Enhancing Distributed CPU Inference For DeepSeek-R1

Osama Dabbousi
KAUST AANSLab
osama.dabbousi@kaust.edu.sa

Chao Fei
KAUST InfoCloud
chao.fei@kaust.edu.sa

Ziwu Liu
KAUST InfoCloud
ziwu.liu@kaust.edu.sa

***Abstract*—Report summarizing our findings while attempting to optimize inference of a large language model (LLM) such as DeepSeek-R1 on a distributed multi-node CPU cluster such as Shaheen 3. GitHub link: https://github.com/OsamaDabb/DeepSeek-CPU-Inference**

I. Problem Statement

Despite the growing dominance of GPU-based infrastructures for large language model (LLM) inference, deploying LLMs on pure CPU clusters offers a number of strategic and practical advantages.

First, CPU clusters are often more readily available and cost-effective in traditional high-performance computing (HPC) environments and enterprise data centers. This allows for broader accessibility and lower upfront deployment costs, particularly in scenarios where GPU resources are limited or prohibitively expensive.

Secondly, CPU nodes typically exhibit higher memory capacities per socket compared to GPUs, which is beneficial when serving models with large parameter sizes or long-context workloads. Modern CPUs also support a large number of concurrent threads and benefit from advanced instruction sets such as AVX-512 and AMX (on x86 architectures), enabling substantial performance gains when paired with optimized low-level libraries.

While CPU clusters present notable potential for LLM deployment, they also introduce a unique set of architectural limitations that must be addressed to ensure efficient inference performance. In particular, the memory hierarchy and computational density of CPU hardware differ significantly from those of modern GPUs, which are typically the default targets for large-scale model serving.

a) Memory Bandwidth and Latency.: Unlike GPUs that are optimized for high-throughput data movement via dedicated high-bandwidth memory, CPUs generally rely on shared DDR4/DDR5 memory with considerably lower bandwidth. As LLM inference involves repeated access to large parameter matrices and activation tensors, the memory subsystem can quickly become a bottleneck. Additionally, CPUs exhibit higher memory access latencies, further impeding real-time token generation. This mismatch between the memory demand of transformer-based models and the bandwidth availability on CPUs results in underutilized compute units and degraded throughput.

b) Lower Computational Density.: Modern GPUs are designed with thousands of parallel cores and specialized matrix multiplication units (e.g., Tensor Cores), enabling them to achieve extremely high FLOPs per watt. By contrast, CPUs prioritize general-purpose compute and sequential task flexibility over dense arithmetic performance. Even with advanced vector instruction sets such as AVX-512 or AMX, the effective compute density of CPUs remains significantly lower. Consequently, when serving LLMs that require intensive matrix-vector operations during each layer's forward pass, CPUs face inherent disadvantages in raw throughput.

c) Thread-Level Concurrency and Synchronization Overhead.: Although CPUs support multi-threading across physical and logical cores, the parallel execution of LLM inference steps is often impeded by the need for fine-grained synchronization. For instance, attention layers and residual connections require coordinated access to intermediate states, increasing inter-thread communication costs. Furthermore, thread contention over shared resources (e.g., caches, memory bandwidth) can lead to performance degradation, particularly when the number of active threads approaches or exceeds the saturation point of the system. This makes it challenging to fully exploit CPU concurrency without careful scheduling and load balancing strategies.

Therefore, from the very beginning, we realized that this was a meaningful and challenging task. Currently, due to the limitations above, the mainstream large model inference hardware is GPU. And influenced by the prosperity of the related open-source community, at least for now and for a long period of time in the future, GPU inference will still be the mainstream. However, this does not prevent us from exploring the deployment on pure CPU clusters. Based on our research, we have listed as many meaningful challenges as possible for improving throughput. We will present these challenges in the next section.

II. Potential Strategies

To address the performance bottlenecks identified in CPU-based LLM inference, we explore two broad categories of optimization strategies aimed at improving system throughput. These approaches are designed to better align the workload characteristics of transformer-based models with the capabilities of CPU hardware.

