FIDDLER: CPU-GPU ORCHESTRATION FOR FAST INFERENCE OF MIXTURE-OF-EXPERTS MODELS

Keisuke Kamahori, Yile Gu, Kan Zhu, Baris Kasikci
University of Washington
{kamahori,yilegu,kanzhu,baris}@cs.washington.edu

ABSTRACT

Large Language Models (LLMs) based on Mixture-of-Experts (MoE) architecture are showing promising performance on various tasks. However, running them on resource-constrained settings, where GPU memory resources are not abundant, is challenging due to huge model sizes. Existing systems that offload model weights to CPU memory suffer from the significant overhead of frequently moving data between CPU and GPU. In this paper, we propose *Fiddler*, a resource-efficient inference engine with CPU-GPU orchestration for MoE models. The key idea of *Fiddler* is to use the computation ability of the CPU to minimize the data movement between the CPU and GPU. Our evaluation shows that *Fiddler* can run the uncompressed Mixtral-8x7B model, which exceeds 90GB in parameters, to generate over 3 tokens per second on a single GPU with 24GB memory, showing an order of magnitude improvement over existing methods. We are going to release the code of *Fiddler* as open-source software.

1 INTRODUCTION

Large Language Models (LLMs) based on Mixture-of-Experts (MoE) architectures are showing remarkable performance on various tasks (Du et al. (2022); Fedus et al. (2022); Jiang et al. (2024)). By activating a subset of experts inside feed-forward layers with a gating mechanism, such models scale up model size and improve model performance with a small computation overhead.

There has been growing interest in self-hosting these LLMs in local settings (Giacinto (2023); Anand et al. (2023); Song et al. (2023)) for enhanced privacy (Martínez Toro et al. (2023)) and customization of LLMs on proprietary or personal data (Lyu et al. (2023)). The ability to run these models in resource-constrained settings democratizes state-of-the-art LLM technologies, especially for those with difficulty accessing high-end GPU computation resources.

Despite MoE models' superior performance, running them in low-resource settings with limited GPU memory resources is challenging. On the one hand, the huge parameter size of MoE models makes it difficult to store all the weights on GPU memory. On the other hand, model compression techniques like quantization and sparsification come with degradation of model quality (Frantar & Alistarh (2023); Eliseev & Mazur (2023)). For example, the Mixtral-8x7B (Jiang et al. (2024)) model takes more than 90GB of memory for model weights in 16-bit precision, which is beyond the reach of most consumer GPUs. To fit this model into a single GPU with 24GB memory, one would need to compress it to 4-bit per parameter or smaller, which comes with significant accuracy degradation (Eliseev & Mazur (2023)).

In this paper, we tackle the challenge of efficiently deploying uncompressed MoE models in a local setting (*i.e.*, latency-oriented and single-batch) with a single GPU. It is particularly interesting to consider MoE models for this setting because of their sparsity; *i.e.*, the amount of computation in relation to the parameter size is smaller than the dense counterparts. In this setting, investing in additional GPUs is not cost-effective since GPUs have limited memory capacity despite their high compute throughput. This situation is exacerbated by the fact that the scalability of expert components within MoE models is virtually unbounded; for instance, Switch Transformers have been demonstrated to effectively incorporate thousands of experts (Fedus et al. (2022)). As a result, provisioning of sufficient GPU resources becomes a significant challenge.

Previous works proposed to offload expert weights to CPU memory (Eliseev & Mazur (2023); Xue et al. (2024b)), which is usually more abundant than GPU memory. In those methods, GPU memory holds only a subset of expert weights, and the expert weights will be brought from CPU memory to GPU memory when they are required for computation. While these works overcome the limitation of memory capacity, there is a significant runtime overhead due to the frequent copying of expert weights between CPU and GPU over the low bandwidth PCIe connection.

