# **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. These applications often feature 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 the approach taken by many model providers such as HuggingFace and Ollama. However, this approach incurs excessive storage costs since a higher precision model (e.g., BF16) is a strict 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-precision and highprecision models separately, QStore stores low-precision model and only the residual information needed to reconstruct high-precision models. The size of residual information is significantly smaller than the original high-precision models, thus, achieving high savings in storage cost. Moreover, QStore does not compromise the speed of model loading. The low-precision models can be loaded quickly just like before. The high-precision models can also be reconstructed efficiently in memory by merging low-precision data and the residual with QStore's lightweight decoding logic. We evaluate QStore for compressing multiple precisions of popular foundation models, and show that QStore reduces overall storage footprint by up to  $2.2\times$  (45% of the original size) while enabling up to 1.7× and 1.8× faster model saving and loading versus existing approaches.

#### INTRODUCTION 1

Foundation models have become highly accessible to users thanks to the availability of model hosting platforms such as Hugging-Face [59], Ollama [12], and ModelScope [55]. Developers download the pre-trained models hosted on these platforms (e.g., from cloud storage), and then apply them to various tasks such as finetuning [25, 53, 57], distillation [64, 67] and inference [44, 68]. Commonly, different tasks demand different model precisions; for example, fine-tuning is often performed using higher precisions such as FP16 [42], then, the fine-tuned model would be quantized to a lower precision format such as INT8 [13, 39] for faster inference. Hence, many workflows require access to the same model under different precisions (e.g. FP16 and INT8): in addition to fine-tuningthen-inference, other tasks with this requirement include Model Cascade [26, 66] and Model Chaining [22, 58, 61]. Moreover, data scientists and researchers also iterate between different-precision models for testing, experimentation and benchmarking [10, 11].

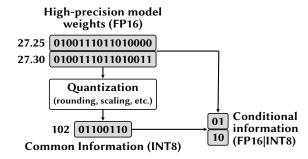


Figure 1: QStore stores the conditional bit representation of the high-precision model weights alongside the common low-precision, quantized weights to store a model in both high and low-precisions using fewer bits.

Storing Multiple Models is Costly. Currently, a common approach to maintaining multiple models of varying precisions while doing the aforementioned tasks is to store them as is (i.e., separately storing the multiple precision versions) [10, 11]. However, as newer, more complex tasks demand ever-increasing model sizes (e.g., Mistral-7B [38] being sufficient for simple math tasks, while more complex, multi-modal tasks [60] require larger models such as Qwen2.5-VL 32B [21]), the storage cost incurred by storing multiple versions of a model can quickly become prohibitive - for example, 91.8 GB of space is required to store just the BF16 [40] and INT8 (quantized) versions of the Deepseek-Coder [69] 33B parameter model. While this is a significant issue for developers using these models, it also increases the incurred cloud storage cost for model hubs like HuggingFace, Ollama, and ModelScope, since model providers and users end up storing multiple precisions of these models separately on these platforms to account for user accesses to models in different precisions.

One potential approach to reduce storage cost is to only store the highest-precision model (e.g., FP16 or BF16), then quantize inmemory if lower-precision versions (e.g., INT8) are needed [30]. 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) a computationally expensive quantization process (e.g., up to 21 GPU minutes for a 13B model [31]). Alternatively, stored models can be compressed with an algorithm such as LZ4 [3], ZSTD [4], or ZipNN [34]. However, these algorithms either utilize generic techniques that underperform on ML model weights (e.g., LZ4 and ZSTD), or are tailored to one specific precision (e.g., ZipNN for FP16/BF16 weights).

