Orchestrating Heterogeneous Architecture for Fast Inference of Mixture-of-Experts Models

Anonymous ACL submission

Abstract

Large Language Models (LLMs) with the Mixture-of-Experts (MoE) architectures have shown promising performance on various tasks. However, due to the huge model sizes, running them in resource-constrained environments where GPU memory is not abundant is challenging. Some existing systems propose to use CPU resources to solve that, but they either suffer from significant overhead of frequently moving data between CPU and GPU, or fail to consider different characteristics of CPU and GPU. This paper proposes *Twiddler*, a resource-efficient inference system for MoE models with limited GPU resources. *Twiddler* strategically utilizes the heterogeneous computing architecture of CPU and GPU resources by determining the optimal execution strategy. Our evaluation shows that, unlike state-of-the-art systems that optimize for specific scenarios such as single batch inference or long prefill, *Twiddler* has better performance in all scenarios. *Twiddler* achieves 1.26 times speed up in single batch inference, 1.30 times in long prefill processing, and 11.57 times in beam search inference, compared against different baselines.

1 Introduction

Recently, running Large Language Models (LLMs) in a resource-constrained environment is becoming increasingly important and relevant. There is a growing interest in running LLMs in local environments such as a personal computer or an edge device (Giacinto, 2023; Anand et al., 2023; Song et al., 2023) to improve privacy (Martínez Toro et al., 2023) and to customize models using proprietary or personal data (Lyu et al., 2023). Enabling these models to operate in resource-limited settings democratizes access to advanced LLM technologies, particularly for those without access to high-end GPU resources. This trend is strengthened by the proposals to use LLMs at the core of all computer systems. (Packer et al., 2023; Berger and Zorn, 2024). Hence, it is desirable to be able to run large models on a wide range of computers or servers, unlike the status quo where LLMs are usually served with large GPU clusters (Patel and Ahmad, 2023).

Especially, LLMs utilizing Mixture-of-Experts (MoE) architectures have demonstrated outstanding performance across a range of tasks (Du et al., 2022; Fedus et al., 2022; Jiang et al., 2024; Databricks, 2024). MoE models selectively activate a subset of parameters via a gating mechanism, lowering computational requirements for both training and inference compared to dense counterparts. As a result, MoE models are easier to scale to larger sizes, leading to the development of many powerful models (Rajbhandari et al., 2022).

Although MoE models appear well-suited for resource-constrained environments due to their relatively low computational requirements, implementing local inference with these models presents several challenges. Firstly, the model size is usually very large and scales up quickly as the number of experts or the hidden dimension increases. Recently released MoE models include Mixtral-8x7B (47B parameters), Mixtral-8x22B (141B parameters), DBRX (132B parameters), DeepSeek-V2 (236B parameters), Grok-1 (314B parameters), and Snowflake Arctic (479B parameters) (Jiang et al., 2024; AI, 2024; Databricks, 2024; DeepSeek-AI, 2024; xAI, 2024; Snowflake, 2024). Hence, a very large number of GPUs are required just to store model parameters, given the limited capacity of GPU memory. The situation is further exacerbated by the virtually unlimited number of expert components in MoE models; for example, the largest variant of Switch Transformer has 2048 experts in each layer and has a total of 1.6T parameters (Fedus et al., 2022). Without model compression or quantization techniques, the model could take 3.2TB of storage. This means that 40 NVIDIA A100 80GB GPU will be needed just to store all

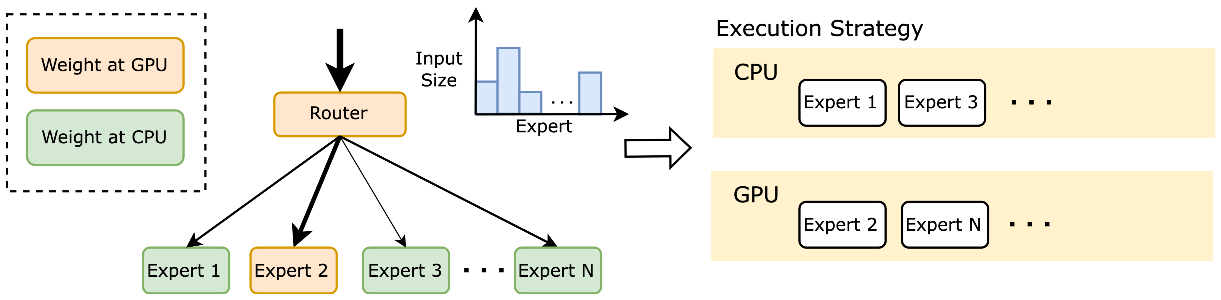

Figure 1: High level overview of *Twiddler*. Each layer of the MoE model is placed at either the CPU memory or the GPU memory, and *Twiddler* determines the optimal execution strategy using both the CPU and the GPU based on the input size of each expert.

the model weight.