- **Thread-Level Parallelism via Multi-Core Execution.** Modern CPUs are equipped with dozens of physical cores and can support hundreds of simultaneous threads through simultaneous multi-threading (SMT). By parallelizing the execution of computationally intensive components—such as matrix multiplications in self-attention and feed-forward layers—across multiple threads, it is possible to significantly improve throughput. This thread-level parallelism allows for the concurrent execution of token generation tasks across different input requests or different parts of the model computation graph. Libraries such as `ggml` (used in `llama.cpp`) provide low-level primitives that are optimized for such multi-threaded execution, taking advantage of advanced CPU instruction sets like AVX-512 or AMX. However, careful thread affinity and load balancing are crucial to mitigate contention over shared memory resources.
- **Pipeline Parallelism and Request-Level Batching.** Another promising direction involves improving resource utilization through intra-model and inter-request scheduling. At the model level, pipeline parallelism can be employed to execute different transformer layers concurrently on separate worker threads or processes. This approach overlaps computation and reduces idle time per token step. At the request level, batching multiple input sequences together allows shared operations (e.g., matrix multiplications, KV cache lookups) to be amortized across tokens, improving arithmetic intensity and memory locality. Although batching is traditionally more effective on GPUs, it can still yield considerable benefits on CPUs when combined with static scheduling and cache-aware memory management. These techniques collectively improve both latency-bound and throughput-bound workload scenarios on CPU clusters.

While exploring optimization strategies, it is equally important to recognize certain approaches that are ill-suited for CPU clusters. First, frequent inter-node communication—such as those arising from RPC during distributed inference—can introduce significant latency and network overhead, counteracting gains from parallelism. Second, many GPU-oriented frameworks rely on deeply optimized kernels, such as fused attention or FlashAttention, which are not easily portable to CPU environments and lack equivalent performance benefits. Lastly, complex distributed data parallelism schemes, which require fine-grained synchronization across devices, may incur prohibitive coordination and memory costs when applied to CPU-based systems. These methods should be avoided or carefully re-engineered to suit the CPU context.

III. Selection of Methods

A. Evaluation of Existing Toolchains

In the early stages of our exploration, we experimented with several popular frameworks that support LLM inference at scale. Each tool provides distinct design trade-offs and architectural assumptions, many of which are tailored primarily for GPU environments. This section briefly summarizes their characteristics and the limitations we encountered when adapting them to a CPU-only cluster.

a) SGLang: SGLang is a high-level inference framework that emphasizes compositional instruction tuning and multi-model orchestration. It supports conversational agent construction, function calling, and structured output generation, with an integrated runtime that manages token streaming and inter-request scheduling. While the framework demonstrates strong modularity and ease of use, it heavily depends on GPU-specific features such as CUDA kernel scheduling and asynchronous device queues. Its runtime leverages GPU-based token sampling and fused transformer kernels, which are **absent on CPU backends**. Furthermore, the framework lacks an abstraction layer for CPU-specific parallelism (e.g., thread pools or SIMD-optimized backends), making it **unsuitable for deployment in no-GPU environments without significant re-engineering**.

b) vLLM: vLLM introduces a novel runtime design centered around *PagedAttention*, a memory-efficient approach to managing KV caches during multi-request inference. Unlike traditional implementations that store KV pairs in a fixed-size contiguous buffer, vLLM uses a virtual memory paging mechanism to dynamically allocate and compact memory regions. This significantly improves memory utilization and enables large-scale batching on GPU. However, the core memory manager and scheduling policies are deeply tied to GPU memory semantics, relying on CUDA streams and device-level atomic operations. In its current form, **vLLM lacks an equivalent CPU-optimized backend and cannot leverage CPU-side memory paging features** (e.g., NUMA-aware allocators or TLB-optimized mappings), rendering it impractical for efficient CPU-based LLM inference.

c) Megatron-LM: Megatron-LM is a well-established framework primarily designed for pretraining and fine-tuning large transformer models across GPU clusters. It supports tensor parallelism, pipeline parallelism, and sequence parallelism, making it highly effective for model training at scale. However, its inference path is not designed for lightweight deployment. The framework assumes a tightly coupled multi-GPU environment with NCCL-based all-reduce operations and fused GPU kernels for transformer layers. When ported to a CPU cluster, the absence of GPU accelerators not only leads to degraded performance but also poses challenges in managing model sharding, communication scheduling, and memory layout. As a result, Megatron-LM is unsuitable for real-time or on-demand inference in CPU-only environments.