Our Intuition. We propose QStore, a data format for efficiently storing varying precision versions of a model. We observe that despite being quantized, a lower-precision (e.g., INT8) version of a model contains significant information that is also present in a higher-precision (e.g., FP16, BF16) version. Hence, compared to separately compressing and storing a pair of higher and lower-precision models, it is possible to use less space to *simultaneously* represent both models in a unified format. Fig 1 illustrates this idea: much of the information present in the weights of a high-precision FP16 model is already contained in the low-precision (i.e., quantized) INT8 version. Hence, given an already efficiently stored low-precision model, we can also store the high-precision model using only a few *additional* bits per weight representing 'extra information' not present in the low-precision model (i.e., the 'FP16 | INT8' *conditional* model). Such a unified data format would (1) save storage space versus storing both 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 pair of high and low-precision models is challenging. First, we need to carefully define the 'extra information' not present in the lower-precision model required to reconstruct the higher-precision model. Significant information is lost while quantizing a higher-precision model to a lower-precision one (e.g., from operations like rounding), hence, our definition should effectively encapsulate this information gap for lossless reconstruction. Identifying this information gap is nontrivial, as a quantized weight may be significantly different from the original weight in both bit representation and numerical magnitude (Fig 1). Second, our representations of the lower-precision model's information and 'extra information' should strike an acceptable storage/processing speed trade-off: for example, naïvely defining and storing information at a 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 [48]. Our decoding is notably lossless (e.g., versus dequantization [47]).

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 popular foundation models of varying sizes that QStore reduces storage costs of a pair of

high and low-precision models by up to 55% while enabling up to  $1.6\times$  and  $2.2\times$  faster loading and saving of the model pair, respectively, versus alternative approaches. (§6)

#### 2 BACKGROUND

Efficiently storing and deploying large foundation models is challenging. Our work addresses this challenge through proposing a compressed format capable of concurrently storing multiple model representations of different precisions. This section overviews 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 [46], 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 [35], BF16 [7, 40]) 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 [63]), further quantization is common—typically to 8-bit (INT8) [28, 36], but sometimes more aggressively to 4-bit (INT4, NF4) [29, 30, 43] or even lower [56]. Recently, FP8 quantization has also been used during inference [45].

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 representable value in low-precision format (e.g.,  $42.25 \rightarrow 42$ ), which is fast, but can significantly degrade model accuracy (e.g., with outlier weights). (2) Channel-wise quantization such as LLM.int8() [28] and SmoothQuant [62] apply per-channel scaling and quantization to model weights to better preserve outliers. (3) Reconstruction-based approaches such as AWQ [43] and GPTQ [30] are also applied on a per-channel or per-block level, but they aim to quantize in a fashion such that the original high-precision weights can be reconstructed with minimal error. While these methods are 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 this nature allows our approach to be generally extendable (§4.2).

QStore vs Lossy Quantization. Quantization is inherently a lossy transformation aimed at reducing model complexity. In comparison, our approach for model storage via QStore is orthogonal, since it takes 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 precision levels (16-bit FP16 and BF16, 8-bit INT8) in this paper, our approach does not assume any specific closed form for

the quantization method that is used; hence, our techniques can be generalized to other quantization levels (e.g., INT4, or other custom levels). We briefly describe how this can be done in §7.

### 2.2 Data compression

Model hosting platforms (e.g., HuggingFace [59]) store foundation models in wrapper formats such as Safetensors [9, 24], ONNX [6], TensorFlow, and SavedModel[8] that allow transparent storage of additional information such as tensor names and quantization information along with the model weights. However, these formats store weights in an uncompressed fashion. Another approach orthogonal to quantization that has been explored to reduce model sizes (for storage) is compression. We discuss the pros and cons of various compression techniques applicable to foundation models.

Generic Compression Algorithms. Standard compressors such as GZip [1], ZSTD [4], LZ4 [3] can be applied to model weights. These approaches treat (the sequence of) weights as a generic byte stream and are agnostic to specific structural and numerical properties of the model weights. ALP [20] targets general floating point numbers, but only supports 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 [34]) rendering common techniques such as dictionary coding [52] ineffective.

Compression for ML Models. Recently, some approaches have been proposed for specifically compressing ML models: ZipNN [34] compresses BF16 weights by reordering the 16-bit float into 2 byte streams, and compressing each stream separately with Huffman coding. Additionally, they propose numerical delta storage to store multiple perturbed versions (e.g., after fine-tuning) of the same base model at the same precision. NeuZip [33] uses lossy compression to speed up inference by quantizing mantissa bits, and applying loss-less compression to exponent bits with an entropy coder to speed up training. Huf-LLM [65] uses hardware-aware huffman compression, breaking the 16-bit value into non-standard bit-level patterns and compressing these streams separately for fast inference.