Secondly, while all parameters must be stored in GPU memory for efficient inference, the unique property of MoE models means that not all parameters are used to generate a new token. The fact that only a subset of parameters is active during the generation of each token leads to underutilized GPU memory. This is particularly problematic since the soaring global demand for generative AI technologies has driven GPU prices up. The end result is an environment, where investing in a large number of GPUs is not a cost-effective proposition for all but major hyperscalers that serve many users simultaneously to achieve greater GPU utilization through significant batching.

Some existing systems use CPU resources to address the challenge of running large models in resource-limited environments, but they struggle with the efficient execution of MoE models. On the one hand, methods that offload model weights to CPU memory and transfer required weights to GPU memory on demand address memory capacity issues. However, they introduce significant runtime overhead due to the lower bandwidth of PCIe connection compared to memory access (Eliseev and Mazur, 2023; Xue et al., 2024b). On the other hand, CPU-based inference frameworks that partially utilize GPUs can reduce parameter transfer overhead (ggml authors, 2023). However, they fail to account for MoE model properties or different device characteristics of CPUs and GPUs, leading to suboptimal performance in critical use cases like long prefill or beam search, which is essential for enhanced generation quality (Dong et al., 2022; von Platen, 2023). We discuss more detail in §2.

In this paper, we tackle the challenge of efficiently running MoE models with limited GPU resources, by strategically utilizing both CPU and GPU resources. We introduce *Twiddler*, a resource-efficient MoE inference system that intelligently leverages the heterogeneous computing architecture of both CPUs and GPUs. Unlike prior work that either only uses CPU memory or naively splits execution between CPUs and GPUs, our approach generates optimal execution strategies by considering the different characteristics of CPUs and GPUs. As CPUs have larger memory capacity despite having weaker computational power, MoE models are particularly interesting for this context due to their small computational requirement relative to their parameter size.

During inference, *Twiddler* develops a latency model based on different batching effects of CPUs and GPUs to determine the optimal execution strategy for MoE layers, as shown in Figure 1. When expert layers are executed on the CPU, latency increases almost linearly with the input size (see detailed analysis in §A). In contrast, GPU execution latency remains nearly constant regardless of input size but incurs an overhead if the weights need to be transferred from CPU memory to GPU memory. Therefore, for smaller input sizes, it is more efficient to execute expert layers on CPUs, avoiding the overhead of weight transfer. However, for larger batch sizes, CPU computation becomes too time-consuming, making it more efficient to transfer weights to GPU memory and perform computations on the GPU. *Twiddler* dynamically chooses the execution plans that run MoE models efficiently with limited GPU memory across various workloads, including long prefill and beam search.

We also incorporate several optimizations into the design of *Twiddler*. To maximize the likelihood that the required expert is available in GPU memory, we place frequently used experts on the GPU based on offline profiling of expert popularity. Additionally, we design a specialized computation kernel for expert processing on the CPU using the AVX512_BF16 instruction set, which is not supported in the native PyTorch implementation (Paszke et al., 2019).

We evaluate *Twiddler* with the uncompressed (16-bit) Mixtral-8x7B model, which has over 90GB of parameters, on two environments with single GPUs each. *Twiddler* achieves 1.26 times speed up in single batch inference, 1.30 times in long prefill processing, and 11.57 times in beam search inference, compared against different state-of-the-art systems each, on average across different environments (§4) Notably, while existing systems show different trade-offs (*e.g.*, offloading-based approaches excel in long prefill scenarios, while CPU-based methods perform well with single batch latency), our system integrates the advantages of both, achieving balanced and efficient results in diverse conditions.

To summarize, our contributions are as follows:

- We design *Twiddler*, an inference system for MoE models for heterogeneous architecture, that finds the optimal execution strategy using both the GPU and CPU.

- We evaluate *Twiddler* and show that it achieves better performance in single batch inference, long prefill processing, and beam search inference, compared to different state-of-the-art systems each. It shows that *Twiddler* integrates the advantages of different types of existing systems.

2 Related Work

2.1 Mixture-of-Experts

LLMs based on MoE architecture have been showing promising performance in various applications (Rajbhandari et al., 2022; Du et al., 2022; Fedus et al., 2022; Jiang et al., 2024; Xue et al., 2024a; Dai et al., 2024). Unlike original dense Transformers (Vaswani et al., 2017), MoE models add sparsity to the feed-forward layer through a system of experts and a gating mechanism. Each MoE layer contains multiple expert layers that match the shape of the feed-forward layer, and a gating network determines which experts are activated for each input. While an MoE layer can include thousands of experts (Fedus et al., 2022), only a select few are activated by the gating network during training or inference.

2.2 Large Models Deployment with Heterogeneous Architecture

Deploying MoE models efficiently can be challenging because of their large model size, particularly in resource-constrained settings. Some existing systems utilize CPU resources to solve the challenge of running large models in resource-limited environments, but they fall short of running MoE models efficiently.

