Distributed Inference

Boris Gans, Yousef Amirghofran, Anze Zgonc, Léa Abou Jaoudé, Matthew Porteous, Alp Baghirov

Introduction and Motivation

Large language models frequently exceed the memory capacity of a single GPU, which makes even inference a distributed systems problem. This project is a hands-on exploration of running a small-sized model (OpenLLaMA 3B v2) across multiple GPUs and nodes under Slurm, using pipeline parallelism as the sharding strategy. Although the target model is modest compared to frontier LLMs, the intent is to develop a reproducible playbook for scaling larger models with constrained resources. The work emphasizes: (a) reproducible orchestration via Slurm, (b) explicit control of device mapping and offload behavior to work within tight GPU memory limits, and (c) metrics collection for later performance analysis.

Approach

Our setup is built around a simple goal: run a modest LLM across multiple GPUs and nodes in a way that mirrors how larger models must be handled. To do this, we split the model into two sequential stages—one handled by each GPU—and coordinate their work during inference. Each stage loads only the layers it is responsible for, keeping GPU memory usage predictable and low. Any layers that aren't needed on a particular device stay on disk, which helps us fit the model cleanly within the available hardware limits.

All model files and the tokenizer are already stored on the cluster, so nothing is downloaded at runtime. We rely on Slurm to coordinate the distributed job, with each task running inside the same containerized environment. The container provides a stable PyTorch/CUDA stack, while lightweight Python dependencies are added at runtime in a shared workspace. Prompts and configuration files are also staged ahead of time so each rank runs with identical inputs and parameters.

Cluster and Filesystem Layout

Before the job starts, we place three main things on the cluster: the project code, a scratch directory for temporary runtime data, and a pre-downloaded copy of the model. During execution, these pieces are mounted into the container so that every task sees the same code, data, and model path. The shared scratch directory becomes the working area for configs, logs, intermediate outputs, and any CPU-offloaded model fragments created during inference. See below for a sketch of the file structure on the cluster.

```
Host prep (before sbatch)
/home/<USER>/projects/def-sponsor00/<USER>/distributed-inference
/home/<USER>/scratch/group1/pipeline_run
├── exp_config.json
├── ds_config.json
├── prompts.jsonl
└── outputs/
/home/<USER>/scratch/group1/models/openllama-3b
/home/<USER>/projects/def-sponsor00/shared/images/pytorch-2.3.1-cuda11.8.sif

Container view during the job (binds set in slurm/submit.sbatch)
/app/
/tmp/workspace/
├── exp_config.json
├── ds_config.json
├── prompts.jsonl
├── outputs/
│   ├── rank_0.log, rank_1.log
│   ├── completions_rank_0.jsonl, completions_rank_1.jsonl
│   ├── sacct_<jobid>.txt (if sacct is available)
│   ├── nsys_rank_<r>.* or perf_rank_<r>.txt (when PROFILER is set)
│   └── other per-rank stdout from run.sh
├── hf_cache/               # Hugging Face cache populated at runtime
├── .venv/                  # runtime Python deps installed by slurm/run.sh
└── .pip-cache/, .triton-cache/
/workspace/models/openllama-3b -> ${SCRATCH_ROOT}/models/openllama-3b
```

