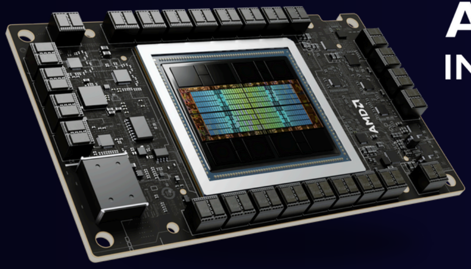
Supercharge Multi-Node Training Performance with Dell PowerEdge™ XE9680 and RoCE

| September 2024

In this blog, Metrum AI and Dell have partnered to demonstrate how to double training performance with the Llama 3.1 70B Model using a distributed system of Dell PowerEdge XE9680 Servers equipped with AMD Instinct MI300X Accelerators and Broadcom Thor 2 NICs enabled with RoCE.



AMD INSTINCT

DELL Technologies AMD METRUM AI

Introduction

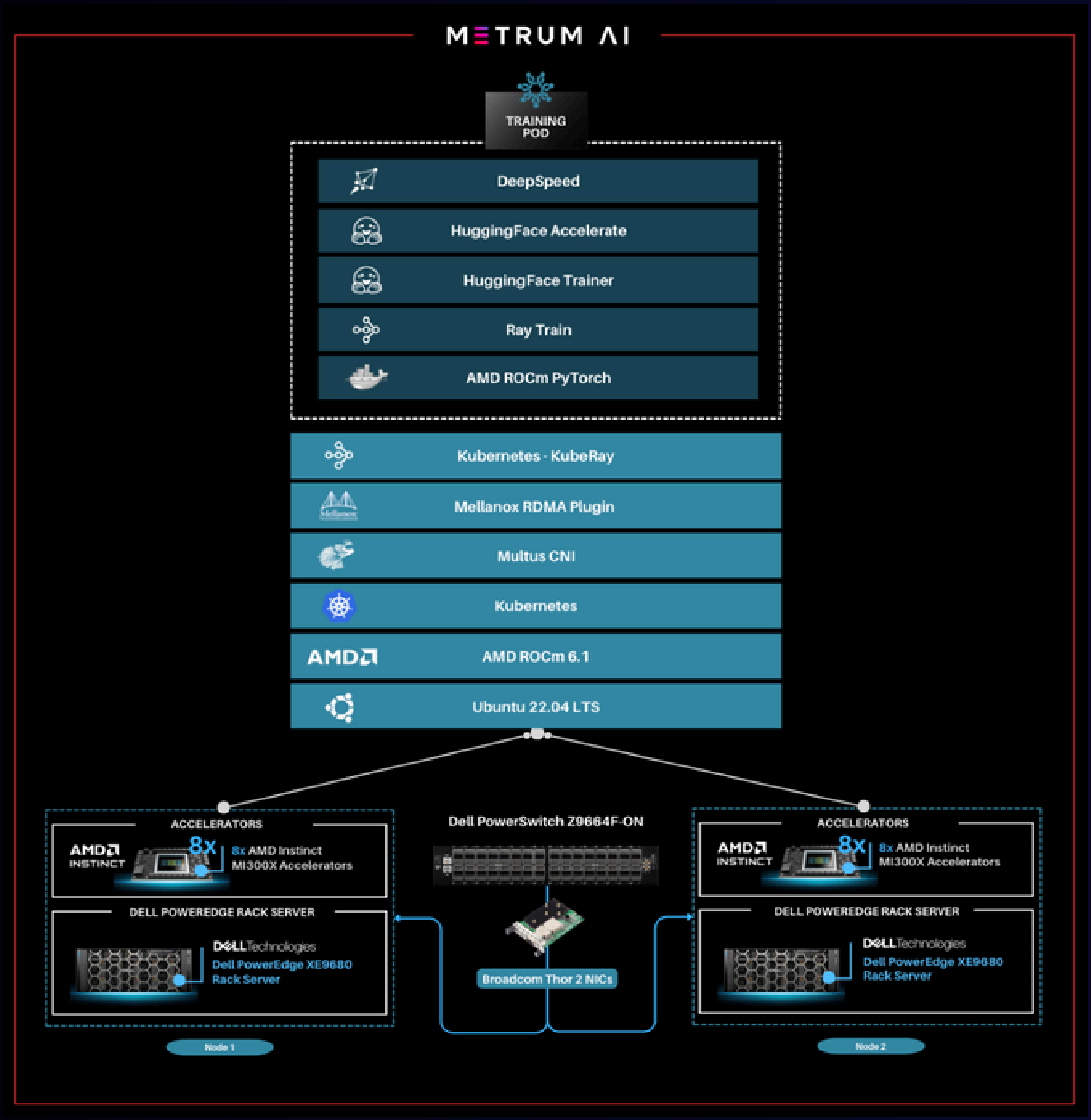
Training large language models at scale demands substantial computational power, memory, and time, often requiring massive server clusters. For example, training Meta's Llama 3 family of models required 24000 Nvidia H100 GPUs, equating to hundreds of millions in infrastructure costs.

