MIXED-PRECISION QUANTIZATION FOR LANGUAGE MODELS: TECHNIQUES AND PROSPECTS

Mariam Rakka[1], **Marios Fournarakis**[2], **Olga Krestinskaya**[3], **Jinane Bazzi**[3],
Khaled N. Salama[3], **Fadi Kurdahi**[1], **Ahmed M. Eltawil**[3], **Mohammed E. Fouda**[4,*]

[1]University of California Irvine, Irvine, CA, USA,
[2]Wayve AI, London, UK,
[3]King Abdullah University of Science and Technology, Thuwal, Saudi Arabia,
[4]RAIN AI, San Francisco, USA.
[*] Email: `foudam@uci.edu`

ABSTRACT

The rapid scaling of language models (LMs) has resulted in unprecedented computational, memory, and energy requirements, making their training and deployment increasingly unsustainable. Quantization has emerged as an essential compression technique to reduce model size, alleviate memory bottlenecks, and accelerate inference. However, while uniform low-bit quantization (e.g., INT8, INT4) provides significant efficiency gains, it can degrade accuracy in sensitive components of transformer-based LMs. Mixed-precision quantization offers a promising alternative by selectively allocating precision across layers or within tensors to balance efficiency and accuracy. This survey provides a comprehensive overview of Mixed-Precision quantization frameworks for LMs (MX-PLMs). We first review quantization fundamentals, including uniform and non-uniform quantizers, quantization granularity, and methods widely used in post-training quantization. We then categorize and compare recent MXPLM frameworks according to their bit allocation strategies and precision configurations across weights, activations, and key-value caches. A comparative analysis highlights differences in perplexity, zero-shot task performance, and deployment trade-offs. Furthermore, we contrast MXPLMs with earlier mixed-precision quantization methods for deep neural networks, identifying strategies that transfer and those that face challenges in the LM setting. Finally, we summarize open issues and future directions, including hardware-aware design, activation quantization, and scalable optimization methods for billion-parameter models. By consolidating recent advances, this work serves as a reference for understanding the current landscape and research prospects of mixed-precision quantization for large-scale language models.

Keywords Language Models · Quantization · Mixed Precision Quantization · Post-Training Quantization (PTQ) · Quantization-Aware Training (QAT) · Model Compression · Bit Allocation.

1 Introduction

Over the past three years, the size of language models (LMs) has grown exponentially, placing increasing demands on modern computing infrastructure and driving the need for more advanced hardware (Fig. 1). Model sizes have increased, outpacing the growth of hardware and memory capabilities. For example, OpenAI's GPT-3, released in 2020 with 175 billion (175B) parameters, requires an estimated 3.14×10^{23} floating-point operations for training, equivalent to 355 GPU-years on an NVIDIA V100 cluster and roughly \$4.6 million in compute cost for a single run [1]. Deploying such models requires extensive cloud infrastructure and GPU resources due to their high compute and memory requirements. In 16-bit precision, the 175B-parameter GPT-3 model requires roughly 350 GB just to store its weights, necessitating model-parallel inference across multiple GPUs [1]. Further scaling of these models will amplify both compute and memory demands. Therefore, model compression techniques and efficient methods for quantizing and processing such models with lower bit precision are essential to sustainably support the growth of LMs.

Since the Transformer architecture and multi-head self-attention were introduced in 2017 [2], language models have rapidly scaled in size and capability. This trend started with early models such as BERT [3], GPT [4], and T5 [5], released between 2018 and 2019. By 2020, models such as GPT-2 (1.5B parameters) [6] and GPT-3 (175B parameters) [7] demonstrated few-shot prompting and strong transfer learning capabilities. GPT-3 marked the rise of foundation models, massive pretrained models that serve as a general platform for a wide range of downstream tasks. Early examples included Google's PaLM (540B parameters) [8], DeepMind's Chinchilla [9], and Microsoft and NVIDIA's 530B-parameter Megatron-Turing NLG [10], released in 2022. By 2023, models like GPT-4 [11] introduced multimodality, enabling input across text and images within a unified framework. In general, multi-modal foundation models are AI systems designed to process and reason over multiple types of data (modalities), including text, images, audio, video, tables, and graphs, within a single unified architecture [12]. These include OpenAI's GPT-4 [13], Anthropic's Claude 3 [14], Google's Gemini [15], among others. Further, the most recent models, like GPT-5 [16], are more complex and designed to be more agentic. For example, GPT-5 has been explicitly optimized for agentic tasks, autonomously chaining multiple tool calls to address complex multi-step problems [16]. Similarly, Google's Gemini 2 emphasizes the development of AI assistants that can interpret their environment, reason several steps ahead, and autonomously take actions on behalf of the user under supervision [17]. Building on this trajectory, Anthropic's Claude 4 offers advanced tool-use and extended reasoning capacities, supporting multi-step workflows across thousands of sequential steps and substantially expanding the capabilities of AI-driven agents [18]. The advanced reasoning abilities and problem-solving skills of such models represent an early step toward Artificial General Intelligence (AGI) [12, 19].

As illustrated in Fig. 1, the size of Language Models (LMs) and the number of parameters have been increasing each year to achieve improvements in model capability. Fig. 1 shows the state-of-the-art LMs that have appeared in the last few years and their ELO scores, a ranking metric that quantifies how likely one model is to outperform another in a head-to-head evaluation [20]. While scaling up has brought significant gains in reasoning, language understanding, and generation, this growth trajectory raises concerns about sustainability due to increasing computational, memory, and energy demands. To address these challenges, model compression techniques, such as quantization, are essential for reducing model size, decreasing memory requirements, and improving inference efficiency, all while maintaining competitive performance.

Quantization is a key model compression technique that benefits both training and inference of LMs. Many state-of-the-art LMs are memory-bound during inference [21], making model size reduction crucial for alleviating memory bottlenecks and enabling deployment on smaller or more cost-effective hardware. For instance, deploying the 27B-parameter Gemma 3 model in BF16 (16-bit brain floating-point) format requires 54 GB of GPU memory, whereas INT4 (4-bit integer) quantization reduces this requirement to only 14.1 GB, which is nearly a 4x reduction [22, 23]. This reduction significantly decreases the number of GPUs needed for distributed inference. Beyond memory savings, quantization also improves system throughput and latency. For example, the FP16 version of the LLaMA 3–8B model achieves 135.79 tokens/sec on the NVIDIA H100, while the INT8 and INT4 versions achieve 158.90 and 211.50 tokens/sec, respectively [24].

While low-bit quantization formats, such as INT8 and INT4, offer substantial memory and speed benefits, they can compromise accuracy, particularly when quantizing sensitive layers, such as attention projections or embedding matrices, in transformer architectures. To balance efficiency and performance, mixed-precision quantization is a promising approach for model compression, enabling the selective allocation of precision to preserve accuracy while reducing resource usage.

A growing body of related works addresses efficiency for LMs, with different emphases on quantization breadth versus mixed-precision depth. [25] presents a dedicated survey of *low-bit* Large LMs (LLMs) spanning fundamentals, numeric formats, system/toolchain aspects, and algorithmic strategies for training and inference; mixed precision is covered as one instrument within the broader low-bit toolbox rather than the organizing principle. Complementary surveys organize *efficient LLM inference* end-to-end, covering data-, model-, and system-level optimizations (e.g., KV cache, attention scaling, batching, scheduling), where quantization appears as one category among many [26, 27]. Broader compression overviews position quantization alongside pruning and distillation to map the overall landscape rather than concentrating on mixed-precision design [28].

Several widely cited works are evaluation-centric rather than surveys; for instance, [29] benchmark post-training quantization (PTQ) across weights, activations, and KV cache on multiple model families and task types, providing practical guidance but not a taxonomy of mixed-precision frameworks. There are also survey-adjacent syntheses focused specifically on quantization for LLMs, offering broad methodological coverage without centering mixed-precision allocation [30]. Beyond LLM-specific surveys, [31] provides a cross-domain survey of *mixed-precision neural networks* that categorizes frameworks by optimization strategies (e.g., reinforcement learning-based search) and quantization/rounding choices; we reference this work for general mixed-precision taxonomies and contrasts with earlier DNN literature, while our focus remains on LMs. Related community overviews for Transformer compression [32]

and quantization in adjacent domains (e.g., ViTs and general DNNs) offer transferable taxonomies and hardware perspectives [33, 34].

Our article is intentionally *narrower and deeper* on **mixed precision** for LMs. First, it *formalizes* mixed precision at both inter- and intra-layer levels and clarifies scope by excluding the common misnomer in which a-) only weights are quantized while activations remain floating point (e.g., W-int4/A-FP16) and b-) weights being uniformly quantized to one precision and activations being uniformly quantized to a different precision (e.g., W-int4/A-int8) from being labeled as "mixed precision". Second, it surveys *MXPLM* frameworks with an explicit emphasis on **bit-allocation strategies** across weights, activations, and KV cache, and it provides a structured taxonomy (MPW; MPW+UPA; MPW+MPA) that distinguishes inter- and intra-layer allocation. Third, it offers a comparative discussion across frameworks (perplexity and zero-shot performance, where available) together with deployment considerations and hardware-awareness notes germane to billion-parameter models. These contributions complement prior low-bit and efficiency surveys by *centering* mixed precision as the organizing axis and by analyzing how allocation policies (by layer, tensor, group) interact with LM-specific sensitivities and real-world deployment constraints.

To aid the reader, we first consolidate *quantization fundamentals*: uniform vs. nonuniform, symmetric vs. asymmetric, and common granularities (per tensor, per channel, per group, per token) because these choices directly shape feasible mixed precision allocations and their accuracy/efficiency trade-offs in LMs. Within this background, we review families of methods that inform or enable mixed precision in practice, including second-order (OBS/GPTQ-style) approaches, equivalent transformations (e.g., AWQ/SmoothQuant), and rotation-based techniques (e.g., QuaRot), focusing on how these choices affect allocation granularity, sensitive layers, and activation/KV handling.

Compared to low-bit overviews [25] and general efficiency surveys [26, 27], our contribution is to **center** mixed precision and to articulate the interactions between allocation policies and LM-specific sensitivities. Where broader compression surveys necessarily distribute attention across pruning and distillation [28], we provide a *focused synthesis* of MXPLM frameworks and their evaluation patterns. And whereas evaluation papers emphasize empirical outcomes under uniform low-bit settings [29], our discussion integrates those insights into a *framework-level* perspective on when and why mixed precision preserves accuracy while reducing memory and latency.

The rest of the paper is organized as follows: Section 2 introduces quantization methods and background. Section 3 presents recent MXPLM frameworks. Section 4 provides insights and discussions, including comparative evaluation across frameworks. Section 5 discusses quantization-compatible hardware, and Section 6 discusses prospects and future directions. Finally, Section 7 concludes the work.

2 Quantization methods

This section provides an overview of quantization fundamentals and discusses various types of quantizers. We categorize quantizers based on their properties: *uniform* vs. *non-uniform* and *symmetric* vs. *asymmetric*. We also explore quantization granularity (e.g., per-tensor, per-channel, per-group). Finally, we summarize and compare popular quantization techniques designed to improve the efficiency of LM deployment.

In practice, these quantization schemes are most commonly applied using the post-training quantization (PTQ) strategy [35–40], where quantization is applied to a pre-trained model without requiring retraining. Its key advantage is the ability to scale efficiently to models with billions of parameters, where retraining is often impractical. Although alternative approaches, such as quantization-aware training (QAT) [41–43] and fully quantized training [44], incorporate quantization during training to improve robustness under low-precision constraints, their significant computational and memory overheads limit their use in large language models. As a result, PTQ remains the most widely used method for compressing LMs, enabling efficient deployment across various platforms.

2.1 Quantizers

Quantizers map high-precision numerical values to lower-precision formats. In the context of LMs, integer quantization is one of the most widely adopted techniques, where high-precision floating-point values (e.g., FP32 or FP16) are converted into lower-precision integer formats such as INT8 or INT4. This transformation significantly reduces memory footprint and computational demands while preserving model accuracy. Recent studies have shown that INT4 and INT8 quantization can preserve over 99% of full-precision accuracy on benchmarks, such as the Massive Multitask Language Understanding (MMLU) and the AI2 Reasoning Challenge (ARC) benchmarks, when combined with proper calibration and quantization-aware techniques [25, 30]. For instance, methods, such as GPTQ [35] and QLoRA [45], report less than 1% degradation in average accuracy when compressing LLaMA and GPT-style models to INT4.
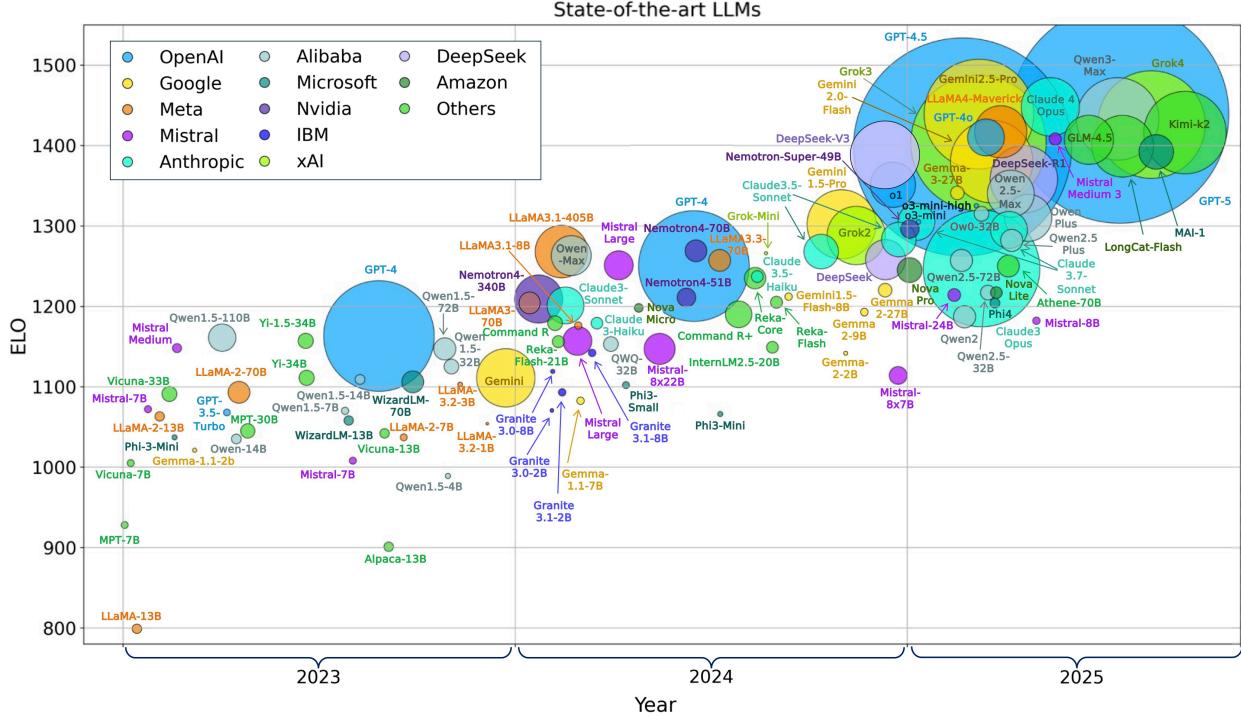
Figure 1: ELO scores of state-of-the-art LMs released over the last three years. Scores are obtained from [20], where the circle size represents the total number of parameters in a model. For models whose sizes were not officially disclosed, the values shown reflect the average of multiple open-source estimates.

Furthermore, the storage requirement for models, such as GPT-3, decreases from 350 GB in FP16 to approximately 90 GB in INT4, enabling deployment on resource-constrained systems, including single GPUs or edge devices [46]. Besides memory savings, integer operations are simpler and more hardware-efficient, enabling faster inference and lower power consumption on accelerators that natively support integer arithmetic. As such, integer quantization is a key enabler for scalable, efficient, and cost-effective deployment of large-scale models without compromising performance.

In the following sections, we describe the most common quantization schemes used in LMs, starting with uniform quantization, which includes both symmetric and asymmetric variants, followed by non-uniform quantization. These schemes are illustrated in Fig. 2.

2.2 Uniform quantization

Uniform quantization is the simplest and most widely adopted form of quantization in LMs due to its straightforward implementation and compatibility with hardware accelerators. It divides the numerical range into equal-sized intervals, resulting in uniformly spaced quantization levels. A key parameter in uniform quantization is the scale factor S, which defines the fixed step size between consecutive quantization levels. This scale enables the mapping of real-valued inputs to discrete integer levels and vice versa.

2.2.1 Symmetric uniform quantization

The simplest and most commonly used uniform quantization is symmetric quantization, where the quantized range is (roughly) centered around zero[1]. In this scheme, the original floating-point values are mapped symmetrically around zero in the integer domain. In this case, the integer value corresponding to the real-valued zero, referred to as the zero point, is fixed at 0. This eliminates the need for an offset, simplifying both the quantization and dequantization computations. Symmetric uniform quantization is characterized by two parameters: (1) a real-valued scaling factor S that defines the step size of each quantization interval and (2) a bitwidth b, which determines the number of representable

---

[1]Integer formats have an odd number of bin; therefore, it is common to assign an additional bin to the positive or negative range or drop one bin to make the grid fully symmetric around zero.

quantization levels. Given a real-valued input $x \in [\beta, \alpha]$, the quantization to a signed integer with b-bit precision is defined as:

$$x_q = \text{clamp}\left(\left\lfloor \frac{x}{S} \right\rceil, \; -(2^{b-1} - 1), \; 2^{b-1} - 1\right), \tag{1}$$

$$S = \frac{\max\left(|\alpha|, |\beta|\right)}{2^{b-1} - 1}, \tag{2}$$

where $\lfloor \cdot \rceil$ denotes the round-to-nearest operator. The above choice for the scaling factor is based on a *min-max* range [47] and ensures that the maximum absolute value is represented in the new grid. The clamping function ensures that the result remains within the representable range. It is defined as:

$$\text{clamp}(x; q_{\min}, q_{\max}) = \begin{cases} q_{\min}, & \text{if } x < q_{\min}, \\ x, & \text{if } q_{\min} \leq x \leq q_{\max}, \\ q_{\max}, & \text{if } x > q_{\max}, \end{cases} \tag{3}$$

where $q_{\min}$ and $q_{\max}$ correspond to the minimum and maximum integers representable by the chosen bitwidth b. For instance, in 8-bit signed symmetric quantization, $q_{\min}$ = -127 and $q_{\max}$ = 127. Fig. 2(a) illustrates an example of 8-bit (INT8) symmetric uniform quantization, where the quantization grid is centered around zero, and zero in the FP domain aligns exactly with zero in the quantized INT domain. To approximate the original real-valued input x from its quantized form x_q, a dequantization step is applied:

$$x \approx S \cdot x_q \tag{4}$$

2.2.2 Asymmetric uniform quantization

While symmetric quantization is simple and effective for zero-centered data distributions, it becomes suboptimal when the data range is significantly skewed or not centered around zero. In such cases, many quantization levels may be wasted on underutilized regions, leading to reduced precision for the actual data. To address this limitation, asymmetric uniform quantization introduces an integer zero-point offset Z that shifts the quantization grid, allowing it to better align with the input distribution. The asymmetric uniform quantization of a real-valued input $x \in [\beta, \alpha]$ using a *min-max* range is defined in the following equations and illustrated in Fig.2 for an INT8 example:

$$x_q = \text{clamp}\left(\left\lfloor \frac{x}{S} \right\rceil + Z, \; 0, \; 2^b - 1\right), \tag{5}$$

$$S = \frac{\alpha - \beta}{2^b - 1}; \quad Z = \left\lfloor \frac{-\beta}{S} \right\rceil \tag{6}$$

2.3 Non-uniform quantization

Unlike uniform quantization, non-uniform quantization uses variable step sizes, allocating different levels of precision across the numerical range. It assigns finer precision (smaller intervals) to regions where the data is denser and coarser precision (larger intervals) to sparser regions. This flexibility can improve representation efficiency and quantization accuracy, especially for skewed data distributions.

The non-uniform quantization operation is formally defined as:

$$x_q = D_i, \quad \text{if} \quad x \in [\Delta_i, \; \Delta_{i+1}) \tag{7}$$

where D_i denotes the discrete quantization levels and Δ_i represents the corresponding quantization intervals. When the input value x falls within the interval $[\Delta_i, \; \Delta_{i+1})$, it is mapped to the quantization level D_i. Unlike uniform quantization, neither the levels D_i nor the intervals Δ_i are evenly spaced.

Non-uniform quantization has seen practical success in recent LM compression efforts. For instance, QLoRA [45] introduces the NF4 (NormalFloat-4) quantization format; a data-aware, non-uniform quantizer designed to better capture the distribution of pre-trained model weights by allocating higher resolution near zero. This approach has been demonstrated to maintain downstream task performance while significantly reducing memory and computational requirements during fine-tuning.

As with uniform quantization, non-uniform schemes can be designed in symmetric or asymmetric forms, depending on whether the quantization grid is centered at zero or shifted to better match the data distribution, as illustrated in Fig. 2.

(a) Signed symmetric uniform quantization

(b) Signed symmetric non-uniform quantization

(c) Asymmetric uniform quantization
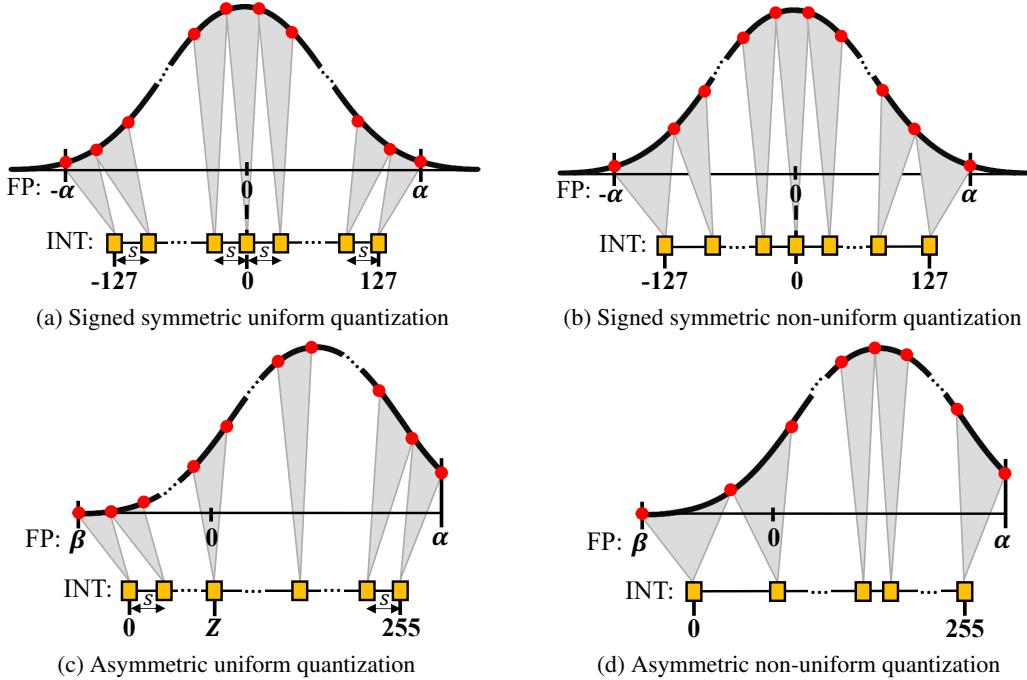
(d) Asymmetric non-uniform quantization

Figure 2: Illustration of different quantization schemes using 8-bit representation.

2.3.1 Quantization granularity

Quantization in LMs can be applied at varying levels of granularity, each offering different trade-offs between computational efficiency and model accuracy. The granularity level determines how many elements share the same quantization parameters, which impacts both the precision of the quantized model and its hardware efficiency.

At the coarsest level, **per-tensor quantization** assigns a single scaling factor and zero-point to an entire tensor [48]. This method is computationally efficient and memory-friendly, making it promising for deployment on resource-constrained devices. Per-tensor quantization is more commonly used for weight tensors, which can be pre-processed offline to become quantizable. In contrast, it is very rarely used for activations, whose dynamic nature and higher variation across hidden dimensions and tokens can lead to significant accuracy degradation. An example of successful per-tensor *static* quantization can be found in SmoothQuant [39], where weights and activation of the OPT-175B and BLOOM-176B models are quantized to 8 bits with less than 1% drop in average accuracy on zero-shot tasks.

A finer level of granularity is **per-channel quantization**, where each channel in a tensor, typically corresponding to rows or columns of a weight matrix, is assigned its own quantization parameters [49]. This approach strikes a balance between efficiency and accuracy by reducing quantization error compared to per-tensor quantization, while maintaining a moderate level of computational complexity. For these reasons, per-channel quantization is very commonly used for weight quantization with great success [35, 36, 50, 51].

However, applying per-channel quantization to activations in LMs introduces practical challenges. Specifically, assigning different scaling factors to each channel complicates matrix multiplications with weight matrices, which typically assume uniform scaling across the input. To avoid this complexity and preserve the structure of matrix operations, a widely used alternative is **per-token quantization**, where each token in the input sequence is assigned a single scaling factor [52]. While this approach is computationally convenient, it is suboptimal in terms of accuracy, as activation values in LMs vary more across channels than across tokens. To address this, recent works such as RPTQ [53] and MergeQuant [54] propose solutions like channel clustering and static per-channel calibration. Nonetheless, per-token quantization remains widely used due to its simplicity and compatibility with efficient hardware execution.