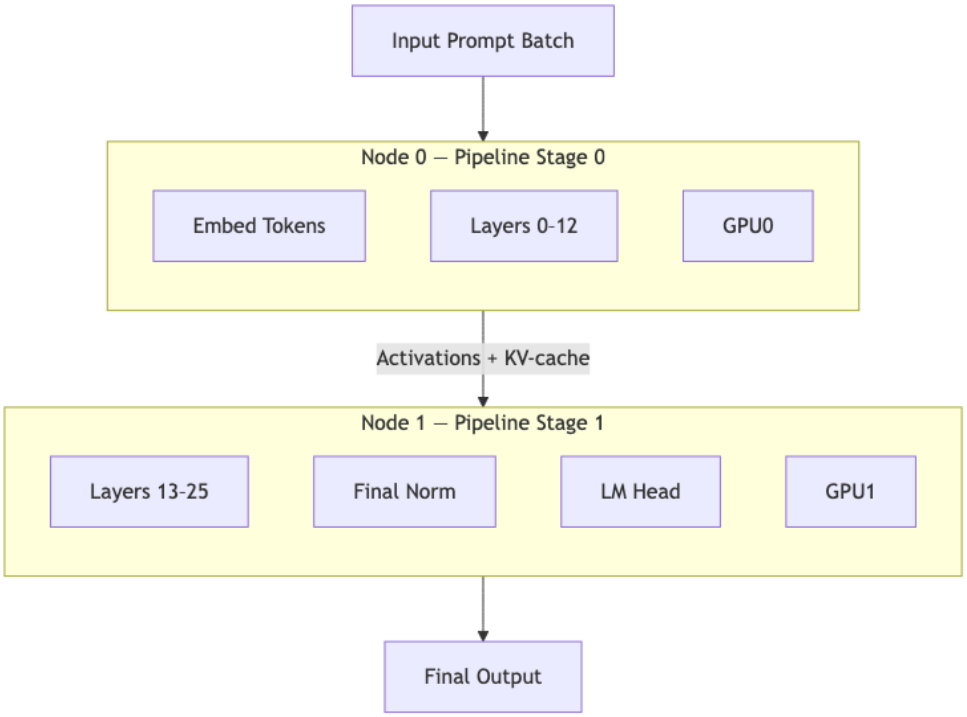
Launch Flow and Configuration

Slurm is responsible for allocating nodes and GPUs, loading the required modules, and launching one containerized task per rank. At job start, each task sets up its environment (DeepSpeed variables, CUDA/NCCL settings, and basic threading options) and prepares a small virtual environment with the Python packages needed for the run.

The experiment configuration defines where to find the model and prompts, where to store outputs, and what inference parameters to use (e.g., generation length, sampling settings). DeepSpeed is included only for its parallelism utilities; in our case, its primary role is confirming the pipeline structure we apply manually inside the inference script.

Model Loading and Partitioning Logic

Inside the main inference script, each rank loads only its portion of the model directly from the local model directory. Layers are divided between the two pipeline stages—roughly the first half on rank 0 and the second half on rank 1—while shared components like embeddings or final output layers live exclusively on one side. Anything not assigned to a rank is offloaded to disk since the memory is tight.



Inference proceeds in a simple pipeline:

- Input token IDs are prepared before pipeline execution begins.
- During each decoding step, rank 0 processes embeddings and early layers, then forwards activations and the KV cache to rank 1, which completes the forward pass and computes logits.
- A token is selected (greedy decoding), and that token is used as input for the next decoding step, re-entering the pipeline at rank 0.
- This process repeats until generation completes.

We keep the decoding strategy intentionally simple (greedy decoding in small batches) to focus on correctness and the behavior of the pipeline rather than model quality.

Data and Prompts

We evaluate the system with a small set of general-knowledge prompts, each roughly a few thousand tokens in length. For instance, one such prompt asks:

> "Summarize the key factors that determine scalability in distributed inference for large language models."

These prompts serve as a testbed for checking correctness and measuring throughput. For each prompt, we also include a longer variant (approximately four thousand tokens) to provide an additional testing dimension–either to stress the pipeline harder or to benchmark lighter configurations.

Hardware and Container Notes

Each rank runs inside an Apptainer container with GPU access passed through from the host. The cluster nodes are equipped with NVIDIA Tesla T4 GPUs, and our experiments allocate one T4 per task. The T4 provides 16 GB of VRAM, supports CUDA 12.4 via the host driver, and offers efficient FP16/INT8 inference performance well-suited for distributed model serving.

All runtime caches, temporary model shards, and offload directories are placed within the shared scratch filesystem, allowing both ranks to read and inspect intermediate artifacts. This shared storage model ensures cache coherence and avoids redundant downloads across ranks.

