Improving GPU Utilization using Kubernetes Engine

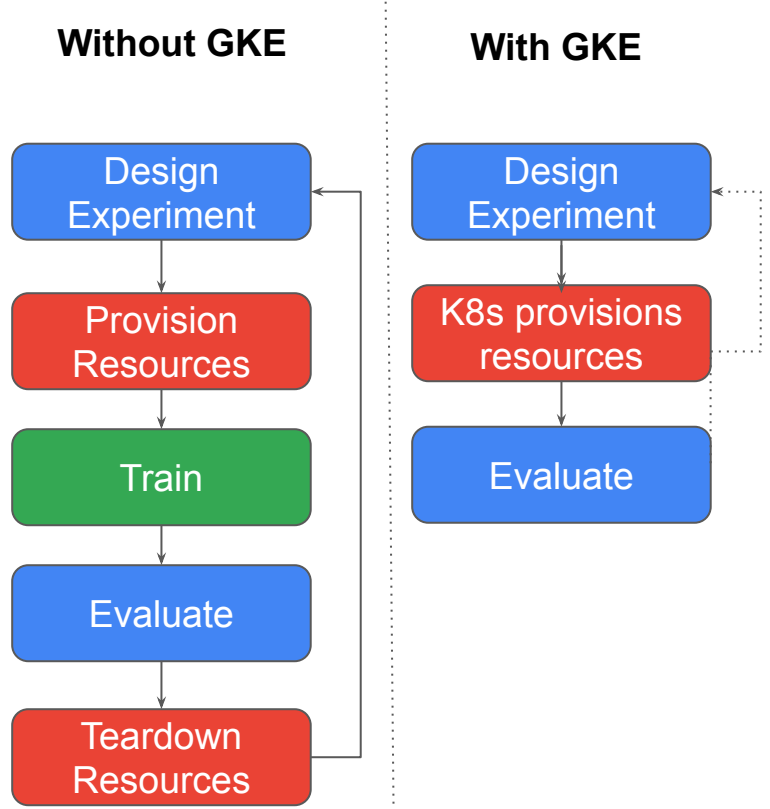
Pradeep Venkatachalam (pradvenkat@google.com)

Maulin Patel (maulinpatel@google.com)

Google Cloud

Kubernetes is Ideal for AI/ML and HPC

- **Portability**
  - Cloud native, open, standard APIs
    - Seamlessly port workloads between Laptop/Cloud
- **Scalability**
  - Kubernetes scales from a single workstation to thousands of nodes
    - Support for GPU and distributed computing frameworks
- **Productivity**
  - Frees up users from managing their own workstations, servers and VMs.
    - Lets you focus on model building and training



Google Cloud

Google Kubernetes Engine

Cost Optimized
Industry leading Auto-Provisioning, Auto-scaling

Secure By Design
Battle-tested and hardened configuration

Proven Reliability
Hardened by years of experience running mission critical workloads

