HCInfer: An Efficient Inference System via Heterogeneous Error Compensation for Resource-Constrained Devices

Shen Xu[1], **Xiangwen Zhuge**[1], **Zhe Xu**[1], **Yingkun Hu**[1], **Zheng Yang**[1], **Yunhao Liu**[1]
[1]Tsinghua University

Abstract

LLMs often struggle with memory-constrained deployment on consumer-grade hardware due to their massive parameter sizes. While existing solutions such as model compression and offloading improve deployment feasibility, they often suffer from substantial accuracy degradation or severe throughput bottlenecks. Recent error compensation methods recover accuracy through auxiliary LoRA-style branches, and we observe that these branches are inherently amenable to offloading: they require substantial parameter storage but access only a small subset of compensation parameters during each inference step. Motivated by this opportunity, we propose HCInfer, a heterogeneous inference system that offloads error compensation to the CPU while executing the compressed backbone on the GPU, and further introduces an asynchronous compensation pipeline and sensitivity-aware dynamic rank allocation to hide compensation overhead and maximize accuracy recovery. Experimental results show that HCInfer achieves a maximum accuracy improvement of 5.2% on downstream tasks compared to compression model and sustaining a maximum speedup of 10.4x compared to full-precision model.

1 Introduction

Large language models(LLMs) continue to demonstrate breakthroughs in scenarios such as autonomous scientific [1], multi-agent collaboration [2], and complex decision-making [3]. However, deploying LLMs with tens of billions of parameters on consumer-grade hardware remains challenging due to limited GPU memory. For example, Qwen3-30B-A3B [4] requires over 60 GB of memory for parameter storage, far exceeding the 24 GB VRAM capacity of a representative consumer GPU such as the NVIDIA GeForce RTX 4090.

To improve the accessibility of LLMs on consumer-grade hardware, prior work has followed two main technical directions. (1) Compression methods trade weight precision for storage reduction and computational acceleration [5, 6, 7], as illustrated in Figure 1(a). Although such methods can fully utilize GPUs and achieve high inference throughput, model accuracy often drops substantially under extremely aggressive compression ratios. (2) Offloading methods store full-precision parameters that cannot fit into GPU memory in CPU memory and load them to GPU memory on demand during inference [8, 9, 10]. Although offloading preserves the original model accuracy, its throughput is limited by the bandwidth bottleneck of the PCIe bus. Frequent data movement often causes orders-of-magnitude degradation in inference throughput, as shown in Figure 1(b).

The limitations of existing methods naturally raise the following question: *Can we design a new LLM deployment approach that preserves model accuracy as much as possible while still achieving high inference throughput?*

Recent progress in low-rank quantization residual compensation provides a promising direction to resolve this tension [11, 12]. Methods observe that quantization error matrices exhibit clear low-rank

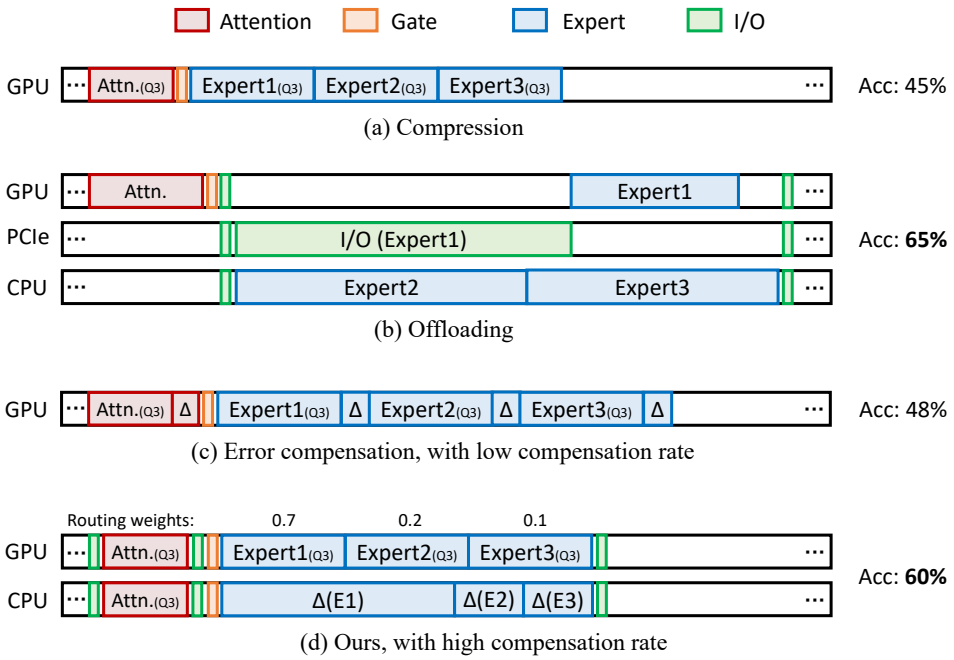


Figure 1: Overview of different inference paradigms.

structure [11]. By approximating the quantization error with LoRA-style low-rank matrices, these methods can repair quantization errors with limited additional computation, thereby improving inference accuracy, as illustrated in Figure 1(c). However, when deploying such compensation methods on devices where GPU memory is already highly constrained, the remaining GPU memory can store only a very limited number of compensation parameters, resulting in restricted compensation effectiveness and narrow applicability.

To address this challenge, we observe that the two components of low-rank compensation methods are naturally suitable for offloading techniques. We find that residual compensation is highly memory-intensive: although it depends on large compensation matrices, only a small subset of parameters is accessed during actual computation. This behavior fundamentally differs from the compute-intensive nature of quantized weight computation, and is naturally suitable for CPU-GPU heterogeneous parallel inference. We further observe that different weight matrices exhibit different sensitivities to quantization error compensation. This suggests that dynamic rank allocation can achieve more effective accuracy recovery than static strategies under the same computation budget.

Based on these observations, we design and implement **HCInfer**, a **H**eterogeneous error-**C**ompensation **Infer**ence system, as shown in Figure 1(d). HCInfer builds a heterogeneous compensation pipeline that places quantized operators and residual compensation on the GPU and CPU, respectively, and achieves near-zero-overhead compensation through asynchronous parallel execution. In addition, HCInfer introduces a sensitivity-aware dynamic rank allocation algorithm that adaptively assigns compensation ranks to different matrices. This design enables the system to run quantized models with near full-precision accuracy on consumer-grade hardware while incurring almost no throughput loss.

In summary, this paper makes the following contributions:

- We propose and implement HCInfer, a novel heterogeneous inference system based on dynamic error compensation, making high-accuracy and high-throughput deployment of LLMs on consumer-grade hardware feasible.
- We develop a heterogeneous compensation pipeline. HCInfer fully exploits the memory-bound nature of residual compensation and achieves substantial temporal overlap between heterogeneous computation.

- We design a sensitivity-aware dynamic rank allocation algorithm. HCInfer allocates compensation resources according to multi-dimensional sensitivity indicators to achieve maximum compensation efficiency.
- Experiments on Qwen-30B-A3B [4] and Llama-3.1-8B [13] show that, HCInfer improves accuracy by 2.3% to 5.2% while preserving at least 85% of throughput compared with quantized baselines, and achieves up to 10.4× speedup compared with full-precision baselines.

2 Related Work

2.1 Quantization

Model quantization compresses LLMs by converting high-precision weights and activations into low-bit representations. Existing approaches mainly include post-training quantization (PTQ) and quantization-aware training (QAT). PTQ avoids retraining by calibrating on small datasets [5, 14, 15, 16], with methods improving accuracy via channel protection [6] and other methods. QAT incorporates quantization noise during training to achieve higher accuracy [7, 17, 18], with recent works improving efficiency through low-rank training [17], distillation [18], and block-wise optimization [7]. Despite these advances, aggressive low-bit quantization still leads to significant accuracy degradation.

2.2 Offloading

Parameter offloading enables large-model inference on limited hardware by dynamically transferring weights or states between GPU and external memory. Early systems [19, 20] focus on pure offloading, while later works introduce hybrid CPU-GPU execution, overlapping computation and I/O to improve efficiency [8, 9, 10]. For MoE models, prior methods further exploit expert sparsity via selective loading [21] and mixed-precision compression [22]. However, offloading performance is fundamentally limited by I/O bandwidth, restricting throughput in practical deployments.

2.3 CPU-Offloaded Error Compensation

DecDEC [23] compresses quantization residuals, stores them in host memory, and transfers selected residual blocks to the GPU, where they are applied to the quantized backbone during inference. HCInfer shares DecDEC's goal of recovering accuracy without keeping a full-precision model on the GPU, but adopts a different execution path, which shift the resource bottleneck from PCIe bandwidth and GPU correction toward otherwise underutilized CPU compute and memory bandwidth, motivating our heterogeneous pipeline and sensitivity-aware allocation policy.

2.4 LoRA-Style Quantization Error Compensation

Recent studies show that quantization errors exhibit low-rank structure, making LoRA-style adapters effective for error compensation [11, 12]. Prior works leverage low-rank approximation for fine-tuning or direct reconstruction, improving the accuracy of quantized models with varying training and overhead costs [24, 25, 26].

Inspired by these studies, we propose HCInfer, a deployment-oriented inference system that adopts LoRA-style compensation for quantization errors while further addressing the practical challenge of executing the compensation branch efficiently under memory-constrained heterogeneous hardware.

3 Observations

3.1 Non-Uniform Impact of Quantization Error Across Model Components

Existing studies have shown that quantization error matrices often exhibit strong low-rank structure [11, 12]. However, our analysis reveals that the usefulness of compensation varies substantially across different components of the model.

Singular-value distribution. We examine the singular value spectrum of quantization residual matrices from different operators in Qwen-30B-A3B, shown in Figure 2. While many matrices exhibit concentrated spectra, the degree of concentration differs significantly across operators. Matrices with more concentrated singular values should receive higher compensation priority for their higher intrinsic recoverability.

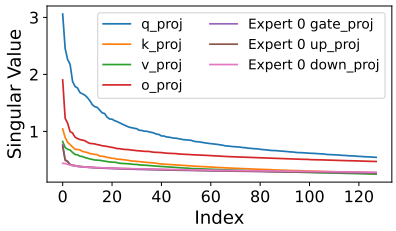


Figure 2: Singular value distribution of quantization error matrices for various modules in layer 0 of Qwen-30B-A3B, based on GPTQ Q4 quantization.

Output sensitivity. By sequentially replacing each weight matrix with its quantized counterpart and measuring the KL divergence between the outputs before and after the replacement, we observe in Figure 3 that quantization errors introduced at different decoder layers and projections affect final predictions unevenly. This suggests that compensation strategies should account not only for local matrix recoverability but also for layer-level structural sensitivity.

Expert weights in MoE models. For MoE architectures, since the output of an MoE block is a weighted aggregation of activated experts, the benefit of compensating a particular expert is naturally modulated by its routing weight. Experts with larger routing probabilities should receive more compensation resources.

These observations motivate the sensitivity-aware dynamic rank allocation strategy introduced in § 4.3, where compensation resources are allocated adaptively to maximize accuracy recovery.

3.2 Compute-Memory Characteristics of Quantized Weights and LoRA Compensation

We analyze the distinct computational properties of the quantized backbone and the low-rank compensation branch.

Quantized backbone is compute-bound. The primary advantage of low-bit quantization is that it significantly reduces model storage cost [5], thereby increasing arithmetic intensity during inference. As a result, GPUs are well suited for executing the quantized backbone path.

Low-rank compensation is memory-bound. Since the full pool of compensation parameters for all matrices in a model can approach the size of the original model [12], it is impractical to store them entirely in GPU VRAM. Despite the large model footprint, only a small subset of these parameters is activated during each inference step, rendering the compensation branch highly memory-bound with low arithmetic intensity. Given the significantly larger memory capacity of the CPU and the relatively low computational demand of the compensation task, offloading these operations to the CPU is both feasible and highly efficient.