Distributed initialization relies on Slurm environment variables (`SLURM_PROCID`, `SLURM_NODEID`, `SLURM_JOBID`, etc.) and uses NCCL for communication between the two ranks. NCCL automatically utilizes the available PCIe paths on the T4 nodes since this system typically does not provide NVLink.

Experiment Plan

We developed three complementary experiments: strong scaling, weak scaling, and a batch-size sweep. Each experiment isolates a different aspect of parallel behavior and provides insight into where performance limits come from and how future scaling decisions should be made.

Strong Scaling:
This experiment holds the total workload constant while increasing the number of nodes or GPUs (for example, running the same prompt set on one node versus two). It measures how runtime and throughput respond when additional resources are applied to a fixed job. Ideally, doubling the number of nodes would nearly halve the wall-clock time. Deviations from this ideal reflect parallel inefficiencies such as communication overhead, startup costs, load imbalance across stages, or pipeline bubbles. The results tell us how well the current model partitioning and distributed strategy take advantage of extra hardware for a fixed job size. Strong-scaling efficiency directly informs whether investing in more nodes or deeper pipeline splits is worthwhile. Poor efficiency indicates that scaling further will produce diminishing returns unless communication patterns, partition boundaries, or infrastructure bandwidth are improved.

Weak scaling:
In weak scaling, both the problem size and the number of nodes increase together so that each node sees approximately the same amount of work (e.g., keeping prompts/tokens per GPU constant). This experiment measures whether per-GPU throughput remains stable as the system scales outward. Ideally, the time-to-solution would stay roughly constant as total work and total hardware grow in proportion. Degradation in weak-scaling performance reveals the cost of coordination relative to computation, including communication overhead, synchronization points,

scheduler behavior, or data-loading contention. Weak-scaling characteristics also guide capacity planning. If adding nodes causes per-GPU throughput to drop, scaling out to serve more users or prompts will not be effective until underlying communication, I/O, or pipeline structures are optimized.

Batch-size sweep:
This experiment varies the batch size (especially in the small-batch regime) to observe how GPU memory usage, latency, and throughput change. It measures the trade-off between utilization and resource constraints. Larger batches generally improve tokens-per-second due to better GPU saturation, but they also increase memory pressure and can hurt latency. Small batches minimize the risk of out-of-memory errors and keep response times low, but may leave computational capacity underutilized. The sweep identifies the optimal “knee point” where throughput gains are achieved without unstable memory usage or excessive latency. These results influence how to configure default batch sizes in production-grade environments. If even modest increases in batch size lead to out-of-memory issues, the system may require memory-saving techniques such as quantization, activation checkpointing, or altered pipeline partitions before further scaling is feasible.

Metrics and Instrumentation

Each run records per-prompt latency, throughput, and token-generation timings in the output logs. These measurements allow us to compute strong-scaling speedups and overall pipeline efficiency across ranks. Additional diagnostic information—such as NCCL connectivity checks, Slurm-provided rank metadata, and full environment summaries—is also captured at startup to assist with troubleshooting distributed execution issues. GPU visibility, driver/CUDA information, and host configuration details are printed automatically by each rank.

We also evaluated whether NVIDIA’s profiling tools could be incorporated into the workflow. In particular, we attempted to install and use Nsight Systems (nsys) on the compute nodes but were unsuccessful. The details of this issue are documented below, but as a result profiling was limited to in-application timing and logging.

Limitations and Challenges

A number of practical and system-level constraints influenced the design, execution, and observability of the experiments. The cluster’s hardware layout presented a primary limitation: each compute node contains a single NVIDIA T4 GPU with 16 GB of device memory. This restricted experimentation to comparatively small models and a two-stage pipeline configuration, without the option to employ tensor or data parallelism. Model execution therefore relied on selective GPU residency, with only the active transformer layers loaded onto the device and the remainder offloaded to host memory. While this approach conserved GPU resources, it increased pressure on system RAM and occasionally resulted in CPU-side out-of-memory failures.

