
MEMORY- AND LATENCY-CONSTRAINED INFERENCE OF LARGE LANGUAGE MODELS VIA ADAPTIVE SPLIT COMPUTING

Mingyu Sung, Vikas Palakonda, Suhwan Im, Sunghwan Moon,
Il-Min Kim, Sangseok Yun, and Jae-Mo Kang [‡]

Abstract

Large language models (LLMs) have achieved near-human performance across diverse reasoning tasks, yet their deployment on resource-constrained Internet-of-Things (IoT) devices remains impractical due to massive parameter footprints and memory-intensive autoregressive decoding. While split computing offers a promising solution by partitioning model execution between edge devices and cloud servers, existing approaches fail to address the unique challenges of autoregressive inference, particularly the iterative token generation process and expanding key-value (KV) cache requirements. This work introduces the first autoregressive-aware split computing framework designed explicitly for LLM deployment on edge devices. Our approach makes three key contributions. First, we develop one-point split compression (OPSC), a mixed-precision quantization scheme that prevents out-of-memory failures by strategically partitioning models into front-end and back-end segments with different precision levels. Second, we propose a two-stage intermediate compression pipeline that combines threshold splitting (TS) and token-wise adaptive bit quantization (TAB-Q) to preserve accuracy-critical activations while dramatically reducing communication overhead. Third, we formulate a unified optimization framework that jointly selects optimal split points, quantization settings, and sequence lengths to satisfy strict memory and latency constraints. Extensive evaluations across diverse LLMs and hardware platforms demonstrate superior performance compared to state-of-the-art quantization methods, including SmoothQuant, OmniQuant, and Atom. The framework achieves a $1.49\times$ inference speedup and significant communication overhead reduction while maintaining or improving model accuracy. Notably, the approach enables deployment of models with hundreds of gigabytes of memory requirements on edge devices with severely constrained resources, making large-scale LLMs practically accessible for real-time IoT applications.

Keywords Collaborative intelligence, deep learning, large language models (LLMs), neural network compression, quantization, split computing.

1 INTRODUCTION

Deep neural networks (DNNs) have revolutionized fields such as computer vision, mobile sensing, and the Internet of Things (IoT), driving significant advancements in data analysis and decision-making. The groundbreaking transformer architecture introduced by Vaswani *et al.* [1] has further accelerated progress in

^{*}Mingyu Sung, Vikas Palakonda, Suhwan Im and Jae-Mo Kang are with the Department of Artificial Intelligence, Kyungpook National University, Daegu, South Korea (Corresponding author: Jae-Mo Kang, e-mail: jmkang@knu.ac.kr).

[†]Il-Min Kim is with the Department of Electrical and Computer Engineering, Queen’s University, Kingston, K7L 3N6, Canada.

[‡]Sangseok Yun is with the Department of Information and Communications Engineering, Pukyong National University, Busan 48513, South Korea (Corresponding author: Sangseok Yun, e-mail: ssyun@pknu.ac.kr).

natural language processing (NLP), facilitating the development of sophisticated large language models (LLMs) [2–4]. These models demonstrate remarkable capabilities in tasks including natural language understanding, natural language generation, complex reasoning, and code generation [5,6]. They power various groundbreaking applications, such as ChatGPT, GitHub Copilot, and the new Bing search experience [7]. Beyond NLP, LLMs have also shown promise in super-resolution, IoT sensor processing, image generation/synthesis, and voice processing [8].

Despite their exceptional performance, LLMs impose substantial computational and energy costs during both training and inference [9]. This challenge has become a critical operational burden for large technology corporations deploying LLMs as cloud-based services, particularly as user demand escalates exponentially.

Modern edge devices possess considerable processing capabilities and can handle substantial portions of these computations. However, the prevailing server-centric operational model concentrates the entire computational and financial burden on centralized infrastructure. This paradigm results in severe underutilization of high-performance edge resources, while demand for real-time, on-device processing in applications such as personalized AI continues to surge [10].

1.1 Related Work and Motivations

1.1.1 Merits of Split Computing

To bridge this gap, split computing (SC) has emerged as a promising paradigm, enabling collaborative inference that leverages the capabilities of both edge devices and cloud servers [11]. In this model, an edge device executes the initial network layers and offloads the intermediate outputs to the server, which completes the inference task. This method presents a viable alternative by distributing the computational workload, thus alleviating the server’s burden while utilizing the processing power of edge devices. However, its effectiveness depends on identifying optimal split points that balance on-device computation against communication overhead. Studies have explored various strategies to reduce inference latency, including selective splitting, architectural optimization, and intermediate feature compression [12–16]. However, most of these efforts focus on image-based models, leaving SC for LLMs remains underexplored.

1.1.2 Limitations of Split Computing for LLMs

Deploying LLMs on edge devices presents two crucial challenges: high computational demand and stringent memory constraints⁴. While SC should split a DNN between an edge device and the cloud, modern LLMs behave *autoregressively*, generating text by iteratively refeeding newly produced tokens through the entire model. This property complicates conventional SC in two ways. First, each freshly generated token must pass through all layers again, undermining naive strategies that place only the first few layers on the edge. Second, the repeated transformations of the growing token sequence can easily lead to out-of-memory (OOM) problems if the edge device lacks sufficient memory for the expanding intermediate states.

Although recent works on SC [18,19] have addressed smaller transformer models or partially transmitted hidden layer outputs, no study has fully tackled the massive parameter sizes and iterative nature of large-scale LLMs. For instance, Bajpai *et al.* [18] employed confidence-based splitting; their evaluation was limited to RoBERTa (123M parameters) without considering the complexities introduced by autoregressive inference. Ohta *et al.* [19] focused on privacy-sensitive intermediate outputs but did not explore split-layer optimization or the increased memory footprint from token-by-token generation. Consequently, naive application of existing SC frameworks to LLMs may yield suboptimal performance, OOM failures, or underutilization of edge resources.

1.1.3 Need for Split Computing in Deploying LLMs

Fig. 1 highlights the need for SC in deploying LLMs by illustrating three typical deployment scenarios: (a) local computing (b) edge computing, and (c) SC [20]. In the local computing scenario, researchers have attempted to compress and quantize LLMs to fit on a single edge device [17,21–25], but these methods encounter significant memory limitations and performance degradation. In contrast, the edge computing scenario offloads all inference tasks to a cloud server, with edge devices functioning solely as data transmitters. Although this simplifies on-device requirements, it risks overloading the server and underutilizing edge capabilities. By comparison, the SC approach enables LLM deployment by distributing computational and memory burdens between edge devices and the cloud, making it ideal for resource-constrained environments.

⁴For instance, GPT-3 requires about 1.7 s to process a 512-token input and generate a 32-token output on eight Nvidia A100 GPUs [17], which is impractical for most edge devices without extensive optimization.

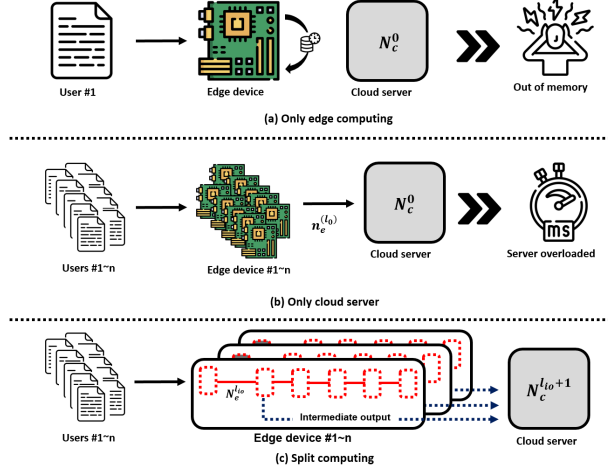


Figure 1: Schematic diagram of three scenarios for deploying LLM to edge devices. (a) local computing, (b) edge computing, and (c) split computing (SC)

1.2 Contributions

Given these challenges, we propose an SC strategy centered on three core objectives for efficiently deploying LLMs:

1. **Accommodating Autoregressive Inference:** Because LLMs re-invoke their entire architecture for each newly produced token, an SC design must effectively handle iterative data flow and avoid repeated large-scale data transfers between the edge and the cloud. In particular, we focus on leveraging the key-value (KV) cache.
2. **Compressing Massive Parameters:** Modern LLMs contain billions of parameters; thus, advanced compression and mixed-precision techniques are essential for preventing OOM errors and excessive latency on edge devices.
3. **Maximizing Edge Utilization:** Offloading most computations to the cloud wastes the increasing capabilities of edge hardware and risks bottlenecks in cloud resources. By intelligently selecting split layers and employing local inference, one can avoid overburdening the cloud and leverage the computational potential of the edge device.