Distributed training streamlines this process, improving resource usage by parallelizing tasks across multiple GPUs or devices and enhancing resource utilization. However, communication overhead can become a major bottleneck. **RDMA over Converged Ethernet** (**RoCE**) boosts data transfer efficiency and cuts latency, optimizing communication between devices. This accelerates the training of large language models, making it faster and more cost-effective, allowing you to fine-tune bigger models or reduce costs for models of the same size.

In this blog, we will demonstrate how to achieve optimal multi-node training performance with RoCE, significantly reducing both training time and infrastructure costs for industry-leading models. To showcase this, we will fine-tune state-of-the-art large language models, which involve updating pre-trained models with domain-specific data, which is critical for enterprises looking to customize models for their use cases. Using Dell PowerEdge XE9680 Servers with AMD Instinct MI300X Accelerators, we will showcase the following:

- How to deploy a distributed fine-tuning software with RoCE enabled to reduce training time
- How to fine-tune Llama 3.1 models using the PubMedQA, pqa-artificial medical dataset
- How to demonstrate distributed fine-tuning performance with and without RoCE

Distributed Fine-tuning Solution Architecture



This solution utilizes Dell PowerEdge Rack Servers and Broadcom Thor 2 Ethernet NICs, which support RoCE to facilitate high-speed, low-latency communication across nodes through direct memory access over Ethernet. By bypassing the CPU, RoCE minimizes overhead and speeds up GPU-to-GPU data transfers, significantly improving throughput and scalability for distributed fine-tuning. In this setup, two Dell PowerEdge servers equipped with AMD Instinct MI300X accelerators are linked through a Dell PowerSwitch Z9664F-ON, with each server featuring eight Broadcom Thor 2 NICs—one dedicated per accelerator.

The following key libraries are utilized to enable distributed fine-tuning:

- KubeRay: A Kubernetes operator that acts as an orchestration framework for training on a distributed hardware system. KubeRay is responsible for setting up a Ray cluster, i.e., setting up pods on the requested machines. It then handles the pod lifecycles, which involves checking that pods are alive as well as restarting pods if they are unresponsive. The pods can be started with different Docker images, but they must be able to communicate through Ray. KubeRay also handles job submissions, and provides a dashboard displaying hardware utilization metrics for all machines currently in the cluster.
- Accelerate: Provides the distributed training capabilities to PyTorch code. Accelerate simplifies the distribution of the current PyTorch fine-tuning code across multiple nodes. The Accelerate library primarily acts as a wrapper, meaning that the configuration options you can pass to the library are actually configuration options of underlying libraries, such as DeepSpeed.
- DeepSpeed: DeepSpeed optimizes distributed fine-tuning with advanced algorithms that manage memory and computation efficiently. ZeRO Stage 3, a key feature, partitions model states across devices, reducing memory consumption and enabling the training of larger models. DeepSpeed also minimizes communication overhead by distributing optimizer states, gradients, and parameters, which leads to faster convergence. Notably, with ZeRO, superlinear scaling has been observed, where adding accelerators results in greater-than-linear improvements in performance. This happens due to reduced communication bottlenecks and better utilization of accelerator memory, making DeepSpeed an ideal solution for scaling massive models across distributed environments.

This solution integrates additional AI libraries and drivers including Kubernetes, AMD ROCm™ 6.1 and PyTorch. To enhance network performance for distributed fine-tuning in Kubernetes environments, we utilized both Multus CNI and Mellanox RDMA Plugin.

With this comprehensive hardware and software infrastructure, we conducted rigorous performance testing to evaluate the system's capabilities in distributed fine-tuning scenarios. In the following section, we present our distributed fine-tuning performance testing results. We fine-tuned the Llama 3.1 8B and Llama 3.1 70B models using the medical domain PubMedQA, pqa-artificial dataset and tracked training loss as a measure of the fine-tuning progress, after which we collected the time to train.

