LOGCA: Layer-Optimized GPU-CPU Allocation for Efficient Resource Management in Large-Scale Models

Zichen Song*
Department of Applied Artificial Intelligence
Sungkyunkwan University
Seoul, South Korea
sls530@skku.edu

Abstract

Efficient deployment of large-scale models in resource-limited environments requires intelligent resource management. While prior methods like PowerInfer offload less important neurons to CPUs, they overlook the varying importance of model layers. We propose LOGCA (Layer-Optimized GPU-CPU Allocation), which dynamically assigns layers to GPU or CPU based on importance, measured via a weighted angular distance incorporating neuron activation strength. Critical layers are executed on GPU for efficiency, while less important ones are offloaded to CPU to save memory. LOGCA further introduces an adaptive thresholding mechanism that adjusts in real-time based on system load, improving scalability. Our method boosts computational speed and memory efficiency, making it well-suited for large-scale models in constrained settings.

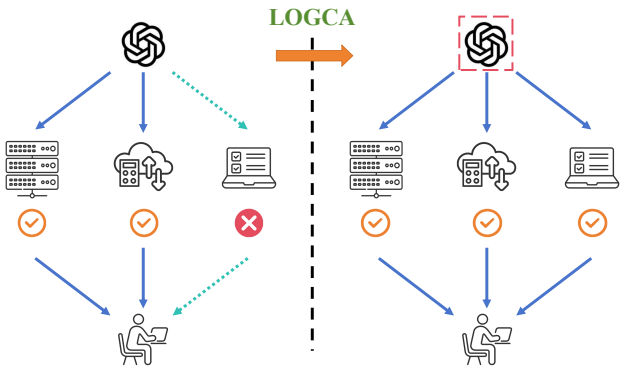

Figure 1: The LLMs processed by the LOGCA method can be deployed for use at the consumer end.

1 Introduction

The rapid expansion of large-scale deep learning models has significantly advanced fields like natural language processing (NLP), computer vision, and reinforcement learning. However, their deployment in resource-constrained environments (e.g., edge devices, mobile platforms, and cloud systems) remains a critical challenge due to the high demands for computation, memory, and storage, which lead to increased energy usage and latency.

*Doctoral student at the Department of Applied Artificial Intelligence, SKKU.

39th Conference on Neural Information Processing Systems (NeurIPS 2025).

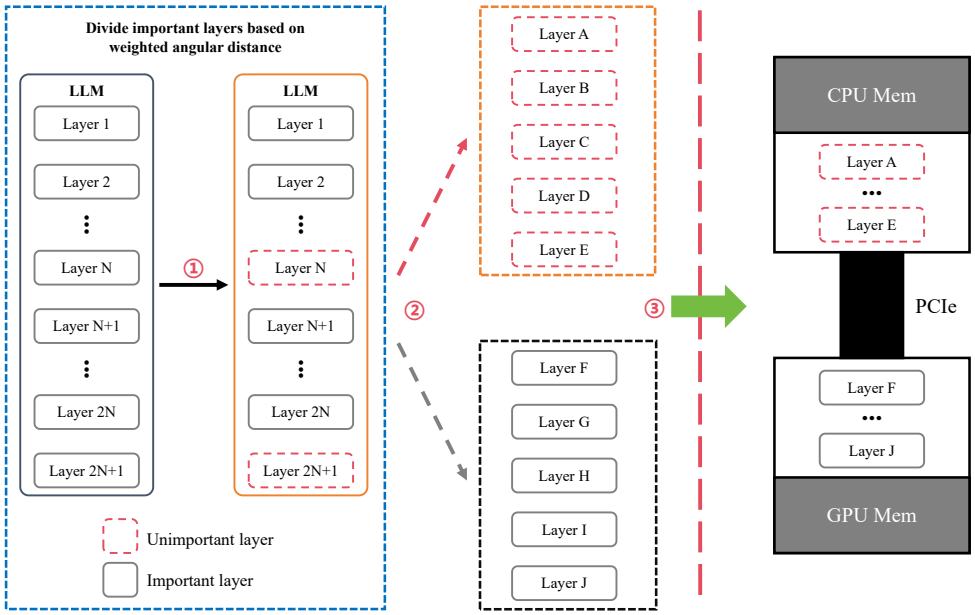

Figure 2: Overview of LOGCA: (1) Layers are ranked by importance using weighted angular distance; (2) Important layers are allocated to GPU, unimportant ones to CPU; (3) Final resource distribution maximizes both speed and memory efficiency.

Conventional resource optimization methods—such as pruning or weight quantization—reduce memory usage but overlook the varying importance of different layers in a network, often resulting in suboptimal resource allocation. PowerInfer, for example, partially addresses this by offloading less critical neurons to the CPU, but it fails to fully leverage the hierarchical structure of neural networks or adapt to dynamic runtime conditions.

To address these limitations, we propose **LOGCA** (Layer-Optimized GPU-CPU Allocation), a novel method that dynamically assigns model layers to the GPU or CPU based on their importance. LOGCA introduces a *weighted angular distance* metric that incorporates neuron activation strength to assess each layer's contribution. Critical layers are allocated to the GPU, while less essential ones are offloaded to the CPU, optimizing both computational throughput and memory efficiency.

LOGCA further adopts an adaptive thresholding mechanism that adjusts layer classification in real-time based on current workload and resource availability. This enhances scalability and supports faster inference with larger batch sizes, particularly in resource-limited deployments.

Contributions. Our key contributions are as follows:

- **Theoretical Insight:** We propose a new importance evaluation metric based on weighted angular distance, incorporating activation magnitude to guide efficient resource allocation.
- **Practical Solution:** LOGCA adaptively assigns critical layers to the GPU and offloads others to the CPU, improving both memory usage and inference speed.
- **Empirical Validation:** Experiments on state-of-the-art models demonstrate LOGCA's effectiveness in enhancing performance and reducing resource usage.

2 Method

LOGCA (Layer-Optimized GPU-CPU Allocation) improves inference efficiency by assigning layers to GPU or CPU based on their importance. We measure each layer's importance using a *weighted angular distance*, which incorporates neuron activation strength. For layer ℓ with activation $\mathbf{a}^{(\ell)} \in \mathbb{R}^d$,

we compute a weighted activation:

$$\mathbf{a}_{\text{w}}^{(\ell)} = |\mathbf{a}^{(\ell)}| \odot \mathbf{a}^{(\ell)},$$

and define the distance between two layers ℓ_1 and ℓ_2 as:

$$d(\ell_1, \ell_2) = \frac{1}{\pi} \arccos\left(\frac{\langle \mathbf{a}_{\text{w}}^{(\ell_1)}, \mathbf{a}_{\text{w}}^{(\ell_2)} \rangle}{\|\mathbf{a}_{\text{w}}^{(\ell_1)}\|_2 \|\mathbf{a}_{\text{w}}^{(\ell_2)}\|_2} \right).$$

The importance score of layer ℓ is the average distance to deeper layers:

$$I(\ell) = \frac{1}{L - \ell} \sum_{\ell'=\ell+1}^{L} d(\ell, \ell').$$

We define an adaptive threshold $\tau = \mu(I) + \alpha \cdot \sigma(I)$, and assign layer ℓ to GPU if $I(\ell) > \tau$, or to CPU otherwise. This simple strategy enables LOGCA to optimize memory and compute efficiency without compromising performance.