By jointly addressing these three aspects, the proposed SC framework preserves high accuracy and minimizes resource usage, even under the demanding conditions of autoregressive generation. This approach ensures that large-scale LLMs become more accessible for diverse IoT and real-time AI scenarios, balancing on-device performance and cloud-based scalability.

The main contributions of this work are summarized as follows:

1. **Autoregressive-Aware Split Computing Framework:** This work proposes an autoregressive-aware split computing framework explicitly designed for large language model (LLM) inference. The framework systematically addresses token-by-token generation and key-value (KV) cache expansion through one-point split compression (OPSC), a mixed-precision quantization scheme that prevents out-of-memory failures on edge devices.
2. **Two-Stage Intermediate Feature Compression with Unified Optimization:** A novel two-stage compression pipeline combines threshold splitting (TS) and token-wise adaptive bit quantization (TAB-Q) to minimize communication overhead while preserving accuracy-critical activations. Additionally, a unified optimization framework jointly determines split points, quantization configurations, and sequence lengths under strict memory and latency constraints.
3. **Comprehensive Validation Across Models and Platforms:** Extensive experiments across diverse LLM architectures and hardware platforms demonstrate the effectiveness and generalizability of the proposed framework, establishing its practicality for resource-constrained edge-cloud deployments.

Table 1: List of notations used in this paper.

Notation	Description
D	Dimension of each head
H	Number of attention heads
T_w	Current hidden state tensor with length w
W	Maximum token sequence length
$Q^w = \{Q_1^w, Q_2^w\}$	Set of front/back weight quantization bits
$Q^a = \{Q_1^a, Q_2^a\}$	Set of front/back activation quantization bits
M, D	Memory and delay constraint of edge device
$M(\ell_w, Q^w)$	Total memory footprint of OPSC
$B_{kv}(w, \ell; Q^a)$	Incremental memory usage for KV cache
B_{io}	Intermediate output size with all parameters
T_{above}, T_{below}	Split tensors above/below threshold
τ	Threshold for splitting
Δ	Distortion tolerance for TAB-Q
A_Δ	Acceptable accuracy drop
\bar{Q}_a	Maximum activation quantization bits
γ	Signal-to-noise ratio
$\Psi(Q^a)$	Total activation-bit precision measure
ϵ	Target outage probability
$P_o(R)$	Outage probability at rate R
$L_\epsilon(D_{tx}; R)$	Worst-case latency for data size D_{tx}
$L_t(T_w, \ell, Q^a, I_{kv}; R)$	Total edge-device inference latency
I_{kv}	Indicator function for KV cache inclusion
$g(R)$	Rate optimization function

The remainder of the paper is organized as follows. Section II introduces the framework of the proposed method, and Section III presents the experimental setup and result analysis. Finally, Section IV concludes the paper.

2 Proposed Framework

2.1 One-Point Split Compression for Memory Constraint

This work considers a scenario in which multiple memory-constrained edge devices perform inference tasks for an LLM via SC with a single cloud server (Fig. 1(c)). The primary goal of the proposed scheme is to maximize edge device utilization while satisfying memory and latency constraints.

Strict edge-side memory budgets inherently cap the deployable model size, rendering aggressive LLM compression indispensable. Consequently, a number of techniques have been proposed (e.g., [22–24]). These lightweight approaches, however, introduce an accuracy–compression trade-off: excessive compression degrades performance, whereas insufficient compression can still trigger out-of-memory failures under SC deployments.

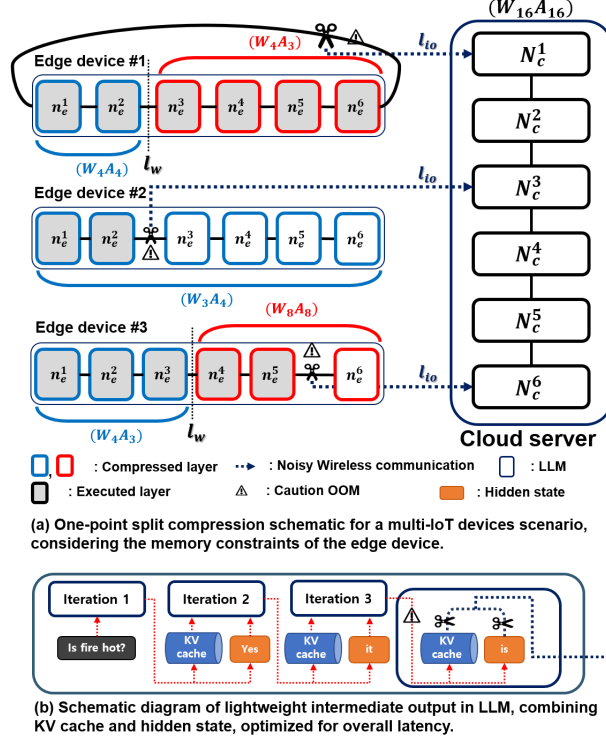


Figure 2: (a) One-point split compression schematic. (b) Intermediate output of LLM

To address this issue, we propose a *one-point split compression* (OPSC) method. It is particularly beneficial for the cloud server to maintain only a single, high-precision model⁵. As illustrated in Fig. 2(a), OPSC employs a mixed-precision technique⁶ to apply various quantization levels at a single split point in the model. In contrast to existing methods that assign layerwise quantization, thereby requiring separate fine-tuning for each device, OPSC partitions the model into front and back segments, each quantized differently, leading to less distortion than layerwise quantization and eliminating the need for additional fine-tuning for every device configuration. Consequently, it accommodates diverse memory constraints and compression settings across edge devices while preserving model accuracy, making it suitable for distributed environments with heterogeneous hardware resources⁷. Note that OPSC uses mixed-precision quantization to optimize model weights and parameters on edge devices.

Let us define the memory footprint of OPSC as follows:

$$M(\ell_w, \mathcal{Q}^w) = \sum_{i=1}^{\ell_w} B_w(i; Q_{w1}) + \sum_{j=\ell_w+1}^L B_w(j; Q_{w2}), \quad (1)$$

where $M(\ell_w, \mathcal{Q}^w)$ is the total memory footprint of the front (layers 1 to ℓ_w) and back segments (layers $\ell_w + 1$ to L) under distinct quantization levels. Here, $\mathcal{Q}^w = \{Q_{w1}, Q_{w2}\}$ denotes the front/back weight quantization bits.

⁵Compressing weights or biases of a DNN causes information loss. Thus, the original precision of the layers must be preserved to minimize any degradation in inference accuracy [12].

⁶Mixed-precision techniques are widely used for lightweight LLMs (e.g., [22, 24, 26–29]) and achieve excellent trade-offs between compression ratio and performance.

⁷We adopt the state-of-the-art LLM compression framework, Atom [24], which considerably reduces memory usage and computational cost while maintaining high accuracy.

2.2 Intermediate Output of LLM

2.2.1 Intermediate Output Definition

The core component of an LLM is its transformer decoder block, incorporating multi-head attention (MHA) to focus on salient information via query (Q), key (K), and value (V) vectors. In an *autoregressive* generation scheme, the model produces each token sequentially; once a token is generated, it is fed back into the model from the first layer. As the sequence length grows to w , the dimensions of Q , K , and V update accordingly:

$$Q, K, V \in \mathbb{R}^{w \times HD} \longrightarrow Q', K', V' \in \mathbb{R}^{(w+1) \times HD}.$$

However, repeatedly reprocessing all tokens is computationally expensive. To alleviate this, many LLMs use a *KV cache* to store key and value states for previously generated tokens,⁸ enabling *incremental decoding* in which only the newly generated token is processed at each step.