Further increasing granularity, **per-group quantization** divides each channel into smaller sub-groups, typically of fixed size (e.g., 128 elements), with each group assigned its own quantization parameters. Group quantization is particularly effective for low-bit quantization (e.g., 4-bit), offering improved accuracy by reducing intra-group variation without significantly increasing computational cost. Per-group quantization has become a standard technique in many recent LM quantization approaches [55–58].
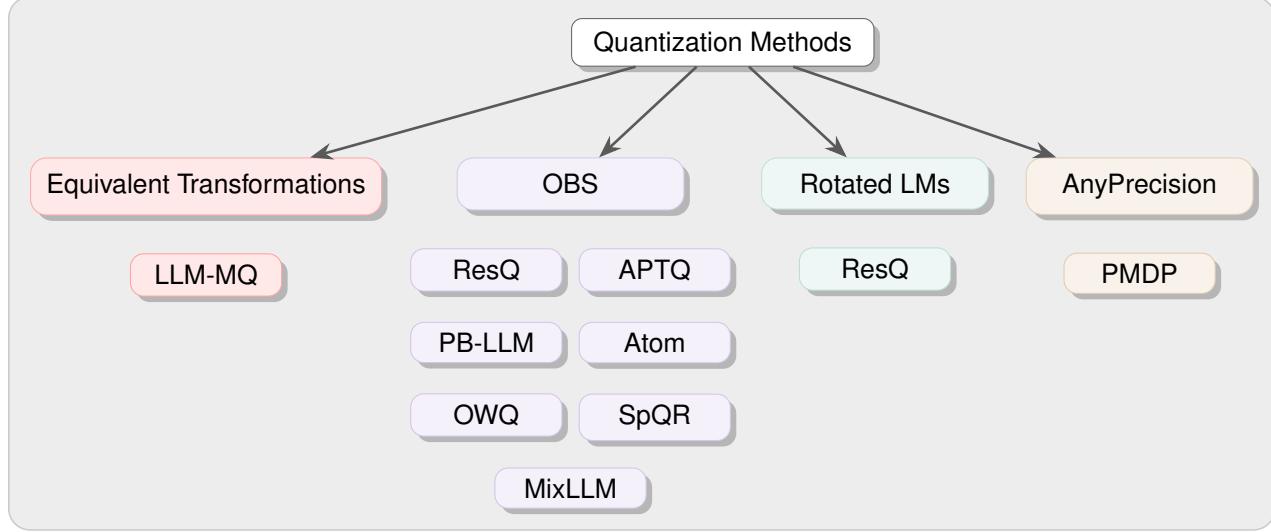
Figure 3: Taxonomy of mixed-precision methods based on the quantization methods they employ.

In practice, LMs often employ hybrid quantization strategies, where different model components use varying levels of granularity [59–61]. For instance, weights are often quantized using per-channel quantization to maintain accuracy with minimal overhead, while activations may use per-token or per-group quantization to capture dynamic input variations better. This adaptive approach helps optimize both memory efficiency and model performance.

2.4 Quantization methods for LMs

The substantial body of literature and methods that cover LM quantization confirms the effectiveness of quantization in accelerating LM inference. Despite the seemingly vast collection of methods and papers, most of them can be classified under a handful of families that share a common underlying principle. In this section, we highlight three primary quantization method families and discuss the most relevant and impactful papers related to each family.

2.4.1 OBS-inspired methods

These methods trace their origin back to 1993 with the *optimal brain surgeon* (OBS) [62], which used second-order derivatives to prune a neural network. To determine which weights to prune, the authors calculate the resulting change in error induced by the removal of each weight (also known as *saliency*). They then iteratively remove the weights with the smallest saliency and update the remaining weights to compensate for the increase in error that results from eliminating the weight.

Later in 2022, *optimal brain compression* (OBC) [63] extended the idea of OBS to quantization by jointly optimizing weight sparsity and low-bit precision using blockwise second-order approximations that scale to large models. Their formulation enables practical quantization and pruning of modern larger neural networks. However, it is the influential work of GPTQ [35] that solidified the OBS idea for LM quantization by tailoring the second-order approximation to the unique structure and scale of transformer models: (1) by recognizing that the exact order of quantization in large transformers did *not* matter, they eliminate the search for least salient weights, and (2) the Cholesky decomposition of the Hessian allowed for a parallelized and iterative computation. We later provide a detailed description of the GPTQ algorithm.

Recent advancements have extended GPTQ's efficacy, particularly in achieving extremely low-bit quantization. Output-adaptive Calibration (AOC) [64] directly addresses a limitation of GPTQ and similar methods by minimizing the distortion of the model's final output loss (e.g., cross-entropy) rather than just the layer-wise output reconstruction error. This "output-adaptive" calibration explicitly considers the global impact of quantization on the model's performance, leading to superior accuracy, especially at very low bit widths, such as 2-bit or binary, where traditional layer-wise methods often suffer significant degradation.

Generative pre-trained transformer quantization (GPTQ) [35] is a highly effective and widely adopted post-training quantization (PTQ) technique designed to compress large-scale transformer models to 3–4 bits per weight with minimal

accuracy degradation. It builds upon the Optimal Brain Compression (OBC) [63] framework by applying second-order approximations to minimize the output error induced by quantization, but introduces two key innovations that make it practical for LM-scale models: (1) a relaxation of the saliency-based quantization order, and (2) an efficient Cholesky-based approximation of the Hessian inverse.

GPTQ treats quantization as a reconstruction problem, where each weight matrix $\mathbf{W}$ is quantized to $\widehat{\mathbf{W}}$ so as to minimize the squared output error with respect to a calibration set:

$$\arg\min_{\widehat{\mathbf{W}}} \|\mathbf{W}\mathbf{X} - \widehat{\mathbf{W}}\mathbf{X}\|_2^2 \tag{8}$$

where $\mathbf{X}$ denotes the input activations. GPTQ processes each linear layer *independently* and partitions the weight matrix into blocks of B consecutive columns. Within each block, weights are quantized iteratively in a fixed left-to-right order, an important simplification over previous OBS-inspired methods, which required searching for the least salient weight at each step. GPTQ empirically demonstrates that this simplification has a negligible impact on accuracy while drastically reducing computational cost.

To correct for the error introduced by quantizing each column, GPTQ uses second-order information based on a Hessian matrix $\mathbf{H} = \mathbf{X}\mathbf{X}^\top$ that captures local curvature. Instead of explicitly computing the inverse Hessian, which is computationally expensive, GPTQ approximates $\mathbf{H}^{-1}$ using its Cholesky decomposition:

$$\mathbf{H}^{-1} \approx (\mathbf{L}\mathbf{L}^\top)^{-1} = \mathbf{L}^{-\top}\mathbf{L}^{-1} \tag{9}$$

where $\mathbf{L}$ is a lower-triangular matrix. This enables efficient updates and stable numerical inversion during calibration.

For each column j in a block, the quantized weights and quantization error are computed as:

$$\mathbf{Q}_{:,j} \leftarrow \text{quant}(\mathbf{W}_{:,j}) \tag{10a}$$

$$\mathbf{E}_{:,j-i} \leftarrow \frac{\mathbf{W}_{:,j} - \mathbf{Q}_{:,j}}{[\mathbf{H}^{-1}]_{jj}} \tag{10b}$$

The error is propagated to the remaining unquantized columns using second-order corrections:

$$\mathbf{W}_{:,j:(i+B)} \leftarrow \mathbf{W}_{:,j:(i+B)} - \mathbf{E}_{:,j} \cdot \mathbf{H}^{-1}_{j,j:(i+B)} \tag{11}$$

After all columns in a block are quantized, a final refinement is applied to subsequent blocks:

$$\mathbf{W}_{:,(i+B):} \leftarrow \mathbf{W}_{:,(i+B):} - \mathbf{E} \cdot \mathbf{H}^{-1}_{i:(i+B),(i+B):} \tag{12}$$

By combining these efficient updates with a fixed ordering and Cholesky-based Hessian inversion, GPTQ provides a practical and scalable solution for second-order PTQ of LMs.

2.4.2 Equivalent transformations

Another category of techniques improves quantization by applying mathematically *equivalent transformations* (ET) to the model weights or activations before quantization, making their distributions more amenable to low-bit representation while preserving the original network outputs. These methods employ simple reparameterizations, such as per-channel scaling or shifting, that do not change any layer's output in higher precision but have the effect of spreading out or reducing troublesome outlier values. Activation-aware weight quantization (AWQ) [36] is a popular method in this category that applies per-channel weight scaling guided by activation statistics. We describe the technique in more detail below.

Several methods have extended the equivalent transformation (ET) paradigm beyond AWQ by varying both the type of transformation and the way parameters are selected. SmoothQuant [39] introduced the idea of balancing quantization difficulty between weights and activations using per-channel scaling factors that are predefined based on layer norms. Outlier Suppression+ [65] builds on this by adding shifting operations alongside scaling, allowing it to align activation channels and suppress consistently large outliers. It applies grid search for scaling and uses predefined shift values. Finally, OmniQuant [66] takes this further by treating ET parameters as learnable, optimizing them via gradient descent to directly minimize quantization loss. While more computationally expensive, OmniQuant enables fine-grained adaptation of both scaling and shifting operations, supporting both weight and activation quantization.

Activation-aware weight quantization (AWQ) [36] is a weight-only post-training quantization method designed for LMs. The central observation is that a small number of weight channels (0.1%–1%) contribute disproportionately to large-magnitude activations. Quantizing these *salient* weights naively leads to large output errors. AWQ addresses this by rescaling the weight and activation channels such that the overall product remains unchanged, while salient weights

are protected via reduced dynamic range during quantization. Similar methods have been adopted in earlier literature to alleviate issues with per-tensor quantization of MobileNets [67].

Concretely, for a given weight matrix $\mathbf{W}$, and input activation $\mathbf{X}$, AWQ introduces a set of per-channel scales $\mathbf{s} = (s_1, \ldots, s_n)$ expressed a diagonal matrix $\mathbf{S} = \text{diag}(\mathbf{s})$ to rewrite the layers output:

$$\mathbf{W}\mathbf{X} = (\mathbf{W}\mathbf{S})\left(\mathbf{S}^{-1}\mathbf{X}\right), \tag{13}$$

ensuring the layer output remains unchanged under full-precision computation. AWQ then applies quantization to the rescaled weight matrix:

$$\widehat{\mathbf{W}} = Q\left(\mathbf{W}\mathbf{S}\right), \tag{14}$$

The resulting quantized weights can be folded back into the model, and the inverse scaling into preceding normalization layers, allowing AWQ to introduce no additional runtime cost.

The optimal scaling factors $\mathbf{s}^*$ are determined via grid search on a small calibration dataset, solving the following optimization problem:

$$\mathbf{s}^* = \arg\min_{\mathbf{s}} \|Q\left(\mathbf{W} \cdot \text{diag}\left(\mathbf{s}\right)\right)\left(\text{diag}\left(\mathbf{s}\right)^{-1} \cdot \mathbf{X}\right) - \mathbf{W}\mathbf{X}\| \tag{15}$$

After optimal scaling is applied, the quantization error is significantly reduced for important weight channels, particularly in the presence of high-variance activations. In practice, AWQ achieves competitive 4-bit weight-only quantization accuracy with full-precision activations and has been shown to outperform or match methods like GPTQ in some settings, particularly where speed and implementation simplicity are prioritized.

2.4.3 Rotated LMs

A third line of attack for quantizing LMs involves removing heavy-tailed outliers by making the weight and activation distributions more *coherent* [68] or uniform through orthogonal transformations. These methods apply random or learned orthonormal *rotations* to either the weight matrices or the activations, so that troublesome outliers are dispersed across dimensions. Intuitively, a rotation changes the basis of the vectors. For example, rotating a 2D vector $(1, 10)$ by $45°$ yields approximately $(7.8, 6.4)$, thereby eliminating the extreme value.

The core principle enabling rotation-based quantization is computational invariance. For a linear operation $\mathbf{Y} = \mathbf{X}\mathbf{W}$, where $\mathbf{X}$ is the input activation and $\mathbf{W}$ is the weight matrix, an orthogonal matrix $\mathbf{R}$ (satisfying $\mathbf{R}^\top\mathbf{R} = \mathbf{I}$) can be introduced without changing the output:

$$\mathbf{Y} = \mathbf{X}\mathbf{W} = \mathbf{X}\left(\mathbf{R}^\top\mathbf{R}\right)\mathbf{W} = \left(\mathbf{X}\mathbf{R}^\top\right)\left(\mathbf{R}\mathbf{W}\right), \tag{16}$$

where $\mathbf{X}' = \mathbf{X}\mathbf{R}^\top$ and $\mathbf{W}' = \mathbf{R}\mathbf{W}$ represent the rotated input weight matrix respectively. The weights can be readily rotated and then quantized *offline*, but for invariance to hold, the inputs must also be transformed accordingly during inference in an *online* manner. Randomized Hadamard transformations have been employed recently due to their efficient hardware implementation, leading to impressive low-bit quantized performance. QuIP# [69] first applied randomized Hadamard transformation along with vector quantization to achieve sub-4-bit weight-only quantization of Llama2 models. QuaRot [70] extends by applying *online* Hadamard transformations to the attention module to remove outlier features in keys and values, enabling low-bit quantization of activations & KV cache (see section below for more details).

QuaRot and QuIP# are only part of a growing family of rotation-based quantization methods that aim to remove statistical outliers and improve numerical conditioning before quantization. SpinQuant [71] goes a step further using backpropagation to learn the optimal rotation matrices jointly with quantization parameters. SpinQuant can better tailor transformations to model-specific distributions, albeit at a higher calibration cost. DuQuant [72] extends the rotation approach by using dual transformations: a combination of rotations and permutations. DuQuant first applies a "zigzag" permutation, reordering channels to even out the activation outliers across rotation blocks, followed by a minor rotation to *smooth* the activation landscape. These methods highlight a rich design space where orthogonal transformations – whether fixed, random, or learned – serve as a preprocessing tool to make quantization both more robust and more aggressive, opening the door to fully 4-bit quantized models across weights, activations, and KV cache.

Outlier-free 4-Bit inference in rotated LMs (QuaRot) [70] builds on the principle of output-invariant orthogonal transformations by strategically inserting randomized Hadamard matrices at multiple locations within each transformer block. These transformations aim to uniformly redistribute activation energy and suppress large outliers that typically impede low-bit quantization. Unlike earlier methods that focused solely on weight transformations, QuaRot introduces a mix of *offline* (fused) and *online* (runtime) transformations across the attention and feed-forward pathways, each targeting a distinct quantization bottleneck.

Specifically, QuaRot introduces a global rotation matrix $\mathbf{Q}$ to the residual stream between transformer blocks, which is folded into the pre- and post-projection weights of the FFN. Within the FFN itself, a randomized Hadamard transformation $\mathbf{H}$ is applied online before the down-projection and absorbed post-hoc into the weight matrix. In the attention path, two uses of $\mathbf{H}_{dh}$ target value projections and KV cache activations: it is first fused into the value and output projection matrices using Kronecker structure, and then applied online to queries and keys *after* RoPE, which prevents folding due to positional encoding interference. This online transformation is later inverted to preserve attention semantics. These combined mechanisms enable QuaRot to reduce quantization error across weights, activations, and KV buffers, allowing for end-to-end 4-bit quantization with minimal accuracy loss on LLaMA and OPT models.

By combining rotation-aware preprocessing with efficient runtime rotations, QuaRot enables uniform 4-bit quantization across weights, activations, and KV cache buffers with virtually no accuracy drop in large-scale LMs, such as Llama-2 and OPT. Its design showcases how orthogonal transformations, when inserted with careful awareness of transformer structure, can yield significant reductions in quantization error without compromising compatibility or inference speed.

2.4.4 Other quantization methods

Not all methods neatly fall within the three families of methods described above, and some recent work utilizes a combination of these principles to achieve the best quantized accuracy [73]. Any-Precision LM [37] is another PTQ framework that addresses the need for different-sized LMs without incurring the high memory and computational costs associated with storing separate models. This is particularly useful when the same LM has to be served on devices with heterogeneous hardware or latency requirements.

Any-precision LLM: low-cost deployment of multiple, different-sized LLMs

Unlike conventional quantization techniques that produce a fixed-precision model, Any-Precision LM extends the concept of Any-Precision DNNs to LMs, enabling a single model to dynamically support multiple bitwidths. The core idea behind Any-Precision LM is to store an n-bit quantized model in a way that enables the derivation of lower-bit models ($(n-1)$-bit, $(n-2)$-bit, ...) by selecting only the most significant bits (MSBs). This approach allows for adaptive precision selection at runtime, optimizing the trade-off between model quality and inference latency. The framework employs incremental upscaling, starting from a low-bit seed model and progressively refining it to higher bitwidths, along with a bitplane-based memory representation. In this representation, each bit position of the quantized weights is stored separately, enabling efficient access to different precision levels. Mathematically, the Any-Precision Quantization process can be expressed as follows. Given a weight matrix W, the quantized model at bitwidth b is obtained as:

$$Q_b(W) = \text{Truncate}(Q_n(W), b) \tag{17}$$

where $Q_n(W)$ represents the highest-precision quantized model (parent model), and $\text{Truncate}(\cdot, b)$ extracts the most significant b bits from each quantized weight. To minimize quantization error, Any-Precision LM employs a non-uniform quantization strategy based on clustering (e.g., SqLLM [74]), which divides weight centroids into sub-clusters during upscaling, preserving weight distributions across bitwidths.

2.5 Discussion

To better understand the relative strengths and trade-offs between quantization methods, we compare the three main families, OBS-inspired methods, equivalent transformations (ET), and rotated LMs, along three critical dimensions: *computational effort*, *quantization scope*, and *deployment complexity*.

Computational effort. OBS-inspired methods are the most computationally demanding among the three families. Techniques like GPTQ and OAC require computing approximate second-order statistics, such as the Hessian of the layer outputs, to guide quantization. Despite the approximations and factorization introduced by the authors, e.g. Cholesky factorization or block-wise Hessian, these approaches still require significant calibration time and memory overhead. In contrast, equivalent transformation methods are much lighter: AWQ [36], SmoothQuant [39], and Outlier Suppression+ [65] rely only on statistics from a small calibration set (e.g., max activations or layer norms), and perform inexpensive operations such as per-channel scaling or shifting. Rotation-based methods fall somewhere in between. When using fixed, untrained orthogonal transforms, such as Hadamard matrices in QuaRot or QuIP, the preparation cost is negligible. However, methods such as SpinQuant, which learn rotations jointly with quantization parameters, require gradient-based optimization, resulting in higher offline cost akin to light fine-tuning.

Quantization scope. OBS-based techniques, including GPTQ and OAC, are primarily focused on quantizing weights. They operate on individual linear layers and do not directly address the quantization of activations or the attention KV cache, which are typically kept in higher precision (e.g., FP16 or INT8). In contrast, ET methods offer greater

flexibility. SmoothQuant [39] and OmniQuant [75] explicitly target both weights and activations by scaling one to make the other more quantizable, and can achieve full W8A8 or W4A8 quantization. However, these methods generally do not extend to KV cache quantization. Rotation-based methods offer the most comprehensive coverage. QuaRot [70], for instance, enables uniform 4-bit quantization of weights, activations, and KV cache by carefully applying orthogonal transformations to suppress outliers across all components. As such, they represent the most complete solution when extreme compression is needed across the entire model pipeline.

Deployment complexity. From a deployment standpoint, ET and OBS-based methods are the most attractive: they modify weight magnitudes or normalization layers offline, and the quantized model requires no special operations during inference. In contrast, rotated LMs can introduce deployment challenges. While some rotations can be absorbed into weights offline, others, particularly those applied to activations or KV cache, must be executed online during inference. This adds runtime overhead and may require custom kernels to implement fast Hadamard transforms or channel permutations. Nevertheless, these costs are often modest and justifiable in settings where full low-bit quantization is essential. Some methods, such as DuQuant, attempt to strike a balance by limiting online operations to lightweight block-wise rotations or permutations that are hardware-friendly.

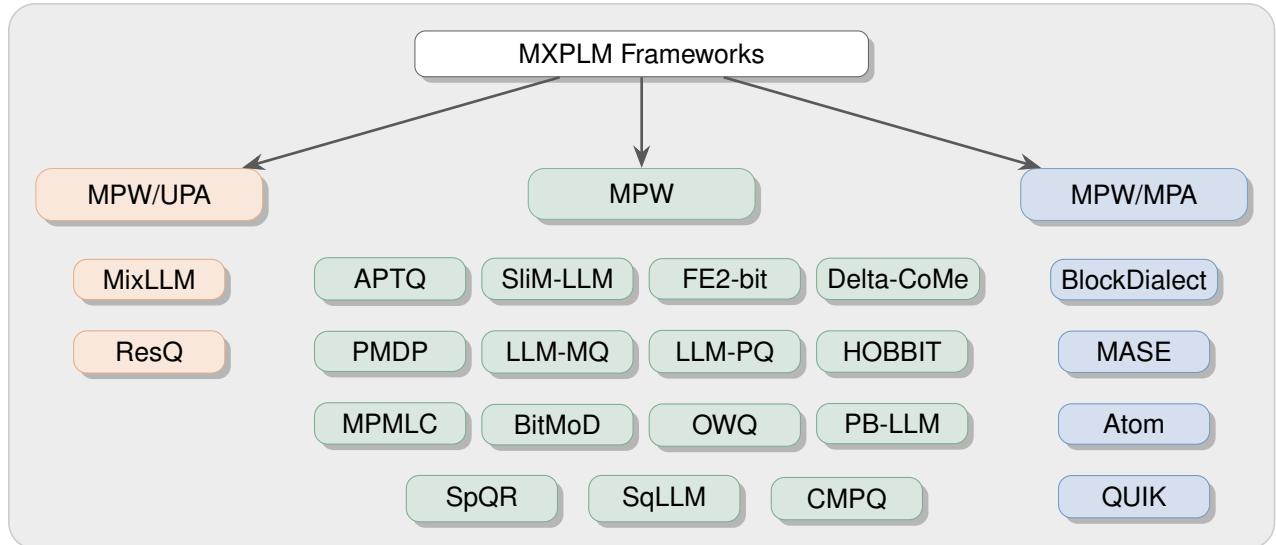# 3 Mixed-precision language model frameworks

Figure 4: Overview of mixed-precision LM (MXPLM) frameworks; UPW/UPA: uniform-precision weights/activations, MPW/MPA: mixed-precision weights/activations.

In this section, we summarize different recent mixed-precision language model (MXPLM) frameworks in the literature. We begin by clarifying the terminology of mixed-precision. In general, mixed-precision refers to the practice of allocating different bitwidths to various numerical elements, such as weights, activations, and key-value (KV) caches, either across layers or within a single tensor. In this work, we focus on mixed-precision across layers (*inter-layer*) and across tensors of the same layer (*intra-layer*).

To establish clarity, we define a "layer" in the context of LMs as a single transformer block, which typically includes a multi-head attention mechanism (projection, attention, linear) and a feed-forward network. Suppose a single weight tensor within such a transformer layer, for instance, the projection weights for keys ($Projection_K$), is partitioned into groups, and each group is quantized to a different precision. In that case, this constitutes *intra-layer* mixed-precision. Since this approach inherently varies precision within a layer, it also naturally subsumes the *inter-layer* case, where different transformer layers have different uniform precisions.

Although it is common in the literature to refer to models with uniformly but differently quantized weights and activations (e.g., W-INT4/A-INT8) or uniformly quantized weights only (e.g., W-INT4/A-FP16) as "mixed-precision," we do not adopt this convention.. Our focus is on mixing precisions amongst weights within and across layers of the model. As such, we define three main categories of the MXPLM framework below:

1. *Mixed-precision* weights (MPW): where the weights in the model are quantized using mixed-precision while activations are left in full-precision, e.g., W-int{4,8}/A-FP16.

2. *Mixed-precision* weights & *uniform-precision* activations (MPW, UPA): where weights and activations undergo mixed and uniform-precision quantization, respectively. e.g, W-int{4,8}/A-int8.

3. *Mixed-precision* weights & *mixed-precision* activations (MPW, MPA): where weights and activations are quantized with mixed-precision, e.g. W-int{4,8}/A-{int8, BF16}.

Fig. 4 shows the mixed-precision LM frameworks based on these categories. In the rest of this section, we elaborate on the formulation of the mixed-precision bit allocation in the different SOTA MXPLM frameworks shown in Fig. 4. We note that for our surveyed MXPLM works below, all frameworks are intra- and hence also inter-layer unless specified otherwise in their corresponding text.

3.1 Mixed-precision weights frameworks

Here, we start with an overview of the frameworks with mixed precision quantization for weight parameters.

3.1.1 SliM-LLM: salience-driven mixed-precision quantization for LLMs

Huang et al. [76] propose SliM-LLM, a mixed-precision framework that can be integrated into backbone PTQ methods such as GPTQ [35] and OmniQuant [75] (with the latter yielding $SliM\text{-}LLM^+$). The key idea is to quantize LMs using structured mixed-precision. SliM-LLM approximates the Hessian of each weight matrix from a small calibration set, and leverages this to compute weight saliency. Columns are grouped in blocks of 128, after which two steps are applied: (1) *salience-determined bit allocation* (SBA) and (2) *salience-weighted quantizer calibration* (SQC).

In SBA, each group is ranked by average salience and assigned a higher or lower bitwidth, while maintaining an overall average bitwidth of 2 or 3. Salient groups thus receive more bits, whereas less important groups are compressed more aggressively. The allocation is determined via a double-pointer search that solves:

$$\arg \min_{g_1,\ldots,g_n} D_{KL}\left(xw_f^\top \,\Big\|\, x\,Q\left(w_f \mid [g_1,\ldots,g_n]\right)^\top\right) \text{ subject to } |G_{N-1}| = |G_{N+1}|, \tag{18}$$

where g_i is the bitwidth of group i, $Q(\cdot)$ denotes group-wise mixed-precision quantization, and D_{KL} is the KL divergence between original and quantized outputs. The constraint $|G_{N-1}| = |G_{N+1}|$ ensures that the average bitwidth equals N (typically 2 or 3). In $SliM\text{-}LLM^+$, SBA is integrated into OmniQuant with its learnable quantizer.