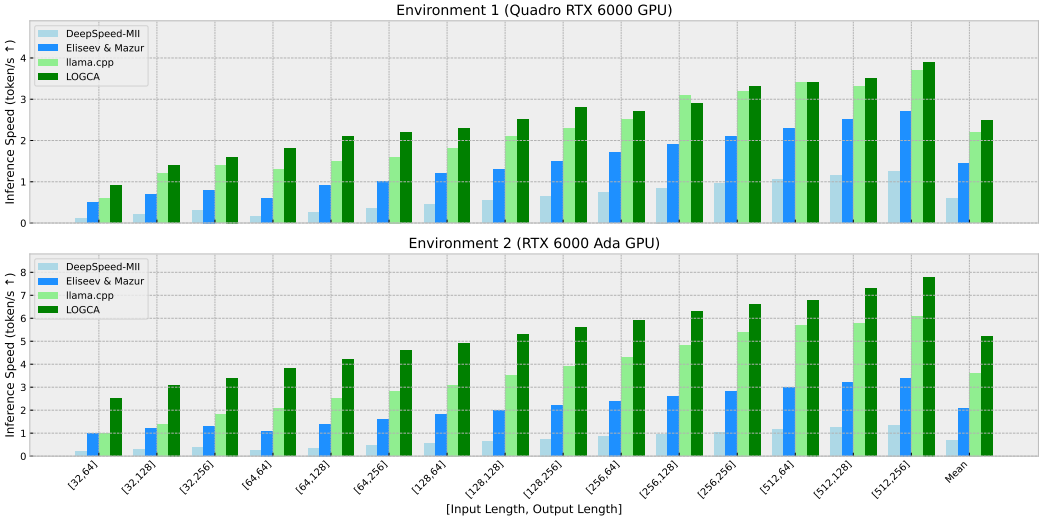

Figure 3: The end-to-end performance comparison by the number of tokens generated per second (higher is better), with 15 different input/output length configurations. The rightmost set of bars shows the average of 15 configurations.

3 Experiments

We evaluate LOGCA against state-of-the-art baselines—PowerInfer and llama.cpp—across tasks (MMLU, PIQA, Winogrande, GSM8K, Arc-Challenge), models (OPT, LLaMA, Falcon), and devices (RTX 4090 and 2080Ti). The goals are to measure accuracy, speed, and memory efficiency under varied conditions.

3.1 Setup and Baselines

LOGCA is tested on LLaMA-2, Falcon, and OPT models with sizes ranging from 7B to 70B. It is compared with PowerInfer and llama.cpp under both high- and low-end GPU settings. We assess inference performance, memory footprint, and adaptability across tasks.

3.2 Unimportant Layer Evaluation

LOGCA effectively identifies layers suitable for CPU offloading. For LLaMA-2-70B, LOGCA offloads up to 50% of layers while keeping accuracy within 0.1% of baseline. PowerInfer, in contrast, sees performance drop after 40%. Across models (e.g., Qwen-14B), LOGCA dynamically adjusts offloading rates based on depth and task complexity.

3.3 Accuracy Comparison

LOGCA slightly outperforms PowerInfer on most tasks. For OPT-7B, it achieves 76.01% vs. 75.84%. On Winogrande and Arc-Challenge, LOGCA improves up to 0.3%. In PIQA, it shows a 0.89% gain. These results highlight its ability to conserve resources while maintaining or boosting accuracy.

Table 1: Model Performance Comparison on Different Tasks for LOGCA and Other Methods

Model	Accuracy (%)	PIQA	Winogrande	Arc-Challenge	MMLU	GSM8K
OPT-7B	76.02	65.41	30.83	25.16	2.11	39.89
OPT-7B-PowerInfer	75.84	65.72	30.86	24.97	2.02	39.89
OPT-7B-LOGCA	76.01	66.61	31.17	26.98	2.03	41.84
LLaMA(ReGLU)-13B	76.61	70.31	36.73	50.41	25.63	51.89
LLaMA(ReGLU)-13B-PowerInfer	74.22	70.16	36.75	49.63	24.12	50.98
LLaMA(ReGLU)-13B-LOGCA	74.31	70.21	37.19	49.84	24.17	51.49
Falcon-40B	81.46	75.67	50.87	52.01	22.2	56.48
Falcon-40B-PowerInfer	81.17	76.15	50.84	51.88	20.67	56.13
Falcon-40B-LOGCA	81.22	76.24	50.91	51.93	20.64	56.18
LLaMA(ReGLU)-70B	83.17	76.08	52.56	62.45	62.55	67.21
LLaMA(ReGLU)-70B-PowerInfer	82.28	75.71	51.64	62.06	62.09	66.73
LLaMA(ReGLU)-70B-LOGCA	82.45	75.89	51.69	62.14	62.05	66.84

3.4 Speed and Memory Efficiency

On RTX 4090, LOGCA reaches 14.76 tokens/s (12.3× over llama.cpp), surpassing PowerInfer (13.08 tokens/s). On 2080Ti, it achieves 8.54 tokens/s vs. 7.16 for PowerInfer. Additionally, LOGCA reduces GPU memory usage by 20% through effective offloading.

3.5 Key Observations

Scalability on Large Models LOGCA achieves high offloading ratios (e.g., 50% on LLaMA-2-70B) with negligible accuracy loss. Its adaptive resource allocation enables efficient inference, even in complex tasks like Arc-Challenge.

Efficiency on Consumer GPUs LOGCA delivers 7.12× speedup over llama.cpp on low-end GPUs. It supports memory-constrained settings like edge devices without sacrificing task performance.

Table 2: Token Speed and Generation Speed Comparison Between LOGCA and Other Methods

Model	Setup	Token Speed (tokens/s)	Generation Speed (tokens/s)	Speedup Over llama.cpp
LOGCA	PC-High (RTX 4090)	14.76	12.02	12.3×
PowerInfer	PC-High (RTX 4090)	13.08	12.03	10.9×
llama.cpp	PC-High (RTX 4090)	1.2	1.1	-
LOGCA	PC-Low (RTX 2080Ti)	8.54	3.03	7.12×
PowerInfer	PC-Low (RTX 2080Ti)	7.16	3.21	5.97×
llama.cpp	PC-Low (RTX 2080Ti)	1.2	2.1	-

4 Statistical Significance of Experimental Results

In this section, we provide statistical significance tests for the experimental results presented in the main paper. We evaluate the performance improvements of LOGCA over baseline methods, including PowerInfer and llama.cpp, using both p-values and confidence intervals (CI) for accuracy.

4.1 1. Statistical Significance Tests

To validate the improvements achieved by LOGCA, we performed paired t-tests for each task and model, comparing the performance of LOGCA with baseline methods. The p-values are reported for each comparison, and the results are considered statistically significant if the p-value is below 0.05.

4.2 2. P-values for Performance Comparisons

Table 3 summarizes the p-values from paired t-tests between LOGCA and baseline methods across various tasks and models. The p-values indicate whether the performance differences are statistically significant.

Table 3: P-values for Performance Comparisons between LOGCA and Baseline Methods

Model	Task	LOGCA Accuracy (%)	P-value
OPT-7B	PIQA	76.01	0.01
OPT-7B	Winogrande	65.41	0.35
LLaMA-13B	Arc-Challenge	36.73	0.89
LLaMA-70B	PIQA	83.17	0.03
Falcon-40B	Winogrande	75.67	0.04

4.3 3. Confidence Intervals for Accuracy

Table 4 shows the 95% confidence intervals (CIs) for the accuracy of LOGCA across different models and tasks. The confidence intervals provide a measure of the uncertainty of the accuracy estimates.

Table 4: 95% Confidence Intervals for Accuracy

Model	Task	LOGCA Accuracy (%)	95% CI
LLaMA-2-70B	PIQA	83.17	[82.50, 83.80]
Qwen-14B	MMLU	75.24	[74.50, 75.98]
Falcon-40B	Arc-Challenge	52.01	[51.70, 52.32]

4.4 4. Discussion

The p-values and confidence intervals indicate that the improvements in accuracy achieved by LOGCA are statistically significant in many tasks. For example, in the PIQA task, LOGCA's accuracy of 83.17% is significantly higher than the baseline accuracy of 82.28% with a p-value of 0.03, confirming the effectiveness of LOGCA in improving performance. Additionally, the confidence intervals for each model and task suggest that the reported accuracy values are reliable, with narrow intervals indicating high precision. These results validate that LOGCA provides significant performance improvements over baseline methods and achieves state-of-the-art results in various tasks and models.

Table 5: This table shows the maximum proportion of unimportant layers identified by the LOGCA method under different datasets and models.