Although KV caching significantly improves inference speed, it may cause OOM errors on memory-constrained devices [30]. Hence, our SC framework carefully monitors the growth of the KV cache. We quantify how activation bit-widths affect KV-cache memory at each layer as follows:

$$\begin{aligned} B_{kv}(w, \ell; \mathcal{Q}^a) = & 2 \sum_{k=1}^{\ell} (T_w Q_{a,k}) + 2 \sum_{k=\ell+1}^L (T_{w-1} Q_{a,k}) \\ & + \underbrace{HD Q_{a,\ell}}_{\text{Memory usage of } w\text{-th token at layer } \ell}, T_w \in \mathbb{R}^{w \times HD}. \end{aligned} \quad (2)$$

With $T_w = wHD$ representing the size of the key (or value) tensor for w tokens, the required edge memory is

$$Q_{a,k} = \begin{cases} Q_{a1}, & k < \ell_w, \\ Q_{a2}, & k \geq \ell_w, \end{cases} \quad \ell_w: \text{Eq. (1)}.$$

The first term of Eq. (2) captures the KV tensors of the newly generated token w for layers computed on the edge ($1 \leq k \leq \ell$). The second term accounts for the KV tensors of the $(w-1)$ previously generated tokens, which must still be buffered for the $\ell+1 \leq k \leq L$ layers executed in the cloud. Finally, the $HD Q_{a,\ell}$ term represents the transient hidden state of token w at layer ℓ , which is produced locally and transmitted together with the KV cache.

This approach is vital for edge devices with limited capacity; by identifying when and how much KV cache expands, the system can efficiently control or offload computations without sacrificing fast, incremental generation. Finally, we can define the intermediate output as follows:

$$B_{io}(w, \ell, I_{kv}; \mathcal{Q}^a) = I_{kv} B_{kv}(w, \ell, \mathcal{Q}^a) + (1 - I_{kv}) T_w Q_{a,\ell}. \quad (3)$$

where I_{kv} is a factor that determines whether to transmit the KV cache ($I_{kv} = 1$) or only the hidden state ($I_{kv} = 0$). In noisy communication scenarios that demand strict latency bounds, sending large KV caches may be infeasible, prompting $I_{kv} = 0$ and thus transmitting only the layer outputs. Notably, the KV cache provides significant benefits for speed and server-side efficiency, as it avoids reprocessing all previously generated tokens during autoregressive decoding. However, the KV cache is much larger than transmitting hidden states because it accumulates key and value tensors across multiple layers. Therefore, depending on the communication capacity and latency requirements, sending only the hidden states (smaller but losing the benefits of the cache) may be more advantageous than the entire KV cache. This tradeoff is captured by the binary switch I_{kv} .

Even transmitting layer outputs alone can incur substantial overhead in poor communication conditions. Therefore, in the following subsection, we propose a method to compress these intermediate outputs and decide I_{kv} accordingly, mitigating such challenges.

2.3 Adaptive intermediate output compression technique

Although OPSC already compresses model parameters on the edge device, the intermediate activations generated at the split layer must be further compressed before they are transmitted to the cloud; without

⁸An LLM processes tokens one by one, appending the key and value for each newly generated token to the KV cache. This allows efficient “look back” at previously processed tokens without rerunning all computations.

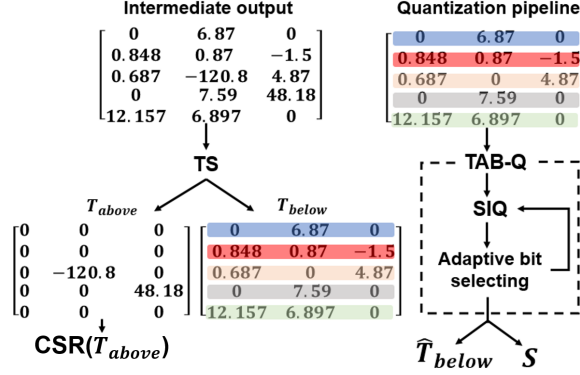


Figure 3: An example of the overall pipeline for applying the proposed intermediate output compression technique.

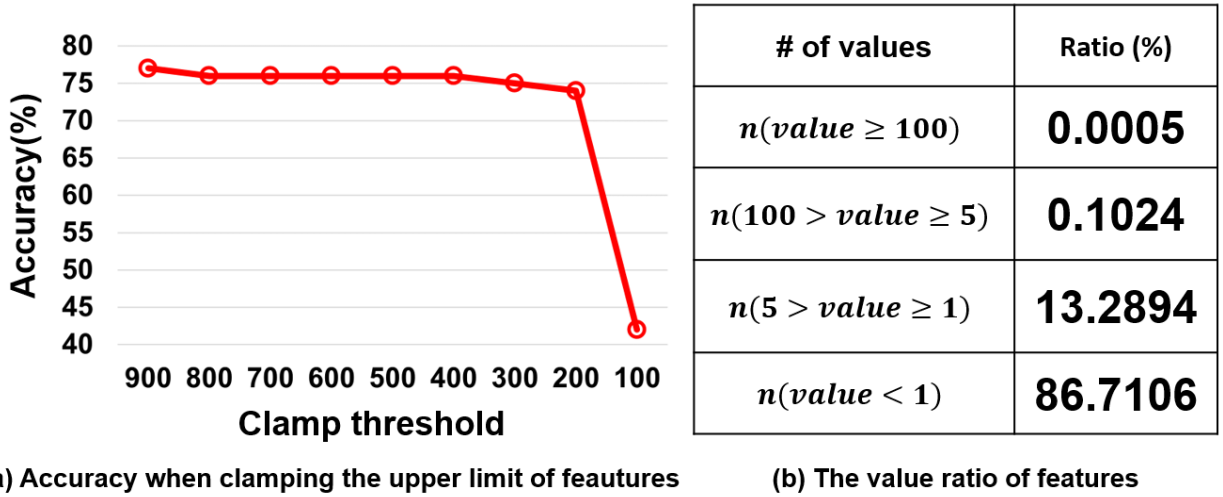


Figure 4: Effect of intermediate output magnitudes-based (Clamping) on the Llama-2 13B model on performance in HellaSwag. (a) Accuracy depending on the upper limit setting of the intermediate output’s large value. (b) Distribution of values in intermediate output.

this additional reduction, their sheer size would negate the bandwidth and latency advantages of SC. Fig. 3 presents an example of applying the proposed intermediate output compression technique. The proposed method compresses the intermediate output via a two-stage pipeline: TS and TAB-Q. The two-stage pipeline achieves substantial size reduction while incurring negligible accuracy loss.

2.3.1 Threshold Splitting

Because MHA is highly sensitive to outlier activations, quantizing high-magnitude elements can severely degrade accuracy. Fig. 4(a) illustrates the effect on accuracy when an upper limit is clamped on the intermediate output values to demonstrate the significance of these large values. A substantial change in accuracy was observed when only the values with absolute magnitudes exceeding 100 in the intermediate output were arbitrarily altered in these examples. This observation suggests that large values in intermediate output drastically influence LLM performance. Furthermore, Fig. 4(b) illustrates the distribution of intermediate output magnitudes. The data reveal that approximately 0.0005% of the intermediate output values exceed 100, while over 99% are 100 or below. This distribution suggests that the performance of LLM is highly dependent on a small fraction of values greater than 100, representing only 0.0005% of the total. As a result, Fig. 4 demonstrates that modifying or restricting these high-magnitude values adversely affects the model’s performance.