Even within a group, certain weights may remain disproportionately salient. To address this, SQC amplifies such "local outliers" by introducing a calibration parameter γ into the quantization interval:

$$\Delta = \gamma \cdot \frac{w_{\max} - w_{\min}}{2^N - 1}, \quad z = -\left\lfloor \frac{\gamma w_{\min}}{\Delta} \right\rfloor. \tag{19}$$

This expansion of the quantizer's solution space is optimized by minimizing the saliency-weighted quantization loss:

$$\arg \min_{\gamma} \; \mathcal{L}_s\big(w_s, \text{dequant}(\hat{w}_{sq}, \Delta, z)\big) + \mathcal{L}_u\big(w_u, \text{dequant}(\hat{w}_{uq}, \Delta, z)\big), \tag{20}$$

where $\mathcal{L}$ is the ℓ_2 loss, w_s/w_u are salient and less salient weights, and $\hat{w}_{sq}/\hat{w}_{uq}$ their quantized versions. Salient components are identified via a 3σ rule on the saliency distribution. The search for γ is performed over $[1 - \lambda, 1 + \lambda]$ with $2n$ candidates (empirically, $\lambda = 0.1$, $n = 50$). Importantly, SQC shares the same quantizer for w_s and w_u, introducing no extra parameters or inference overhead.

By combining SBA's global saliency allocation with SQC's local refinement, SliM-LLM jointly enhances global and local quantization awareness of salient weights. Its group-wise (structured) design ensures hardware efficiency, avoiding scatter-gather codebooks or fine-grained bitmaps. When integrated into PTQ backbones like GPTQ [35] or OmniQuant [75], it produces 2–3 bit LMs that maintain strong performance while running efficiently on GPUs.

3.1.2 LLM-MQ: mixed-precision quantization for efficient LLM deployment

LLM-MQ [56] is a mixed-precision post-training quantization method for LMs, built on two main ideas: *sparse outlier protection* and *layer-wise precision allocation* using first-order (gradient-based) sensitivity. First, it detects outlier weights and keeps 0.5% of them in FP16 (compressed sparse row, CSR, format) while quantizing the rest of the outliers into INT2. Second, LLM-MQ assigns the bitwidth per layer to either 2, 3, or 4 bits, given a certain target memory

budget by leveraging a first-order Taylor approximation of the model's output loss. Specifically, it measures how much the loss changes if the ith layer is quantized to b bits, denoted as $s_{i,b}$. Then it solves an integer programming problem:

$$\arg \min_{\{c_{i,b}\}} \sum_{i=1}^{N} \sum_{b \in \{2,3,4\}} c_{i,b} \, s_{i,b}$$

$$\text{s.t.} \quad \sum_b c_{i,b} = 1, \quad \sum_{i=1}^{N} \sum_b c_{i,b} \, M\big(Q_b(W_i)\big) \ \leq \ B, \tag{21}$$

where N is the LM's layer number, $c_{i,b} \in 0,1$ is an indication of whether layer i uses bitwidth b, $M(\cdot)$ is the memory usage of quantized weights, and B is the target weight memory budget. Solving this yields per-layer bitwidths that minimize total quantization error while respecting the budget. Finally, LLM-MQ supplies GPU kernels that fuse dequantization with GEMV and organize 2/3/4-bit weights in efficient layouts, thereby achieving an end-to-end speedup over FP16 baselines.

3.1.3   PMDP: progressive mixed-precision decoding for efficient LLM Inference

Chen et al. [77] introduce PMPD which combines two complementary strategies. The first is a *phase-aware scheme*, where weight precision differs between the prefill phase and the decoding phase. The second is a progressive mixed-precision scheme, where the level of precision is adjusted across tokens in the sequence. In this setting, tokens that appear earlier are processed with higher precision, while tokens that appear later are more resilient to approximation and are, therefore, assigned lower precision. For example, the prefill phase may use 3-bit weights, and during decoding, the model gradually transitions from three 2-bit weights as the token position increases.

These two strategies work together to maintain generation quality while improving efficiency. To support precision switching, PMPD introduces two kinds of runtime schedulers: one that is *prompt-agnostic* and static, and another that is *task-agnostic* and learned. Because precision changes both across phases and across tokens, the weights within a layer can be quantized to different precisions at different points in the inference process.

The method operates in two stages. In the *offline* stage, quantized variants of the model are generated, the phase-aware precision allocation is determined by selecting the lowest precisions that still meet the target quality, and the scheduler is configured. In the deployment stage, the scheduler assigns the chosen precisions to phases and triggers switching for each prompt. To avoid additional memory requirements when storing multiple weight precisions, PMPD relies on Any Precision LLM [37].

The scheduler is designed to minimize the average bitwidth subject to a quality constraint. Considering decoding up to an output length OL, the search space includes switching times $\{st(p)\}$ for each precision p. The optimization problem is given as

$$\min st(p), \quad \forall p \in P \setminus \{p_{\min}\}$$

$$\begin{aligned} \text{s.t.} \quad & q_{\text{ref}} - \epsilon \ \leq \ q(S), \\ & 0 \ \leq \ st(p) \ < \ OL, \quad p \in P, \\ & p > q \ \implies \ st(p) \ \leq \ st(q), \quad p, q \in P \end{aligned} \tag{22}$$

where $q(\cdot)$ measures the quality of the scheduler S, P is the set of available precisions, ϵ is the tolerated quality drop, and $st(p)$ denotes the switching step for precision p.

By progressively lowering precision from three to two bits across the decoding process, PMPD reduces memory bandwidth demands, which are a key bottleneck in autoregressive generation. This yields faster inference on resource-constrained devices while maintaining performance close to the original model.

3.1.4   APTQ: attention-aware post-training mixed-precision quantization

APTQ [58] extends the second-order Hessian-based approach of GPTQ [35] to the broader structure of transformer attention. Instead of treating each linear projection independently, APTQ formulates quantization at the level of the entire attention mechanism. This shift allows it to account for interactions between the Q, K, V, and O matrices, capturing dependencies that are otherwise neglected.

Formally, APTQ defines an objective over the multi-head attention function F, seeking quantized weights $\widehat{W}$ that minimize the reconstruction error:

$$\arg \min_{\widehat{W}} \| F(W, X) - F(\widehat{W}, X) \|_2^2, \tag{23}$$

To solve this problem, APTQ employs a Levenberg–Marquardt approximation [78] of the Hessian, keeping first-order derivative components while retaining sufficient curvature information for effective error correction. This design makes the optimization tractable while preserving the benefits of second-order updates.

For each projection in the attention block, APTQ derives both first- and second-order gradients with respect to quantized weights. Quantization error is then corrected by propagating updates to the remaining unquantized weights using the Hessian inverse:

$$E = -\frac{W_q - \text{quant}(W_q)}{\left[H_{\widehat{W}}^{-1}\right]_{qq}}, \qquad \delta_F = E \cdot \left(H_{\widehat{W}}^{-1}\right)_{:,q}, \tag{24}$$

where E is the local quantization error for group q, w_q are the original weights of group q, and δ_F denotes the propagated correction applied to the remaining unquantized weights of the layer . This formulation mirrors the reconstruction updates in GPTQ but adapts them to the multi-head attention setting.

Beyond improving attention quantization, APTQ also incorporates a Hessian-trace-based mixed-precision allocation. The trace of the Hessian provides a measure of sensitivity, and layers with larger values are assigned higher bitwidths. In practice, APTQ allocates 4 bits to the most sensitive projections, most notably the K matrices, while compressing less sensitive components to 2 bits. Let R denote the fraction of 4-bit weights across the model. The average precision is then given by:

$$\text{Average bits} = 4R + 2(1 - R), \tag{25}$$

enabling a tunable trade-off between compression and accuracy.

This hybrid allocation scheme allows APTQ to push LMs into the 2–4 bit regime with limited accuracy loss. By integrating attention-aware objectives with Hessian-guided sensitivity analysis, APTQ demonstrates how structured mixed-precision quantization can reduce model size while maintaining performance, offering a practical path for deploying LMs under tight memory and latency constraints.

3.1.5 MPMLC: harnessing DRAM and SSD for sustainable and accessible LLM inference with mixed-precision and multi-level caching

MPMLC [79] introduces *M2Cache*, a co-design framework that combines mixed-precision quantization with multi-level caching to enable inference on commodity servers with limited GPU memory. The method utilizes a pre-trained predictor, Deja Vu [80], to dynamically estimate neuron activity. Important neurons are executed at higher precision (e.g., FP16), while less active ones are quantized more aggressively or offloaded to DRAM. This selective allocation conserves GPU HBM capacity without sacrificing accuracy on critical paths.

To support this dynamic scheme, MPMLC extends the memory hierarchy beyond HBM. A GPU-managed least recently used (LRU) cache reduces transfers between DRAM and GPU memory by keeping frequently accessed neurons in HBM. When both HBM and DRAM are insufficient, SSDs serve as a final cache tier, providing additional capacity at the cost of latency. By combining precision-aware execution with hierarchical caching, MPMLC enables the execution of large LMs on hardware that would otherwise be unable to host them.

Precision allocation is further refined through an uncertainty-guided search. The method evaluates how different ratios of low- and high-precision neurons affect decoding stability using the uncertainty score:

$$\text{UQEst}(\text{LM}, r_{\text{low}}, r_{\text{high}}) = -\sum_{i>j} \sum_k p_k^i \log(p_k^i), \tag{26}$$

where j is the prompt length, p_k^i is the probability assigned to token k at generation step i, and $r_{\text{low}}, r_{\text{high}}$ are neuron ratios at different precisions. This iterative process identifies a balance between memory savings and output reliability. By retaining only the most essential neurons in higher precision and caching the rest, MPMLC provides a sustainable and accessible solution for LLM deployment on resource-limited GPUs.

3.1.6 Delta-CoMe: training-free delta-compression with mixed precision

Delta-CoMe [81] targets the efficient compression of *delta weights*, i.e., the differences between a fine-tuned model and its base LLM. Instead of retraining, the method applies a quantization strategy guided by the statistical properties of these deltas. Singular Value Decomposition (SVD) reveals that delta weights follow a long-tailed spectrum: a few large singular values dominate, while the rest decay rapidly. Delta-CoMe exploits this structure to assign precision adaptively.

The framework allocates higher precision to singular vectors associated with large singular values, since these components carry most of the performance-critical information. Vectors linked to smaller singular values are quantized into

lower-bit formats, and those corresponding to negligible singular values can be pruned entirely (0-bit). This creates a natural hierarchy of mixed-precision representations aligned with the importance of each component.

Formally, for a delta matrix, indices r_{begin} and r_{end} define the quantization range of singular vectors, and the precision assignment must satisfy:

$$k \times (r_{\text{end}} - r_{\text{begin}})(h_{\text{out}} + h_{\text{in}}) = 16 \times \alpha \times h_{\text{out}} h_{\text{in}}, \tag{27}$$

where k is the bitwidth for a group of singular vectors, h_{out} and h_{in} are the output and input dimensions, and α is the target compression ratio. This constraint ensures that mixed-precision allocation meets the desired memory footprint.

Delta-CoMe evaluates three configurations: *single precision* with all singular vectors at 3-bit, *double precision*, a split between 3-bit and 8-bit representations, and *triple precision*, a mix of 8,3,2-bits, depending on singular value magnitude.

Empirical results on mathematical reasoning tasks show that the triple-precision scheme achieves the best trade-off between accuracy and compression, outperforming uniform quantization baselines. By aligning quantization with the intrinsic structure of delta weights, Delta-CoMe provides a training-free yet effective strategy for compressing fine-tuned LLMs.

3.1.7 FE2-bit: fast and efficient 2-bit LLM inference on GPU

Li et al. [82] propose FE2-bit, a framework that enables efficient 2-bit inference on GPUs. The method addresses three main challenges: (i) accuracy loss from uneven weight distributions when quantized uniformly, (ii) speed degradation caused by sparse outliers, and (iii) heavy computational overhead during GPU dequantization.

To mitigate these issues, FE2-bit adopts a group-wise mixed-precision scheme guided by sensitivity analysis. Each weight matrix is divided into groups of size M, and the sensitivity of group i is computed as:

$$S_i = \sum_{m=1}^{M} \sum_{j=1}^{OC} \frac{w_{Mi+m,j}^2}{[H^{-1}]_{Mi+m}^2}, \tag{28}$$

where $\mathbf{H}^{-1}$ is the offline-computed inverse Hessian, $w_{Mi+m,j}$ is the weight of element m in group i for output channel j, and OC is the number of output channels. Groups with high sensitivity are quantized to 4 bits, while low-sensitivity groups are quantized to 2 bits. A small subset of very large weights is further preserved as 16-bit sparse outliers to reduce error without substantially increasing average bitwidth.

The framework refines this quantization through QAT, which reduces residual error. The same sensitivity analysis used for weights is then extended to input channels: channels with wider dynamic ranges are assigned 4-bit precision, while others remain at 2 bits. To support GPU deployment, FE2-bit incorporates memory alignment strategies to prevent accuracy degradation and minimize bitwidth overhead.

A key bottleneck in 2/4-bit inference is dequantization to 16-bit before matrix multiplication. FE2-bit introduces an asynchronous dequantization pipeline that utilizes shared memory to overlap the computation of group scales with the loading of quantized weights. This process is further accelerated by CUDA primitives that parallelize partial result reductions inside a warp. Together, these optimizations ensure that low-bit inference does not incur excessive runtime overhead.

Finally, FE2-bit carefully manages sparse outliers. Their ratio is capped at 0.5%, and they are preferentially preserved in 2-bit groups, where their impact on accuracy is greatest. This balance of sensitivity-guided precision, GPU-aware optimizations, and controlled outlier retention allows FE2-bit to achieve accurate and efficient 2-bit inference for LLMs.

3.1.8 LLM-PQ: serving LLMs on heterogeneous clusters with phase-aware partition and adaptive quantization

LLM-PQ [83] enables inter-layer mixed-precision inference of LLMs on heterogeneous GPU clusters. Its goal is to reduce resource demands and computational costs by jointly optimizing precision assignment and model partitioning. The framework integrates two components: an *offline assigner*, which determines layer partition, micro-batch sizing, and quantization bitwidths; and a *distributed runtime*, which executes the plan to perform inference. The assigner relies on two complementary cost models. The first is an analytical memory model that predicts GPU memory occupation under different mixed-precision assignments. The second is a latency model that estimates the execution time of each partition during the two critical inference phases: *prefill* and *decode*. Latency prediction is based on a linear regression model calibrated for the target hardware.

To measure the sensitivity of each layer to quantization, LLM-PQ introduces a *variance indicator*, derived from an upper bound on output variance introduced by quantization. For layer i and bitwidth b, the sensitivity is defined as:

$$w_{i,b} = \sum_{o \in O_i} D_{W_o} \left(S_{W_o}(b_i) \right)^2 G(X_o), \tag{29}$$

where O_i is the set of linear operators in layer i, W_o denotes the weights of operator o, X_o is the input feature, and $G(\cdot)$ captures variance under deterministic or stochastic rounding. This metric ranks layers by their robustness to different quantization precisions and guides the allocation of bitwidths.

Using this indicator, LLM-PQ formulates the precision assignment and layer partition problem as an Integer Linear Programming (ILP) objective:

$$\min_Z \left(\left\lceil \frac{B}{\eta} - 1 \right\rceil T_{\max}^{\text{pre}} + \left\lceil \frac{B}{\xi} - 1 \right\rceil (n-1)T_{\max}^{\text{dec}} + T_{\text{pre}} + T_{\text{dec}} \right) + \theta \sum_{j=1}^{N} \sum_{i=1}^{L} \sum_{b \in \text{BITs}} z_{i,j,b} w_{i,b}, \tag{30}$$

where $z_{i,j,b}$ is a binary variable indicating if layer i is placed on device j at bitwidth b, Z is the set of bitwidth assignments, n is the number of generated tokens, and B, η, and ξ denote the global batch size, prefill micro-batch size, and decode micro-batch size, respectively. The T terms represent execution latencies in prefill and decode phases, while θ balances quality against acceleration. Constraints ensure that (i) each layer is assigned to exactly one device and precision, and (ii) per-device memory capacity is not exceeded.

To solve the ILP, LLM-PQ uses an off-the-shelf solver GUROBI [84]. To address scalability challenges, LLM-PQ employs several optimizations: pruning by enumerating micro-batch sizes in the prefill phase, grouping layers to reduce solution space, and a *bitwidth transfer heuristic* that exploits GPU performance asymmetries to swap precision assignments across devices while respecting memory constraints.

In addition, the authors propose an efficient plugin for on-the-fly quantized model loading, which reduces the required DRAM for model loading and improves recovery speed from potential failures. Particularly, they overlap the disk-to-CPU weight loading time with the on-GPU model quantization and the CPU-to-GPU memory copy, thereby determining the granularity of processed weights at runtime.

3.1.9 HOBBIT: a mixed-precision expert offloading system for fast MoE inference

HOBBIT [85] accelerates inference of MoE-based LLMs on memory-limited devices by combining mixed-precision expert loading, prefetching, and caching. The framework builds on sparse MoE layers [86], where feed-forward networks (FFNs) serve as experts, and introduces multiple precision versions: FP16 experts can be replaced by INT4, and INT8 experts by INT2. When a cache miss occurs, HOBBIT loads low-precision experts (INT4/INT2) in place of less important high-precision ones, thereby reducing loading latency without significant accuracy loss.

At the token level, HOBBIT employs a *Dynamic Expert Loader* to decide which experts should be loaded in high or low precision. Expert importance is estimated using the magnitude of gating weights obtained during the Top-k expert selection. For a token input x, the unimportance degree of expert e_i is defined as:

$$s_{e_i}(x) = \begin{cases} 0, & i = 0, \\ \sum_{j < i} \|G(x)_{e_j}\|, & i > 0, \end{cases} \tag{31}$$

where $\|G(x)_{e_j}\|$ is the gating weight magnitude of expert e_j. If $s_{e_i} > T_1$, expert e_i is loaded in low precision, since its contribution is minimal. If $s_{e_i} \le T_1$, it is loaded in high precision. A second threshold is also introduced to skip unimportant experts entirely. This mechanism allows HOBBIT to dynamically assign precision to experts based on runtime input during the prefill phase.

At the layer level, an *Adaptive Expert Predictor* exploits the strong similarity of gating inputs across adjacent layers to prefetch future experts in mixed precision with minimal overhead. At the sequence level, a *Multidimensional Cache Manager* combines four replacement policies: Least Recently Used (LRU), Least Frequently Used (LFU), Least High-Precision Frequently Used (LHU), and Farthest Layer Distance (FLD). Each expert t is assigned a priority

$$p_t = w_{\text{lru}} p_t^{\text{lru}} + w_{\text{lfu}} p_t^{\text{lfu}} + w_{\text{lhu}} p_t^{\text{lhu}} + w_{\text{fld}} p_t^{\text{fld}}, \tag{32}$$

where the weights w. are hyperparameters. The expert with the lowest priority is evicted on cache replacement.

HOBBIT is implemented on Llama.cpp by redistributing model weights and computation patterns. All non-expert weights and a fraction of multi-precision experts are stored in GPU memory, while full expert weights reside in CPU memory or SSD. This design, coupled with dynamic precision-aware loading, enables efficient MoE inference on edge-class hardware. Evaluated on Mixtral and Phi-MoE across Jetson Orin and RTX-4090 setups, HOBBIT's $FP16 + INT\{2, 4, 8\}$ mix cuts expert-loading I/O by up to $4\times$ and yields $9.93\times$ decoding speed-up ($3 - 4\times$ on desktop GPUs) and $60 - 80\%$ lower prefill latency, with a small accuracy loss.

3.1.10   BitMoD: bit-serial Mixture-of-Datatype LLM Acceleration

BitMoD [87] introduces an innovative algorithm-hardware co-design method aimed at efficiently accelerating LMs via mixed-precision post-training per-group quantization of weights. The authors introduce a new fine-grained data type adaptation which utilizes a different numerical data type to quantize a group of weights at 3-bit and 4-bit precision. Their new data type adaptation includes FP3 and FP4 extensions. For FP3, the extension allows the redundant zero, which is present due to the sign-magnitude representation that has both positive and negative zero, to be replaced by one of the four pre-defined special values.

Four special values, divided into two sets, are chosen to balance encoding overhead and hardware complexity, and the special values must satisfy one of these two properties: 1) some values should fall inside the numerical range of FP3, ensuring they do not change its original absolute maximum, or 2) some values should fall outside the numerical range of FP3 to add asymmetry. The two values which are part of one set and which satisfy the first property (+3 and -3) are denoted as the new data type FP3-ER (ER for Extra Resolution). Since infinite values satisfy the second properties, the authors determine these two values which are part of one set in a way that minimizes the quantization error, and achieves a balanced asymmetry across all weight groups. By observing the quantization error of different values, BitMod adopts +6 and -6 as the special values that satisfy the second property, resulting in a new data type denoted as FP3-EA (EA for Extra Symmetry). Similarly +5 and -5 are added to FP4 to define FP4-ER, and +8 and -8 are added to define FP4-EA. As each weight group can be quantized with only one special value out of the possible four for FP3 or FP4 in addition to the basic values; i.e. those of FP3 and FP4, BitMoD relies on a fine-grained data type adaptation where each group is quantized using a different special value to minimize the Mean Squared Error (MSE):

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(W_{f,i} - W_{q,i})^2,\tag{33}$$

where $W_{f,i}$ and $W_{q,i}$ are full-precision and quantized weight tensors, respectively. Specifically, the adaptation is based on a heuristic algorithm which eventually results in a mixed-precision quantization.

After iterating through all special values and adding a special value to the set of basic values in every iteration, non-linear PTQ quantization is performed, and then the special value that results in the lowest MSE is assigned. It is worth noting that BitMod's algorithm can be vectorized on a GPU to simultaneously assign the best special value for all groups of a weight tensor, and that they build on top of a prior work (VS-Quant [88]) to quantize via second-level quantization the scaling factors to low-precision integers, hence reducing the dequantization cost. On the hardware side, BitMoD implements custom bit-serial processing elements capable of efficiently handling mixed-precision floating-point and INT weights, thus significantly enhancing computational efficiency and reducing hardware overhead. Please refer to the paper for more details on the microarchitecture of BitMoD's proposed custom hardware.

3.1.11   PB-LLM: partially Binarized Large Language Models

PB-LLM [89] introduces a novel approach for quantizing LMs through partial binarization, enabling significant model compression while maintaining language reasoning capabilities. Under PTQ, PB-LLM, now denoted as PB-GPTQ, leverages a Hessian-guided reconstruction of the binarized weight matrix. Adapting concepts from GPTQ [35], and for each column in the weight matrix, it iteratively binarizes the less significant un-salient weights, quantizes the salient weights at higher precision (e.g., INT4), and subsequently applies compensation to the remaining weights. Specifically, the optimization in PB-GPTQ is expressed through minimizing the layer-wise quantization error:

$$\arg\min_{\widehat{W}} \|WX - \widehat{W}X\|_2^2,\tag{34}$$

where $\mathbf{W}$ is the original weight matrix that includes salient weights and un-salient (to-be-binarized) weights, $\widehat{W}$ is the quantized weight matrix, and X represents inputs. To detect the salient weights using the Hessian criterion, PB-LLM calculates the saliency metric inspired by SparseGPT [90] as follows: $v_i = \frac{w_i^2}{[H^{-1}]_{ii}^2}$, where H is the Hessian matrix of the layer-wise quantization error with respect to the weights. Accordingly, the un-salient weights are binarized while the salient weights are quantized to a higher precision, whereby the quantization for both cases is asymmetric per-channel. The iterative process repeats until all the weights are quantized.

In the QAT framework that enhances the reasoning capacity of PB-LLM, PB-LLM freezes salient weights throughout training and optimizes scaling factors to mitigate the quantization error of quantized binary weights. For the first optimization, and at the beginning of QAT, PB-LLM filters out a number of weights from a pre-trained weight matrix and keeps them fixed throughout the training. For the second optimization, and inspired by AWQ [36], PB-LLM analytically derives optimal scaling factors to minimize quantization error in residual binarized weights. The optimal

values of column-wise scaling factor α are computed explicitly as: $\alpha^* = \frac{\|w_F\|_1}{n_{w_F}}$, after minimizing the L2 error. Here, w_F is the vector of full-precision weights, and n_{w_F} is its number of elements. This approach eliminates the need for empirical scaling factor searches, enhancing training efficiency.

3.1.12 OWQ: outlier-aware weight quantization for ffficient fine-tuning and inference of LLMs

The OWQ framework [91] introduces an outlier-aware (activation outliers) mixed-precision technique that combines INT3/4 and FP16 weight precisions to reduce memory and computational demands while minimizing quantization error effectively. The outlier-aware quantization has two steps: 1) OWQ first detects the weak columns (these are not quantized) by defining a sensitivity metric related to the layer-wise Hessian matrix, and 2) it quantizes the remaining weights to low-precision by relying on tuned quantization parameters. The $j - th$ weight column sensitivity metric is computed as follows:

$$\text{sensitivity}_j = \lambda_j \|\Delta W_{:,j}\|_2^2, \tag{35}$$

where λ_j is the $j - th$ diagonal element of the Hessian matrix, and $\Delta W_{:,j} = W_{:,j} - \hat{W}_{:,j}$ represents the error for the $j - th$ output channel. OWQ can utilize this metric to select the top-k sensitive columns if the goal is to choose a specific number (k) of weak columns. OWQ then relies on OPTQ [92] to quantize the rest of the weights in a sequential column-wise manner. OWQ modifies OPTQ by incorporating truncation with min-max quantization. After quantization, OWQ then stores the weak columns as FP16. To further enhance model robustness for certain tasks, OWQ utilizes Parameter-Efficient Fine-Tuning (PEFT) in conjunction with Weak Column Tuning (WCT). Specifically, WCT first quantizes the base model with OWQ and then fine-tunes only the weak columns that were kept in FP16 due to OWQ.

3.1.13 SpQR: a sparse quantized representation for near-lossless LLM weight compression

SpQR [50] introduces a novel Sparse-Quantized Representation designed for near-lossless efficient LM compression. After identifying small groups of sensitive weights and individual outlier weights, SpQR keeps the outlier weights in high precision (16-bits), while compressing all other weights to 3/4 bits. To capture individual outliers, SpQR computes the sensitivity of each weight by measuring the layer-wise squared error between full-precision weights and quantized weights. The per-layer sensitivity metric guiding bit allocation is defined as:

$$s_{ij} = \frac{(w_{ij} - Q(w_{ij}))^2}{2(XX^\top)^{-1}}, \tag{36}$$

where s_{ij} is the sensitivity of weight w_{ij}, X represents the calibration inputs, and $(XX^\top)^{-1}$ is the inverse Hessian matrix. These inputs are collected by running them through the model up to the particular layer. SpQR approximates this sensitivity metric by relying GPTQ [35], which quantizes weight matrices column-by-column while adjusting the not-yet-quantized weights to compensate for the quantization error in each step.