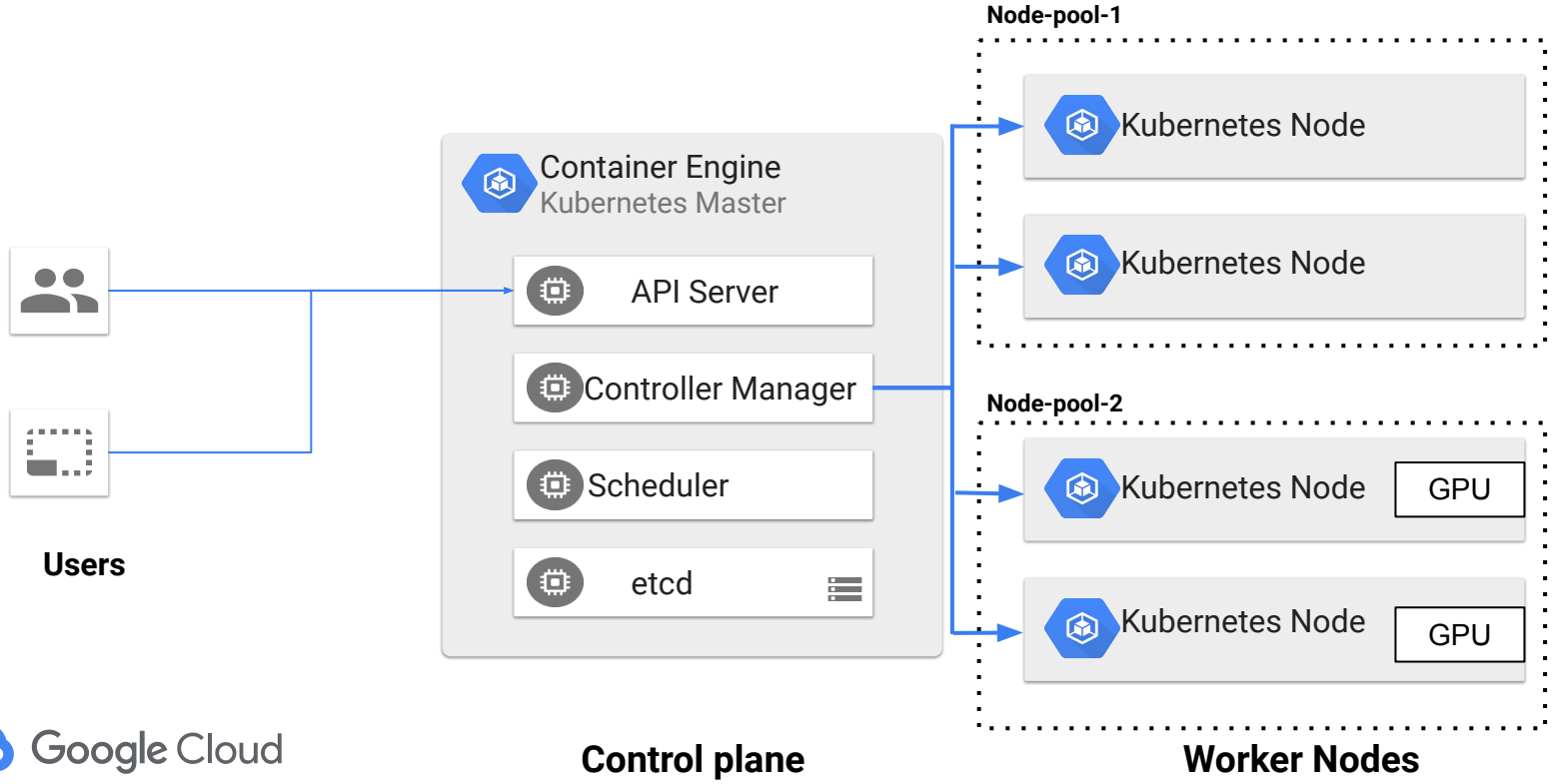
Container

Container Orchestration

P4 T4 K80 P100 V100 A100

Hardware Accelerators

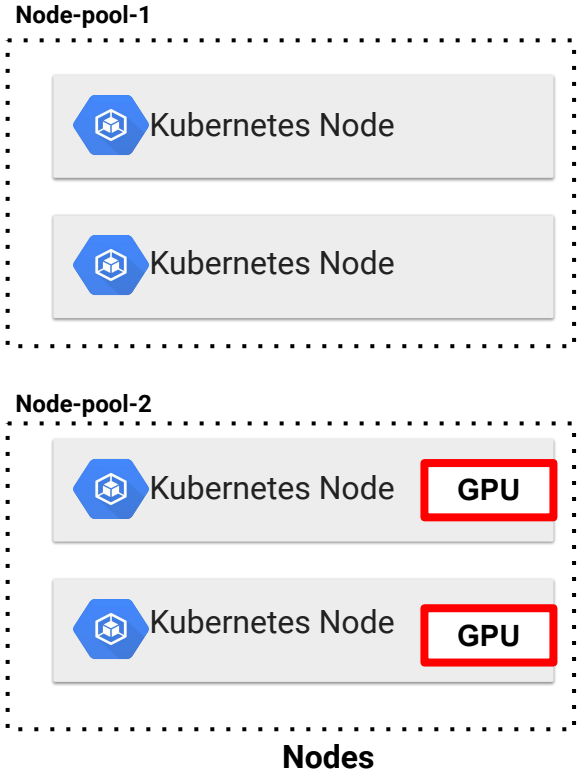
GKE Overview



Google Cloud

Node-pools

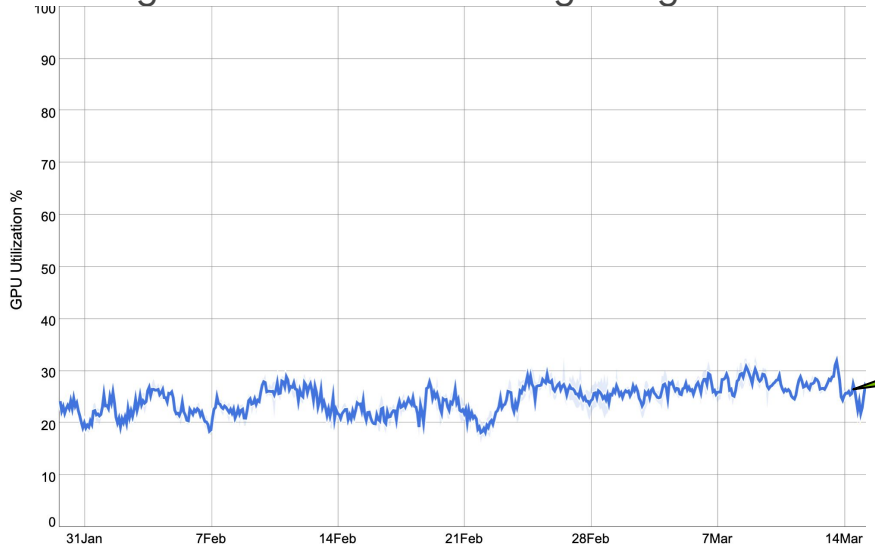
- Group of nodes within a cluster that share the same configuration
- Basic unit of autoscaling
- Nodes can have one or more GPUs attached



Google Cloud

Problem: GPU Utilization is Low

- GPU duty cycle is often very low => $ wasted
- GPU utilization is getting worse as GPUs are getting more powerful
  - A single workload may not saturate powerful GPUs
- Under utilization problem is acute for
  - Inference/serving/visualization/notebook/gaming workloads



Real World Use Cases

- Notebooks
- Chat Bots
- Vision Product Search
- Product Recommendations

Key Considerations

- Business critical apps
- Real-time serving
- Latency sensitivity
- Load is spiky
- GPU utilization very low
- Serving costs can quickly add up

Auto-provisioning and Auto-scaling features of Kubernetes are essential but not sufficient

- It takes minutes to add a new node to a cluster
- Bin packing is not supported for GPU workload