QStore (ours): Joint Compression. Unlike existing compression methods, QStore targets the joint compression of a quantized and unquantized pair of models, and achieves higher compression ratios versus compressing them separately (empirically verified in §6). Additionally, QStore runs purely on CPU, and does not depend on the availability of specific architectures (e.g., systolic arrays, TPUs/NPUs) 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

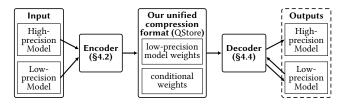


Figure 2: QStore pipeline. A high and low-precision model pair is encoded into a unified, storage-friendly format (QStore), from which both models can be efficiently retrieved.

as it stores tensors separately (e.g., safetensors [9], PyTorch pickle objects [2], TensorFlow SavedModel [8], etc. are acceptable).

Encoding. QStore's compression process utilizes an encoder to encode the weights of the models: the encoder first compresses the weights of the low-precision model, then compresses the conditional information present in the high-precision model but not in the low-precision model (i.e., 'extra information', §1). We describe QStore's encoding in detail in §4.2.

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 decompression process utilizes a decoder to act on the two files contained 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 to reconstruct the high-precision model. We describe QStore's decoding in §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:

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

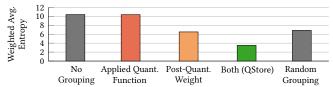


Figure 3: Weighted entropy of different grouping strategies on the Llama 3.1 8B Instruct model's 16-bit weights. QStore's combined grouping achieves high entropy reduction (hence compression ratio) versus alternative grouping strategies.

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 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 independent quantization functions to a single tensor and perform block-wise quantization (§2.1). For example, LLM.int8() [28] uses a different scaling factor to quantize each tensor row (e.g.,  $Q_{row=i}(w_i) = round(\frac{128w_i}{s_i})$ , where  $s_i$  is the scaling factor for row *i*). The quantization function is often chosen w.r.t. the 16-bit weights; a common choice is  $s_i = abs(max(w_i))$ , the magnitude of the largest/smallest weight in group i [28, 43]. Hence, the conditional information of a group of floating point weights w.r.t. their quantized integer weights H(W|Q(W)) will change as Q(W)changes. While grouping floats by the quantization function applied alone achieves negligible entropy reduction (due to the intra-group float distributions still being largely random), we observe that a combined grouping of the quantization function applied, and the quantized weight value achieves significant compression benefits (e.g., versus grouping only by one of the two criteria, or randomly grouping with the same number of groups, 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 the conditional information using weights of both the high and low-precision model (w and Q(w), respectively) as input. Following intuition described in §4.1, the weights of the high-precision model w are first grouped according to the applied quantization function (e.g., for LLM.int8() [28] each group will consist of all tensors with the same applied scale value). Then, weights in each group are further divided into subgroups of weights quantizing to the same value. Figure 4 depicts an example: rows  $w_1$ ,  $w_3$ , and  $w_2$  are quantized with distinct scale values (32 and 16, respectively), hence their weights are placed into group 1 ( $s_1 = s_3 = 32$ ) and group 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.

### 4.3 QStore Format

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

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

Compressed Conditional Information. QStore stores the conditional information following group (i.e., applied quantization function), then subgroup (i.e., post-quantization value) order. It maintains a header, which stores (1) the mapping from groups to their positions in the original model (e.g., row number), and within each group, (2) per-subgroup data (i.e., whether compression was applied, and chunk sizes, similar to the quantized weights). Notably, despite QStore also reordering the weights in each group based on subgroups, it does not store the mapping of weight positions within each sub-group (row): this is because the information is already present in the quantized weights, e.g.  $w_{13}$  assigned to group

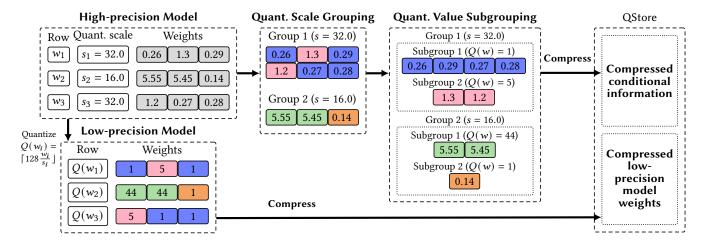


Figure 4: Compressing a tensor in the model pair with QStore. Weights in the high-precision model are grouped by the quantization function (scale) applied, then subgrouped by the post-quantization value from the low-precision model.