The framework isolates weights identified as outliers, which cause disproportionately large quantization errors, storing them in higher precision, while compressing the remaining majority of weights to 3/4 bits. Specifically, outliers detected via the sensitivity analysis are encoded using a Compressed-Sparse-Row (CSR) format [93] due to their scattered and unstructured nature. To capture small groups of sensitive weights, SpQR employs a bilevel quantization scheme with small groups (typically groups with 8-32 weights, where for each group of 8-32 consecutive weights, there is a separate quantization scale and zero-point), maintaining an efficient storage of quantization statistics by applying asymmetric min-max quantization to the quantization statistics themselves. Additionally, SpQR employs an optimized GPU-based decoding for SpQR format, leveraging the memory-bound nature of autoregressive inference on GPUs to hide decoding overheads with high compression rates. Specifically, SpQR loads the quantized weights and group statistics into SRAM, dequantizes them to 16-bits, and performs matrix multiplication with 16-bit activations. SpQR also proposes a sparse matrix algorithm which handles outliers with load balancing, and runs faster than the cuSPARSE sparse algorithm in PyTorch [94].

3.1.14 SqLLM: dense-and-sparse quantization

What SqLLM [74] is a PTQ framework that introduces two main innovations: 1) a sensitivity-based non-uniform quantization method using weighted K-means clustering guided by second-order information from the Hessian, and 2) a Dense-and-Sparse decomposition that extracts a small percentage of outlier and sensitive weights and stores them in FP16 using an efficient sparse format. SqLLM supports mixed-precision with weight representations composed of INT3/4 and FP16 for outliers, and activations retained in FP16. The optimization objective guiding mixed-precision bit allocation is based on sensitivity K-means clustering, and comprises minimizing the overall perturbation across all layers with respect to the final loss. SqLLM first approximates the objective function to get a form weighted by the scaling factor introduced by the second-order derivative of the Hessian. But since computing the Hessian is costly,

SqLLM further approximates the objective based on the Fisher information matrix. Finally, the authors approximate the Fisher information matrix as a diagonal matrix as follows:

$$Q(w)^* = \arg\min_Q \sum_i F_{ii}(w_i - Q(w_i))^2, \tag{37}$$

where $Q(w)^*$ quantifies how much the model gets perturbed after quantization, F_{ii} is the Fisher information approximation to the Hessian diagonal, and $Q(w_i)$ is the quantized value of weight w_i. This formulation biases the K-means centroids to be closer to more sensitive weights. In addition, the Dense-and-Sparse decomposition is used to filter out the outliers from the weight matrix and enhance the quantization resolution with minimal overhead. Particularly, SqLLM decomposes the weight matrix into a sparse matrix (S) with the outliers (stored in CSR format) and a dense matrix (D) without outliers. SqLLM also identifies sensitive outliers within the isolated sparse matrix by using the Fisher information and retains those weights in FP16. For efficient implementation, the authors develop 3/4-bit CUDA LUT-based kernels for matrix- vector multiplication which load the non-uniformly quantized weights and dequantize them "piece-by-piece" to minimize the utilization of memory bandwidth. The quantized matrices store 3/4-bit indices, corresponding to LUT entries containing FP16 values. After dequantization, all arithmetic is performed in FP16. They also develop kernels for sparse matrix-vector multiplication which load the sparse matrix in CSR format. SqLLM implements balanced hybrid kernels similar to [95].

3.1.15 CMPQ: channel-wise mixed-precision quantization for large language models

CMPQ [96] is a PTQ framework that supports mixed-precision quantization, quantizing non-sensitive weight channels to 2/3/4 bits, while preserving outlier channels in full precision (FP16). CMPQ develops an algorithm to adaptively quantize LMs under any given average bit constraint, applying channel-wise non-uniform quantization, where different weight channels within each layer are quantized to different bitwidths based on their sensitivity, identified using activation distributions. To perform non-uniform quantization, a K-means clustering algorithm is applied to each channel in the weight matrix, where each weight in the matrix is represented by its nearest centroid from the set of K centroids after clustering. CMPQ chooses the value of K depending on the assigned precision to the channel.

To determine the bit allocation for the channels, CMPQ computes the per-channel L2-norm of the activations and uses this metric to allocate precision according to an average bitwidth constraint $b \in [2, 4]$: when the average bitwidth $b > 3$, the most salient channels; i.e. those with large activation norms, are assigned higher precision (4-bit) to boost model performance, when $b < 3$, the less important channels are quantized to lower precision (2-bit) to reduce quantization loss. For the case of $b = 3$ (3-bit quantization), approximately 1% of the most salient weight channels are protected by assigning them 4-bit precision to preserve accuracy. Additionally, CMPQ introduces an efficient outlier protection mechanism that detects both activation-based outliers and a small subset of quantization-sensitive outliers, storing them in a sparse FP16 format to minimize distortion in clustering and centroid alignment during quantization. Specifically, for activation-based outliers, channels corresponding to the top 0.45% largest values in the L2-norm vector of the activation are kept in FP16. For detecting a small subset of quantization-sensitive outliers, CMPQ applies another K-means clustering step before quantization, and accordingly preserves 0.05% of the magnitude-based outliers in FP16.

3.2 Mixed weights, fixed activations quantization frameworks

3.2.1 MixLLM: LLM quantization with global mixed-precision between output-features and highly-efficient system design

MixLLM [57] tackles LM compression by assigning different bitwidths to the output features or channels in linear layers; those with high salience receive higher precision than the rest forming structured mixed-precision. It relies on a global mixed-precision search procedure, where the precision of all output features in all linear layers is determined globally rather than locally. MixLLM relies on a global salience identification step that estimates how much each feature contributes to the final loss. First, Taylor Expansion is used to approximate $l(.)$, the loss function with respect to a single channel, relying on the second-order gradient (the Hessian matrix) with respect to the channel. The Hessian is further approximated by the Fisher information matrix (with respect to a channel) on a calibration dataset. MixLLM then applies a final approximation to get a single vector product, and final approximated saliency metric, S_c, of channel c can be derived as follows:

$$S_c = \frac{1}{|D|} \sum_{d \in D} |g_d^T(c_q - c_0) + \frac{1}{2}(g_d^T(c_q - c_0))^2|, \tag{38}$$

where c_q is the quantized weight of the channel, c_0 represents the original weight, D is the calibration dataset, and g is the gradient of the loss with respect to the channel. First, the global precision search procedure calculates the salience of all output channels of all linear layers. Then, it sorts them in global descending order, and quantizes $N_{largebits}$

channels ($N_{largebits}$ is a global user-defined threshold) to 8-bits, while quantizing the rest of the channels into 4-bits. Group-wise quantization is applied: for 4-bit weights, MixLLM uses group-wise asymmetric quantization, while for the 8-bit weights (including those "high salience" channels in the weight), it employs group-wise symmetric quantization.

Additionally, activations are quantized to 8 bits with symmetric group-wise quantization. It is worth noting that any quantization methodologies (like GPTQ [35], clip search) can be applied independently to quantize into 4-bits and 8-bits. System-wise, MixLLM uses a two-step dequantization where it partially dequantizes the weight, and then multiplies it by the quantized activation utilizing the 8-bit Tensor Core. After that, it multiplies this matrix-multiplication result by the two scales within each group, converting the integer to float instruction into two add/sub instructions, and fusing the integer subtraction into the Tensor Core Matrix Multiply-Accumulate. In addition, MixLLM designs an end-to-end software pipeline of the quantized kernel which overlaps memory transfers with computations, dequantization computation with Single Instruction Multiple Threads (SIMT) Core, and matrix-multiplication computation with Tensor Core for maximum efficiency. It can also minimize the overhead of group-wise dequantization. Moreover, MixLLM executes different sub-problems in parallel on the GPU with CUDA Graph, implementing this function with the fused epilogue of the matrix-multiplication kernel.

3.2.2 ResQ: mixed-precision quantization of large language models with low-rank residuals

ResQ [97] is a PTQ mixed-precision weight, activation, and KV cache quantization framework for LMs which identifies, via Principal Component Analysis (PCA), a low-rank high-variance subspace; i.e., with high activation variances. Then it retains the coefficients of the projections along the basis in 8-bits. The rest of the coefficients are quantized to 4-bits (or 2-3 bits, depending on the target). For each subspace, ResQ applies an invariant random rotation to handle outliers. In particular, ResQ projects weights, activations, and the KV cache into an orthogonal basis, U, constructed as a combination of two rotation matrices having the following properties: 1) the more important components are captured by the low-rank space for high-precision quantization, and 2) the quantization error in the high-precision group and low-precision group is minimal. U is defined as follows:

$$U = PR = [P_l P_h] \begin{bmatrix} R_l & 0 \\ 0 & R_h \end{bmatrix}, \tag{39}$$

where P_l, R_l correspond to low-precision components and P_h, R_h correspond to high-precision components. To reduce the outliers, R_l, R_h are made random orthogonal matrices, and that way the rotated matrices are easier to quantize. ResQ shows that the low-rank subspace for high-precision quantization can be obtained by PCA, while the subspace for low-precision quantization can be obtained using the following: $U_h U_h^T + U_l U_l^T = P h P_h^T + P_l P_l^T = I$, where U_h represents bases of a low-rank space of high-precision components, and U_l represents the complementary subspace of low-precision components.

Quantized activations (X_q) and weights (W_q) are obtained by projecting the input space of activations and weights by U and U^T respectively and quantizing the coefficients. Once the projection matrices are obtained, and during inference, the output of the layer, $X_q W_q$, is calculated by multiplying the weights and activations by U as follows: $X_q W_q = Q_L(X U_l) Q_L(U_l^T W) + Q_H(X U_h) Q_H(U_h^T W)$, where Q(.) is the quantization function. L represents the subscript for low-precision and H is the subscript for high-precision. Weights can be projected and quantized offline, and the projection of an activation is merged to the weight of a previous linear layer.

ResQ introduces four different kinds of projections based on decoder only LM architecture: projection U_A at block boundaries (modifying the inputs across blocks and enabling better quantization), two projections U_B and U_C within the attention block enabling mixed precision quantization of KV cache, and a projection U_D within the feedforward block projecting the activations and weights of $down\_proj$ layer. Only a single projection at block boundaries is shared across all layers while the rest of the projections are generated per layer. ResQ applies per-token asymmetric quantization to activations, per-channel symmetric quantization to weights, and per-head asymmetric quantization to the KV cache. It fuses the projection matrices U_A, U_B, and U_D into adjacent weights and applies GPTQ [35] for weight quantization. Moreover, implement on-the-fly projections, U_D is a Hadamard matrix and U_C and its activations are quantized to a precision of 8.

3.3 Mixed weights, mixed activations quantization frameworks

3.3.1 Atom: low-bit quantization for efficient and accurate LLM serving

Atom [60] is a low-bit weight-activation mixed-precision quantization framework for LM serving that combines: 1) mixed-precision quantization with channel reordering, 2) fine-grained group quantization, 3) dynamic activation quantization, and 4) KV-cache quantization. For mixed-precision quantization, Atom identifies activation outlier channels; i.e., those with high mean value, reorders them and their corresponding weight channels to the end of the

matrix inspired by RPTQ [53], and quantizes them to a higher precision (INT8) compared to the other channels, which are quantized to a lower bitwidth (4-bits).

While weight reordering incurs a one-time cost (since outlier channels can be identified offline using a calibration dataset), activation reordering is performed online. To lower the cost of activation reordering, Atom fuses these reordering operators with prior operators. To tackle the limited representation capability of 4-bit precision, Atom uses group-wise quantization of the non-outlier channels. Specifically, the activation groups are first multiplied by their corresponding weight groups via Tensor Cores. Then, Cuda Cores are used to dequantize the temporary results (with different quantization parameters) to FP16 before performing addition. We note that Atom fuses the dequantization and addition into the GEMM kernel. In addition, Atom uses symmetric dynamic quantization for activations so that activation matrices would have their tailored quantization parameters during inference, where quantization operators are fused into prior operators.

Atom also incorporates GPTQ [35] in an offline stage for weight matrix quantization. Meanwhile, the KV cache is quantized asymmetrically to shrink self-attention's memory overhead, and is dequantized directly before performing FP16 calculation. Implementation wise, Atom relies on a low-precision unit for compute-bound layers, while fusing dequantization with FlashInfer [98], an LM serving library. Moreover, it also uses PagedAttention [99] to enable efficient inference with large batch sizes.

3.3.2   MASE: a dataflow compiler for efficient llm inference using custom microscaling formats

MASE [100] is a dataflow compiler and co-design framework that leverages microscaling (MX) data formats, particularly MXInt, to balance accuracy and hardware efficiency for LM inference on hardware accelerators. A key contribution is the MASE Intermediate Representation (IR), a hardware-aware trainable software which allows exploring software–hardware optimizations like scheduling and parallelism for custom low-bit formats for both weights and activations at granularities ranging from the model level down to the bit level. MASE IR uses the static single-assignment form which enables an easy translation into a dataflow hardware representation. The hardware design attributes carried by a model in MASE IR include the shape of streaming tiles, the streaming order, the interface between hardware and data, and the estimated throughput. These attributes enable parallelism exploration by the model optimizer.

MASE has 44 general and type-independent "analysis and optimization" passes which target the different granularities. For the "quantize" pass, a model is first quantized into a set of user-defined bitwidths. MASE supports both PTQ and QAT. After that the "parallelize" pass explores the best hardware resources for parallelism given a certain hardware resource budget. The "evaluate" pass estimates the accuracy and the area efficiency of the hardware because MASE IR has both software and hardware design parameters. Guided by a hardware-aware cost function, MASE's "search" pass iterates over both quantization and hardware parallelism settings using existing algorithms like Tree-structured Parzen Estimator (TPE) [101] to find mixed-precision assignments.

Eventually, after a given number of iterations, MASE automatically maps the model into a SystemVerilog hardware design for the chosen configuration in the "emit" pass. To enable MASE to support exploration of custom data formats at scale, software emulators and hardware components must be added by users. While software emulators are used to indicate how to quantize/dequantize the value from/to the custom formats/floating-point numbers, MASE provides a Verilog template of a dataflow component with a set of parallelism parameters for each hardware component. MASE then automatically explores hardware designs by sweeping those parallelism parameters restricting the design space. We note that MASE IR provides a compact representation to explore large models up to billions of parameters.

3.3.3   BlockDialect: block-wise fine-grained mixed format for energy-efficient LLM inference

BlockDialect [102] is a block-wise, fine-grained, mixed-format quantization framework designed to boost energy efficiency and accuracy for LM inference at 4-bit precision, by focusing on "how to represent" data distribution over "how to scale". Particularly, it introduces a "formatbook" of FP4 variants called DialectFP4, where each "dialect" adjusts large-magnitude values slightly differently. The dialects for the formatbook were determined based on profiling several models (like Llama3-8B, Llama2-7B, Mistal-7B, OPT-6.7B, and others), where the matrix is first divided into blocks (size 32), then each block is scaled by the shared exponent, and finally the magnitude distribution histograms for each block are accumulated. The profiling analysis reveals that FP4 E2M1 (referred to as FP4) aligns with the observed distribution, and is hence chosen as the dialects' base format.

The authors then choose 16-Dialect DialectFP4 to meet three main criteria: 1) dialects cover all possible maximum magnitudes, 2) on the one hand, each pair of dialects shares the maximum magnitude, while on the other hand, the pairs differ in one large magnitude value, and 3) the unit of these dialects is set to 0.5 to align with FP4. Accordingly, in addition to the 4-bit identifier assigned to each block, each data point in DialectFP4 has 1 bit for sign and 3 bits for index. Unlike weights whose optimal per-block dialect can be determined by calculating the MSE, determining the

optimal per-block dialect for activations is done at runtime via a lightweight, two-stage selection process preceded by a pre-processing stage. In the pre-processing stage, a 5-bit shared exponent is computed, then each element's exponent is adjusted by subtracting the shared exponent.

After that, BlockDialect shifts the mantissa by the adjusted exponent, and truncates the lower bits to form a 5-bit intermediate representation with 3 bits for the integer part and 2 bits for the fractional part. Following the pre-processing stage, BlockDialect determines the block's maximum magnitude by rounding from the second fractional bit and limits the choice to the two dialects (FP4 variants) whose largest magnitudes are the same as the block's maximum. In the second stage and for each one of the two selected dialects in the first stage, BlockDialect looks at how many elements in the block lie within that dialect's "beneficial range," i.e., the specific segment of magnitudes where that dialect is more precise. The beneficial range is calculated as the midpoint between the differing value, its adjacent value, and the differing value of the paired dialect. Whichever dialect can accommodate more elements in this beneficial range is assigned to the entire block. Implementation-wise, since DialectFP4 is compatible with 5-bit integer arithmetic operations, it enables two implementation options. BlockDialect can either use the general INT4 MAC with simple logic to deal with residual bits for 5-bit multiplication or it can design optimized MACs with 4-bit unsigned integer multiplier and an additional XOR gate to deal with the sign bit.

3.3.4 QUIK: towards end-to-end 4-bit inference on generative LLMs

QUIK [103] is a joint mixed-precision PTQ framework for both weights and activations in LMs, achieving speedups while maintaining near-baseline perplexity. QUIK quantizes most weights and activations into a precision of 4 bits, while keeping outliers in higher precision. Its selective-precision approach improves upon GPTQ [35] by identifying the outlier weight columns; i.e. those with the largest $l\infty$ norm, for each layer using a small calibration set. After that, the same outlier indices are used to identify the outlier columns. The outlier weight columns are rearranged to the end of the matrices and are quantized up to the index of the outliers offline. QUIK does not quantize the outliers to preserve them.

At runtime, the activation columns corresponding to the outlier weights identified by the indices are extracted and kept in high precision while the rest of the columns are quantized. By aggregating the quantization errors to the columns kept in high precision (FP16 or BF16), and by excluding weight outliers from the 4-bit quantization scale, QUIK enhances GPTQ quantization. Moreover, QUIK identifies weight clipping thresholds by a linear search to improve end-to-end perplexity. We note that QUIK quantizes the down projection layers into 8 bits rather than 4 bits as these layers are more sensitive to quantization as suggested by the $l\infty$ norm analysis.

While the weight columns kept in full-precision are multiplied by their corresponding full-precision activation columns regularly, the quantized columns go through the following quantized matrix multiplication pipeline: Quick first dynamically applies asymmetric quantization per token for activations, then, by the help of the GPU's INT8/INT4 tensor-cores, performs the matrix multiplication of the dynamically quantized activations and weights (which are quantized offline in a symmetric per output manner). Finally, it dequantizes the result back to full-precision. It also fuses the quantization and dequantization for efficient implementation. In addition, for some experiments, QUIK extends SparseGPT [90] to jointly quantize and sparsify the models, while keeping the outliers in dense FP16.

4 Insights & discussions

4.1 Comparison between MXPLM frameworks

Most existing MXPLM frameworks rely on PTQ due to its lower computational cost, with only a few exploring quantization-aware training QAT (like FE2-bit, PB-LLM, MASE). The selection of weights/activations to quantize is often guided by first- or second-order information, leveraging gradient-based methods such as the Hessian matrix. While weight quantization has been the primary focus, activation quantization remains more challenging, and a common approach is to use a single low-bitwidth setting alongside a higher-bitwidth setting for select outlier parameters. While the goal of MXPLM is to a hardware-friendly implementation, only few works (MASE and LLM-PQ) explicitly consider mixed-precision with hardware in the loop (these frameworks are called hardware-aware). Notably, some works in literature define mixed-precision merely as using different bitwidths for weights and activations while maintaining uniform precision within each category, but that was outside the scope of our definition of mixed-precision. As indicated in section 6.4, due to the lack of hardware support for mixed-precision computation, weights that are quantized to INT4 are usually dequantized to FP32 for GEMM operations, with accumulation typically performed in higher precision formats such as BF16 or FP32 [104, 105].

We henceforth analyze the perplexity/zero-shot accuracy behavior of MXPLM frameworks across LLaMA2-13B and LLaMA3-8B. We omit frameworks that do not evaluate on these models. As shown in Fig. 5a, for LLaMA2-13B, the
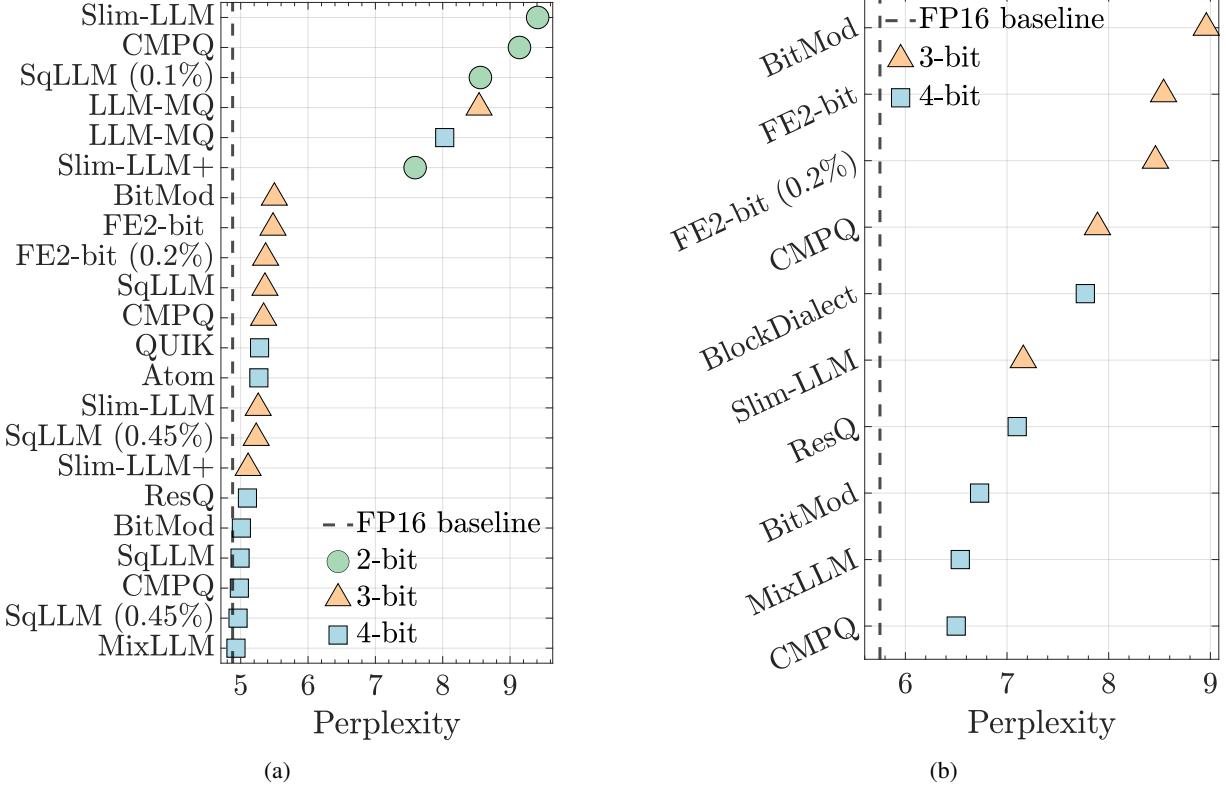
Figure 5: Perplexity of MXPLM frameworks reported on (a) LLaMA2-13B and (b) LLaMA3-8B using WikiText2 for different average precisions. Note: "(x%)" indicates sparsity of $x\%$.

FP16 baseline perplexity is 4.88 and several 4-bit (on average) MXPLM frameworks, notably MixLLM, CMPQ, and SqLLM (0.45% sparsity) achieve near-baseline performance (perplexity ranges between $4.93 - 4.98$), and BitMod and ResQ are also competitive (within $2.66 - 4.51\%$ difference with respect to baseline). The best (perplexity-wise) 3-bit (on average) methods (Slim-LLM+, SqLLM (0.45%), and Slim-LLM) remain below 7.17% difference with respect to FP16. In contrast, the performance of 2-bit average precision frameworks such as Slim-LLM+, SqLLM (0.1%), CMPQ, and Slim-LLM degrades severely as perplexity increases ($> 55\%$ difference with respect to FP16).

For LLaMA3-8B (Fig. 5b, the FP16 baseline perplexity is 5.75. The best performing MXPLM framework, CMPQ, has a 13.04% higher perplexity than the baseline at an average precision of 4-bits, while BlockDialect performs worst among the frameworks at average precision of 4 with a perplexity of 7.77. At an average precision of 3-bits, MXPLM frameworks exhibit larger performance loss, with BitMod approaching a perplexity of 9, indicating that no 2-bit or low 3-bit configuration is competitive with respect to the FP16 baseline.

The best-performing frameworks in both models (MixLLM, SqLLM, CMPQ, BitMod, ResQ) belong primarily to the MPW and MPW,UPA categories, which exploit mixed-precision weights and uniform precision or FP precision for activations, with frameworks at an average precision of 4-bits performing best. MPW,MPA frameworks (Atom, BlockDialect, QUIK) can offer comparative performance with respect to FP16 at an average precision of 4 bits. The perplexity results of all surveyed MXPLM frameworks can be found in Tables 3 and 4, where the best performing framework metric is in bold.