Google Cloud

Solution: Share a single GPU with multiple containers

Challenge: Kubernetes allows fractional requests for CPUs but NOT for GPUs

`spec.containers[].resources.requests.cpu` of `0.5` gets half CPU

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: my-gpu-pod
spec:
  containers:
  - name: my-gpu-container
    image: nvidia/cuda:10.0-runtime-ubuntu18.04
    command: ["/bin/bash"]
    resources:
      limits:
       nvidia.com/gpu: 1
```

one GPU is fully allocated to one container even if the container only needs a fraction of GPU for its workload => Wastage

Solution Options

- **Application level sharing**
  - CUDA Streams
  - Application frameworks like TF serving
- **GPU System s/w level sharing**
  - Timesharing
  - NVIDIA MPS
  - NVIDIA vGPU manager
- **H/w level sharing**
  - Multi Instance GPUs

Solution Options

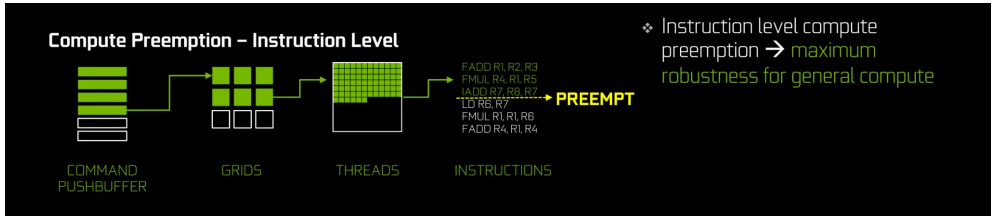
- **Application level sharing**
  - CUDA Streams
  - Application frameworks like TF serving
- **GPU System s/w level sharing**
  - Timesharing
  - NVIDIA MPS
  - NVIDIA vGPU manager
- **H/w level sharing**
  - Multi Instance GPUs

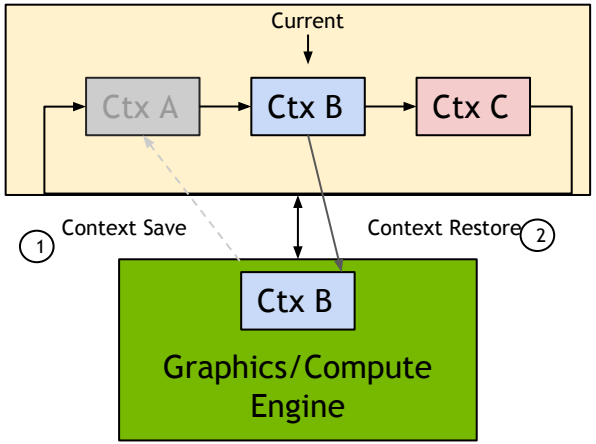
Focus of this talk

Temporal Multiplexing

GPU fractionalization on GKE[Preview]

- Allows multiple containers to run on a GPU, each container gets a timeslice.
- GPU time is allocated fairly to all containers.
- Compute Preemption Levels
  - At CUDA kernel boundaries (Kepler)
  - CILP (Pascal+) allows for context switching via timeslice for async compute
- GKE manages GPU configuration
- Simple Interface exposed to customers





GPU Timesharing User Experience

- Create a Cluster with GPU sharing enabled:

```
gcloud container clusters create <CLUSTER_NAME> -accelerator
type=nvidia-tesla-t4,count=1,max-time-shared-clients-per-gpu=10
```

- Or directly create a node-pool in an existing cluster:

```
gcloud container nodepools create <NODEPOOL_NAME> -cluster=<CLUSTER_NAME>
-accelerator type=nvidia-tesla-t4,count=1,max-time-shared-clients-per-gpu=10
```

- Directly call GKE cluster/node-pool API
- Feature will be available in GKE UI also.

User Experience (Node)

- After nodes are created and drivers are installed, inspect a node:

```
$ kubectl describe nodes

metadata:
  labels:
    cloud.google.com/gke-gpu-sharing-strategy: time-sharing
    cloud.google.com/gke-max-time-shared-clients-per-gpu: 10
spec:
 taints:
  - effect: NoSchedule
    key: nvidia.com/gpu
    value: present
  - effect: NoSchedule
    Key: gke-gpu-sharing-strategy
    value: time-sharing
status:
  capacity:
    nvidia.com/gpu:             10
  allocatable:
    nvidia.com/gpu:             10
```

User Experience (Node)

- After nodes are created and drivers are installed, inspect a node:

```
$ kubectl describe nodes

metadata:
  labels:
    cloud.google.com/gke-gpu-sharing-strategy: time-sharing
    cloud.google.com/gke-max-time-shared-clients-per-gpu: 10
spec:
 taints:
  - effect: NoSchedule
    key: nvidia.com/gpu
    value: present
  - effect: NoSchedule
    Key: gke-gpu-sharing-strategy
    value: time-sharing
status:
  capacity:
    nvidia.com/gpu:             10
  allocatable:
    nvidia.com/gpu:             10
```

10 shared GPUs available under the standard GPU resource name, each resource represents a GPU time-slice.

User Experience (Node)

- After nodes are created and drivers are installed, inspect a node:

