-tuning [26, 77, 81], distillation [89, 95] and inference [37, 38, 65, 85, 93, 96]. Different tasks demand different model precisions; for example, fine-tuning is often performed with higher precisions such as FP16 [63], then, the fine-tuned model would be *quantized* to a lower precision such as INT8 [13, 51] for faster inference. Hence, many workflows require accessing the same model under different precisions: in addition to fine-tuning-then-inference, other tasks with this requirement include Model Cascade [27, 94] and Chaining [23, 82, 86]. Moreover, data scientists and researchers also iterate between different-precision models for testing and benchmarking [10, 11, 53].

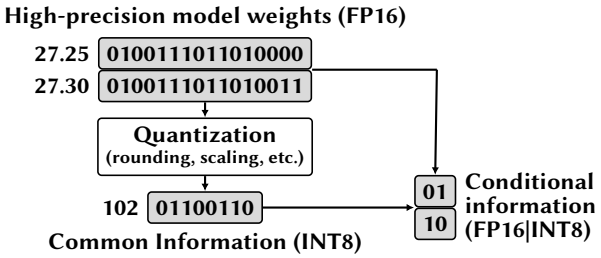


Figure 1: QStore **stores conditional bit representation of high-precision weights alongside common low-precision, quantized weights to store both model precisions using fewer bits.**

Storing Multiple Models is Costly. Currently, a common approach to maintaining multiple varying-precision models during the aforementioned tasks is to store them as is (i.e., separately storing the versions) [10, 11]. Yet, as newer, more complex tasks demand ever-increasing model sizes (e.g., Mistral-7B [50] being sufficient for simple math, while complex, multi-modal tasks [85] require larger models such as Qwen2.5-VL 32B [21]), storage of multiple model version can become costly — for example, 91.8 GB is required to store just the BF16 [52] and INT8 (quantized) versions of Deepseek-Coder 33B [97]. Space cost is a significant issue for model developers, and also increases cloud storage costs for model hubs like HuggingFace, Ollama, and ModelScope, which end up storing multiple precisions of these models separately to account for anticipated user accesses.

One potential approach to reduce storage cost is to only store the highest-precision model (e.g., FP16 or BF16), then quantize in-memory if lower precisions (e.g., INT8) are needed [39]. However, retrieving a low-precision model with this approach is inefficient as it requires (i) loading more data than necessary (i.e., the high-precision model) and (ii) an expensive quantization process (e.g., up to 21 GPU minutes for a 13B model [39]). Alternatively, models can be compressed with an algorithm such as LZ4 [3], ZSTD [4], or ZipNN [42]. However, these algorithms either use generic techniques that underperform on ML weights (e.g., LZ4 and ZSTD), or are tailored to specific precisions (e.g., ZipNN for FP16/BF16).

Proceedings of the VLDB Endowment, Vol. 19, No. 3 ISSN 2150-8097.
doi:10.14778/3778092.3778100

Our Intuition. We propose QStore, a data format for efficiently storing varying precision model versions. We observe that despite being quantized, a lower-precision (e.g., INT8) model version contains information that is also in a higher-precision (e.g., FP16, BF16) version. Hence, versus separately compressing and storing a high and low-precision model pair, it is possible *simultaneously* represent both models (Fig 1): much information in the high-precision FP16 model is already in the low-precision (i.e., quantized) INT8 version. Hence, given an already stored low-precision model, we can also store the high-precision model using few *additional* bits per weight as 'extra information' not in the low-precision model (i.e., the 'FP16 | INT8' *conditional* model). Such a unified data format would (1) save storage space versus storing models separately (regardless of compression), (2) enable faster loading of the lower-precision model versus loading a high-precision model and quantizing it, while (3) still enabling fast loading of the high-precision model.

Challenges. Designing a unified data format for simultaneously and efficiently storing a high and low-precision model pair is challenging. First, we need to define 'extra information' absent from the lower-precision model required to rebuild the higher-precision model. Quantizing a higher-precision model to a lower-precision one loses significant information (e.g., from operations like rounding); our definition should encapsulate this information gap for lossless reconstruction. Identifying this information gap is nontrivial, as a quantized weight may significantly differ from the original in both bit representation and numerical magnitude (Fig 1). Second, our representations of the lower-precision model's information and 'extra information' should acceptably balance storage/processing speed: for example, naïvely defining and storing information at bit-level granularity would enable the most efficient model storage, but can result in unacceptable model loading and saving times.

Our Approach. Our key idea for QStore is to design a generalized compressed representation for conditional information that can work well despite the differences between floats and integers; such a format would allow us to load low and high-precision models, regardless of their data type, with perfect accuracy.

First, for storage, given a high and low-precision model pair's weights, we separately encode the low-precision model weights and *conditional* weights (i.e., the 'extra information') with novel entropy coding and intelligent grouping strategies to enable significantly better compression ratios versus separately compressing the two models using off-the-shelf compression algorithms.

Then, for model loading from QStore, we process the encoded low-precision model's weights, or additionally the conditional weights, to retrieve the low-precision or high-precision model, respectively. We perform decoding at a byte-level granularity to ensure high decoding speeds on common computing architectures [70]. Our decoding is notably lossless (e.g., versus dequantization [69]).

Contributions. Our contributions are as follows:

(1) **Format.** We describe how QStore, a data format to efficiently store a high and low-precision model pair. (§3)
(2) **Usage.** We describe efficient encoding and decoding schemes for storing/loading models to/from QStore. (§4)
(3) **Evaluation.** We verify on 6 foundation models that QStore reduces model pair storage costs by up to 55%, enables up to 1.6× and 2.2× faster loading and saving, respectively, versus alternative methods, and generalizes to various data types, quantization algorithms, and >2 model chains. (§6)

2 BACKGROUND

Efficiently storing and deploying large foundation models is challenging, which QStore addresses via efficiently and simultaneously compressing multiple model precision versions. This section covers related work on quantization (§2.1) and compression (§2.2).

2.1 Quantization

Quantization is commonly applied to models to achieve desired quality-resource consumption tradeoffs. In this section, we overview the pros and cons of common quantization techniques, and key differences between QStore and quantization.

Common Quantization Targets. While 32-bit floating-point (FP32) precision was once standard [68], the recent increases in model sizes and corresponding increases in computational and memory requirements have driven the adoption of lower-precision, quantized model formats. For example, 16-bit precision (FP16 [43], BF16 [7, 52]) formats have become a de-facto standard for training and fine-tuning to balance between accuracy and resource consumption. For more resource-constrained scenarios or latency-sensitive applications (e.g., on-device processing [88]), further quantization is common—typically to 8-bit (INT8) [34, 46], but sometimes more aggressively to 4-bit (INT4, NF4) [35, 39, 64] or even lower [80]. Recently, FP8 quantization has also been used during inference [67].

Quantization Methods. There exists several notable classes of quantization methods commonly applied to foundation models. (1) **RTN (round to nearest)** rounds weights to the nearest low-precision value (e.g., $42.25 \rightarrow 42$), which is fast, but can significantly degrade model accuracy with outlier weights. (2) **Channel-wise quantization** such as LLM.int8() [34] and SmoothQuant [87] apply per-channel scaling and quantization to model weights to better preserve outliers. (3) **Reconstruction-based approaches** such as AWQ [64] and GPTQ [39] are also applied on a per-channel or per-block level, but their quantization objective is to minimize the original high-precision weights' reconstruction error: While capable of quantizing to very low precisions such as INT4 and INT3, they incur higher computational overhead versus alternatives.

Quantization methods operate at a per-block level, since it allows them to be efficient, permitting parallelization over multiple threads (including GPUs), and requiring less metadata compared to quantizing every element separately. We will later show how QStore utilizes this standardized block-based approach to be generally applicable to various quantization methods (§4.2).

QStore *vs Lossy Quantization.* Quantization is inherently a lossy transformation aimed at reducing model complexity. QStore takes an orthogonal approach to model storage by taking the quantized and unquantized models as input, and subsequently performs *lossless* compression to store them efficiently into a unified format. While we focus on storing a pair of models at two specific precisions (e.g. FP16/BF16, INT8) in this paper for brevity, our approach can be generalized to other datatypes (e.g., INT4) and to more-than-two model chains, which we discuss in §5 and show in §6.

2.2 Data Compression

Model hosting platforms (e.g., HuggingFace [84]) store foundation models in wrapper formats such as Safetensors [9, 25], ONNX [6], TensorFlow, and SavedModel [8] allowing transparent storage of information such as tensor names and quantization information along with model weights. However, compression—another approach explored for reducing model storage costs—is currently unutilized by these formats. We discuss the pros and cons of various compression techniques applicable to foundation models.