Significant difficulties were also encountered when acquiring and downloading the model onto the cluster. Direct downloads from Hugging Face repeatedly failed due to missing system libraries, blocked network access, and authentication restrictions on the compute nodes. As a workaround, the model was downloaded on a local machine and then transferred to the cluster’s filesystem. Given the model size (approximately 7 GB), both the local download and the subsequent file transfer took about thirty minutes each. This introduced delays in experimentation as we switched models (from a 7B model to 3B) and highlighted the practical limitations of relying on remote model hosting in a constrained HPC environment.

As mentioned earlier, attempts to incorporate hardware-level profiling tools encountered systemic barriers. Nsight Systems, Nsight Compute, and related PerfWorks components could not be installed on compute nodes: NVIDIA's previously available `.run` and `.deb` installer packages have been removed from public repositories, resulting in reproducible `404 Not Found` errors across all tested versions. The CUDA-toolkit modules present on the cluster do not include `nsys`, and the node operating system lacks key libraries (such as `libdw` and `libunwind`) and tracing permissions required for kernel-level profiling. As a result, external profilers could not be deployed, and instrumentation was limited to application-level timing, throughput measurements, NCCL diagnostics, and environment logging.

Containerization and environment management also imposed important constraints. A fully self-contained Apptainer image was avoided due to size and quota limitations, so the workflow instead relies on a minimal PyTorch base image with runtime dependency installation. This approach required careful orchestration of Python environment bootstrapping, dependency caching, and network-aware pip configuration, as outbound connectivity on compute nodes is restricted. Early container executions revealed that the image lacked access to the host's NVIDIA driver stack: although distributed initialization completed successfully, both ranks failed during CUDA setup. The logs show that immediately after initializing `torch.distributed` and configuring local Hugging Face caches, the CUDA backend could not be activated, resulting in the following runtime exception emitted by both ranks:

RuntimeError: Found no NVIDIA driver on your system. Please check that you have an NVIDIA GPU and installed a driver from http://www.nvidia.com/Download/index.aspx

These failures were ultimately traced to missing driver bindings and incomplete library paths inside the container. Correcting the `--nv` bindings and adjusting `LD_LIBRARY_PATH` resolved the issue, enabling CUDA visibility and restoring correct GPU allocation within the Apptainer environment.

A final and more consequential limitation concerned model loading and execution on the cluster's single-GPU nodes. Even after resolving NVIDIA driver binding issues, the system was unable to load the model weights into device memory in a usable configuration. Initial attempts to place all layers of the 7B model on each GPU consistently triggered out-of-memory terminations, as reflected in the following Slurm output:

Some of the step tasks have been OOM Killed. srun: error: gpu-node1: task 0: Out Of Memory

To mitigate this, the model was partitioned such that the first half of the transformer layers were assigned to the first GPU node and the second half to the second node. However, this configuration failed at model construction time: the pretrained weights could not be mapped reliably to their assigned devices, and the loader reported unmapped parameters, for example:

ValueError: model.layers.0.self_attn.q_proj.weight doesn't have any device set.
and, on the other rank:
ValueError: model.embed_tokens.weight doesn't have any device set.

Even after downgrading to a smaller 3B model, the same failures occurred, indicating that the underlying issue was not model size but the incompatibility between multi-node device placement and the Hugging Face `from_pretrained` loading path within this constrained environment. The combined effect of limited GPU memory, the absence of multi-GPU nodes, and the framework's

inability to load disjoint model partitions across node boundaries ultimately prevented successful construction of the distributed model. As a result, execution could not proceed beyond this stage, and the project could not be completed under the current cluster configuration.

Hypothesized Results

Due to the unresolved model-loading and memory-allocation failures described previously, it was not possible to execute the full experimental suite on the cluster. As a consequence, the performance data presented below are hypothetical and should be interpreted as reasoned projections rather than empirical measurements. These projections are grounded in established characteristics of transformer inference workloads, expected scaling behavior in GPU-bound distributed systems, and observations from representative hardware such as NVIDIA's T4 accelerator.

