Enabling GPU Time-Slicing on EKS

Below is a step-by-step guide on how to configure GPU time-slicing (GPU “slicing”) in Amazon EKS, along with notes on using it in clusters that leverage the Karpenter autoscaler.

Implementation

Prerequisites

1. An **EKS cluster** with **NVIDIA GPU-backed EC2 instances**
2. **kubectl**, **AWS CLI**, and **Helm** installed locally.
3. The **NVIDIA device plugin** Helm chart reference (`nvdp/nvidia-device-plugin`).
4. **Karpenter** already configured, if you intend to autoscale GPU nodes with Karpenter.

Label the GPU Nodes

By labeling the GPU node(s), you can explicitly target them for GPU workloads and the device plugin:

```
kubectl label node <gpu-node-name> eks-node=gpu
```

(Replace `<gpu-node-name>` with the actual node name.)

Install the NVIDIA Device Plugin

Deploy the NVIDIA device plugin as a DaemonSet using Helm:

```
helm repo add nvdp https://nvidia.github.io/k8s-device-plugin
helm repo update

helm upgrade -i nvdp nvdp/nvidia-device-plugin \
  --namespace nvidia-device-plugin \
  --create-namespace \
  --version 0.17.0
```

This step exposes `nvidia.com/gpu` as a schedulable resource in your cluster.

Enable GPU Time-Slicing

1. **Create a ConfigMap** that sets the desired number of time-sliced “virtual” GPUs per physical GPU. For example, `replicas: 10` means each physical GPU will appear as 10 vGPUs:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-device-plugin
  namespace: kube-system
data:
  any: |-
    version: v1
    flags:
      migStrategy: none
    sharing:
      timeSlicing:
        resources:
          - name: nvidia.com/gpu
            replicas: 10
```

```
kubectl apply -f nvidia-device-plugin.yaml
```

2. **Redeploy/Update** the device plugin using Helm to apply the new time-slicing config:

```
helm upgrade -i nvdp nvdp/nvidia-device-plugin \
  --namespace nvidia-device-plugin \
  --create-namespace \
  --version 0.17.0
  --set config.name=nvidia-device-plugin \
  --force
```

3. **Verify** that Kubernetes sees multiple vGPUs. You should see an increased GPU capacity (e.g., 40 if you have 4 physical GPUs and `replicas: 10`):

```
kubectl get nodes -o json | jq -r '.items[]
  | select(.status.capacity."nvidia.com/gpu" != null)
  | {name: .metadata.name, capacity: .status.capacity}'
```

Deploy GPU Workloads (Time-Sliced)

When you create deployments or pods that request GPUs, use:

```yaml
resources:
  limits:
    nvidia.com/gpu: 1
```

Although each pod “sees” `1` GPU request, that actually corresponds to 1 time-slice (vGPU). You can safely schedule multiple pods on a single physical GPU, as orchestrated by the plugin’s time-slicer.

Scaling with Karpenter Autoscaler

- Karpenter sees `nvidia.com/gpu` as an extended resource. The time-slicing plugin makes each physical GPU appear as multiple slices, so pods requesting `nvidia.com/gpu: 1` can fit onto fewer physical GPUs. Often reducing the need to scale out new GPU instances.
- **Implementation Steps:**
    1. In your **Karpenter NodePool**, include instance types with GPUs (e.g., `p3.8xlarge`).

Ensure your Provisioner is configured to accept pods requesting `nvidia.com/gpu`. For example:

```
spec:
  requirements:
    - key: "node.kubernetes.io/instance-type"
      operator: In
      values: ["p3.8xlarge"]
    - key: "nvidia.com/gpu"
      operator: Exists
```

    2. As pods require GPU resources, Karpenter checks if existing GPU nodes can accommodate them (including time-sliced vGPUs). If not, it automatically spins up new GPU nodes.
    3. The device plugin DaemonSet runs on every new GPU node, enabling time-slicing automatically.

Observing and Verifying

- Use `kubectl get pods` to ensure all pods are running and not stuck in `Pending`.
- Use AWS Systems Manager Session Manager (SSM) on the GPU node and run `nvidia-smi`. You should see multiple processes sharing each GPU, confirming that time-slicing is working.

Conclusion

By configuring the **NVIDIA device plugin** with **time-slicing**, you can split a single physical GPU into multiple vGPUs. This approach maximizes GPU usage efficiency, can help reduce GPU costs, and still allows for pod-level scheduling via Kubernetes. For autoscaling, **Karpenter** can recognize these virtual GPU resources and scale GPU nodes only when needed, further optimizing costs and performance.

References

1. **GPU sharing on Amazon EKS with NVIDIA time-slicing and accelerated EC2 instances**
   An AWS blog post demonstrating how to implement time-slicing for GPUs on Amazon EKS, including prerequisites, configuration steps, and best practices.
2. **NVIDIA/k8s-device-plugin (GitHub)**
   The official repository for NVIDIA's Kubernetes device plugin, covering installation, configuration (including time-slicing), and advanced usage for GPU workloads in Kubernetes.