Performance Testing Methodology

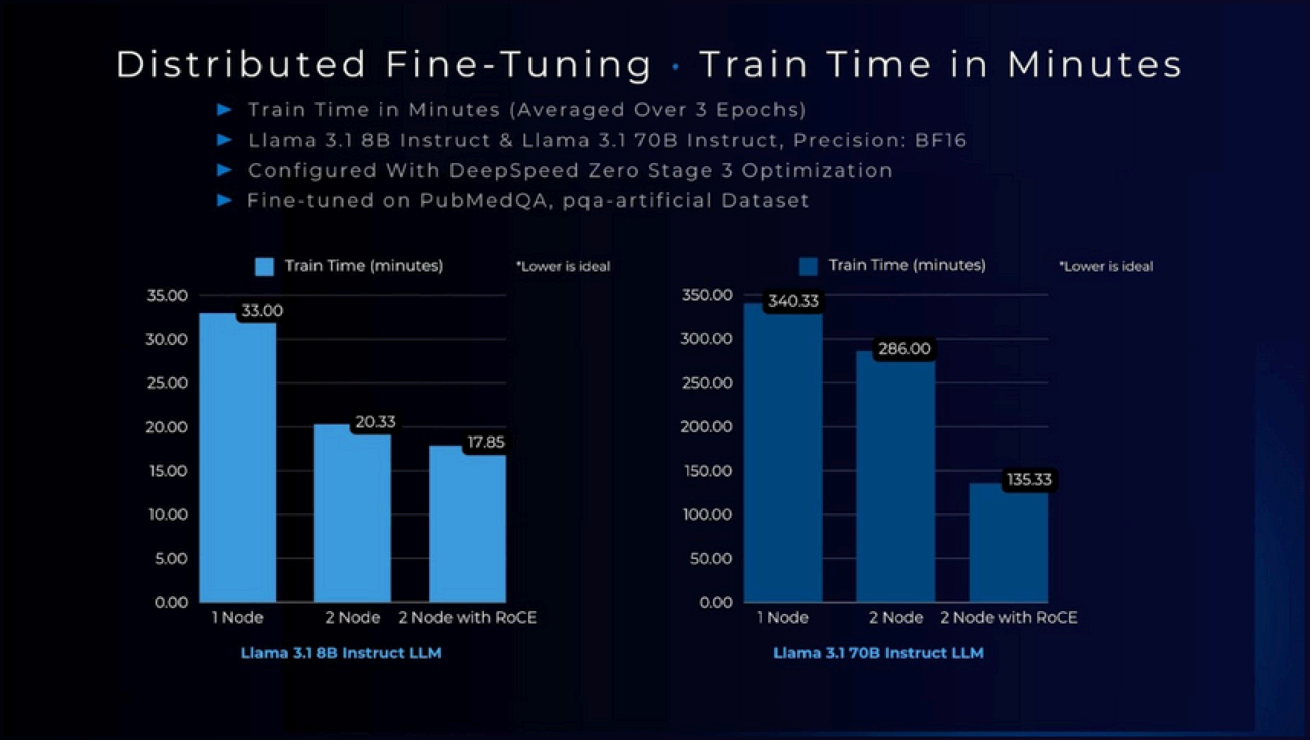
Our performance testing methodology incorporates an industry-specific dataset, several leading open-weight models, and industry-leading fine-tuning tools to accurately reflect enterprises' experience in fine-tuning.

We use the pubmed_qa, pqa-artificial dataset for fine-tuning the Llama 3.1 8B Instruct and Llama 3.1 70B Instruct models. This dataset was selected to represent domain-specific dataset that reflects enterprise fine-tuning workloads. The Llama 3.1 8B Instruct and Llama 3.1 70B Instruct models were selected because they are current, leading open-weight models, well positioned for enterprise fine-tuning due to their performance, quality, and commercial friendly licensing. Both models were fine-tuned with DeepSpeed ZeRO Stage3, an advanced memory optimization technique that partitions the model parameters, gradients, and optimizer states across all available accelerators, allowing for significant time and memory savings.

The batch size was selected to utilize more than 90% of GPU memory during fine-tuning, and all fine-tuning was performed with BF16 precision. Each model was fine-tuned for three epochs, and the final time taken for fine-tuning for one epoch was calculated by averaging times over the three measured epochs. This methodology focuses solely on the time required for training, excluding model loading, checkpointing, and evaluation.