Offloading is one approach to run large models in such an environment. They store a subset of model weights in the CPU memory instead of the GPU memory to utilize the larger capacity (Sheng et al., 2023). The required weights are transferred on demand from the CPU memory to the GPU memory during computation for inference. For MoE models, some previous works attempted to offload expert weights with caching or prefetching mechanisms (Eliseev and Mazur, 2023; Xue et al., 2024b). These approaches address memory capacity limitations and are good for throughput-oriented scenarios. However, they suffer significant latency overhead due to the frequent transfer of expert weights between the CPU and GPU over the PCIe connection, because its bandwidth is smaller than memory access bandwidth. As a result, they show suboptimal performance for the settings where latency is critical for user experience. *Twiddler* overcomes this challenge by utilizing the computation resources of CPUs.

Another line of work proposes CPU-based inference frameworks that support running LLMs by partially using GPUs (ggml authors, 2023). Depending on the availability of GPU memory, such systems execute a subset of the model layers on the GPU and the rest on the CPU. Although they can reduce the overhead of transferring model parameters, such approaches show suboptimal performance for important use cases, such as long prefill or beam search, that are essential for enhanced generation quality (Dong et al., 2022; von Platen, 2023). This is because they do not consider the different batching effects of GPUs and CPUs (Chen, 2023) and the properties of MoE models.

Even though model compression techniques like quantization (Frantar and Alistarh, 2023; Zhao et al., 2023) or sparsification (Alizadeh et al., 2023) can reduce the model size and improve inference efficiency, they come with degraded output quality of models, especially when trying to fit large models to a GPU with limited memory capacity

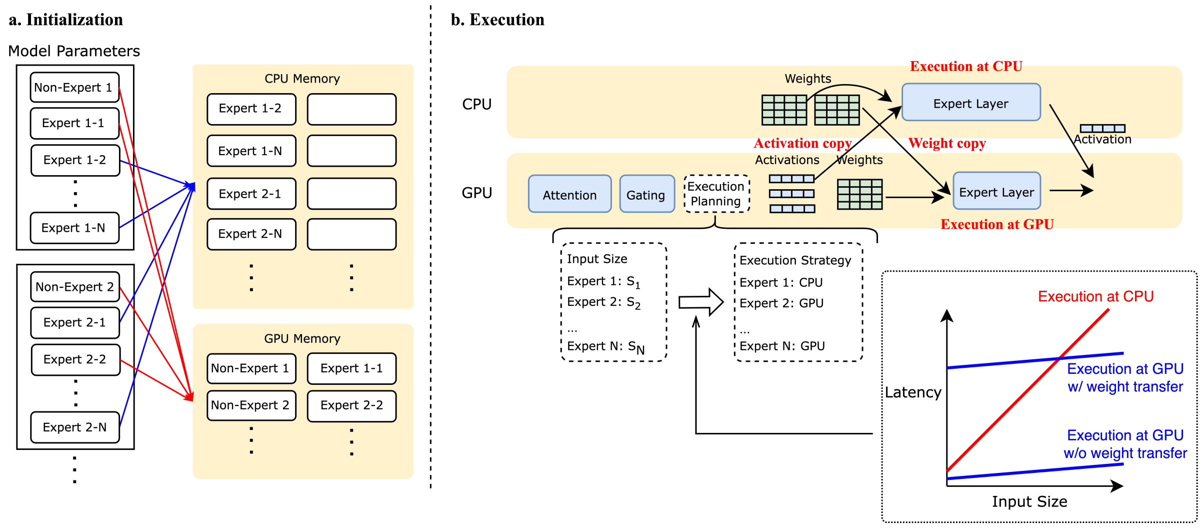

Figure 2: Overview of *Twiddler*. (a) During the initialization phase, the parameters of non-expert layers and a selected subset of expert layers are allocated to GPU memory as availability permits; the remaining parameters are allocated to CPU memory. (b) At runtime, *Twiddler* dynamically determines the optimal execution strategy by considering the volume of inputs that activate each expert layer along with the different expected latencies of CPU and GPU processing.

(Eliseev and Mazur, 2023). Recently, Song et al., 2023 proposed to exploit the LLMs sparsity for faster 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, common in state-of-the-art LLMs, to ReLU models requires additional training and causes degradation of model quality (Mirzadeh et al., 2023; SparseLLM). For example, Mixtral-8x7B uses the Sigmoid Linear Units (SiLU) function (Elfwing et al., 2018), and only a small portion of values are close to zero. Therefore, it is difficult to exploit the sparsity (a more detailed discussion is given in Appendix B). *Twiddler* can achieve better performance without modifying the model structure or accuracy. We note that *Twiddler* is orthogonal to the compression techniques, and these optimizations could be applied on top of *Twiddler*.

3 Design

This section explains the design of *Twiddler*. *Twiddler* is designed for the scenario where GPU memory capacity is insufficient to store all the MoE model parameters. Therefore, the weights of some of the experts are stored in the CPU memory instead of the GPU memory. *Twiddler* finds the optimal execution strategy for such cases, given the expert selection by the input and differing batching behavior of CPUs and GPUs.