In this work, we instead propose to utilize the CPU computation resources in addition to the CPU memory resources for MoE model inference. We design *Fiddler*, a resource-efficient inference system with CPU-GPU orchestration for MoE models. The key insight is that, in terms of latency, it is better to execute expert layers on the CPUs than to load the expert weights from CPU memory to GPU memory, especially when the batch size is small. This is particularly suitable for local deployment of MoE models, where latency is critical, and the model needs to process a single request at a time. *Fiddler* is able to take advantage of CPU computational resources to minimize the data movement between the CPU and GPU. *Fiddler* can run the uncompressed Mixtral-8x7B model, which has more than 90GB of parameters, to generate over 3 tokens per second on a single GPU with 24GB memory. Compared to existing offloading methods, *Fiddler* improves the single-batch inference latency by 8.2 times on Quadro RTX 6000 and 10.1 times on L4 GPU on average across different input/output lengths.

2 Related Work

2.1 Mixture-of-Experts

MoE models have been demonstrating promising performance in the era of LLMs (Rajbhandari et al. (2022); Du et al. (2022); Fedus et al. (2022); Jiang et al. (2024); Xue et al. (2024a); Dai et al. (2024)). Different from the original dense language model, MoE models introduce sparsity to the feed-forward layer with experts and employ a gating mechanism. Each MoE layer consists of a number of expert layers that have the same dimensions as the feed-forward layer, and a gating network decides which expert layers will be activated for each input. Although the number of experts in an MoE layer could be up to thousands, only a few experts will be activated by the gating network during training or inference (Fedus et al. (2022)).

2.2 Efficient Deployment of MoE Models

Efficiently serving MoE models is challenging due to the large model size, especially in resource-constrained settings. One approach to running large models in such an environment is offloading, where the subset of parameters are stored in CPU memory instead of GPU memory (Sheng et al. (2023)). For MoE models, previous works attempted to offload expert weights with caching or prefetching mechanisms (Eliseev & Mazur (2023); Xue et al. (2024b)). However, these approaches cause significant latency overhead due to the frequent copying of expert weights between CPU and GPU over the low bandwidth PCIe connection, making them suboptimal for local settings where latency is critical for user experience. *Fiddler* overcomes this challenge by utilizing the computation resource of CPUs.

Another direction is model compression, such as quantization (Frantar & Alistarh (2023); Zhao et al. (2023)) or sparsification (Alizadeh et al. (2023)). While those techniques reduce the model size and improve inference efficiency, they come with degraded output quality of models, especially when trying to fit large models like Mixtral-8x7B to a GPU with small memory capacity (Eliseev & Mazur (2023)). Recently, Song et al. (2023) attempted to exploit the sparsity of LLMs for faster model inference with CPU offloading. However, this approach requires the model to use the Rectified Linear Units (ReLU) function for the nonlinear activation. Converting non-ReLU models, which is common in state-of-the-art LLMs, to ReLU models requires additional training and causes performance degradation (Mirzadeh et al. (2023); SparseLLM). Mixtral-8x7B uses Sigmoid Linear Units (SiLU) function (Elfwing et al. (2018)), and only a small portion of values are close to zero, making it difficult to exploit the sparsity (we discuss more detail in Appendix A). *Fiddler* can achieve better performance without modifying the model structure or accuracy. Also, *Fiddler* is orthogonal to the compression techniques and these optimizations could be applied on top of *Fiddler*.