1, subgroup 1 in Fig 4 can be inferred to be the third element in row  $w_1$  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 following the procedure described above. Then, QStore's decoder first decompresses the conditional information, then applies the decompressed information 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.

Remark. QStore's decoding process for retrieving the high or low-precision model is faster than loading the respective model uncompressed, and comparable to loading the respective model (separately) compressed using an off-the-shelf algorithm (e.g. LZ4). However, as QStore jointly stores the model pair, QStore's approach achieves significant time savings for loading the low-precision model versus the common practice of loading the unquantized model, then quantizing it in memory (§6.4).

#### 5 IMPLEMENTATION

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. For efficiency, we implement QS-tore's per-tensor decoding for model loading (§4.4) in a pipelined manner, where one tensor's decompression overlaps with the next tensor's read. However, other parallelization strategies can be used in its place [49, 51], such as completely parallelizing both the reading and decompression of tensors, which may bring larger benefits on specific hardware (e.g., local SSD [23, 50]).

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]). In this situation we would not apply decode pipelining, and only read and decompress tensors when required; we defer detailed performance optimization and engineering to future work.

#### **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 (45% of the original size) for storing a high and low-precision model pair—up to 1.6× better than the next best method. (§6.2)
- (2) 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 and/or compression methods applied separately on the two models (§6.3).
- (3) Fast Retrieval: A model pair stored in the QStore format can be loaded up to 1.8× faster versus alternative formats. Specifically, the low-precision model can be loaded from QStore up to 2.5× faster versus loading and quantizing the high-precision model in-memory (§6.4).

#### Deeper Performance analysis of QStore (Ours)

(1) Effectiveness Under Constrained Bandwidth: QStore's effective model compression and storage enables up to 2.2×

Table 1: Summary of models used for evaluation.

Model	Params.	<b>Model Pair Size</b>	Modality
Qwen 2 Audio [27]	7B	19.9 GB	Audio-Text
Mistral v0.3 [37]	7B	19.5 GB	Text
Llama 3.1 [32]	8B	19.5 GB	Text
Gemma 3 [54]	27B	72.7 GB	Image-Text
Qwen 2.5 VL [21]	32B	87.7 GB	Video-Image-Text
Deepseek Coder [69]	33B	91.9 GB	Text (Coding)

faster model loading times versus loading uncompressed models under I/O-constrained scenarios (§6.5).

(2) Effective Encoding of Conditional Information: QS-tore efficiently compresses conditional information—despite being necessary for reconstructing the high-precision model from the low-precision model, its comprises only up to 36.2% of the total QStore file size (§6.6).

### 6.1 Experimental Setup

Dataset. We select 6 popular foundation models across various modalities, domains, and languages for comprehensive 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() [28]) weights. We summarize models and their characteristics in Table 1.

*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 of HuggingFace's transformers library [59]. We use its Python API [16, 18].
- Iz4 [3]: We use the default compression level of 1.
- Zstd [4]: We use a compression level of 2.
- ZipNN [34]: 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 the methods to sequentially process each tensor to and from a single file for both model saving and loading. Tensor read/write and decompression/compression is pipelined (where applicable) 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, with read latency of 7.49ms. <sup>2</sup>

Time Measurements. We measure (1) save time as the time taken to compress and store a model onto storage, and (2) load time as the time taken to read and decompress the selected model (high or low-precision) from storage 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

Table 2: Average bits per weight to store each model pair.

Model	Safetensors	Zstd	QStore (Ours)
Qwen 2 Audio [27]	24	19.564	11.434
Mistral v0.3 [37]	24	19.518	11.216
Llama 3.1 [32]	24	19.482	11.127
Gemma 3 [54]	24	19.379	10.925
Qwen 2.5 VL [21]	24	19.173	10.732
Deepseek Coder [69]	24	19.591	10.865

compression/decompression with 48 threads for all methods. The OS cache is cleared between consecutive experiment runs.