3.1 Overview

Figure 2 illustrates the overview of *Twiddler*. In the initialization phase, *Twiddler* allocates the parameters for non-expert layers along with those for a selected subset of expert layers to the GPU memory, as much as the GPU memory capacity permits. *Twiddler* selects those experts to be placed on the GPU memory based on their popularity, which we explain in §3.4. *Twiddler* always allocates the weights of non-expert layers on the GPU memory because they are used for every token, irrespective of expert choice. The size of non-expert layers is usually not big (*e.g.*, less than 2 billion parameters for the Mixtral-8x7B model), and we assume they fit in the GPU memory in this paper. The parameters of expert layers that do not fit in the GPU memory due to capacity constraints are stored in the CPU memory.

During the execution phase, *Twiddler* carefully assesses and selects the most effective execution strategy. This decision is informed by the number of inputs each expert layer receives and the CPU's and GPU's differing processing latencies. The gating layer of the model determines the number of inputs each expert gets, and the processing latencies can be predicted using the device properties.

Twiddler considers the different batching effects of CPUs and GPUs. In processing expert layers, the number of inputs affects the execution latency differently on the CPU and the GPU. Specifically, GPU processing exhibits a relatively stable latency across varying input sizes, which can be attributed to its parallel processing capabilities. These capabilities make the execution latency bounded by the time it takes to load parameters from memory. In contrast, the latency associated with CPU processing tends to scale almost linearly with the input size. This linear increase is due to the CPU's weaker computation capabilities than the GPUs', which makes the latency bounded by the computation part, not the memory movement part. We give a more elaborate analysis in the Appendix A.

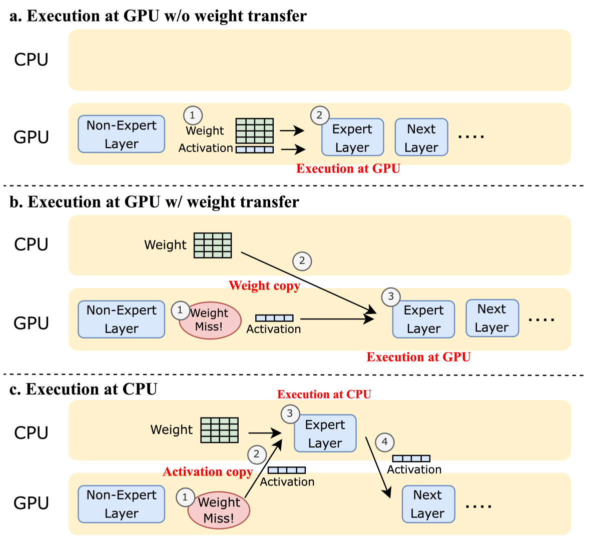

Figure 3: Three different scenarios for the execution of expert layers. When the expert weight is present in GPU memory, the expert layer can be executed at GPU without any data transfer (a). When the expert weight is missing in GPU memory, the expert weight can be copied from CPU memory to GPU memory and executed at GPU (b), or the activation can be copied from GPU memory to CPU memory and executed at CPU (c).

3.2 Execution Strategies

There are three scenarios for the processing of expert layers, as shown in Figure 3. After the expert is selected for each input token at the non-expert layer, (Figure 3 a) ① if the corresponding weight is present on the GPU memory, ② the expert layer is executed on the GPU, without any data transfer between the CPU and the GPU.

However, as all the model parameters do not fit in the GPU memory, sometimes the expert weights are not present in the GPU memory. In that case, two different strategies exist to execute the expert layer. The first method is to copy the model weight from CPU memory to GPU memory, then execute the expert using the GPU (Figure 3 b). 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). Existing offloading systems use this method.

Another approach is to copy the activations from the GPU memory to CPU memory and execute the expert layer on the CPU (Figure 3 c). In this approach, when some expert weights are missing on the GPU memory (①), the activation values are copied 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 activations are copied back to the GPU after the computation finishes (④). A similar method is used by llama.cpp.

The latter two strategies (b. and c. above) have different trade-offs. On the one hand, GPUs have stronger computation ability that is suited for expert processing. Therefore, (b) has an advantage over (c) regarding computation latency. Moreover, as discussed before, the computation latency of (c) becomes longer as the input size grows due to the different batching effects of the CPUs and the GPUs.

On the other hand, considering the CPU-GPU communication, the method in (b) needs to transfer model weights, while (c) only needs to transfer activation values. As the size of activations (input 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, consuming more than 300MB with 16-bit precision) for small input sizes, (c) has an advantage in reducing communication overhead.

Overall, (c) is advantageous when the input size to an expert is small, while (b) is better if the input size is above some threshold, even with large communication overhead. When processing long prefills, the input size can reach a thousand. However, in such scenarios, the computation latency of method (c) becomes more prohibitive than the weight transfer latency of (b), making (c) an impractical choice. Consequently, the transfer latency for activation is negligible when (c) is employed. We give a more detailed quantitative analysis in Appendix A.