Performance Results

The fine-tuning performance testing results are summarized in the charts and table below.



To assess the efficiency of distributed fine-tuning, we collected measurements of train time in minutes, comparing a baseline single node hardware configuration to a two node distributed hardware configuration, both with and without RoCE enabled. Here, the training time represents an average over the measured train time for three epochs.

As illustrated in the chart above, enabling RoCE in the two-node distributed hardware configuration resulted in a 1.14x reduction in training time for the Llama 3.1 8B model and more than 2x training time reduction for the Llama 3.1 70B model. This improvement can be attributed to RoCE's ability to bypass the CPU and enable direct memory access between GPUs across nodes, which significantly reduces communication overhead and allows more efficient GPU-to-GPU communication.

In the following section, we will:

1. Prepare the dataset.
2. Define the training parameters and settings.
3. Fine-tune industry-leading models on the given dataset using the defined parameters.

Follow these steps to replicate the results.

Setup

The fine-tuning performance testing results are summarized in the charts and table below.

Name	Dell PowerEdge XE9680
CPU	2x Intel Xeon Platinum 8460+
Memory	2 TB
Accelerators	8x AMD Instinct MI300X
Accelerators Count	8
OS	Ubuntu 22.04.4 LTS
Embedded NIC	Broadcom Gigabit Ethernet BCM5720
RoCE NICs	8x Broadcom BCM57608 2x200G PCIe (Thor 2) - 1 Per GPU
Ethernet Switch	Dell PowerSwitch Z9664F-ON withEnterprise SONiC Distribution by Dell Technologies

Step 1. Set up the distributed cluster.

Follow the k8s setup and introduce additional parameters for the k8s installation script. This involves configuring flannel, the networking fabric for kubernetes, with a user-selected specified network interface and utilizing the "host-gw" backend for networking. Then, Helm, the package manager for Kubernetes, will be used, and AMD plugins will be incorporated to grant access to AMD Instinct MI300X accelerators for the cluster pods.

Step 2. Install KubeRay and configure Ray Cluster.

The next steps include installing Kuberay, a Kubernetes operator, using Helm. The core of KubeRay comprises three Kubernetes Custom Resource Definitions (CRDs):

- **RayCluster**: This CRD enables KubeRay to fully manage the lifecycle of a RayCluster, automating tasks such as cluster creation, deletion, and autoscaling, while ensuring fault tolerance.
- **RayJob**: KubeRay streamlines job submission by automatically creating a RayCluster when needed. Users can configure RayJob to initiate job deletion once the task is completed, enhancing operational efficiency.
- **RayService**: RayService is made up of two parts: a RayCluster and a Ray Serve deployment graph. RayService offers zero-downtime upgrades for RayCluster and high availability.

```
helm repo add kuberay https://ray-project.github.io/kuberay-helm/
helm install kuberay-operator kuberay/kuberay-operator --version
1.0.0
```

This RayCluster consists of a head node followed by one worker node. In a YAML file, the head node is configured to run Ray with specified parameters, including the dashboard host and the number of accelerators,, as shown in the excerpt below. Here, the worker node is under the name "gpu-group”.

```
 ...
 headGroupSpec:
  rayStartParams:
   dashboard-host: "0.0.0.0"
   # setting num-gpus on the rayStartParams enables
   # head node to be used as a worker node
   num-gpus: "8"
  ...
```

The Kubernetes service is also defined to expose the Ray dashboard port for the head node. The deployment of the Ray cluster, as defined in a YAML file, will be executed using kubectl.

kubectl apply -f cluster.yml

Step 3. Fine-tune Llama 3.1 8B Model and Llama 3.1 70B with BF16 Precision.

You can either create your own dataset or select one from Hugging Face. The dataset must be available as a single json file with the specified format below.

```
# example 1
{"question":"Do mitochondria play a role in remodeling lace plant leaves during programmed cell death?", "context":"Programmed cell death (PCD) is the regulated death of cells within an organism. The lace plant (Aponogeton madagascariensis) produces perforations in its leaves through PCD. The role of mitochondria during PCD has been recognized in animals; however, it has been less studied during PCD in plants.", "answer":"Results depicted mitochondrial dynamics in vivo as PCD progresses within the lace plant, and highlight the correlation of this organelle with other organelles during developmental PCD."}

# example 2
{"question":"Syncope during bathing in infants, a pediatric form of water-induced urticaria?", "context":"Apparent life-threatening events in infants are a difficult and frequent problem in pediatric practice. The prognosis is uncertain because of risk of sudden infant death syndrome.", "answer":"\"Aquagenic maladies\" could be a pediatric form of the aquagenic urticaria."}
```

To recreate our results, make sure the models are configured as follows.