*Reproducibility.* Our implementation of QStore and experiment scripts can be found in our Github repository.<sup>3</sup>

### 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 Fig 5.

QStore's file size is consistently the smallest, and is up to  $2.2\times$  and  $1.6\times$  smaller versus Safetensors (uncompressed) and next best compression method, respectively. As hypothesized in §2.2, Zstd and lz4 achieve suboptimal compression ratios due to the traditional compression techniques they utilize being ineffective on model tensor data—When Zstd is used along with ZipNN (Fig 5, the size decreases slightly, but is still  $1.6\times$  bigger than our model pair. Iz4 achieves no benefits compared to the uncompressed storage. QStore's high compression ratio translates to significant (52%-55%) space savings across model sizes (Fig 5b): storing the Deepseek Coder's model pair with QStore takes only 42GB versus the 92GB of storing the models as is without compression.

Savings Versus Storing Only High-Precision Model. We additionally compare QStore's storage cost versus storing only the high precision model (BF16) with baselines in Fig 6. Notably, QStore's storage cost for the entire model pair is still up to 33% smaller than storing only the high precision model without compression, up to 13% smaller versus general compression algorithms (Zstd), and is comparable to (only up to 7% greater) the specialized ZipNN method designed for 16-bit models.

#### 6.3 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.

We report results in Fig 7. QStore's model pair storing time is up to  $1.7\times$  and  $2.8\times$  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 this number to 10.7-11.5 (Table 2), 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 (Fig 5).

 $<sup>^{1}</sup>$ Measured with dd with 1MB block size, reading 1024 blocks from a model file.

 $<sup>^2</sup>$ Measured with iostat -x.

 $<sup>^3</sup> https://github.com/illinoisdata/qstore\\$ 

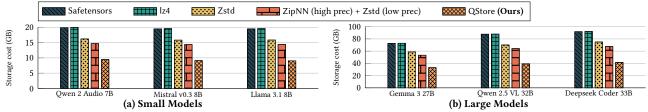


Figure 5: QStore's storage cost for storing a high and low precision model pair versus baselines. QStore achieves up to  $2.2\times$  space savings versus storing the models uncompressed with file sizes up to  $1.6\times$  smaller than the next best alternative.

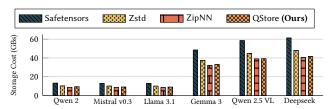


Figure 6: QStore's model pair storage cost versus only storing the high-precision model with baselines. QStore's size is up to 1.5× smaller versus no compression and is comparable to storing with ZipNN (only up to 5% larger).

#### 6.4 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 a high-precision model, a low-precision model, and both models in Fig 8, Fig 10, and Fig 9, respectively. QStore loads the high-precision model up to  $1.4\times$  faster versus loading it without compression (Safetensors), and exhibits comparable loading times versus loading it with a specialized compression algorithm ( $\pm 5\%$ , ZipNN). QStore loads the low-precision model with comparable time ( $\pm 5\%$ ) versus loading it with (Zstd) or without compression (Safetensors).

Time savings for Simulataneous Model Access. Notably, QStore saves significant time in cases where simultaneous access to both models (e.g., model cascade and chaining §1 or interactive computing [41]) is required; it loads the model pair up to 2.2× and 1.8× faster versus separately loading the two models stored without compression (Safetensors) or with an applicable compression algorithm (Zstd), respectively; this is because the size of QStore's model pair being significantly smaller than that incurred by separately storing the two models with alternative approaches (§6.2).

#### 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 11).

While QStore is faster than uncompressed loading at all bandwidths, the speedup increases from  $1.7\times$  (500MB/s) to  $2.1\times$  and  $2.2\times$  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 2.5 VL model at 20MB/s; this significantly

improves user experience with models in the common scenario where models are downloaded from cloud storage with limited network bandwidth (typical speeds of 30MB/s [34], grey vertical lines in Fig 11).

### 6.6 Effective Conditional Information Storage

This section studies the effectiveness of QStore's compression of conditional information. We store the model pair using QStore, and measure the space taken by the low-precision weights and conditional information, respectively (results in Fig 12). QStore's compressed conditional information only takes up to 39% of the total size, and accordingly contributes only up to 40% of the model pair loading time across all 6 models. This shows the effectiveness of QStore's conditional encoding 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).