3.3 Algorithm

Based on the analysis described above, *Twiddler* serves MoE models in the following way.

Initialization. Before starting the inference process, *Twiddler* distributes the model weights between the CPU and GPU memory. First, the weights of non-expert layers are placed on the 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 *Twiddler* assumes they fit in the GPU memory in this paper. Next, *Twiddler* puts a subset of expert layers into the GPU memory. For this, it selects as many experts as the memory capacity permits to maximize the hit rate, *i.e.*, the likelihood an expert's weight is in GPU memory. For the expert selection, we apply an optimization as discussed in §3.4.

We also measure the latency to copy weights and execute experts on either the CPU or the GPU with different input sizes to inform the decision at runtime.

Execution. At runtime, *Twiddler* identifies the optimal configurations to execute expert layers across the GPU and CPU. *Twiddler* knows which expert(s) should be used for each token being processed after executing the gating function for each layer. This allows *Twiddler* to learn the input size for each expert. Note that multiple inputs can be processed simultaneously, even for a single request, during the prefill stage or when beam search is utilized.

Based on the input size information, *Twiddler* determines the most efficient execution strategy for distributing workloads across the CPU and GPU. To achieve this, *Twiddler* employs Algorithm 1. The function `is_at_gpu(i, j)` checks whether the weight of expert `j` in the `i`-th layer was placed in the GPU memory at the initialization time. Additionally, `cpu_lat(s)` and `gpu_lat(s)` provide the expected latency for executing an expert on the CPU and GPU, respectively, given an input size of `s`. The function `transfer_lat()` estimates the latency required to transfer an expert's weight from CPU memory to GPU memory.

When executing an expert on a GPU along with weight transfer, the latency is primarily dominated by the time it takes to transfer the expert's weight from the CPU to the GPU memory, which is independent of the batch size. In contrast, executing an expert layer on the CPU demonstrates different behavior: as the number of input tokens increases, the latency also increases. However, the time required to copy activation from the GPU to the CPU is negligible, accounting for less than 1% of the total latency (see Appendix A for more details).

Algorithm 1 Expert Execution Strategy

```
1:  Inputs:
2:     $n_e$: number of experts in one layer
3:     $i$: the layer to consider (we consider $i$-th layer)
4:     $inp\_size$: array of size of input for each expert
5:  for $j = 1$ to $n_e$ do
6:     $s \leftarrow inp\_size[j]$
7:     if $s == 0$ then
8:        continue
9:     end if
10:    if is_at_gpu($i, j$) then
11:       // run $j$-th expert at GPU
12:    else if cpu_lat($s$) $>$ gpu_lat($s$) + trans_lat() then
13:       // run $j$-th expert at GPU
14:    else
15:       // run $j$-th expert at CPU
16:    end if
17: end for
```

To optimize the processing of the prefill stage, we employ a model where the GPU execution time is considered constant, whereas the CPU execution time is assumed to increase linearly with the number of input tokens. Specifically, for the number of input tokens s, `gpu_lat(s)` returns a constant value, while `cpu_lat(s)` returns a value proportional to s, multiplied by another constant. These constants are determined in the initialization phase.

3.4 Optimizations

The best execution performance is achieved when the approach of Figure 3 a is used as frequently as possible, *i.e.*, when the expert weight required by the input is present in GPU memory as frequently as possible. To maximize the likelihood that the required expert is available in GPU memory, we place frequently used experts on the GPU based on offline profiling. For this, we select as many experts as the memory capacity permits in order of popularity to 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 almost universal across different input domains (Jiang et al., 2024;

Table 1: Evaluation setups

	Environment 1	Environment 2
GPU	Quadro RTX 6000 (NVIDIA, a)	RTX 6000 Ada (NVIDIA, b)
GPU Memory	24576MiB	49140MiB
PCIe	Gen3 x16 (32GB/s)	Gen4 x16 (64GB/s)
CPU	Intel(R) Xeon(R) Gold 6126 (48 core)	Intel Xeon Platinum 8480+ (112 core)
Number of Experts on GPU	$56/256$	$125/256$

Xue et al., 2024a). Appendix C discusses expert selection in more detail.

Additionally, we design a specialized computation kernel for expert processing on the CPU using the `AVX512_BF16` instruction set, which is not supported in the native PyTorch implementation (Paszke et al., 2019).

4 Evaluation

4.1 Setup

Model and Data. We use Mixtral-8x7B model (Jiang et al., 2024) with 16-bit precision for the evaluation. For the evaluation and calibration data, we use ShareGPT (ShareGPT), 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 *Twiddler* on top of PyTorch (Paszke et al., 2019).

Environments. We evaluate *Twiddler* on two environments as shown in Table 1. None of the environments has enough GPU memory capacity to store all the model parameters. The "Number of Experts on GPU" row shows the maximum number of experts that can fit on the GPU memory out of 256 experts (32 layers $\times$ 8 experts/layer), giving the memory capacity.