Based on this observation, HCInfer decomposes model inference into GPU and CPU pathes, and design efficient heterogeneous inference pipeline in § 4.2.

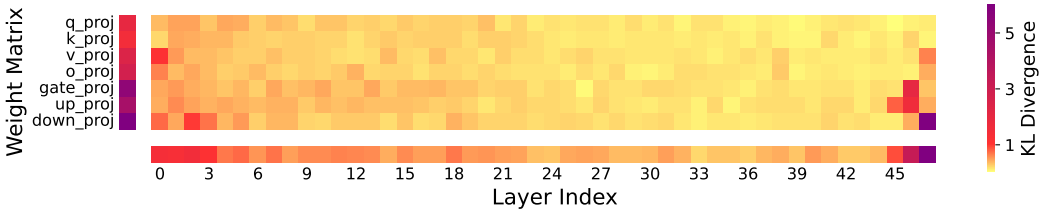


Figure 3: Matrix-wise output sensitivity analysis of Qwen-30B-A3B under GPTQ Q4 quantization. A higher KL divergence value indicates a greater discrepancy between the quantized model's output distribution and the original full-precision output.

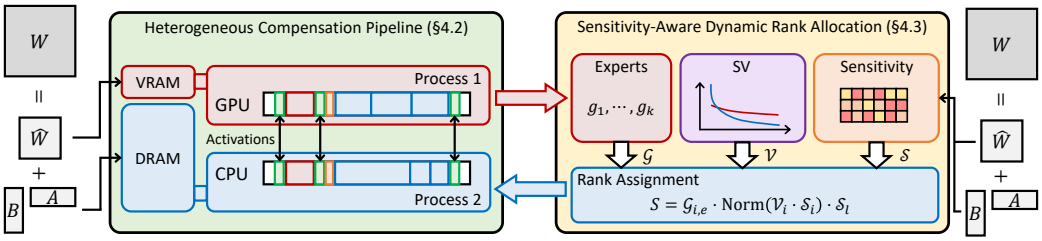


Figure 4: Overview of HCInfer.

4 HCInfer: Heterogeneous Error-Compensation Inference

In this section, we present HCInfer, an inference system that improves the inference accuracy of quantized large language models. HCInfer consists of two key components: a heterogeneous compensation pipeline, where the GPU executes the quantized backbone model while the CPU asynchronously computes the error compensation branch; a sensitivity-aware dynamic rank allocation strategy, which measures the sensitivity of different model components prior to deployment and allocates compensation ranks accordingly. Figure 4 illustrates the overall design of HCInfer.

4.1 Theoretical Framework

Consider a linear layer in an LLM with weight matrix $W \in \mathbb{R}^{d\times k}$ and input tensor $X \in \mathbb{R}^{n\times d}$. Its original full-precision computation is given by $Y = XW$. After low-bit quantization, the weight matrix becomes $\hat{W}$, which introduces a quantization error $\Delta W = W - \hat{W}$.

Based on the observations in § 3.1, the quantization error matrix ΔW exhibits strong low-rank structure, shown as:

$$Y = X\hat{W} + X\Delta W \approx X\hat{W} + (XA_r)B_r,$$

where $A_r \in \mathbb{R}^{d\times r}$ and $B_r \in \mathbb{R}^{r\times k}$ are low-rank factors obtained from the singular value decomposition of ΔW, and $r \ll \min(d, k)$ denotes the compensation rank.

4.2 Heterogeneous Compensation Pipeline

To make full use of computing resources on resource-constrained devices, HCInfer constructs an asynchronous inference pipeline through *workload decoupling* and *execution flow reorganization*. Detailed implementation of the pipeline is discribed in Appendix A.

4.2.1 Workload Decoupling

As discussed in § 3.2, the quantized backbone computation $\hat{Y} = X\hat{W}$ is a compute-bound task suited for the GPU, while the residual compensation $\Delta Y = X\Delta W = (XA)B$ branch's memory-bound nature makes it ideal for CPU offloading. Accordingly, HCInfer decomposes each linear layer into two parallel execution paths:

- **Backbone path, GPU-driven.** The quantized model resides entirely in GPU memory. During execution, the GPU directly invokes optimized low-bit matrix multiplication kernels to compute $\hat{Y} = X\hat{W}$.
- **Compensation path, CPU-driven.** The large LoRA parameter pool is stored in host DRAM. During execution, the GPU transfers intermediate activations to the CPU process and asynchronously triggers compensation computation. The CPU then applies the dynamic ranks and computes $\Delta Y = (XA_r)B_r$, after which the compensation result is sent back to the GPU.

To eliminate repeated process launch overhead, HCInfer adopts a persistent dual-process architecture. At initialization, two long-lived processes are created in parallel: a GPU process for quantized inference and a CPU process for compensation computation. These processes remain active throughout execution and communicate via shared memory and lightweight signaling primitives.

4.2.2 Execution Flow Reorganization

To enable shared compensation resources across multiple operators, HCInfer reorganizes the execution flow by merging communication stages and restructuring computation for MoE architectures.

Communication Fusion. HCInfer merges communication and calculation according to operator dependencies to enable cross-operator CPU sharing. The attention projection operators $Q(X), K(X), V(X)$ are mutually independent. Therefore, HCInfer logically groups these projections at the beginning of the layer. The GPU sends the input activation X only once, after which the CPU computes three compensation branches in parallel $\Delta Y_{Q,K,V} = (XA_{Q,K,V})B_{Q,K,V}$. The three compensation outputs are then returned to the GPU in a single synchronized transfer. For the feed-forward block, HCInfer applies the same strategy and fuses communication for the compensation branches associated with W_{up} and W_{gate}.

Cross-Expert Pipeline Reorganization. HCInfer restructures standard MoE inference workflow by grouping identical operators across all activated experts. After router execution on the GPU, the GPU and CPU jointly process all activated experts' W_{up} and W_{gate} operators in parallel. Once compensation results are synchronized, nonlinear activation is applied for all experts on the GPU. The system then jointly processes all W_{down} operators before aggregating expert outputs. The GPU transmits intermediate activations once at the start of the $W_{\text{up}}, W_{\text{gate}}$ stage and once at the start of the W_{down} stage. The CPU returns compensation outputs only once after each stage.

By grouping homogeneous operators, HCInfer allows multiple matrices to dynamically share CPU compensation capacity within the same execution window while reducing communication frequency.

4.3 Sensitivity-Aware Dynamic Rank Allocation

To maximize accuracy recovery under limited CPU compute throughput and memory bandwidth, HCInfer dynamically allocates rank budgets according to (i) the intrinsic recoverability of the quantization error, (ii) matrix- and layer-level output sensitivity, and (iii) expert activation for MoE models. Details of the algorithm are discribed in Appendix B.

4.3.1 Intrinsic Recoverability

For a quantization residual matrix $\Delta W_i \in \mathbb{R}^{d_i \times k_i}$, its singular value decomposition is $\Delta W_i = U_i \Sigma_i V_i^\top$, where $\Sigma_i = \text{diag}(\sigma_{i,1}, \sigma_{i,2}, \ldots, \sigma_{i,n_i})$, $\sigma_{i,1} \geq \sigma_{i,2} \geq \cdots \geq \sigma_{i,n_i} \geq 0$. We partition the singular values into a salient set S_i and a residual set R_i by maximizing difference in their variance.

We then define the singular value salience of matrix i as

$$\phi_i = \frac{\frac{1}{|S_i|}\sum_{\sigma \in S_i} \sigma}{\frac{1}{|R_i|}\sum_{\sigma \in R_i} \sigma},$$

Within each layer ℓ, these scores are normalized by layer to obtain relative recoverability score $\mathcal{V}_i = \text{Norm}(\phi_i)$.

4.3.2 Multi-Dimensional Output Sensitivity

Although V_i captures how efficiently a matrix can be approximated, it does not reflect the impact of that matrix on the final model output. We first measure the importance of each matrix by individually replacing it with its quantized counterpart and measure KL divergence between output distributions:

$$D_i = D_{\text{KL}}\big(P(Y|X) \,\|\, Q_i(Y|X)\big).$$

We then therefore introduce two complementary sensitivity measurements.

Matrix-Level Sensitivity. For each layer ℓ, we normalize the KL divergence among matrices within that layer as matrix-level sensitivity score $\mathcal{S}_i = \text{Norm}(D_i)$.

Layer-Level Sensitivity. Different transformer layers contribute unequally to final model quality. To capture this effect, we take a similar approach to matrix-level sensitivity measurement by replacing all matrices in layer ℓ with their quantized versions, and measure the KL divergence D_ℓ.

To emphasize highly sensitive layers within the limited compute budget, we identify the set of top-K most sensitive layers, denoted as $\mathcal{T}$, and derive the layer-level sensitivity score S_ℓ as follows:

$$\mathcal{S}_\ell = \begin{cases} 1, & \ell \in \mathcal{T}, \\ \frac{D_\ell}{\min_{m \in \mathcal{T}} D_m}, & \ell \notin \mathcal{T}. \end{cases}$$

4.3.3 Expert Activation for MoE Models

For MoE models, HCInfer further reallocates compensation ranks using routing weights. Suppose the top-k activated experts are indexed by $e \in \mathcal{A}$, with routing probabilities g_e. Computational resource is shared among the activated k experts, therefore the expert activation score $\mathcal{G}_{i,e}$ is scaled by $\mathcal{G}_{i,e} = k \cdot g_e$.

4.3.4 Priority Score and Rank Allocation

We combine all three factors into a unified compensation priority score $\mathcal{P}_i$. For matrix i in layer ℓ, the score is defined as

$$\mathcal{P}_i = \mathrm{Norm}(\mathcal{V}_i \cdot \mathcal{S}_i) \cdot \mathcal{S}_\ell.$$

For MoE models, the score is further scaled by activation score $\mathcal{G}_{i,e}$:

$$\mathcal{P}_{i,e} = \mathcal{G}_{i,e} \cdot \mathrm{Norm}(\mathcal{V}_i \cdot \mathcal{S}_i) \cdot \mathcal{S}_\ell.$$

Next, we determine a hardware-dependent standard rank r_{std}, defined as the maximum rank whose CPU computation and communication latency can still be hidden behind GPU execution:

$$r_{\mathrm{std}} = \max_r \left\{ T_{\mathrm{CPU}}(r) + T_{\mathrm{comm}} \leq T_{\mathrm{GPU}} \right\}.$$

Finally, the allocated compensation rank for matrix i is given by $r_i = \lfloor \mathcal{P}_i \cdot r_{\mathrm{std}} \rfloor$. Appendix B.2 explains the motivation and assumptions behind this policy.

5 Evaluation

5.1 Experimental Setup

Implementation: We implemented HCInfer based on the Hugging Face Transformers (v4.57.1) [27]. GPTQ [5] was selected as our fundamental quantization method.

Hardware Environment: Experiments were conducted on NVIDIA RTX 4090 with 24GB VRAM and Intel i9-10980XE with 256GB DRAM, interconnected via PCIe Gen3 x16.

Models and Quantization Configuration: We selected Qwen-30B-A3B [4], a large-scale MoE model, and Llama-3.1-8B [13], a small-to-medium scale dense model, for implementation. Our method was implemented based on a 4-bit and 3-bit quantized version respectively. For Llama-3.1-8B, we limited the available GPU VRAM to 8GB to simulate an extremely constrained environment.

Baselines: We compared HCInfer against two categories of mainstream inference deployment solutions:

1. **Pure Quantized Inference (GPU-only):** GPTQ [5] and llama.cpp [28] quantization. These methods run the quantized model exclusively on the GPU without any error compensation.
2. **Full-precision Offloading:** vLLM (v0.20.0) [29], llama.cpp (v0.3.20) [28], and Hugging Face Accelerate (v1.13.0) [19]. These methods offload part or all of the model parameters to CPU memory to run in full precision.

Datasets: We selected ARC-Easy, ARC-Challenge [30], MathQA [31], MMLU [32], WikiText2, and C4 [33] as the evaluation benchmarks.