To preserve large-magnitude values that significantly impact the model, we first introduce the TS pipeline. Initially, the intermediate output⁹, \mathbf{T} , is partitioned into $\mathbf{T}_{\text{above}}$ and $\mathbf{T}_{\text{below}}$ using TS based on a specified threshold, τ . The formula for TS can be developed as follows:

$$\begin{aligned}\mathbf{T}_{\text{above}} &= \mathbf{T}_{ij} \cdot \mathbf{M}_{ij}, & \mathbf{T}_{\text{below}} &= \mathbf{T}_{ij} \cdot (1 - \mathbf{M}_{ij}), \\ \mathbf{M}_{ij} &= \begin{cases} 1 & \text{if } |\mathbf{T}_{ij}| \geq \tau \\ 0 & \text{otherwise} \end{cases}\end{aligned}\quad (4)$$

Hence, $\mathbf{T}_{\text{above}}$ becomes a highly sparse tensor and is compressed using the compressed sparse row (CSR) format [31], a widely recognized technique for sparse matrix compression. The characteristic of CSR indicates that the compression rate is positively correlated to the sparsity of the data (i.e., the higher the sparsity of the data, the higher the compression rate). Therefore, $\mathbf{T}_{\text{above}}$ can be transmitted with very low throughput without distortion [32]. This separation using TS helps to focus on the most significant values while minimizing the distortion of $\mathbf{T}_{\text{below}}$.

2.3.2 Token-Wise Adaptive Bit Integer Quantization

The remaining $\mathbf{T}_{\text{below}}$ is processed through the TAB-Q pipeline. This pipeline implements token-wise operations to preserve the model’s ability to differentiate contextual importance. In MHA, different weights are assigned to each token, enabling the model to infer the contextual significance of individual words. By applying operations on a token-wise basis, the relative importance disparities among tokens are maintained throughout the quantization process.

This work introduces the integer quantization (IQ) technique to compress $\mathbf{T}_{\text{below}}$ efficiently. The IQ technique is widely used for lightweight model deployment because it is straightforward to implement and compatible with most hardware architectures [26, 27, 33]. Specifically, we adopt a asymmetric integer quantization (AIQ) approach, which can be written as follows:

$$\hat{\mathbf{T}}, s, z = \text{AIQ}(\mathbf{T}, Q) = \left\lceil \frac{\mathbf{T}}{s} + z \right\rceil, \quad (5)$$

$$s = \frac{T_{\max} - T_{\min}}{Q_{\max}}, \quad z = \left\lfloor \frac{T_{\min}}{s} \right\rfloor, \quad Q_{\max} = 2^{(Q-1)} - 1. \quad (6)$$

where T_{\max} and T_{\min} are the maximum, minimum value in \mathbf{T} and Q represents the quantization level, respectively.

Although IQ is widely applicable, high-variance data can intensify quantization distortion. In NLP tasks, the distribution of intermediate outputs can vary significantly according to token attention, which complicates selecting a fixed quantization level. This work proposes TAB-Q, an adaptive algorithm that adjusts the quantization level based on the data distribution to address this problem. The AIQ approach provides fast computation that is particularly valuable in SC environments while preventing excessive bit usage or severe distortion.

Algorithm 1 presents the *TAB-Q* procedure, which adaptively adjusts the quantization level based on the data distribution and a predefined distortion tolerance, Δ . Below is a line-by-line overview:

- **Lines 1–2 (Sign and Magnitude Extraction):** The input tensor \mathbf{T} is decomposed into its sign component, \mathbf{T}_{sig} , and absolute value, $\bar{\mathbf{T}}$. Handling the sign and magnitude separately helps mitigate quantization distortion for high-variance data.
- **Lines 3–4 (Initial Quantization):** The variable Q is set to $\bar{Q} - 1$ because one bit is reserved for the sign. Then $\bar{\mathbf{T}}$ is quantized using the maximum level \bar{Q} to obtain the initial quantized tensor $\hat{\mathbf{T}}_0$ and scaling factor \mathbf{S}_0 .
- **Lines 5–9 (Adaptive Bit Reduction and Distortion Check):** The algorithm iteratively decreases Q , reapplies quantization, and measures the distortion δ . If δ *exceeds* Δ , the loop halts.

By terminating as soon as δ surpasses Δ , **TAB-Q** avoids excessive distortion that could compromise performance. This approach ensures the method preserves both computational and communication efficiency

⁹Whether the KV cache is included or not is determined by I_{kv} . However, in the proposed method, the KV cache and layer output are processed separately but in parallel.

Algorithm 1 TAB-Q

Require: $\mathbf{T}, \bar{Q}, \Delta$

- 1: $\mathbf{T}_{\text{sig}} \leftarrow \text{sign}(\mathbf{T})$
- 2: $\bar{\mathbf{T}} \leftarrow \text{abs}(\mathbf{T})$
- 3: $n \leftarrow W \times HD$ /* The number of elements, \mathbf{T}
- 4: $\bar{Q} \leftarrow \bar{Q} - 1$
- 5: $\hat{\mathbf{T}}_0, \mathbf{S}_0, \mathbf{Z}_0 \leftarrow \text{AIQ}(\bar{\mathbf{T}}, \bar{Q})$
- 6: **repeat**
- 7: $Q \leftarrow Q - 1$
- 8: $\hat{\mathbf{T}}, s, z \leftarrow \text{AIQ}(\bar{\mathbf{T}}, Q)$
- 9: $\delta \leftarrow \frac{\sum |\lfloor \hat{\mathbf{T}}_0 / 2^{(\bar{Q}-Q)} \rfloor - \hat{\mathbf{T}}|}{n}$
- 10: **if** $\delta > \Delta$ **then**
- 11: $\hat{\mathbf{T}}^* \leftarrow \hat{\mathbf{T}} \odot \mathbf{T}_{\text{sig}}, \hat{\mathbf{S}}^* \leftarrow s, \hat{\mathbf{Z}}^* \leftarrow z$
- 12: **break**
- 13: **end if**
- 14: **until** $Q < \bar{Q}$
- 15: **return** $\hat{\mathbf{T}}^*, \hat{\mathbf{S}}^*, \hat{\mathbf{Z}}^*, Q^*$

in SC environments. To leverage our complex quantization strategy, we introduce symmetric numeral systems (rANS) [34] for encoding, which can efficiently encode multiple quantum variables. Specifically, DietGPU [35] operates very efficiently because it utilizes GPUs for computation.

In contrast, the compressed intermediate output can be restored on the cloud server as follows:

$$\tilde{\mathbf{T}} = (\hat{\mathbf{T}}_{\text{below}}^* - \hat{\mathbf{Z}}^*) \odot \hat{\mathbf{S}}^* + \mathbf{T}_{\text{above}}. \quad (7)$$

Recovering the intermediate output is straightforward and efficient, even in the assumed many-to-one scenario, which helps to reduce the load on the cloud server.

2.4 Selection of Split Layer

2.4.1 Maximizing Activation Precision Under Memory Constraints

We now aim to find the split layer ℓ_w , the weight-quantization settings Q^w , and the “largest” activation-quantization settings Q^a that satisfy both an accuracy constraint and a memory limit. Define

$$\Psi(Q^a) = \sum_{k=1}^L Q_{a,k},$$

which measures the total activation-bit precision over all layers. Our objective is then to maximize $\Psi(Q^a)$. Formally,

$$(\ell_w^*, Q^{w*}, \bar{Q}^a) = \arg \max_{\ell_w, Q^w, Q^a} \Psi(Q^a), \quad (8a)$$

$$\text{s.t.: } A(\ell_w, Q^w, Q^a) \geq A_{\text{base}} - A_{\Delta}, \quad (8b)$$

$$M(\ell_w, Q^w) + B_{kv}(\bar{W}, \ell; Q^a) \leq \mathcal{M}. \quad (8c)$$

Here, $A_{\text{base}} - A_{\Delta}$ denotes the lower bound for acceptable accuracy, ensuring performance does not drop by more than A_{Δ} . The constraint (8c) ensures that the total memory usage $M(\ell_w, Q^w) + B_{kv}(\bar{W}, \ell; Q^a)$ does not exceed \mathcal{M} . Note that \bar{W} is treated as *fixed*: it corresponds to the maximum number of tokens the edge device is expected to generate and thus must be fully accommodated under the memory limit. The solution $(\ell_w^*, Q^{w*}, \bar{Q}^a)$ provides the configuration that maximizes overall activation precision without violating the accuracy or memory constraints.