```
$ kubectl describe nodes

metadata:
  labels:
    cloud.google.com/gke-gpu-sharing-strategy: time-sharing
    cloud.google.com/gke-max-time-shared-clients-per-gpu: 10
spec:
 taints:
  - effect: NoSchedule
    key: nvidia.com/gpu
    value: present
  - effect: NoSchedule
    Key: gke-gpu-sharing-strategy
    value: time-sharing
status:
  capacity:
    nvidia.com/gpu:             10
  allocatable:
    nvidia.com/gpu:             10
```

Nodes are labelled with GPU sharing configuration

User Experience (Node)

- After nodes are created and drivers are installed, inspect a node:

```
$ kubectl describe nodes

metadata:
  labels:
    cloud.google.com/gke-gpu-sharing-strategy: time-sharing
    cloud.google.com/gke-max-time-shared-clients-per-gpu: 10
spec:
 taints:
  - effect: NoSchedule
    key: nvidia.com/gpu
    value: present
  - effect: NoSchedule
    Key: gke-gpu-sharing-strategy
    value: time-sharing
status:
  capacity:
    nvidia.com/gpu:             10
  allocatable:
    nvidia.com/gpu:             10
```

Nodes are tainted to prevent whole GPU workloads from being scheduled on this node

User Experience (Workload)

- Use node-selector or node-affinity to schedule workloads onto GPU timeslices.

```
apiVersion: apps/v1
kind: Deployment
...
spec:
  replicas: 10
  template:
    spec:
      nodeSelector:
        cloud.google.com/gke-gpu-sharing-strategy: time-sharing
        cloud.google.com/gke-max-time-shared-clients-per-gpu: 10 # optional
      containers:
      - name: example
        resources:
          limits:
            nvidia.com/gpu:1
```

User Experience (Workload)

- Use node-selector or node-affinity to schedule workloads onto GPU timeslices.

```
apiVersion: apps/v1
kind: Deployment
...
spec:
  replicas: 10
  template:
    spec:
      nodeSelector:
        cloud.google.com/gke-gpu-sharing-strategy: time-sharing
        cloud.google.com/gke-max-time-shared-clients-per-gpu: 10 # optional
      containers:
      - name: example
        resources:
          limits:
            nvidia.com/gpu:1
```

- **All containers on nodes with these labels will share GPUs.**
- **If nodes do not exist, GKE can auto-provision or auto-scale to create nodes with correct GPU sharing configuration.**

User Experience (Workload)

- Use node-selector or node-affinity to schedule workloads onto GPU timeslices.

```
apiVersion: apps/v1
kind: Deployment
...
spec:
  replicas: 10
  template:
    spec:
      nodeSelector:
        cloud.google.com/gke-gpu-sharing-strategy: time-sharing
        cloud.google.com/gke-max-time-shared-clients-per-gpu: 10 # optional
      containers:
      - name: example
        resources:
          limits:
            nvidia.com/gpu:1
```

- **"1" is not a measure of GPU time allocated to the container. When multiple containers are competing for the GPU, they will all get a fair timeslice.**
- **What if >1 nvidia.com/gpu is requested? [see next slide]**

User Experience (Multiple GPUs)

- Why request **>1** nvidia.com/gpu timeslice?
  - Workload sizing - fit heterogeneous workloads onto a single GPU. Example: pack one large and two small containers on a single GPU.
  - However, all containers on GPU will get equal GPU time.
- Restrictions:
  - Cannot request >1 shared GPU on multi-GPU nodes, as k8s does not support device level allocation.

User Experience (GPU memory)

- All processes get separate address space.
- However, no memory limits are enforced, and it may result in OOMs.
- Responsibility of restricting memory usage is up to each workload.
- Some options to avoid OOMs:
    - CUDA unified memory - enables on demand paging between Host and GPU memory.
    - Frameworks like Tensorflow and PyTorch expose settings to control total GPU memory allocation.

Auto-scaling shared GPUs

- Typical Autoscaling workflow:
  - Expose GPU utilization metrics per container
  - Configure Horizontal Pod Autoscaler (HPA) to look at GPU utilization
  - HPA creates new pods when GPU utilization > threshold
  - Cluster Autoscaler (CA) will add new nodes to schedule these pods

Auto-scaling shared GPUs

- Typical Autoscaling workflow:
  - Expose GPU utilization metrics per container
  - Configure Horizontal Pod Autoscaler (HPA) to look at GPU utilization
  - HPA creates new pods when GPU utilization > threshold
  - Cluster Autoscaler (CA) will add new nodes to schedule these pods

Auto-scaling shared GPUs

- **Scale Up**: when unschedulable pods are detected, matching node-pools are scaled up
- **Scale Down**: Underutilized nodes are removed from the cluster
- **Auto Provisioning**: Create new node-pools based on demand from workload

Scale Up

- When pods are unschedulable, the most cost effective node-pool is scaled up
- Autoscaler extracts GPU sharing strategy from the node selector (either MIG or timesharing)
- Autoscaler looks at existing nodes in the node-pool to estimate resources per node, and determines # of nodes to add
- What if there are no nodes in a node-pool?
    - Autoscaler can estimate resource from node-pool configuration

Scale Down

- Iterates over all nodes to find under-utilized nodes
  sum(pod_requests) / node_allocatable < threshold
- Verifies that all workloads on node can be safely rescheduled
- Remove nodes that meet above criteria

Auto-provisioning

- Node auto-provisioner (NAP) can automatically create node-pools based on workload spec
- NAP extracts information from the workload spec, to create a node-pool configuration that can run the workload
- Enable auto-provisioning when creating a cluster with **--enable-autoprovisioning** flag

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example
spec:
  replicas: 7
  template:
    spec:
      nodeSelector:
       cloud.google.com/gke-gpu-sharing-strategy: time-sharing
      containers:
      - name: cuda-simple
        image: nvidia/cuda:11.0-base
        resources:
          limits:
            nvidia.com/gpu: 1
```