While perplexity provides a useful proxy for evaluating LM compression quality, it does not fully capture downstream task performance. Many real-world applications depend on zero-shot generalization across diverse benchmarks (like commonsense reasoning, natural language inference, or knowledge-intensive QA), where error propagation can be amplified even when perplexity remains close to the FP baseline. Evaluating on zero-shot tasks therefore complements perplexity analysis by exposing robustness and transferability under task-driven conditions. In Tables 1 and 2, we report the presented zero-shot accuracy results across 6 tasks: PIQA, ARC-e, ARC-c, BoolQ, HellaSwag, and Winogrande on LLaMA2-13B and LLaMA3-8B. The highest reported zero-shot accuracies on LLaMA2-13B are 80.42% (BitMod), 76.10% (ResQ), 49.1% (ResQ), 79.70% (ResQ), 78.41% (BitMod), and 72.14% (BitMod) across

#Bits	Method	PIQA	ARC-e	ARC-c	BoolQ	HellaSwag	Winogrande
FP16 [106]	-	80.55	80.52	72.22	77.44	48.98	79.38
2-bit	LLM-MQ	**75.84**	**54.29**	–	–	**68.32**	**65.51**
3-bit	SqLLM	78.62	56.73	–	–	74.53	67.40
	LLM-MQ	79.00	57.79	–	–	75.08	68.59
	BitMod	**79.22**	–	–	–	**76.79**	**72.37**
	FE2-bit	77.20	71.2	–	–	57.6	68.50
	FE2-bit (+0.2%)	77.80	**71.3**	–	–	58.8	68.60
4-bit	SqLLM	78.94	57.70	–	–	76.05	69.14
	LLM-MQ	79.49	58.50	–	–	76.31	69.30
	BitMod	**80.42**	–	–	–	**78.41**	**72.14**
	ResQ	79.10	**76.10**	**49.1**	**79.70**	77.90	69.90
	QUIK	79.22	74.92	48.04	–	78.36	71.90

Table 1: Zero-shot accuracy (%) on LLaMA2-13B across tasks (PIQA, ARC-e, ARC-c, BoolQ, HellaSwag, Winogrande). Frameworks with no reported results are omitted. Note: "(x%)" means a sparsity of "x%".

#Bits	Method	PIQA	ARC-e	ARC-c	BoolQ	HellaSwag	Winogrande
FP16 [106]	–	81.44	80.79	72.85	77.74	53.33	79.16
3-bit	BitMod	**77.91**	–	–	–	**73.56**	70.32
	FE2-bit	75.3	69.4	–	–	54.9	70.3
	FE2-bit (+0.2%)	76.8	**70.3**	–	–	55.4	**70.5**
4-bit	BitMod	**79.98**	–	–	–	**78.49**	**73.09**
	MixLLM	–	–	**53.67**	–	78.2	–
	ResQ	78.3	**75.0**	49.2	72.5	76.5	71.0
	BlockDialect	75.24	71.63	47.18	**77.37**	74.12	66.38

Table 2: Zero-shot accuracy (%) on LLaMA3-8B across tasks (PIQA, ARC-e, ARC-c, BoolQ, HellaSwag, Winogrande). Note: "(x%)" means a sparsity of "x%"

PIQA, ARC-e, ARC-c, BoolQ, HellaSwag, and Winogrande respectively. At an average precision of 4-bits, and on LLaMA3-8B, the highest reported zero-shot accuracies are 79.98% (BitMod), 75.0% (ResQ), 53.67% (MixLLM), 77.37% (BlockDialect), 78.49% (BitMod), and 73.09% (BitMod) across PIQA, ARC-e, ARC-c, BoolQ, HellaSwag, and Winogrande respectively. Again, the best performing frameworks are MPW and MPW, UPA. We note that ResQ which performs well across tasks quantizes weights, activations, and also the KV cache, offering even higher compression gains than frameworks that quantize weights/activations. The zero-shot accuracy of all the surveyed frameworks on different models and across different tasks with the best performing metrics in bold can be found in Table 5.

#Bits	Method	Llama1-7B		Llama1-13B		Llama1-30B		Llama1-65B	
		WikiText2	C4	WikiText2	C4	WikiText2	C4	WikiText2	C4
FP16	-	5.68	7.08	5.09	6.61	4.1	5.98	3.53	5.62
2-bit	Slim-LLM	14.58	32.91	8.87	13.85	7.33	11.27	5.9	10.95
	Slim-LLM+	**9.68**	**14.99**	**7.17**	**10.22**	**6.41**	**9.33**	**5.74**	**7.52**
	PB-LLM	24.61	49.73	17.73	26.93	12.65	17.93	7.85	11.85
3-bit	Slim-LLM	6.4	**6.14**	5.48	**6.05**	4.61	**6.33**	3.99	**5.94**
	Slim-LLM+	**6.07**	7.75	**5.37**	6.91	**4.34**	6.36	**3.72**	5.96
	SqLLM	6.32	7.75	5.6	7.08	4.66	6.37	4.05	5.99
	SqLLM (+0.45%)	6.13	7.56	5.45	6.92	–	–	–	–
	OWQ	6.66	–	5.66	–	4.75	–	4.25	–
	APTQ	6.76	6.24	–	–	–	–	–	–
	FE2-bit	6.61	–	5.92	–	–	–	–	–
	FE2-bit (+0.2%)	6.56	–	5.89	–	–	–	–	–
	Atom	11.77	15.43	8.4	10.81	6.94	9.14	5.89	7.94
4-bit	SqLLM	5.79	7.21	5.18	6.71	**4.22**	**6.06**	3.76	**5.69**
	SqLLM (+0.45%)	**5.77**	7.18	**5.17**	**6.68**	–	–	–	–
	OWQ	5.94	–	5.25	–	4.25	–	3.74	–
	SpQR	5.87	7.28	5.22	6.72	4.25	6.08	**3.68**	5.7
	APTQ	6.45	**5.23**	–	–	–	–	–	–
	Atom	6.16	7.7	5.46	7.03	4.54	6.35	3.89	5.92

Table 3: Perplexity across LLaMA1 model (WikiText2, C4). Note: "(x%)" means a sparsity of "x%"

#Bits	Method	Llama2-7B		Llama2-13B		Llama2-70B		Llama3-8B		Llama3-70B	
		WikiText2	C4	WikiText2	C4	WikiText2	C4	WikiText2	C4	WikiText2	C4
FP16	-	5.47	6.97	4.88	6.46	3.31	5.52	5.75	9.22	2.9	6.85
2-bit	Slim-LLM	16.01	16	9.41	**9.41**	6.28	**7.01**	39.66	110	**9.46**	**15.92**
	Slim-LLM+	**10.87**	18.18	**7.59**	10.24	6.44	8.4	–	–	–	–
	SqLLM (+0.1%)	13.64	–	8.56	–	**5.38**	–	–	–	–	–
	CMPQ	14.37	**15.97**	9.14	11.25	–	–	120	110	–	–
	LLM-MQ	–	–	12.17	–	–	–	–	–	–	–
	PB-LLM	25.37	29.84	49.81	19.82	–	8.95	44.12	**79.21**	11.68	33.91
3-bit	Slim-LLM	6.24	7.74	5.26	**5.26**	3.67	**5.09**	**7.16**	13.1	**4.08**	**8.64**
	Slim-LLM+	**5.94**	7.71	**5.11**	6.9	**3.35**	5.85	–	–	–	–
	SqLLM	6.18	7.72	5.36	6.97	3.77	5.83	–	–	–	–
	SqLLM (+0.45%)	5.96	**7.51**	5.23	6.82	3.63	5.73	–	–	–	–
	CMPQ	6.14	7.66	5.34	6.93	–	–	7.89	**11.3**	–	–
	FE2-bit	6.62	–	5.48	–	–	–	8.54	–	–	–
	FE2-bit (+0.2%)	6.59	–	5.37	–	–	–	8.46	–	–	–
4-bit	SqLLM	5.62	7.12	4.99	6.57	3.41	5.58	–	–	–	–
	SqLLM (+0.45%)	5.57	**7.08**	4.96	**6.54**	**3.39**	**5.57**	–	–	–	–
	CMPQ	5.61	7.1	4.98	6.55	–	–	6.5	9.39	–	–
	MixLLM	**5.55**	–	**4.93**	–	–	–	6.54	9.62	**3.3**	**7.24**
	ResQ	5.8	–	5.1	–	–	–	7.1	–	4.1	–
	Atom	6.03	–	5.27	–	3.68	–	–	–	–	–
	LLM-MQ	–	–	8.03	–	–	–	–	–	–	–
	BlockDialect	6.35	–	–	–	–	–	7.77	–	–	–
	QUIK	5.84	–	5.28	–	3.74	–	–	–	–	–
	BitMod	5.72	7.26	5.01	6.61	–	–	6.73	9.66	–	–

Table 4: Perplexity across LLaMA2 and Llama3 families (WikiText2, C4). Frameworks with no reported numbers are omitted. Note: "(x%)" means a sparsity of "x%"

#Bits	Method	PIQA	ARC-e	ARC-c	BoolQ	HellaSwag	Winogrande
					LLaMA1-7B		
2-bit	Slim-LLM	57.83	33.46	25.09	56.05	36.7	52.64
	Slim-LLM+	**64.96**	**45.66**	**28.67**	**64.59**	**48.86**	**53.35**
3-bit	SpQR	**78.13**	65.87	**38.05**	–	55.27	67.48
	APTQ	74.5	57.9	36.4	–	68.3	65.3
	FE2-bit	76.6	70.4	–	–	72.8	68.5
	FE2-bit (+0.2%)	76.8	**71**	–	–	**73.8**	**68.7**
	Atom	65.56	41.41	30.72	**61.77**	53.19	55.56
4-bit	SpQR	78.45	67.13	38.23	–	56.01	67.48
	APTQ	**78.6**	**72.4**	**44.4**	–	**75.7**	69.3
	PB-LLM	78	69	42.3	–	74.3	**69.7**
	Atom	76.28	52.1	38.99	**69.79**	69.81	63.69
					LLaMA1-13B		
2-bit	Slim-LLM	73.19	47.95	36.27	55.92	63.04	61.79
	Slim-LLM+	**74.15**	**50.26**	**37.04**	**64.31**	**63.57**	**63.11**
3-bit	SpQR	78.73	**73.27**	**42.75**	–	58.22	68.9
	APTQ	74.4	64.1	41	–	71.2	68
	FE2-bit	79.2	72.7	–	–	76.9	71.6
	FE2-bit (+0.2%)	**79.3**	73	–	–	**77.8**	**71.9**
	Atom	70.08	47.94	33.7	**63.46**	62.93	56.75
4-bit	SpQR	78.94	**74.37**	43.17	–	59.02	69.77
	APTQ	**79.9**	73.9	**47**	–	**78.8**	**72.1**
	PB-LLM	–	–	–	–	–	–
	Atom	77.69	57.58	42.92	**67.46**	73.77	68.51
					LLaMA1-30B		
2-bit	Slim-LLM	75.52	51.3	39.29	62.01	66.1	64.07
	Slim-LLM+	**76.31**	**54.07**	**39.79**	**63.35**	**67.14**	**64.93**
3-bit	SpQR	**80.74**	**74.75**	**46.93**	–	61.96	**73.32**
	Atom	72.47	49.54	37.8	**66.75**	**66.99**	60.14
4-bit	SpQR	**81.01**	**76.05**	**47.18**	–	62.5	72.93
	Atom	78.73	58.92	45.82	**68.47**	**77.4**	**73.09**
					LLaMA1-65B		
2-bit	Slim-LLM	77.09	53.72	40.25	77.51	72.05	**70.91**
	Slim-LLM+	**78.06**	**53.9**	**41.18**	**78.33**	**75.59**	69.99
3-bit	SpQR	**81.18**	**74.37**	**45.05**	–	63.54	**76.09**
	Atom	75.84	51.43	41.3	**74.07**	**72.22**	64.33
4-bit	SpQR	**81.56**	**75.25**	**46.93**	–	63.76	**76.95**
	Atom	80.41	58.12	45.22	**82.02**	**79.1**	72.53
					LLaMA2-7B		
3-bit	BitMod	**77.53**	–	–	–	**72.68**	66.22
	FE2-bit	75.9	66.9	–	–	51.3	66.4
	FE2-bit (+0.2%)	76.1	**67.2**	–	–	52	**66.8**
4-bit	BitMod	78.45	–	–	–	75.43	**68.19**
	ResQ	77.9	**72.6**	44	75.3	74	66.9
	BlockDialect	75.84	70.96	**44.28**	75.44	73.28	67.17

Table 5: Zero-shot accuracy (%) on Llama1-7B, Llama1-13, LLaMA1-30B, LLaMA1-65B, and LLaMA2-7B across tasks (PIQA, ARC-e, ARC-c, BoolQ, HellaSwag, Winogrande). Note: "(x%)" means a sparsity of "x%"

4.2 Comparison between optimization techniques of MXPLM frameworks and MXPDNN frameworks

In the context of mixed-precision quantization for DNNs (MXPDNNs), Rakka et al. [31] identify categories of optimization techniques for bitwidth allocation, including heuristic, gradient-, Reinforcement Learning (RL)-based, and meta-heuristic (like evolutionary algorithms) methods.

Our survey of MXPLM frameworks reveals a clear convergence toward heuristic-based methods, where local weight or activation sensitivity information guides the precision assignment. The reliance on Hessian information in MXPLMs is driven by scalability and practicality: RL- and meta-heuristic approaches, though powerful in smaller-scale DNN settings, introduce prohibitive search overheads when applied to billion-parameter language models, where each evaluation cycle would require costly fine-tuning or extensive forward passes. Moreover, RL-based allocation strategies often depend on reward signals derived from task-level accuracy, which are challenging to compute in MXPLMs due to the vast training cost and the weak correlation between perplexity or zero-shot metrics and individual layer-level bitwidth choices.

The divergence of MXPLM frameworks optimization techniques compared to those of DNNs underscores a critical distinction: while the mixed-precision quantization of DNNs has served as a testbed for a variety of search and optimization paradigms, current MXPLM frameworks demand methods that balance accuracy with feasibility at scale, effectively narrowing the optimization techniques toward sensitivity-driven approaches. Looking ahead, it is conceivable that RL- or meta-heuristic-based optimization could be revisited for MXPLMs if future research develops efficient surrogate objectives for perplexity or zero-shot accuracy, lightweight fine-tuning proxies to reduce evaluation cost, and hardware-in-the-loop frameworks capable of rapidly estimating energy-latency trade-offs; such advancements would enable more global search methods to become computationally tractable even at the scale of billion-parameter language models.

5 Quantization-compatible hardware

Current-generation hardware for LMs includes advanced GPUs, specialized AI accelerators, and large-scale cloud systems. These platforms are designed for high-throughput execution of matrix-intensive workloads using low-precision arithmetic, balancing compute performance (in TFLOPS/TOPS) with memory bandwidth and interconnect efficiency. This section reviews the state-of-the-art in GPU architectures, such as NVIDIA's Hopper [107, 108] and Blackwell [109], and AMD's CDNA3 [110, 111], emerging custom accelerators, such as Tensor Processing Units (TPUs) [112], Intelligence Processing Units (IPUs) [113], and other novel AI chips [114, 115]. It also covers large-scale deployments in cloud infrastructure [116, 117], with a focus on architectural design, supported precision formats, peak performance, and mixed-precision capabilities. Figure 6 compares peak compute throughput (TOPS) versus power consumption across three hardware categories: GPUs, custom AI accelerators, and large-scale systems. Some devices include multiple data points corresponding to different supported numerical precisions (e.g., INT8, FP8, BF16), illustrating how performance scales with precision and highlighting the architectural trade-offs in energy efficiency and compute density.

5.1 Precision casting

Before discussing the precision support of various GPUs and accelerators, the concept of precision casting should be introduced, particularly in the context of mixed-precision computation. Precision casting refers to the conversion of data between different numerical formats (e.g., from FP32 to FP16 or FP8) [118]. During casting, data is reinterpreted and rescaled, which may involve rounding or applying scaling factors to minimize quantization error. Down-casting reduces the bit width through techniques, such as exponent rebiasing, mantissa truncation, and rounding, often accompanied by additional scaling to preserve dynamic range. In contrast, up-casting restores precision by expanding the exponent and mantissa and reapplying the original scale. State-of-the-art hardware, such as NVIDIA's H100, supports automatic precision casting: often low-precision formats like FP8 are used for compute operations, while higher-precision formats (e.g., FP16 or FP32) are used for accumulation and weight updates [119, 120].

5.2 Graphics processing units (GPUs)

Modern GPUs play a central role in LMs training and inference, offering high parallelism, memory bandwidth, and specialized matrix computation units. NVIDIA's data center GPUs, such as the A100 (Ampere), H100 (Hopper) and B200 (Blackwell), are built for deep learning, utilizing Tensor Cores that execute matrix multiply-accumulate operations. The A100 [121] introduced TF32 precision format (a 19-bit floating-point representation with FP32-range exponents and 10-bit mantissa, optimized for tensor operations) and enhanced support for FP16 and BF16, while the H100 [122] added FP8 via its Transformer Engine [123, 124]. Hopper's Streaming Multiprocessors combine traditional
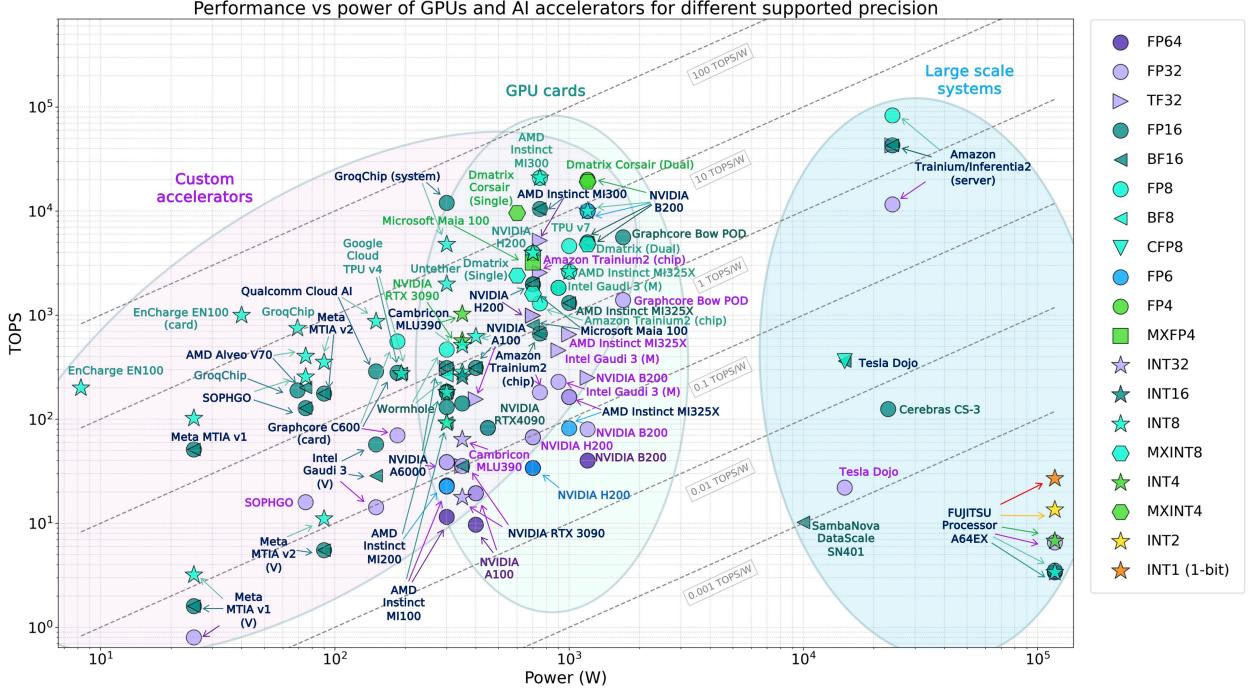
Figure 6: Peak compute performance (in TOPS) versus power consumption for a range of AI hardware platforms, grouped into GPUs, custom AI accelerators, and large-scale systems. Multiple points per device indicate precision-dependent performance based on supported data formats.

compute units (FP64, FP32, INT32) with Tensor Cores that perform FP16/BF16/TF32 matrix operations (with FP32 accumulation) and FP8 operations (with higher-precision accumulation) [124, 125]. Two FP8 formats, E5M2 and E4M3, offer flexibility in dynamic range versus precision trade-offs [124]. The Transformer Engine enables layer-wise mixed-precision and automatic precision casting, performing matrix multiplications in FP8 with accumulation in FP16 or FP32. The newer H200 [107, 108] and B200 [126, 127] extend Hopper's capabilities. The H200 significantly improves upon the H100, offering enhanced performance for memory-intensive and long-context LM inference. The B200, based on the Blackwell architecture, integrates two reticle-limited dies via chip-to-chip interconnect into a unified GPU [109], and adds support for FP4 for ultra-low-precision inference [128]. With 192 GB of HBM3e memory and 8 TB/s bandwidth, it is well-suited for large model inference and memory-bound workloads. Blackwell GPU is also an architecture used in state-of-the-art NVIDIA's Edge AI solutions, e.g. Jetson Thor which also contains transformer engine and low-precision formats support and multi-instance GPU support for running multiple models simultaneously [129].

High-end desktop GPUs like the RTX 3090 [130, 131], RTX 4090 [132, 133], and RTX A6000 [134] also support LMs. The RTX 3090 (Ampere) provides FP32 and FP16 support via third-gen Tensor Cores and supports automatic mixed-precision (AMP) through NVIDIA's software stack (e.g., PyTorch AMP) [135]. The RTX 4090 (Ada Lovelace) introduces fourth-gen Tensor Cores with support for FP8, BF16, FP16, and TF32. The A6000, also Ampere-based, offers 48 GB of GDDR6 and full Tensor Core support. While not designed for large-scale training of LMs, these GPUs are effective for fine-tuning, medium-scale training, and inference.

Overall, NVIDIA GPUs support a broad precision spectrum, from FP64 to INT4, with peak throughput at lower precisions via Tensor Cores (Fig. 6). In addition, the most recent solutions, like Blackwell architecture, support a novel custom-based 4-bit floating point format NVFP4 for inference. NVFP4 uses a 4-bit value (E2M1 format: 1 sign, 2 exponent, 1 mantissa) combined with a per-16-values FP8 scale factor (E4M3) and an additional per-tensor FP32 scale [136]. Note that sparsity support is not shown in Fig. 6, but many GPUs. For instance, the A100 GPU can deliver 156 TFLOPS in TF32 precision, and this throughput is effectively doubled to 312 TFLOPS when structured sparsity is utilized.

AMD's Instinct accelerators, based on the CDNA architecture, are another major platform for LMs [137]. The MI200 series (CDNA2) introduced multi-chip modules and Matrix Cores for efficient computation [110, 111]. The MI250X integrates two GPU dies, each with 110 compute units (CUs) and HBM2e memory, connected via Infinity Fabric.

It supports FP64, FP16, and BF16 matrix operations, with Matrix Cores operating similarly to NVIDIA's Tensor Cores. The latest MI300X (CDNA3) comprises 8 GPU chiplets (XCDs) and four memory/cache dies on a 2.5D interposer [138, 139], providing 192 GB of HBM3 suitable for models requiring hundreds of gigabytes of parameters. Its CUs include Matrix Cores supporting FP64, FP32, and low-precision Fused Multiply-Add (FMA) operations [140]. CDNA3 also introduces support for FP8 and INT8. Like NVIDIA, AMD supports mixed-precision, using low-precision arithmetic (FP8/FP16/BF16) [141] combined with FP32 accumulation [142], and supports automated precision casting through libraries such as JAX [143].

5.3 Emerging custom AI accelerators

Domain-specific AI accelerators for ML and AI workloads include Google's Tensor Processing Units (TPUs) [112], Graphcore's Intelligence Processing Units (IPUs) [113], Groq processor [144], Untether's AI accelerator [114, 145], Etched "Sohu" Transformer ASIC [115], and others. The architectural design of domain-specific AI accelerators can be broadly categorized based on their approach to computation and memory into four classes: traditional tensor-core matrix engines, massively parallel many-core processors, near- or in-memory computing architectures, and wafer-scale or ultra-scalable platforms optimized for extreme workloads.

Matrix-compute accelerators such as Google's TPU [112, 146], Intel's Gaudi2 and Gaudi3 [147, 148], AMD's Alveo V70 [149], Microsoft's MAIA 100, Meta's MTIA (v1 and v2) [150, 151], Cambricon [152], SOPHGO SC7 [153, 154], and Qualcomm's Cloud AI 100 [155] rely on dense matrix engines, such as Tensor Cores or matrix-multiply units (MXUs) to execute high-throughput linear algebra operations. These accelerators can be further divided into those supporting training, e.g., Google's TPU, Intel's Gaudi, Microsoft's MAIA 100, Meta's MTIA v2, Cambricon, and SOPHGO's SC7, and inference-optimized accelerators, e.g., AMD's Alveo V70, Meta's MTIA v1, and Qualcomm's Cloud AI 100.

Training-capable accelerators are optimized for forward and backward passes, and contain large memory, and high-precision accumulation, compared to inference-only accelerators. Google's TPUs are ASICs built for ML acceleration. TPU v4 incorporates multiple Tensor Cores (MXUs) designed for 128x128 systolic matrix operations supporting BF16 and INT8 precision, and also includes sparse execution cores [146]. The 6th-generation TPU improves performance with dual MXUs, a vector unit, and a scalar unit per core [112]. The most recent 7th-generation TPU Ironwood, containing larger HBM, higher bandwidth and improved chip-to-chip communication, is mostly focusing on efficient inference achieving 4614 TFLOPS per chip for FP8 operation [156]. Intel Habana Gaudi (now Gaudi2) is deployed in AI datacenters (e.g., AWS DL1). It includes Matrix Math Engines (MMEs) for FP16/BF16/INT8 matrix operations [147] and 24 integrated tensor-centric networking links (RoCE over Ethernet) for inter-node scaling; while Gaudi3 adds also FP8 support [148]. Gaudi accelerates AI computations in a GPU-like manner, offering high internal bandwidth and supporting BF16/FP16 multiplications with FP32 accumulation. Microsoft Azure MAIA 100, a 5nm chip co-designed with OpenAI, contains high-density matrix engines and supports a wide range of precision types including the Microscaling (MX) formats [157]. It achieves peak performance of 0.8 PFLOPS in BF16, 1.5 PFLOPS in a 9-bit MX hybrid format, and 3.0 PFLOPS in 6-bit MX [150, 151]. MX Format operates as a block floating point (BFP), where a group of numbers shares a common exponent, allowing for more compact data representation and improved processing efficiency. MAIA performs most training in BF16, with 6-/9-bit modes for efficient forward passes or weight updates. Meta's MTIA (Meta Training and Inference Accelerator) supports recommender models and LM inference; while MTIA v2 (5nm) supports both training and inference [158], combining dense matrix engines with general-purpose cores [159]. Cambricon's AI accelerators [152] also support a wide range of formats, including FP32, FP16, BF16, INT16, INT8, INT4. The SOPHGO SC7 HP75-I card supports compute with FP32 down to INT8 [153, 154] and includes a 24-core ARM cluster and a neural accelerator.