Generic Compression. Standard compressors such as GZip [1], ZSTD [4], LZ4 [3] are applicable to model weights, treating the weight (sequence) as a generic byte stream and are agnostic to specific structural and numerical properties of model weights. ALP [20] targets general 32-bit and 64-bit floats, so their method cannot be directly applied to 16-bit models. Generic methods do not achieve optimal compression ratios on model weights due to their high entropy (e.g., the mantissa bits of floats [42]) rendering common techniques such as dictionary coding [76] ineffective.

Compression for ML Models. Recently, approaches have been proposed for specifically compressing ML models: ZipNN [42] compresses BF16 weights by reordering 16-bit floats into 2 byte streams which are separately compressed with Huffman coding. NeuZip [41] uses lossy compression to speed up inference by quantizing mantissa bits, and losslessly compressing exponent bits with an entropy coder to speed up training. Huf-LLM [91] uses hardware-aware Huffman compression, breaking 16-bit values into non-standard bit-level patterns and separately compressing streams for fast inference. MLWeaving stores models in bit-major order, and utilizes FPGA acceleration to enable models to be loaded at arbitrary precisions.

QStore *(**ours**): Joint Compression.* Unlike existing compression methods, QStore targets the joint storage of a pair of models, and achieves higher overall compression ratios for the model pair versus compressing them separately via delta storage-like techniques [45] for storing the conditional information (§6). Additionally, QStore runs purely on CPU, and does not depend on the availability of specific architectures (e.g., systolic arrays, TPUs/NPUs, FPGAs) required by some of the aforementioned methods.

3 QSTORE OVERVIEW

This section presents the QStore pipeline. QStore is a format that efficiently stores a pair of high and low-precision models: first, the model pair is compressed using an encoder into the unified QStore format. Then, a decoder is applied onto the QStore files to losslessly retrieve the high or low-precision model (or both).

QStore Input. QStore's encoding takes in the weights of the high and low-precision model versions (w and $Q(w)$, respectively) as input. QStore does not impose restrictions on the input format; our approach can work within any format implementation as long as it stores tensors separately (e.g., safetensors [9], PyTorch pickle objects [2], TensorFlow SavedModel [8], etc. are acceptable).

Encoding. QStore's utilizes an encoder to encode model weights: the encoder first compresses the weights of the low-precision model, then compresses the conditional information present only in the high-precision model (i.e., 'extra information', §1) (§4.2).

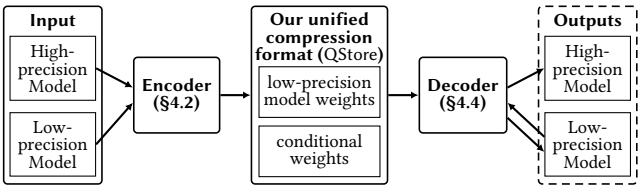


Figure 2: QStore **pipeline. A high and low-precision model pair is encoded into a unified, storage-friendly format (**QStore**), from which both models can be efficiently retrieved.**

Format. The unified QStore format, generated by encoding the input model pair, consists of two files: the compressed low-precision weights and the compressed conditional information (§4.3).

Decoding. QStore's utilizes a decoder to act on the two files within QStore to reconstruct either the low or high-precision model (or both): If the user requests the low-precision model, the decoder is invoked on the compressed quantized model weights to reconstruct it. If (additionally) the high-precision model is requested, the decoder is invoked on the newly decompressed low-precision model weights and the compressed conditional information (§4.4).

4 QSTORE: UNIFIED FORMAT

This section details the QStore format and its encoding and decoding algorithms. We describe our intuition to encode conditional information in §4.1, the encoding of a model pair into the QStore format in §4.2, the QStore format itself in §4.3, and decoding to obtain the original high or low-precision weights (or both) in §4.4.

4.1 Key Intuition

This section describes our intuition for compressing conditional information present in the high-precision model but not in the low-precision model. Without loss of generality, we will be describing QStore's operations with a FP16/BF16 and INT8 model pair.

Conditional Information. Given a high and low-precision model pair, it is possible to derive the low-precision model from the high-precision model (e.g., via quantization). Hence, all information present in the low-precision model is contained within the high precision model. Given the weights of the high-precision model W and a quantization function Q that maps it to the corresponding quantized weights, we can model the information in the model pair:

$$H(W) = H(Q(W)) + H(W|Q(W)) \quad (1)$$

QStore aims to find an efficient bit-level representation corresponding to $H(Q(W)) + H(W|Q(W))$ in Eq. (1). Notably, the representation of the conditional data $W|Q(W)$ must be *lossless* regardless of the quantization function Q used, which QStore will not know in advance (i.e., prior to compression). In particular, given floating point W and quantized $Q(W)$, the key challenge is in finding overlapping bit-level patterns in dynamic-precision floating point data that is informed by the corresponding quantized data, which the remainder of this section will aim to address.

Grouping by Quantized Weight. Most common recent quantization schemes use a combination of scaling (e.g., normalizing weights

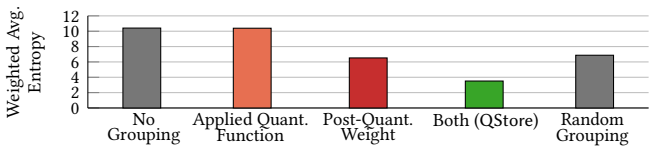


Figure 3: Weighted entropy of different grouping strategies on the Llama 3.1 8B Instruct model's 16-bit weights. QStore's combined grouping achieves high entropy reduction (hence compression ratio) versus alternative grouping strategies.

into a range) and rounding (§2.1). Given such quantization schemes, we observe that two floats that quantize to the same value (with the same quantization function, described shortly) can be expected to have more overlapping bits compared to two randomly selected floats, such as those that quantize to different values (Fig 3). Higher bit-level overlap between floats is directly correlated with compressibility (e.g., via entropy coding schemes); hence, QStore groups the high-precision (floats) weights by quantized value during encoding.

Grouping by Quantization Function. Recent popular quantization schemes apply multiple block-wise independent quantization functions to a single tensor (§2.1). For example, LLM.int8() [34] uses a different scaling factor to quantize each row (e.g., $Q_{row=i}(w_i) = round(\frac{128 w_i}{s_i})$, where s_i is row i's scaling factor). The quantization function is often chosen w.r.t. the weights; a common choice is the magnitude-based $s_i = abs(max(w_i))$ [34, 64]. Hence, the conditional information of a group of floating point weights w.r.t. their quantized integer weights $H(W|Q(W))$ changes with $Q(W)$. While grouping floats by applied quantization function alone achieves negligible entropy reduction (as intra-group float distributions are still largely random), we observe that a combined grouping of the applied quantization function and quantized weight value achieves significant compression benefits (e.g., versus grouping only by one criteria, or randomly grouping with the same group count, Fig 3).

4.2 Encoding to QStore

This section describes how a high and low-precision model pair is encoded into the QStore format. As described in §3, QStore's encoder compresses the low-precision model and the high-precision model's conditional information w.r.t. low-precision model (§4.1).

Encoding Quantized Weights. QStore's encoder utilizes an entropy coding scheme to compress the (quantized) weights of the low-precision model $Q(w)$. It follows zstd's approach [4] to divide $Q(w)$ into sequential, fixed-size chunks, on which per-chunk Huffman compression is applied for up to 12% size reduction (§6.2).

Encoding Conditional Information. QStore's encoder computes conditional information using weights of both the high and low-precision model (w and $Q(w)$) as input. Following intuition described in §4.1, the high-precision model weights w are first grouped according to applied quantization function (e.g., for LLM.int8() [34], each group will consist of tensors with the same applied scale value). Then, weights in each group are further divided into subgroups of weights quantizing to the same value. For example, in 4, rows w_1, w_3, and w_2 are quantized with distinct scale values (32 and 16), hence their weights are placed into groups 1 ($s_1 = s_3 = 32$) and 2 ($s_2 = 16$). In group 1, w_{11}, w_{13}, w_{32}, and w_{33} quantize to the same value (yellow) and are placed in one subgroup; w_{12} and w_{32} quantize to another value (blue) and are placed in another subgroup.

Per-subgroup compression. Similar to how we compress the low-precision quantized weights, QStore's conditional encoder then compresses conditional information using Huffman compression on a per-subgroup basis. If a chunk is not compressible enough (e.g., due to high entropy, or very few unique values in a subgroup), QStore skips encoding and stores that chunk uncompressed.

Remark. The combined size of QStore's compressed quantized weights and conditional information is much lower than the original uncompressed size of both models; in fact, QStore's size is close to *only* compressing the high-precision model (e.g., via ZipNN, §6.2); however, QStore additionally allows the low-precision model to be directly retrieved without requiring in-memory quantization (§6.6).

4.3 QStore Format

This section describes how QStore stores an encoded high and low-precision model pair. Each QStore model pair consists of two files—the compressed quantized weights and conditional information.