Solution Approach Since ℓ_w , Q^w , and Q^a typically come from discrete sets (e.g., bitwidths 4, 8, 16), one can:

1. Set w to \bar{W} given maximum feasible token length (i.e., the largest token length we want to support on the edge device).
2. *Enumerate* all possible ℓ_w , Q^w , and Q^a .
3. *Check* each candidate configuration against constraints (8b)–(8c).
4. *Select* the combination $(\ell_w^*, Q^{w*}, \bar{Q}^a)$ that yields the largest $\Psi(Q^a)$.

Because \bar{T}_w is fixed and thus not minimized, the solution ensures the edge device can handle the *full* token length \bar{T}_w without running out of memory, while still maximizing the activation precision and maintaining accuracy within Δ .

2.4.2 Early Exit Strategy for Delay Constraints

Objective Function Setting While Eq (8) focuses on satisfying memory constraints without compromising accuracy, some applications further demand strict end-to-end latency guarantees. Let D denote the maximum allowable delay for completing a single inference. To analyze communication overhead, we adopt an ε -outage reliability framework, in which the worst-case latency for transmitting data of size D_{tx} at rate R is given by:

$$\mathcal{L}_\varepsilon(D_{\text{tx}}; R) = \frac{D_{\text{tx}}}{R} \left\lceil \frac{\ln(\varepsilon)}{\ln(P_o(R))} \right\rceil, \quad (9)$$

$$\text{where } P_o(R) = 1 - \exp\left(-\frac{2^{\frac{R}{W}} - 1}{\gamma}\right). \quad (10)$$

Here, $\varepsilon > 0$ is the target outage probability, W is the available bandwidth, and γ is the received signal-to-noise ratio (SNR).

On the computation side, let $\mathcal{L}_c(w)$ denote the local (on-device) processing time required for w tokens¹⁰. Combining both terms, the total edge-device inference latency when generating the w -th token up to layer ℓ is:

$$\begin{aligned} \mathcal{L}_t(T_w, \ell, Q^a, I_{kv}; R) &= \underbrace{\mathcal{L}_c(w)}_{\text{Local computation}} \\ &\quad + \underbrace{\mathcal{L}_\varepsilon(B_{\text{io}}(w, \ell, I_{kv}; Q^a), R)}_{\text{communication}}. \end{aligned} \quad (11)$$

To ensure timely inference, we impose the constraint $\mathcal{L}_t(T_w, \ell, Q^a, I_{kv}; R) \leq D$. Note that, because the cloud server’s computation and queueing latencies fluctuate with the instantaneous number of active clients—making precise analytical modeling impractical—the server instead communicates to each edge device a load-aware deadline that implicitly reflects its current operating state. We now formalize an *early-exit* objective, which permits the system to reduce the number of generated tokens or skip specific layers whenever the latency constraint would otherwise be violated. Let \bar{W} be the maximum number of tokens we are willing to generate. We define the following objective:

$$\max_{\substack{0 \leq w \leq \bar{W} \\ 1 \leq \ell \leq L}} w \ell \quad \text{subject to} \quad \mathcal{L}_t(T_w, \ell, Q^a, I_{kv}; R) \leq D. \quad (12)$$

Here, $w \times \ell$ measures the total inference “depth” (in terms of tokens and layers) that can be processed before exceeding the target latency D . Larger values of $w \times \ell$ typically correspond to higher-quality outputs.

¹⁰Local computation latency was profiled in real time on the target edge device.

Optimization Notice that $\mathcal{L}_\epsilon(\cdot, R)$ in (9) is non-monotonic in R . Following the approach in [13], define the function

$$g(R) = \frac{\ln(1/P_o(R))}{R},$$

and constrain R to lie in a feasible interval $[\underline{R}, \bar{R}]$. Then the optimal rate R^* that minimizes $g(\cdot)$ and hence yields the smallest worst-case communication delay is found via:

$$R^* = \arg \min_{R \in [\underline{R}, \bar{R}]} g(R). \quad (13)$$

In practice, the solution R^* can be found by simple one-dimensional search.

Having solved (13) for R^* , we then use (12) to determine how many tokens w and which layers ℓ can be processed while keeping total latency below D . As soon as $\mathcal{L}_t(\cdot)$ threatens to exceed D , an early-exit decision is triggered: the system either reduces the number of tokens, disables KV caching ($I_{kv} = 0$), or compresses the intermediate outputs more aggressively.

Algorithmic Solution Algorithm 2 details our proposed approach, which integrates the memory-feasible configuration $(\ell_w^*, Q^{w*}, \bar{Q}^a)$ obtained from (8) with a real-time monitoring of the total latency \mathcal{L}_t . As the model processes each token, we evaluate whether sending the intermediate outputs (possibly including the KV cache) at the current precision will exceed the time limit D . If it does, we apply additional compression steps (e.g., skipping KV caching or reducing token count) until the latency is once again under control. By adaptively trading off token generation depth and intermediate-output compression, the algorithm ensures adherence to strict delay budgets even under time-varying network conditions.

3 Evaluation

3.1 Experimental setup

We validate the proposed SC method on two Llama2 variants—7B-hf (7B) and 13B-hf (13B) [36]—each comprising 32 and 40 decoder layers, respectively. In this work, we treat all decoder layers as a single “splittable” stack, allowing the split point ℓ to vary from 1 to 32 (for 7B) or 1 to 40 (for 13B). Unless otherwise noted, we fix the split layer at $\ell = 20$, the threshold at $\tau = 5$, the acceptable distortion $A_\Delta = 1\%$, and $\Delta = 0.2$, which balances accuracy and communication overhead based on our preliminary tests. Note that the clip for activation in OPSC settings is not used. For the communication experiments, we set $\varepsilon = 0.001$, $W = 10$ MHz, $\sigma_h^2 = 1$, and $\gamma = 10$. We run the edge inference on a Jetson Xavier NX (16 GB) and the cloud inference on an A6000 GPU. All evaluations are conducted in a zero-shot setting on HellaSwag (HS) [37], PIQA [38], ARC-e/c [39], BoolQ [40], and Winogrande (Wino.) [41]. Models are assessed using pretrained weights and fixed prompt templates without task-specific fine-tuning or in-context demonstrations. This experimental design isolates the effects of the proposed split-compression framework from confounding factors, reflects realistic deployment scenarios where edge devices cannot fine-tune large models, and ensures fair comparisons across benchmark tasks.

3.2 Performance comparison

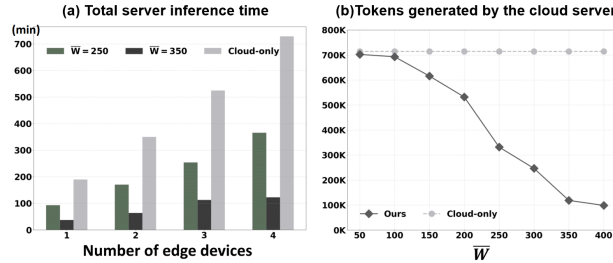


Figure 5: (a) Total server inference time (in minutes) versus the number of edge devices for three configurations: ‘Cloud-only’ (all tokens processed by the server) and our SC method with $\bar{W} = 250$ and $\bar{W} = 350$. (b) Number of tokens generated by the server as \bar{W} varies. Our approach gradually offloads more inference steps to the edge device, significantly reducing both server inference time and token generation overhead.

Algorithm 2 Early Exit Strategy under Delay Constraints**Require:** $(\ell_w^*, Q^{w*}, \bar{Q}^a)$ satisfying (8) for a maximum token count \bar{W} .1: Delay tolerance D .2: Latency function $\mathcal{L}_t(T_w, \ell, \bar{Q}^a, I_{kv}; R)$ from (11).