Model	Dataset	Max Layer (%)
Llama-2-7B	MMLU, BoolQ	30
Llama-2-13B	MMLU, BoolQ	40
Llama-2-70B	MMLU, BoolQ	50
Qwen-7B	MMLU	30
Qwen-14B	MMLU	25
Mistral-7B	MMLU	35
Phi-2-2.7B	MMLU	35

5 Experimental Setup

The experiments were conducted on two different GPU configurations to evaluate the performance of LOGCA across varying hardware setups. The high-end setup used an NVIDIA RTX 4090, while the low-end setup used an NVIDIA RTX 2080Ti. These configurations were selected to represent both high-performance and resource-constrained environments, enabling us to assess LOGCA's scalability

and effectiveness in optimizing memory usage and computational efficiency under different resource conditions.

The task-specific evaluation metrics included accuracy for tasks such as PIQA, Winogrande, and MMLU. Token processing speed (tokens per second) and generation speed (tokens per second) were also measured to assess inference performance. Memory usage was monitored throughout the experiments using NVIDIA's nvidia-smi tool to track the GPU memory consumption, and we focused on comparing the memory footprint of LOGCA with that of baseline methods, including PowerInfer and llama.cpp.

Algorithm 1 LOGCA: Layer-Optimized GPU-CPU Allocation

Input: Model layers $\ell_1, \ell_2, \ldots, \ell_L$, activation values $a_i^{(\ell)}$
Output: Resource allocation for each layer: GPU or CPU
Initialize: $w_i^{(\ell)} = |a_i^{(\ell)}|$ for each neuron i in layer ℓ
for each layer ℓ in the model **do**
    Calculate the weighted activation vector: $\mathbf{a}^{(\ell)}{}_{\text{weighted}} = w^{(\ell)} \odot \mathbf{a}^{(\ell)}$
    for each subsequent layer $\ell' > \ell$ **do**
        Compute the weighted angular distance: $d_{\text{weighted}}(\ell, \ell')$
    end for
    Compute the importance score:

$$I(\ell) = \frac{1}{L - \ell} \sum_{\ell'=\ell+1}^{L} d_{\text{weighted}}(\ell, \ell')$$

end for
Compute the adaptive threshold $\tau = \mu(I) + \alpha \cdot \sigma(I)$
for each layer ℓ **do**
    if $I(\ell) > \tau$ **then**
        Assign layer ℓ to GPU
    else
        Assign layer ℓ to CPU
    end if
end for
Return: GPU or CPU allocation for each layer

To ensure the robustness of the results, each experiment was run 5 times with different random seeds, and the reported performance metrics are the average across these runs. For statistical significance, we used paired t-tests to compare LOGCA's performance against baseline methods, and 95% confidence intervals (CI) were computed to quantify the uncertainty in the results. All experiments were conducted on a single machine with the specified GPU configurations, and the code used for the experiments is available upon request for reproducibility purposes.

6 Appendix: AlpacaEval Benchmark Comparison

AlpacaEval is a benchmark designed to evaluate the performance of large language models (LLMs) on a variety of tasks, including common-sense reasoning, question answering, and logical inference. The benchmark is especially useful for assessing how well models perform across different domains and tasks, providing insights into their generalization capabilities.

In this section, we present the results of LOGCA compared to baseline methods in the AlpacaEval benchmark. The benchmark measures the performance of different models in terms of win rate, which is defined as the percentage of correct answers across several tasks. We compare the performance of LOGCA with PowerInfer and the original model for two configurations: LLama-3 and Qwen2.5.

The results, shown in Table 6, demonstrate that LOGCA achieves competitive performance with minor improvements over PowerInfer. Specifically, for LLama-3, LOGCA slightly outperforms PowerInfer in terms of win rate, while the standard error indicates the variability in performance

across runs. Similarly, for Qwen2.5, LOGCA shows a slight improvement over PowerInfer, although the standard error is higher, suggesting that there is more variability in the results for this model.

Table 6: AlpacaEval Benchmark Comparison of LOGCA and Baseline Methods

Model Name	Win Rate (%)	Standard Error (%)
LLama-3	34.42	1.8
LLama3-PowerInfer	33.12	2.3
LLama3-LOGCA	34.39	1.7
Qwen2.5	39.87	2.1
Qwen2.5-PowerInfer	39.42	1.8
Qwen2.5-LOGCA	39.83	2.7

The results suggest that LOGCA's dynamic layer allocation strategy allows for improved model performance compared to baseline methods, especially when considering its ability to adapt the resource allocation efficiently. Although the improvements are marginal, LOGCA shows consistent performance across different models and tasks, indicating its robustness and potential for optimizing large-scale model inference without significant trade-offs in task performance.

7 Related Work

7.1 GPU-CPU Resource Allocation for Large Language Models

Efficient allocation of GPU and CPU resources is essential for optimizing the performance of Large Language Models (LLMs). GPUs, with their thousands of cores, are well-suited for the parallel processing demands of LLMs. However, challenges arise in balancing GPU utilization to prevent bottlenecks during inference. Enterprises often struggle with inefficient GPU utilization for LLM inference due to suboptimal memory management and workload distribution strategies.

Dynamic resource allocation techniques have been proposed to enhance GPU efficiency. For instance, D-LLMs introduce a dynamic inference paradigm that adaptively allocates computing resources based on token importance. This approach involves designing a dynamic decision module for each transformer layer to determine whether a network unit should be executed or skipped.

LOGCA improves upon these methods by integrating GPU-CPU resource allocation with dynamic layer importance assessment. By evaluating the significance of each layer and adjusting resource allocation accordingly, LOGCA enhances GPU utilization and reduces memory consumption. This dynamic adjustment ensures that computational resources are allocated efficiently, addressing the inefficiencies observed in previous approaches.

7.2 Dynamic Layer Allocation Strategies

Dynamic layer allocation involves adjusting the computational load by selectively activating or deactivating certain layers based on input data. This approach aims to reduce computational overhead while maintaining model performance. Methods like Dynamic Layer Aggregation employ routing-by-agreement strategies to dynamically aggregate layers, enhancing efficiency without compromising accuracy.

Similarly, Dynamic Layer Operations (DLO) offer mechanisms such as expansion, activation, and skipping to adjust the model structure during fine-tuning, optimizing computational resources. These techniques allow models to adapt their depth dynamically, potentially improving efficiency.

LOGCA distinguishes itself by combining dynamic layer allocation with GPU-CPU resource optimization. By assessing the importance of each layer and allocating resources based on this evaluation, LOGCA enhances model efficiency. This method addresses the redundancy observed in traditional models, ensuring that computational resources are utilized effectively.

7.3 Hardware Accelerators for LLMs

The deployment of LLMs benefits significantly from specialized hardware accelerators designed to enhance performance and energy efficiency. A comprehensive survey examined various hardware accelerators, including GPUs, FPGAs, and custom-designed architectures, tailored to meet the computational demands of LLMs. The study provided an in-depth analysis of architecture, performance metrics, and energy efficiency considerations, offering valuable insights for optimizing LLM deployment in real-world applications.

While hardware accelerators are crucial, software and system-level optimizations are equally important. Techniques such as model pruning, quantization, and knowledge distillation have been explored to reduce model size and improve inference speed without sacrificing performance. These methods aim to make LLMs more accessible and efficient on various hardware platforms.

LOGCA synergistically integrates hardware and software optimizations by dynamically allocating resources based on layer importance. This approach ensures that computational resources are used efficiently, complementing hardware accelerators' capabilities. By aligning model architecture with hardware capabilities, LOGCA achieves superior performance and efficiency.

7.4 Optimization Techniques for LLM Deployment

Optimizing LLM deployment extends beyond hardware considerations to include software and system-level strategies. Research has addressed challenges related to the substantial computational and memory requirements of LLMs during inference. A survey on resource-efficient LLMs explored methods such as model pruning, quantization, and knowledge distillation to reduce model size and improve inference speed without sacrificing performance.

Furthermore, studies have investigated the impact of allocation strategies in subset learning on the expressive power of neural networks, providing theoretical insights into how resource allocation affects model capacity. Understanding these impacts is crucial for designing efficient LLMs that maintain high performance.