Compressed Quantized Weights. QStore stores compressed low-precision model weights alongside a header—chunk count, tensor dimensions, and per-chunk metadata of (1) whether compression was applied and (2) compressed and uncompressed chunk sizes.

Compressed Conditional Information. QStore stores conditional information following group (i.e., quantization function), then subgroup (i.e., post-quantization value) order. It maintains a header, which stores (1) group-to-position mappings in the original model (e.g., row number), and within each group, (2) the aforementioned per-subgroup data. Notably, despite QStore also reordering the weights in each group based on subgroups, it does not store per-subgroup (row) weight position mappings: this is because the quantized weights already contain the information, e.g., w_{13} assigned to group 1, subgroup 1 in Fig 4 can be inferred to be row w_1's 3^{rd} element based on the corresponding quantized weights in $Q_1(w_1)$.

4.4 Decoding from QStore

This section covers how a model pair stored with QStore can be losslessly decoded to retrieve the high and/or low-precision models.

Retrieving the Low-Precision Model. The model's quantized weights are encoded to QStore with per-chunk Huffman compression into a file (§4.2). Hence, directly loading the compressed quantized weights from QStore, and applying per-chunk huffman decompression allows the low-precision model to be retrieved losslessly.

Retrieving the High-Precision Model. As QStore stores the encoded conditional information for the high-precision model w.r.t. the low-precision model, it requires the low-precision model to be retrieved first. Then, QStore's decoder first decompresses the conditional information, which is applied onto the low-precision model weights to retrieve correct per-group weight ordering (§4.3. Finally, QStore uses the stored group-to-row mappings to losslessly reconstruct the high-precision model's weight tensor.

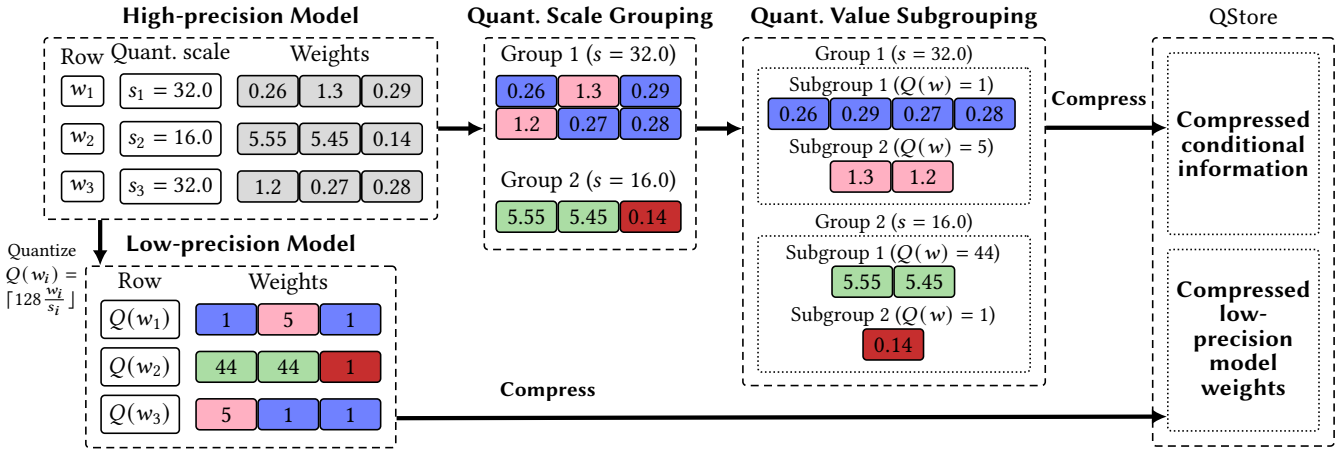


Figure 4: Compressing a tensor in the model pair with QStore. Weights in the high-precision model are grouped by the quantization function (scale) applied, then subgrouped by the post-quantization value from the low-precision model.

Remark. QStore's loading of the high or low-precision model is faster than loading the respective model uncompressed, and comparable to loading the respective model (separately) compressed with an off-the-shelf tool (e.g., ZipNN, §6.3). However, as QStore jointly stores the model pair, QStore achieves significant time savings for loading the low-precision model versus the common practice of loading the unquantized model, then quantizing it in memory (§6.6).

5 IMPLEMENTATION & DISCUSSION

Choice of Encoding Scheme. Our implementation of QStore uses the FiniteStateEntropy library's near-state-of-the-art Huffman encoding Huff0 [5]. However, other entropy-based encoding schemes can be used instead, such as the FiniteStateEntropy coder from the same library or non-Huffman methods. (e.g., arithmetic coding [15])

Efficient Decode Pipelining. We pipeline QStore's per-tensor decoding for model loading (§4.4), where one tensor's decompression overlaps with the next tensor's read. However, other parallelization strategies can be used in its place [62, 73, 75], such as completely parallelizing tensor read and decompression, which may bring larger benefits on specific hardware (e.g., local SSD [24, 74]).

Lazy Model Loading. As QStore's encoding and decoding of model pairs operate independently on each tensor, it can be naturally extended to support lazy loading (e.g., similar to Safetensors [9]). For lazy loading, we would not apply decode pipelining, and only read and decompress tensors when required; we defer detailed performance optimization and engineering to future work.

Generalizing to Multiple Precisions & Datatypes. While the use of model pairs is common, many modern deployments require models in more than two quantization formats (e.g., 4-bit mobile inference, 8-bit cloud inference, and 16-bit training). In such situations, all low precision models (4-bit, 8-bit) are created by quantizing the high precision (16-bit) model. QStore can be extended to store more-than-two model chains, only requiring that the same group size is used for the quantization to different precisions (verified in §6.7): for the prior example, QStore would store the INT4 model, the INT8 | INT4 conditional encoding, and the FP16 | INT8 conditional encoding. As mentioned in §1, this extension would especially benefit model storage hubs like HuggingFace [84] which can store multiple quantized representations of the same model with significantly lower storage cost versus separately storing precisions.

Table 1: Summary of models used for evaluation.

Model	Params.	Model Pair Size	Modality
Qwen 2 Audio [31]	7B	19.9 GB	Audio-Text
Mistral v0.3 [49]	7B	19.5 GB	Text
Llama 3.1 [40]	8B	19.5 GB	Text
Gemma 3 [78]	27B	72.7 GB	Image-Text
Qwen 2.5 VL [21]	32B	87.7 GB	Video-Image-Text
Deepseek Coder [97]	33B	91.9 GB	Text (Coding)

Compatibility with Precisions and Datatypes. While we evaluate QStore with LLM.int8() and GPT-Q [39] quantization and the FP16/BF16/INT8/INT4 datatypes (§6), QStore can be extended to support other datatypes and group-based quantization methods: QStore directly applies byte-level entropy coding for storage (§4.2); only the group-wise weight ordering (in the low-precision model) and conditional information are required to losslessly reconstruct the high-precision model (§4.4), both of which are datatype-agnostic.

6 EVALUATION

In this section, we empirically study the effectiveness of QStore's quantization-aware model storage. We make the following claims:

(1) **Effective Compression:** QStore achieves up to 2.2× compression ratio for storing a high and low-precision model pair—up to 1.6× better than the next best method. (§6.2)
(2) **Fast Retrieval:** A model pair stored with QStore can be loaded up to 1.8× faster versus alternative formats (§6.3).
(3) **Fast Storage:** A model pair can be stored with QStore up to 2.8× faster than uncompressed storage, and 1.7× faster versus alternative storage/compression methods (§6.4).

Deeper Performance analysis of QStore (Ours)

Table 2: QStore's storage cost (GBs) for storing a high and low precision model pair versus baselines. QStore achieves up to 2.2× and 1.6× space savings versus storing the models uncompressed (Safetensors) and the next best alternative (ZipNN + Zstd).