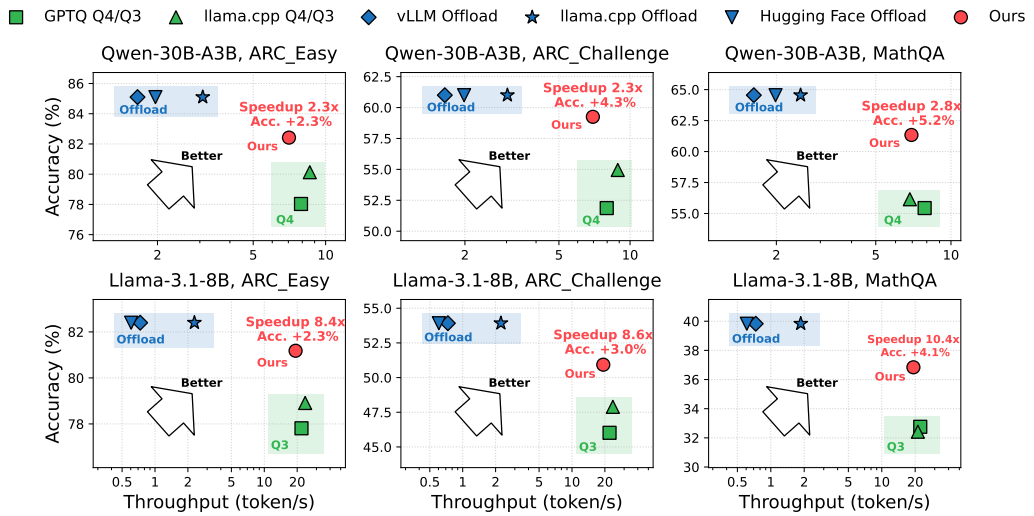


Figure 5: End-to-end throughput and accuracy comparison of HCInfer across models and datasets. The green section represents high throughput region with significant accuracy loss; the blue section represents high accuracy area with low inference efficiency.

5.2 End-to-End Performance

In Figure 5, we compare the throughput and accuracy of HCInfer against baseline methods across various datasets. In tests with the Qwen-30B-A3B model, HCInfer achieved significant accuracy improvements of $+2.3\%$, $+4.3\%$, and $+5.2\%$ on the ARC-Easy, ARC-Challenge, and MathQA tasks, respectively, compared to the best quantized baseline. Simultaneously, the throughput of HCInfer reached $2.3\times$ to $2.8\times$ that of the offloading schemes. In the memory-constrained scenario using the more aggressive 3-bit quantization for Llama-3.1-8B, HCInfer achieved up to a $10.4\times$ throughput improvement on MathQA, and $8.4\times$ and $8.6\times$ improvements on ARC-Easy and ARC-Challenge, respectively, while outperforming the quantized model's accuracy by 2.3% to 4.1%. These results demonstrate that HCInfer successfully enhance model accuracy while maintaining high throughput.

5.3 Fine-grained Analysis

In this section, we delve into the operational efficiency of HCInfer under different workloads and showcase its precision across multiple NLP tasks.

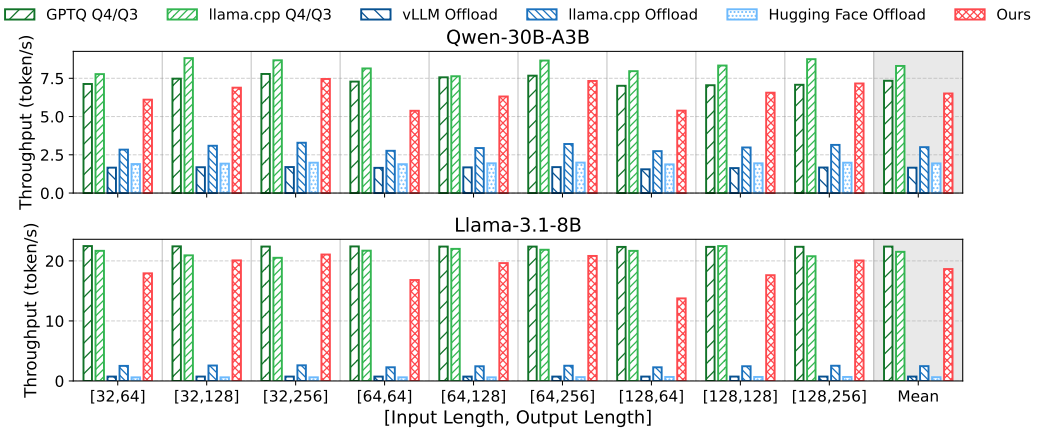


Figure 6: End-to-end performance comparison for various input and output length combinations.

Table 1: Perplexity and accuracy comparison across datasets. Best results are marked in bold.

Model	Method	WikiText2 (↓)	C4 (↓)	ARC-e (↑)	ARC-c (↑)	MathQA (↑)	MMLU (↑)
Qwen-30B-A3B	Full Precision	8.80	18.32	85.10%	61.00%	64.55%	81.93%
	GPTQ INT4	11.32	24.39	78.03%	51.87%	55.44%	76.40%
	Llama.cpp INT4	10.03	24.57	80.13%	54.95%	56.15%	76.85%
	HCInfer	**9.43**	**19.51**	**82.42%**	**59.25%**	**61.34%**	**79.92%**
Llama-3.1-8B	Full Precision	8.81	20.15	82.40%	53.92%	39.83%	67.90%
	GPTQ INT3	12.696	26.53	77.81%	46.02%	32.76%	64.19%
	Llama.cpp INT3	12.050	24.89	78.91%	48.89%	33.24%	65.08%
	HCInfer	**9.32**	**22.13**	**81.18%**	**50.93%**	**36.84%**	**66.74%**

Throughput across different sequence lengths: Figure 6 illustrates the throughput performance for various input and output length combinations. The results show that HCInfer exhibits a clear advantage over full-precision offloading: for Qwen-30B-A3B, throughput improved by $1.9\times$ to $2.2\times$; for Llama-3.1-8B, it improved by $6.0\times$ to $8.2\times$. Our method's advantage is more pronounced in scenarios with longer output lengths, as the CPU's limited parallel processing capability for multiple tokens makes prompt processing significantly less efficient than token generation.

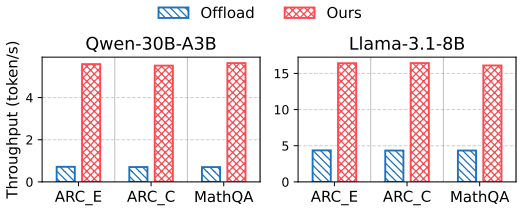


Figure 7: Throughput of HCInfer w/ and w/o heterogeneous pipeline. The baseline is a naive offloading scheme where all compensation weights are offloaded to CPU DRAM and moved to GPU only when accessed.

Accuracy and Perplexity: Table 1 details the comparison between HCInfer and other methods regarding language modeling perplexity and downstream task accuracy. Compared to quantized models, HCInfer significantly improves downstream accuracy: 2.2% to 5.2% for Qwen-30B-A3B and 1.7% to 3.6% for Llama-3.1-8B, approaching full-precision levels. For language modeling, HCInfer reduces perplexity on WikiText2 and C4 by 0.6 to 5.1 for Qwen-30B-A3B and 2.7 to 2.8 for Llama-3.1-8B.

5.4 Ablation Studies

We conducted two sets of ablation experiments to verify the core design of HCInfer.

Pipeline Ablation: To verify the contribution of heterogeneous parallel computing, we compared HCInfer with a naive compensation parameter offloading scheme. Figure 7 shows that HCInfer improves throughput by $3.7\times$ to $8.0\times$ over the naive version. HCInfer avoids slow parameter loading by transmitting only intermediate activation.

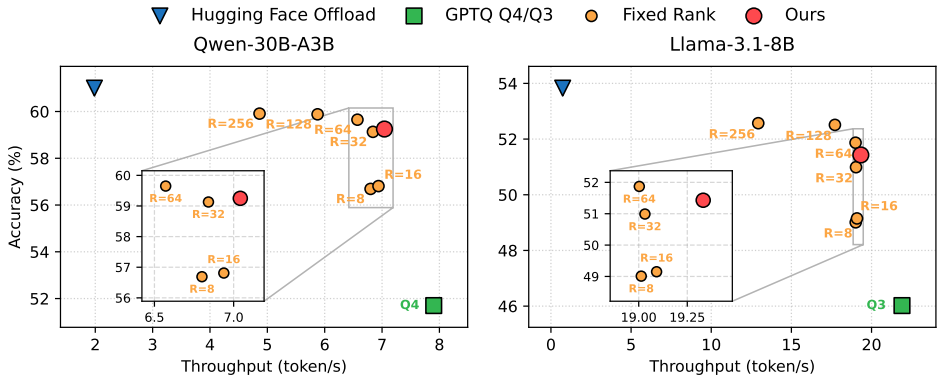


Figure 8: Throughput and accuracy comparison of HCInfer w/ and w/o dynamic rank allocator. $R = k$ refers to a fixed-rank allocation scheme where all matrices are assigned with rank-k compensation.

Parameter Allocation Ablation: We compared HCInfer against fixed-rank compensation schemes. Figure 8 shows that HCInfer's throughput exceeds all fixed-rank schemes while matching the accuracy of a rank-32 compensation. The throughput gains stem from dynamically skipping compensation for certain matrices, reducing communication and kernel loading overhead. Other experiments yielded similar results, which are listed in Appendix C.

6 Conclusion

In this paper, we introduced HCInfer, an efficient heterogeneous inference system that addresses the memory bottlenecks of LLMs on consumer-grade hardware. By integrating quantized base matrices on GPUs with low-rank residual compensation on CPUs, our method achieves high throughput, high accuracy inference. Comprehensive evaluations demonstrate that HCInfer achieves near full-precision accuracy while maintaining a peak speedup of x10.4 compared to full precision models. Future work will extend HCInfer to broader model architectures to further enhance deployment efficiency.

Acknowledgments and Disclosure of Funding

This work is supported in part by the NSFC under grant No. 62372265, No. 62332016, and No. U24B20152.

References

[1] Tianshi Zheng, Zheye Deng, Hong Ting Tsang, Weiqi Wang, Jiaxin Bai, Zihao Wang, and Yangqiu Song. From automation to autonomy: A survey on large language models in scientific discovery. In *Proceedings of the 2025 Conference on Empirical Methods in Natural Language Processing*, pages 17744–17761, 2025.

[2] Xinyi Li, Sai Wang, Siqi Zeng, Yu Wu, and Yi Yang. A survey on llm-based multi-agent systems: workflow, infrastructure, and challenges. *Vicinagearth*, 1(1):9, 2024.

[3] Jincai Huang, Yongjun Xu, Qi Wang, Qi Cheems Wang, Xingxing Liang, Fei Wang, Zhao Zhang, Wei Wei, Boxuan Zhang, Libo Huang, et al. Foundation models and intelligent decision-making: Progress, challenges, and perspectives. *The Innovation*, 6(6), 2025.

[4] An Yang, Anfeng Li, Baosong Yang, Beichen Zhang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Gao, Chengen Huang, Chenxu Lv, et al. Qwen3 technical report. *arXiv preprint arXiv:2505.09388*, 2025.

[5] 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.

[6] 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.

[7] 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.

[8] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Beidi Chen, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu. In *International Conference on Machine Learning*, pages 31094–31116. PMLR, 2023.

[9] Xuanlei Zhao, Bin Jia, Haotian Zhou, Ziming Liu, Shenggan Cheng, and Yang You. Hetegen: Efficient heterogeneous parallel inference for large language models on resource-constrained devices. *Proceedings of Machine Learning and Systems*, 6:162–172, 2024.

[10] Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. In *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles*, pages 590–606, 2024.