3:

Ensure: $w^* \leq \bar{W}$ maximizing $w \times \ell$ s.t. $\mathcal{L}_t \leq D$.4: **Compute** R^* using (13)5: $w \leftarrow 0, I_{kv} \leftarrow 1$ 6: **for** $w = 1$ to \bar{W} **do**7: **for** $\ell = 1$ to L **do**8: Forward pass up to layer ℓ for token w 9: latency $\leftarrow \mathcal{L}_t(T_w, \ell, \bar{Q}^a, I_{kv}; R^*)$ 10: **if** latency $> D$ **then**11: $\hat{T}_w \leftarrow \text{TABQ}(T_w; I_{kv})$ ▷ Compress12: latency $\leftarrow \mathcal{L}_t(\hat{T}_w, \ell, \bar{Q}^a, I_{kv}; R^*)$ 13: **if** latency $\leq D$ **then**14: **return** \hat{T}_w ▷ Early exit successful15: **else**16: $I_{kv} \leftarrow 0$ 17: $\hat{T}_w \leftarrow \text{TABQ}(T_w; I_{kv})$ 18: latency $\leftarrow \mathcal{L}_t(\hat{T}_w, \ell, \bar{Q}^a, I_{kv}; R^*)$ 19: **if** latency $> D$ **then**20: **while** latency $> D$ **do**21: $w \leftarrow w - 1$ ▷ Reduce token22: latency $\leftarrow \mathcal{L}_t(\hat{T}_w, \ell, \bar{Q}^a, I_{kv}; R^*)$ 23: **end while**24: **return** \hat{T}_w ▷ Early exit25: **end if**26: **end if**27: **end if**28: **end for**29: **end for**30: $\hat{T}_w \leftarrow \text{TABQ}(T_w; I_{kv})$ 31: **return** \hat{T}_w

Fig. 5(a) displays the server's total inference time as the number of edge devices increases. We compare a baseline in which all tokens are processed by the server ("Cloud-only") against two SC variants with maximum sequence lengths of $\bar{W} = 250$ or $\bar{W} = 350$ on the edge devices. As the number of connected edge devices increases, the proposed SC approach maintains a lower server workload than the Cloud-only scheme, demonstrating superior scalability. We also note in Fig. 5(a) that the server inference time exhibits nonlinear growth with increasing edge device count. This nonlinear behavior stems from server-side bottlenecks, including dynamic batching overhead, queueing delays, and GPU memory management constraints, which collectively intensify computational overhead under high concurrency conditions. Despite these practical limitations, the proposed SC framework consistently achieves lower server latency than the cloud-only baseline, demonstrating its scalability benefits in realistic multi-user deployments. Fig. 5(b) further illustrates the number of tokens generated on the server for different values of \bar{W} . The proposed method maximizes the on-board inference at the edge, significantly reducing the number of tokens the server must generate. In contrast, the Cloud-only approach saturates at a high token count irrespective of \bar{W} . These findings demonstrate that the proposed framework effectively reduces server-side overhead while preserving inference quality.

Fig. 6 illustrates the relationship between intermediate output size (in kilobytes) and the maximum token length \bar{W} across various threshold (τ) and maximum activation-bit (\bar{Q}^a) settings. The Baseline curve corresponds to transmitting the uncompressed hidden states or KV caches, yielding the most significant data size. In contrast, the two-stage compression framework (TS + TAB-Q) reduces the data size substantially

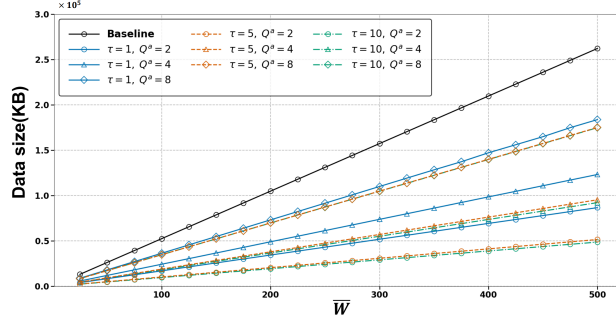


Figure 6: Data size of intermediate outputs versus token length W , comparing our proposed compression approach for various threshold values $\tau \in \{1, 5, 10\}$ and maximum activation bits $\bar{Q}^a \in \{2, 4, 8\}$. The “Baseline” line (black circles) denotes no compression, resulting in the largest data size.

by isolating the large-magnitude values (via τ) and then performing token-wise integer quantization (via \bar{Q}^a). The higher τ values filter out larger elements, resulting in increased sparsity, whereas lower \bar{Q}^a values enforce more aggressive quantization. Therefore, varying τ and \bar{Q}^a allows flexible control over communication overhead, making it feasible to process extended token sequences with minimal data transfer requirements.

Table 2 compares zero-shot performance when deploying the 7B model under tight memory constraints on the edge device. We contrast two schemes: (i) fully lightweight quantization using Atom [24], and (ii) the proposed SC method with $\bar{W} = 50$, employing the adaptive intermediate output compression at the split layer ($\tau = 5$, $\bar{Q}^a = 4$). We tested multiple split layers (ℓ) in each setting. Overall, the proposed approach achieves higher accuracy than Atom across all datasets and split layers. By compressing only the portion of the model that must reside on the edge device while offloading the remainder to the cloud, the proposed framework mitigates the accuracy degradation often caused by aggressive quantization. Our method outperforms Atom through three advantages: TS preserves critical values without quantization loss versus Atom’s 8-bit outlier quantization; TAB-Q enables per-token adaptation versus static quantization; and SC architecture allows cloud-side full-precision computation.

Table 2: Performance comparison across different split layers.

ℓ	Method	PIQA	ARC-e	BoolQ	HS	Wino.
5	Atom	75.63	52.69	67.71	67.18	64.48
	Ours	76.50	54.12	70.67	70.73	67.32
10	Atom	74.86	51.73	65.84	66.57	63.69
	Ours	76.01	52.15	68.84	68.35	66.38
15	Atom	75.08	51.43	67.98	67.81	64.56
	Ours	76.33	52.44	69.60	68.57	65.27
20	Atom	75.73	52.40	67.25	68.23	65.04
	Ours	76.17	53.37	67.22	68.63	65.11
25	Atom	75.63	52.69	67.28	68.56	65.04
	Ours	75.79	53.11	68.17	68.65	65.90
30	Atom	76.06	52.40	67.19	68.49	64.40
	Ours	76.12	52.57	67.31	68.55	64.25

Table 3 shows the accuracy of our proposed SC compression strategy versus three established quantization techniques—SmoothQuant (E1) [22], OmniQuant (E2) [23], and Atom (E3) [24]—for both 7B and 13B models. All approaches apply $Q_{w1} = 4$ and $Q_{w2} = 4$ for weight quantization; the table reports results under different activation-bit settings, $Q^a \in \{3, 4\}$. Unlike traditional activation quantization methods developed for general inference settings, the proposed approach more effectively applies quantization at the split layer, making it highly suitable for SC environments. The crucial advantage of the proposed method is its ability to balance the tradeoff between minimizing communication delay and maximizing accuracy in edge-cloud

systems. Strict delay requirements are met without compromising accuracy by applying intermediate output compression, even in resource-constrained environments.

Table 3: Comparison with different LLM compression techniques.

\tilde{Q}^a	Method	PIQA	ARC-e	ARC-c	BoolQ	HS	Wino.
7B							
3	E-1	51.88	29.33	31.84	46.07	29.01	50.63
	E-2	53.05	30.70	30.77	38.53	28.59	51.03
	E-3	67.79	43.73	31.40	62.35	55.26	57.38
	Ours	75.19	52.65	38.05	73.21	69.22	63.30
4	E1	62.30	40.00	31.37	59.32	43.16	47.00
	E2	65.30	45.17	30.94	64.43	56.16	47.56
	E3	75.46	51.43	38.91	68.47	69.67	63.30
	Ours	76.33	54.63	40.44	74.43	70.77	67.09
13B							
3	E1	48.25	27.18	29.12	48.85	25.65	51.15
	E2	50.49	27.67	29.30	39.40	25.77	52.63
	E3	68.28	47.22	34.64	65.11	60.79	56.43
	Ours	70.46	49.58	35.49	66.09	62.92	58.48
4	E1	64.15	40.50	30.52	62.29	46.75	50.92
	E2	65.35	45.97	32.71	62.84	59.05	54.96
	E3	77.31	55.85	42.41	67.52	73.88	67.48
	Ours	78.07	57.49	43.33	70.15	74.76	69.69

Further, the Tables 2 and 3 demonstrate that the proposed framework consistently outperforms Atom under identical memory constraints. This performance advantage results from the split-aware architecture, where only the front-end layers executed on edge devices undergo quantization, while the back-end layers maintain complete precision on the server. This design preserves semantic fidelity in the most critical stages of inference. In contrast, Atom applies uniform quantization across the entire model, including accuracy-sensitive final layers, thereby introducing additional distortion that degrades overall performance.