Method	Safetensors			lz4			Zstd			ZipNN (high prec) + Zstd (low prec)			QStore **(Ours)**		
Model	Low	High	**Total**	Low	High	**Total**	Low	High	**Total**	Low (Zstd)	High (ZipNN)	**Total**	Low	**High|Low**	**Total**
Qwen 2 Audio 7B	6.622	13.244	19.866	6.647	13.296	19.943	5.891	10.303	16.194	5.891	8.780	14.671	5.893	3.572	**9.465**
Mistral v0.3 8B	6.500	13.000	19.500	6.524	13.051	19.575	5.734	10.124	15.858	5.734	8.629	14.363	5.734	3.379	**9.113**
Llama 3.1 8B	6.500	13.000	19.500	6.523	13.051	19.574	5.746	10.083	15.829	5.746	8.629	14.375	5.746	3.295	**9.041**
Gemma 3 27B	24.223	48.446	72.669	24.317	48.636	72.954	21.230	37.450	58.680	21.230	32.153	53.383	21.230	11.848	**33.078**
Qwen 2.5 VL 32B	29.248	58.496	87.744	29.164	58.718	87.882	25.226	44.869	70.095	25.226	39.013	64.239	25.295	13.940	**39.235**
Deepseek Coder 33B	30.621	61.243	91.864	30.697	61.483	92.180	27.030	47.959	74.989	27.030	40.572	67.602	27.041	14.548	**41.589**

(1) **Effective Under Constrained Bandwidth:** QStore enables up to 2.2× faster model loading versus loading uncompressed models under I/O-constrained scenarios (§6.5).
(2) **Comparison with Online Quantization:** The low-precision model can be loaded from QStore up to 2.5× faster versus loading and quantizing a high-precision model (§6.6).
(3) **Effective Compression of Multi-Level Model Chains:** QStore achieves up to 2.46× compression ratio when storing model chains with more than two precision levels; this is up to 1.78× better than the next best alternative. (§6.7)

6.1 Experimental Setup

Dataset (Table 1). We select 6 popular foundation models for evaluation, which we further divide into 3 'small' (<20B parameters) and 3 'large' (≥20B parameters) models. For each model, we create a high and low-precision model pair consisting of the (1) original BF16 model and (2) quantized INT8 model (via LLM.int8() [34], unless otherwise stated, e.g., in §6.6 and §6.7) weights.

Methods. We evaluate QStore against existing tools and methods capable of storing the high and low-precision model pairs:

- Safetensors **[9]:** The default uncompressed model storage format [16, 18] of HuggingFace's transformers library [84].
- lz4 **[3]:** We use the default compression level of 1.
- Zstd **[4]:** We use a compression level of 2.
- ZipNN **[42]**: A Huffman-based compression algorithm that targets compression of 16-bit model weights. Since it cannot compress 8-bit weights, in order to compare the storage cost of both precisions, we use ZipNN for high precision and the best alternative baseline (Zstd) for low precision.

We implement all methods to sequentially process tensors to and from a single file for model saving and loading. Tensor read/write and (de)compression are pipelined to overlap I/O and compute (§5).

Environment. We use an Azure Standard E80is (Intel(R) Xeon Platinum 8272CL, 64-bit, little-endian) VM instance with 504GB RAM. We read and write (compressed) model data to and from local SSD for all methods. The disk read and write speeds are 1.5 GB/s and 256.2 MB/s, respectively,[1] with read latency of 7.49ms.[2]

Time Measurements. We measure (1) *save time* to compress and store a model pair onto storage, and (2) *load time* to read and decompress the selected model(s) into memory. We force full data writing (via `sync` [14]) and reading during model saving and loading. We perform data reading and writing with a single thread and compression/decompression with 48 threads for all methods.

Reproducibility. Our implementation of QStore and experiment scripts can be found in our Github repository.

6.2 QStore Saves Model Storage Cost

This section studies QStore's model storage cost savings. We store model pairs to disk with each method, and compare the resulting on-disk file sizes of QStore versus alternative methods in Table 2.

QStore's file size is consistently smallest, and is up to 2.2× and 1.6× smaller versus Safetensors (uncompressed) and next best compression method (ZipNN + Zstd), respectively. As hypothesized in §2.2, Zstd and lz4 achieve suboptimal compression ratios due to the traditional compression techniques they utilize being ineffective on noisy, high-entropy model tensor data—notably, lz4 achieves *no* benefits storage-wise. While ZipNN effectively compresses just the high-precision model into a size smaller (up to 6.3%) versus QStore's model pair, its specialization for the FP16/BF16 data formats leads to it having to rely on a different, less effective compression algorithm (Zstd) for storing the low-precision model, leading to QStore's stored model pair being 1.6× smaller than the pair stored with ZipNN + Zstd. QStore's high compression ratio translates to significant (52%-55%) space savings across model sizes: storing the Deepseek Coder's model pair with QStore takes only 42GB versus the 92GB of storing the models as is without compression.

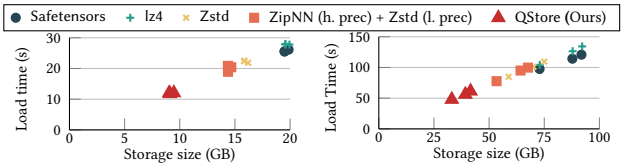
Effective Conditional Information Storage. QStore's compressed conditional information (*High|Low*) only takes up to 39% of the total size, and accordingly contributes only up to 40% of the model pair loading time (Fig 5, Fig 6) across all 6 models. This shows QStore's conditional encoding's effectiveness in reducing storage and load time redundancies incurred by the typical approach of users storing and using both the high and low-precision models as is (§1).

6.3 QStore Saves Model Load Time

We investigate QStore's time savings for loading a model pair. We store the model pair using each method, then measure the time taken for loading one or both models from storage into memory.

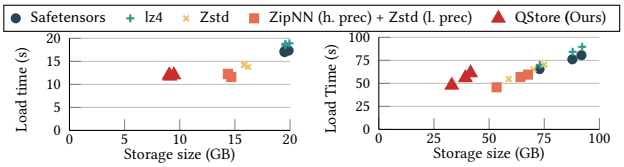
We report results for loading both models in Fig 5. QStore saves significant time in cases where simultaneous access to both models (e.g., model cascade and chaining §1 or interactive computing [57–60]) is required; it loads the model pair up to 2.2× and 1.8× faster than separately loading the two models stored without compression

[1] Measured with *dd* with 1*MB* block size, reading 1024 blocks from a model file.
[2] Measured with `iostat -x`.



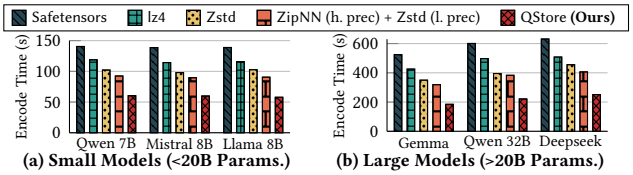
(a) Small Models (<20B Params.) (b) Large Models (>20B Params.)

Figure 5: Decoding time when we need both high-precision and low-precision models, versus storage costs: QStore's loads the model pair up to 2.2× and 1.8× faster versus loading uncompressed models and compression baselines.



(a) Small Models (<20B Params.) (b) Large Models (>20B Params.)

Figure 6: Decoding time when we only need high-precision models, versus storage costs: QStore loads the model up to 1.4× faster versus loading uncompressed, comparable (±5%) to loading with a specialized compression algorithm (ZipNN).



(a) Small Models (<20B Params.) (b) Large Models (>20B Params.)

Figure 7: QStore's encoding time for saving a model pair versus baselines. QStore saves the models 2.8× faster versus uncompressed storage, and is up to 1.7× faster than storing the models with an applicable compression algorithm.

(Safetensors) or with an applicable compression algorithm (Zstd), respectively; one significant cause of faster loading is the QStore's model pair size being significantly smaller than that of by separately storing the two models with alternative approaches (§6.2), which saves I/O costs especially under constrained bandwidths (§6.5).

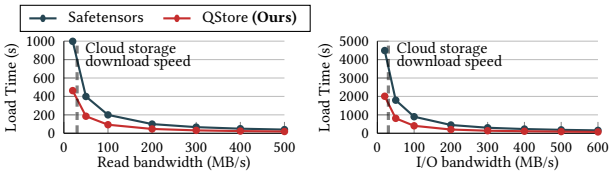
Comparable High-Precision Model Load Time (Fig 6). QStore loads the high-precision model up to 1.4× faster versus loading uncompressed (Safetensors), and has comparable loading times versus loading it with a specialized method (±5%, ZipNN). QStore still saves storage space in this case as it jointly stores the low-precision model (when it is needed): While storing the low-precision model is not required for the latter to load the high-precision model (unlike QStore), the alternative of not storing the low-precision model can result in high online quantization costs (§6.6).

6.4 QStore Enables Faster Model Storage

This section investigates QStore's time for storing model pairs. We measure the time taken for storing a model pair from memory into storage with the QStore format versus alternative methods.

Table 3: Average bits per weight to store each model pair.

Model	Safetensors	Zstd	QStore **(Ours)**
Qwen 2 Audio [31]	24	19.564	**11.434**
Mistral v0.3 [49]	24	19.518	**11.216**
Llama 3.1 [40]	24	19.482	**11.127**
Gemma 3 [78]	24	19.379	**10.925**
Qwen 2.5 VL [21]	24	19.173	**10.732**
Deepseek Coder [97]	24	19.591	**10.865**



(a) Llama 3.1 8B (b) Qwen 2.5 VL 32B

Figure 8: QStore's decoding time (secs) versus read bandwidth for two selected models. QStore's smaller incurred storage size saves loading time by 2.2× at lower bandwidths.