3 DESIGN

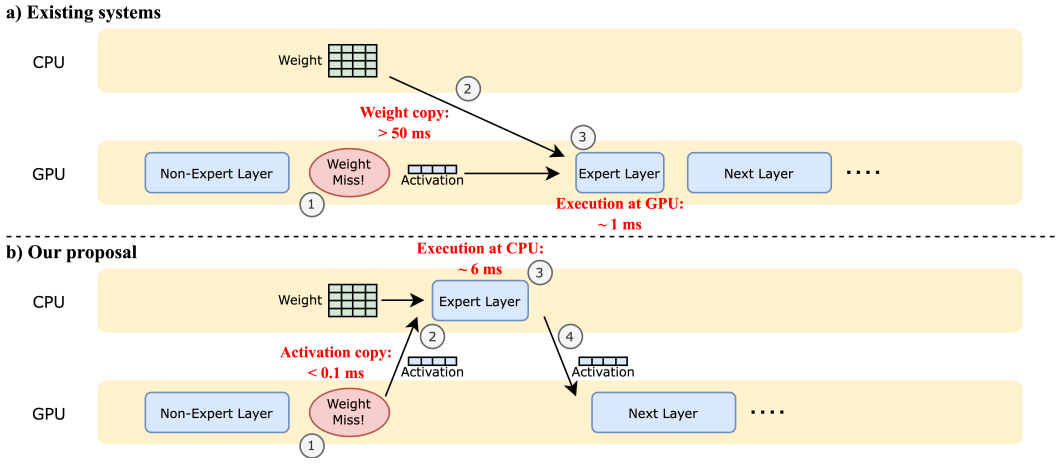

Figure 1: Main idea of *Fiddler*. a) In existing offloading systems, expert weights are copied from the CPU to the GPU, and the computation happens on the GPU. b) In *Fiddler*, only the activations are copied between the CPU and GPU, and the computation happens on the CPU, drastically reducing the latency for CPU-GPU communication.

3.1 OVERVIEW

Figure 1 illustrates the key idea of *Fiddler*, comparing the two different approaches for CPU offloading in the case of single-batch inference of MoE models. Existing offloading systems (Figure 1 a)) only use the memory resources in the CPU. The computation mainly happens on the GPU. When some expert weights are missing on the GPU memory (①), they are copied from the CPU memory to the GPU memory (②), and then the GPU executes the expert layer (③). Although the execution on the GPU is faster than the CPU, the data movement causes significant overhead. For instance, each expert of the Mixtral-8x7B model has more than 300MB of parameters in 16-bit precision. We observe a latency of around 50ms to copy data from the CPU memory to the GPU memory for Quadro RTX 6000 or L4 GPUs (more microbenchmark results are discussed in Appendix B).

On the other hand, our proposed method (Figure 1 b)) uses CPU computation resources in addition to memory resources. In *Fiddler*, when some expert weights are missing on the GPU memory (①), we copy the activation values from the GPU memory to the CPU memory (②) instead of copying the weights. Then, the computation of the expert layer happens on the CPU (③), and the output activation is copied back to the GPU after the computation finishes (④). The benefit of this approach is that we can drastically reduce the latency of CPU-GPU communication because the size of activations (batch size $\times$ 4096 for the Mixtral-8x7B) is significantly smaller than the weight size (3 matrices with size 4096×14336 per expert for the Mixtral-8x7B) for a small batch size. Despite the slower computation at the CPU compared to the GPU, the weight copying process is even more time-consuming, making the approach outlined in Figure 1 b) superior to a). Detailed data is available in Appendix B.

3.2 ALGORITHM

Based on the idea described in the previous section, *Fiddler* serves MoE models in the following way.

Initialization. Before starting the inference process, *Fiddler* distributes the model weights between the CPU and GPU memory. First, the weights of non-expert layers are placed on GPU memory because they are used for every token, irrespective of expert choice. The size of non-expert layers is usually not big (less than 2 billion parameters for the Mixtral-8x7B model), and we assume they fit in the GPU memory in this paper. Next, we put a subset of expert layers into the GPU memory.

For this, we select as many experts as the memory capacity permits in order of popularity so that we can maximize the hit rate, *i.e.*, the likelihood an expert's weight is in GPU memory. We determine the popular experts based on the profile of expert selection using calibration data. We assume this method is enough as the expert selection is known to be based on token characteristics, and the popularity of experts is universal across different input domains (Jiang et al. (2024); Xue et al. (2024a)). Appendix C discusses expert selection in more detail.

Decode Stage. In the decode stage, only one token is processed, so each expert gets at most one token at a time for the single batch inference. In this case, offloading computation to the CPU is always faster than bringing the weight to the GPU as discussed in the previous section. Therefore, if the expert weight to be used is missing on the GPU memory, we take the approach as shown in Figure 1 b) to minimize the latency.

