Table of Contents

Inleiding

In dit document wordt beschreven hoe je een gedistribueerde inferentie-opstelling kunt opzetten met vLLM, waarbij Ray of Message Passing Interface (MPI) als backend wordt gebruikt, om te draaien over meerdere machines. Dit maakt het mogelijk om grootschalige taalmodellen efficiënt te hosten en te schalen.

Dit document is samengesteld uit een reeks LogSeq-notities en is bedoeld om praktische handvatten te bieden, niet als een volledige handleiding.

Aan de hand van het aantal pagina's is te merken dat mijn voorkeur en interesse uitgaat naar de implementatie met de Ray-backend boven die met MPI. Het Ray open-source framework biedt:

> *Grote taalmodellen (LLM's) en generatieve AI veranderen industrieën in hoog tempo en eisen een verbazingwekkende hoeveelheid rekenkracht. Ray biedt een distributed compute framework om deze modellen te schalen, waardoor ontwikkelaars modellen sneller en efficiënter kunnen trainen en implementeren. Met gespecialiseerde libraries voor datastreaming, training, fine-tuning, hyperparameter-tuning en serving vereenvoudigt Ray het proces van het ontwikkelen en implementeren van grootschalige AI-modellen.*

Standaardzaken zoals versiecompatibiliteit, firewall-instellingen, activering van services, SELinux, het configureren van Nvidia GPU's en problemen met modelformaten (bijvoorbeeld quantized models) worden als bekende kennis beschouwd en daarom niet behandeld in dit document.

Opzetten met MPI

MPI (Message Passing Interface) is een standaard voor gedistribueerde computing, vaak wordt dit gebruikt in high-performance computing (HPC).

Installeer vLLM en MPI

Er bestaan meerdere MPI-implementaties. In dit document wordt er gebruik gemaakt van Open MPI.

Red Hat Enterprise Linux biedt Open MPI, een open source en vrij beschikbare implementatie van zowel de MPI-1- als de MPI-2-standaarden. Het combineert technologieën en middelen uit verschillende andere projecten (FT-MPI, LA-MPI, LAM/MPI en PACX-MPI) om de beste beschikbare MPI-library te bouwen.

RHEL of een Ubuntu installatie ziet er als volgt uit:

```
sudo dnf install openmpi
sudo apt-get install openmpi-bin libopenmpi-dev
```

Installeer vLLM op alle machines:

```
pip install vllm
```

MPI-Communicatie

Zorg ervoor dat de machines met elkaar kunnen communiceren via MPI. Hiervoor kan een hostfile geconfigureerd worden met de hostnamen of IP-adressen van de machines, bijvoorbeeld:

```
machine1
machine2
machine3
```

Start de vLLM-server met MPI

Bij het starten van de vLLM-server, specificeer MPI als de distributed backend. Gebruik *mpirun* om de vLLM-server te starten over meerdere machines. Voorbeeld voor 3 machines:

```
mpirun -np 3 -hostfile hostfile.txt vllm serve <model_name> --tensor-
parallel-size 4 --distributed-executor-backend mpi
```

Verifiëren van de opstelling

Om te controleren of de opstelling correct, kunt u een testverzoek naar de vLLM-server sturen. Dit kan bijvoorbeeld met *curl*:

```
curl -X POST http://<head_node_ip>:8000/generate -H "Content-Type:
application/json" -d '{"prompt": "Hello, world!"}'
```

Vervang <*head_node_ip*> door het IP-adres van de *head* node.

Opzetten met Ray

Ray is een open-source framework voor gedistribueerde computing, ideaal voor het schalen van Python-applicaties.

Installeren vLLM en Ray

Op alle machines, voer het volgende commando uit in de terminal:

```
pip install vllm ray
```

Start de Ray-cluster

Eén van de machines dienst als *head node*. Op deze machine , voer uit:

```
ray start --head
```

Dit start de *head node* en geeft u een adres, bijvoorbeeld `192.168.1.100:6379`

Op de andere machines (worker nodes), voer uit:

```
ray start --address=192.168.1.100:6379
```

Vervang natuurlijk het adres door het adres van de *head node*.