B. `llama.cpp`

After evaluating several mainstream LLM serving frameworks, we ultimately selected `llama.cpp` as the foundation for our CPU cluster deployment. This decision was guided by the framework's design philosophy, practical performance on CPU hardware, and extensibility in large-scale multi-process environments. Below, we elaborate on the core factors that make `llama.cpp` particularly suitable for our deployment scenario.

`llama.cpp` is explicitly engineered for efficient inference on CPU devices. It utilizes a custom tensor library, `ggml`, which implements low-level operations such as matrix multiplications using cache-aware blocking techniques and SIMD instructions (e.g., AVX, AVX2, AVX-512). These optimizations ensure that the memory hierarchy of modern CPUs—including L2/L3 caches—is effectively utilized during inference. Additionally, the framework supports multi-threaded execution through fine-grained control over thread pools, enabling parallel token generation and pipelined execution across transformer layers. Unlike many GPU-first frameworks, `llama.cpp` makes no assumptions about GPU availability, making it well-suited to CPU-only clusters.

To reduce the memory footprint and accelerate inference, `llama.cpp` supports a variety of quantization formats, including `Q8_0`, `Q4_0`, and `Q2_K`, among others. These formats allow large models such as DeepSeek-R1 to be loaded and executed within the memory limits of commodity CPU nodes without significant loss in generation quality. Moreover, the framework provides native support for key-value (KV) caching, which stores intermediate hidden states from previous tokens to avoid redundant computation in autoregressive decoding. This feature is essential for reducing latency in multi-turn or long-context scenarios, and is implemented in a way that is memory-efficient and thread-safe.

In addition to its performance advantages, `llama.cpp` offers a modular and transparent codebase, primarily written in portable C/C++. This makes it easy to integrate with existing scheduling frameworks and implement custom control logic for parallel inference, fault recovery, or memory reuse. The framework can be compiled and deployed on a wide range of CPU architectures, including x86_64 and ARMv8, with minimal platform-specific modification. This architectural flexibility facilitates deployment in heterogeneous CPU clusters and hybrid edge-cloud settings. Furthermore, `llama.cpp`'s self-contained nature and minimal external dependencies allow us to construct multi-process pipelines without relying on heavyweight RPC layers, simplifying both scaling and debugging.

The four frameworks exhibit distinct design philosophies and deployment characteristics. `SGLang` and `vLLM` are both optimized for GPU environments, relying on CUDA-based scheduling and memory management mechanisms that are not transferable to CPU-only setups. `Megatron-LM`, while powerful for distributed training, lacks lightweight inference support and depends heavily on GPU communication libraries such as NCCL. In contrast, `llama.cpp` is uniquely designed for efficient execution on CPUs, supporting native multi-threading, quantization, and key-value caching without requiring GPU resources. It stands out as the only framework among the four that enables practical and scalable deployment of large language models on pure CPU clusters.

IV. Setting Up on Shaheen

Scripts for building `llama.cpp`, launching Shaheen RPC nodes, and running inference are located in the `slurm` directory of the GitHub repository.

A. Scripts Introduction

Building `llama.cpp`

The script `build_llama_cpp.sh` compiles the `llama.cpp` project with the `RPC` flag enabled, thereby starting the RPC server.

Pipeline

1) Submit the server job, specifying the number of nodes with `--ntasks`:
2) Submit the client or benchmark job:
   - `sbatch client_node.slurm` to run `llama-cli`.
   - `sbatch benchmark.slurm` to execute performance benchmarks.