Prefill Stage. In the prefill stage, multiple tokens are processed simultaneously even for a single batch inference, so some experts can get multiple tokens as inputs. For this case, we need to consider the different batching effects of GPU and CPU. When executing an expert on a GPU, the latency is dominated by the time required to transfer the expert's weight from the CPU to GPU memory. Hence, this latency remains largely unchanged regardless of the input size. In contrast, executing an expert layer on the CPU exhibits different behavior. As the number of input tokens increases, so does the latency, while the time to copy activation from the GPU is negligible (less than 1% of the total latency; see Appendix B for more details). To find the optimal way to process the prefill stage, we adopt a model where the GPU execution time is considered constant, while the CPU execution time is assumed to increase linearly with the number of input tokens. We then find the best expert execution configuration by solving the following problem

$$\underset{\text{cpu\_expert},\text{gpu\_expert}}{\arg\min} \max\left(\sum_{i\in\text{cpu\_expert}} (\text{n\_input}_i \times \text{latency}_{\text{cpu}}), \sum_{i\in\text{gpu\_expert}} ((1-\text{is\_on\_gpu}_i) \times \text{latency}_{\text{gpu}})\right)$$

where $\text{latency}_{\text{cpu}}$ and $\text{latency}_{\text{gpu}}$ denotes the latency of processing one input at CPU/GPU, n_input_i denotes the number of inputs that i-th expert gets, and is_on_gpu_i is a variable indicating whether the i-th expert is on GPU memory (1 if yes and 0 otherwise). The cpu_expert and gpu_expert denote the (mutually exclusive) sets of experts to be processed at CPU and GPU, respectively, and their union is the set of all experts. We take the maximum of two values because the CPU and GPU can run in parallel, and the latency is dominated by whichever part is longer.

4 Evaluation

4.1 Setup

Model and Data. We use the Mixtral-8x7B model with 16-bit precision for the evaluation. For the evaluation and calibration data, we use the ShareGPT (ShareGPT) dataset, a dataset of conversations between humans and chatbots, to model the realistic behavior of expert selection. We pick the subset of conversations randomly. We implement *Fiddler* on top of PyTorch (Paszke et al. (2019)).

Environments. We evaluate *Fiddler* on two environments as shown in Table 1. Each environment lacks enough GPU memory to store all the model weights. The "Number of Experts on GPU" row shows the maximum number of experts that could fit on the GPU memory (out of 32 layers $\times$ 8 experts/layer $=$ 256 total experts).

Table 1: Evaluation setups

	Environment 1	Environment 2
GPU	Quadro RTX 6000 (NVIDIA (b))	L4 (NVIDIA (a))
GPU Memory	24576MiB	23034MiB
PCIe	Gen3 x16 (32GB/s)	Gen4 x16 (64GB/s)
CPU	Intel Skylake (48 core, 2.60GHz)	Intel Cascade Lake (32 core, 2.20GHz)
Number of Experts on GPU	$56/256$	$52/256$

Baselines. For baselines, we evaluate DeepSpeed-MII (Microsoft) and Mixtral-Offloading (Eliseev & Mazur (2023)). For DeepSpeed-MII, we enable ZeRO-Infinity optimization (Rajbhandari et al. (2021)) to offload model weights to CPU memory. We choose to evaluate performance in non-persistent pipeline mode as we target local settings instead of client-server applications. We enable `pin_memory` in the configuration to use paged-locked CPU memory, which could boost the performance of the prefill stage. Mixtral-Offloading only supports a quantized version of the Mixtral-8x7B model by default. For a fair comparison, we extend Mixtral-Offloading to support running the original version of the model with 16-bit precision. We set `offload_per_layer` parameter to 7 as this is the only configuration available to fit the unquantized model weights in 24GB GPU memory.

Metrics. We evaluate the performance of *Fiddler* in a local deployment scenario, *i.e.*, the latency for single batch inference, with different lengths of input and output tokens. For the evaluation with N input tokens, we randomly select samples from ShareGPT that have N tokens or more of prompt and use the initial N tokens. We measure the average number of tokens generated per second as calculated by the ratio of the number of output tokens to the end-to-end latency (including both prefill and decode stages). We choose the input length from [16, 32, 64, 128] and the output length from [16, 32, 64, 128, 256, 512]. For *Fiddler*, we show the average of 10 runs, and for the baseline methods, we show the average of 3 runs due to longer execution time.