#### 7 DISCUSSION

Compatibility with other Quantization Methods and Datatypes. While we present our entropy analysis (Fig 3) and experiments (§6) for one of the default quantization methods on HuggingFace, LLM.int8() [28], (i.e. a FP16/BF16-INT8 model pair), QStore is compatible with other quantization schemes and datatypes (e.g., integertyped low-precision models). This is because QStore does not use specific values of the high or low-precision models and directly applies byte-level entropy coding for storage (§4.2); only the ordering of weights in each group (present in the low-precision model), along with the stored conditional information are required to losslessly reconstruct the high-precision model (§4.4), and both are datatypeagnostic. Hence, QStore can be trivially extended to support other datatypes (e.g., FP16-FP8 or FP32-BF16 model pairs).

Data Compressibility. QStore's compression ratios may differ based on the datatype of the high-precision model. For example, given a low-precision INT8 model, and a choice of either BF16 or FP16 for the high-precision model, the conditional information of BF16|INT8 compresses slightly better ( $\sim 2\%$ ) compared to FP16|INT8. This is because two floats in the same group quantizing to the same value are likely to overlap in their significant (exponent) bits. The first byte of BF16 has 7 exponent bits, vs 5 exponent and 2 mantissa bits of FP16; hence, two BF16 floats quantizing to the same value enables more effective compression versus 2 FP16 values.

Storing more than Two Models. Fundamentally, QStore relies on using conditional information to simultaneously store model pairs (§4.1). Hence, QStore's approach can be extended to store more than two precisions, for instance, a three-level FP32-BF16-INT8

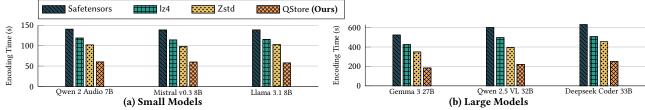


Figure 7: QStore's encoding time for saving a model pair versus baselines. QStore enables up to  $2.8 \times$  faster model saving versus storing the models uncompressed, and is up to  $1.7 \times$  faster than storing the models with an applicable compression algorithm.

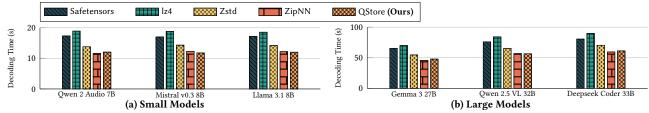


Figure 8: QStore's decoding time for loading the high precision model versus baselines. QStore allows up to  $1.4 \times$  faster loading compared to safetensors, and is comparable ( $\pm 5\%$ ) to loading the models with a specialized compression algorithm (ZipNN).

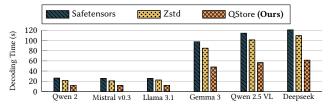


Figure 9: QStore's decoding time (secs) to load both the high and low precision model pair versus baselines. QStore is up to  $2.2\times$  faster compared to loading uncompressed models, and up to  $1.8\times$  faster than applicable compression baselines.

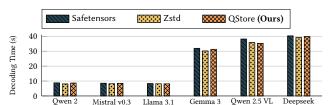


Figure 10: QStore's decoding time (secs) to load only the low-precision model is comparable ( $\pm 5\%$ ) to other baselines.

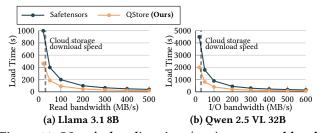


Figure 11: QStore's decoding time (secs) versus read bandwidth for two selected models. QStore's smaller incurred storage size saves loading time by  $2.2\times$  at lower bandwidths.