Strong Scaling
Nodes: 1
Num Prompts: 5
Batch size: 1

Speedup	Wall clock time	Throughput (tokens/sec)	Parallel Efficiency	Resource Use	I/O Time (s)	Communication Time (s)	Preprocessing cost (s)
1.0 (baseline)	17	15 tok/s	1.0	~70% GPU	1	0	1

The single-node baseline indicates that the runtime is dominated by fixed overheads—such as initialization, preprocessing, and framework startup—rather than purely GPU-bound computation. A throughput of approximately 15 tokens per second and a wall-clock time of 17 seconds on a single T4 GPU suggests moderate utilization, reflecting the fact that small prompt batches do not sufficiently saturate the GPU.

Nodes: 2
Num Prompts: 5
Batch size: 1

Speedup	Wall clock time	Throughput (tokens/sec)	Parallel Efficiency	Resource Use	I/O Time	Communication Time	Preprocessing cost
1.3x	14	19 tok/s	0.65	~65% GPU	1.5	2	1

The projected two-node performance shows a speedup of 1.3× and a parallel efficiency of approximately 65%. While throughput increases from 15 to 19 tokens per second and wall-clock time decreases to 14 seconds, the gains are clearly sublinear. This behavior aligns with expectations for distributed inference on single-GPU nodes, where every additional node incurs communication, synchronization, and pipeline fill/drain overheads.

Weak Scaling
Nodes: 1

Num Prompts: 5
Batch size: 1

Speedup	Wall clock time	Throughput (tokens/sec)	Parallel Efficiency	Resource Use	I/O Time	Communication Time	Preprocessing cost
1.0 (baseline)	17	15 tok/s	1.0	~70% GPU	1	0	1

In the weak-scaling baseline, the performance metrics mirror those from the single-node strong-scaling case: a 17-second runtime and 15 tokens per second throughput.

Nodes: 2
Num Prompts: 10
Batch size: 1

Speedup	Wall clock time	Throughput (tokens/sec)	Parallel Efficiency	Resource Use	I/O Time	Communication Time	Preprocessing cost
1.3x	26	38 tok/s	0.90	~70% GPU	2	3	1.2

The hypothetical two-node weak-scaling results suggest that each GPU maintains nearly the same per-device throughput as in the single-node baseline, leading to an aggregate throughput of approximately 38 tokens per second and a parallel efficiency near 90%. Although the wall-clock time increases to 26 seconds—reflecting the doubled problem size—the per-node performance remains stable, indicating that the distributed inference pipeline is capable of effectively preserving throughput under proportionally increased workloads. These results would suggest that weak scaling is highly favorable on this cluster and that distributed inference becomes increasingly effective when the workload is sufficiently large.

Sensitivity Sweep (Batch size)
Nodes: 2
Num Prompts: 10
Batch size: 1

Speedup	Wall clock time	Throughput (tokens/sec)	Parallel Efficiency	Resource Use	I/O Time	Communication Time	Preprocessing cost
1.0 (baseline)	26	38 tok/s	0.90	~70% GPU	2	3	1.2

Under the hypothetical batch size 1 configuration, the system requires 26 seconds to process 10 prompts, achieving an aggregate throughput of 38 tokens per second across the two T4 GPUs. This corresponds to a relatively modest level of GPU utilization (around 70%), which is consistent with the expectation that very small micro-batches underutilize the hardware due to frequent kernel launches, higher relative Python overhead, and increased sensitivity to fixed costs such as tokenization and logging. This data indicates that while the distributed setup is functional, the system is operating in a conservative, latency-oriented regime that does not fully exploit the available compute capacity.

Nodes: 2
Num Prompts: 10
Batch size: 2

Speedup	Wall clock time	Throughput (tokens/sec)	Parallel Efficiency	Resource Use	I/O Time	Communication Time	Preprocessing cost
1.6x	8	61 tok/s	0.80	~85% GPU	1.5	2.5	1

When the batch size is increased to 2, the hypothetical results show a substantial improvement in performance. The wall-clock time drops to 8 seconds and throughput increases to 61 tokens per second, yielding an effective speedup of approximately 1.6× relative to the batch size 1 baseline. With a batch-size ratio of 2, this corresponds to a batch efficiency of about 0.80, which is a reasonable outcome in a distributed environment where communication and memory overheads prevent ideal scaling. These results suggest that batch size 2 strikes a favorable balance between hardware utilization, memory footprint, and communication overhead, and would likely be a strong default choice for throughput-oriented inference on this cluster.