Monitoring

Both server and client jobs are monitored for CPU usage, memory consumption, and network I/O. Monitoring is performed by the `worker_diagnostic.py` script.

B. Shaheen 3 Usage

You can generate a Slurm job configuration via the Jobscript Generator[1].

After submitting a job with `sbatch`, identify the allocated compute node by running:

```
squeue --me -o "%N"
```

For example, once you've submitted the server job:

```
sbatch server.slurm
```

execute

```
squeue --me -o "%N"
```

to display the node name. Then connect via SSH:

```
ssh <node>
```

and monitor or debug your processes directly on that node.

V. Experiments

A. RPC feature

The RPC implementation of llama.cpp is available on GitHub[2]. To enable it, add the `--rpc` flag when using the llama-cli.

RPC implements a model-parallel strategy: the input layer resides on the client node, while the remaining layers are evenly distributed across the RPC server nodes. For example, when running the DeepSeek-R1-8bit model (≈600 GB), the client node uses about 70 GB of memory, and the two server nodes each consume roughly 340 GB.

Thread number

By default, the RPC subsystem spawns four threads, which may bottleneck tokens-per-second throughput. You can adjust the thread count to improve performance option). See GitHub Issue [3] and Pull Request [4] for details.

[1]https://docs.hpc.kaust.edu.sa/quickstart/jobscriptGenerator.html#jobscript-generator

[2]https://github.com/ggml-org/llama.cpp/tree/master/tools/rpc

[3]https://github.com/ggml-org/llama.cpp/issues/13051

[4]https://github.com/ggml-org/llama.cpp/pull/13060

Stage	Total	Compute	Weight (R)	Cache (R)	Cache (W)
Prefill	2711	2220	768	0	261
Decoding	11315	1498	3047	7046	124

Fig. 1: Cost Analysis of different stages, from FlexGen [6]. (R) denotes Read and (W) denotes Write.

Low CPU Usage during TG

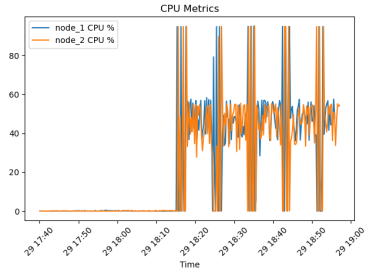
During the token-generation phase (decoding), CPU utilization is low, whereas it peaks during prompt-processing (prefill). These correspond to the two main stages of LLM inference. As shown in Figure 1, the computational load during decoding is minimal, which explains the overall low CPU usage: there simply isn't enough work to parallelize across many threads, so adding more threads yields diminishing returns. Moreover, increasing the thread count does not reduce communication latency between nodes.

B. Node utilization

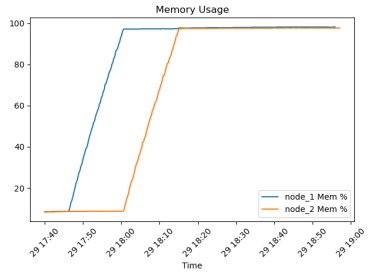
We have run a number of tests to better analyze and understand the way the llama.cpp RPC package works. In particular, by using a Python script we asynchronously log diagnostic stats during inference at each node in the system. Monitoring CPU utilization, memory usage and network traffic into/out of the system we get the following results. Figure 2 shows the 2 node case, and Figure 3 shows the 20 node case. The most significant observations to be made are: the very long model load-time- taking over an hour for these experiments, The spikes in CPU utiliziation followed by lower utilization regions, and the very low network communication during inference.

