(deployment-k8s)=

Using Kubernetes

Using Kubernetes to deploy vLLM is a scalable and efficient way to serve machine learning models. This guide will walk you through the process of deploying vLLM with Kubernetes, including the necessary prerequisites, steps for deployment, and testing.

Prerequisites

Before you begin, ensure that you have the following:

- A running Kubernetes cluster
- NVIDIA Kubernetes Device Plugin (`k8s-device-plugin`): This can be found at `https://github.com/NVIDIA/k8s-device-plugin/`
- Available GPU resources in your cluster

Deployment Steps

1. Create a PVC, Secret and Deployment for vLLM

PVC is used to store the model cache and it is optional, you can use hostPath or other storage options

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
```

```
  metadata:
    name: mistral-7b
    namespace: default
  spec:
    accessModes:
    - ReadWriteOnce
    resources:
      requests:
        storage: 50Gi
    storageClassName: default
    volumeMode: Filesystem
```

Secret is optional and only required for accessing gated models, you can skip this step if you are not using gated models

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: hf-token-secret
  namespace: default
type: Opaque
stringData:
  token: "REPLACE_WITH_TOKEN"
```

Next to create the deployment file for vLLM to run the model server. The following example deploys the `Mistral-7B-Instruct-v0.3` model.

Here are two examples for using NVIDIA GPU and AMD GPU.

NVIDIA GPU:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mistral-7b
  namespace: default
  labels:
    app: mistral-7b
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mistral-7b
  template:
    metadata:
      labels:
        app: mistral-7b
    spec:
      volumes:
      - name: cache-volume
```

```
      persistentVolumeClaim:
        claimName: mistral-7b
      # vLLM needs to access the host's shared memory for tensor parallel inference.
      - name: shm
        emptyDir:
          medium: Memory
          sizeLimit: "2Gi"
      containers:
      - name: mistral-7b
        image: vllm/vllm-openai:latest
        command: ["/bin/sh", "-c"]
        args: [
            "vllm serve mistralai/Mistral-7B-Instruct-v0.3 --trust-remote-code --enable-chunked-prefill
--max_num_batched_tokens 1024"
        ]
        env:
        - name: HUGGING_FACE_HUB_TOKEN
          valueFrom:
            secretKeyRef:
              name: hf-token-secret
              key: token
        ports:
        - containerPort: 8000
        resources:
          limits:
            cpu: "10"
            memory: 20G
```

```
            nvidia.com/gpu: "1"
          requests:
            cpu: "2"
            memory: 6G
            nvidia.com/gpu: "1"
        volumeMounts:
        - mountPath: /root/.cache/huggingface
          name: cache-volume
        - name: shm
          mountPath: /dev/shm
        livenessProbe:
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 60
          periodSeconds: 10
        readinessProbe:
          httpGet:
            path: /health
            port: 8000
          initialDelaySeconds: 60
          periodSeconds: 5
```

AMD GPU:

You can refer to the `deployment.yaml` below if using AMD ROCm GPU like MI300X.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mistral-7b
  namespace: default
  labels:
    app: mistral-7b
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mistral-7b
  template:
    metadata:
      labels:
        app: mistral-7b
    spec:
      volumes:
      # PVC
      - name: cache-volume
        persistentVolumeClaim:
          claimName: mistral-7b
      # vLLM needs to access the host's shared memory for tensor parallel inference.
      - name: shm
        emptyDir:
```

```
      medium: Memory
      sizeLimit: "8Gi"
    hostNetwork: true
    hostIPC: true
    containers:
    - name: mistral-7b
      image: rocm/vllm:rocm6.2_mi300_ubuntu20.04_py3.9_vllm_0.6.4
      securityContext:
        seccompProfile:
          type: Unconfined
        runAsGroup: 44
        capabilities:
          add:
          - SYS_PTRACE
      command: ["/bin/sh", "-c"]
      args: [
                          "vllm serve mistralai/Mistral-7B-v0.3 --port 8000 --trust-remote-code
--enable-chunked-prefill --max_num_batched_tokens 1024"
      ]
      env:
      - name: HUGGING_FACE_HUB_TOKEN
        valueFrom:
          secretKeyRef:
            name: hf-token-secret
            key: token
      ports:
      - containerPort: 8000
```

```
        resources:
          limits:
            cpu: "10"
            memory: 20G
            amd.com/gpu: "1"
          requests:
            cpu: "6"
            memory: 6G
            amd.com/gpu: "1"
        volumeMounts:
        - name: cache-volume
          mountPath: /root/.cache/huggingface
        - name: shm
          mountPath: /dev/shm
```

You can get the full example with steps and sample yaml files from <https://github.com/ROCm/k8s-device-plugin/tree/master/example/vllm-serve>.

2. Create a Kubernetes Service for vLLM

Next, create a Kubernetes Service file to expose the `mistral-7b` deployment:

```yaml
apiVersion: v1
kind: Service
metadata:
```

```
  name: mistral-7b
  namespace: default
spec:
  ports:
  - name: http-mistral-7b
    port: 80
    protocol: TCP
    targetPort: 8000
  # The label selector should match the deployment labels & it is useful for prefix caching feature
  selector:
    app: mistral-7b
  sessionAffinity: None
  type: ClusterIP
```

3. Deploy and Test

Apply the deployment and service configurations using `kubectl apply -f <filename>`:

```console
kubectl apply -f deployment.yaml
kubectl apply -f service.yaml
```

To test the deployment, run the following `curl` command:

```console
```

```
curl http://mistral-7b.default.svc.cluster.local/v1/completions \
  -H "Content-Type: application/json" \
  -d '{
        "model": "mistralai/Mistral-7B-Instruct-v0.3",
        "prompt": "San Francisco is a",
        "max_tokens": 7,
        "temperature": 0
      }'
```

If the service is correctly deployed, you should receive a response from the vLLM model.

Conclusion

Deploying vLLM with Kubernetes allows for efficient scaling and management of ML models leveraging GPU resources. By following the steps outlined above, you should be able to set up and test a vLLM deployment within your Kubernetes cluster. If you encounter any issues or have suggestions, please feel free to contribute to the documentation.