[11] Weibo Zhao, Yubin Shi, Xinyu Lyu, Wanchen Sui, Shen Li, and Yong Li. Aser: activation smoothing and error reconstruction for large language model quantization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 22822–22830, 2025.

[12] Shih-Yang Liu, Maksim Khadkevich, Nai Chit Fung, Charbel Sakr, Chao-Han Huck Yang, Chien-Yi Wang, Saurav Muralidharan, Hongxu Yin, Kwang-Ting Cheng, Jan Kautz, et al. Eora: Fine-tuning-free compensation for compressed llm with eigenspace low-rank approximation. *arXiv preprint arXiv:2410.21271*, 2024.

[13] 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.

[14] 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. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 39, pages 22299–22307, 2025.

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

[16] Fuwen Tan, Royson Lee, Shell Xu Hu, Sourav Bhattacharya, Timothy Hospedales, Georgios Tzimiropoulos, Brais Martinez, et al. Mobilequant: Mobile-friendly quantization for on-device language models. *arXiv preprint arXiv:2408.13933*, 2024.

[17] Wenjin Ke, Zhe Li, Dong Li, Lu Tian, and Emad Barsoum. Dl-qat: Weight-decomposed low-rank quantization-aware training for large language models. *arXiv preprint arXiv:2504.09223*, 2025.

[18] 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.

[19] Sylvain Gugger, Lysandre Debut, Thomas Wolf, Philipp Schmid, Zachary Mueller, Sourab Mangrulkar, Marc Sun, and Benjamin Bossan. Accelerate: Training and inference at scale made simple, efficient and adaptable. https://github.com/huggingface/accelerate, 2022.

[20] Reza Yazdani Aminabadi, Samyam Rajbhandari, Ammar Ahmad Awan, Cheng Li, Du Li, et al. Deepspeed-inference: enabling efficient inference of transformer models at unprecedented scale. In *SC22: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–15. IEEE, 2022.

[21] Keisuke Kamahori, Tian Tang, Yile Gu, Kan Zhu, and Baris Kasikci. Fiddler: Cpu-gpu orchestration for fast inference of mixture-of-experts models. *arXiv preprint arXiv:2402.07033*, 2024.

[22] 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.

[23] Yeonhong Park, Jake Hyun, Hojoon Kim, and Jae W Lee. Decdec: A systems approach to advancing low-bit llm quantization. In *19th USENIX Symposium on Operating Systems Design and Implementation (OSDI 25)*, pages 803–819, 2025.

[24] Han Guo, Philip Greengard, Eric P Xing, and Yoon Kim. Lq-lora: Low-rank plus quantized matrix decomposition for efficient language model finetuning. *arXiv preprint arXiv:2311.12023*, 2023.

[25] Yixiao Li, Yifan Yu, Chen Liang, Pengcheng He, Nikos Karampatziakis, Weizhu Chen, and Tuo Zhao. Loftq: Lora-fine-tuning-aware quantization for large language models. *arXiv preprint arXiv:2310.08659*, 2023.

[26] Rongzhi Zhang, Kuang Wang, Liyuan Liu, Shuohang Wang, Hao Cheng, Chao Zhang, and Yelong Shen. Lorc: Low-rank compression for llms kv cache with a progressive compression strategy. *arXiv preprint arXiv:2410.03111*, 2024.

[27] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 conference on empirical methods in natural language processing: system demonstrations*, pages 38–45, 2020.

[28] G. Gerganov and contributors. llama.cpp: Inference of llama model in pure c/c++. https://github.com/ggml-org/llama.cpp, 2025.

[29] 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.

[30] 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.

[31] Aida Amini, Saadia Gabriel, Shanchuan Lin, Rik Koncel-Kedziorski, Yejin Choi, and Hannaneh Hajishirzi. Mathqa: Towards interpretable math word problem solving with operation-based formalisms. 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 2357–2367, 2019.

[32] Dan Hendrycks, Collin Burns, Steven Basart, Andy Zou, Mantas Mazeika, Dawn Song, and Jacob Steinhardt. Measuring massive multitask language understanding. *arXiv preprint arXiv:2009.03300*, 2020.

[33] Jesse Dodge, Maarten Sap, Ana Marasović, William Agnew, Gabriel Ilharco, Dirk Groeneveld, Margaret Mitchell, and Matt Gardner. Documenting large webtext corpora: A case study on the colossal clean crawled corpus. In *Proceedings of the 2021 conference on empirical methods in natural language processing*, pages 1286–1305, 2021.

A Implementation Details and Algorithm Workflow

A.1 Implementation Details

A.1.1 Dual-Process Execution Architecture

HCInfer is implemented using a persistent dual-process architecture to enable efficient heterogeneous execution. At the beginning of inference, two long-lived processes are initialized:

- **GPU main process.** This process resides on the GPU and is responsible for executing the quantized backbone model, as well as orchestrating the overall inference workflow.
- **CPU worker process.** This process is forked from the main process but does not inherit the loaded model parameters. Instead, it independently loads the LoRA-style compensation parameters and is dedicated to executing residual compensation tasks when triggered.

The two processes communicate via shared memory using Python's `multiprocessing` primitives, including shared variables and event-based synchronization. This design avoids redundant data copies and minimizes communication overhead.

A.1.2 Asynchronous Communication and Execution

From the perspective of the GPU process, inference proceeds in a sequence of task blocks. When reaching a block that requires compensation, the GPU process performs the following steps:

1. Writes the current activation tensor into shared memory.
2. Signals the CPU process via an event to start compensation computation.
3. Immediately continues executing the quantized backbone operators.

After completing the corresponding backbone computation, the GPU process waits for the CPU process to finish, retrieves the compensated outputs from shared memory, and merges them with the current activations before proceeding.

From the CPU process perspective, it acts as a lightweight task executor. It continuously waits for task signals from the GPU process. Upon receiving a signal, it:

1. Reads the activation tensor from shared memory.
2. Determines the current execution context based on its locally maintained state.
3. Selects the corresponding subset of LoRA parameters according to the dynamic rank allocation results.
4. Computes the compensation outputs sequentially.
5. Writes the results back to shared memory and signals completion.

To reduce communication overhead, the CPU process maintains its own synchronized execution state (e.g., current layer and projection index), eliminating the need for the GPU process to repeatedly transmit meta-information. As a result, only activation tensors are exchanged during runtime.

A.1.3 Computation and Communication Co-Design

Naively, LoRA-style compensation operates at the granularity of individual weight matrices. A straightforward implementation would require launching a compensation task for each projection independently, leading to:

- Frequent activation transfers and synchronization events, introducing significant communication latency.
- Fine-grained CPU workloads with poor utilization.
- Strict per-matrix synchronization between GPU and CPU, preventing effective overlap.

Moreover, if the compensation computation of a particular projection exceeds the execution time of its corresponding quantized operator, the excess latency directly impacts the end-to-end inference time. This design also prevents different projections from flexibly sharing computation windows based on their importance.

To address these limitations, HCInfer introduces computation and communication fusion, which groups multiple operators into shared compensation windows.

Attention Blocks. The computation graph of attention can be expressed as:

$$X \to W_Q(X), W_K(X), W_V(X) \to \text{Attn}\left(\frac{QK^\top}{\sqrt{d}}\right) V \to W_O(\cdot).$$

Since the Q, K, and V projections are independent, HCInfer fuses them into a single compensation window, allowing them to share CPU computation and communication. The output projection W_O is processed in a separate window due to its dependency on the attention result.

Dense Feed-Forward Networks (FFN). The FFN computation is:

$$X \to W_{\text{up}}(X), W_{\text{gate}}(X) \to h = \sigma(W_{\text{gate}}(X)) \odot W_{\text{up}}(X) \to W_{\text{down}}(h).$$

Here, W_{up} and W_{gate} are independent and thus fused into a shared compensation window, while W_{down} is assigned a separate window.

MoE Layer. For MoE layers, the computation is:

$$X \to \text{Router}(X) \to \Big(W_{\text{up}}(X), W_{\text{gate}}(X) \to h \to W_{\text{down}}(h)\Big) \times k \to \sum_i w_i E_i(X).$$

In addition to projection-level independence, MoE introduces independence across experts. HCInfer further reorganizes execution by grouping operators across all activated experts: All experts' W_{up} and W_{gate} share a single compensation window, and all experts' W_{down} share another compensation window. This design enables cross-expert sharing of CPU computation windows, significantly improving resource utilization.

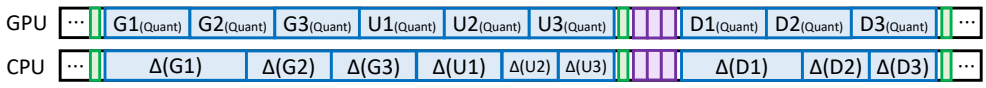
A.1.4 Computation Pipeline Diagram

We illustrate the compensation window sharing and execution pipeline in Figure 9. By fusing communication and computation across operators and experts, HCInfer establishes a structured heterogeneous execution pipeline that maximizes overlap between GPU and CPU.

Specifically, this design reduces synchronization frequency and communication overhead,enables effective overlap between GPU and CPU execution,and allows multiple matrices to share a unified compensation budget within the same execution window. Importantly, this pipeline organization maximizes the flexibility of dynamic rank allocation, allowing compensation resources to be adaptively distributed across matrices according to their importance while maintaining high system efficiency.



(a) Attention



(b) MoE

Figure 9: Detailed illustration of the heterogeneous execution pipeline. The purple blocks in (b) denote the non-linear activation functions for each expert. To reduce communication overhead, these activations are recomputed on the CPU rather than transferred from the GPU, leveraging their negligible computational cost.

A.2 End-to-End Algorithm Workflow

We summarize the end-to-end execution of HCInfer using two cooperating processes: a GPU process that executes the quantized backbone and controls the inference flow, and a CPU process that performs asynchronous LoRA-based compensation.

Algorithm 1 describes the GPU-side execution, including activation dispatch, backbone computation, and result merging. Algorithm 2 describes the CPU worker, which continuously fetches tasks, applies dynamic rank allocation, and computes compensation outputs. Together, these two processes form an asynchronous pipeline that overlaps computation and communication.

Algorithm 1: GPU Process: Quantized Backbone with Asynchronous Compensation

Input: Input tokens X
Shared memory: $\mathcal{M}$
Events: `cpu_start`, `cpu_done`
Initialize model state and layer index $\ell \leftarrow 1$
for each generation step **do**
  for each layer ℓ **do**
    Identify computation blocks (attention / FFN / MoE)
    for each compensation window w in layer ℓ **do**
      Extract activation A_w
      Write A_w to shared memory $\mathcal{M}$
      Set `cpu_start` event
      Compute $\hat{Y}_w$ using quantized operators on GPU
      Wait until `cpu_done` is triggered
      Read ΔY_w from shared memory $\mathcal{M}$
      $Y_w \leftarrow \hat{Y}_w + \Delta Y_w$
      Clear `cpu_done` event
    end for
    Update layer state $\ell \leftarrow \ell + 1$
  end for
end for
Return: Final output

Algorithm 2: CPU Process: LoRA-Based Compensation Worker

Shared memory: $\mathcal{M}$
Events: `cpu_start`, `cpu_done`
Load LoRA compensation parameters
Initialize local execution state (layer index, operator type)
while inference is running **do**
  Wait for `cpu_start` event
  Read activation A_w from shared memory $\mathcal{M}$
  Identify current layer ℓ and window type w
  Retrieve allocated ranks $\{r_i\}$ for matrices in window w
  $\Delta Y_w \leftarrow 0$
  for each matrix i in window w **do**
    Select LoRA parameters (A_i, B_i) with rank r_i
    $\Delta Y_w \leftarrow \Delta Y_w + (A_w A_i) B_i$
  end for
  Write ΔY_w to shared memory $\mathcal{M}$
  Update local execution state
  Clear `cpu_start` event
  Set `cpu_done` event
end while

B Theoretical Analysis of Dynamic Rank Allocation

B.1 Mathematical Formulation of the Dynamic Rank Allocator

The sensitivity-aware dynamic rank allocator in HCInfer integrates three complementary factors to determine the allocation priority for each weight matrix: (i) *intrinsic recoverability*, captured by the singular value salience score $\mathcal{V}_i$; (ii) *output sensitivity*, characterized by matrix- and layer-level

sensitivity scores $\mathcal{S}_i$ and $\mathcal{S}_\ell$; and (iii) *expert activation*, modeled by the expert-level score $\mathcal{G}_{i,e}$ for MoE architectures. We describe each component in detail below.