Inference-only accelerators generally achieve higher throughput per watt compared to training-capable accelerators. AMD's Alveo V70, built on the Versal AI Edge platform, is designed for power-efficient inference in edge and cloud environments [149]. It supports FP32, FP16, BF16, INT8, and INT4, combining programmable logic and AI engines for mixed-precision inference, using lower precisions for compute and higher precision for accumulation. The other inference-focused accelerator is Meta's first-generation MTIA v1 (7nm chip) [160]. Qualcomm's Cloud AI 100 is also an inference-centric accelerator prioritizing energy efficiency and on-chip memory [155]. It includes 16 AI cores and 144 MB of SRAM; therefore, large models need to be partitioned across chips for large-scale inference.

Many-core and distributed compute architectures, including Graphcore's IPU [113, 161], Untether AI's SpeedAI [114, 145], and Tenstorrent's Wormhole, utilize thousands of lightweight cores with distributed on-chip memory, offering fine-grained parallelism and flexible execution. Graphcore's IPU is a massively parallel processor with distributed on-chip memory [113, 161]. Each chip contains 1,472 IPU-Cores and 900 MB of SRAM, partitioned locally. It supports operation-wise mixed-precision, primarily in FP16 [162, 163]. Multi-chip systems (IPU-PODs) scale IPUs via high-speed interconnects, with model sharding required due to the memory limits. Untether AI's SpeedAI employs a

near-memory compute architecture, with more than 1,400 RISC-V cores adjacent to local SRAM banks [114, 145]. Each memory bank contains 512 Processing Elements (PEs) supporting INT4/INT8/FP8/BF16, with structured sparsity and mixed-precision arithmetic (e.g., FP8 compute and BF16 accumulation). In turn, Tenstorrent's Wormhole processors combine RISC-V CPUs with specialized AI accelerators [164]. Each chip contains 72 Tensix cores, with scalar units supporting various data types from FP32 to INT8. Mixed-precision is achieved by using low-precision for matrix operations and higher precision on scalar cores. The scalable "Galaxy" mesh interconnect enables efficient multi-chip deployment.

In-memory and near-memory compute accelerators, such as d-Matrix's Corsair [165, 166], Etched's Sohu [115], EnCharge AI's EN100 [167], Hailo-8 [168, 169], and again Untether AI's SpeedAI [114, 145], reduce data movement by placing compute units directly within or adjacent to memory banks, which is especially advantageous for memory-bound inference tasks. d-Matrix's Corsair targets low-latency LM inference using a digital in-memory compute (DIMC) system [165, 166]. Each Corsair's card contains 8 chiplets with 512 MB SRAM and utilizes a custom BFP12 (Block Floating Point 12-bit) format, where groups share an exponent. It performs INT8/BFP12 matrix operations with higher-precision accumulation. Etched's Sohu Transformer ASIC implements the transformer pipeline in hardware [115]. Instead of generic MAC arrays, it uses dedicated "transformer cores" to sequentially process tokens through all layers. With 144 GB of HBM3E, performance is bandwidth-bound, and the entire model can reside in memory. Hailo-8 is an efficient edge AI chip using a structure-defined dataflow architecture [168, 169]. It performs inference entirely within on-chip SRAM using INT8/INT4, avoiding costly DRAM access, and supports small-scale LM models. The EN100 from EnCharge AI is an analog in-memory computing–based accelerator that supports 8-bit operations, delivering 200 TOPS per module at only 8.5 W and up to 1,000 TOPS per PCIe card (with four processing units) at 40 W, demonstrating exceptionally low power consumption. Each PCIe card integrates 128 GB of LPDDR memory and provides a memory bandwidth of 272 GB/s [167].

Finally, wafer-scale and ultra-scalable designs, such as Groq's Tensor Streaming Processor (TSP) [170] and Cerebras's CS-2 and CS-3 wafer-scale engines [117, 171] achieve extreme compute and memory scalability by spanning large die areas or statically pipelining entire model graphs across chip-scale dataflow fabrics. Groq's Tensor Streaming Processor (TSP) unrolls neural network graphs into a pipelined loop with static scheduling via compiler [170]. Its single-core design achieves high throughput and low latency but has limited on-chip memory, requiring model distribution across chips [144]. Cerebras's CS-2 and CS-3 wafer-scale engines are built as a single large "wafer-scale" chip. In terms of energy efficiency, it is closer to a large-scale system; therefore, it is discussed below [117, 171].

Across accelerators, mixed-precision arithmetic is implemented using similar principles, e.g., FP16/BF16 for training, INT8 or FP8 for inference, and hybrid schemes (such as block floating point) to balance accuracy and performance. Mixed-precision is often operation-wise, with lower precision for matrix multiplications and higher precision for sensitive operations (e.g., softmax, normalization). Most accelerators rely on software–hardware co-design, with compiler/runtime support crucial for enabling precision casting and operation scheduling. While GPUs and custom AI accelerators dominate LM training, some work explores LM inference on CPUs [172, 173]. For example, Apple's M3 Ultra is marketed as capable of running large language models on device, potentially leveraging its Neural Engine for inference support [174], though practical deployment and performance details remain unclear.

5.4 Large-scale cloud-based systems

Cloud-based AI infrastructure can be built using either GPU-based clusters or custom accelerator clusters. For example, Microsoft Azure uses clusters composed of tens of thousands of NVIDIA A100 GPUs interconnected via high-speed networking for large-scale training and inference workloads [175]. NVIDIA also provides its own reference design for GPU superclusters under the DGX SuperPOD architecture [124]. In addition, recently announced Microsoft Azure ND MI300X v5 Instances are based on AMD Instinct MI300X GPUs clusters [176].

In contrast, several cloud platforms deploy clusters based on custom accelerators. Google's TPUv4 Pods comprise up to 4,096 TPU chips connected in a high-radix torus mesh network with optical circuit switching [177]. Each Pod can deliver up to 1.1 exaFLOPS for BF16 or INT8 operations [146]. The most recent 7th-generation TPU Ironwood achieves 42.5 ExaFLOP per 9,216-chip pod for FP8 operation [156]. Similarly, GroqRack is Groq's cluster-scale architecture that integrates multiple GroqNodes for high-throughput inference and low-latency execution [178, 179].

Amazon Web Services (AWS) uses its own custom chips: Trainium for training and Inferentia for inference [116, 180, 181]. Trainium supports a wide range of formats, including FP32, TF32, FP16, BF16, INT8, and a configurable FP8 (cFP8), where exponent and mantissa bits can be tuned for performance [182]. These accelerators are organized into NeuronCores, with Trainium2 supporting 64-chip UltraClusters interconnected via NeuronLink, a mesh-style fabric for multi-node scaling. The NeuronCore-v2 ScalarEngine enables mixed-precision execution, performing arithmetic in BF16 and handling sensitive operations (e.g., LayerNorm, Softmax, accumulation) in FP32 [183].

SambaNova's SN40L chip uses a Reconfigurable Dataflow Unit (RDU) to map neural networks onto a spatial array of functional units [184]. Each SN40L contains two AI cores, on-package HBM for fast access, and direct DDR connections for capacity [144]. A full system contains 16 SN40L chips. The RDU enables model-specific hardware reconfiguration, avoiding instruction overhead and optimizing compute patterns for GEMMs, convolutions, or attention. This chip supports FP32, BF16, and INT8, with mixed-precision modes and a two-tier memory hierarchy managed by runtime software [185].

Cerebras takes a radically different approach with its Wafer-Scale Engine (WSE). The second-generation WSE-2, used in the CS-2 system, is a full 7nm wafer with 850,000 AI cores and 40 GB of on-wafer SRAM [171]. Its massive size (approximately 46,225 mm^2) eliminates off-chip communication, providing exceptional memory and inter-core bandwidth [186]. The WSE supports FP32, FP16, and BF16, and for larger models, implements weight streaming from external memory. The Cerebras CS-3 adds support for CB16, a custom 16-bit format with BF16-like 8 exponent bits for wide dynamic range and custom mantissa rounding or scaling optimized for AI kernels [117, 187].

Tesla's Dojo supercomputer is built around the custom Dojo D1 chip, designed for internal computer vision and AI workloads [188, 189]. Each D1 die has 354 training nodes arranged in a 2D mesh, all of which use local SRAM without DRAM. Chips are tiled together into larger "ExaPod" systems. Dojo performs training in BF16 or Tesla's CFloat8, a custom 8-bit floating-point format allowing tunable exponent and mantissa allocation [190]. Dojo achieves scalability through fast inter-chip communication and pipeline parallelism across chips rather than large per-chip memory.

Fujitsu's A64FX CPU, used in the Fugaku supercomputer, demonstrates that general-purpose CPUs can still play a role in AI workloads [191, 192]. A64FX features 48 ARM cores, 512-bit Scalable Vector Extension (SVE) vector units, and 32 GB HBM2, supporting FP16/INT8 arithmetic. Although slower than GPUs, CPUs like A64FX offer flexibility, large memory capacity, and the ability to run models with long sequence lengths or high parameter counts without compiler modifications.

6   Prospects and future directions

This section discusses prospects and future directions for fully quantized language models, spanning activation and KV-cache compression, attention/softmax quantization, training-time mixed precision, hardware and software enablement, and emerging low-bit numeric formats.

6.1   Towards fully quantized LMs

The majority of the mixed-precision methods discussed above focus on weight quantization, with a few also applying mixed-precision to activations [60, 97, 100, 102, 103]. Quantizing weights is generally easier, as it can be performed offline with pre-computed scales stored in memory, whereas activation quantization often requires *dynamic* (on-the-fly) scaling to avoid large accuracy drops in LLMs. This dynamic scaling introduces runtime overhead due to extra computations and data, reducing the overall efficiency benefits of quantization.

As both sequence length and model size increase, the memory footprint of the key–value (KV) cache in attention grows proportionally and can become a major bottleneck [193, 194]. In long-context scenarios, this growth can be quadratic in sequence length, making compression essential. Several works in our survey quantize the KV cache, either with uniform [38] or mixed-precision schemes [100, 102, 195], reducing its size while aiming to minimize the resulting accuracy loss. Efficient KV-cache quantization is especially important for deployment in constrained memory settings, where the cache can otherwise dominate total inference memory.

Even when weights, activations, and the KV cache are quantized, many components in transformer inference pipelines remain in higher precision. These include normalization layers, non-linear activations, residual connections, the softmax function, and the query–key–value matrix multiplications in self-attention. Notably, even if the KV cache itself is stored in low precision, it is often dequantized back to higher precision before attention matmuls [73]. Both uniform [70] and mixed-precision [100, 102] approaches have explored fully quantizing the self-attention matmuls.

Recent works have started to address the challenge of low-bit softmax [196–199], which remains one of the key obstacles to achieving truly fully quantized LMs. This is particularly relevant for edge deployments, where the non-linear nature and relatively high computational cost of softmax can become a latency bottleneck. A combination of robust softmax quantization and fully quantized attention could close one of the last major gaps toward end-to-end low-bit inference in LMs.

Reasoning Language Models (RLMs) decompose problems into intermediate steps; explicit long chain-of-thought (CoT) when elicited and implicit reasoning when not, enabling strong performance on math, logic, code, and multi-hop QA [200, 201]. Yet these behaviors are compute-hungry: long CoT traces and large parameter counts inflate memory

and latency. Quantization addresses this by compressing parameters - and sometimes activations and the KV cache - into low-bit formats to cut memory and boost throughput with minimal architectural change. Recent reasoning-centric studies sharpen the picture: broad sweeps over weight/activation/KV settings find that W8A8 and W4A16 are typically near-lossless for CoT-style tasks, while pushing to $\leq$ 3-bit induces disproportionate errors on harder multi-step problems [202, 203]. Complementarily, evidence on smaller models suggests that careful training and post-training compression (including quantization) can deliver competitive reasoning, challenging "scale-only" assumptions [204].

Looking ahead, the community would benefit from a standardized, CoT-sensitive evaluation protocol for quantized RLMs. Rather than ad-hoc leaderboards, future work could anchor comparisons to BF16/FP16 baselines while jointly tracking final-answer accuracy (e.g., GSM8K, MATH/AIME, BBH, MMLU) and process signals (tokens-to-answer, CoT length, self-consistency dispersion, refusal rate, and error taxonomies). Such a protocol would enable principled accuracy–efficiency frontiers across W8A8, W4A16, W4A8, and W4A4, and clarify where performance bends under compression, especially on competition-level math where recent studies suggest degradation accelerates below 4-bit [202, 203]. Future directions also include instance-level agreement analyses (how often quantized models follow the same intermediate steps as full precision), sensitivity to decoding hyperparameters, and robustness under dataset shift, yielding a more causal picture of how quantization perturbs multi-step reasoning.

Automated mixed-precision is a promising direction for better accuracy-efficiency tradeoff through learning per-layer/per-head bit-widths from sensitivity profiles. Fragile computations would retain higher precision while the rest go low-bit. This could generalize current recipes, weight-only W4 with saliency protection (AWQ), full-stack W8A8 stabilized by outlier handling (SmoothQuant) and 4-bit backbones adapted via higher precision LoRA (QLoRA), into data-driven allocation schemes [45, 205, 206]. For long CoT, future systems may adopt adaptive KV-cache precision that varies across layers, spans, or tokens, building on KVQuant baselines and recent advances like PM-KVQ and OTT to compress more aggressively while preserving accuracy [207–209]. Additional directions include dynamic precision during decoding (e.g., temporarily elevating precision for difficult steps), precision-aware training objectives that encourage quantization-robust internal states, and lightweight post-quantization finetunes targeted at the specific failure modes surfaced by the proposed evaluation suite.

6.2 Microscaling format

An important advancement in quantization and low-precision arithmetic is the emergence of the Microscaling (MX) formats [210] that has been standardized by the Open Compute Project (OCP) [211]. Microscaling formats introduce a block floating-point or integer data representation that couples a shared scaling exponent per block with low-bit payloads, such as MXFP8, MXFP6, MXFP4, and MXINT8, to balance efficiency, accuracy, and deployment convenience. An MX block encodes a vector of k values using a single shared *power-of-two* scale X and k quantized elements P_i, with each reconstructed value given by:

$$v_i = X \cdot P_i, \quad i = 1, \ldots, k \tag{40}$$

This fine-grained scaling enables robust quantization down to sub-8-bit regimes while preserving model quality, even when applied with minimal changes to training workflows [210].

These MX formats are a natural fit for mixed strategies in LM inference. For example, [100] introduced a framework for MX-integer mixed precision, while more recent work from MicroMix [212] exploits MX's flexibility by dynamically assigning tensor channels to different precisions, MXFP4, MXFP6, or MXFP8, based on quantization error thresholds. This enables high-speed mixed-precision computation while maintaining output in BF16.

Looking ahead, Microscaling is rapidly emerging as the de-facto standard for low-precision arithmetic in large-scale language models, driven by its strong support from both hardware vendors and deep learning frameworks. From the hardware side, Blackwell [213] architecture natively accelerates MXFP4/8 GEMMs with tensor-core parity, and AMD's CDNA™ 4 [142] introduces dedicated instructions for MX arithmetic. From the software side, Microsoft has released experimental support via its `microxcaling` repository [214], PyTorch has integrated MX into its quantization toolkit through `torchao` [215], and third-party projects such as `torchmx` [216] provide additional research-grade implementations. Taken together, these developments position Microscaling as the likely gold standard for both training and inference of next-generation language models with mature hardware and software support.

6.3 Mixed-precision training for LMs

Mixed-precision training (MPT) is now the default path to scale LLM training on NVIDIA GPUs, delivering large memory and throughput gains with minimal accuracy loss. Tooling such as NVIDIA Apex[2] and native Automatic Mixed Precision (AMP) in PyTorch/TensorFlow automatically casts eligible operations to lower precision, applies

---

[2]A PyTorch Extension for mixed precision and distributed training

dynamic loss scaling, and keeps numerically sensitive paths in higher precision—so models can train end-to-end in FP16 or BF16 with only small code changes [217]. Beyond op casting, modern stacks also vary gradient precision by statistics, while reserving FP32 for critical updates, accelerating both forward and backward passes. On recent hardware, NVIDIA's Transformer Engine extends automatic mixed precision to FP8, selecting FP8 or FP16 per layer and handling re-casting to preserve convergence; emerging formats such as NVFP4 further push the efficiency frontier [218, 219]. Existing tools such as NVIDIA Apex and pytorch/Tensorflow AMP do not have the full mixed precision support that is performed in afromentioned mixed precision frameworks in addition does not support low precision, including FP8/4 or INT8/4. NVIDIA Apex and AMP in PyTorch/TensorFlow do not provide full mixed-precision coverage like the frameworks discussed above, and they lack native support for ultra-low precisions (e.g., FP8/FP4 or INT8/INT4). In this section, we explore the challenges and possible research directions.

6.3.1 Challenges in mixed-precision training

Here, we outline the challenges of mixed precision training of language models.

Numerical stability and dynamic range management: The most significant challenge in MPT is maintaining numerical stability when using lower-precision formats [218, 219]. The limited dynamic range of reduced-precision formats causes gradient values to frequently fall outside the representable range, leading to underflow where small gradients are rounded to zero [219]. This gradient underflow prevents proper weight updates and can cause training divergence, particularly in deeper networks where gradients diminish through multiple layers [135, 218]. The challenge intensifies as precision decreases from FP16 to FP8 and further to FP4, requiring progressively more sophisticated techniques to maintain stability [218–220]. In parallel, activations, weights, and gradients in transformer-based language models exhibit significant outliers that disproportionately affect low-precision training [218, 219]. Addressing these outliers necessitates fine-grained quantization strategies to extend the effective dynamic range of narrow formats [218]. DeepSeek-V3 leverages tile-wise grouping ($1 \times N_c$ elements) or block-wise grouping ($N_c \times N_c$ elements) for FP8 training, while NVFP4 uses 16-element blocks with two-level scaling (per-block E4M3 scales combined with per-tensor FP32 scales) to better capture local dynamic range [218, 219].

Bit-allocation search : Bit-allocation search is computationally expensive because the configuration space grows exponentially with both network depth and the number of candidate precisions per layer. Even "relaxed" differentiable search methods still require large-scale proxy training and bilevel optimization to evaluate allocations at scale [221]. Heuristic strategies, such as greedy layer ranking or gradient-free metaheuristics, can narrow the search, but they still demand many calibrated evaluations or short fine-tuning runs to estimate accuracy and latency. This quickly becomes prohibitive for LLMs or long calibration sets [222, 223].

Hardware constraints make the problem even more complex. Legal tile sizes, accumulation rules, and scale formats (e.g., FP8 tile/block groupings or NVFP4's 16-element two-level scaling) invalidate many nominal precision assignments, requiring hardware-aware feasibility checks that further inflate the search cost [218, 219]. Reliable validation also depends on hardware-accurate kernels and end-to-end timing. For training-time mixed precision, consistency must be maintained across forward, backward, and optimizer paths, often necessitating partial retraining or repeated calibration passes [221, 224].

Sensitivity-guided methods can reduce exploration but introduce their own costs: computing Hessian traces, saliency metrics, or activation/KV sensitivities at LLM scale is expensive, especially when repeated across layers, blocks, and tensor granularities [225]. Finally, bit allocations are not static during training. Configurations that work early can degrade during learning-rate decay or curriculum transitions, prompting reallocation or precision upshifts—effectively embedding the search process within training itself [219].

Layer-wise and component sensitivity: Different architectural components exhibit varying sensitivity to precision reduction [218, 219]. Embedding layers, output projection heads, attention mechanisms (including softmax and attention score computations), normalization layers, and gating modules in MoE architectures consistently require higher precision [218, 219]. For NVFP4, keeping the final 15% of linear layers in BF16 proves necessary for convergence, with sensitivity concentrated in the final blocks of the network [219]. DeepSeek-V3 similarly maintains the first and last few blocks in BF16 while executing the majority of GEMMs in FP8 [218].

Training stability across scale and duration: Maintaining stability becomes increasingly difficult as both model size and training duration grow [219]. NVFP4 training of a 12B-parameter model on 10 trillion tokens, the longest publicly documented FP4 run—showed that relative loss error stays below 1% during stable phases but widens to around 1.5% during learning-rate decay [219]. Techniques that prove essential at scale, such as Random Hadamard transforms, 2D weight scaling, or stochastic rounding, may have little measurable effect on smaller models or shorter training runs, highlighting the scale-dependent nature of precision sensitivity [219].

6.3.2 Future research directions

Below, we outline key directions to address the challenges of mixed-precision training.

Knowledge transfer from pretrained models through leveraging inference quantization insights: Mixed-precision training can be accelerated by transferring precision sensitivity patterns learned from pretrained models to guide precision allocation in new runs [226–228]. Post-training quantization (PTQ) methods such as GPTQ, APTQ, and SliM-LLM estimate layer sensitivity using Hessian-based metrics. They consistently identify attention projections (especially key matrices), the first and last transformer blocks, embeddings, and normalization layers as requiring higher precision across architectures [218, 219, 227, 229]. This prior knowledge can initialize mixed-precision policies by assigning higher precision formats, such as FP8 or BF16, to sensitive components and lower precision formats, such as FP4, to more robust ones. These allocations can then be refined during training through periodic Hessian-based re-evaluations on checkpoints [218, 219, 226, 227, 229].

First-order gradients from pretrained checkpoints provide efficient sensitivity maps that can guide precision allocation using gradient magnitude, $|\nabla_{\mathbf{w}}\mathcal{L}|$, even when gradients are non-zero at convergence [227, 230]. Community sensitivity priors capture common architectural patterns: MoE gating, softmax, and routing operations need higher precision; most feed-forward GEMMs tolerate FP8; and the final 15% of layers are often more fragile [218, 219, 228, 229]. These priors can serve as templates that are refined online through loss or Hessian-based feedback. A practical procedure would be to initialize from small-data sensitivity maps, assign the top 20–30% most sensitive parameters to FP8 or BF16, reassess periodically, and gradually lower precision for robust regions as training stabilizes. The resulting precision patterns can be transferred across model sizes within the same architecture family [219, 226–229].

Selective precision transitions during training: NVFP4 experiments demonstrate that transitioning from FP4 to higher precision during learning rate decay phases can close the loss gap with higher-precision baselines [219]. This suggests a broader research direction: learning optimal precision schedules that vary not just between training phases but potentially between model components, batch positions, or training dynamics [219]. Meta-learning approaches could discover when to upshift precision based on gradient statistics, loss surface sharpness, or convergence indicators, maximizing efficiency while preserving final model quality [218].

Automated mixed-precision search and hyperparameter tuning: Neural architecture search (NAS) techniques could be adapted to discover optimal mixed-precision configurations across the training pipeline, such as zero- and one-shot NAS [231]. Recent work, such as AMQ (Automated Mixed-precision Quantization) [232] demonstrates the potential of using quality predictors to efficiently navigate the mixed-precision search space for weight-only quantization, while FLIQS [233] enables one-shot mixed-precision search across both floating-point and integer formats. This includes searching over precision assignments for forward/backward passes, identifying which operations benefit most from higher precision, and determining layer-specific quantization granularity. The search space could extend to quantization-specific hyperparameters like block sizes, scale factor formats, and rounding strategies, potentially discovering configurations superior to manually designed approaches [218, 219].

Theoretical understanding of low-precision training dynamics: Despite empirical success, theoretical understanding of why low-precision training converges remains limited [219]. Research should investigate how quantization noise affects gradient descent trajectories, loss landscape geometry, and generalization properties [234]. Understanding the conditions under which stochastic rounding provides sufficient gradient estimation, or when chain rule violations from inconsistent quantization remain acceptable, would inform better training methodologies [219]. By clarifying how low-precision updates behave, we can design mixed-precision schemes that allocate bits where they matter most and reduce sensitivity to outliers and dynamic-range limits.

6.4 Hardware and software support

One of the key areas where hardware support is lagging behind the rapid development of LM models is in low-precision arithmetic. Only a few systems currently support INT4 or FP4 formats, and FP4 remains non-standardized. In contrast, NVIDIA has standardized the FP8 format with E5M2 and E4M3 variants [124], enabling the development of robust software stacks, improved portability, and greater adoption. More recently, NVIDIA's Blackwell architecture added support for the 4-bit floating-point inference format NVFP4 [136]. However, the support for ultra-low-precision formats, such as INT2 and INT1, is missing across state-of-the-art accelerators, except the Fujitsu A64FX processor [191, 192]. Efficient use of these extremely low-precision formats requires not only hardware support but also co-designed software and training algorithms that ensure stable convergence under reduced precision.

Another emerging insight from current accelerator designs is the lack of fine-grained precision control. While many platforms support mixed-precision using lower precision (e.g., FP8 or INT8) for matrix multiplications and higher precision (e.g., FP16 or FP32) for accumulation, irregular or intra-matrix precision variation is not supported. Allowing

operations within a kernel or matrix to use different precisions could unlock further efficiency gains in LM training and inference, but would require significant hardware changes and sophisticated scheduling logic. Some recent academic works [235–237] have proposed hardware accelerators for fine-grained mixed-precision, but these designs are often evaluated in isolation, without integration into broader CPU/GPU-based systems. As a result, significant challenges remain in translating such research-proposed hardware from the academic domain into deployable commercial-grade industrial hardware.

Sparsity support is another critical challenge. Structured pruning is primarily supported by NVIDIA and AMD GPUs and Google's TPU SparseCores. For example, NVIDIA's support for structured sparsity in the A100 is limited to specific patterns [121]. Custom accelerators like Untether AI and d-Matrix support fine-grained sparse computation via pattern-based zero skipping [114, 145]. However, unstructured or dynamic sparsity, which is more representative of real model pruning, presents irregular memory access patterns that most hardware struggles to handle efficiently. Combining sparsity with mixed-precision, e.g. using low-precision formats for dense regions and skipping zero-valued blocks, offers further efficiency potential but requires fine-grained control and coordinated software–hardware co-design to avoid underutilization of compute units.