We report results in Fig 7. QStore's model pair storing time is up to 1.7× and 2.8× faster compared to the next best compression scheme and non-compression method, respectively. Notably, given each model pair, uncompressed methods need to write 24 (16 + 8) bits per model weight to disk, whereas QStore significantly reduces the bit count to 10.7-11.5 (Table 3), which is also smaller than the 19.1-19.6 bits incurred by separately compressing both models with Zstd. Expectedly, QStore's number of incurred bits is in alignment with QStore's high compression ratio (Table 2).

6.5 High Savings on Constrained Bandwidths

This section studies the effect of I/O bandwidth on QStore's time savings. We perform a parameter sweep on bandwidth from SSD by throttling with `systemd-run [19]` (verified using `iostat` [17]) and measure the time to load a model pair stored with QStore vs uncompressed storage (Safetensors) at various bandwidths (Fig 8).

While QStore is faster than uncompressed loading at all bandwidths, the speedup increases from 1.7× (500MB/s) to 2.1× and 2.2× in the lowest bandwidth settings (20MB/s) for the small Llama 3.1 model and large Qwen 2.5 VL model, respectively. Notably, the absolute time saving of QStore versus uncompressed is 2483 seconds for loading the Qwen model at 20MB/s; this time saving significantly improves user experience in the common scenario where models are downloaded from cloud storage with limited network bandwidth (typically 30MB/s [42], grey vertical lines in Fig 8).

6.6 Faster Versus Online Quantization

This section studies QStore's time savings for loading the low-precision model versus online quantization. We create QStore model pairs for the same high-precision model quantized with different quantization algorithms (GPT-Q and LLM.int8()), then measure time taken for QStore to load the (i) low-precision model versus (ii) loading the high-precision model and then quantizing it on-the-fly.

We report results in Fig 9. QStore saves up to 47.3× loading time versus online quantization with the nonlinear, 2^{nd} order method

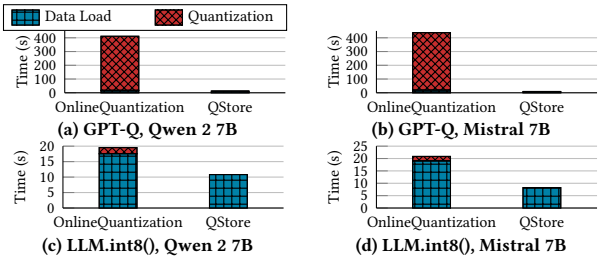


Figure 9: QStore's decoding via loading the low precision model saves up to 47.3× time versus loading then quantizing the high precision model on-the-fly.

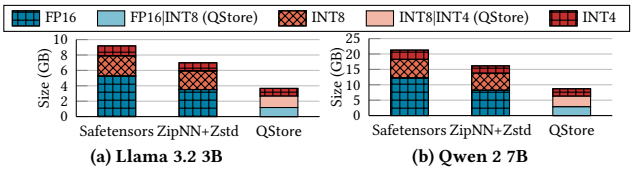


Figure 10: QStore's storage costs for a 3-model chain versus baselines. QStore's conditional compression of high-precision models w.r.t. low-precision models saves up to 2.46× and 1.78× space versus uncompressed storage and ZipNN +Zstd.

GPT-Q (Fig 9a), whose complex and loss-aware quantization takes ~7 minutes on the Mistral 7B model. Even versus the much faster quantization method LLM.int8() (as implemented in bitsandbytes), which only performs grouping, scaling and rounding in < 2 seconds on Mistral 7B, QStore still saves up to 2.53× in end-to-end model loading time as QStore's compressed model pair is significantly smaller than the uncompressed high-precision model (Fig 9c).

6.7 Handles Multi-Level Model Chains

This section studies QStore's generalizability to more-than-two model chains. We create a 3-model chain of FP16, INT8, and INT4 precisions using GPT-Q quantization with group size of 512, then compare storage cost of storing the model chain with QStore (following procedures described in §5) versus alternative methods.

We report results in Fig 10. QStore achieves up to 2.46× and 1.78× savings versus Safetensors and ZipNN +Zstd, respectively. QStore's largest savings come from the FP16 and INT8 models: this is because QStore stores the INT8 model as a conditional of the INT4 (i.e., INT8|INT4), then stores the FP16 model as a conditional of the INT8 (i.e., FP16|INT8). By bypassing the storage of redundant information (§6.2), QStore achieves up to 4.46× and 2.98× savings for the FP16 portion alone versus Safetensors and ZipNN, respectively. Note that these savings are even greater than the model pair storage case because of the nature of our conditional encodings.

7 RELATED WORK

Matrix Compression for Machine Learning. Matrix compression has been extensively explored for ML model weights [22, 36, 54, 56, 72]. These works exploit specific properties present in ML model weight matrices such as inherent sparsity, column correlations, and low distinct value counts [92] to apply custom compression schemes more effective than off-the-shelf compression algorithms, aiming to fit the weight matrices in memory for performing efficient computations directly on the compressed data [56, 72]. Grouping techniques are prevalent in these works, for example, CLA [36] jointly performs column grouping and selection of per-column-group compression algorithms to apply, SLACID [54] and [22] performs per-group compression on a matrix block granularity, and TOC [56] applies compression on minibatches of input data. In comparison, QStore exploits the natural grouping incurred by quantization schemes to perform joint compression on multiple models of varying precision on LLM weights, which are more difficult to compress due to different properties such as inherent randomness and density [47].

Mixed-Strategy Compression. Works have explored applying different compression algorithms on different data subsets to achieve higher compression rates [44, 45, 48, 71, 90]. DeepSqueeze uses autoencoders to store tuples in tabular data, then applies per-dataset code-to-tuple mappings with the best compression algorithm found by brute-force (e.g., run-length, delta compression). HIRE [90] uses reinforcement learning to estimate the best compression algorithm for each point in a timeseries. While QStore orthogonally studies multi-precision model storage, incorporate these works' techniques into QStore such as replacing our current Huffman compression on a per-model basis (§4.2) can be valuable future work.

Adaptive Data Formats for Machine Learning. There exists works that adapt to the input data for ML model weights [32, 33, 55, 66, 83] and learned indexes [28–30, 61]. These works have influenced recent quantization schemes (e.g., LLM.int8() [34]), focusing on dynamically selecting the per-group exponent bit count for vectors based on data distribution (e.g., max/min values [32, 83]). Groups are often user-defined (e.g., tensors [32, 83] or group size n [33]), while per-group exponent bit count selection can be done either reactively (e.g., increasing on overflow [32, 83]) or proactively (e.g., tracking value trend during model training [55]). QStore complements these methods, being in principle applicable to these works' group-based techniques (now also seen in quantization) to reduce data sizes.

8 CONCLUSION

In this paper, we introduced QStore, a unified file format for storing a high and low-precision model pair. QStore defines a novel representation for storing conditional information present in the high-precision model but not in the low-precision model. For storage, QStore stores the low-precision model and applies novel grouping techniques on the conditional information to achieve high compression ratios. Then, a model pair stored in the QStore format can be losslessly decoded to load the low or high-precision model (or both). We showed via experimentation that QStore reduces the storage footprint of model pairs by up to 2.2× while enabling up to 2× and 1.6× faster model pair saving and loading versus existing approaches, respectively, while also generalizing to a wide range of datatypes, quantization methods, and more-than-two model chains.

9 ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under Awards #2440498 and #2312561, and by the National Center for Supercomputing Applications.

REFERENCES

[1] 1992. *GZIP.* Retrieved Apr 18, 2025 from https://www.gnu.org/software/gzip/

[2] 1996. *Pickle.* Retrieved Apr 18, 2025 from https://github.com/python/cpython/blob/main/Lib/pickle.py

[3] 2011. *LZ4.* Retrieved Apr 18, 2025 from https://github.com/lz4/lz4

[4] 2016. *ZSTD.* Retrieved Apr 18, 2025 from https://github.com/facebook/zstd

[5] 2017. *fse.* Retrieved Apr 18, 2025 from https://github.com/Cyan4973/FiniteStateEntropy

[6] 2017. *ONNX.* Retrieved Apr 18, 2025 from https://github.com/onnx/onnx

[7] 2019. *BF16 - The Secret to High Performance on Cloud TPUs.* Retrieved Apr 18, 2025 from https://cloud.google.com/blog/products/ai-machine-learning/bfloat16-the-secret-to-high-performance-on-cloud-tpus

[8] 2019. *Is running a quantized model worth it?* Retrieved Apr 18, 2025 from https://www.tensorflow.org/guide/saved_model

[9] 2021. *Safetensors.* Retrieved Apr 18, 2025 from https://github.com/huggingface/safetensors

[10] 2023. *Is running a quantized model worth it?* Retrieved Apr 18, 2025 from https://www.reddit.com/r/LocalLLaMA/comments/13aidav/is_running_quantized_but_bigger_model_worth_it/