To evaluate the effectiveness of our OPSC quantization strategy under different \mathcal{Q}^w configurations, we conducted detailed tests on both the front- and back-end portions of the Llama2 model. Table 4 summarizes the perplexity¹¹ on the WikiText2 and C4 datasets for the 7B and 13B variants. Quantifying fewer layers in the front-end method yields lower perplexity, implying that the final layers are susceptible to precision reduction. Quantizing the back-end (i.e., the last layers) generally produces slightly higher perplexity at the same ℓ_w , confirming that later layers play a more critical role in accurate language modeling. For both 7B and 13B models, increasing ℓ_w leads to progressively higher perplexity, as more of the network is placed under 4-bit quantization. These findings highlight the importance of strategically selecting which layers to compress to balance memory savings with minimal performance loss in SC scenarios.

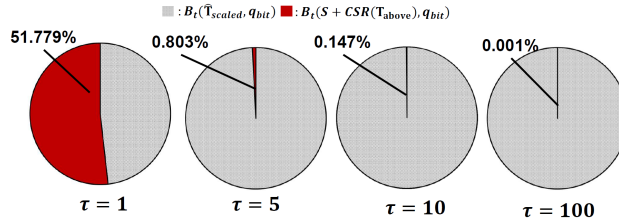


Figure 7: Data size ratio of $\hat{\mathbf{T}}_{\text{below}}$ (gray) and $\mathbf{T}_{\text{above}}$ (red) according to τ

Fig. 7 illustrates how the required data size of intermediate outputs varies with τ , and how it contributes to $\mathbf{T}_{\text{below}}$ and $\mathbf{T}_{\text{above}}$. When τ is 1, the cost of compressing and transmitting $\mathbf{T}_{\text{above}}$ is high, resulting

¹¹Perplexity measures how well a language model predicts a given text. Lower perplexity indicates better predictive performance, meaning the model is less “confused” about the next word.

Table 4: Perplexity of quantized Llama models with split layer quantization on WikiText2 and C4 datasets

	ℓ_w	front-end method (\downarrow)		ℓ_w	back-end method (\downarrow)	
		Wiki	C4		Wiki	C4
7B	4	5.538	7.030	4	5.607	7.123
	8	5.608	7.115	8	5.694	7.222
	12	5.682	7.206	12	5.781	7.338
	16	5.739	7.302	16	5.831	7.427
	20	5.840	7.435	20	6.074	7.684
	24	5.920	7.562	24	6.394	7.999
	28	5.997	7.673	28	6.566	8.160
	32	6.030	7.755	32	6.103	7.818
13B	4	4.915	6.496	4	4.950	6.531
	8	4.956	6.537	8	5.642	7.429
	12	5.000	6.585	12	5.032	6.623
	16	5.049	6.643	16	5.094	6.694
	20	5.096	6.705	20	5.172	6.784
	24	5.139	6.766	24	5.357	7.019
	28	5.180	6.821	28	5.531	7.281
	32	5.222	6.874	32	5.642	7.429
	36	5.261	6.926	36	5.704	7.468
	40	5.312	6.991	40	5.312	6.991

in a reduced compression ratio. However, when τ exceeds 1, the sparsity of $\mathbf{T}_{\text{above}}$ increases significantly. Consequently, the impact on the transmission latency of $\mathbf{T}_{\text{above}}$ becomes negligible.

Table 5 evaluates the impact of the proposed two-stage compression by comparing the 13B model with TAB-Q alone against the combined TS + TAB-Q design. The baseline involves no intermediate output compression, whereas ‘Baseline + TAB-Q’ applies quantization alone. Although TAB-Q significantly reduces accuracy when used in isolation (e.g., HellaSwag drops from 77.31% to 45.26%), adding TS (‘Baseline + TS + TAB-Q’) restores performance nearly to baseline levels. This observation demonstrates that TS effectively preserves large-magnitude values critical to the model, mitigating the distortion introduced by TAB-Q.

Table 5: Ablation study on our proposed method in 13B

Ablation study	HS	ARC-e	ARC-c	PIQA
Baseline	77.31	79.12	43.96	78.89
Baseline+TAB-Q	45.26	33.68	24.50	55.60
Baseline+TS+TAB-Q	77.09	76.61	43.62	78.02

Further, to evaluate the cross-model generalization capability of the proposed framework, Table 6 compares baseline performance [42–45] with our method across four distinct LLMs on five benchmark tasks (ARC-e, ARC-c, BoolQ, HellaSwag, and Wino). Values in **red** denote improvements over the baseline, while those in **blue** indicate performance drops.

The proposed approach preserves or enhances accuracy in most cases, demonstrating compatibility with diverse model architectures. Where minor performance declines occur, the associated memory and communication efficiency gains provide favorable trade-offs, particularly in resource-constrained edge-cloud deployments. The experimental parameters reflect an optimized balance between quantization aggressiveness and acceptable performance degradation. These results demonstrate that the proposed framework maintains inference accuracy across heterogeneous LLM architectures while consistently reducing server computational requirements, confirming its broad applicability beyond the baseline evaluation models.

4 Conclusion

This paper presented an autoregressive-aware split computing framework for deploying large language models (LLMs) on memory- and latency-constrained edge devices. The framework integrated one-point

Table 6: Benchmark results for various LLMs across multiple tasks. For each model, the *first row* lists its baseline performance, whereas the *second row* shows results after applying our proposed method.

Model	ARC-e	ARC-c	BoolQ	HS	Wino.
Qwen2.5-14B	93.64	91.64	89.76	80.45	73.95
+ <i>Ours</i>	93.56	91.98	89.79	80.46	74.27
Mistral-Nemo-Instruct-2407	88.34	81.14	89.76	80.40	71.19
+ <i>Ours</i>	88.55	81.14	87.06	80.22	71.19
Llama-3.1-8B-Instruct	88.64	81.57	83.67	77.45	68.82
+ <i>Ours</i>	88.47	81.06	83.27	77.25	68.67
Phi-4	93.52	91.98	86.18	79.31	81.45
+ <i>Ours</i>	93.60	91.98	86.09	79.20	81.69

split compression (OPSC) to prevent out-of-memory failures, a two-stage intermediate output compression pipeline (TS+TAB-Q) to preserve accuracy while reducing communication costs, and a unified optimization strategy that jointly determined split placement, quantization settings, and sequence length under strict system constraints. Extensive experiments across multiple LLMs, hardware platforms, and benchmark tasks demonstrated that the framework consistently outperformed state-of-the-art quantization methods, including Atom, SmoothQuant, and OmniQuant. The framework significantly reduced server-side computation, communication overhead, and end-to-end latency while maintaining or improving accuracy. These results confirmed that the proposed method was effective and scalable for practical edge-cloud LLM deployments.