Singular Value Salience Score $\mathcal{V}_i$. We measure the intrinsic recoverability of a quantization residual matrix via the concentration of its singular value spectrum. For a given residual matrix $\Delta W_i \in \mathbb{R}^{d_i \times k_i}$, we perform singular value decomposition:

$$\Delta W_i = U_i \Sigma_i V_i^\top, \quad \Sigma_i = \mathrm{diag}(\sigma_{i,1}, \sigma_{i,2}, \ldots, \sigma_{i,n_i}),$$

where $\sigma_{i,1} \geq \sigma_{i,2} \geq \cdots \geq \sigma_{i,n_i} \geq 0$.

We first normalize the singular values:

$$\hat{\sigma}_{i,j} = \frac{\sigma_{i,j}}{\sigma_{i,1}}.$$

To identify the salient components, we compute the second-order discrete difference:

$$k_{i,j} = \hat{\sigma}_{i,j-1} - 2\hat{\sigma}_{i,j} + \hat{\sigma}_{i,j+1}.$$

Let

$$r_i = \arg\max_j k_{i,j}.$$

If $\max_j k_{i,j} > \tau$, where τ is a predefined threshold (set to 0.01 in our implementation), we partition the singular values into:

$$S_i = \{1, \ldots, r_i\}, \quad R_i = \{r_i + 1, \ldots, n_i\}.$$

Otherwise, we set $S_i = \emptyset$ and $R_i = \{1, \ldots, n_i\}$.

We then define the salience ratio:

$$\phi_i = \begin{cases} \frac{\frac{1}{|S_i|}\sum_{j \in S_i} \sigma_{i,j}}{\frac{1}{|R_i|}\sum_{j \in R_i} \sigma_{i,j}}, & S_i \neq \emptyset, \\ 1, & S_i = \emptyset. \end{cases}$$

Finally, we normalize ϕ_i within each compensation window $\mathcal{W}$ (as defined in Appendix A.1.3) to obtain:

$$\mathcal{V}_i = \frac{\phi_i}{\sum_{j \in \mathcal{W}} \phi_j}.$$

Matrix- and Layer-Level Sensitivity $\mathcal{S}_i, \mathcal{S}_\ell$. To quantify the impact of quantization on model outputs, we measure the divergence between the output distributions before and after replacing a matrix with its quantized version.

For a given matrix i, we define:

$$D_i = D_{\mathrm{KL}}\big(P(Y|X) \,\|\, Q_i(Y|X)\big),$$

where $P(Y|X)$ denotes the output distribution of the full-precision model, and $Q_i(Y|X)$ denotes the output distribution after replacing matrix i with its quantized counterpart. The KL divergence is computed as:

$$D_{\mathrm{KL}}(P\|Q) = \sum_y P(y|X) \log \frac{P(y|X)}{Q(y|X)}.$$

We normalize matrix-level sensitivity within each compensation window:

$$\mathcal{S}_i = \frac{D_i}{\sum_{j \in \mathcal{W}} D_j}.$$

For layer-level sensitivity, we replace all matrices in layer ℓ with their quantized versions and compute:

$$D_\ell = D_{\mathrm{KL}}\big(P(Y|X) \,\|\, Q_\ell(Y|X)\big).$$

To emphasize the most critical layers under limited computation budgets, we select the top-K layers with the largest D_ℓ, denoted as $\mathcal{T}$. The layer-level sensitivity score is defined as:

$$\mathcal{S}_\ell = \begin{cases} 1, & \ell \in \mathcal{T}, \\ \dfrac{D_\ell}{\min_{m \in \mathcal{T}} D_m}, & \ell \notin \mathcal{T}. \end{cases}$$

This formulation ensures that highly sensitive layers receive uniformly maximal priority, while less sensitive layers are proportionally down-weighted.

Expert Activation Score $\mathcal{G}_{i,e}$. For MoE architectures, the contribution of each expert is modulated by routing probabilities. Let $\mathcal{A}$ denote the set of activated experts, and g_e be the routing weight for expert $e \in \mathcal{A}$, satisfying:

$$\sum_{e \in \mathcal{A}} g_e = 1.$$

Given that $k = |\mathcal{A}|$ experts are activated, we define:

$$\mathcal{G}_{i,e} = k \cdot g_e.$$

This scaling redistributes the normalized routing probabilities across the activated experts, ensuring that the total contribution is consistent with the number of active experts.

Rank Allocation Strategy. Given the unified priority score, we allocate the compensation rank proportionally under a hardware-constrained budget. For matrix i (and expert e in MoE models), we first define the normalized priority:

$$\mathcal{P}_{i,e} = \mathcal{G}_{i,e} \cdot \mathrm{Norm}_{\mathcal{W}}(\mathcal{V}_i \cdot \mathcal{S}_i) \cdot \mathcal{S}_\ell,$$

where $\mathrm{Norm}_{\mathcal{W}}(\cdot)$ denotes normalization within the corresponding compensation window $\mathcal{W}$.

Let r_{std} denote the reference rank determined by the hardware constraint. The initial continuous rank allocation is given by:

$$\tilde{r}_{i,e} = \mathcal{P}_{i,e} \cdot r_{\mathrm{std}}.$$

Two-Stage Singular Value Allocation. Recall that the singular values of each residual matrix are partitioned into a salient set S_i and a residual set R_i. To maximize reconstruction effectiveness, we adopt a two-stage allocation strategy:

- **Stage 1 (Salient allocation).** Ranks are first assigned proportionally across matrices using only the singular values in S_i. That is, each matrix preferentially selects components from its salient subspace according to $\tilde{r}_{i,e}$.
- **Stage 2 (Residual allocation).** If the total allocated rank exceeds the total size of all salient sets, the remaining budget is further distributed proportionally over the residual sets R_i.

This design ensures that the most informative low-rank components are utilized first, improving the efficiency of limited rank budgets.

Discrete Rank Alignment. To facilitate efficient parallel execution on the CPU, we restrict the final rank values to a discrete set aligned with hardware-friendly sizes. Specifically, we constrain:

$$r_{i,e} \in \{0\} \cup \{2^k \mid k \geq k_0\},$$

where k_0 is chosen such that the minimum non-zero rank (e.g., $2^{k_0} = 8$ in our implementation) balances computational granularity and overhead.

The final rank is obtained by rounding $\tilde{r}_{i,e}$ to the nearest admissible value:

$$r_{i,e} = \mathrm{Align}(\tilde{r}_{i,e}),$$

where $\mathrm{Align}(\cdot)$ denotes projection onto the nearest allowed rank level.

This alignment improves SIMD efficiency and memory access patterns on the CPU, while introducing negligible distortion to the proportional allocation.

B.2 Motivation and Analysis of the Allocation Heuristic

This section motivates the proposed dynamic rank allocation strategy as a practical heuristic for distributing a hardware-constrained rank budget. The analysis explains why recoverability, output sensitivity, and expert activation are relevant proxies for marginal compensation utility.

Problem Formulation. The goal of the proposed inference system is to maximize the overall compensation effect under pipeline constraints. Let Y denote the model output, and ΔY denote the model compensation effect. Thus, the allocation problem can be formulated as:

$$\max_{\{r_i\}} \sum_{i \in \mathcal{W}} \Delta Y \quad \text{s.t.} \quad \sum_{i \in \mathcal{W}} T_{\text{CPU}}(r_i) + T_{\text{comm},\mathcal{W}} \leq T_{\text{GPU},\mathcal{W}}, \quad r_i \geq 0.$$

The constraints are imposed by the heterogeneous execution pipeline in Appendix A.1.3, where for each compensation window $\mathcal{W}$, the CPU computation and communication must be fully hidden behind GPU execution.

We start our analysis from the fundamental constraint imposed by the heterogeneous execution pipeline. Due to CPU's low computational parallelism, the CPU computation latency is approximately linear in the rank:

$$T_{\text{CPU}}(r_i) \approx k \cdot r_i,$$

where k is a hardware-dependent constant. We define the standard rank r_{std} as the maximum rank satisfying:

$$T_{\text{CPU}}(r_{\text{std}}) + T_{\text{comm}} = T_{\text{GPU}}.$$

Therefore, the pipeline constraint can be rewritten as:

$$\sum_{i \in \mathcal{W}} T_{\text{CPU}}(r_i) \leq T_{\text{CPU}}(r_{\text{std}}).$$

Since the total CPU workload is the sum of all compensation tasks, we have:

$$\sum_{i \in \mathcal{W}} T_{\text{CPU}}(r_i) \approx T_{\text{CPU}} \left(\sum_{i \in \mathcal{W}} r_i \right),$$

which leads to the equivalent constraint:

$$\sum_{i \in \mathcal{W}} r_i \leq r_{\text{std}}, \quad r_i \geq 0.$$

We then analyze the optimization objectives. Since compensation introduces a small perturbation, we approximate the improvement using a first-order expansion:

$$\Delta Y \approx \sum_{i \in \mathcal{W}} \frac{\partial Y}{\partial r_i} r_i.$$

Thus, the allocation problem can be formulated as:

$$\max_{\{r_i\}} \sum_{i \in \mathcal{W}} \frac{\partial Y}{\partial r_i} r_i \quad \text{s.t.} \quad \sum_{i \in \mathcal{W}} r_i \leq r_{\text{std}}, \quad r_i \geq 0.$$

This formulation reduces the dynamic rank allocation problem to a constrained resource allocation problem within each compensation window.

Optimal Allocation via Greedy Strategy. We define marginal gain as

$$g_i = \frac{\partial Y}{\partial r_i}.$$

g_i is non-increasing with respect to r_i, so the above problem is a classical resource allocation problem. Its optimal solution is to allocate resources in descending order of g_i, which is equivalent to a proportional allocation:

$$r_i \propto g_i.$$

This result can be obtained via a greedy argument or by observing that the problem reduces to a linear program whose optimal solution lies at an extreme point.

Approximation of $\partial Y/\partial r_i$. We now derive an explicit approximation of the marginal gain $\frac{\partial Y}{\partial r_i}$ that governs the optimal allocation.

Using the chain rule, we decompose:

$$\frac{\partial Y}{\partial r_i} = \frac{\partial Y}{\partial W_i} \cdot \frac{\partial W_i}{\partial r_i},$$

(1) Output Sensitivity Term. The first term measures how sensitive the model output is to perturbations in W_i.

We approximate this quantity using the KL divergence between the original and perturbed outputs. Let $P(Y|X)$ be the original output distribution and $Q_i(Y|X)$ be the distribution after perturbing W_i. Then:

$$D_i = D_{\mathrm{KL}}\big(P(Y|X) \,\|\, Q_i(Y|X)\big).$$

Using a first-order perturbation approximation, we have:

$$\left\| \frac{\partial Y}{\partial W_i} \right\| \;\propto\; D_i.$$

After normalization within a compensation window $\mathcal{W}$, this yields:

$$\left\| \frac{\partial Y}{\partial W_i} \right\| \;\approx\; \mathcal{S}_i.$$

(2) Weight Recoverability Term. The second term $\frac{\partial W_i}{\partial r_i}$ quantifies how much of the quantization error can be recovered as the rank increases.