Start de vLLM-server

Bij het starten van de vLLM-server, specificeer Ray als de distributed backend. Op de head node, voer het volgende commando uit:

```
vllm serve <model_name> --tensor-parallel-size <num_gpus> --distributed-executor-backend ray
```

Bijvoorbeeld:

```
vllm serve meta-llama/Meta-Llama-3-8B-Instruct --tensor-parallel-size 4 --distributed-executor-backend ray
```

Dit voorbeeld verdeelt het model over 4 GPU’s in de Ray-cluster.

vLLM met Ray in Containers

Natuurlijk is het mogelijk om een vLLM met Ray backend middels containers te implementeren. Hieronder een korte beschrijving.

Creeër een Dockerfile

In onderstaand Dockerfile voorbeeld is een ubuntu implementie gebruikt, dit kan natuurlijk vervangen worden door Red Hat Universal Base Image (UBI).

```
FROM python:3.10-slim

# Install system dependencies
RUN apt-get update && apt-get install -y \
    build-essential \
    && rm -rf /var/lib/apt/lists/*

# Install vLLM and Ray
RUN pip install vllm ray

# Expose Ray ports (head node dashboard and communication)
EXPOSE 6379 8265 8000

# Command to keep container running (will be overridden in deployment)
CMD ["bash"]
```

Bouw de container image middels:

```
docker build -t vllm-ray:latest .
```

Start een Ray-cluster met containers

Kies een machine als head-node en start de container op deze machine als head-node:

```
docker run -d --name ray-head -p 6379:6379 -p 8265:8265 -p 8000:8000 \
vllm-ray:latest ray start --head --port 6379 --dashboard-host 0.0.0.0
```

Op de andere machines (worker nodes) start een container die verbinding maakt met de head-node (vervang <*head-ip*> met de head-node IP-adres):

```
docker run -d --name ray-worker vllm-ray:latest ray start \
--address=<head-ip>:6379
```

Start vLLM Server in Head Container

Middels een *docker exec* ga in de head container en start de vLLM server:

```
docker exec -it ray-head bash

vllm serve <model_name> --tensor-parallel-size <num_gpus> --distributed-
executor-backend ray
```

Natuurlijk is het mogelijk om de Dockerfile CMD aan te passen met dit commando en op deze manier vLLM server te starten.

vLLM en Ray backend middels Python

Natuurlijk zijn er allerlei manieren om middels python vLLM en Ray te gebruiken.

Commandline Module

De *vllm.entrypoints.openai.api_server* ondersteunt een vlag genaamd *--worker-use-ray*, die vLLM instrueert om Ray te gebruiken voor het beheren van worker-processen (bijvoorbeeld het verdelen van het model over GPU's). Voorbeeld:

```
python -m vllm.entrypoints.openai.api_server \
    --model /srv/Meta-Llama-3-8B-Instruct-8bit \
    --tensor-parallel-size 2 \
    --worker-use-ray
```

Waarbij *--worker-use-ray* vLLM de opdracht geeft om Ray te gebruiken voor gedistribueerde uitvoering in plaats van de standaard multiprocessing-backend. En *--tensor-parallel-size 2* het model splitst over 2 GPU's, die Ray zal beheren binnen het cluster.

Programmatisch Uitvoeren

Natuurlijk kan het ook vanuit een python programma gestart worden. Hieronder een kort voorbeeld:

```
from vllm import LLM, SamplingParams

llm = LLM(
    model="/srv/Meta-Llama-3-8B-Instruct-8bit",
    tensor_parallel_size=2,
    distributed_executor_backend="ray"
)

# Example inference
output = llm.generate("Hello, world!", SamplingParams(max_tokens=50))
print(output)
```

Ray Framework

Ray Dashboard

De Ray-dashboard is een essentieel hulpmiddel binnen de Ray-distributed computing framework, ontworpen om gebruikers te helpen bij het monitoren en beheren van hun Ray-clusters. Het biedt real-time inzichten in de clusterstatus, resourcegebruik en de prestaties van lopende jobs en taken.

Ga naar *<head_node_ip>:8265* in je webbrowser om toegang te krijgen. Zorg ervoor dat je cluster draait en dat je verbinding kunt maken met het IP-adres (firewall).