References

- [1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. 30:5998–6008, Dec 2017.
- [2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. In *Proc. 2019 Conf. North American Chapter Assoc. Comput. Linguistics: Human Lang. Technol. (NAACL-HLT)*, pages 4171–4186, Jun 2019.
- [3] 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. In *Proc. 34th Int. Conf. Neural Inf. Process. Syst.*, volume 33, pages 1877–1901, Dec 2020.
- [4] Aaron Grattafiori, Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Alex Vaughan, et al. The llama 3 herd of models. *arXiv preprint arXiv:2407.21783*, 2024.
- [5] Shervin Minaee, Tom Mikolov, Narjes Nikzad, Meysam Asgari Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey. *arXiv preprint arXiv:2402.06196*, 2024.
- [6] Narendra Patwardhan, Stefano Marrone, and Carlo Sansone. Transformers in the real world: A survey on nlp applications. *Information*, 14(4):242, 2023.
- [7] 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.
- [8] Yiheng Liu, Tianle Han, Siyuan Ma, Jiayue Zhang, Yuanyuan Yang, Jiaming Tian, Hao He, Antong Li, Mengshen He, Zhengliang Liu, et al. Summary of chatgpt-related research and perspective towards the future of large language models. *Meta-radiology*, 1(2):100017, 2023.
- [9] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, et al. A survey of large language models. *arXiv preprint arXiv:2303.18223*, 1(2), 2023.
- [10] Haozhao Wang, Zhihao Qu, Qihua Zhou, Haobo Zhang, Boyuan Luo, Wenchao Xu, Song Guo, and Ruixuan Li. A comprehensive survey on training acceleration for large machine learning models in iot. *IEEE Internet Things J.*, 9(2):939–963, Sep. 2021.

- [11] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Comput. Archit. News*, 45(1):615–629, 2017.
- [12] Mingyu Sung, Vikas Palakonda, Il-Min Kim, Sangseok Yun, and Jae-Mo Kang. Deco-mesc: Deep compression-based memory-constrained split computing framework for cooperative inference of neural network. *IEEE Trans. Veh. Technol.*, pages 1–6, 2025.
- [13] Sangseok Yun, Wan Choi, and Il-Min Kim. Cooperative inference of dnns for delay-and memory-constrained wireless iot systems. *IEEE Internet Things J.*, 9(17):16113–16127, Feb. 2022.
- [14] Arian Bakhtiarnia, Nemanja Milošević, Qi Zhang, Dragana Bajović, and Alexandros Iosifidis. Dynamic split computing for efficient deep edge intelligence. In *Proc. 2023 IEEE Int. Conf. Acoustics, Speech Signal Process. (ICASSP)*, pages 1–5. IEEE, 2023.
- [15] Robert A Cohen, Hyomin Choi, and Ivan V Bajić. Lightweight compression of neural network feature tensors for collaborative intelligence. In *Proc. 2020 IEEE Int. Conf. Multimedia Expo (ICME)*, pages 1–6. IEEE, 2020.
- [16] Yongjeong Oh, Jaeho Lee, Christopher G. Brinton, and Yo-Seb Jeon. Communication-efficient split learning via adaptive feature-wise compression. *IEEE Trans. Neural Netw. Learn. Syst.*, 2025.
- [17] Zheng Lin, Guanqiao Qu, Qiyuan Chen, Xianhao Chen, Zhe Chen, and Kaibin Huang. Pushing large language models to the 6g edge: Vision, challenges, and opportunities. *arXiv preprint arXiv:2309.16739*, 2023.
- [18] Divya Jyoti Bajpai, Vivek Kumar Trivedi, Sohan L Yadav, and Manjesh Kumar Hanawal. Splitee: Early exit in deep neural networks with split computing. In *Proc. 3rd Int. Conf. AI-ML Syst.*, pages 1–9, 2023.
- [19] Shoki Ohta and Takayuki Nishio. λ -split: A privacy-preserving split computing framework for cloud-powered generative ai. *arXiv preprint arXiv:2310.14651*, 2023.
- [20] Yoshitomo Matsubara, Marco Levorato, and Francesco Restuccia. Split computing and early exiting for deep learning applications: Survey and research challenges. *ACM Comput. Surv.*, 55(5):1–30, 2022.
- [21] Rongjie Yi, Liwei Guo, Shiyun Wei, Ao Zhou, Shangguang Wang, and Mengwei Xu. Edgemoe: Fast on-device inference of moe-based large language models. *arXiv preprint arXiv:2308.14352*, 2023.
- [22] 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 *Proc. 40th Int. Conf. Mach. Learn.*, pages 38087–38099. PMLR, 2023.
- [23] 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.
- [24] 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. *Proc. Mach. Learn. Syst.*, 6:196–209, 2024.
- [25] Ruiyang Qin, Jun Xia, Zheng Jia, Meng Jiang, Ahmed Abbasi, Peipei Zhou, Jingtong Hu, and Yiyu Shi. Enabling on-device large language model personalization with self-supervised data selection and synthesis. In *Proc. ACM/IEEE Des. Autom. Conf. (DAC)*, pages 1–6, 2024.
- [26] Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.
- [27] Elias Frantar, Saleh Ashkboos, Torsten Hoeftler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.
- [28] Germà Coenders and Núria Arimany Serrat. Accounting statement analysis at industry level. a gentle introduction to the compositional approach. *arXiv preprint arXiv:2305.16842*, 2023.
- [29] 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. *Proc. Mach. Learn. Syst.*, 6:87–100, 2024.
- [30] 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 *Proc. 29th Symp. Operating Syst. Principles (SOSP)*, pages 611–626, 2023.

- [31] Zbigniew Koza, Maciej Matyka, Sebastian Szkodas, and Łukasz Mirosław. Compressed multirow storage format for sparse matrices on graphics processing units. *SIAM J. Sci. Comput.*, 36(2):C219–C239, 2014.
- [32] Zhaoyang Du, Yijin Guan, Tianchan Guan, Dimin Niu, Nianxiong Tan, Xiaopeng Yu, Hongzhong Zheng, Jianyi Meng, Xiaolang Yan, and Yuan Xie. Predicting the output structure of sparse matrix multiplication with sampled compression ratio. In *Proc. IEEE 28th Int. Conf. Parallel Distributed Syst. (ICPADS)*, pages 483–490. IEEE, 2023.
- [33] Benoit Jacob, Skirmantas Kligys, Bo Chen, Menglong Zhu, Matthew Tang, Andrew Howard, Hartwig Adam, and Dmitry Kalenichenko. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, pages 2704–2713, 2018.
- [34] Jarek Duda. Asymmetric numeral systems: entropy coding combining speed of huffman coding with compression rate of arithmetic coding. *arXiv preprint arXiv:1311.2540*, 2013.
- [35] J. Johnson. DietGPU: GPU-based lossless compression for numerical data. *arXiv preprint arXiv:1311.2540*, 2013.
- [36] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.
- [37] Rowan Zellers, Ari Holtzman, Yonatan Bisk, Ali Farhadi, and Yejin Choi. Hellaswag: Can a machine really finish your sentence? *arXiv preprint arXiv:1905.07830*, 2019.
- [38] Yonatan Bisk, Rowan Zellers, Jianfeng Gao, Yejin Choi, et al. Piqa: Reasoning about physical commonsense in natural language. In *Proc. AAAI Conf. Artif. Intell.*, volume 34, pages 7432–7439, 2020.
- [39] Peter Clark, Isaac Cowhey, Oren Etzioni, Tushar Khot, Ashish Sabharwal, Carissa Schoenick, and Oyvind Tafjord. Think you have solved question answering? try arc, the ai2 reasoning challenge. *arXiv preprint arXiv:1803.05457*, 2018.
- [40] Christopher Clark, Kenton Lee, Ming-Wei Chang, Tom Kwiatkowski, Michael Collins, and Kristina Toutanova. Boolq: Exploring the surprising difficulty of natural yes/no questions. *arXiv preprint arXiv:1905.10044*, 2019.
- [41] Keisuke Sakaguchi, Ronan Le Bras, Chandra Bhagavatula, and Yejin Choi. Winogrande: An adversarial winograd schema challenge at scale. *Commun. ACM*, 64(9):99–106, 2021.
- [42] Qwen Team. Qwen2.5: A party of foundation models. Sep 2024.
- [43] Mistral AI. Mistral NeMo: Collaborative innovation with NVIDIA. 2024. Accessed: Oct. 14, 2024.
- [44] Meta. Introducing llama 3.1: Our most capable models to date. 2024. Accessed: Oct. 14, 2024.
- [45] M. Abdin, J. Aneja, H. Behl, S. Bubeck, R. Eldan, S. Gunasekar, and et al. Phi-4 technical report. 2024.