[11] 2024. *NVIDIA TensorRT Accelerates Stable Diffusion Nearly 2x Faster with 8-bit Post-Training Quantization.* Retrieved Apr 18, 2025 from https://developer.nvidia.com/blog/tensorrt-accelerates-stable-diffusion-nearly-2x-faster-with-8-bit-post-training-quantization/

[12] 2024. *Ollama.* Retrieved Apr 18, 2025 from https://ollama.com/search

[13] 2024. *Serving Quantized LLMs on NVIDIA H100 Tensor Core GPUs.* Retrieved Apr 18, 2025 from https://www.databricks.com/blog/serving-quantized-llms-nvidia-h100-tensor-core-gpus

[14] 2024. *Sync.* Retrieved Apr 18, 2025 from https://man7.org/linux/man-pages/man2/sync.2.html

[15] 2025. *Arithmetic Coding.* Retrieved Apr 18, 2025 from https://en.wikipedia.org/wiki/Arithmetic_coding

[16] 2025. *How to load safetensors without lazy loading.* Retrieved Apr 18, 2025 from https://github.com/huggingface/safetensors/issues/577

[17] 2025. *iostat man page.* Retrieved Apr 18, 2025 from https://man7.org/linux/man-pages/man1/iostat.1.html

[18] 2025. *Source code for safetensors load() function.* Retrieved Apr 18, 2025 from https://github.com/huggingface/safetensors/blob/7d5af853631628137a79341ddc5611d18a17f3fe/bindings/python/py_src/safetensors/mlx.py#L74

[19] 2025. *systemd-run man page.* Retrieved Apr 18, 2025 from https://man.archlinux.org/man/systemd-run.1.en

[20] Azim Afroozeh, Leonardo X Kuffo, and Peter Boncz. 2023. Alp: Adaptive lossless floating-point compression. *Proceedings of the ACM on Management of Data* 1, 4 (2023), 1–26.

[21] Shuai Bai, Keqin Chen, Xuejing Liu, Jialin Wang, Wenbin Ge, Sibo Song, Kai Dang, Peng Wang, Shijie Wang, Jun Tang, Humen Zhong, Yuanzhi Zhu, Mingkun Yang, Zhaohai Li, Jianqiang Wan, Pengfei Wang, Wei Ding, Zheren Fu, Yiheng Xu, Jiabo Ye, Xi Zhang, Tianbao Xie, Zesen Cheng, Hang Zhang, Zhibo Yang, Haiyang Xu, and Junyang Lin. 2025. Qwen2.5-VL Technical Report. *arXiv preprint arXiv:2502.13923* (2025).

[22] Nathan Bell and Michael Garland. 2009. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the conference on high performance computing networking, storage and analysis*. 1–11.

[23] Maciej Besta, Nils Blach, Ales Kubicek, Robert Gerstenberger, Michal Podstawski, Lukas Gianinazzi, Joanna Gajda, Tomasz Lehmann, Hubert Niewiadomski, Piotr Nyczyk, et al. 2024. Graph of thoughts: Solving elaborate problems with large language models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 38. 17682–17690.

[24] Zhen Cao, Vasily Tarasov, Hari Prasath Raman, Dean Hildebrand, and Erez Zadok. 2017. On the performance variation in modern storage stacks. In *15th USENIX conference on file and storage technologies (FAST 17)*. 329–344.

[25] Beatrice Casey, Kaia Damian, Andrew Cotaj, and Joanna Santos. 2025. An Empirical Study of Safetensors' Usage Trends and Developers' Perceptions. *arXiv preprint arXiv:2501.02170* (2025).

[26] Sapana Chaudhary, Ujwal Dinesha, Dileep Kalathil, and Srinivas Shakkottai. 2024. Risk-Averse Fine-tuning of Large Language Models. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=1BZKqZphsW

[27] Lingjiao Chen, Matei Zaharia, and James Zou. 2023. Frugalgpt: How to use large language models while reducing cost and improving performance. *arXiv preprint arXiv:2305.05176* (2023).

[28] Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. 2022. Automatically finding optimal index structure. *arXiv preprint arXiv:2208.03823* (2022).

[29] Supawit Chockchowwat, Wenjie Liu, and Yongjoo Park. 2023. Airindex: versatile index tuning through data and storage. *Proceedings of the ACM on Management of Data* 1, 3 (2023), 1–26.

[30] Supawit Chockchowwat, Chaitanya Sood, and Yongjoo Park. 2022. Airphant: Cloud-oriented document indexing. In *2022 IEEE 38th International Conference on Data Engineering (ICDE)*. IEEE, 1368–1381.

[31] Yunfei Chu, Jin Xu, Xiaohuan Zhou, Qian Yang, Shiliang Zhang, Zhijie Yan, Chang Zhou, and Jingren Zhou. 2023. Qwen-audio: Advancing universal audio understanding via unified large-scale audio-language models. *arXiv preprint arXiv:2311.07919* (2023).

[32] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2014. Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024* (2014).

[33] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, et al. 2020. Pushing the limits of narrow precision inferencing at cloud scale with microsoft floating point. *Advances in neural information processing systems* 33 (2020), 10271–10281.

[34] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. 2022. Gpt3. int8 (): 8-bit matrix multiplication for transformers at scale. *Advances in neural information processing systems* 35 (2022), 30318–30332.

[35] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. 2023. Qlora: Efficient finetuning of quantized llms. *Advances in neural information processing systems* 36 (2023), 10088–10115.

[36] Ahmed Elgohary, Matthias Boehm, Peter J Haas, Frederick R Reiss, and Berthold Reinwald. 2016. Compressed linear algebra for large-scale machine learning. *Proceedings of the VLDB Endowment* 9, 12 (2016), 960–971.

[37] Hanxi Fang, Supawit Chockchowwat, Hari Sundaram, and Yongjoo Park. 2025. Enhancing Computational Notebooks with Code+ Data Space Versioning. In *Proceedings of the 2025 CHI Conference on Human Factors in Computing Systems*. 1–17.

[38] Hanxi Fang, Supawit Chockchowwat, Hari Sundaram, and Yongjoo Park. 2025. Large-scale Evaluation of Notebook Checkpointing with AI Agents. In *Proceedings of the Extended Abstracts of the CHI Conference on Human Factors in Computing Systems*. 1–8.

[39] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate Post-training Compression for Generative Pretrained Transformers. *arXiv preprint arXiv:2210.17323* (2022).

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

[41] Yongchang Hao, Yanshuai Cao, and Lili Mou. 2024. NeuZip: Memory-Efficient Training and Inference with Dynamic Compression of Neural Networks. arXiv:2410.20650 [cs.LG] https://arxiv.org/abs/2410.20650

[42] Moshik Hershcovitch, Andrew Wood, Leshem Choshen, Guy Girmonsky, Roy Leibovitz, Ilias Ennmouri, Michal Malka, Peter Chin, Swaminathan Sundararaman, and Danny Harnik. 2024. ZipNN: Lossless Compression for AI Models. arXiv:2411.05239 [cs.LG] https://arxiv.org/abs/2411.05239

[43] Nhut-Minh Ho and Weng-Fai Wong. 2017. Exploiting half precision arithmetic in Nvidia GPUs. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–7. https://doi.org/10.1109/HPEC.2017.8091072

[44] Shengya Huang, Zhaoheng Li, Derek Werner, and Yongjoo Park. 2025. MojoFrame: Dataframe Library in Mojo Language. *arXiv preprint arXiv:2505.04080* (2025).

[45] Amir Ilkhechi, Andrew Crotty, Alex Galakatos, Yicong Mao, Grace Fan, Xiran Shi, and Ugur Çetintemel. 2020. Deepsqueeze: Deep semantic compression for tabular data. In *Proceedings of the 2020 ACM SIGMOD international conference on management of data*. 1733–1746.

[46] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. 2018. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2704–2713.

[47] Ajay Jaiswal, Shiwei Liu, Tianlong Chen, Zhangyang Wang, et al. 2023. The emergence of essential sparsity in large pre-trained models: The weights that matter. *Advances in Neural Information Processing Systems* 36 (2023), 38887–38901.

[48] Ipoom Jeong, Jinghan Huang, Chuxuan Hu, Dohyun Park, Jaeyoung Kang, Nam Sung Kim, and Yongjoo Park. 2025. UPP: Universal Predicate Pushdown to Smart Storage. In *Proceedings of the 52nd Annual International Symposium on Computer Architecture*. 419–433.

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

[50] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL] https://arxiv.org/abs/2310.06825

[51] Renren Jin, Jiangcun Du, Wuwei Huang, Wei Liu, Jian Luan, Bin Wang, and Deyi Xiong. 2024. A comprehensive evaluation of quantization strategies for large language models. In *Findings of the Association for Computational Linguistics ACL 2024*. 12186–12215.