Het is ook mogelijk om een dashboard poort in te stellen middels –dashboard-port, bijvoorbeeld via *ray start --head --dashboard-port 8265*. Voor Docker-gebruikers wordt aanbevolen *--dashboard-host=0.0.0.0* te gebruiken om externe toegang mogelijk te maken.

Als Prometheus en Grafana zijn ingesteld, biedt de **Metrics-weergave** diepgaande inzichten in resourcegebruik en prestaties. Zonder deze setup is deze weergave beperkt, maar de andere weergaven bieden al waardevolle basisinformatie.

Let op: de dashboard-gegevens zijn **alleen beschikbaar** zolang het cluster draait, dus voor productie is aanvullende logging aanbevolen!

De Ray Dashboard pagina van de Ray documentatie biedt enkele youtube filmpjes met meer achtergrond.

Ray Example Gallery

Een flinke voorbeeld lijst van implementatie middels Ray zijn te vinden in de example gallery.

Ray Production Guide

De aanbevolen methode om Ray Serve te gebruiken in productie is op Kubernetes (of OpenShift) middels KubeRay en RayService custom resource.

Hier is de ingang na de productie guide te vinden.

Vervolg onderzoek ingangen

De volgende items zijn n.a.v. enkele issues tijdens het starten van voorgaande beschrijving. Deze items hebben geholpen om een bepaald doel te halen, maar zijn ook een goede ingang voor verder onderzoek.

RayCluster Kubernetes

Er zijn een aantal quickstart guides te vinden op de Ray documentatie pagina die helpen om RayCluster, RayJob en RayService op een lokale K8S te draaien (b.v. met KubeRay Operator).

Zelf heb ik de quickstart guides doorlopen en deze bieden een leuke ingang tot de mogelijkheden van Ray (internet toegang is wel nodig).

Convert Model

Als de kwantisatie of indeling niet compatibel is, converteer het model dan naar een ondersteunde indeling (bijvoorbeeld safetensors of non-quantized PyTorch).

N.a.v. niet beschikbare libraries die quantized models niet ondersteunen.

Bijvoorbeeld via laden met transformers (model in huggingface format):

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model = AutoModelForCausalLM.from_pretrained("/srv/Meta-Llama-3-8B-
Instruct-8bit")
model.save_pretrained("/srv/Meta-Llama-3-8B-Instruct-converted")
```

Als het model een Hugging Face Transformers-model is, kun je het laden met *LlamaForCausalLM.from_pretrained*. Dit is een gebruikelijke manier om Llama-modellen te laden.

PyTorch biedt ook een functie *torch.dequantize* om een gekwantiseerde tensor te dekwantiseren. Voor een volledig model:

- Laad het gekwantiseerde model: `quantized_model = torch.load("quantized_model.pt")`
- Dekwantiseer elke parameter: `for param in quantized_model.parameters(): if isinstance(param, torch.nn.quantized.Tensor): param.data = torch.dequantize(param)`
- Sla het dekwantiseerde model op: `torch.save(quantized_model, "unquantized_model.pt")`

Als het model niet in *Hugging Face-formaat* is opgeslagen maar bijvoorbeeld als een standalone .pt-bestand (bijvoorbeeld model.pt), kun je het direct laden met *torch.load*:

```
import torch

# Laad de state_dict direct
state_dict = torch.load('/srv/Meta-Llama-3-8B-Instruct-8bit/model.pt')

# Dekwantiseer
```

```
dequantized_state_dict = {
    k: torch.dequantize(v) if v.is_quantized else v
    for k, v in state_dict.items()
}
```

Gebruik voor inference

Als je het model alleen voor inference wilt gebruiken, is het vaak niet nodig om *torch.dequantize* handmatig toe te passen. De forward pass van het model converteert de uitvoer intern naar floating-point, zelfs als de gewichten gekwantiseerd zijn

Fine-tuning

Als je het model wilt dekwantiseren voor fine-tuning op floating-point hardware, is bovenstaande aanpak geschikt. Zorg ervoor dat je het model daarna opslaat of verder verwerkt.

Let op: Dekwantiseren kan **precisieverlies** veroorzaken, wat de modelprestaties kan beïnvloeden.