The CPU usage graph shows an important interaction. We found through benchmarking that prompt processing is sped up by higher node counts, but token generation is not. Looking at the two CPU graphs we point out the distinct pattern of medium utilization followed by individual spikes in computation. The regions of medium utilization across all nodes are the prompt processing sections where all of the tokens of a prompt can be processed in parallel by the different nodes. However, when it comes to token generation, we see the work being split between a small handful of sequentially processing nodes, with spikes going from 0-100% utilization for short bursts. This aligns with our understanding from literature- the sequential and auto-regressive nature of LLMs means that work is inherently sequential. However, literature also tells us there are a number of work-arounds to this issue. Prompt-batching allows the processing of more requests at a time, tensor/data parallelism would allow multiple nodes to work on each token together, reducing their idle time. The network

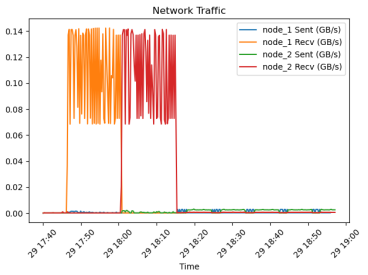
Fig. 2: CPU utilization, memory usage and network activity for 2 Nodes



(a) CPU Utilization



(b) Memory Usage



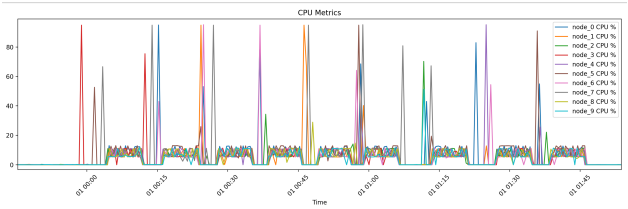
(c) Network Traffic

traffic graphs show that after the high activity during model-loading, the network traffic is incredibly low. So, more communication heavy allreduce/allgather operations needed for tensor parallelism and prompt batching would take better advantage of the high bandwidth architecture of Shaheen.

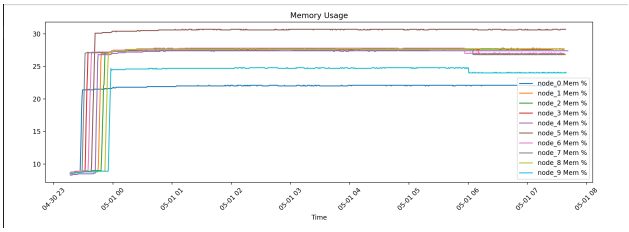
C. Important Code Snippets

While exploring the code of llama.cpp, we have narrowed down a few snippets of code that impact the loading and inference of the model. For starters, the loading of the DeepSeek model across CPU nodes occurs in the llama-model.cpp file under the load-tensor() method, line 1338 onwards. It distributes model parameters evenly across a set of input devices, which are assigned in the llama.cpp file at line 188- in our case assigned as RPC instances. Moreover, from lines 1457 we see how model layers are split- distributed across all nodes

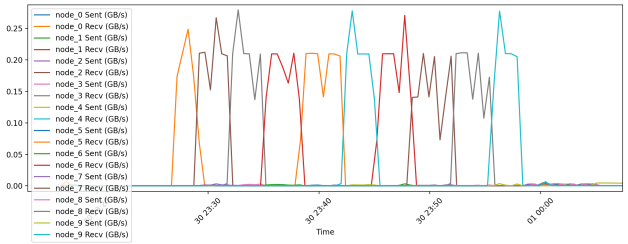
Fig. 3: Same tests for the 20 Nodes case



(a) CPU usage



(b) Memory Usage



(c) Network Traffic

proportionally to their fraction of total RAM. As for how the layers are distributed, for DeepSeek-R1 (listed as DeepSeek2 in the code) each of its 61 transformer layers are split vertically across the nodes, and each layer contains 256 experts. For example, in the two node case the first 31 layers would go to node 1 comprising the first half of each expert, while the remaining 30 layers go to node 2. The distribution of these layers can be found in llama-model.cpp in lines 3206-3286 under the switch-case LLM_ARCH_DEEPSEEK2.

As for inference itself, one will likely have to dive one level deeper into the source to control communication and distribution of tasks. According to a primary contributer of the library [1], inference is built on the GGML backend- a tensor computation library for machine learning [2]. Llama.cpp is only an API which handles LLM specific features like KV-caching, context, vocabulary and sampling. The /ggml/ and /ggml/ggml-cpu/ directories inside of llama.cpp will be the first places to investigate, especially the ggml/ggml.c file which handles the execution of the computational graph for the model.