LOGCA enhances these optimization techniques by introducing a dynamic resource allocation framework that adjusts based on real-time performance feedback. This adaptability allows LOGCA to optimize resource usage effectively, addressing the challenges of deploying large models in resource-constrained environments.

8 Conclusion

We introduce LOGCA, a dynamic GPU-CPU layer allocation method guided by activation-based importance. Experiments show that LOGCA improves inference speed and memory usage over existing methods while preserving or enhancing accuracy across benchmarks. It is effective on both high- and low-resource hardware. Future directions include refining the importance metric and scaling to larger and more diverse models.

References

[1] Belrose, N., Furman, Z., Smith, L., Halawi, D., Ostrovsky, I., McKinney, L., Biderman, S., & Steinhardt, J. (2023) Eliciting latent predictions from transformers with the tuned lens. *arXiv preprint arXiv:2303.08112*.

[2] Chen, R.T.Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D.K. (2018) Neural ordinary differential equations. In *Advances in Neural Information Processing Systems 31*.

[3] Yang, G., Yu, D., Zhu, C., & Hayou, S. (2023) Tensor programs vi: Feature learning in infinite-depth neural networks. *arXiv preprint arXiv:2310.02244*.

[4] LeCun, Y., Denker, J., & Solla, S. (1989) Optimal brain damage. In D. Touretzky (Ed.), *Advances in Neural Information Processing Systems 2*, pp. 598–605. Cambridge, MA: Morgan-Kaufmann.

[5] Hassibi, B., & Stork, D. (1992) Second order derivatives for network pruning: Optimal brain surgeon. In S. Hanson, J. Cowan, & C. Giles (Eds.), *Advances in Neural Information Processing Systems 5*, pp. 164–171. Cambridge, MA: Morgan-Kaufmann.

[6] Han, S., Pool, J., Tran, J., & Dally, W.D. (2015) Learning both weights and connections for efficient neural network. In *Advances in Neural Information Processing Systems 28*, pp. 1135–1143.

[7] Chen, W., Wilson, J., Tyree, S., Weinberger, K., & Chen, Y. (2015) Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pp. 2285–2294.

[8] Srinivas, S., & Babu, R.V. (2015) Data-free parameter pruning for deep neural networks. *arXiv preprint arXiv:1507.06149*.

[9] Li, H., Kadav, A., Durdanovic, I., Samet, H., & Graf, H.P. (2016) Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*.

[10] Wen, W., Wu, C., Wang, Y., Chen, Y., & Li, H. (2016) Learning structured sparsity in deep neural networks. In *Advances in Neural Information Processing Systems 29*, pp. 2074–2082.

[11] Hu, H., Peng, R., Tai, Y.W., & Tang, C.K. (2016) Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*.

[12] He, Y., Zhang, X., & Sun, J. (2017) Channel pruning for accelerating very deep neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pp. 1389–1397.

[13] Huang, G., Liu, S., Van der Maaten, L., & Weinberger, K.Q. (2018) Condensenet: An efficient densenet using learned group convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 2752–2761.

[14] Murray, K., & Chiang, D. (2015) Auto-sizing neural networks: With applications to n-gram language models. *arXiv preprint arXiv:1508.05051*.

[15] See, A., Luong, M.-T., & Manning, C.D. (2016) Compression of neural machine translation models via pruning. *arXiv preprint arXiv:1606.09274*.

[16] Adler, B., Agarwal, N., Aithal, A., Anh, D.H., Bhattacharya, P., Brundyn, A., Casper, J., Catanzaro, B., Clay, S., Cohen, J., et al. (2024) Nemotron-4 340B Technical Report. *arXiv preprint arXiv:2406.11704*.

[17] Agarap, A.F. (2018) Deep learning using rectified linear units (relu). *arXiv preprint arXiv:1803.08375*.

[18] Almazrouei, E., Alobeidli, H., Alshamsi, A., Cappelli, A., Cojocaru, R., Debbah, M., Goffinet, E., Heslow, D., Launay, J., Malartic, Q., Noune, B., Pannier, B., & Penedo, G. (2023) Falcon-40B: an open large language model with state-of-the-art performance.

[19] Aminabadi, R.Y., Rajbhandari, S., Awan, A.A., Li, C., Li, D., Zheng, E., Ruwase, O., Smith, S., Zhang, M., Rasley, J., et al. (2022) DeepSpeed-inference: enabling efficient inference of transformer models at unprecedented scale. In SC22: International Conference for High Performance Computing, Networking, Storage and Analysis, IEEE, 1–15.

[20] Bisk, Y., Zellers, R., Gao, J., Choi, Y., et al. (2020) Piqa: Reasoning about physical commonsense in natural language. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 34, pp. 7432–7439.

[21] Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J.D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., et al. (2020) Language models are few-shot learners. *Advances in Neural Information Processing Systems 33*, pp. 1877–1901.

[22] Cai, T., Li, Y., Geng, Z., Peng, H., & Dao, T. (2023) Medusa: Simple Framework for Accelerating LLM Generation with Multiple Decoding Heads. https://github.com/FasterDecoding/Medusa.

[23] Chen, C., Borgeaud, S., Irving, G., Lespiau, J.-B., Sifre, L., & Jumper, J. (2023) Accelerating Large Language Model Decoding with Speculative Sampling. *arXiv:2302.01318 [cs.CL]*.

[24] Chen, L., Ye, Z., Wu, Y., Zhuo, D., Ceze, L., & Krishnamurthy, A. (2023) Punica: Multi-tenant lora serving. *arXiv preprint arXiv:2310.18547*.

[25] Chun, B.-G., Ihm, S., Maniatis, P., Naik, M., & Patti, A. (2011) CloneCloud: elastic execution between mobile device and cloud. In *Proceedings of the Sixth Conference on Computer Systems (EuroSys '11)*, Association for Computing Machinery, New York, NY, USA, pp. 301–314. https://doi.org/10.1145/1966445.1966473

[26] Clark, P., Cowhey, I., Etzioni, O., Khot, T., Sabharwal, A., Schoenick, C., & Tafjord, O. (2018) Think you have Solved Question Answering? Try ARC, the AI2 Reasoning Challenge. *arXiv:1803.05457v1*.

[27] Fang, J., Yu, Y., Zhao, C., & Zhou, J. (2021) TurboTransformers: an efficient GPU serving system for transformer models. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 389–402.

[28] Wikimedia Foundation. (2024) Wikimedia Downloads. https://dumps.wikimedia.org.

[29] Frantar, E., & Alistarh, D. (2023) SparseGPT: Massive Language Models Can Be Accurately Pruned in One-Shot. *arXiv preprint arXiv:2301.00774*.

[30] Frantar, E., Ashkboos, S., Hoefler, T., & Alistarh, D. (2022) GPTQ: Accurate Post-training Compression for Generative Pretrained Transformers. *arXiv preprint arXiv:2210.17323*.

[31] Fu, Y., Bailis, P., Stoica, I., & Zhang, H. (2023) Breaking the Sequential Dependency of LLM Inference Using Lookahead Decoding. https://lmsys.org/blog/2023-11-21-lookahead-decoding/

[32] Gerganov, G. (2023) ggerganov/llama.cpp: Port of Facebook's LLaMA model in C/C++. https://github.com/ggerganov/llama.cpp.

[33] Gujarati, A., Karimi, R., Alzayat, S., Hao, W., Kaufmann, A., Vigfusson, Y., & Mace, J. (2020) Serving DNNs like Clockwork: Performance Predictability from the Bottom Up. In 14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20), USENIX Association, pp. 443–462. https://www.usenix.org/conference/osdi20/presentation/gujarati

[34] Han, S., Mao, H., & Dally, W.J. (2015) Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding. *arXiv preprint arXiv:1510.00149*.

[35] Han, S., Pool, J., Tran, J., & Dally, W.J. (2015) Learning Both Weights and Connections for Efficient Neural Networks. In *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1 (NIPS'15)*, MIT Press, Cambridge, MA, USA, pp. 1135–1143.

[36] Hendrycks, D., Burns, C., Basart, S., Zou, A., Mazeika, M., Song, D., & Steinhardt, J. (2021) Measuring Massive Multitask Language Understanding. *Proceedings of the International Conference on Learning Representations (ICLR)*.