Another open problem is integrating quantization into the hardware pipeline. Currently, quantization is handled as a preprocessing step during model export or compilation. A more advanced design would enable hardware to dynamically determine quantization scales during execution, especially embedding quantization-aware calibration within the compute pipeline. Incorporating more quantization-aware intelligence in hardware could reduce the gap between INT4/INT8 and full precision inference quality, but commercial accelerators have not yet offered this capability.

From a broader hardware perspective, a consistent limitation is the imbalance between memory bandwidth and compute throughput. While GPUs and AI accelerators now reach multi-petaflop compute performance, memory capacity and bandwidth (typically in the 80–200 GB range) have not kept pace. This limits performance for large-scale LMs with hundreds of billions of parameters [238], as high compute throughput is only effective when the memory subsystem can feed it fast enough. Solutions such as larger caches, activation reuse, and gradient compression are becoming essential to improve memory efficiency. Finally, power and thermal limits are growing concerns. The compute demands of large LMs often require merging multiple AI accelerators into massive clusters, resulting in extremely high power consumption. Achieving higher efficiency for training and inference increasingly depends on improving energy efficiency per operation, which will require innovations in both circuit design and system-level thermal management.

6.4.1 Hardware-aware MXPLMs

For DNNs, hardware-aware methods have been investigated that integrate device constraints such as memory bandwidth, latency, and energy directly into the bit-allocation optimization process [31]. These approaches often rely on hardware-in-the-loop evaluation or hardware proxies to guide bitwidth allocation, enabling quantization decisions that are not only accuracy-preserving but also resource-efficient on the target accelerator.

While effective for DNNs, our survey of MXPLM frameworks reveals that very few works have extended this line of thinking to LMs. Among the surveyed methods, only MASE and LLM-PQ explicitly incorporate hardware-awareness in their formulation. The majority of MXPLM frameworks remain accuracy-centric, optimizing perplexity while assuming hardware costs can be addressed post hoc. This gap stems from the unique challenges of LMs: the scale of billions of parameters renders hardware-in-the-loop evaluation prohibitively expensive, simulation environments for LMs are less standardized than for DNNs, and the latency/energy profile of transformer blocks depends strongly on sequence length, cache management, and memory bandwidth all of which complicate proxy modeling.

Looking forward, enabling hardware-aware MXPLMs will require the development of lightweight and accurate hardware proxies tailored to LM workloads, such as analytic cost models for transformer inference (based on energy, latency, area, etc.), or learned surrogates trained on representative workloads. Such methods would allow the quantization search to jointly optimize perplexity and hardware cost, bridging the current disconnect between accuracy-driven MXPLM design and deployability on real hardware platforms.

6.4.2 Software support of MXPLMs

Modern AI compiler stacks expose first-class quantization flows, though scope and maturity differ by project: in PyTorch, the quantization axis is PyTorch AO ("TorchAO"), which provides end-to-end APIs for post-training and training-time quantization and integrates with PyTorch's compilation/export toolchain [215, 239]. NVIDIA TensorRT offers both calibration-driven INT8 and explicit quantization via Q/DQ graphs and, at the kernel/runtime level, enumerates low-precision types including FP8, INT4 (weight-only), and FP4 on supported hardware; at very low precision, production support is strongest in NVIDIA's stack, where current releases document FP8 and 4-bit variants that trade precision for bandwidth/throughput, with quantized values represented in INT8, FP8 (E4M3), INT4, or FP4 (E2M1)

and explicit rules for de/quantization and range handling [240, 241]. Apache TVM compiles both pre-quantized graphs exported from upstream frameworks and TVM-side quantized graphs via its QNN dialect, whose Relay operators include `quantize`, `dequantize`, and `requantize`; its design keeps integer domains first-class in QNN and frontends, enabling import of pre-quantized models while providing TVM-side passes and kernels to execute these graphs portably across targets [242, 243]. Within the Modular ecosystem, MAX (backed by Mojo kernels) supports loading and executing pre-quantized models and provides a quantization module that specifies the admissible encodings and tunable configuration parameters for deployment [244–246]. As a summary, TorchAO serves as the framework-level quantizer, TVM as a portable compiler for quantized IR, TensorRT as a hardware-tuned deployer with aggressive low-precision coverage, and MAX/Mojo as a runtime and kernel toolchain that consumes quantized graphs.

Mixed-precision execution is handled explicitly in deployment compilers. TensorRT supports mixed-precision inference, combining FP32, TF32, BF16, and FP16 within the same network. Users can maintain numerically sensitive operations in higher precision while using reduced precision for less sensitive operations. These mechanisms coexist with explicit quantization graphs and extend to FP8/INT4/FP4 where kernels exist and constraints are met [241, 247, 248]. TVM provides automated mixed-precision transformations that lower FP32 graphs to FP16 (optionally retaining FP32 accumulation), and can be composed with quantization workflows or applied when importing mixed-precision graphs from external frameworks [249]. PyTorch formalizes mixed floating point execution via Automatic Mixed Precision (AMP), which selects dtypes of op level (e.g., BF16 / FP16 for tensor core friendly ops and FP32 for critical reductions) to improve throughput and memory efficiency while maintaining model quality; this is the standard mechanism for heterogeneous precision during training and inference in the PyTorch stack and is routinely paired with export/compile backends for deployment [250].

7   Conclusions

Mixed-precision quantization has emerged as an effective strategy for balancing efficiency and accuracy in language models. Our survey shows that most mixed-precision frameworks with a 4-bit average precision achieve near floating-point accuracy/perplexity, while some 2/3-bit frameworks are competitive. Compared to mixed-precision quantization techniques for DNNs, MXPLM frameworks are distinguished by their scale, memory-bound inference characteristics, and reliance on sensitivity-driven heuristics rather than search-based optimization.

Beyond efficiency gains, MXPLM frameworks hold transformative potential for managing the computational intensity of large-scale LMs. By adaptively distributing precision budgets across layers or tokens, they can significantly mitigate the quadratic scaling of attention and reduce the massive memory footprint of KV-cache operations, the dominant cost drivers in inference. As LMs continue to expand into trillion-parameter regimes, such adaptive precision strategies could become essential not just for inference compression but for enabling training and fine-tuning within feasible energy and latency bounds.

Looking forward, several advancements are required to fully realize the potential of mixed-precision quantization for LMs. First, hardware and software co-design must evolve to natively support fine-grained mixed-precisions, enabling efficient execution without costly dequantization overhead. Second, activation and KV-cache quantization remain open bottlenecks, demanding more robust methods that preserve accuracy. Third, scalable optimization techniques that move beyond local Hessian heuristics, such as lightweight reinforcement learning or gradient methods, are needed to enable global precision allocation at billion-parameter scale. Finally, integration with multi-level memory hierarchies and dynamic precision scheduling offers a promising path toward sustainable deployment across cloud and edge platforms.

References

[1] Chuan Li. Openai's gpt-3 language model: A technical overview. `https://lambda.ai/blog/demystifying-gpt-3`, June 2020. Accessed: 2025-06-02.

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

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 conference of the North American chapter of the association for computational linguistics: human language technologies, volume 1 (long and short papers)*, pages 4171–4186, 2019.

[4] Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever, et al. Improving language understanding by generative pre-training. 2018.

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

[6] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, Ilya Sutskever, et al. Language models are unsupervised multitask learners. *OpenAI blog*, 1(8):9, 2019.

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

[8] Aakanksha Chowdhery, Sharan Narang, Jacob Devlin, Maarten Bosma, Gaurav Mishra, Adam Roberts, Paul Barham, Hyung Won Chung, Charles Sutton, Sebastian Gehrmann, et al. Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240):1–113, 2023.

[9] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

[10] Shaden Smith, Mostofa Patwary, Brandon Norick, Patrick LeGresley, Samyam Rajbhandari, Jared Casper, Zhun Liu, Shrimai Prabhumoye, George Zerveas, Vijay Korthikanti, et al. Using deepspeed and megatron to train megatron-turing nlg 530b, a large-scale generative language model. *arXiv preprint arXiv:2201.11990*, 2022.

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

[12] Zijing Liang, Yanjie Xu, Yifan Hong, Penghui Shang, Qi Wang, Qiang Fu, and Ke Liu. A survey of multimodel large language models. In *Proceedings of the 3rd International Conference on Computer, Artificial Intelligence and Control Engineering*, pages 405–409, 2024.

[13] OpenAI. Gpt-4, 2023. Accessed: 2025-05-24.

[14] Anthropic. Introducing the next generation of claude, 2024. Accessed: 2025-05-24.

[15] Sundar Pichai. Introducing gemini: our largest and most capable ai model, 2023. Accessed: 2025-05-24.

[16] OpenAI. Gpt-5, 2025. Accessed: 2025-08-07.

[17] Google. Introducing gemini 2.0: our new ai model for the agentic era, 2024. Accessed: 2025-05-24.

[18] Anthropic. Introducing claude 4, 2025. Accessed: 2025-08-07.

[19] Jingfeng Yang, Hongye Jin, Ruixiang Tang, Xiaotian Han, Qizhang Feng, Haoming Jiang, Shaochen Zhong, Bing Yin, and Xia Hu. Harnessing the power of llms in practice: A survey on chatgpt and beyond. *ACM Transactions on Knowledge Discovery from Data*, 18(6):1–32, 2024.

[20] Wei-Lin Chiang, Lianmin Zheng, Ying Sheng, Anastasios Nikolas Angelopoulos, Tianle Li, Dacheng Li, Hao Zhang, Banghua Zhu, Michael Jordan, Joseph E. Gonzalez, and Ion Stoica. Chatbot arena: An open platform for evaluating llms by human preference, 2024.

[21] Inc. Qualcomm Technologies. Unlocking on-device generative ai with an npu and heterogeneous computing, 2024. Accessed: 2025-05-25.

[22] Gemma Team, Aishwarya Kamath, Johan Ferret, Shreya Pathak, Nino Vieillard, Ramona Merhej, Sarah Perrin, Tatiana Matejovicova, Alexandre Ramé, Morgane Rivière, et al. Gemma 3 technical report. *arXiv preprint arXiv:2503.19786*, 2025.

[23] Edouard Yvinec and Phil Culliton. Gemma 3 qat models: Bringing state-of-the-art ai to consumer gpus, 2025. Accessed: 2025-05-25.

[24] NVIDIA. Post-training quantization of llms with nvidia nemo and nvidia tensorrt model optimizer, May 2025. Accessed: 2025-05-31.

[25] Ruihao Gong, Yifu Ding, Zining Wang, Chengtao Lv, Xingyu Zheng, Jinyang Du, Haotong Qin, Jinyang Guo, Michele Magno, and Xianglong Liu. A survey of low-bit large language models: Basics, systems, and algorithms. *arXiv preprint arXiv:2409.16694*, 2024.

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

[27] Zhongwei Wan, Xin Wang, Che Liu, Samiul Alam, Yu Zheng, Jiachen Liu, Zhongnan Qu, Shen Yan, Yi Zhu, Quanlu Zhang, et al. Efficient large language models: A survey. *arXiv preprint arXiv:2312.03863*, 2023.

[28] Xunyu Zhu, Jian Li, Yong Liu, Can Ma, and Weiping Wang. A survey on model compression for large language models. *Transactions of the Association for Computational Linguistics*, 12:1556–1577, 2024.

[29] Shiyao Li, Xuefei Ning, Luning Wang, Tengxuan Liu, Xiangsheng Shi, Shengen Yan, Guohao Dai, Huazhong Yang, and Yu Wang. Evaluating quantized large language models. *arXiv preprint arXiv:2402.18158*, 2024.

[30] Jiedong Lang, Zhehao Guo, and Shuyu Huang. A comprehensive study on quantization techniques for large language models. In *2024 4th International Conference on Artificial Intelligence, Robotics, and Communication (ICAIRC)*, pages 224–231. IEEE, 2024.

[31] Mariam Rakka, Mohammed E Fouda, Pramod Khargonekar, and Fadi Kurdahi. A review of state-of-the-art mixed-precision neural network frameworks. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(12):7793–7812, 2024.

[32] Yehui Tang, Yunhe Wang, Jianyuan Guo, Zhijun Tu, Kai Han, Hailin Hu, and Dacheng Tao. A survey on transformer compression. *arXiv preprint arXiv:2402.05964*, 2024.

[33] Dayou Du, Gu Gong, and Xiaowen Chu. Model quantization and hardware acceleration for vision transformers: A comprehensive survey. *arXiv preprint arXiv:2405.00314*, 2024.

[34] Jiawei Yang, Zhongbo Li, Zeqin Feng, and Yongqiang Xie. A survey on neural network quantization. In *Proceedings of the 2025 6th International Conference on Computer Information and Big Data Applications*, pages 384–394, 2025.

[35] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

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

[37] Yeonhong Park, Jake Hyun, SangLyul Cho, Bonggeun Sim, and Jae W Lee. Any-precision llm: Low-cost deployment of multiple, different-sized llms. *arXiv preprint arXiv:2402.10517*, 2024.

[38] Chao Zeng, Songwei Liu, Yusheng Xie, Hong Liu, Xiaojian Wang, Miao Wei, Shu Yang, Fangmin Chen, and Xing Mei. Abq-llm: Arbitrary-bit quantized inference acceleration for large language models. *arXiv preprint arXiv:2408.08554*, 2024.

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

[40] Li Zhang, Olga Krestinskaya, Mohammed E Fouda, Ahmed M Eltawil, and Khaled Nabil Salama. Quantized convolutional neural networks: a hardware perspective. *Frontiers in Electronics*, 6:1469802, 2025.

[41] Zechun Liu, Barlas Oguz, Changsheng Zhao, Ernie Chang, Pierre Stock, Yashar Mehdad, Yangyang Shi, Raghuraman Krishnamoorthi, and Vikas Chandra. Llm-qat: Data-free quantization aware training for large language models. *arXiv preprint arXiv:2305.17888*, 2023.

[42] Mengzhao Chen, Wenqi Shao, Peng Xu, Jiahao Wang, Peng Gao, Kaipeng Zhang, and Ping Luo. Efficientqat: Efficient quantization-aware training for large language models. *arXiv preprint arXiv:2407.11062*, 2024.

[43] Wanyun Cui and Qianle Wang. Cherry on top: Parameter heterogeneity and quantization in large language models. *arXiv preprint arXiv:2404.02837*, 2024.

[44] Tao Yu, Gaurav Gupta, Karthick Gopalswamy, Amith Mamidala, Hao Zhou, Jeffrey Huynh, Youngsuk Park, Ron Diamant, Anoop Deoras, and Luke Huan. Collage: light-weight low-precision strategy for llm training. *arXiv preprint arXiv:2405.03637*, 2024.

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

[46] Fali Wang, Zhiwei Zhang, Xianren Zhang, Zongyu Wu, Tzuhao Mo, Qiuhao Lu, Wanjing Wang, Rui Li, Junjie Xu, Xianfeng Tang, et al. A comprehensive survey of small language models in the era of large language models: Techniques, enhancements, applications, collaboration with llms, and trustworthiness. *arXiv preprint arXiv:2411.03350*, 2024.

[47] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *CoRR*, abs/2106.08295, 2021.

[48] Yi Zhang, Fei Yang, Shuang Peng, Fangyu Wang, and Aimin Pan. Flattenquant: Breaking through the inference compute-bound for large language models with per-tensor quantization. *arXiv preprint arXiv:2402.17985*, 2024.

[49] Jeonghoon Kim, Jung Hyun Lee, Sungdong Kim, Joonsuk Park, Kang Min Yoo, Se Jung Kwon, and Dongsoo Lee. Memory-efficient fine-tuning of compressed large language models via sub-4-bit integer quantization. *Advances in Neural Information Processing Systems*, 36:36187–36207, 2023.

[50] Tim Dettmers, Ruslan Svirschevski, Vage Egiazarian, Denis Kuznedelev, Elias Frantar, Saleh Ashkboos, Alexander Borzunov, Torsten Hoefler, and Dan Alistarh. Spqr: A sparse-quantized representation for near-lossless llm weight compression. *arXiv preprint arXiv:2306.03078*, 2023.

[51] Tim Dettmers, Mike Lewis, Younes Belkada, and Luke Zettlemoyer. Llm.int8(): 8-bit matrix multiplication for transformers at scale. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[52] Zhewei Yao, Reza Yazdani Aminabadi, Minjia Zhang, Xiaoxia Wu, Conglong Li, and Yuxiong He. Zeroquant: Efficient and affordable post-training quantization for large-scale transformers. *Advances in Neural Information Processing Systems*, 35:27168–27183, 2022.

[53] Zhihang Yuan, Lin Niu, Jiawei Liu, Wenyu Liu, Xinggang Wang, Yuzhang Shang, Guangyu Sun, Qiang Wu, Jiaxiang Wu, and Bingzhe Wu. Rptq: Reorder-based post-training quantization for large language models. *arXiv preprint arXiv:2304.01089*, 2023.

[54] Jinguang Wang, Jingyu Wang, Haifeng Sun, Tingting Yang, Zirui Zhuang, Wanyi Ning, Yuexi Yin, Qi Qi, and Jianxin Liao. Mergequant: Accurate 4-bit static quantization of large language models by channel-wise calibration. *arXiv preprint arXiv:2503.07654*, 2025.

[55] Jung Hwan Heo, Jeonghoon Kim, Beomseok Kwon, Byeongwook Kim, Se Jung Kwon, and Dongsoo Lee. Rethinking channel dimensions to isolate outliers for low-bit weight quantization of large language models. *arXiv preprint arXiv:2309.15531*, 2023.

[56] Shiyao Li, Xuefei Ning, Ke Hong, Tengxuan Liu, Luning Wang, Xiuhong Li, Kai Zhong, Guohao Dai, Huazhong Yang, and Yu Wang. Llm-mq: Mixed-precision quantization for efficient llm deployment. In *NeurIPS 2023 Efficient Natural Language and Speech Processing Workshop*, pages 1–5, 2023.

[57] Zhen Zheng, Xiaonan Song, and Chuanjie Liu. Mixllm: Llm quantization with global mixed-precision between output-features and highly-efficient system design. *arXiv preprint arXiv:2412.14590*, 2024.

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

[59] Jung Hyun Lee, Jeonghoon Kim, Se Jung Kwon, and Dongsoo Lee. Flexround: Learnable rounding based on element-wise division for post-training quantization. In *International Conference on Machine Learning*, pages 18913–18939. PMLR, 2023.

[60] Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. *Proceedings of Machine Learning and Systems*, 6:196–209, 2024.

[61] Jung Hyun Lee, Jeonghoon Kim, June Yong Yang, Se Jung Kwon, Eunho Yang, Kang Min Yoo, and Dongsoo Lee. Lrq: Optimizing post-training quantization for large language models by learning low-rank weight-scaling matrices, 2025.

[62] Babak Hassibi and David Stork. Second order derivatives for network pruning: Optimal brain surgeon. In S. Hanson, J. Cowan, and C. Giles, editors, *Advances in Neural Information Processing Systems*, volume 5. Morgan-Kaufmann, 1992.

[63] Elias Frantar, Sidak Pal Singh, and Dan Alistarh. Optimal brain compression: a framework for accurate post-training quantization and pruning. In *Proceedings of the 36th International Conference on Neural Information Processing Systems*, NIPS '22, Red Hook, NY, USA, 2022. Curran Associates Inc.

[64] Ali Edalati, Alireza Ghaffari, Mahsa Ghazvini Nejad, Lu Hou, Boxing Chen, Masoud Asgharian, and Vahid Partovi Nia. Oac: Output-adaptive calibration for accurate post-training quantization. *Proceedings of the AAAI Conference on Artificial Intelligence*, 39(16):16453–16461, April 2025.

[65] Xiuying Wei, Yunchen Zhang, Yuhang Li, Xiangguo Zhang, Ruihao Gong, Jinyang Guo, and Xianglong Liu. Outlier suppression+: Accurate quantization of large language models by equivalent and effective shifting and scaling. In *The 2023 Conference on Empirical Methods in Natural Language Processing*, 2023.

[66] Wenqi Shao, Mengzhao Chen, Zhaoyang Zhang, Peng Xu, Lirui Zhao, Zhiqian Li, Kaipeng Zhang, Peng Gao, Yu Qiao, and Ping Luo. Omniquant: Omnidirectionally calibrated quantization for large language models. In *The Twelfth International Conference on Learning Representations*, 2024.

[67] Markus Nagel, Mart van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction, 2019.

[68] Jerry Chee, Yaohui Cai, Volodymyr Kuleshov, and Christopher De Sa. Quip: 2-bit quantization of large language models with guarantees, 2024.

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

[70] Saleh Ashkboos, Amirkeivan Mohtashami, Maximilian L. Croci, Bo Li, Pashmina Cameron, Martin Jaggi, Dan Alistarh, Torsten Hoefler, and James Hensman. Quarot: Outlier-free 4-bit inference in rotated llms. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 100213–100240. Curran Associates, Inc., 2024.

[71] Zechun Liu, Changsheng Zhao, Igor Fedorov, Bilge Soran, Dhruv Choudhary, Raghuraman Krishnamoorthi, Vikas Chandra, Yuandong Tian, and Tijmen Blankevoort. Spinquant: Llm quantization with learned rotations, 2025.

[72] Haokun Lin, Haobo Xu, Yichen Wu, Jingzhi Cui, Yingtao Zhang, Linzhan Mou, Linqi Song, Zhenan Sun, and Ying Wei. Duquant: Distributing outliers via dual transformation makes stronger quantized llms. In A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, editors, *Advances in Neural Information Processing Systems*, volume 37, pages 87766–87800. Curran Associates, Inc., 2024.

[73] Yujun Lin*, Haotian Tang*, Shang Yang*, Zhekai Zhang, Guangxuan Xiao, Chuang Gan, and Song Han. Qserve: W4a8kv4 quantization and system co-design for efficient llm serving. *arXiv preprint arXiv:2405.04532*, 2024.

[74] Sehoon Kim, Coleman Hooper, Amir Gholami, Zhen Dong, Xiuyu Li, Sheng Shen, Michael W Mahoney, and Kurt Keutzer. Squeezellm: Dense-and-sparse quantization. *arXiv preprint arXiv:2306.07629*, 2023.

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

[76] Wei Huang, Haotong Qin, Yangdong Liu, Yawei Li, Xianglong Liu, Luca Benini, Michele Magno, and Xiaojuan Qi. Slim-llm: Salience-driven mixed-precision quantization for large language models. *arXiv preprint arXiv:2405.14917*, 2024.

[77] Hao Mark Chen, Fuwen Tan, Alexandros Kouris, Royson Lee, Hongxiang Fan, and Stylianos I Venieris. Progressive mixed-precision decoding for efficient llm inference. *arXiv preprint arXiv:2410.13461*, 2024.

[78] Yann LeCun, John Denker, and Sara Solla. Optimal brain damage. *Advances in neural information processing systems*, 2, 1989.

[79] Jie Peng, Zhang Cao, Huaizhi Qu, Zhengyu Zhang, Chang Guo, Yanyong Zhang, Zhichao Cao, and Tianlong Chen. Harnessing your dram and ssd for sustainable and accessible llm inference with mixed-precision and multi-level caching. *arXiv preprint arXiv:2410.14740*, 2024.

[80] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Re, et al. Deja vu: Contextual sparsity for efficient llms at inference time. In *International Conference on Machine Learning*, pages 22137–22176. PMLR, 2023.

[81] Bowen Ping, Shuo Wang, Hanqing Wang, Xu Han, Yuzhuang Xu, Yukun Yan, Yun Chen, Baobao Chang, Zhiyuan Liu, and Maosong Sun. Delta-come: Training-free delta-compression with mixed-precision for large language models. *arXiv preprint arXiv:2406.08903*, 2024.

[82] Jinhao Li, Jiaming Xu, Shiyao Li, Shan Huang, Jun Liu, Yaoxiu Lian, and Guohao Dai. Fast and efficient 2-bit llm inference on gpu: 2/4/16-bit in a weight matrix with asynchronous dequantization. *arXiv preprint arXiv:2311.16442*, 2023.

[83] Juntao Zhao, Borui Wan, Yanghua Peng, Haibin Lin, and Chuan Wu. Llm-pq: Serving llm on heterogeneous clusters with phase-aware partition and adaptive quantization. *arXiv preprint arXiv:2403.01136*, 2024.

[84] LLC Gurobi Optimization. Gurobi optimizer reference manual. 2023.

[85] Peng Tang, Jiacheng Liu, Xiaofeng Hou, Yifei Pu, Jing Wang, Pheng-Ann Heng, Chao Li, and Minyi Guo. Hobbit: A mixed precision expert offloading system for fast moe inference. *arXiv preprint arXiv:2411.01433*, 2024.

[86] Noam Shazeer, Azalia Mirhoseini, Krzysztof Maziarz, Andy Davis, Quoc Le, Geoffrey Hinton, and Jeff Dean. Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*, 2017.

[87] Yuzong Chen, Ahmed F AbouElhamayed, Xilai Dai, Yang Wang, Marta Andronic, George A Constantinides, and Mohamed S Abdelfattah. Bitmod: Bit-serial mixture-of-datatype llm acceleration. *arXiv preprint arXiv:2411.11745*, 2024.

[88] Steve Dai, Rangha Venkatesan, Mark Ren, Brian Zimmer, William Dally, and Brucek Khailany. Vs-quant: Per-vector scaled quantization for accurate low-precision neural network inference. *Proceedings of Machine Learning and Systems*, 3:873–884, 2021.

[89] Yuzhang Shang, Zhihang Yuan, Qiang Wu, and Zhen Dong. Pb-llm: Partially binarized large language models. *arXiv preprint arXiv:2310.00034*, 2023.

[90] Elias Frantar and Dan Alistarh. Sparsegpt: Massive language models can be accurately pruned in one-shot. In *International conference on machine learning*, pages 10323–10337. PMLR, 2023.

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

[92] E Frantar, S Ashkboos, T Hoefler, and D Alistarh. Optq: Accurate quantization for generative pre-trained transformers. 2023. In *https://openreview.net/forum?id=tcbBPnfwxS*.