Let ΔW_i denote the quantization residual with singular values $\{\sigma_{i,j}\}$. According to the Eckart–Young–Mirsky theorem, the optimal rank-r approximation satisfies:

$$\|\Delta W_i - (\Delta W_i)_r\|_F^2 = \sum_{j>r} \sigma_{i,j}^2.$$

Therefore, the marginal gain from increasing the rank by one is:

$$\frac{\partial}{\partial r} \|(\Delta W_i)_r\|_F^2 \;\approx\; \sigma_{i,r}^2.$$

This implies:

$$\left\| \frac{\partial W_i}{\partial r_i} \right\| \;\propto\; \sigma_{i,r}.$$

To obtain a stable and scale-invariant measure, we aggregate this effect using the singular value salience score:

$$\left\| \frac{\partial W_i}{\partial r_i} \right\| \;\approx\; \mathcal{V}_i.$$

(3) MoE Activation Term. For MoE layer, the output is given by:

$$Y_{\mathrm{MoE}} = \sum_{e \in \mathcal{A}} g_e \, E_e(X),$$

where g_e are routing weights and $\sum_e g_e = 1$.

Let $W_{i,e}$ denote the parameters of matrix i in expert e. Then:

$$\frac{\partial Y}{\partial W_{i,e}} = g_e \cdot \frac{\partial E_e(X)}{\partial W_{i,e}}.$$

Thus, the marginal gain is scaled by:

$$\left\| \frac{\partial Y}{\partial W_{i,e}} \right\| \;\propto\; g_e,$$

which corresponds to the expert activation score:

$$\mathcal{G}_{i,e} = k \cdot g_e.$$

Combined Expression. Combining all terms, we obtain:

$$\frac{\partial Y}{\partial r_{i,e}} \propto \mathcal{V}_i \cdot \mathcal{S}_i \cdot \mathcal{G}_{i,e}.$$

Substituting this into the optimization problem yields:

$$r_{i,e} \propto \mathcal{V}_i \cdot \mathcal{S}_i \cdot \mathcal{G}_{i,e},$$

which is exactly the allocation rule adopted in HCInfer.

C Extensive Experimental Analysis

This section consolidates the additional experiments conducted during the rebuttal. We evaluate compatibility with AWQ, compare against DecDEC across its accuracy–throughput trade-off, report exact end-to-end performance statistics, test cross-dataset sensitivity stability, and analyze hardware characteristics and generality. We then provide the original rank-allocation and pipeline profiling analyses.

C.1 Compatibility with AWQ at Different Bit Widths

We evaluate whether HCInfer composes with a stronger weight-quantization method by applying it on top of AWQ for Llama-3.1-8B. Experiments use an NVIDIA RTX 4090 GPU and an Intel Core i9-10980XE CPU. Table 2 reports GPU memory footprint, end-to-end dataset throughput under the original evaluation protocol, and ARC-Challenge accuracy.

At INT4, AWQ already retains most of the full-precision accuracy, leaving limited room for compensation: HCInfer improves accuracy by 0.49 percentage points. Under the more aggressive INT3 setting, HCInfer recovers 2.24 percentage points over standalone AWQ. Moving from an INT4 to an INT3 backbone reduces the GPU footprint from 5.41 GB to 4.12 GB, a 23.8% reduction, illustrating that the main benefit arises when GPU memory pressure motivates more aggressive quantization.

C.2 Comparison with DecDEC

We compare HCInfer with DecDEC using Llama-3.1-8B with an AWQ INT3 backbone on the same RTX 4090 and Intel Core i9-10980XE platform connected through PCIe Gen3 $\times$16. We sweep DecDEC's `target_slowdown` parameter rather than selecting only its default configuration. Table 3 reports the resulting accuracy–throughput trade-off. All throughput values in this table follow the original end-to-end dataset protocol and include prefill.

The initial comparison used `target_slowdown=80` to place DecDEC in a similar accuracy-recovery regime. The full sweep shows the trade-off more transparently. Around 50% accuracy, HCInfer is 19.3% faster than DecDEC at `target_slowdown=50` and 29.1% faster than DecDEC at `target_slowdown=80`. At a similar throughput regime, HCInfer reaches 50.06% accuracy, compared with 48.98% for DecDEC at `target_slowdown=20`. These results characterize the observed trade-off on the evaluated PCIe Gen3 platform; the absolute comparison can vary with PCIe bandwidth and CPU performance.

Table 2: Compatibility with AWQ on Llama-3.1-8B. Throughput follows the original end-to-end dataset protocol and includes prefill.

Method	Bit-width	GPU VRAM (GB)	Throughput (token/s)	Accuracy (%)
Full Precision	16	16.08	2.237	53.92
AWQ	INT4	5.41	24.665	52.23
HCInfer + AWQ	INT4	5.41	18.439	52.72
AWQ	INT3	4.12	20.591	47.82
HCInfer + AWQ	INT3	4.12	17.974	50.06

Table 3: Comparison with the DecDEC autotuner on Llama-3.1-8B with AWQ INT3. Accuracy is measured on ARC-Challenge.

Method/configuration	n_{tb}	k_{chunk}	Throughput (token/s)	Relative speed	Accuracy (%)
AWQ INT3 baseline	0	$[0, 0, 0, 0]$	20.591	$1.000\times$	47.82 $(+0.00)$
DecDEC, 2.5 (default)	24	$[2, 2, 12, 2]$	20.332	$0.987\times$	47.95 $(+0.13)$
DecDEC, 10	28	$[10, 8, 11, 10]$	19.535	$0.948\times$	48.64 $(+0.82)$
DecDEC, 20	24	$[14, 15, 15, 15]$	18.678	$0.907\times$	48.98 $(+1.16)$
DecDEC, 50	38	$[24, 24, 22, 23]$	15.072	$0.731\times$	50.48 $(+2.66)$
DecDEC, 80	38	$[30, 31, 27, 27]$	13.920	$0.676\times$	51.25 $(+3.43)$
HCInfer	–	Dynamic ranks	17.974	$0.873\times$	50.06 $(+2.24)$

Table 4: Exact end-to-end dataset throughput for Llama-3.1-8B. Parentheses show speed normalized to GPTQ INT3.

Dataset	Full Precision	GPTQ INT3	HCInfer	Slowdown
ARC-Easy	3.084 $(0.14\times)$	21.899 $(1.00\times)$	19.332 $(0.88\times)$	12%
ARC-Challenge	2.237 $(0.10\times)$	22.072 $(1.00\times)$	19.335 $(0.88\times)$	12%
MathQA	1.861 $(0.08\times)$	22.223 $(1.00\times)$	19.277 $(0.87\times)$	13%

C.3 Detailed End-to-End Throughput

Table 4 provides the exact throughput values underlying the aggregate slowdown reported in the original evaluation. The measurements use a single-batch, weight-only setting and the original end-to-end dataset protocol.

C.4 Cross-Dataset Stability of Sensitivity Scores

We examine whether the calibrated sensitivity scores primarily reflect model structure or a particular calibration distribution. For Qwen-30B-A3B, we independently compute matrix- and layer-level sensitivity using WikiText-2 and ARC-Challenge. The datasets have substantially different input distributions.

The broad ordering is stable across both datasets: `down_proj` and `gate_proj` are substantially more sensitive than attention projections, while the first and final layers are more sensitive than the sampled intermediate layers. The absolute scores, particularly for `up_proj`, still vary across datasets; we therefore interpret this experiment as evidence of stability on these two calibration distributions rather than universal dataset independence.

C.5 Roofline Analysis and Hardware Generality

We use `torch.profiler` to characterize the quantized backbone and low-rank compensation on the RTX 4090 and Intel Core i9-10980XE platform. The measurements are summarized in Table 6.

The GPU ridge point is computed from 82.6 TFLOP/s peak compute and 1.01 TB/s memory bandwidth. The CPU ridge point uses 2.764 TFLOP/s and 157.6 GB/s. The quantized backbone lies near the GPU ridge point, whereas the compensation intensity is below the CPU ridge point and is therefore memory-bound under this model. This contrast motivates mapping the two branches to different devices.

We additionally evaluate Llama-3.1-8B with AWQ INT3 on an RTX 3090 GPU and an Intel Xeon Gold 5218R CPU. Table 7 reports the results using the original end-to-end dataset throughput protocol.

On this platform, HCInfer improves accuracy by 4.58 percentage points over standalone AWQ INT3 while retaining substantially higher throughput than full-precision offloading. A slower GPU backbone also provides a wider overlap window for CPU compensation. This experiment supports operation on a second consumer-GPU platform, but does not imply that full overlap is guaranteed on arbitrary hardware.

Table 5: Sensitivity scores for Qwen-30B-A3B calibrated on two datasets.

Category	Component	WikiText-2	ARC-Challenge
Matrix type	`q_proj`	1.961	1.367
	`k_proj`	1.484	1.729
	`v_proj`	2.437	2.255
	`o_proj`	2.781	2.841
	`gate_proj`	5.375	5.222
	`up_proj`	4.437	2.519
	`down_proj`	49.751	48.252
Layer	Layer 0	1.406	1.335
	Layer 11	0.628	0.601
	Layer 23	0.318	0.292
	Layer 35	0.287	0.249
	Layer 47	39.750	35.859

Table 6: Arithmetic-intensity analysis of the two execution branches.

Component	Work (GFLOPs)	Referenced data (GB)	Intensity (FLOP/Byte)	Ridge point
Quantized backbone (GPU)	384.0	4.7	81.70	81.78
Low-rank compensation (CPU)	31.9	3.8	8.41	17.53

The latency terms used by the allocator are obtained through offline profiling before deployment. A hardware-specific script benchmarks the GPU backbone, activation transfer, and CPU low-rank kernels across candidate ranks. Recoverability and output-sensitivity scores are also computed offline; only MoE routing weights are obtained from the normal forward pass at runtime.

C.6 Activation-Transfer Latency across Compensation Ranks

HCInfer transfers intermediate activations rather than rank-dependent residual weights. Across four representative candidate-rank configurations, the measured activation-transfer latencies are 3.19, 3.10, 3.21, and 3.13 ms, respectively, with a mean of approximately 3.16 ms. The small variation supports the expectation that activation-transfer volume is determined by batch size and sequence length rather than compensation rank.

C.7 Rank Allocation Profiles across Model Layers

To better understand the effectiveness of the proposed allocator, we analyze the allocation profiles produced by HCInfer and evaluate nearby alternative allocation strategies.

Specifically, starting from the allocation generated by our sensitivity-aware allocator, we construct multiple perturbed allocation schemes by redistributing compensation ranks across different layers and operator groups while keeping the overall rank budget unchanged. We then measure both model accuracy and inference throughput for each configuration.

Table 8 summarizes the results. We observe that the allocation produced by HCInfer consistently achieves a superior balance between accuracy and throughput compared with neighboring allocation strategies. In particular, manually shifting compensation ranks toward less sensitive layers or

Table 7: Results on a second hardware platform: RTX 3090 and Intel Xeon Gold 5218R.

Method	Bit-width	Throughput (token/s)	Accuracy (%)
Full Precision	16	2.129	53.92
AWQ	INT3	17.487	46.69
HCInfer + AWQ	INT3	14.952	51.27

Table 8: Accuracy and throughput under different rank allocation strategies. The first row corresponds to the reference allocation produced by HCInfer. Values in parentheses denote the relative change compared with HCInfer in terms of accuracy and throughput, respectively.