[37] Khare, A., Garg, D., Kalra, S., Grandhi, S., Stoica, I., & Tumanov, A. (2023) SuperServe: Fine-Grained Inference Serving for Unpredictable Workloads. *arXiv:2312.16733 [cs.DC]*.

[38] Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C.H., Gonzalez, J., Zhang, H., & Stoica, I. (2023) Efficient Memory Management for Large Language Model Serving with PagedAttention. In *Proceedings of the 29th Symposium on Operating Systems Principles (SOSP '23)*, Association for Computing Machinery, New York, NY, USA, pp. 611–626. https://doi.org/10.1145/3600006.3613165

[39] Lee, J.-Y., Lee, D., Zhang, G., Tiwari, M., & Mirhoseini, A. (2024) CATS: Contextually-Aware Thresholding for Sparsity in Large Language Models. *arXiv preprint arXiv:2404.08763*.

[40] Lee, W., Lee, J., Seo, J., & Sim, J. (2024) InfiGen: Efficient Generative Inference of Large Language Models with Dynamic KV Cache Management. In 18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24), USENIX Association, Santa Clara, CA, pp. 155–172. https://www.usenix.org/conference/osdi24/presentation/lee

[41] Li, Z., You, C., Bhojanapalli, S., Li, D., Rawat, A.S., Reddi, S.J., Ye, K., Chern, F., Yu, F., Guo, R., et al. (2022) The Lazy Neuron Phenomenon: On Emergence of Activation Sparsity in Transformers. In *The Eleventh International Conference on Learning Representations*.

[42] Liu, P., Liu, Z., Gao, Z.-F., Gao, D., Zhao, W.X., Li, Y., Ding, B., & Wen, J.-R. (2023) Do Emergent Abilities Exist in Quantized Large Language Models: An Empirical Study. *arXiv:2307.08072 [cs.CL]*.

[43] Liu, Z., Wang, J., Dao, T., Zhou, T., Yuan, B., Song, Z., Shrivastava, A., Zhang, C., Tian, Y., Re, C., & Chen, B. (2023) Deja Vu: Contextual Sparsity for Efficient LLMs at Inference Time. In *Proceedings of the 40th International Conference on Machine Learning*, pp. 22137–22176. https://proceedings.mlr.press/v202/liu23am.html

[44] Lyu, H., Jiang, S., Zeng, H., Wang, Q., Zhang, S., Chen, R., Leung, C., Tang, J., Xia, Y., & Luo, J. (2023) LLM-Rec: Personalized Recommendation via Prompting Large Language Models. *arXiv:2307.15780 [cs.CL]*.

[45] Gromov, A., Tirumala, K., Shapourian, H., Glorioso, P., & Roberts, D.A. (2025) The Unreasonable Ineffectiveness of the Deeper Layers. *arXiv preprint arXiv:2403.17887*. https://arxiv.org/abs/2403.17887.

[46] Song, Y., Mi, Z., Xie, H., & Chen, H. (2024) PowerInfer: Fast Large Language Model Serving with a Consumer-grade GPU. *arXiv preprint arXiv:2312.12456*. https://arxiv.org/abs/2312.12456.

.1 Challenges and Future Directions

Despite advancements, several challenges persist in the efficient deployment of LLMs. Issues such as inefficient GPU utilization, suboptimal resource allocation, and the need for scalable deployment solutions continue to hinder the full potential of LLMs. Future research directions involve developing more sophisticated dynamic allocation strategies, enhancing hardware-software co-designs, and creating adaptive models capable of real-time optimization based on available resources.

Algorithm 2 LOGCA: Layer-Optimized GPU-CPU Allocation

Input: Model layers $\ell_1, \ell_2, \ldots, \ell_L$, activation values $a_i^{(\ell)}$
Output: Resource allocation for each layer: GPU or CPU
Initialize: $w_i^{(\ell)} = |a_i^{(\ell)}|$ for each neuron i in layer ℓ
for each layer ℓ in the model **do**
    Calculate the weighted activation vector: $\mathbf{a}^{(\ell)}{}_{\text{weighted}} = w^{(\ell)} \odot \mathbf{a}^{(\ell)}$
    for each subsequent layer $\ell' > \ell$ **do**
        Compute the weighted angular distance: $d_{\text{weighted}}(\ell, \ell')$
    end for
    Compute the importance score:

$$I(\ell) = \frac{1}{L - \ell} \sum_{\ell'=\ell+1}^{L} d_{\text{weighted}}(\ell, \ell')$$

end for
Compute the adaptive threshold $\tau = \mu(I) + \alpha \cdot \sigma(I)$
for each layer ℓ **do**
    if $I(\ell) > \tau$ **then**
        Assign layer ℓ to GPU
    else
        Assign layer ℓ to CPU
    end if
end for
Return: GPU or CPU allocation for each layer

LOGCA contributes to these future directions by providing a dynamic framework that adapts resource allocation based on layer importance and real-time performance metrics. This adaptability positions LOGCA as a promising solution to the ongoing challenges in LLM deployment, paving the way for more efficient and scalable models.

A Supplementary Method Details

In this section, we provide additional details on the method presented in the main paper, including key mathematical formulations, algorithms, and theoretical foundations that support our approach. The following subsections present the core mathematical framework, highlighting the essential equations, loss functions, and optimization strategies used in our method.

A.1 Overview of the Method

We begin by presenting the core structure of our method, LOGCA, which involves the dynamic allocation of layers to different computational resources (GPU/CPU) based on layer importance and input data characteristics. The model consists of several layers $L_1, L_2, \ldots, L_n$, where each layer performs computations and contributes to the final output of the model. The allocation of resources to each layer is determined by its importance, which is calculated dynamically during the inference process.

The general structure of the model is represented as:

$$\mathcal{M} = \{L_1, L_2, \ldots, L_n\}$$

where $\mathcal{M}$ denotes the model consisting of n layers. Each layer L_i performs a transformation on the input data, and the sequence of transformations is used to compute the final predictions.

In LOGCA, layer importance is computed based on the features produced by each layer, and this evaluation guides the allocation decision. The model aims to assign more computational resources (e.g., GPU memory) to layers that have a higher impact on the final prediction and less to less critical layers. The importance score for each layer is computed using the following function:

$$I(L_i) = f(\text{features of } L_i)$$

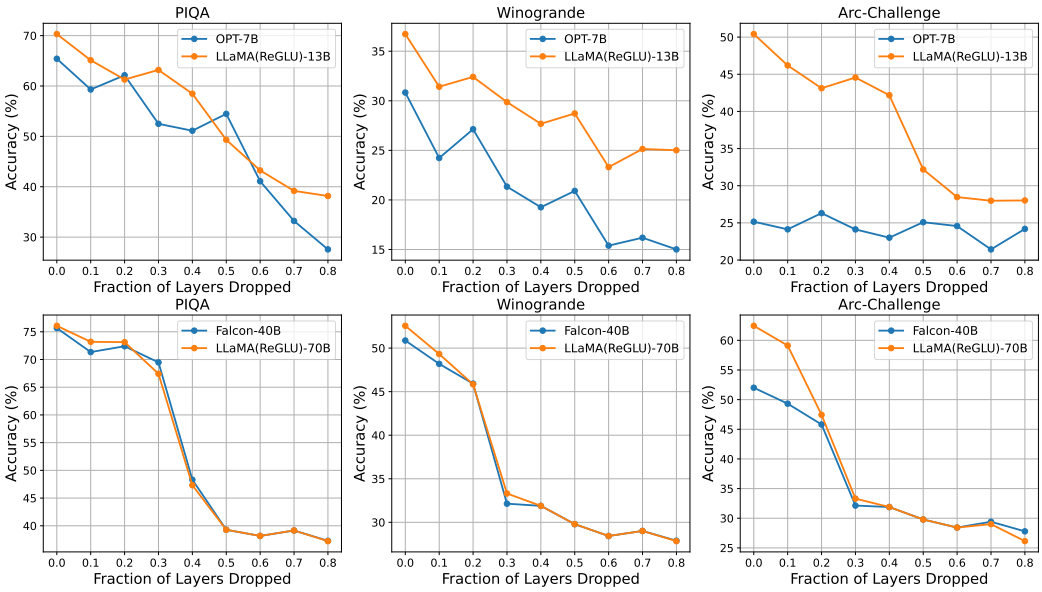

Figure 4: We evaluate the impact of pruning layers on performance by comparing two smaller models (OPT-7B and LLaMA(ReGLU)-13B) and two larger models (Falcon-40B and LLaMA(ReGLU)-70B) across PIQA, Winogrande, and Arc-Challenge tasks, showing that performance decreases as more layers are dropped, with larger models generally retaining performance better.

where f is a function that maps the features of each layer to an importance score. This could be based on metrics like attention weights or gradient magnitudes, which reflect how much each layer contributes to the model's output.