[93] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks. *Journal of Machine Learning Research*, 22(241):1–124, 2021.

[94] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[95] Goran Flegar and Enrique S Quintana-Ortí. Balanced csr sparse matrix-vector product on graphics processors. In *European Conference on Parallel Processing*, pages 697–709. Springer, 2017.

[96] Zihan Chen, Bike Xie, Jundong Li, and Cong Shen. Channel-wise mixed-precision quantization for large language models. *arXiv preprint arXiv:2410.13056*, 2024.

[97] Utkarsh Saxena, Sayeh Sharify, Kaushik Roy, and Xin Wang. Resq: Mixed-precision quantization of large language models with low-rank residuals. *arXiv preprint arXiv:2412.14363*, 2024.

[98] Zihao Ye, Lequn Chen, Ruihang Lai, Yilong Zhao, Size Zheng, Junru Shao, Bohan Hou, Hongyi Jin, Yifei Zuo, Liangsheng Yin, et al. Accelerating self-attentions for llm serving with flashinfer, february 2024. *URL https://flashinfer.ai/2024/02/02/introduce-flashinfer.html*.

[99] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th symposium on operating systems principles*, pages 611–626, 2023.

[100] Jianyi Cheng, Cheng Zhang, Zhewen Yu, Christos-Savvas Bouganis, George A Constantinides, and Yiren Zhao. A dataflow compiler for efficient llm inference using custom microscaling formats. *arXiv preprint arXiv:2307.15517*, 2023.

[101] Yoshihiko Ozaki, Yuki Tanigaki, Shuhei Watanabe, and Masaki Onishi. Multiobjective tree-structured parzen estimator for computationally expensive optimization problems. In *Proceedings of the 2020 genetic and evolutionary computation conference*, pages 533–541, 2020.

[102] Wonsuk Jang and Thierry Tambe. Blockdialect: Block-wise fine-grained mixed format for energy-efficient llm inference. *arXiv e-prints*, pages arXiv–2501, 2025.

[103] Saleh Ashkboos, Ilia Markov, Elias Frantar, Tingxuan Zhong, Xincheng Wang, Jie Ren, Torsten Hoefler, and Dan Alistarh. Quik: Towards end-to-end 4-bit inference on generative large language models. *arXiv preprint arXiv:2310.09259*, 2023.

[104] Sheng Shen, Zhen Dong, Jiayu Ye, Linjian Ma, Zhewei Yao, Amir Gholami, Michael W Mahoney, and Kurt Keutzer. Q-bert: Hessian based ultra low precision quantization of bert. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 8815–8821, 2020.

[105] Ali Hadi Zadeh, Isak Edo, Omar Mohamed Awad, and Andreas Moshovos. Gobo: Quantizing attention-based nlp models for low latency and energy efficient inference. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 811–824. IEEE, 2020.

[106] Yelysei Bondarenko, Riccardo Del Chiaro, and Markus Nagel. Low-rank quantization-aware training for llms. *arXiv preprint arXiv:2406.06385*, 2024.

[107] NVIDIA. Nvidia h200 tensor core gpu, 2024. Accessed: 2025-03-19.

[108] NVIDIA. Nvidia h200 hpc datasheet, 2024. Accessed: 2025-03-19.

[109] NVIDIA. Nvidia blackwell architecture, 2024. Accessed: 2025-03-23.

[110] AMD. Amd instinct mi200 series accelerator, 2024. Accessed: 2025-03-19.

[111] TechPowerUp. Amd instinct mi200 specifications, 2024. Accessed: 2025-03-19.

[112] Google Cloud. Google tpu v6e documentation, 2024. Accessed: 2025-03-19.

[113] Graphcore. Graphcore c600 ipu processor pcie card specifications, 2024. Accessed: 2025-03-19.

[114] Untether AI. Untether ai products, 2024. Accessed: 2025-03-19.

[115] Etched. Etched ai accelerators, 2024. Accessed: 2025-03-19.

[116] Amazon AWS. Trainium 2 and inferentia 2 architecture overview, 2024. Accessed: 2025-03-19.

[117] Futurum Group. Cerebras cs-3: Bringing on the nvidia blackwell competition, 2024. Accessed: 2025-03-19.

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

[119] NVIDIA. Transformer engine. `https://github.com/NVIDIA/TransformerEngine`, 2024. Accessed: 2025-03-23.

[120] Piotr Luszczek, Ahmad Abdelfattah, Hartwig Anzt, Atsushi Suzuki, and Stanimire Tomov. Batched sparse and mixed-precision linear algebra interface for efficient use of gpu hardware accelerators in scientific applications. *Future Generation Computer Systems*, 160:359–374, 2024.

[121] NVIDIA. Nvidia a100 tensor core gpu, 2024. Accessed: 2025-03-19.

[122] NVIDIA. Nvidia h100 tensor core gpu, 2024. Accessed: 2025-03-19.

[123] NVIDIA. Nvidia hopper architecture whitepaper, 2022. Accessed: 2025-03-23.

[124] NVIDIA. Nvidia hopper architecture in-depth, 2022. Accessed: 2025-03-23.

[125] Timothy Prickett Morgan. Deep dive into nvidia's hopper gpu architecture, 2022. Accessed: 2025-03-23.

[126] NVIDIA. Nvidia blackwell b200 tensor core gpu, 2024. Accessed: 2025-03-19.

[127] NVIDIA. Nvidia blackwell b200 datasheet, 2024. Accessed: 2025-03-19.

[128] Ruizhe Wang, Yeyun Gong, Xiao Liu, Guoshuai Zhao, Ziyue Yang, Baining Guo, Zhengjun Zha, and Peng Cheng. Optimizing large language model training using fp4 quantization. *arXiv preprint arXiv:2501.17116*, 2025.

[129] NVIDIA. Jetson thor: Advanced ai for physical robotics, 2025. Accessed: 2025-09-29.

[130] NVIDIA. Nvidia geforce rtx 3090 graphics card, 2024. Accessed: 2025-03-19.

[131] HotHardware. Nvidia geforce rtx 3090 review, 2024. Accessed: 2025-03-19.

[132] NVIDIA. Nvidia geforce rtx 4090 graphics card, 2024. Accessed: 2025-03-19.

[133] TechPowerUp. Nvidia geforce rtx 4090 specifications, 2024. Accessed: 2025-03-19.

[134] NVIDIA. Nvidia professional visualization datasheet, 2024. Accessed: 2025-03-19.

[135] NVIDIA. Train with mixed precision. `https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/`, 2023. Accessed: 2025-10-08.

[136] Eduardo Alvarez, Omri Almog, Eric Chung, Simon Layton, Dusan Stosic, Ronny Krashinsky, and Kyle Aubrey. Introducing nvfp4 for efficient and accurate low-precision inference, 2025. Accessed: 2025-09-29.

[137] AMD. Amd instinct mi100 accelerator, 2024. Accessed: 2025-03-19.

[138] AMD. Amd instinct mi300x platform data sheet, 2024. Accessed: 2025-03-19.

[139] Patrick Kennedy. Amd instinct mi300x architecture at hot chips 2024, 2024. Accessed: 2025-03-23.

[140] AMD. Amd instinct mi300 cdna3 instruction set architecture, 2024. Accessed: 2025-03-23.

[141] Sergio P Perez, Yan Zhang, James Briggs, Charlie Blake, Josh Levy-Kramer, Paul Balanca, Carlo Luschi, Stephen Barlow, and Andrew William Fitzgibbon. Training and inference of large language models using 8-bit floating point. *arXiv preprint arXiv:2309.17224*, 2023.

[142] AMD. Amd cdna 3 architecture white paper, 2024. Accessed: 2025-03-23.

[143] AMD. Accelerating jax with mixed precision on amd gpus, 2024. Accessed: 2025-03-23.

[144] SambaNova Systems. Introducing sn40l: The best inference solution for the generative ai era, 2024. Accessed: 2025-03-23.

[145] Untether AI. Ieee paper on untether ai technology. *IEEE Xplore*, 2024. Accessed: 2025-03-19.

[146] Google Cloud. Cloud tpu v4, 2023. Accessed: 2025-03-23.

[147] Intel Habana Labs. Habana gaudi ai accelerator overview, 2024. Accessed: 2025-03-19.

[148] Intel. Intel gaudi 3 ai accelerator, 2024. Accessed: 2025-03-19.

[149] AMD. Amd alveo v70 ai accelerator, 2024. Accessed: 2025-03-19.

[150] HostZealot. Microsoft announced the first ai chip maia 100, 2023. Accessed: 2025-03-23.

[151] Microsoft Azure. Azure maia for the era of ai: From silicon to software to systems, 2023. Accessed: 2025-03-23.

[152] Cambricon. Cambricon ai accelerators, 2024. Accessed: 2025-03-19.

[153] SOPHGO. Sophgo sc7 ai accelerator, 2024. Accessed: 2025-03-19.

[154] SOPHGO. Sc7 series ai accelerator product introduction, 2024. Accessed: 2025-03-19.

[155] Qualcomm. Qualcomm cloud ai 100, 2024. Accessed: 2025-03-19.

[156] Google Blog. Ironwood: The first google tpu for the age of inference, 2025. Accessed: 2025-09-30.

[157] Microsoft. Inside maia 100: Revolutionizing ai workloads with microsoft's custom ai accelerator, 2024. Accessed: 2025-03-19.

[158] Meta AI. Next-generation meta training and inference accelerator (mtia), 2024. Accessed: 2025-03-19.

[159] TechPowerUp. Meta announces new mtia ai accelerator with improved performance to ease nvidia's grip, 2024. Accessed: 2025-03-23.

[160] Meta AI. Meta training and inference accelerator (mtia), 2024. Accessed: 2025-03-19.

[161] Graphcore. Bow pod16 datasheet, 2024. Accessed: 2025-03-19.

[162] Graphcore. Mixed precision training in pytorch on ipus, 2024. Accessed: 2025-03-23.

[163] Hongwu Peng, Caiwen Ding, Tong Geng, Sutanay Choudhury, Kevin Barker, and Ang Li. Evaluating emerging ai/ml accelerators: Ipu, rdu, and nvidia/amd gpus. In *Companion of the 15th ACM/SPEC International Conference on Performance Engineering*, pages 14–20, 2024.

[164] Tenstorrent. Wormhole ai accelerator, 2024. Accessed: 2025-03-19.

[165] d Matrix. d-matrix technical white paper, 2024. Accessed: 2025-03-19.

[166] d Matrix. d-matrix ai accelerator products, 2024. Accessed: 2025-03-19.

[167] EnCharge AI. Encharge — en100, 2025. Accessed: 2025-09-29.

[168] Hailo AI. Hailo-8 ai accelerator for edge computing, 2024. Accessed: 2025-03-19.

[169] Things Embedded. Benefits of hailo-8 ai accelerators for edge computing, 2024. Accessed: 2025-03-19.

[170] Groq. Groqchip processor product brief, 2024. Accessed: 2025-03-19.

[171] Cerebras. Cerebras cs-2 data sheet, 2024. Accessed: 2025-03-19.

[172] Haihao Shen, Hanwen Chang, Bo Dong, Yu Luo, and Hengyu Meng. Efficient llm inference on cpus. *arXiv preprint arXiv:2311.00502*, 2023.

[173] Seonjin Na, Geonhwa Jeong, Byung Hoon Ahn, Jeffrey Young, Tushar Krishna, and Hyesoon Kim. Understanding performance implications of llm inference on cpus. In *Proceedings of the IEEE International Symposium on Workload Characterization (IISWC)*, 2024.

[174] Apple Inc. Apple reveals m3 ultra, taking apple silicon to a new extreme, 2025. Accessed: 2025-03-23.

[175] NVIDIA. Microsoft azure nd a100 v4 vm series: Making ai supercomputing accessible, 2021. Accessed: 2025-03-23.

[176] Microsoft Learn. Nd-mi300x-v5 size series — azure virtual machines, 2024. Accessed: 2025-09-30.

[177] Norm Jouppi, George Kurian, Sheng Li, Peter Ma, Rahul Nagarajan, Lifeng Nai, Nishant Patil, Suvinay Subramanian, Andy Swing, Brian Towles, et al. Tpu v4: An optically reconfigurable supercomputer for machine learning with hardware support for embeddings. In *Proceedings of the 50th annual international symposium on computer architecture*, pages 1–14, 2023.

[178] Groq. Groqrack compute cluster product brief, 2024. Accessed: 2025-03-19.

[179] Groq. Groq scalability technical document, 2024. Accessed: 2025-03-19.

[180] SemiAnalysis. Amazon's ai self-sufficiency: Trainium2 architecture and networking, 2024. Accessed: 2025-03-19.

[181] Amazon Web Services. Aws inferentia: High performance ml inference chip, 2024. Accessed: 2025-03-23.

[182] Amazon Web Services. Aws neuron: Supported data types, 2024. Accessed: 2025-03-23.

[183] Amazon Web Services. Aws neuron core v2 architecture, 2024. Accessed: 2025-03-23.

[184] SambaNova Systems. Sambanova datascale sn30 datasheet, 2024. Accessed: 2025-03-19.

[185] Raghu Prabhakar, Ram Sivaramakrishnan, Darshan Gandhi, Yun Du, Mingran Wang, Xiangyu Song, Kejie Zhang, Tianren Gao, Angela Wang, Xiaoyan Li, et al. Sambanova sn40l: Scaling the ai memory wall with dataflow and composition of experts. In *2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1353–1366. IEEE, 2024.

[186] Timothy Prickett Morgan. Cerebras wants its piece of an increasingly heterogenous hpc world, 2022. Accessed: 2025-03-23.

[187] Cerebras Systems. Dynamic loss scaling, 2024. Accessed: 2025-03-23.

[188] Tesla. Tesla dojo ai supercomputer overview, 2024. Accessed: 2025-03-19.

[189] IEEE Xplore. Tesla dojo supercomputer. *IEEE Xplore*, 2024. Accessed: 2025-03-19.

[190] Inc. Tesla. Tesla dojo technology presentation, 2023. Accessed: 2025-03-23.

[191] Fujitsu. Fujitsu a64fx processor datasheet, 2024. Accessed: 2025-03-19.

[192] TOP500. Fujitsu a64fx supercomputer information, 2024. Accessed: 2025-03-19.

[193] Haoyi Wu and Kewei Tu. Layer-condensed kv cache for efficient inference of large language models, 2024.

[194] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the 29th Symposium on Operating Systems Principles*, SOSP '23, page 611–626, New York, NY, USA, 2023. Association for Computing Machinery.

[195] June Yong Yang, Byeongwook Kim, Jeongin Bae, Beomseok Kwon, Gunho Park, Eunho Yang, Se Jung Kwon, and Dongsoo Lee. No token left behind: Reliable kv cache compression via importance-aware mixed precision quantization. *arXiv preprint arXiv:2402.18096*, 2024.

[196] Moran Shkolnik, Maxim Fishman, Brian Chmiel, Hilla Ben-Yaacov, Ron Banner, and Kfir Yehuda Levy. Exaq: Exponent aware quantization for llms acceleration, 2024.

[197] Xing Hu, Yuan Cheng, Dawei Yang, Zhihang Yuan, Jiangyong Yu, Chen Xu, and Sifan Zhou. I-llm: Efficient integer-only inference for fully-quantized low-bit large language models, 2024.

[198] Ihor Vasyltsov and Wooseok Chang. Efficient softmax approximation for deep neural networks with attention mechanism, 2021.

[199] Yang Lin, Tianyu Zhang, Peiqin Sun, Zheng Li, and Shuchang Zhou. Fq-vit: Post-training quantization for fully quantized vision transformer, 2023.

[200] Qiguang Chen, Libo Qin, Jinhao Liu, Dengyun Peng, Jiannan Guan, Peng Wang, Mengkang Hu, Yuhang Zhou, Te Gao, and Wanxiang Che. Towards reasoning era: A survey of long chain-of-thought for reasoning large language models. *arXiv preprint arXiv:2503.09567*, 2025.

[201] Jindong Li, Yali Fu, Li Fan, Jiahong Liu, Yao Shu, Chengwei Qin, Menglin Yang, Irwin King, and Rex Ying. Implicit reasoning in large language models: A comprehensive survey. *arXiv preprint arXiv:2509.02350*, 2025.

[202] Zhen Li, Yupeng Su, Runming Yang, Congkai Xie, Zheng Wang, Zhongwei Xie, Ngai Wong, and Hongxia Yang. Quantization meets reasoning: Exploring llm low-bit quantization degradation for mathematical reasoning. *arXiv preprint arXiv:2501.03035*, 2025.

[203] Ruikang Liu, Yuxuan Sun, Manyi Zhang, Haoli Bai, Xianzhi Yu, Tiezheng Yu, Chun Yuan, and Lu Hou. Quantization hurts reasoning? an empirical study on quantized reasoning models. *arXiv preprint arXiv:2504.04823*, 2025.

[204] Gaurav Srivastava, Shuxiang Cao, and Xuan Wang. Towards reasoning ability of small language models. *arXiv preprint arXiv:2502.11569*, 2025.

[205] Ji Lin, Jiaming Tang, Haotian Tang, et al. Awq: Activation-aware weight quantization for llm compression and acceleration. *arXiv preprint arXiv:2306.00978*, 2023.

[206] Guangxuan Xiao et al. Smoothquant: Accurate and efficient post-training quantization for large language models. *arXiv preprint arXiv:2211.10438*, 2022.

[207] Coleman Hooper, Sehoon Kim, Hiva Mohammadzadeh, et al. Kvquant: Towards 10 million context length llm inference with kv cache quantization. In *NeurIPS*, 2024.

[208] Tengxuan Liu, Shiyao Li, Jiayi Yang, Tianchen Zhao, Feng Zhou, Xiaohui Song, Guohao Dai, Shengen Yan, Huazhong Yang, and Yu Wang. Pm-kvq: Progressive mixed-precision kv cache quantization for long-cot llms. *arXiv preprint arXiv:2505.18610*, 2025.

[209] Yi Su, Yuechi Zhou, Quantong Qiu, Juntao Li, Qingrong Xia, Ping Li, Xinyu Duan, Zhefeng Wang, and Min Zhang. Accurate kv cache quantization with outlier tokens tracing. *arXiv preprint arXiv:2505.10938*, 2025. ACL 2025 (Main), submitted.

[210] Bita Darvish Rouhani, Ritchie Zhao, Ankit More, Mathew Hall, Alireza Khodamoradi, Summer Deng, Dhruv Choudhary, Marius Cornea, Eric Dellinger, Kristof Denolf, Stosic Dusan, Venmugil Elango, Maximilian Golub, Alexander Heinecke, Phil James-Roxby, Dharmesh Jani, Gaurav Kolhe, Martin Langhammer, Ada Li, Levi Melnick, Maral Mesmakhosroshahi, Andres Rodriguez, Michael Schulte, Rasoul Shafipour, Lei Shao, Michael Siu, Pradeep Dubey, Paulius Micikevicius, Maxim Naumov, Colin Verrilli, Ralph Wittig, Doug Burger, and Eric Chung. Microscaling data formats for deep learning, 2023.

[211] Bita Darvish Rouhani, Nitin Garegrat, Tom Savell, Ankit More, Kyung-Nam Han, Ritchie Zhao, Mathew Hall, Jasmine Klar, Eric Chung, Yuan Yu, Michael Schulte, Ralph Wittig, Ian Bratt, Nigel Stephens, Jelena Milanovic, John Brothers, Pradeep Dubey, Marius Cornea, Alexander Heinecke, Andres Rodriguez, Martin Langhammer, Summer Deng, Maxim Naumov, Paulius Micikevicius, Michael Siu, and Colin Verrilli. Ocp microscaling formats (mx) specification. *Open Compute Project*, 2023.

[212] Wenyuan Liu, Haoqian Meng, Yilun Luo, Peng Zhang, and Xindian Ma. Micromix: Efficient mixed-precision quantization with microscaling formats for large language models, 2025.

[213] NVIDIA Developer Blog. Openai triton on nvidia blackwell boosts ai performance. 2025.

[214] Microsoft. microxaling. `https://github.com/microsoft/microxcaling`, 2025. Accessed: 2025-08-18.

[215] PyTorch. torchao: Pytorch mx quantization support. `https://github.com/pytorch/ao`, 2025. Accessed: 2025-08-18.

[216] Rain Neuromorphics. torchmx. `https://github.com/rain-neuromorphics/torchmx/tree/main/torchmx`, 2025. Accessed: 2025-08-18.

[217] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, and Hao Wu. Mixed precision training, 2018.

[218] DeepSeek-AI. Deepseek-v3 technical report. Technical report, DeepSeek, November 2024.

[219] Xueying Liu, Mengzhou Xia, Tianyu Chen, Zhihan Wang, Sanjeev Arora, and Beidi Chen. Pretraining large language models with nvfp4. *arXiv preprint arXiv:2509.25149*, 2025.

[220] Joonhyung Lee, Jeongin Bae, Byeongwook Kim, Se Jung Kwon, and Dongsoo Lee. To fp8 and back again: Quantifying the effects of reducing precision on llm training stability. *arXiv preprint arXiv:2405.18710*, 2024.

[221] W. Cai et al. Rethinking differentiable search for mixed-precision neural networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2020.

[222] S. Satoki et al. Greedy search algorithm for mixed precision in post-training quantization. In *Proceedings of Machine Learning Research*, 2021.

[223] N. Shlezinger et al. Gradient-free bit allocation for mixed-precision neural networks. *PMC Article*, 2022.

[224] Y. Zhang et al. Mixed-precision quantization: Make the best use of bits for efficient neural networks. *arXiv preprint*, 2024.

[225] Zhen Dong, Zhewei Yao, Amir Gholami, Michael W. Mahoney, and Kurt Keutzer. HAWQ-V2: Hessian aware trace-weighted quantization of neural networks. *Advances in Neural Information Processing Systems*, 33, 2020.

[226] Hussein Mahmoud. Bringing quantization to the transfer learning world. *ETH Zurich Bachelor Thesis*, 2022.

[227] Yuhang Zhou, Yuxuan Wang, and Liang Zhang. Sensitivity-aware post-training quantization for deep neural networks. *arXiv preprint arXiv:2509.05576*, 2025.

[228] Anshul Shukla, Gullal S. Cheema, Balaraman Ravindran Srinivasan, and Saket Anand. BeST: A novel source selection metric for transfer learning. In *OpenReview*, 2025.

[229] Qingyang Wu, Yueming Liu, Yong Zhang, et al. Hessian-based mixed-precision quantization with transition aware search. *Neural Networks*, 182:107068, 2025.

[230] Zhe Chen, Jiawei Wang, Yuchen Liu, et al. Gradient-aware weight quantization for large language models. *arXiv preprint arXiv:2411.00850*, 2024.

[231] Guihong Li, Duc Hoang, Kartikeya Bhardwaj, Ming Lin, Zhangyang Wang, and Radu Marculescu. Zero-shot neural architecture search: Challenges, solutions, and opportunities. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 46(12):7618–7635, 2024.

[232] Sangjun Lee, Seung-taek Woo, Jungyu Jin, Changhun Lee, and Eunhyeok Park. Amq: Enabling automl for mixed-precision weight-only quantization of large language models. *arXiv preprint arXiv:2509.12019*, 2025.

[233] Jordan Dotzel, Gang Wu, Andrew Li, Muhammad Umar, Yun Ni, Mohamed S Abdelfattah, Zhiru Zhang, Liqun Cheng, Martin G Dixon, Norman P Jouppi, et al. Fliqs: One-shot mixed-precision floating-point and integer quantization search. *arXiv preprint arXiv:2308.03290*, 2023.

[234] Tyler Johnson, Hieu Nguyen, and Wei Chen. Quantifying reduced precision effects on LLM training stability. *arXiv preprint arXiv:2405.18710*, 2024.

[235] Faraz Tahmasebi, Yian Wang, Benji YH Huang, and Hyoukjun Kwon. Flexibit: Fully flexible precision bit-parallel accelerator architecture for arbitrary mixed precision ai. *arXiv preprint arXiv:2411.18065*, 2024.

[236] Hardik Sharma, Jongse Park, Naveen Suda, Liangzhen Lai, Benson Chau, Joon Kyung Kim, Vikas Chandra, and Hadi Esmaeilzadeh. Bit fusion: Bit-level dynamically composable architecture for accelerating deep neural network. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pages 764–775. IEEE, 2018.

[237] Mariam Rakka, Rachid Karami, Ahmed M Eltawil, Mohammed E Fouda, and Fadi Kurdahi. Bf-imna: A bit fluid in-memory neural architecture for neural network acceleration. *arXiv preprint arXiv:2411.01417*, 2024.

[238] Cerebras Systems. Cerebras cs-3 vs nvidia b200: 2024 ai accelerators compared, 2024. Accessed: 2025-03-23.

[239] PyTorch Team. torchao.quantization api reference, 2025. Accessed Oct 4, 2025.

[240] NVIDIA. Working with quantized types — nvidia tensorrt 10.9, 2025. Accessed Oct 4, 2025.

[241] NVIDIA. Tensorrt capabilities, 2025. Accessed Oct 4, 2025.

[242] Apache TVM. Relay qnn operators (quantization) — tvm v0.10, 2022. Accessed Oct 4, 2025.

[243] Apache TVM. Deploy a framework-prequantized model with tvm, 2021. Accessed Oct 4, 2025.

[244] Modular. Max graph: Quantization guide, 2025. Accessed Oct 4, 2025.

[245] Modular. Max graph python api: quantization, 2025. Accessed Oct 4, 2025.

[246] Modular. Mojo kernels: quantization apis, 2025. Accessed Oct 4, 2025.

[247] NVIDIA. Accuracy considerations — mixed precision inference, 2025. Accessed Oct 4, 2025.

[248] NVIDIA. Best practices for tensorrt performance, 2024. Accessed Oct 4, 2025.

[249] Apache TVM. Deploy to adreno™ gpu (mixed precision via `ToMixedPrecision`), 2023. Accessed Oct 4, 2025.

[250] PyTorch Team. Automatic mixed precision package — `torch.amp`, 2025. Accessed Oct 4, 2025.