Baselines. For baselines, we evaluate DeepSpeed-MII version v0.2.3 (Microsoft), Mixtral-Offloading (Eliseev and Mazur, 2023), and llama.cpp version b2956 (ggml authors, 2023). For DeepSpeed-MII, we enable ZeRO-Infinity optimization (Rajbhandari et al., 2021) so that it offloads model parameters to the CPU memory and loads them from CPU to GPU dynamically during inference when needed. We enable `pin_memory` in the configuration to use paged-locked CPU memory, which improves performance of read/writes from CPU memory and reduce memory defragmentation. Mixtral-Offloading originally supports only 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. Mixtral-Offloading provides an `offload_per_layer` parameter to determine how many experts in each expert layer to offload to CPU memory. We set the `offload_per_layer` parameter to 7 for Environment 1 and 5 for Environment 2 as this is the best configuration for the environments we test. For llama.cpp, we set the `ngl` parameters that control the number of layers being executed in the GPU to be 8 for Environment 1 and 16 for Environment 2.

Metrics. We evaluate the performance of *Twiddler* against baselines in three different scenarios that serve a single request: ⓐ end-to-end latency with different lengths of input and output tokens, ⓑ prefill processing for the long context input, and ⓒ end-to-end latency of beam search with different widths. These metrics reflect important use cases: long context input is used for in-context learning or retrieval augmented generation (Dong et al., 2022; Gao et al., 2023), and beam search is used for enhanced quality of generated tokens (von Platen, 2023). We report the inference speed measured by token per second for ⓐ and ⓒ (number of generated tokens divided by the end-to-end latency), and Time To First Token (TTFT) for ⓑ. For the evaluation with N input tokens, we randomly select samples from ShareGPT with N tokens or more of prompt and use the initial N tokens. For ⓐ, the input length is among [32, 64, 128, 256], and the output length is among [64, 128, 256]. The input length for ⓑ is among [512, 1024, 2048, 4096]. We set the beam search width for ⓒ to be among [4, 8, 12] with an input length of 32 and an output length of 64. For the beam search, we compare *Twiddler* only against llama.cpp as the other baselines do not support beam search inference.

4.2 Results

Figure 4 shows the end-to-end performance of three methods in two environments. On average, across all the configurations and environments, *Twiddler* outperforms the best baseline, llama.cpp, *Twiddler* achieves performance that is 1.26 times faster on average across different input/output lengths and

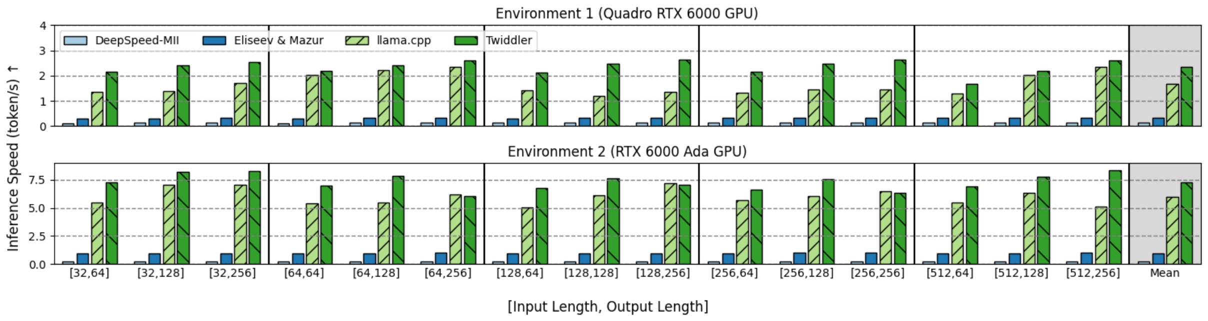
Figure 4: The end-to-end performance comparison by the number of tokens generated per second (scenario ⓐ, higher is better), with 15 different input/output length configurations. The rightmost set of bars shows the average of 15 configurations.

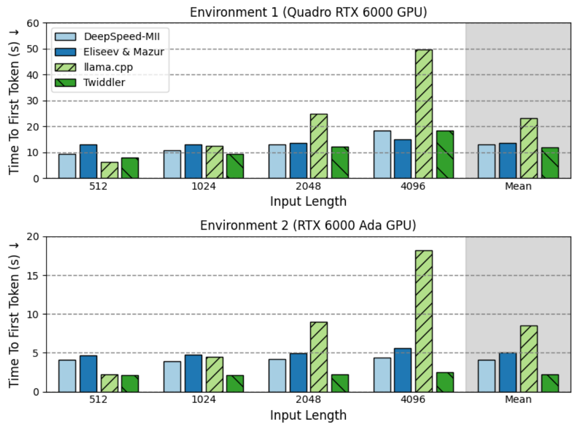
Figure 5: The performance comparison by TTFT (scenario ⓑ, lower is better), with 15 different input/output length configurations. The rightmost set of bars shows the average of 4 different lengths.