Rank Distribution	Accuracy	Throughput (token/s)
(0, 0, 64, 32, 32, 16, 32)	59.25% (+0.0%)	6.960 (+0.0%)
(32, 0, 32, 32, 32, 16, 32)	55.04% (-4.2%)	6.935 (+0.7%)
(0, 32, 32, 32, 32, 16, 32)	55.11% (-4.1%)	6.867 (-0.2%)
(32, 32, 32, 32, 32, 16, 32)	56.88% (-2.3%)	6.963 (+1.1%)
(0, 0, 128, 32, 32, 16, 32)	59.78% (+0.5%)	6.574 (-4.4%)
(64, 0, 64, 32, 32, 16, 32)	58.93% (-0.3%)	6.523 (-5.2%)
(0, 64, 64, 32, 32, 16, 32)	60.02% (+0.7%)	6.491 (-5.6%)
(0, 0, 32, 32, 32, 16, 32)	55.12% (-4.1%)	6.890 (+0.0%)
(16, 0, 16, 32, 32, 16, 32)	56.91% (-2.3%)	6.883 (-0.0%)
(0, 16, 16, 32, 32, 16, 32)	55.04% (-4.2%)	6.851 (-0.4%)
(0, 0, 64, 16, 32, 16, 32)	54.12% (-5.1%)	6.963 (+1.1%)
(0, 0, 64, 8, 32, 16, 32)	53.14% (-6.1%)	6.890 (+0.0%)
(0, 0, 64, 64, 32, 16, 32)	59.71% (+0.4%)	6.731 (-2.2%)
(0, 0, 64, 128, 32, 16, 32)	59.25% (+0.0%)	6.436 (-6.4%)
(0, 0, 64, 32, 16, 32, 32)	55.87% (-3.3%)	6.951 (+0.9%)
(0, 0, 64, 32, 16, 16, 32)	54.19% (-5.0%)	6.960 (+1.1%)
(0, 0, 64, 32, 16, 8, 32)	54.12% (-5.1%)	7.024 (+2.0%)
(0, 0, 64, 32, 64, 16, 32)	59.25% (+0.0%)	6.930 (-0.6%)
(0, 0, 64, 32, 32, 32, 32)	59.71% (+0.4%)	6.889 (-0.0%)
(0, 0, 64, 32, 64, 32, 32)	60.12% (+0.8%)	6.848 (-0.5%)
(0, 0, 64, 32, 32, 16, 16)	54.12% (-5.1%)	6.910 (+0.3%)
(0, 0, 64, 32, 32, 16, 8)	55.87% (-3.3%)	6.861 (-0.3%)
(0, 0, 64, 32, 32, 16, 64)	59.71% (+0.4%)	6.849 (-0.4%)
(0, 0, 64, 32, 32, 16, 128)	59.71% (+0.4%)	6.489 (-5.7%)

Table 9: Latency breakdown during autoregressive decoding.

Type	Total (ms)	CPU (ms)	GPU (ms)	Comm. (ms)	Overlap (ms)
Iteration	128.33	67.99	87.04	3.10	57.54
Attention QKV	13.15	2.74	3.05	1.01	2.39
Attention O	4.69	1.22	1.36	0.33	1.06
FFN/MoE Gate/Up	68.07	66.39	57.28	1.08	56.67
FFN/MoE Down	32.13	31.22	25.35	0.68	25.12

operator groups generally leads to noticeable accuracy degradation while providing limited throughput improvement.

These results validate the core design principle of our allocator: compensation resources should be preferentially assigned to matrices with high marginal utility, rather than distributed uniformly across the model.

Furthermore, we observe that the performance landscape around the optimal allocation is relatively smooth, indicating that the proposed strategy is robust to small perturbations. This behavior suggests that the derived allocation scores effectively capture the relative importance of different matrices and layers.

C.8 Computational Latency Breakdown and Profiling

To analyze the runtime behavior of HCInfer, we perform detailed profiling of the heterogeneous execution pipeline during autoregressive decoding.

We first measure the average per-token and per-window latency breakdown across the entire inference process, including: (i) total CPU compensation time, (ii) total GPU backbone computation time, (iii) CPU–GPU communication overhead, and (iv) effective overlap time between CPU and GPU execution. The results are summarized in Table 9.

We observe that a substantial fraction of the CPU-side compensation latency is successfully hidden behind GPU execution, validating the effectiveness of the asynchronous pipeline design.

Moreover, the communication overhead accounts for only a small portion of the total runtime, demonstrating that the proposed window-level communication fusion effectively reduces synchronization and data transfer costs.

From per-window breakdown experiment, we observe that different operator groups exhibit distinct latency characteristics. In particular, attention projection windows generally achieve higher overlap efficiency due to their larger GPU execution regions, while certain FFN or MoE windows become more sensitive to CPU-side workload imbalance.

These observations further motivate the importance of dynamic rank allocation and window-level scheduling in maximizing hardware utilization.

Finally, we observe that the overlap ratio remains consistently high across most compensation windows, indicating that the proposed pipeline organization successfully converts otherwise exposed CPU computation into effectively hidden latency. This behavior is critical for maintaining high throughput while enabling adaptive low-rank compensation.

In Table 10, we present the detailed profiling results.

Table 10: Detailed latency breakdown by layer.

Layer	Window	Total (ms)	CPU (ms)	GPU (ms)	Comm. (ms)	Overlap (ms)
layer0	Attention QKV	0.37	0.29	0.30	0.06	0.25
layer0	Attention O	0.12	0.08	0.09	0.03	0.07
layer0	FFN/MoE Gate/Up	1.54	0.22	0.24	1.31	0.20
layer0	FFN/MoE Down	0.75	0.18	0.18	0.57	0.16
layer4	Attention QKV	0.27	0.21	0.21	0.06	0.18
layer4	Attention O	0.10	0.06	0.07	0.03	0.05
layer4	FFN/MoE Gate/Up	1.38	0.20	0.22	1.15	0.18
layer4	FFN/MoE Down	0.65	0.14	0.14	0.51	0.12
layer8	Attention QKV	0.25	0.17	0.19	0.06	0.15
layer8	Attention O	0.09	0.06	0.07	0.03	0.06
layer8	FFN/MoE Gate/Up	1.33	0.19	0.20	1.13	0.17
layer8	FFN/MoE Down	0.63	0.12	0.12	0.51	0.11
layer12	Attention QKV	0.26	0.20	0.20	0.06	0.17
layer12	Attention O	0.09	0.06	0.07	0.03	0.06
layer12	FFN/MoE Gate/Up	1.35	0.22	0.23	1.12	0.20
layer12	FFN/MoE Down	0.65	0.14	0.15	0.50	0.13
layer16	Attention QKV	0.28	0.20	0.21	0.07	0.18
layer16	Attention O	0.10	0.07	0.07	0.03	0.06
layer16	FFN/MoE Gate/Up	1.38	0.11	0.11	1.27	0.10
layer16	FFN/MoE Down	0.67	0.10	0.11	0.56	0.09
layer20	Attention QKV	0.25	0.18	0.18	0.06	0.16
layer20	Attention O	0.09	0.06	0.07	0.03	0.05
layer20	FFN/MoE Gate/Up	1.34	0.15	0.16	1.18	0.13
layer20	FFN/MoE Down	0.64	0.10	0.11	0.54	0.09
layer24	Attention QKV	0.25	0.17	0.18	0.06	0.15
layer24	Attention O	0.09	0.06	0.06	0.03	0.05
layer24	FFN/MoE Gate/Up	1.31	0.13	0.14	1.18	0.11
layer24	FFN/MoE Down	0.78	0.27	0.27	0.51	0.24
layer28	Attention QKV	0.27	0.19	0.20	0.07	0.17
layer28	Attention O	0.10	0.07	0.07	0.03	0.06
layer28	FFN/MoE Gate/Up	1.46	0.28	0.29	1.17	0.24
layer28	FFN/MoE Down	0.71	0.18	0.18	0.53	0.16
layer32	Attention QKV	0.27	0.20	0.21	0.06	0.17
layer32	Attention O	0.10	0.07	0.07	0.03	0.06
layer32	FFN/MoE Gate/Up	1.33	0.17	0.17	1.16	0.15
layer32	FFN/MoE Down	0.64	0.12	0.14	0.51	0.11
layer36	Attention QKV	0.25	0.18	0.19	0.06	0.16
layer36	Attention O	0.09	0.06	0.06	0.03	0.06
layer36	FFN/MoE Gate/Up	1.32	0.18	0.19	1.13	0.16
layer36	FFN/MoE Down	0.64	0.13	0.14	0.50	0.11
layer40	Attention QKV	0.25	0.18	0.19	0.06	0.16
layer40	Attention O	0.09	0.06	0.07	0.03	0.05
layer40	FFN/MoE Gate/Up	1.34	0.19	0.21	1.14	0.17
layer40	FFN/MoE Down	0.64	0.13	0.13	0.51	0.11
layer44	Attention QKV	0.25	0.19	0.19	0.06	0.16
layer44	Attention O	0.09	0.06	0.07	0.03	0.06
layer44	FFN/MoE Gate/Up	1.33	0.20	0.20	1.13	0.17
layer44	FFN/MoE Down	0.65	0.13	0.14	0.51	0.11

NeurIPS Paper Checklist

1. **Claims**

   Question: Do the main claims made in the abstract and introduction accurately reflect the paper's contributions and scope?

   Answer: [Yes]

   Justification: All claims are supported by the experimental results.

   Guidelines:

   - The answer [N/A] means that the abstract and introduction do not include the claims made in the paper.
   - The abstract and/or introduction should clearly state the claims made, including the contributions made in the paper and important assumptions and limitations. A [No] or [N/A] answer to this question will not be perceived well by the reviewers.
   - The claims made should match theoretical and experimental results, and reflect how much the results can be expected to generalize to other settings.
   - It is fine to include aspirational goals as motivation as long as it is clear that these goals are not attained by the paper.

2. **Limitations**

   Question: Does the paper discuss the limitations of the work performed by the authors?

   Answer: [Yes]

   Justification: Limitations are discussed in § 6.

   Guidelines:

   - The answer [N/A] means that the paper has no limitation while the answer [No] means that the paper has limitations, but those are not discussed in the paper.
   - The authors are encouraged to create a separate "Limitations" section in their paper.
   - The paper should point out any strong assumptions and how robust the results are to violations of these assumptions (e.g., independence assumptions, noiseless settings, model well-specification, asymptotic approximations only holding locally). The authors should reflect on how these assumptions might be violated in practice and what the implications would be.
   - The authors should reflect on the scope of the claims made, e.g., if the approach was only tested on a few datasets or with a few runs. In general, empirical results often depend on implicit assumptions, which should be articulated.
   - The authors should reflect on the factors that influence the performance of the approach. For example, a facial recognition algorithm may perform poorly when image resolution is low or images are taken in low lighting. Or a speech-to-text system might not be used reliably to provide closed captions for online lectures because it fails to handle technical jargon.
   - The authors should discuss the computational efficiency of the proposed algorithms and how they scale with dataset size.
   - If applicable, the authors should discuss possible limitations of their approach to address problems of privacy and fairness.
   - While the authors might fear that complete honesty about limitations might be used by reviewers as grounds for rejection, a worse outcome might be that reviewers discover limitations that aren't acknowledged in the paper. The authors should use their best judgment and recognize that individual actions in favor of transparency play an important role in developing norms that preserve the integrity of the community. Reviewers will be specifically instructed to not penalize honesty concerning limitations.

3. **Theory assumptions and proofs**

   Question: For each theoretical result, does the paper provide the full set of assumptions and a complete (and correct) proof?

   Answer: [Yes]

Justification: Proof of theoretical results are given in Appendix B.2.

Guidelines:

- The answer [N/A] means that the paper does not include theoretical results.
- All the theorems, formulas, and proofs in the paper should be numbered and cross-referenced.
- All assumptions should be clearly stated or referenced in the statement of any theorems.
- The proofs can either appear in the main paper or the supplemental material, but if they appear in the supplemental material, the authors are encouraged to provide a short proof sketch to provide intuition.
- Inversely, any informal proof provided in the core of the paper should be complemented by formal proofs provided in appendix or supplemental material.
- Theorems and Lemmas that the proof relies upon should be properly referenced.

4. **Experimental result reproducibility**

   Question: Does the paper fully disclose all the information needed to reproduce the main experimental results of the paper to the extent that it affects the main claims and/or conclusions of the paper (regardless of whether the code and data are provided or not)?