Google Cloud

Auto-provisioning

- Node auto-provisioner (NAP) can automatically create node-pools based on workload spec
- NAP extracts information from the workload spec, to create a node-pool configuration that can run the workload
- Enable auto-provisioning when creating a cluster with **--enable-autoprovisioning** flag

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example
spec:
  replicas: 7
  template:
    spec:
      nodeSelector:
        cloud.google.com/gke-gpu-sharing-strategy: time-sharing
      containers:
      - name: cuda-simple
        image: nvidia/cuda:11.0-base
        resources:
          limits:
            nvidia.com/gpu: 1
```

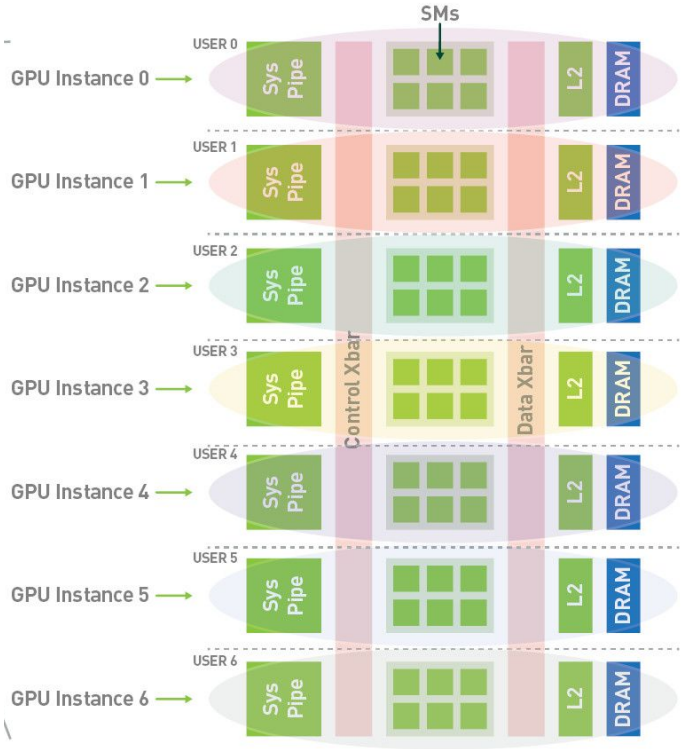
Google Cloud

In this example, NAP creates a node-pool where GPUs are timeshared

Spatial Multiplexing

Multi-Instance GPUs (MIG [GA]) on GKE

- Allows MIG-capable GPU to be partitioned "GPU Instances".
- Partitions are physically isolated with dedicated compute and memory units.
- Supports simultaneous workload execution with guaranteed QoS
- **A100 GPU supports up to seven instances.**
- Throughput increases linearly with additional instances



Google Cloud

MIG partitions

- A100 GPU has 7 compute units and 8 memory units (5GB each)
- These compute and memory units can be used to construct pre-defined GPU partitions
- Partitions are of the form: *<compute>g.<memory>gb*.

Partition size	GPU instances	Compute units per instance	Memory units per instance
1g.5gb	7	1	1
2g.10gb	3	2	2
3g.20gb	2	3	4
7g.40gb	1	7	8

MIG on GKE - Interface

- Each node-pool can be associated with one MIG partition size
- To create a node-pool with *1g.5gb* GPU instances

```
$ gcloud container node-pools create POOL_NAME \
  --cluster CLUSTER_NAME \
  --accelerator type=nvidia-tesla-a100,count=1,gpu-partition-size=1g.5gb \
  --machine-type a2-highgpu-1g \
  --num-nodes 1 \
  --enable-autoscaling --min-nodes 0 --max-nodes 5
```

Google Cloud

MIG on GKE - Interface

For ***gpu-partition-size=1g.5gb***, there will be 7 MIG partitions

```
$ kubectl describe nodes
...
Capacity:
  ...
  nvidia.com/gpu:                7
Allocatable:
  ...
  nvidia.com/gpu:                7
```

Google Cloud

Deploying workloads

- Nodes are labelled with the MIG partition size
- Workloads request GPUs as before
- Workloads can use node selector to run on desired MIG partitions

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example
spec:
  replicas: 7
  template:
    spec:
      nodeSelector:
        cloud.google.com/gke-gpu-partition-size: 1g.5gb
      containers:
      - name: cuda-simple
        image: nvidia/cuda:11.0-base
        resources:
          limits:
            nvidia.com/gpu: 1
```

Google Cloud

Deploying workloads

- Nodes are labelled with the MIG partition size
- Workloads request GPUs as before
- Workloads can use node selector to run on desired MIG partitions

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example
spec:
  replicas: 7
  template:
    spec:
      nodeSelector:
        cloud.google.com/gke-gpu-partition-size: 1g.5gb
      containers:
      - name: cuda-simple
        image: nvidia/cuda:11.0-base
        resources:
          limits:
            nvidia.com/gpu: 1
```

Google Cloud

Deploying workloads

- Nodes are labelled with the MIG partition size
- Workloads request GPUs as before
- Workloads can use node selector to run on desired MIG partitions

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: example
spec:
  replicas: 7
  template:
    spec:
      nodeSelector:
        cloud.google.com/gke-gpu-partition-size: 1g.5gb
      containers:
      - name: cuda-simple
        image: nvidia/cuda:11.0-base
        resources:
          limits:
            nvidia.com/gpu: 1
```