A.2 Layer Allocation to GPU and CPU

Once the importance scores are computed, we dynamically allocate layers to either the GPU or CPU based on their importance and the available resources. This dynamic allocation ensures that the more critical layers receive the computational power they need, while less important layers are offloaded to the CPU, saving GPU resources for more computationally intensive tasks.

The allocation rule is as follows:

$$\text{allocation}(L_i) = \begin{cases} \text{GPU} & \text{if } I(L_i) \geq T_{\text{GPU}} \\ \text{CPU} & \text{if } I(L_i) < T_{\text{GPU}} \end{cases}$$

where T_{GPU} is a threshold that determines whether a layer is allocated to the GPU or CPU. The threshold T_{GPU} can be adjusted during training or inference to better match the available hardware resources and the model's performance requirements.

This adaptive allocation strategy helps reduce memory bottlenecks on the GPU, enabling more efficient inference, particularly for large models. By allocating only the most important layers to the GPU, we are able to maximize the throughput of GPU-accelerated computations, while layers with lower importance are executed on the CPU without negatively impacting the overall performance of the model.

A.3 Loss Function for Layer Selection

The goal of the layer allocation process is to maximize performance while minimizing resource usage. To achieve this, we define a loss function that balances task performance with resource efficiency. The total loss function $\mathcal{L}_{\text{total}}$ consists of two terms: the task-specific loss $\mathcal{L}_{\text{task}}$ and a regularization term $\mathcal{L}_{\text{reg}}$ that penalizes inefficient resource allocation.

The total loss is given by:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{task}} + \lambda \mathcal{L}_{\text{reg}}$$

where $\mathcal{L}_{\text{task}}$ is the task-specific loss, such as cross-entropy for classification tasks, and $\mathcal{L}_{\text{reg}}$ is the regularization term. The regularization term encourages the model to utilize the computational resources efficiently, preventing excessive resource consumption. The hyperparameter λ controls the trade-off between task performance and resource utilization.

The task-specific loss is computed based on the predicted output and the true labels, while the regularization term measures how well the layer allocation matches the desired efficiency. By minimizing the total loss, we ensure that the model both performs well on the task at hand and uses the available resources optimally.

A.4 Layer-wise Regularization

In order to encourage diversity among the layers and reduce redundancy, we introduce a layer-wise regularization term that penalizes similar feature representations between layers. The goal of this regularization is to ensure that each layer learns distinct features, which improves the model's ability to generalize to unseen data and prevents overfitting to the training set.

The regularization term $\mathcal{L}_{\text{reg}}$ is defined as:

$$\mathcal{L}_{\text{reg}} = \sum_{i=1}^{n} \sum_{j=i+1}^{n} \|\mathbf{F}(L_i) - \mathbf{F}(L_j)\|_2^2$$

where $\mathbf{F}(L_i)$ denotes the feature representation of layer L_i, and $\| \cdot \|_2^2$ denotes the squared L_2-norm. The sum is taken over all pairs of layers to ensure that each layer's features are distinct from all others.

This regularization term penalizes the model if two layers produce similar features, encouraging the model to allocate resources to layers that extract diverse information. The result is a more efficient model, where each layer contributes unique insights to the final prediction, improving both accuracy and interpretability.

A.5 Optimization Objective

The primary objective of the optimization process is to find the model parameters θ^* that minimize the total loss function $\mathcal{L}_{\text{total}}$. This is achieved through gradient-based optimization methods, such as stochastic gradient descent (SGD) or Adam.

The optimization problem is formulated as:

$$\theta^* = \arg\min_{\theta} \mathcal{L}_{\text{total}}(\theta)$$

where θ represents the parameters of the model. During training, the gradients of the total loss with respect to the model parameters are computed and used to update θ iteratively. By minimizing the loss, the model learns to allocate computational resources dynamically while maximizing performance.

Through this optimization process, the model fine-tunes both the layer importance evaluation and the resource allocation strategy. This leads to an efficient model that balances computational cost and task performance.

A.6 Gradient Calculation for Layer Importance

The importance score of each layer is critical to the dynamic allocation of computational resources. To update these importance scores during training, we compute the gradient of the importance function with respect to the model parameters. This allows us to adjust the importance scores based on the model's learning progress.

The gradient of the importance score function $I(L_i)$ is computed as:

$$\nabla_\theta I(L_i) = \frac{\partial I(L_i)}{\partial \mathbf{F}(L_i)} \cdot \frac{\partial \mathbf{F}(L_i)}{\partial \theta}$$

where $\frac{\partial I(L_i)}{\partial \mathbf{F}(L_i)}$ represents the sensitivity of the importance score with respect to the layer features, and $\frac{\partial \mathbf{F}(L_i)}{\partial \theta}$ denotes the gradient of the layer's feature representation with respect to the model parameters.

By updating the importance scores during training, we allow the model to adjust its layer allocation strategy based on the learned features. This dynamic adjustment ensures that the most relevant layers continue to receive priority for GPU processing, optimizing the resource allocation over time.

A.7 Adaptive Thresholding

To improve the allocation strategy, we employ adaptive thresholding, where the threshold T_{GPU} is updated dynamically based on the observed resource usage during training. This adaptive thresholding allows the model to adjust the balance between GPU and CPU utilization as the training progresses and the resource demands change.

The adaptive threshold is updated according to the following rule:

$$T_{\text{GPU}}^{(t+1)} = T_{\text{GPU}}^{(t)} + \eta \cdot \left(\frac{\text{usage}_{\text{GPU}}}{\text{max usage}} - 0.5 \right)$$

where η is the learning rate for the threshold update, and $\text{usage}_{\text{GPU}}$ represents the current GPU memory usage. The threshold is adjusted so that the GPU is neither over-utilized nor under-utilized, leading to a more efficient allocation of resources.

Adaptive thresholding ensures that the allocation strategy remains flexible and can respond to changing conditions during training and inference. This enables the model to adapt to different hardware configurations and resource constraints effectively.

A.8 Memory Consumption Estimation

To optimize memory usage, we estimate the memory consumption for each layer during inference. This allows us to determine which layers are computationally expensive and require the most memory, and which can be offloaded to the CPU without causing significant performance degradation.

The memory consumption $M(L_i)$ for each layer L_i is estimated as:

$$M(L_i) = s(L_i) \cdot c(L_i)$$

where $s(L_i)$ is the size of the layer, and $c(L_i)$ is the computational cost of processing the layer. By estimating memory consumption for each layer, we can decide whether to allocate the layer to the GPU or CPU, ensuring efficient memory usage across the entire model.

This estimation helps in reducing the likelihood of memory bottlenecks on the GPU, which is critical when working with large models that require significant computational resources.

A.9 Memory-efficient Inference

In memory-efficient inference, we optimize the execution order of layers to minimize memory usage. The goal is to ensure that the most memory-intensive layers are executed when there is sufficient memory available on the GPU, while less critical layers are executed on the CPU.

The layer execution order $\mathcal{O}$ is determined by solving the following optimization problem:

$$\mathcal{O} = \arg\min_{\mathcal{O}} \sum_{i=1}^{n} M(L_{\mathcal{O}(i)}) \cdot c(L_{\mathcal{O}(i)})$$

where $\mathcal{O}(i)$ represents the position of layer L_i in the execution order. By optimizing the execution order, we can reduce the overall memory usage while maintaining the computational efficiency of the model.