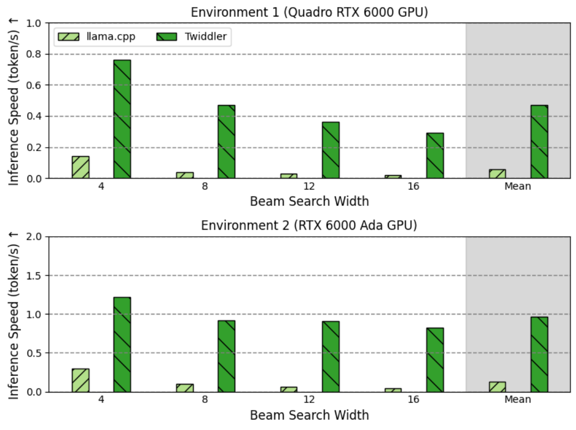
Figure 6: The performance comparison for beam search inference measured by the number of tokens generated per second (scenario ⓒ, higher is better), with input length of 32 and output length of 64. The rightmost set of bars shows the average of 4 beam search widths.

environments. Figure 5 shows the TTFT for the long context prefill. In this case, offloading-based methods (DeepSpeed-MII and Eliseev and Mazur, 2023) are better than llama.cpp. Still, *Twiddler* shows better performance than any existing methods, outperforming DeepSpeed-MII by 1.07 times and Eliseev and Mazur, 2023 by 1.17 times on average across different configurations. Figure 6 shows the end-to-end latency of beam search inference with different search widths, compared against llama.cpp. On average, *Twiddler* achieves 11.57 times better performance than llama.cpp.

These results show that *Twiddler* performs better in a wide range of applications than existing systems. The benefits primarily come from *Twiddler*'s ability to determine execution strategy dynamically based on batching effects of CPUs and GPUs and place experts based on popularity profile. Notably, while existing systems show different trade-offs (*e.g.*, offloading-based approaches excel in long prefill scenarios and methods like llama.cpp perform well with single batch latency), our system integrates the advantages of both, achieving balanced and efficient results in diverse conditions.

5 Conclusion

This paper proposes *Twiddler*, a resource-efficient inference system for MoE models with limited GPU resources. *Twiddler* strategically utilizes the heterogeneous computing architecture of CPU and GPU resources by determining the optimal execution strategy. *Twiddler* achieves better performance in all common scenarios for local inference while state-of-the art systems are only optimized for part of them. Our evaluation shows that compared to state-of-the-art systems, *Twiddler* archives 1.26 times speed up in single batch inference, 1.30 times in long prefill processing, and 11.57 times in beam search inference.

6 Limitations

While *Twiddler* provides valuable insights for using the heterogeneous computing architectures of the CPUs and the GPUs, it has several limitations. First, *Twiddler* is restricted to the case where the model size is larger than the GPU memory but not larger than the CPU memory. Otherwise, compression techniques such as quantization are needed. Second, due to the limited memory of the GPU, *Twiddler* does not support the sequence lengths whose KV cache size exceeds the GPU memory capacity. For such cases, KV cache offloading to the CPU memory is needed. Further research on how to efficiently address this issue would be beneficial.

References

Mistral AI. 2024. Cheaper, better, faster, stronger.

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

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

Emery Berger and Ben Zorn. 2024. Ai software should be more like plain old software.

Lequn Chen. 2023. Dissecting batching effects in gpt inference.

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. 2024. Deepseekmoe: Towards ultimate expert specialization in mixture-of-experts language models. *Preprint*, arXiv:2401.06066.

Databricks. 2024. Introducing dbrx: A new state-of-the-art open llm.

DeepSeek-AI. 2024. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *Preprint*, arXiv:2405.04434.

Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234*.

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

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

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

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

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

Yunfan Gao, Yun Xiong, Xinyu Gao, Kangxiang Jia, Jinliu Pan, Yuxi Bi, Yi Dai, Jiawei Sun, and Haofen Wang. 2023. Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*.

The ggml authors. 2023. llama.cpp.

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

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. 2024. Mixtral of experts. *arXiv preprint arXiv:2401.04088*.

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

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

Iván Martínez Toro, Daniel Gallego Vico, and Pablo Orgaz. 2023. 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. 2023. Relu strikes back: Exploiting activation sparsity in large language models. *arXiv preprint arXiv:2310.04564*.

NVIDIA. a. 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.

NVIDIA. b. Nvidia rtx 6000 ada generation. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/rtx-6000/proviz-print-rtx6000-datasheet-web-2504660.pdf.

Charles Packer, Vivian Fang, Shishir G Patil, Kevin Lin, Sarah Wooders, and Joseph E Gonzalez. 2023. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*.

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

Dylan Patel and Afzal Ahmad. 2023. The inference cost of search disruption – large language model cost analysis.