Google Cloud

CUDA Concurrency Mechanisms

	Streams	MPS	MIG	Timesharing
Partition Type	Single process	Logical	Physical	Logical
Max Partitions	Unlimited	48	7	Unlimited
SM Performance Isolation	No	Yes (by percentage, not partitioning)	Yes	No
Memory Protection	No	Yes	Yes	No
Memory Bandwidth QoS	No	No	Yes	No
Error Isolation	No	No	Yes	Yes
Cross-Partition Interop	Always	IPC	Limited IPC	Yes
Reconfigure	Dynamic	Process launch	When idle	Dynamic

CUDA Concurrency Mechanisms

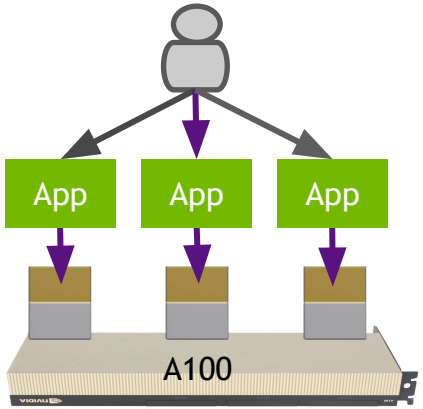
	Streams	MPS	MIG	Timesharing
Partition Type	Single process	Logical	Physical	Logical
Max Partitions	Unlimited	48	7	Unlimited
SM Performance Isolation	No	Yes (by percentage, not partitioning)	Yes	No
Memory Protection	No	Yes	Yes	No
Memory Bandwidth QoS	No	No	Yes	No
Error Isolation	No	No	Yes	Yes
Cross-Partition Interop	Always	IPC	Limited IPC	Yes
Reconfigure	Dynamic	Process launch	When idle	Dynamic

Timesharing vs MIG - Tradeoffs

- TS can share GPU with more than than 7 containers.
  - TS is great for sharing bursty workloads
  - MIG is great for sharing smaller sized workloads
- TS works on all GPUs supported by GCP.
  - T4, V100, A100s and A100 + MIG.
- MIG provides superior resource isolation
- In both cases, no security isolation between containers.

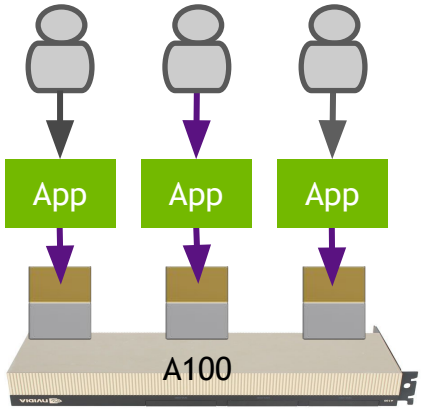
Use Cases Currently Supported

Single User → Multiple Apps



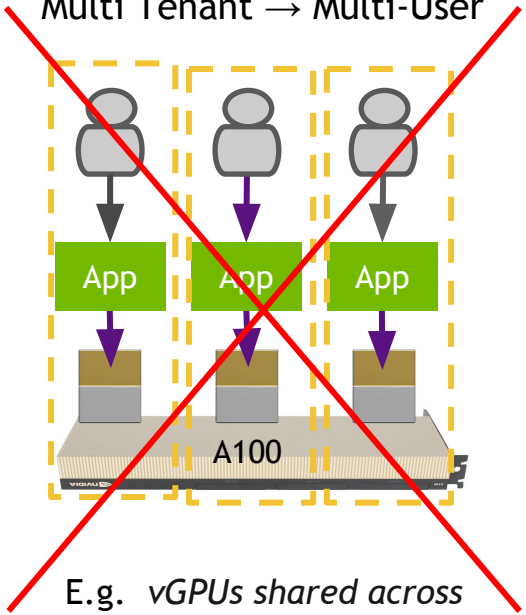
E.g. *Multiple inference jobs*

Single Tenant → Multi-User



E.g. *Jupyter notebooks for model exploration*

Multi Tenant → Multi-User



E.g. *vGPUs shared across customers/projects*

Google Cloud

Using GKE in support of IceCube

SDSC SAN DIEGO SUPERCOMPUTER CENTER

Presented by

Igor Sfiligoi

Lead Scientific Software Developer and Researcher at SDSC

Over the years, Igor Sfiligoi has been engaged in the computing community across a wide breadth of roles, participating among others as researcher, operator, developer, architect and manager. He has been working in both academia and industry, which helps him appreciate the variety of problems computers help to solve. He has been working with both systems that support tightly coupled and worldwide-scale pleasantly parallel codes, and realizes that each approach has its own advantages and challenges. In his current role at SDSC he is putting his extensive knowledge to use across many scientific domains, spanning computational biology, fusion research, high energy physics and astrophysics.