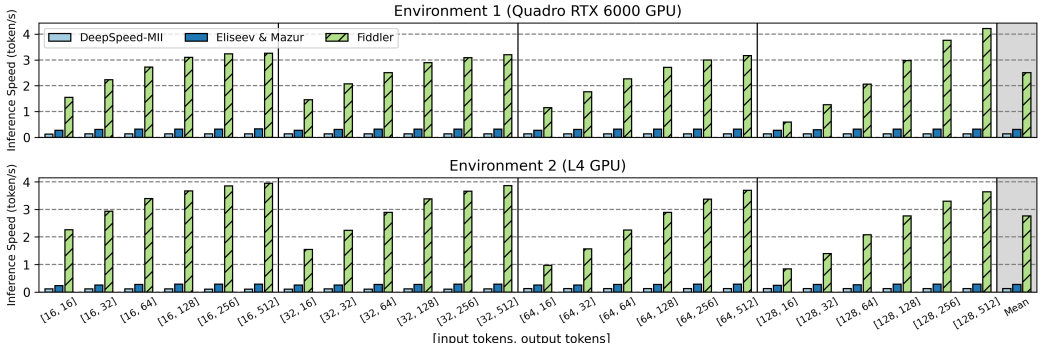

Figure 2: The performance of *Fiddler* and baseline methods measured by the number of tokens generated per second, with 24 different configurations of input/output length. The rightmost set of bars shows the average of 24 configurations.

4.2 Results

Figure 2 shows the end-to-end performance of three methods in two environments. Overall, *Fiddler* outperforms DeepSpeed-MII and Eliseev & Mazur (2023) for all the input and output lengths. The performance measured by tokens per second improves with longer output lengths for the same input length because the latency of the prefill stage is amortized for longer output. On average, *Fiddler* is faster than Eliseev & Mazur (2023) by 8.2 times for Environment 1 and by 10.1 times for Environment 2. Compared with DeepSpeed-MII, *Fiddler* is faster by 19.4 times at Environment 1 and by 22.5 times at Environment 2.

5 Conclusion

This paper proposes *Fiddler*, a resource-efficient inference engine with CPU-GPU orchestration for MoE model deployment in local settings. In addition to CPU memory, *Fiddler* is able to utilize CPU computation resources during inference. The performance benefits of *Fiddler* come from the observation that copying expert weights from CPU to GPU leads to larger overhead than executing experts on CPU when the batch size is sufficiently small. *Fiddler* is evaluated on Quadro RTX 6000 and L4 GPU and achieves 8.2 times and 10.1 times speedup on single-batch inference latency compared to baselines, respectively. *Fiddler* is an important step towards fast inference of large MoE models on resource-constrained settings by fully utilizing heterogeneous hardware resources.

ACKNOWLEDGMENTS

This work was supported by the PRISM Research Center, a JUMP Center cosponsored by SRC and DARPA. Keisuke Kamahori is supported by a Ph.D. Scholarship from Toyota Physical and Chemical Research Institute.

REFERENCES

Keivan Alizadeh, Iman Mirzadeh, Dmitry Belenko, Karen Khatamifard, Minsik Cho, Carlo C Del Mundo, Mohammad Rastegari, and Mehrdad Farajtabar. Llm in a flash: Efficient large language model inference with limited memory. *arXiv preprint arXiv:2312.11514*, 2023.

Yuvanesh Anand, Zach Nussbaum, Brandon Duderstadt, Benjamin Schmidt, and Andriy Mulyar. Gpt4all: Training an assistant-style chatbot with large scale data distillation from gpt-3.5-turbo. `https://github.com/nomic-ai/gpt4all`, 2023.

Damai Dai, Chengqi Deng, Chenggang Zhao, R. X. Xu, Huazuo Gao, Deli Chen, Jiashi Li, Wangding Zeng, Xingkai Yu, Y. Wu, Zhenda Xie, Y. K. Li, Panpan Huang, Fuli Luo, Chong Ruan, Zhifang Sui, and Wenfeng Liang. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models, 2024.

Nan Du, Yanping Huang, Andrew M Dai, Simon Tong, Dmitry Lepikhin, Yuanzhong Xu, Maxim Krikun, Yanqi Zhou, Adams Wei Yu, Orhan Firat, et al. Glam: Efficient scaling of language models with mixture-of-experts. In *International Conference on Machine Learning*, pp. 5547–5569. PMLR, 2022.

Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural networks*, 107:3–11, 2018.