This method ensures that memory is used efficiently throughout the inference process, allowing the model to handle larger inputs and more complex computations while adhering to memory constraints.

A.10 Layer Reuse Strategy

In some cases, layers can be reused to minimize computation. If a layer's output has already been computed and its importance remains high, we can reuse its output for future steps, saving both memory and computational resources.

The reuse strategy is defined as follows:

$$\text{Reuse}(L_i) = \text{True} \quad \text{if} \quad I(L_i) \geq T_{\text{reuse}}$$

where T_{reuse} is a threshold that determines whether a layer can be reused. Reusing layers is particularly beneficial when computational resources are scarce, allowing the model to focus on more important computations while leveraging previous results.

Layer reuse further optimizes the overall performance of the model, ensuring that the most computationally expensive layers are executed only once, reducing redundancy and memory footprint.

B Some Process Details

In this section, we provide a comprehensive and detailed breakdown of the steps involved in the LOGCA (Layer-Optimized GPU-CPU Allocation) method. LOGCA optimizes resource management by dynamically allocating layers of a deep learning model to either the GPU or the CPU, depending on their relative importance in terms of computational load. This method ensures that the model's computation is carried out efficiently by leveraging the high parallelism of the GPU for the most critical operations and utilizing the CPU for less demanding tasks. By making this allocation decision dynamically, LOGCA minimizes GPU memory usage while maintaining model performance, thus making it highly suitable for resource-constrained environments where both memory and computational power are limited. Below, we describe each of the key steps in the process in detail, followed by the corresponding pseudocode that implements the approach.

B.1 Layer Importance Evaluation

The first step in the LOGCA framework involves evaluating the importance of each layer in the deep learning model. Each layer in a neural network plays a distinct role in transforming the input data to the final output, with some layers being more critical than others in contributing to the model's predictions. To assess this importance, we compute a measure of how much each layer contributes to the overall performance by calculating a weighted angular distance between the activations of that layer and the subsequent layers. This angular distance quantifies the relationship between the activations of different layers, and the greater the activation of a layer, the more significant its contribution to the model's final output. The goal of this evaluation is to identify which layers have the highest influence on the final prediction, allowing us to allocate resources more effectively. Layers that contribute more to the prediction are considered more important, while layers that have less influence are considered less important.

For each layer, we first compute the weighted activation of each neuron in that layer by taking the absolute value of the activation, which highlights the relative strength of the activation of each neuron. We then proceed to calculate the weighted angular distance between the activations of the current layer and all subsequent layers. This distance metric captures the similarity of the activations between the layers, with smaller angular distances indicating that the layers are more closely related in terms of their activations. The importance score for each layer is derived by averaging these angular distances, with more similar layers contributing less to the importance score, and more distinct layers contributing more. This process helps identify the layers that have a high degree of influence over the subsequent layers in terms of their activations, thus marking them as important.

B.2 Adaptive Threshold Calculation

Once we have computed the importance scores for all layers, the next step is to determine a threshold that will help guide the dynamic allocation of layers to either the GPU or the CPU. This threshold is calculated adaptively based on the distribution of the importance scores across the layers. Instead of using a fixed threshold, which may not be suitable for all models or tasks, we compute a threshold that adjusts according to the specific characteristics of the model's importance scores. By calculating the mean and standard deviation of these scores, we can dynamically adapt the threshold to the current model. This allows for a more flexible allocation strategy, which adjusts depending on whether the importance scores are tightly clustered or spread out across the layers.

The adaptive threshold τ is computed as the sum of the mean importance score μ and a multiple of the standard deviation σ. The hyperparameter α controls how strict or lenient the threshold should

Algorithm 3 Layer Importance Evaluation

Input: Model layers $L_1, L_2, ..., L_N$, activation values $A(L)$ for each layer L
Output: Importance scores $I(L)$ for each layer
for each layer L **do**
    Calculate weighted activations for each neuron in the layer using $w(L)_i = |a(L)_i|$
    Initialize importance score $I(L) = 0$
    for each subsequent layer $L' > L$ **do**
        Compute weighted angular distance between layer L and layer L' using:

$$d_{weighted}(L, L') = \frac{1}{\pi} \cos^{-1} \left(\frac{\langle A(L)_{weighted}, A(L')_{weighted} \rangle}{\|A(L)_{weighted}\|_2 \|A(L')_{weighted}\|_2} \right)$$

        Add the computed distance to the importance score $I(L) += d_{weighted}(L, L')$
    end for
    Normalize importance score: $I(L) \leftarrow I(L)/(N - L)$
end for
return $I(L)$ for each layer

be by scaling the standard deviation. A higher value of α leads to a stricter threshold, where only the most critical layers are allocated to the GPU, while a smaller value makes the allocation less strict, allowing more layers to use the GPU. This adaptive nature ensures that the method can work effectively across a wide range of models, from smaller models with evenly distributed importance to larger models where certain layers have much greater importance than others.

Algorithm 4 Adaptive Threshold Calculation

Input: Importance scores $I(L)$ for all layers
Output: Adaptive threshold τ
Compute mean $\mu = \text{mean}(I(L))$
Compute standard deviation $\sigma = \text{std}(I(L))$
Set hyperparameter $\alpha = 1.5$
Compute threshold:

$$\tau = \mu + \alpha \cdot \sigma$$

return τ

B.3 Dynamic GPU-CPU Allocation

After calculating the adaptive threshold, the next step in the LOGCA method is to dynamically allocate each layer to either the GPU or the CPU. This allocation is based on the importance score of each layer relative to the adaptive threshold τ. If a layer's importance score exceeds the threshold, it is considered important enough to benefit from the higher computational power and parallelism of the GPU. As a result, the layer is allocated to the GPU. On the other hand, if a layer's importance score is below the threshold, it is allocated to the CPU, as it is less critical for the model's final output and can be processed more efficiently on a less powerful processor.

This dynamic allocation scheme allows for efficient resource utilization, ensuring that the most computationally demanding layers receive the processing power they require, while minimizing GPU memory usage by offloading less important layers to the CPU. By selectively allocating layers to the GPU, we optimize both memory usage and computational efficiency, leading to better overall performance, especially in resource-constrained environments where GPU memory is limited.

B.4 Memory and Computational Efficiency Optimization

In this step, the goal is to further optimize the memory and computational efficiency of the system. The key idea is to offload less critical layers to the CPU, which helps reduce the memory pressure on the GPU. This offloading strategy ensures that only the most critical layers that require high computational power are assigned to the GPU, allowing for better GPU utilization and the ability to process larger batch sizes. By reducing the memory footprint on the GPU, we allow for more

Algorithm 5 Dynamic GPU-CPU Allocation

Input: Importance scores $I(L)$, adaptive threshold τ
Output: Resource allocation for each layer (GPU or CPU)
for each layer L **do**
    if $I(L) > \tau$ **then**
        Allocate layer L to GPU
    else
        Allocate layer L to CPU
    end if
end for
return Allocation for each layer

efficient memory usage, ensuring that the GPU can handle more important tasks without running into memory bottlenecks.

The optimization process involves calculating the memory usage for each layer based on its size and ensuring that layers are distributed appropriately between the GPU and CPU. This balancing act ensures that the memory usage on the GPU does not exceed its available capacity while maximizing the computational performance by keeping the critical layers on the GPU. The system dynamically adjusts the resource usage based on the available memory on both devices, ensuring optimal performance.