D. Execution Diagnostics

We build upon the `llama.cpp` parallel example, a C++-based testbed that emulates a concurrent server processing LLM requests from multiple clients. Each simulated client independently issues an autoregressive generation request based on a user-defined prompt. The requests are served using a shared inference context, where all client sequences utilize a common KV cache memory space. The system prompt is pre-evaluated and copied into each client slot via `llama_kv_self_seq_cp`, and response tokens are generated incrementally using a per-client sampler.

The simulator exposes key runtime parameters such as:

- `n_parallel`: the number of concurrent client sequences processed;
- `n_sequences`: total number of sequential prompts to simulate;
- `n_predict`: the maximum number of tokens to generate per request;
- `n_batch`: maximum number of tokens per decoding batch;
- `cont_batching`: whether to inject new requests continuously.

The backend tracks metrics including total prompt and generation tokens, per-client generation speed, and cache miss statistics when KV space is exhausted.

We simulate a varying number of parallel clients, ranging from 1 to 10, while holding other factors constant (`n_predict = 128`, `n_batch = 32`). Each configuration runs until a fixed number of total sequences has been processed. We enable continuous batching to simulate realistic request arrival patterns.

For each value of `n_parallel`, we collect the following system-level indicators:

- **Cache Miss Probability**: the ratio of batch retries due to KV cache overflow to total decode attempts;
- **KV Cache Usage Ratio**: measured as the ratio of active KV memory tokens to the available capacity at peak;
- **Normalized Throughput**: token generation rate normalized to the maximum observed throughput across all settings.

Three corresponding figures (Figure 4, 5, 6) are produced to visualize the system behavior as client concurrency increases.

The experimental results reveal clear signs of cache saturation and throughput degradation as the number of concurrent clients increases. As shown in Figure 4, the cache miss probability remains near zero when the number of parallel clients is below 7, indicating that the KV cache is sufficient to accommodate simultaneous sequences without eviction. However, beyond 8 clients,

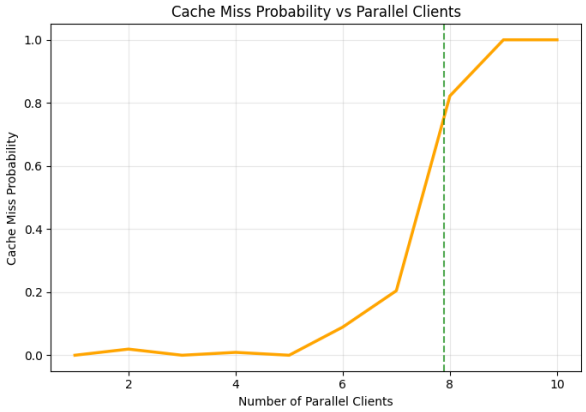


Fig. 4: Cache Miss Probability vs Parallel Clients

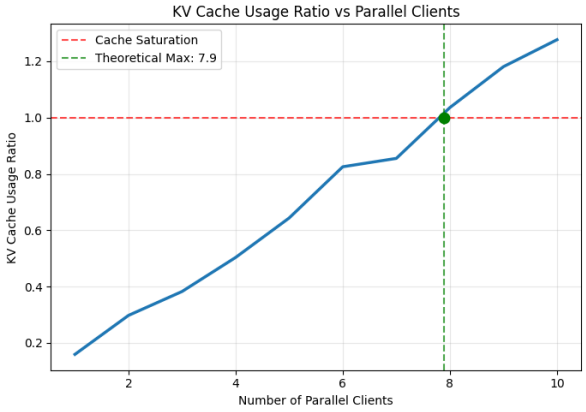


Fig. 5: KV Cache Usage Ratio vs Parallel Clients

the miss probability sharply increases and eventually reaches 100%, suggesting full cache exhaustion and frequent batch decoding retries.