Artyom Eliseev and Denis Mazur. Fast inference of mixture-of-experts language models with offloading. *arXiv preprint arXiv:2312.17238*, 2023.

William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *The Journal of Machine Learning Research*, 23(1): 5232–5270, 2022.

Elias Frantar and Dan Alistarh. Qmoe: Practical sub-1-bit compression of trillion-parameter models. *arXiv preprint arXiv:2310.16795*, 2023.

Ettore Di Giacinto. Localai. `https://github.com/mudler/LocalAI`, 2023.

Albert Q Jiang, Alexandre Sablayrolles, Antoine Roux, Arthur Mensch, Blanche Savary, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Emma Bou Hanna, Florian Bressand, et al. Mixtral of experts. *arXiv preprint arXiv:2401.04088*, 2024.

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

Hanjia Lyu, Song Jiang, Hanqing Zeng, Yinglong Xia, and Jiebo Luo. Llm-rec: Personalized recommendation via prompting large language models. *arXiv preprint arXiv:2307.15780*, 2023.

Iván Martínez Toro, Daniel Gallego Vico, and Pablo Orgaz. PrivateGPT, May 2023. URL `https://github.com/imartinez/privateGPT`.

Microsoft. Deepspeed-mii. `https://github.com/microsoft/DeepSpeed-MII`.

Iman Mirzadeh, Keivan Alizadeh, Sachin Mehta, Carlo C Del Mundo, Oncel Tuzel, Golnoosh Samei, Mohammad Rastegari, and Mehrdad Farajtabar. Relu strikes back: Exploiting activation sparsity in large language models. *arXiv preprint arXiv:2310.04564*, 2023.

NVIDIA. Nvidia l4 gpu accelerator. `https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/l4/PB-11316-001_v01.pdf`, a.

NVIDIA. Nvidia quadro rtx 6000 pcie server card. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/quadro-product-literature/NVIDIA-Quadro-RTX-6000-PCIe-Server-Card-PB-FINAL-1219.pdf, b.

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

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning, 2021.

Samyam Rajbhandari, Conglong Li, Zhewei Yao, Minjia Zhang, Reza Yazdani Aminabadi, Ammar Ahmad Awan, Jeff Rasley, and Yuxiong He. Deepspeed-moe: Advancing mixture-of-experts inference and training to power next-generation ai scale. In *International Conference on Machine Learning*, pp. 18332–18346. PMLR, 2022.

ShareGPT. Sharegpt. https://huggingface.co/datasets/anon8231489123/ShareGPT_Vicuna_unfiltered.

Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E Gonzalez, et al. High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865*, 2023.

Yixin Song, Zeyu Mi, Haotong Xie, and Haibo Chen. Powerinfer: Fast large language model serving with a consumer-grade gpu. *arXiv preprint arXiv:2312.12456*, 2023.

SparseLLM. Relullama-70b. https://huggingface.co/SparseLLM/ReluLLaMA-70B.

Fuzhao Xue, Zian Zheng, Yao Fu, Jinjie Ni, Zangwei Zheng, Wangchunshu Zhou, and Yang You. Openmoe: An early effort on open mixture-of-experts language models. 2024a.

Leyang Xue, Yao Fu, Zhan Lu, Luo Mai, and Mahesh Marina. Moe-infinity: Activation-aware expert offloading for efficient moe serving. *arXiv preprint arXiv:2401.14361*, 2024b.

Susan Zhang, Stephen Roller, Naman Goyal, Mikel Artetxe, Moya Chen, Shuohui Chen, Christopher Dewan, Mona Diab, Xian Li, Xi Victoria Lin, et al. Opt: Open pre-trained transformer language models. *arXiv preprint arXiv:2205.01068*, 2022.

Yilong Zhao, Chien-Yu Lin, Kan Zhu, Zihao Ye, Lequn Chen, Size Zheng, Luis Ceze, Arvind Krishnamurthy, Tianqi Chen, and Baris Kasikci. Atom: Low-bit quantization for efficient and accurate llm serving. *arXiv preprint arXiv:2310.19102*, 2023.