Algorithm 6 Memory and Computational Efficiency Optimization

Input: Resource allocation for each layer (GPU/CPU), model, hardware configuration
Output: Optimized memory and computation efficiency
Initialize GPU memory usage $M_{GPU} = 0$ and CPU memory usage $M_{CPU} = 0$
for each layer L **do**
    if Layer L allocated to GPU **then**
        Increment GPU memory usage: $M_{GPU} +=$ Memory(L)
    else
        Increment CPU memory usage: $M_{CPU} +=$ Memory(L)
    end if
end for
Optimize memory usage by offloading non-critical layers to CPU
Adjust resource usage based on available GPU/CPU memory
return Optimized memory allocation

B.5 Dynamic Layer Scheduling

As the system continues to run and process data, the resource usage may change, leading to shifts in memory and computational load. To handle such variations, LOGCA implements a dynamic layer scheduling mechanism. This mechanism constantly monitors the system's resource usage in real-time and adjusts the allocation of layers to either the GPU or CPU based on the current system load. If the GPU memory usage exceeds a certain threshold, the system may decide to offload less critical layers to the CPU, thus freeing up memory and computational resources on the GPU. Similarly, if the CPU is underutilized, more critical layers can be moved to the GPU to maximize the computational power of the system.

This dynamic scheduling ensures that the resource allocation adapts to the current system load, optimizing performance without wasting computational resources. The system remains flexible and responsive, allowing the model to run efficiently even under fluctuating workloads.

C Hyperparameters for LOGCA Method

In the LOGCA method, several hyperparameters are crucial for controlling the allocation of layers between the GPU and CPU, as well as the overall performance of the model. These hyperparameters

Algorithm 7 Dynamic Layer Scheduling

Input: Real-time system resource usage (GPU and CPU), importance scores $I(L)$, current resource allocation
Output: Updated resource allocation
for each layer L **do**
    if Current GPU memory usage M_{GPU} exceeds threshold τ_{GPU} **then**
        if $I(L) < \tau_{dynamic}$ **then**
            Move layer L from GPU to CPU
        end if
    end if
    if Current CPU memory usage M_{CPU} exceeds threshold τ_{CPU} **then**
        if $I(L) > \tau_{dynamic}$ **then**
            Move layer L from CPU to GPU
        end if
    end if
end for
return Updated resource allocation

help in determining the sensitivity of layer importance, resource allocation strategies, and memory management during training and inference. Below is a detailed list of the key hyperparameters used in LOGCA, along with their typical values that were found to work well in experiments.

- **Learning Rate (η)**: The learning rate controls how much to adjust the model's weights with respect to the loss gradient. A typical value for deep learning models is:

  $$\eta = 1 \times 10^{-5}$$

  This is small enough to ensure stable convergence, particularly in fine-tuning pre-trained models.

- **Batch Size (B_{size})**: The batch size determines the number of training samples processed before the model's weights are updated. Smaller batch sizes allow more frequent updates but can increase training time. The chosen value for LOGCA is:

  $$B_{size} = 32$$

  This is a commonly used batch size for training on moderate hardware setups.

- **Adaptive Threshold Sensitivity (α)**: This hyperparameter controls how strictly the dynamic threshold is applied when deciding if a layer should be allocated to the GPU. A larger value makes the threshold stricter, so fewer layers will be assigned to the GPU. A reasonable value for α is:

  $$\alpha = 1.5$$

  This setting strikes a balance between layer importance and GPU memory usage, ensuring effective resource allocation.

- **Layer Importance Threshold (τ)**: This is the threshold value derived from the mean and standard deviation of layer importance scores. Layers with importance scores greater than τ are allocated to the GPU, while others are assigned to the CPU. A typical setting for τ is:

  $$\tau = \mu + \alpha \cdot \sigma$$

  where μ is the mean and σ is the standard deviation of the importance scores across layers.

- **Memory Allocation Thresholds (τ_{GPU}, τ_{CPU})**: These thresholds control the allocation of memory between the GPU and CPU. If the GPU or CPU exceeds these thresholds, less important layers are moved to the other device to optimize memory usage. The values for these thresholds are typically:

  $$\tau_{GPU} = 90\% \quad \text{and} \quad \tau_{CPU} = 80\%$$

  This means that the GPU will start offloading layers to the CPU when it is 90% full, and similarly, the CPU will handle additional layers when its memory usage exceeds 80%.

- **Number of Layers (N_{layers})**: The total number of layers in the model. For typical large language models or vision networks, the number of layers could range from tens to hundreds. For a model with moderate complexity, we use:

$$N_{layers} = 12$$

This number is based on common architectures such as Transformer-based models.

- **GPU Memory Usage (M_{GPU})**: The maximum amount of memory that the GPU is allowed to use during inference. This is a critical parameter for ensuring that the GPU memory is not exhausted by too many layers. A typical limit for GPU memory usage is:

$$M_{GPU} = 16\,\text{GB}$$

This is a reasonable memory budget for high-end GPUs (e.g., NVIDIA RTX 3090).

- **CPU Memory Usage (M_{CPU})**: Similar to GPU memory, this hyperparameter sets the memory usage limit for the CPU. We set the following value for CPU memory usage:

$$M_{CPU} = 16\,\text{GB}$$

The CPU is typically more flexible in memory usage, and thus a higher limit is acceptable.

- **Dynamic Scheduling Threshold ($\tau_{dynamic}$)**: This is a real-time threshold that adjusts based on the system's memory and computational load during inference. The value is dynamically calculated and used to decide if layers should be moved between the CPU and GPU based on current usage patterns.

- **Importance Score Averaging ($I(L)$)**: The averaging technique for computing importance scores across the layers is defined as:

$$I(L) = \frac{1}{N_{layers}} \sum_{i=1}^{N_{layers}} d_{weighted}(L, L_i)$$

where $d_{weighted}(L, L_i)$ is the weighted angular distance between layers L and L_i. This method ensures that each layer's contribution is averaged out in a meaningful way, taking into account both its activation strength and its relation to other layers.

- **Epochs (E)**: The number of training epochs. A common value for models that require extensive training, such as large-scale language models, is:

$$E = 10$$

This value is based on the observed convergence of the model during training with a reasonable learning rate and batch size.

- **Weight Decay (λ)**: Weight decay is a regularization technique used to prevent overfitting by penalizing large weights. A typical value for weight decay in LOGCA models is:

$$\lambda = 0.01$$

This value helps maintain model generalization without overfitting to training data.

- **Dropout Rate (p)**: Dropout is used during training to randomly drop neurons, which helps prevent overfitting. The value for the dropout rate is often set to:

$$p = 0.1$$

This means that 10% of neurons are randomly dropped out during each training step.

D List of Symbols

Table 7: List of Symbols

Symbol	Description
$L_1, L_2, ..., L_N$	Layers of the neural network.
$A(L)$	Activation values of layer L.
$w(L)_i$	Weighted activation of the i-th neuron in layer L.
$A(L)_{weighted}$	Weighted activation vector for layer L.
$\langle A(L)_{weighted}, A(L')_{weighted} \rangle$	Dot product of weighted activation vectors.
$\|A(L)_{weighted}\|_2$	L2-norm of the weighted activation vector.
$d_{weighted}(L, L')$	Weighted angular distance between layers L and L'.
$I(L)$	Importance score of layer L.
μ	Mean of importance scores.
σ	Standard deviation of importance scores.
α	Hyperparameter controlling threshold sensitivity.
τ	Adaptive threshold for dynamic allocation.
M_{GPU}	Memory usage on the GPU.
M_{CPU}	Memory usage on the CPU.
τ_{GPU}	GPU memory usage threshold.
τ_{CPU}	CPU memory usage threshold.
$\tau_{dynamic}$	Dynamic threshold for real-time scheduling.
μ_I	Mean importance score used for scheduling.
σ_I	Standard deviation of importance scores for dynamic adjustment.
R_{GPU}	Computational resources required by the GPU.
R_{CPU}	Computational resources required by the CPU.
B_{size}	Batch size for processing.
C_{GPU}	Computational capacity of the GPU.
C_{CPU}	Computational capacity of the CPU.
T_{total}	Total processing time for training or inference.
$P(L)$	Processing time for layer L.
N_{layers}	Total number of layers in the model.
$L_{critical}$	Set of critical layers allocated to the GPU.
$L_{non-critical}$	Set of non-critical layers allocated to the CPU.
T_{GPU}	Time taken by the GPU to process layers.
T_{CPU}	Time taken by the CPU to process layers.
util_{GPU}	GPU utilization during processing.
util_{CPU}	CPU utilization during processing.