Llama 3.1 8B Model:

```
{
    "fp16": {
      "enabled": "auto"
    },
    "bf16":{
      "enabled":"auto"
    },
    "zero_optimization": {
      "stage": 3,
      "allgather_partitions": true,
      "allgather_bucket_size": 2e8,
      "overlap_comm": true,
      "reduce_scatter": true,
      "reduce_bucket_size": 2e8,
      "contiguous_gradients": true
    },
    "gradient_accumulation_steps": 1,
    "gradient_clipping": 1.0,
    "steps_per_print": 2000,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

Llama 3.1 70B Model:

```json
{
    "fp16": {
        "enabled": "auto"
    },
    "bf16": {
        "enabled": "auto"
    },
    "zero_optimization": {
        "stage": 3,
        "overlap_comm": true,
        "contiguous_gradients": true,
        "reduce_bucket_size": "auto",
        "stage3_prefetch_bucket_size": "auto",
        "stage3_param_persistence_threshold": "auto",
        "gather_16bit_weights_on_model_save": true,
        "round_robin_gradients": true
    },
    "gradient_accumulation_steps": "auto",
    "gradient_clipping": "auto",
    "steps_per_print": 10,
    "train_batch_size": "auto",
    "train_micro_batch_size_per_gpu": "auto",
    "wall_clock_breakdown": false
}
```

Jobs will be submitted to the Ray Cluster through the Ray Python SDK utilizing the Python script, job.py, provided below. This script uses the Llama 3.1 8B Model as an example.

```
# job.py

from ray.job_submission import JobSubmissionClient

# Update the <Head Node IP> to your head node IP/Hostname
client = JobSubmissionClient("http://<Head Node IP>:30265")

fine_tuning = (
    "python3 create_dataset.py \
    --dataset_path /train/dataset.json \
    --prompt_type 5 \
    --test_split 0.2 ;"
    "python3 train.py \
    --num-devices 16 \
    --batch-size-per-device 24 \
    --ds-config /code/ds_8b.json \
    --model-name meta-llama/Meta-Llama-3.1-8B-Instruct \
    --output-dir /train/ \
    --hf-token <HuggingFace Token> "
)
submission_id = client.submit_job(entrypoint=fine_tuning,)

print("Use the following command to follow this Job's logs:")
print(f"ray job logs '{submission_id}' --address http://<Head Node
IP>:30265 --follow")
```

This script initializes the JobSubmissionClient with the head node IP, sets parameters like prompt_type, batch size, and device count for training, and submits the job with these configurations.

The initial phase involves generating a fine-tuning dataset, which will be stored in a specified format. Configurations such as the prompt used and the ratio of training to testing data can be added. During the second phase, we will proceed with fine-tuning the model. For this fine-tuning, configurations such as the number of Accelerators to be utilized, batch size for each Accelerator, the model name as available on Hugging Face, Hugging Face API Token, and the number of epochs to fine-tune can all be specified.

Finally, in the third phase, we can start fine-tuning the model.

```
python3 job.py
```

The fine-tuning jobs can be monitored using Ray CLI and Ray Dashboard.

- Using Ray CLI:
  - Retrieve submission ID for the desired job.
  - Use the command below to track job logs.