A Sparsity Analysis

This section analyzes sparsity within Mixtral-8x7B models, illustrating the challenges of applying conventional sparsity-based optimization techniques from prior works (Song et al. (2023); Alizadeh et al. (2023)). These methods primarily target LLMs that incorporate the ReLU activation function, leveraging its characteristic of nullifying negative inputs to prune channels with consistently zero outputs. This approach benefits from the binary nature of ReLU's output—either zero or positive—allowing for straightforward identification and elimination of inactive channels, thereby optimizing computational efficiency without losing crucial information.

On the other hand, state-of-the-art MoE models often employ alternative activation functions, which makes it challenging to directly apply these sparsity-exploiting strategies. For example, both Mixtral-8x7B and DeepSeekMoE (Dai et al. (2024)) use SiLU as the activation function. Unlike ReLU, SiLU does not produce a straightforward threshold of zero for pruning, so there is a need for a more sophisticated approach to leverage sparsity. Pruning channels that are not close enough to zero could detrimentally impact the model's performance.

Table 2 presents an analysis of the absolute values after the SiLU function across the layers of the Mixtral-8x7B model. This analysis is based on data derived from 100 samples within the ShareGPT

dataset (ShareGPT), without making distinctions between different experts in identical layers. The data shows a generally low occurrence of values close to zero. Specifically, for all layers, the proportion of channels with absolute values below 0.001 is smaller than 2%, and for 30 out of the 32 layers, this number even falls below 1%. Furthermore, in 28 out of 32 layers, fewer than 5% of the values are smaller than 0.01, and in 24 layers, fewer than 30% of the values are under 0.1. Despite variations across layers, these results collectively suggest a substantial challenge in harnessing sparsity within this model with approaches of previous works. In contrast, Liu et al. (2023) reported that over 90% of values after the ReLU function is zero for the MLP layers of OPT models (Zhang et al. (2022)). Utilizing sparsity within models like Mixtral-8x7B to speed up inference with tolerable quality loss would be an interesting direction for future research.

Table 2: Distribution of absolute values after SiLU function of Mixtral-8x7B model across all layers. Each cell displays the percentage of values whose absolute value is below a specified threshold.

Layer	< 0.001	< 0.01	< 0.1	< 1.0
1	1.75	17.17	93.89	100.00
2	1.21	11.95	85.08	100.00
3	0.92	9.10	74.80	99.99
4	0.71	7.06	63.69	99.99
5	0.50	5.00	49.67	99.95
6	0.41	4.08	41.60	99.93
7	0.36	3.56	36.66	99.91
8	0.30	2.97	31.04	99.88
9	0.29	2.90	29.96	99.86
10	0.27	2.73	28.25	99.80
11	0.24	2.37	24.65	99.74
12	0.24	2.43	25.15	99.69
13	0.24	2.36	24.55	99.65
14	0.22	2.22	23.05	99.53
15	0.20	2.02	21.03	99.32
16	0.18	1.78	18.61	99.14
17	0.15	1.53	16.14	98.91
18	0.15	1.50	15.86	98.58
19	0.13	1.33	14.24	98.15
20	0.12	1.19	12.94	97.95
21	0.11	1.09	12.04	97.86
22	0.10	0.97	11.09	97.96
23	0.10	1.02	11.58	97.61
24	0.10	1.02	11.72	97.36
25	0.09	0.95	11.55	97.34
26	0.10	0.95	11.91	97.05
27	0.09	0.95	12.19	96.72
28	0.09	0.89	12.28	96.76
29	0.08	0.86	13.89	95.86
30	0.09	1.03	15.16	94.02
31	0.12	1.37	16.65	92.12
32	0.36	2.73	20.27	89.64

B MICROBENCHMARKS

In this section, we show the results of microbenchmarks. Figure 3 shows the latency of following workloads:

- `W copy`: Copying weight of one expert from CPU to GPU
- `A copy`: Copying one activation from GPU to CPU
- `GPU N`: Executing one expert at GPU with batch size N (excluding the time to copy weight from CPU)
- `CPU N`: Executing one expert at CPU with batch size N

For each value, we execute the workload 32 times (one per layer of Mixtral-8x7B) and show the average and standard deviation.