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

Samyam Rajbhandari, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning. *Preprint*, arXiv:2104.07857.

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. 2023. High-throughput generative inference of large language models with a single gpu. *arXiv preprint arXiv:2303.06865*.

Snowflake. 2024. Snowflake arctic: The best llm for enterprise ai — efficiently intelligent, truly open.

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

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

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

Patrick von Platen. 2023. How to generate text: using different decoding methods for language generation with transformers.

xAI. 2024. Open release of grok-1.

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

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

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

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

A Microbenchmarks

In this section, we show the results of the microbenchmarks. Figure 7 shows the latency of the following workloads:

- `W copy`: Transferring weight of one expert from the CPU memory to the GPU memory
- `A copy`: Transferring one activation from the GPU memory to the CPU memory
- `GPU N`: Executing one expert at GPU with input size N (excluding the time for transferring weight from CPU)
- `CPU N`: Executing one expert at CPU with input size N

For each value, we execute the workload 32 times (once for each layer of Mixtral-8x7B) and present the average and standard deviation results.

When tasks are executed on a GPU, the latency for transferring weights from CPU memory to GPU memory is about 2-5 times longer than the actual computation time. The computation latency on the GPU remains largely constant regardless of batch size. An exception occurs in Environment 1 when the batch size is 1, as PyTorch uses different implementations for single-batch and multi-batch scenarios. However, this difference is minor (approximately 10%) compared to the overall latency, which includes weight transfer. Therefore, we model GPU latency as a constant in Section 3.3.

On the CPU, execution latency generally increases linearly with the size of the input batch. However, the time needed to transfer activations is negligible (less than 1% of the latency with a single input). Due to this minimal impact, our model in Section 3.3 assumes that CPU latency has a linear relationship with the number of inputs.

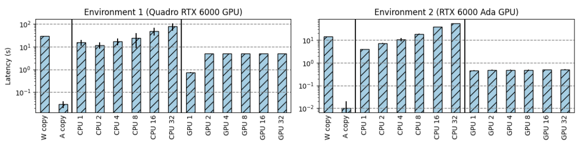

Figure 7: The microbenchmark results measuring the latency of transferring weights or activations between the CPU and GPU, as well as executing an expert layer on either the CPU or GPU with varying input sizes. The y-axis is displayed on a log scale.

B Sparsity Analysis

This section analyzes the sparsity within Mixtral-8x7B models, highlighting the challenges of applying traditional sparsity-based optimization techniques from previous studies (Song et al., 2023; Alizadeh et al., 2023). These methods primarily target LLMs that utilize the ReLU activation function, which nullifies negative inputs and allows for the pruning of channels with consistently zero outputs. This approach leverages the binary nature of ReLU's output—either zero or positive—enabling straightforward identification and elimination of inactive channels, thereby optimizing computational efficiency without sacrificing critical information.

Conversely, state-of-the-art MoE models often use different activation functions, complicating the direct application of these sparsity-exploiting strategies. For instance, Mixtral-8x7B uses SiLU as the activation function. Unlike ReLU, SiLU does not provide a clear threshold of zero for pruning, necessitating a more sophisticated approach to leverage sparsity. Pruning channels that are not sufficiently close to zero could negatively impact the model's output quality.

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 from 100 samples within the ShareGPT dataset (ShareGPT), without differentiating between different experts in identical layers. The data indicates a generally low occurrence of values close to zero. Specifically, for all layers, the proportion of channels with absolute values below 0.001 is less than 2%, and for 30 out of the 32 layers, this figure is even below 1%. Additionally, 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 significant challenge in harnessing sparsity within this model using approaches from previous works. In contrast, Liu et al., 2023 reported that over 90% of values after the ReLU function are 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 remains an intriguing direction for future research.

C Expert Popularity

Figure 8 displays a heat map illustrating the popularity of expert selection within the Mixtral-8x7B model. Similar to the analysis in Appendix B, this profile is generated by running inferences on random samples from the ShareGPT dataset and counting the number of tokens routed to each expert. The color intensity of each cell represents the frequency of expert selection, equivalent to the number of tokens that activated the expert. The value of the most popular expert is normalized to 1, with the popularity of other experts expressed as a ratio relative to this value.

Among the 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. Although the minimum value is 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 that 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 125 experts, the expected hit rates for the best, worst, and random selections are 53.0%, 44.6%, and 48.8%, respectively. Therefore, we can conclude that placing popular experts on the GPU could improve the hit rate by approximately 3 to 5 percentage points compared to random placement.

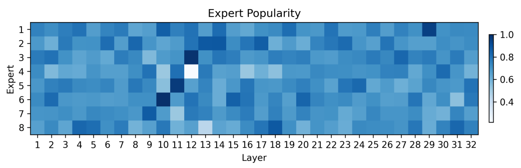

Figure 8: A heat map visualizing expert selection frequency in the Mixtral-8x7B model, using color intensity to represent the frequency, with the most popular expert normalized to 1.

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