This trend is corroborated by Figure 5, which reports the KV cache usage ratio. The KV cache size is set to match a context length of 512 tokens per sequence. This allows for estimating saturation thresholds empirically. A linear growth in usage is observed up to the theoretical saturation point (approximately 7.9 clients), marked by the intersection with the red dashed line. Once this threshold is crossed, the usage exceeds capacity, triggering forced evictions or reallocation failures. The green marker highlights the precise point where cache utilization transitions from efficient reuse to overload.

Consequently, as shown in Figure 6, the normalized throughput begins to decline rapidly beyond the saturation point. While the system maintains optimal or near-optimal token generation rates up to 6–7 clients, performance drops drastically when exceeding the KV cache limit, with throughput falling below 10% of the baseline at 9 or more concurrent clients. This degradation

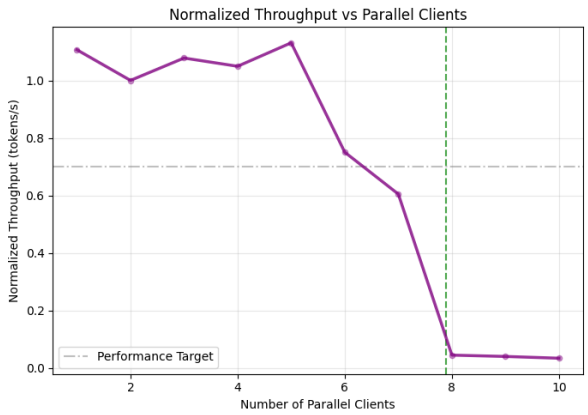


Fig. 6: Normalized Throughput vs Parallel Clients

confirms that cache contention—rather than raw CPU compute—is the primary bottleneck in multi-client LLM inference on CPU clusters under this configuration.

VI. Future Works

Reiterating our findings, we see that a major weakness in current inference with llama.cpp is in token generation- in particular the single-node and sequential nature leads to low CPU utilization and long idle times.

Since a majority of more popular libraries such as SGLang, vLLM, Megatron-LM are GPU only, llama.cpp is so far the only library we have found capable of distributed CPU inference. However, there have been reports from users that the llama.cpp RPC library which we are using is inexplicably slow and struggles in many node contexts- moreover, RPCs are widely known to be slower than message-passing (like MPI) style communication. As a result, it's worth considering a few other libraries which were not tested during our work, which may potentially be CPU compatible. For explicit control over model parameters and communication, the PyTorch distributed library appears capable of multi-node CPU computation, but would require a framework built from the ground up- in particular, using either the distributed RPC [4] or distributed MPI libraries [5]. Another promising solution is the HuggingFace library Accelerate [3]. It implements an Accelerator structure built on the PyTorch distributed library which handles distribution of large models automatically, but seems to have more potential for customization and user control than llama.cpp.

References

[1] ggml-org. How does llama.cpp work? https://github.com/ggml-org/llama.cpp/discussions/4531, 2023. Accessed: 2025-05-12.

[2] ggml-org. ggml: Tensor library for machine learning. https://github.com/ggml-org/ggml, 2025. Accessed: 2025-05-12.

[3] Hugging Face. Accelerate documentation. https://huggingface.co/docs/accelerate/en/index, 2025. Accessed: 2025-05-12.

[4] PyTorch Contributors. Remote procedure call (rpc) — pytorch 2.7 documentation. https://pytorch.org/docs/stable/rpc.html, 2025. Accessed: 2025-05-12.

[5] PyTorch Contributors. Writing distributed applications with pytorch. https://pytorch.org/tutorials/intermediate/dist_tuto.html, 2025. Accessed: 2025-05-12.

[6] Ying Sheng, Lianmin Zheng, Binhang Yuan, Zhuohan Li, Max Ryabinin, Daniel Y. Fu, Zhiqiang Xie, Beidi Chen, Clark Barrett, Joseph E. Gonzalez, Percy Liang, Christopher Ré, Ion Stoica, and Ce Zhang. Flexgen: High-throughput generative inference of large language models with a single gpu, 2023.