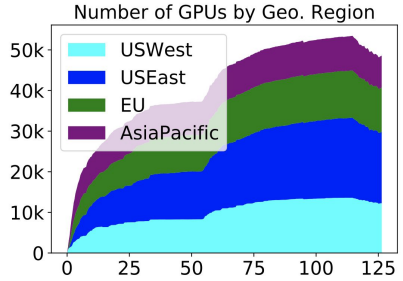
https://www.sdsc.edu/research/researcher_spotlight/sfiligoi_igor.html

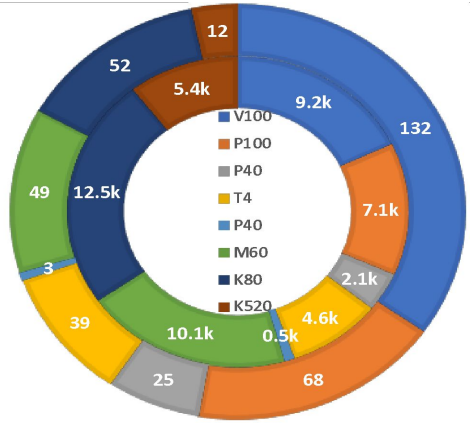
Google Cloud

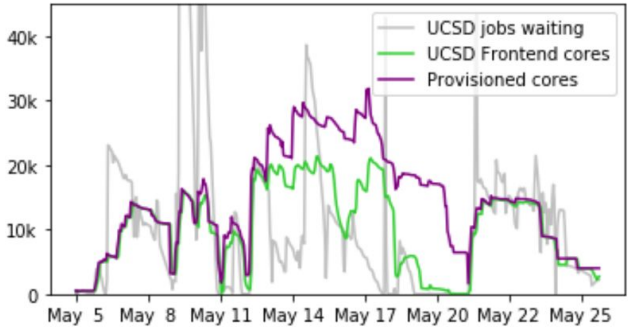
Using Cloud resources for science

Open Science Grid

- Several Cloud provisioning runs over the years
  - Integrating Cloud instances into existing HTCondor pools
- Ranged between
  - A short but big burst (few hours, ~400 fp32 PFLOPS)
  - A month long, multi-user run
- Direct Cloud instance provisioning great for short lived bursts
  - But long-lived and multi-user ones expose **Cloud API's limits**
  - Chiefly, **lack of proper queuing**







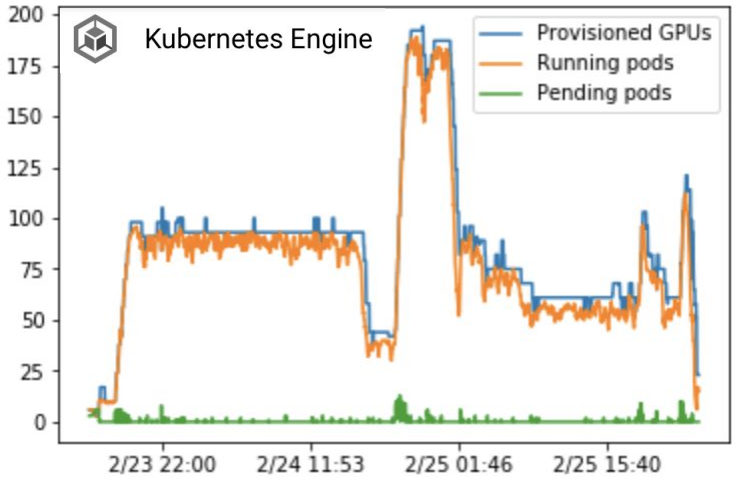
SDSC SAN DIEGO SUPERCOMPUTER CENTER

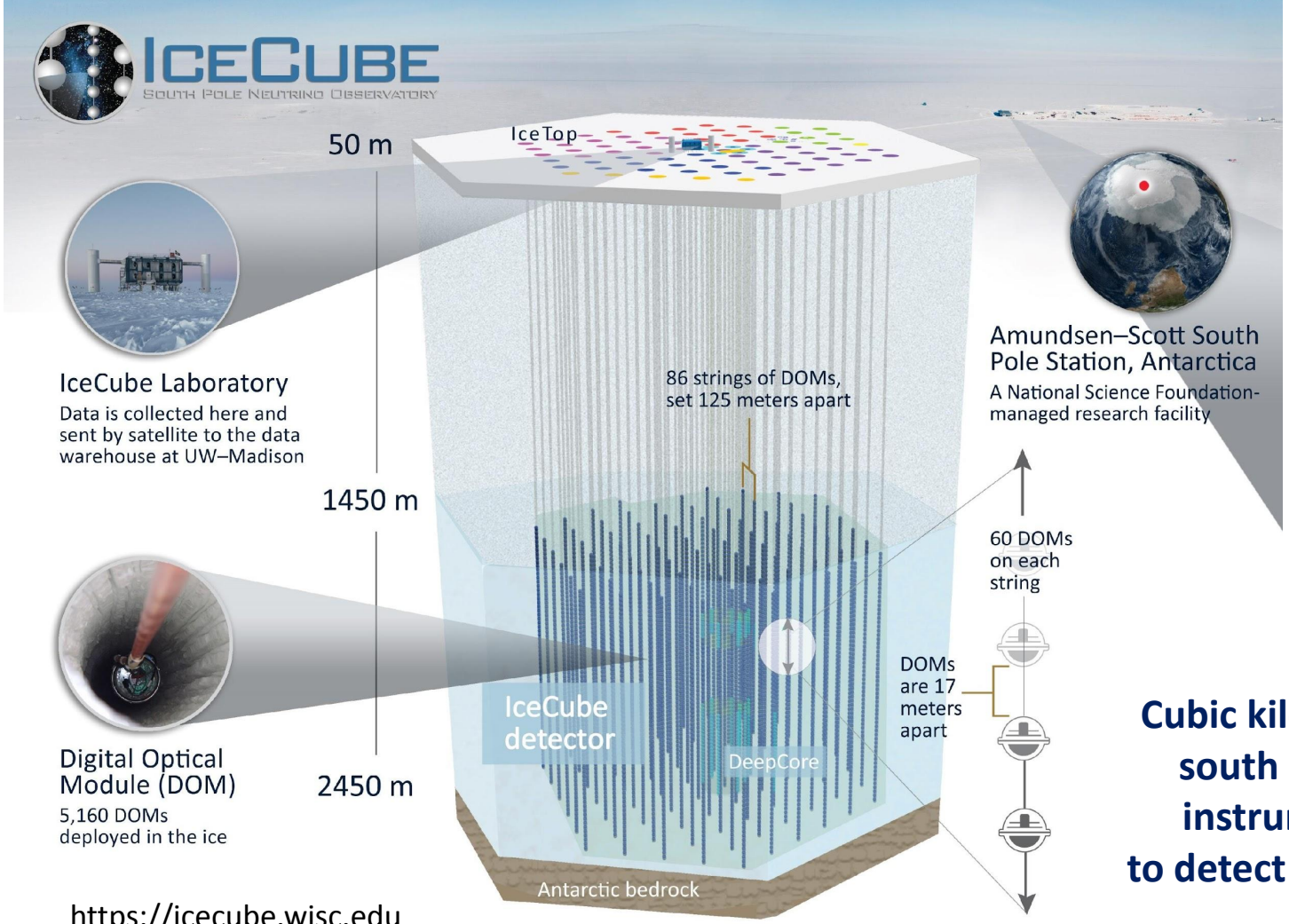
Kubernetes simplifies Cloud provisioning