[52] Dhiraj Kalamkar, Dheevatsa Mudigere, Naveen Mellempudi, Dipankar Das, Kunal Banerjee, Sasikanth Avancha, Dharma Teja Vooturi, Nataraj Jammalamadaka, Jianyu Huang, Hector Yuen, Jiyan Yang, Jongsoo Park, Alexander Heinecke, Evangelos Georganas, Sudarshan Srinivasan, Abhisek Kundu, Misha Smelyanskiy, Bharat Kaul, and Pradeep Dubey. 2019. A Study of BFLOAT16 for Deep Learning Training. arXiv:1905.12322 [cs.LG] https://arxiv.org/abs/1905.12322

[53] Jaeyoung Kang, Qirong Xia, Ipoom Jeong, Yongjoo Park, and Nam Sung Kim. 2025. Intel® in-Memory Analytics Accelerator: Performance Characterization and Guidelines. In *2025 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 1–13.

[54] David Kernert, Frank Köhler, and Wolfgang Lehner. 2014. SLACID-sparse linear algebra in a column-oriented in-memory database system. In *Proceedings of the 26th International Conference on Scientific and Statistical Database Management*. 1–12.

[55] Urs Köster, Tristan Webb, Xin Wang, Marcel Nassar, Arjun K Bansal, William Constable, Oguz Elibol, Scott Gray, Stewart Hall, Luke Hornof, et al. 2017. Flexpoint: An adaptive numerical format for efficient training of deep neural networks. *Advances in neural information processing systems* 30 (2017).

[56] Fengan Li, Lingjiao Chen, Yijing Zeng, Arun Kumar, Xi Wu, Jeffrey F Naughton, and Jignesh M Patel. 2019. Tuple-oriented compression for large-scale mini-batch stochastic gradient descent. In *Proceedings of the 2019 International Conference on Management of Data*. 1517–1534.

[57] Zhaoheng Li, Supawit Chockchowwat, Hanxi Fang, and Yongjoo Park. 2025. Demo of Kishu: Time-Traveling for Computational Notebooks. In *Companion of the 2025 International Conference on Management of Data*. 167–170.

[58] Zhaoheng Li, Supawit Chockchowwat, Hanxi Fang, Ribhav Sahu, Sumay Thakurdesai, Kantanat Pridaphatrakun, and Yongjoo Park. 2024. Demonstration of elasticnotebook: Migrating live computational notebook states. In *Companion of the 2024 International Conference on Management of Data*. 540–543.

[59] Zhaoheng Li, Supawit Chockchowwat, Ribhav Sahu, Areet Sheth, and Yongjoo Park. 2024. Kishu: Time-Traveling for Computational Notebooks. *Proceedings of the VLDB Endowment* 18, 4 (2024), 970–985.

[60] Zhaoheng Li, Pranav Gor, Rahul Prabhu, Hui Yu, Yuzhou Mao, and Yongjoo Park. 2023. ElasticNotebook: Enabling Live Migration for Computational Notebooks. *Proceedings of the VLDB Endowment* 17, 2 (2023), 119–133.

[61] Zhaoheng Li, Silu Huang, Wei Ding, Yongjoo Park, and Jianjun Chen. 2025. SIEVE: Effective Filtered Vector Search with Collection of Indexes. *Proceedings of the VLDB Endowment* 18, 11 (2025), 4723–4736.

[62] Zhaoheng Li, Xinyu Pi, and Yongjoo Park. 2023. S/C: speeding up data materialization with bounded memory. In *2023 IEEE 39th international conference on data engineering (ICDE)*. IEEE, 1981–1994.

[63] Baohao Liao, Shaomu Tan, and Christof Monz. 2023. Make pre-trained model reversible: From parameter to memory efficient fine-tuning. *Advances in Neural Information Processing Systems* 36 (2023), 15186–15209.

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

[65] Ye Lin, Yanyang Li, Tengbo Liu, Tong Xiao, Tongran Liu, and Jingbo Zhu. 2021. Towards fully 8-bit integer inference for the transformer model. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence* (Yokohama, Yokohama, Japan) *(IJCAI'20)*. Article 520, 7 pages.

[66] Naveen Mellempudi, Abhisek Kundu, Dipankar Das, Dheevatsa Mudigere, and Bharat Kaul. 2017. Mixed low-precision deep learning inference using dynamic fixed point. *arXiv preprint arXiv:1701.08978* (2017).

[67] Paulius Micikevicius, Dusan Stosic, Neil Burgess, Marius Cornea, Pradeep Dubey, Richard Grisenthwaite, Sangwon Ha, Alexander Heinecke, Patrick Judd, John Kamalu, et al. 2022. Fp8 formats for deep learning. *arXiv preprint arXiv:2209.05433* (2022).

[68] Sharan Narang, Gregory Diamos, Erich Elsen, Paulius Micikevicius, Jonah Alben, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. 2017. Mixed precision training. In *Int. Conf. on Learning Representation*. https://arxiv.org/pdf/1710.03740

[69] Nvidia. 2024. Working with Quantized Types. https://docs.nvidia.com/deeplearning/tensorrt/latest/inference-library/work-quantized-types.html.

[70] Stack Overflow. 2012. Byte vs Bit Access Speeds. https://stackoverflow.com/questions/7782110/is-it-fastest-to-access-a-byte-than-a-bit-why.

[71] Weston Pace, Chang She, Lei Xu, Will Jones, Albert Lockett, Jun Wang, and Raunak Shah. 2025. Lance: Efficient Random Access in Columnar Storage through Adaptive Structural Encodings. arXiv:2504.15247 [cs.DB] https://arxiv.org/abs/2504.15247

[72] Y Saad and K SPARS. 1990. A basic tool kit for sparse matrix computations. *RIACS, NA SA Ames Research Center, TR90-20, Moffet Field, CA* (1990).

[73] Eric R. Schendel, Saurabh V. Pendse, John Jenkins, David A. Boyuka, Zhenhuan Gong, Sriram Lakshminarasimhan, Qing Liu, Hemanth Kolla, Jackie Chen, Scott Klasky, Robert Ross, and Nagiza F. Samatova. 2012. ISOBAR hybrid compression-I/O interleaving for large-scale parallel I/O optimization. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing* (Delft, The Netherlands) *(HPDC '12)*. Association for Computing Machinery, New York, NY, USA, 61–72. https://doi.org/10.1145/2287076.2287086

[74] Elizabeth AM Shriver, Christopher Small, and Keith A Smith. 1999. Why does file system prefetching work?. In *USENIX Annual Technical Conference, General Track*. 71–84.

[75] Evangelia Sitaridi, Rene Mueller, Tim Kaldewey, Guy Lohman, and Kenneth A Ross. 2016. Massively-parallel lossless data decompression. In *2016 45th International Conference on Parallel Processing (ICPP)*. IEEE, 242–247.

[76] Xiaoyun Sun, Larry Kinney, and Bapiraju Vinnakota. 2004. Combining dictionary coding and LFSR reseeding for test data compression. In *Proceedings of the 41st annual Design Automation Conference*. 944–947.

[77] Mohammadreza Tayaranian Hosseini, Alireza Ghaffari, Marzieh S. Tahaei, Mehdi Rezagholizadeh, Masoud Asgharian, and Vahid Partovi Nia. 2023. Towards Fine-tuning Pre-trained Language Models with Integer Forward and Backward Propagation. In *Findings of the Association for Computational Linguistics: EACL 2023*, Andreas Vlachos and Isabelle Augenstein (Eds.). Association for Computational Linguistics, Dubrovnik, Croatia, 1912–1921. https://doi.org/10.18653/v1/2023.findings-eacl.143

