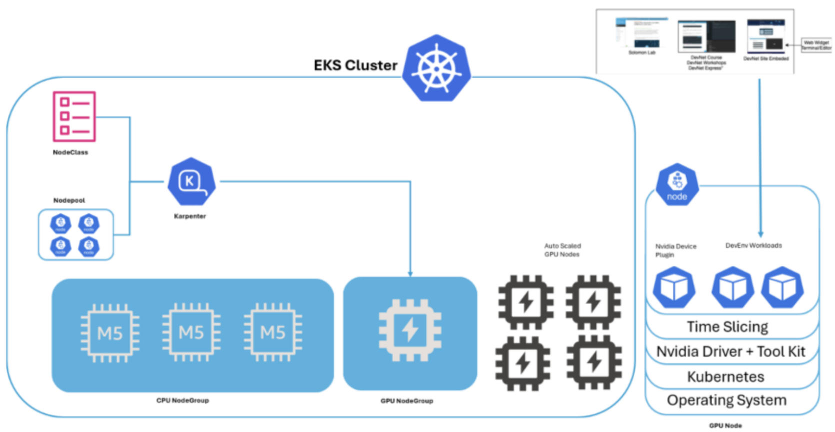
GPU Slicing Optimization Guide for EKS Clusters

AUTHOR-RESEARCHER: OLUWAPELUMI FAPOHUNDA (OPSFLEET RESEARCH)

GPU Slicing, often referred to as GPU Partitioning or NVIDIA's Multi-Instance GPU (MIG) technology, allows a single GPU to be divided into smaller, isolated instances. This technique enhances resource utilization and lowers costs by allocating the right-sized GPU resources for different workloads.



Why Use GPU Slicing?

- Lower GPU Costs: Efficient use of GPU resources reduces infrastructure spending.
- Better Utilization: Avoid underutilized GPUs by splitting them into manageable slices.
- Improved Isolation: Each GPU slice operates independently, minimizing resource contention.
- Flexible Allocation: Allocate just the right amount of GPU power based on workload requirements.

Prerequisites

To get started with GPU Slicing in your EKS cluster, you'll need:
- A Kubernetes cluster on EKS with supported NVIDIA GPUs (e.g., A100, H100).
- NVIDIA GPU Operator installed.
- Kubernetes version 1.22 or newer.
- CUDA 11.0 or higher.
- Supported GPU compute modes configured.

Compatible GPUs for GPU Slicing

GPU Model	MIG Support	Compute Modes
NVIDIA A100	Full Support	1G.5GB, 2G.10GB, 3G.20GB
NVIDIA H100	Partial Support	1G.10GB, 2G.20GB

Setting Up GPU Slicing on EKS

1. Enable MIG Support with the GPU Operator

Update the ClusterPolicy for the GPU Operator to enable MIG support:

```
apiVersion: gpu.nvidia.com/v1
kind: ClusterPolicy
metadata:
  name: gpu-cluster-policy
spec:
  mig:
    enable: true
  daemonsets:
    gpu-device-plugin:
      enabled: true
    gpu-operator-validator:
      enabled: true
```

2. Define GPU Slicing Configurations

Use a ConfigMap to set up MIG configurations tailored to your workloads:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: nvidia-mig-config
  namespace: gpu-operator
data:
  config.yaml: |-
    version: v1
    profile-names:
      - name: small-instance
        devices:
          - mig-config: 1g.5gb
```

3. Integrate GPU Slicing with Karpenter

Leverage Karpenter for dynamic and cost-effective GPU node provisioning. Below is a sample NodePool configuration:

```yaml
apiVersion: karpenter.sh/v1beta1
kind: NodePool
metadata:
  name: gpu-slice-pool
spec:
  template:
    spec:
      requirements:
        - key: "node.kubernetes.io/instance-type"
          operator: In
          values: ["p4d.24xlarge"]
      resources:
        requests:
          nvidia.com/gpu: 1
```

Deployment Best Practices

Request GPU Slices for Pods

Pods can request specific MIG slices by defining resource limits:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: gpu-slice-workload
spec:
  containers:
  - name: ml-workload
    resources:
      limits:
        nvidia.com/mig-1g.5gb: 1
```

Monitoring and Optimizing GPU Usage

Tools for Monitoring

- NVIDIA DCGM Exporter: Tracks GPU metrics for Prometheus.
- Grafana Dashboards: Visualize GPU utilization and performance.

Key Commands

Check MIG status:

```
kubectl exec -it nvidia-gpu-operator-daemonset-xxxxx -- nvidia-smi -L
```

Verify GPU slice usage:

```
kubectl exec -it gpu-pod -- nvidia-smi -i 0 -m
```

Strategies for Cost Optimization

1. Match GPU Slices to Workload Needs

Lightweight inference tasks? Use smaller slices (e.g., 1G.5GB).
Training complex models? Go for larger slices (e.g., 3G.20GB).

```
resources:
  limits:
    nvidia.com/mig-1g.5gb: 1  # Smallest slice for lightweight tasks
```

2. Schedule Intelligently

Build custom scheduling logic to allocate slices efficiently. Use workload characteristics like compute intensity and memory requirements.

```
def schedule_gpu_workload(workload):
    if workload.compute_intensity < LOW_THRESHOLD:
        return allocate_small_gpu_slice()
    elif workload.compute_intensity < MEDIUM_THRESHOLD:
        return allocate_medium_gpu_slice()
    else:
        return allocate_large_gpu_slice()
```

3. Automate Scaling with Karpenter

Dynamically provision GPU nodes and reduce idle time to minimize infrastructure costs.

Known Challenges

- Performance Overhead: Some workloads may experience minor performance drops.
- Configuration Complexity: MIG setup requires precise planning and validation.
- Compatibility Issues: Not all workloads or applications are optimized for MIG.

References

- [NVIDIA MIG Documentation](https://docs.nvidia.com/datacenter/tesla/mig-user-guide/)
- [Kubernetes GPU Scheduling](https://kubernetes.io/docs/tasks/manage-gpus/scheduling-gpus/)