   Answer: [Yes]

   Justification: The details of the implementation of our method is given in Appendix A. All results are reproducible.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - If the paper includes experiments, a [No] answer to this question will not be perceived well by the reviewers: Making the paper reproducible is important, regardless of whether the code and data are provided or not.
   - If the contribution is a dataset and/or model, the authors should describe the steps taken to make their results reproducible or verifiable.
   - Depending on the contribution, reproducibility can be accomplished in various ways. For example, if the contribution is a novel architecture, describing the architecture fully might suffice, or if the contribution is a specific model and empirical evaluation, it may be necessary to either make it possible for others to replicate the model with the same dataset, or provide access to the model. In general. releasing code and data is often one good way to accomplish this, but reproducibility can also be provided via detailed instructions for how to replicate the results, access to a hosted model (e.g., in the case of a large language model), releasing of a model checkpoint, or other means that are appropriate to the research performed.
   - While NeurIPS does not require releasing code, the conference does require all submissions to provide some reasonable avenue for reproducibility, which may depend on the nature of the contribution. For example
     (a) If the contribution is primarily a new algorithm, the paper should make it clear how to reproduce that algorithm.
     (b) If the contribution is primarily a new model architecture, the paper should describe the architecture clearly and fully.
     (c) If the contribution is a new model (e.g., a large language model), then there should either be a way to access this model for reproducing the results or a way to reproduce the model (e.g., with an open-source dataset or instructions for how to construct the dataset).
     (d) We recognize that reproducibility may be tricky in some cases, in which case authors are welcome to describe the particular way they provide for reproducibility. In the case of closed-source models, it may be that access to the model is limited in some way (e.g., to registered users), but it should be possible for other researchers to have some path to reproducing or verifying the results.

5. **Open access to data and code**

   Question: Does the paper provide open access to the data and code, with sufficient instructions to faithfully reproduce the main experimental results, as described in supplemental material?

Answer: [No]

Justification: The source code is currently subject to internal lab management policies regarding intellectual property review. We are committed to releasing the code and data upon paper acceptance. We have provided comprehensive pseudocode and detailed algorithm descriptions in Appendix A to allow for independent verification and implementation.

Guidelines:

- The answer [N/A] means that paper does not include experiments requiring code.
- Please see the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- While we encourage the release of code and data, we understand that this might not be possible, so [No] is an acceptable answer. Papers cannot be rejected simply for not including code, unless this is central to the contribution (e.g., for a new open-source benchmark).
- The instructions should contain the exact command and environment needed to run to reproduce the results. See the NeurIPS code and data submission guidelines (`https://neurips.cc/public/guides/CodeSubmissionPolicy`) for more details.
- The authors should provide instructions on data access and preparation, including how to access the raw data, preprocessed data, intermediate data, and generated data, etc.
- The authors should provide scripts to reproduce all experimental results for the new proposed method and baselines. If only a subset of experiments are reproducible, they should state which ones are omitted from the script and why.
- At submission time, to preserve anonymity, the authors should release anonymized versions (if applicable).
- Providing as much information as possible in supplemental material (appended to the paper) is recommended, but including URLs to data and code is permitted.

6. **Experimental setting/details**

   Question: Does the paper specify all the training and test details (e.g., data splits, hyperparameters, how they were chosen, type of optimizer) necessary to understand the results?

   Answer: [Yes]

   Justification: All substantial details are provided in the paper.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The experimental setting should be presented in the core of the paper to a level of detail that is necessary to appreciate the results and make sense of them.
   - The full details can be provided either with the code, in appendix, or as supplemental material.

7. **Experiment statistical significance**

   Question: Does the paper report error bars suitably and correctly defined or other appropriate information about the statistical significance of the experiments?

   Answer: [No]

   Justification: The runs for the experiments in the paper have low variance.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The authors should answer [Yes] if the results are accompanied by error bars, confidence intervals, or statistical significance tests, at least for the experiments that support the main claims of the paper.
   - The factors of variability that the error bars are capturing should be clearly stated (for example, train/test split, initialization, random drawing of some parameter, or overall run with given experimental conditions).
   - The method for calculating the error bars should be explained (closed form formula, call to a library function, bootstrap, etc.)
   - The assumptions made should be given (e.g., Normally distributed errors).

- It should be clear whether the error bar is the standard deviation or the standard error of the mean.
- It is OK to report 1-sigma error bars, but one should state it. The authors should preferably report a 2-sigma error bar than state that they have a 96% CI, if the hypothesis of Normality of errors is not verified.
- For asymmetric distributions, the authors should be careful not to show in tables or figures symmetric error bars that would yield results that are out of range (e.g., negative error rates).
- If error bars are reported in tables or plots, the authors should explain in the text how they were calculated and reference the corresponding figures or tables in the text.

8. **Experiments compute resources**

   Question: For each experiment, does the paper provide sufficient information on the computer resources (type of compute workers, memory, time of execution) needed to reproduce the experiments?

   Answer: [Yes]

   Justification: All experiment setup details are listed in § 5.1.

   Guidelines:

   - The answer [N/A] means that the paper does not include experiments.
   - The paper should indicate the type of compute workers CPU or GPU, internal cluster, or cloud provider, including relevant memory and storage.
   - The paper should provide the amount of compute required for each of the individual experimental runs as well as estimate the total compute.
   - The paper should disclose whether the full research project required more compute than the experiments reported in the paper (e.g., preliminary or failed experiments that didn't make it into the paper).

9. **Code of ethics**

   Question: Does the research conducted in the paper conform, in every respect, with the NeurIPS Code of Ethics `https://neurips.cc/public/EthicsGuidelines`?

   Answer: [Yes]

   Justification: Yes, we confirm adherence to the NeurIPS Code of Ethics.

   Guidelines:

   - The answer [N/A] means that the authors have not reviewed the NeurIPS Code of Ethics.
   - If the authors answer [No], they should explain the special circumstances that require a deviation from the Code of Ethics.
   - The authors should make sure to preserve anonymity (e.g., if there is a special consideration due to laws or regulations in their jurisdiction).

10. **Broader impacts**

    Question: Does the paper discuss both potential positive societal impacts and negative societal impacts of the work performed?

    Answer: [N/A]

    Justification: Our work is focused on efficient LLM inference. We do not foresee any particular societal impacts from this work.

    Guidelines:

    - The answer [N/A] means that there is no societal impact of the work performed.
    - If the authors answer [N/A] or [No], they should explain why their work has no societal impact or why the paper does not address societal impact.
    - Examples of negative societal impacts include potential malicious or unintended uses (e.g., disinformation, generating fake profiles, surveillance), fairness considerations (e.g., deployment of technologies that could make decisions that unfairly impact specific groups), privacy considerations, and security considerations.

- The conference expects that many papers will be foundational research and not tied to particular applications, let alone deployments. However, if there is a direct path to any negative applications, the authors should point it out. For example, it is legitimate to point out that an improvement in the quality of generative models could be used to generate Deepfakes for disinformation. On the other hand, it is not needed to point out that a generic algorithm for optimizing neural networks could enable people to train models that generate Deepfakes faster.
- The authors should consider possible harms that could arise when the technology is being used as intended and functioning correctly, harms that could arise when the technology is being used as intended but gives incorrect results, and harms following from (intentional or unintentional) misuse of the technology.
- If there are negative societal impacts, the authors could also discuss possible mitigation strategies (e.g., gated release of models, providing defenses in addition to attacks, mechanisms for monitoring misuse, mechanisms to monitor how a system learns from feedback over time, improving the efficiency and accessibility of ML).

11. **Safeguards**

    Question: Does the paper describe safeguards that have been put in place for responsible release of data or models that have a high risk for misuse (e.g., pre-trained language models, image generators, or scraped datasets)?

    Answer: [N/A]

    Justification: This paper does not alter the capabilities of the available models or datasets, but rather provides a more efficient approach to use them on hardware with limited capabilities. Thus, we believe that our paper has a neutral risk impact in this area.

    Guidelines:

    - The answer [N/A] means that the paper poses no such risks.
    - Released models that have a high risk for misuse or dual-use should be released with necessary safeguards to allow for controlled use of the model, for example by requiring that users adhere to usage guidelines or restrictions to access the model or implementing safety filters.
    - Datasets that have been scraped from the Internet could pose safety risks. The authors should describe how they avoided releasing unsafe images.
    - We recognize that providing effective safeguards is challenging, and many papers do not require this, but we encourage authors to take this into account and make a best faith effort.

12. **Licenses for existing assets**

    Question: Are the creators or original owners of assets (e.g., code, data, models), used in the paper, properly credited and are the license and terms of use explicitly mentioned and properly respected?

    Answer: [Yes]

    Justification: All datasets and models used are open-source and publicly available.

    Guidelines:

    - The answer [N/A] means that the paper does not use existing assets.
    - The authors should cite the original paper that produced the code package or dataset.
    - The authors should state which version of the asset is used and, if possible, include a URL.
    - The name of the license (e.g., CC-BY 4.0) should be included for each asset.
    - For scraped data from a particular source (e.g., website), the copyright and terms of service of that source should be provided.
    - If assets are released, the license, copyright information, and terms of use in the package should be provided. For popular datasets, `paperswithcode.com/datasets` has curated licenses for some datasets. Their licensing guide can help determine the license of a dataset.

- For existing datasets that are re-packaged, both the original license and the license of the derived asset (if it has changed) should be provided.
- If this information is not available online, the authors are encouraged to reach out to the asset's creators.

13. **New assets**

    Question: Are new assets introduced in the paper well documented and is the documentation provided alongside the assets?

    Answer: [N/A]

    Justification: This paper does not introduce any new datasets, software libraries, or pre-trained models. All experiments are conducted using publicly available benchmarks, which are cited and described in the main paper.

    Guidelines:

    - The answer [N/A] means that the paper does not release new assets.
    - Researchers should communicate the details of the dataset/code/model as part of their submissions via structured templates. This includes details about training, license, limitations, etc.
    - The paper should discuss whether and how consent was obtained from people whose asset is used.
    - At submission time, remember to anonymize your assets (if applicable). You can either create an anonymized URL or include an anonymized zip file.

14. **Crowdsourcing and research with human subjects**

    Question: For crowdsourcing experiments and research with human subjects, does the paper include the full text of instructions given to participants and screenshots, if applicable, as well as details about compensation (if any)?

    Answer: [N/A]

    Justification: We did not conduct any crowdsourcing experiments or research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Including this information in the supplemental material is fine, but if the main contribution of the paper involves human subjects, then as much detail as possible should be included in the main paper.
    - According to the NeurIPS Code of Ethics, workers involved in data collection, curation, or other labor should be paid at least the minimum wage in the country of the data collector.

15. **Institutional review board (IRB) approvals or equivalent for research with human subjects**

    Question: Does the paper describe potential risks incurred by study participants, whether such risks were disclosed to the subjects, and whether Institutional Review Board (IRB) approvals (or an equivalent approval/review based on the requirements of your country or institution) were obtained?

    Answer: [N/A]

    Justification: We did not conduct any research with human subjects.

    Guidelines:

    - The answer [N/A] means that the paper does not involve crowdsourcing nor research with human subjects.
    - Depending on the country in which research is conducted, IRB approval (or equivalent) may be required for any human subjects research. If you obtained IRB approval, you should clearly state this in the paper.

- We recognize that the procedures for this may vary significantly between institutions and locations, and we expect authors to adhere to the NeurIPS Code of Ethics and the guidelines for their institution.
- For initial submissions, do not include any information that would break anonymity (if applicable), such as the institution conducting the review.

16. **Declaration of LLM usage**

    Question: Does the paper describe the usage of LLMs if it is an important, original, or non-standard component of the core methods in this research? Note that if the LLM is used only for writing, editing, or formatting purposes and does *not* impact the core methodology, scientific rigor, or originality of the research, declaration is not required.

    Answer: [N/A]

    Justification: LLM is used only for writing, editing, and formatting purposes.

    Guidelines:

    - The answer [N/A] means that the core method development in this research does not involve LLMs as any important, original, or non-standard components.
    - Please refer to our LLM policy in the NeurIPS handbook for what should or should not be described.