When executing tasks on a GPU, the latency of transferring weights from the CPU memory to GPU memory is approximately 10 times longer than the time spent on actual computation. The computation latency at the GPU is mostly independent of batch size. There is an exception when the batch size is 1 at Environment 1 because PyTorch employs a different implementation for single-batch and multi-batch scenarios, but the difference is insignificant (approximately 10%) compared to the overall latency which includes weight transfer. Hence, we model the GPU latency as constant in Section 3.2.

On the CPU, the execution latency tends to increase linearly with the size of the input batch. However, the time required to transfer activations is negligible (less than 1% of single batch latency). Given this minimal impact, our model in Section 3.2 assumes that CPU latency exhibits a linear relationship with the number of inputs.

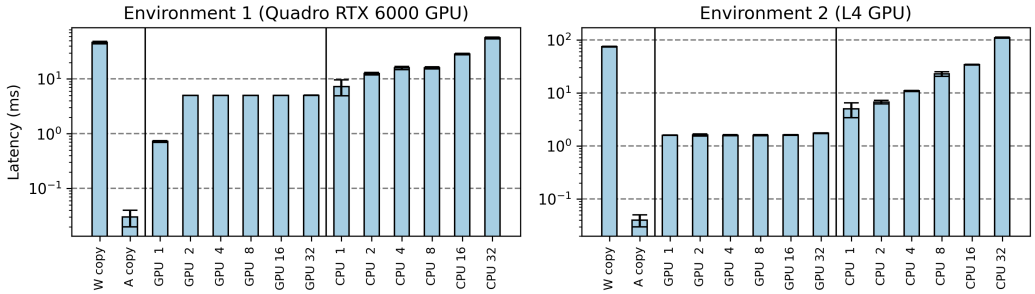

Figure 3: The results of microbenchmarks, measuring the latency to copy weight or activation between the CPU and GPU, and executing an expert layer at CPU or GPU with different batch sizes. We use a log scale for the y-axis.

C EXPERT POPULARITY

Figure 4 shows the heat map about the popularity of expert selection within the Mixtral-8x7B model. Similar to Appendix A, we collect the profile by running inference on random samples from the ShareGPT dataset and counting the number of tokens routed to each expert. The color intensity of each cell indicates the frequency of expert selection (which is equivalent to the number of tokens that activated the expert). The value of the most popular expert has been normalized to 1, with the popularity of other experts expressed as a ratio relative to that.

Across 256 experts, the average value is 0.71, with a standard deviation of 0.08, a 25th percentile of 0.67, and a 75th percentile of 0.76. Despite a minimum value of 0.22, only 15 experts have values below 0.6, and 27 experts exceed 0.8, indicating a relatively balanced distribution.

In Environment 1, selecting the 56 most popular experts out of 256 yields a maximum expected hit rate (the likelihood with which an expert's weight is available in the GPU memory) of 25.2%,

compared to a minimum of 18.7%. Random selection results in an average hit rate of $56/256 = 21.9\%$. In Environment 2, with GPU memory capacity for 52 experts, the expected hit rates for the best, worst, and random selections are 23.5%, 17.2%, and 20.3%, respectively. Therefore, we can conclude that placing popular experts on the GPU could have approximately 3 points of improvement in hit rate compared to random placement.

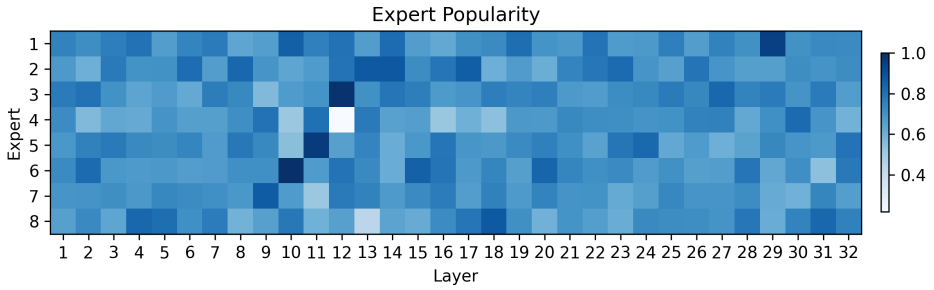

Figure 4: A heat map visualizing expert selection frequency in the Mixtral-8x7B model, using color intensity to reflect the frequency, with the most popular expert normalized to 1.