Nodes: 2
Num Prompts: 10
Batch size: 4

Speedup	Wall clock time	Throughput (tokens/sec)	Parallel Efficiency	Resource Use	I/O Time	Communication Time	Preprocessing cost
1.7x	7	66 tok/s	0.43	~85% GPU	1.5	3	1

At batch size 4, the hypothetical configuration yields more modest incremental gains. The wall-clock time decreases further to 7 seconds and throughput increases to 66 tokens per second, corresponding to an effective speedup of approximately 1.7× over the batch size 1 baseline. Given the 4× increase in batch size, this results in a batch efficiency of roughly 0.43, reflecting significantly diminished returns compared to the move from batch 1 to batch 2. In conclusion, additional batching no longer translates into proportional performance gains, and may even push the system closer to memory limits and instability—consistent with the OOM and device-mapping issues we encountered when attempting to run a larger 7B model

Future Work

The most immediate priority is to resolve the model-loading and device-assignment errors that prevented successful execution of multi-stage distributed inference on the cluster. Addressing these issues—likely requiring adjustments to model partitioning logic, framework-level device mapping, or the underlying memory management strategy—is a prerequisite for all subsequent scaling studies. Once this barrier is removed, the system could be extended to support larger models and additional pipeline stages, enabling a more complete evaluation of multi-node inference. Although the current hardware lacks NVLink and offers only one GPU per node, future work could incorporate tensor-parallel strategies and leverage cluster upgrades with multi-GPU

nodes. This would enable experiments with significantly larger parameter counts and more sophisticated parallelization schemes.

A second area of improvement concerns container infrastructure. While the present work utilized a generic PyTorch-based Apptainer image, a slimmer, purpose-built image with fully baked-in dependencies would substantially reduce runtime overhead, eliminate the need for on-the-fly environment bootstrapping, and improve reproducibility. Such an image could incorporate preinstalled transformer libraries, CUDA and NCCL bindings, deterministic pip wheels, and curated versions of distributed-inference frameworks. This approach would also mitigate dependency drift and better align the execution environment with the model requirements.

Third, integrating automated performance instrumentation, particularly via tools such as NVIDIA Nsight Systems or Nsight Compute, would provide valuable insight into kernel-level behavior, communication bottlenecks, and utilization patterns. Although profiler installation was not feasible under current cluster constraints, a profiling-enabled environment would allow systematic tracing of computation, data movement, and synchronization across nodes. Such information would be vital for diagnosing performance anomalies, guiding optimization decisions, and validating the scaling characteristics of multi-stage pipelines.

Finally, memory optimization represents a critical domain for future exploration. Given the limited 16 GB VRAM on T4 GPUs, techniques such as activation checkpointing, more aggressive CPU or disk offloading strategies, quantization-aware inference, or mixed-precision modes beyond BF16 (e.g. FP8 weight quantization) may significantly improve model fit and performance. Investigating these strategies would help determine whether the cluster can effectively support larger contemporary models, or whether alternative model architectures or hardware platforms will be required.

References

- OpenLLaMA: An Open Reproduction of LLaMA
- Open_llama_3b_v2
- NVIDIA T4 Datasheet and Inference Benchmarks
- Shoeybi et al., “Megatron-LM: Training Multi-Billion Parameter Models Using Model Parallelism,” NeurIPS Systems 2019
- Narayanan et al., “GPT-3 Training: Scaling Laws and Infrastructure,” DeepSpeed Engineering Notes (2021)
- Kwon et al.,”Efficient Memory Management for Large Language Model Serving with PagedAttention” (2023)