- Already using Kubernetes on-prem
  - Have developed a Kubernetes Auto-scaler for HTCondor workloads
- Relying on Kubernetes scheduling
  - For resource sharing on PRP, as it serves multiple user communities
  - But happens to work great with GKE node auto-scaling, too
- With GKE, Cloud provisioning now fully automated

PRP PACIFIC RESEARCH PLATFORM

SDSC SAN DIEGO SUPERCOMPUTER CENTER



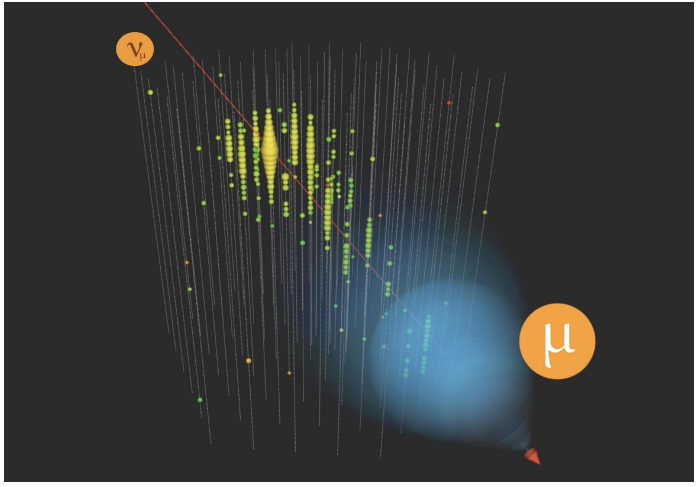


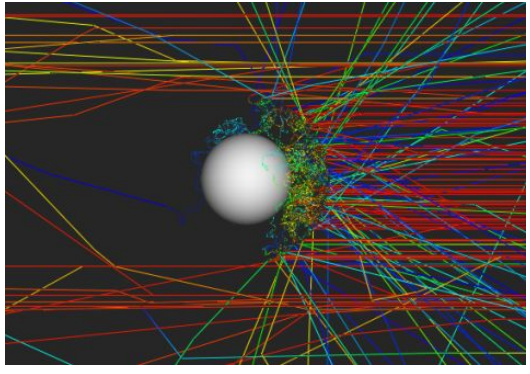
https://icecube.wisc.edu

Cubic kilometer of south pole ice instrumented to detect neutrinos.

IceCube fast photon propagation simulation

- Photon simulation needed for calibrating the detector
  - Which is embedded in natural ice
- **Fast, GPU-based** photon-propagation code used
- But starting point needs expensive CPU-based pre-processing
  - Which is **not parallelized**
  - So, hard to keep GPU busy





IceCube
South Pole Neutrino Observatory

GPU Sharing provides major benefit

- Using **GKE GPU Time Sharing** can map multiple pods to the same GPU, each with dedicated CPU cores
  - Got significantly more science from the same GPUs
    - **4x for A100**
    - **1.8x for V100**
    - **1.2x for T4**
  - But may not work out-of-the-box
    - Many applications try to use all of the **GPU memory**
- Using **GKE A100 MIG partitioning** adds strong isolation guarantees
  - Not essential for dedicated setups
  - But **easier on the users**
  - And **desirable for multi-tenancy**
  - About as effective as Time Sharing
    - **4x for A100**
    - But not available on other GPUs

IceCube South Pole Neutrino Observatory

SDSC San Diego Supercomputer Center

Conclusion

- GKE supports two GPU sharing solutions, with varying tradeoffs: MIG and Timesharing.
- Time-sharing solution is currently in private preview, and we are open for new customers to join the private preview program.
- Time-sharing works on A100 and other GPUs (V100, T4 etc), and has no limit on the number of containers sharing a GPU.
- A100 GPUs and MIG support is generally available on GCP & GKE.
- MIG offers the highest isolation levels, and enables A100 GPU to be shared with up to 7 containers.