[78] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, Louis Rouillard, Thomas Mesnard, Geoffrey Cideron, Jean bastien Grill, Sabela Ramos, Edouard Yvinec, Michelle Casbon, Etienne Pot, Ivo Penchev, Gaël Liu, Francesco Visin, Kathleen Kenealy, Lucas Beyer, Xiaohai Zhai, Anton Tsitsulin, Robert Busa-Fekete, Alex Feng, Noveen Sachdeva, Benjamin Coleman, Yi Gao, Basil Mustafa, Iain Barr, Emilio Parisotto, David Tian, Matan Eyal, Colin Cherry, Jan-Thorsten Peter, Danila Sinopalnikov, Surya Bhupatiraju, Rishabh Agarwal, Mehran Kazemi, Dan Malkin, Ravin Kumar, David Vilar, Idan Brusilovsky, Jiaming Luo, Andreas Steiner, Abe Friesen, Abhanshu Sharma, Abheesht Sharma, Adi Mayrav Gilady, Adrian Goedeckemeyer, Alaa Saade, Alex Feng, Alexander Kolesnikov, Alexei Bendebury, Alvin Abdagic, Amit Vadi, András György, André Susano Pinto, Anil Das, Ankur Bapna, Antoine Miech, Antoine Yang, Antonia Paterson, Ashish Shenoy, Ayan Chakrabarti, Bilal Piot, Bo Wu, Bobak Shahriari, Bryce Petrini, Charlie Chen, Charline Le Lan, Christopher A. Choquette-Choo, CJ Carey, Cormac Brick, Daniel Deutsch, Danielle Eisenbud, Dee Cattle, Derek Cheng, Dimitris Paparas, Divyashree Shivakumar Sreepathihalli, Doug Reid, Dustin Tran, Dustin Zelle, Eric Noland, Erwin Huizenga, Eugene Kharitonov, Frederick Liu, Gagik Amirkhanyan, Glenn Cameron, Hadi Hashemi, Hanna Klimczak-Plucińska, Harman Singh, Harsh Mehta, Harshal Tushar Lehri, Hussein Hazimeh, Ian Ballantyne, Idan Szpektor, Ivan Nardini, Jean Pouget-Abadie, Jetha Chan, Joe Stanton, John Wieting, Jonathan Lai, Jordi Orbay, Joseph Fernandez, Josh Newlan, Ju yeong Ji, Jyotinder Singh, Kat Black, Kathy Yu, Kevin Hui, Kiran Vodrahalli, Klaus Greff, Linhai Qiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Noy, Nathan Byrd, Nick Roy, Nikola Momchev, Nilay Chauhan, Noveen Sachdeva, Oskar Bunyan, Pankil Botarda, Paul Caron, Paul Kishan Rubenstein, Phil Culliton, Philipp Schmid, Pier Giuseppe Sessa, Pingmei Xu, Piotr Stanczyk, Pouya Tafti, Rakesh Shivanna, Renjie Wu, Renke Pan, Reza Rokni, Rob Willoughby, Rohith Vallu, Ryan Mullins, Sammy Jerome, Sara Smoot, Sertan Girgin, Shariq Iqbal, Shashir Reddy, Shruti Sheth, Siim Põder, Sijal Bhatnagar, Sindhu Raghuram Panyam, Sivan Eiger, Susan Zhang, Tianqi Liu, Trevor Yacovone, Tyler Liechty, Uday Kalra, Utku Evci, Vedant Misra, Vincent Roseberry, Vlad Feinberg, Vlad Kolesnikov, Woohyun Han, Woosuk Kwon, Xi Chen, Yinlam Chow, Yuvein Zhu, Zichuan Wei, Zoltan Egyed, Victor Cotruta, Minh Giang, Phoebe Kirk, Anand Rao, Kat Black, Nabila Babar, Jessica Lo, Erica Moreira, Luiz Gustavo Martins, Omar Sanseviero, Lucas Gonzalez, Zach Gleicher, Tris Warkentin, Vahab Mirrokni, Evan Senter, Eli Collins, Joelle Barral, Zoubin Ghahramani, Raia Hadsell, Yossi Matias, D. Sculley, Slav Petrov, Noah Fiedel, Noam Shazeer, Oriol Vinyals, Jeff Dean, Demis Hassabis, Koray Kavukcuoglu, Clement Farabet, Elena Buchatskaya, Jean-Baptiste Alayrac, Rohan Anil, Dmitry, Lepikhin, Sebastian Borgeaud, Olivier Bachem, Armand Joulin, Alek Andreev, Cassidy Hardin, Robert Dadashi, and Léonard Hussenot. 2025. Gemma 3 Technical Report. arXiv:2503.19786 [cs.CL] https://arxiv.org/abs/2503.19786

[79] The ModelScope Team. 2023. ModelScope: bring the notion of Model-as-a-Service to life. https://github.com/modelscope/modelscope.

[80] Hongyu Wang, Shuming Ma, Li Dong, Shaohan Huang, Huaijie Wang, Lingxiao Ma, Fan Yang, Ruiping Wang, Yi Wu, and Furu Wei. 2023. BitNet: Scaling 1-bit Transformers for Large Language Models. arXiv:arXiv:2310.11453

[81] Jason Wei, Maarten Bosma, Vincent Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V Le. 2022. Finetuned Language Models are Zero-Shot Learners. In *International Conference on Learning Representations*. https://openreview.net/forum?id=gEZrGCozdqR

[82] Jason Wei, Xuezhi Wang, Dale Schuurmans, Maarten Bosma, Fei Xia, Ed Chi, Quoc V Le, Denny Zhou, et al. 2022. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems* 35 (2022), 24824–24837.

[83] Darrell Williamson. 1991. Dynamically scaled fixed point arithmetic. In *[1991] IEEE Pacific Rim Conference on Communications, Computers and Signal Processing Conference Proceedings*. IEEE, 315–318.

[84] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander M. Rush. 2020. Transformers: State-of-the-Art Natural Language Processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Association for Computational Linguistics, Online, 38–45. https://www.aclweb.org/anthology/2020.emnlp-demos.6

[85] Mingyuan Wu, Jize Jiang, Haozhen Zheng, Meitang Li, Zhaoheng Li, Beitong Tian, Bo Chen, Yongjoo Park, Minjia Zhang, Chengxiang Zhai, et al. 2025. Cache-of-Thought: Master-Apprentice Framework for Cost-Effective Vision Language Model Inference. *arXiv preprint arXiv:2502.20587* (2025).

[86] Tongshuang Wu, Michael Terry, and Carrie Jun Cai. 2022. Ai chains: Transparent and controllable human-ai interaction by chaining large language model prompts. In *Proceedings of the 2022 CHI conference on human factors in computing systems*. 1–22.

[87] Guangxuan Xiao, Ji Lin, Mickael Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. Smoothquant: Accurate and efficient post-training quantization for large language models. In *International Conference on Machine Learning*. PMLR, 38087–38099.

[88] Jiajun Xu, Zhiyuan Li, Wei Chen, Qun Wang, Xin Gao, Qi Cai, and Ziyuan Ling. 2024. On-device language models: A comprehensive review. *arXiv preprint arXiv:2409.00088* (2024).

[89] Xiaohan Xu, Ming Li, Chongyang Tao, Tao Shen, Reynold Cheng, Jinyang Li, Can Xu, Dacheng Tao, and Tianyi Zhou. 2024. A survey on knowledge distillation of large language models. *arXiv preprint arXiv:2402.13116* (2024).

[90] Xinyang Yu, Yanqing Peng, Feifei Li, Sheng Wang, Xiaowei Shen, Huijun Mai, and Yue Xie. 2020. Two-level data compression using machine learning in time series database. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 1333–1344.

[91] Patrick Yubeaton, Tareq Mahmoud, Shehab Naga, Pooria Taheri, Tianhua Xia, Arun George, Yasmein Khalil, Sai Qian Zhang, Siddharth Joshi, Chinmay Hegde, and Siddharth Garg. 2025. Huff-LLM: End-to-End Lossless Compression for Efficient LLM Inference. arXiv:2502.00922 [cs.LG] https://arxiv.org/abs/2502.00922

[92] Ce Zhang, Arun Kumar, and Christopher Ré. 2016. Materialization optimizations for feature selection workloads. *ACM Transactions on Database Systems (TODS)* 41, 1 (2016), 1–32.

[93] Shuning Zhang and Yongjoo Park. [n.d.]. PBE Meets LLM: When Few Examples Aren't Few-Shot Enough. *Proceedings of the VLDB Endowment. ISSN* 2150 ([n. d.]), 8097.

[94] Xuechen Zhang, Zijian Huang, Ege Onur Taga, Carlee Joe-Wong, Samet Oymak, and Jiasi Chen. 2024. Efficient Contextual LLM Cascades through Budget-Constrained Policy Learning. *arXiv preprint arXiv:2404.13082* (2024).

[95] Qihuang Zhong, Liang Ding, Li Shen, Juhua Liu, Bo Du, and Dacheng Tao. 2024. Revisiting Knowledge Distillation for Autoregressive Language Models. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Lun-Wei Ku, Andre Martins, and Vivek Srikumar (Eds.). Association for Computational Linguistics, Bangkok, Thailand, 10900–10913. https://doi.org/10.18653/v1/2024.acl-long.587

[96] Zixuan Zhou, Xuefei Ning, Ke Hong, Tianyu Fu, Jiaming Xu, Shiyao Li, Yuming Lou, Luning Wang, Zhihang Yuan, Xiuhong Li, et al. 2024. A survey on efficient inference for large language models. *arXiv preprint arXiv:2404.14294* (2024).

[97] Qihao Zhu, Daya Guo, Zhihong Shao, Dejian Yang, Peiyi Wang, Runxin Xu, Y Wu, Yukun Li, Huazuo Gao, Shirong Ma, et al. 2024. DeepSeek-Coder-V2: Breaking the Barrier of Closed-Source Models in Code Intelligence. *arXiv preprint arXiv:2406.11931* (2024).