```
ray job logs <Submission ID> --address http://<Head Node IP>:30265 -
-follow
```

Ensure to replace <Submission ID> and <Head Node IP> with the appropriate values.

- Using Ray Dashboard:
  - To check the status of fine-tuning jobs, simply visit the Jobs page on your Ray Dashboard at <Head Node IP>:30265 and select the specific job from the list.

For more detailed information on how to set up a distributed cluster with RoCE support enabled, fine-tune Llama 3.1 models on the distributed cluster, and track real-time progress of the distributed fine-tuning process using Ray Dashboard and TensorBoard, please request access to the reference code at contact@metrum.ai.

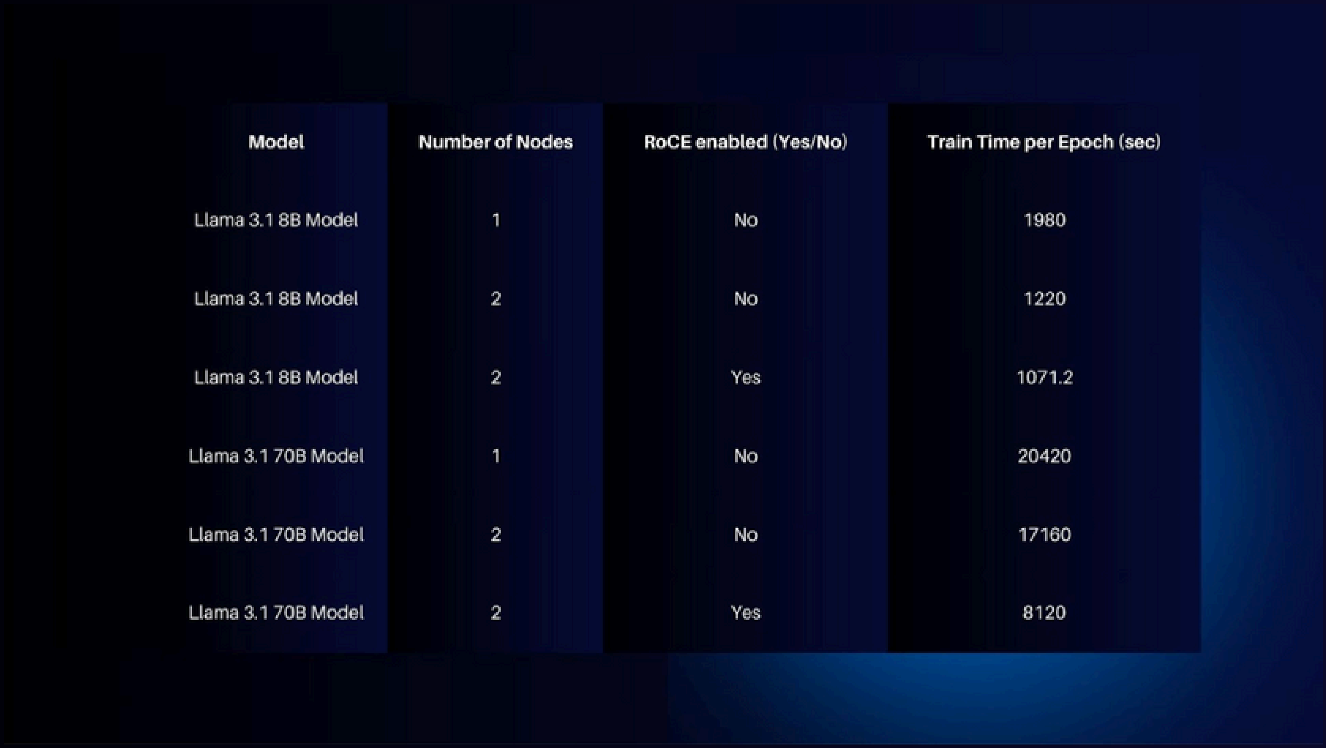
Summary

The Dell PowerEdge XE9680 Server, featuring AMD Instinct MI300X Accelerators, offers enterprises cutting-edge infrastructure for fine-tuning AI solutions tailored to industry-specific needs using their proprietary data, as well as for developing pretrained models. In this blog, we demonstrated how you can accelerate training and fine-tuning by utilizing multi-node hardware clusters with RoCE, achieving the following:

- Deployed a distributed fine-tuning software with RoCE enabled to reduce training time
- Fine-tuned Llama 3.1 models using the PubMedQA, pqa-artificial medical dataset
- Demonstrated distributed fine-tuning performance with and without RoCE

Additional Information

The chart below provides additional time-to-train measurements collected during performance testing.



Model	Number of Nodes	RoCE enabled (Yes/No)	Train Time per Epoch (sec)
Llama 3.1 8B Model	1	No	1980
Llama 3.1 8B Model	2	No	1220
Llama 3.1 8B Model	2	Yes	1071.2
Llama 3.1 70B Model	1	No	20420
Llama 3.1 70B Model	2	No	17160
Llama 3.1 70B Model	2	Yes	8120

References

- PubMedQA, pqa-artificial
- https://ai.meta.com/blog/meta-llama-3/

AMD images: AMD Library, https://library.amd.com/account/dashboard/
Dell images: Dell.com