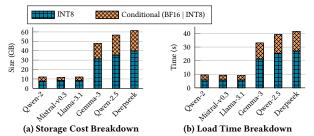


Figure 12: QStore's storage cost and loading time breakdown for the INT8 and conditional (BF16  $\mid$  INT8) encodings. Less than 40% of QStore's size is from the conditional encoding.

model chain: First, QStore would store the largest FP32 model as a BF16 model and a FP32 | BF16 conditional encoding  $E_1$ , then decompose the BF16 model into the INT8 model and BF16 | INT8 conditional encoding  $E_2$ . Hence, the final compressed QStore would be  $\{w_{INT8}, E_2(w_{BF16}|w_{INT8}), E_1(w_{FP32}|w_{BF16})\}$ . As mentioned in §1, this extension would especially benefit model storage hubs like HuggingFace [59] which can store multiple quantized representations of the same model for anticipated user access with significantly lower storage cost versus separately storing the precisions.

#### 8 CONCLUSION

In this paper, we introduced QStore, a unified file format for storing a pair of high and low-precision models. QStore defines a novel representation for storing the conditional information present in the high-precision model but not in the low-precision model. For model pair storage, QStore stores the low-precision model, then applies novel grouping techniques on the conditional information to achieve efficient storage via 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 saving and loading versus existing approaches, respectively.

#### REFERENCES

- $[1] \ \ 1992. \ \textit{GZIP}. \ \ \text{Retrieved Apr 18, 2025 from https://www.gnu.org/software/gzip/}$
- [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-diffusionnearly-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-nvidiah100-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/manpages/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] 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.
- [23] 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.
- [24] 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).
- [25] 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
- [26] 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)
- [27] 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).
- [28] 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.
- [29] 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.
- [30] 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).
- [31] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. Gptq: Accurate post-training quantization for generative pre-trained transformers. arXiv preprint arXiv:2210.17323 (2022).
- [32] 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).
- [33] 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
- [34] 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
- [35] 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
- [36] 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.
- [37] 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).
- [38] 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
- [39] 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.
- [40] 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
- [41] Zhaoheng Li, Supawit Chockchowwat, Ribhav Sahu, Areet Sheth, and Yongjoo Park. 2024. Kishu: Time-Traveling for Computational Notebooks. arXiv preprint arXiv:2406.13856 (2024).
- [42] 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.
- [43] 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.
- [44] 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.
- [45] 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)
- [46] 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
- [47] Nvidia. 2024. Working with Quantized Types. https://docs.nvidia.com/ deeplearning/tensorrt/latest/inference-library/work-quantized-types.html.
- [48] Stack Overflow. 2012. Byte vs Bit Access Speeds. https://stackoverflow.com/questions/7782110/is-it-fastest-to-access-a-byte-than-a-bit-why.
- [49] 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

- [50] 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.
- [51] 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.
- [52] 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.
- [53] Mohammadreza Tayaranian Hosseini, Alireza Ghaffari, Marzieh S. Tahaei, Mehdi Rezagholizadeh, Masoud Asgharian, and Vahid Partovi Nia. 2023. Towards Finetuning 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
- 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, CI 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 Oiu, Marcella Valentine, Marina Coelho, Marvin Ritter, Matt Hoffman, Matthew Watson, Mayank Chaturvedi, Michael Moynihan, Min Ma, Nabila Babar, Natasha Nov. Nathan Byrd. Nick Rov. Nikola Momchey, Nilay Chauhan, Noveen Sachdeya, 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
- [55] The ModelScope Team. 2023. ModelScope: bring the notion of Model-as-a-Service to life. https://github.com/modelscope/modelscope.
- [56] 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
- [57] 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
- [58] 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.
- [59] 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
- [60] Mingyuan Wu, Jize Jiang, Haozhen Zheng, Meitang Li, Zhaoheng Li, Beitong Tian, Bo Chen, Yongjoo Park, Minjia Zhang, Chengxiang Zhai, et al. 2025. Cacheof-Thought: Master-Apprentice Framework for Cost-Effective Vision Language Model Inference. arXiv preprint arXiv:2502.20587 (2025).
- [61] 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
- [62] 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—38090
- [63] 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).
- [64] 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).
- [65] 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. 00022
- [66] Xuechen Zhang, Zijian Huang, Ege Onur Taga, Carlee Joe-Wong, Samet Oy-mak, and Jiasi Chen. 2024. Efficient Contextual LLM Cascades through Budget-Constrained Policy Learning. arXiv preprint arXiv:2404.13082 (2024).
- [67] 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
- [68] 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).
- [69] 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).