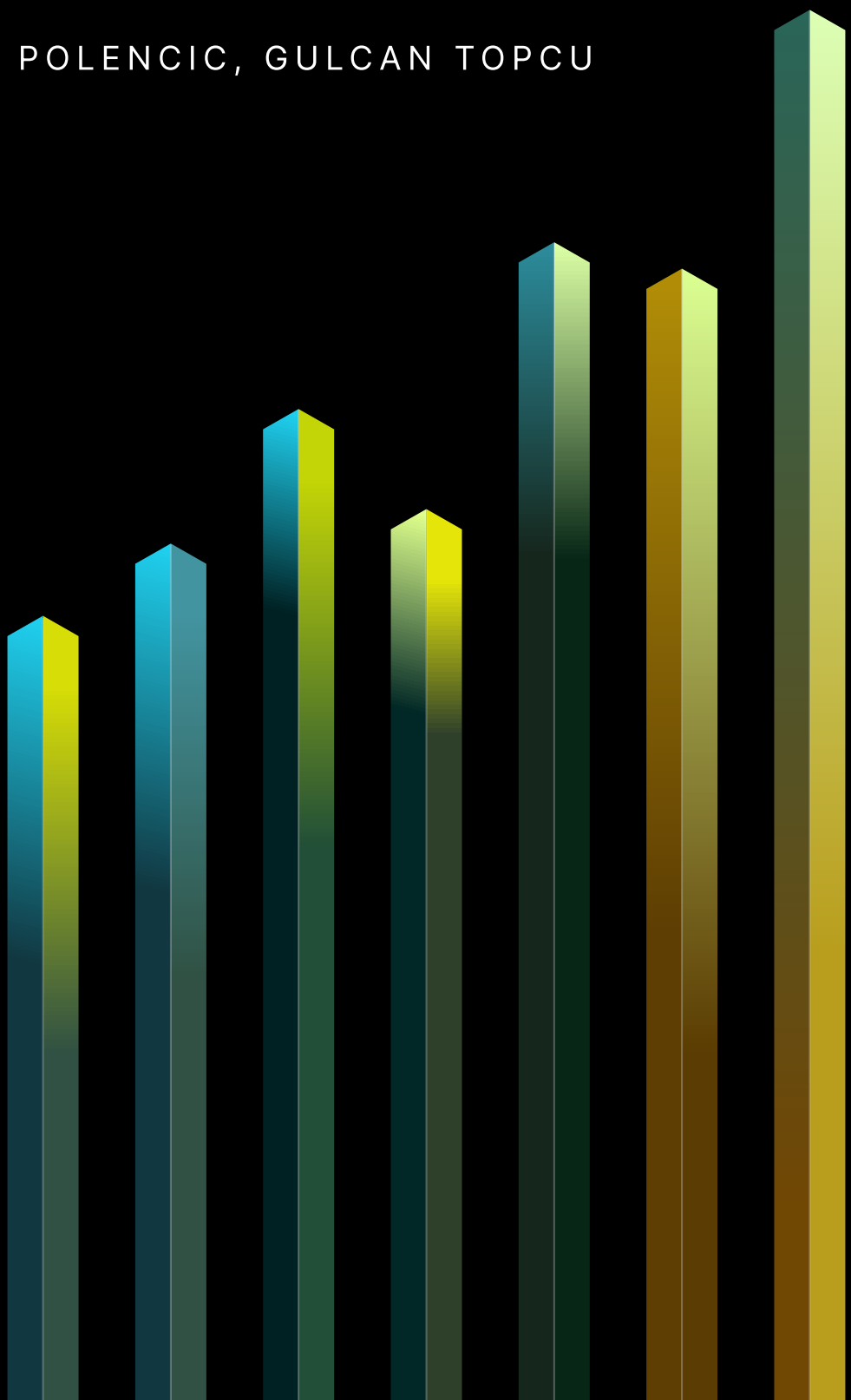
FOREWORD BY ANDREW HILLIER

RIGHT-SIZING GPUS IN KUBERNETES

A PLATFORM ENGINEER'S GUIDE TO AI INFRASTRUCTURE EFFICIENCY

DANIELE POLENCIC, GULCAN TOPCU



learnKUBE

Kubex

Foreword

AI infrastructure is entering a new phase.

For a while, training dominated the conversation. You ran big jobs, pushed GPUs hard, finished, and moved on. That model is relatively easy to reason about. You size for throughput, accept that jobs are batch-like, and utilization tends to be obvious.

Inference changes everything.

Inference workloads are persistent, bursty, and unpredictable. They sit in the critical path of products and services. Latency matters. Demand fluctuates. And small inefficiencies compound quickly when workloads never really turn off.

This is where GPUs start behaving very differently from the way most infrastructure teams are used to thinking.

With CPUs, the operating system constantly schedules work, fills gaps, and smooths contention. You can oversubscribe, throttle, and adapt. GPUs are far more rigid. There is no equivalent scheduler inside the device. A GPU runs exactly what it is given, and once capacity is reserved, it is effectively unavailable to anything else, whether it is being used efficiently or not.

Inference exposes this rigidity. To meet peak latency requirements, GPUs are sized for the worst case. But worst case demand is not constant. Most of the time, utilization rises and falls, leaving expensive hardware underused between bursts. Unlike CPUs, it is hard to safely reclaim that slack.

Kubernetes adds another layer to the problem. A pod that requests a GPU gets exclusive access. Placement depends on CPU, memory, NUMA topology, and node shape as much as the GPU itself. The system is doing exactly what it was designed to do, but the outcome is often inefficient for inference-heavy workloads.

This is why mechanisms like MIG and time slicing exist. MIG provides isolation and predictability. Time slicing allows higher packing density but comes with trade-offs around memory safety and visibility. Neither approach is inherently right or wrong. What matters is understanding when each makes sense and what you give up when you choose it.

That is what this book is about.

It focuses on how inference workloads actually behave, what GPU metrics are meaningful in that context, and how to think about yield rather than raw utilization. It connects measurement to real architectural decisions around scheduling, instance selection, and sharing strategies.

The ecosystem is moving quickly. Better device APIs, in-place resizing, and more intelligent automation are emerging. We are even starting to see AI used to optimize AI infrastructure. But none of that replaces understanding the fundamentals.

If you want to run inference at scale, you need to know how efficiently your GPUs are being used, not just whether they are allocated. This book helps you build that understanding.

Andrew Hillier
CTO, Kubex

Right-Sizing GPUs in Kubernetes: A Platform Engineer's Guide to AI Infrastructure Efficiency

Table of contents

The GPU Yield Problem

Your GPU cluster looks perfect on the dashboard.

Finance sees 100% utilization, and everyone is happy because the numbers show you're using all the resources you're paying for.

But if you ask your ML team, you'll get a different story.

They can't get GPU access, training jobs are stuck in queues, and inference runs slower than expected.

The dashboard tells one story, but the real situation is different.

To see why this happens, let's look at the difference between allocation and yield.

Allocation vs Yield: What You Pay For vs What You Get

Yield is just the ratio of what you get out to what you put in.

For GPUs, yield is the useful work you get from the hardware compared to what you pay for.

In Kubernetes, allocation is just a reservation.

GPUs are listed as an extended resource (like `nvidia.com/gpu`) and scheduled as whole integer devices.

When a pod sets `limits: nvidia.com/gpu: 1`, Kubernetes reserves one GPU for that pod and will not place another GPU-requesting pod onto that same device.

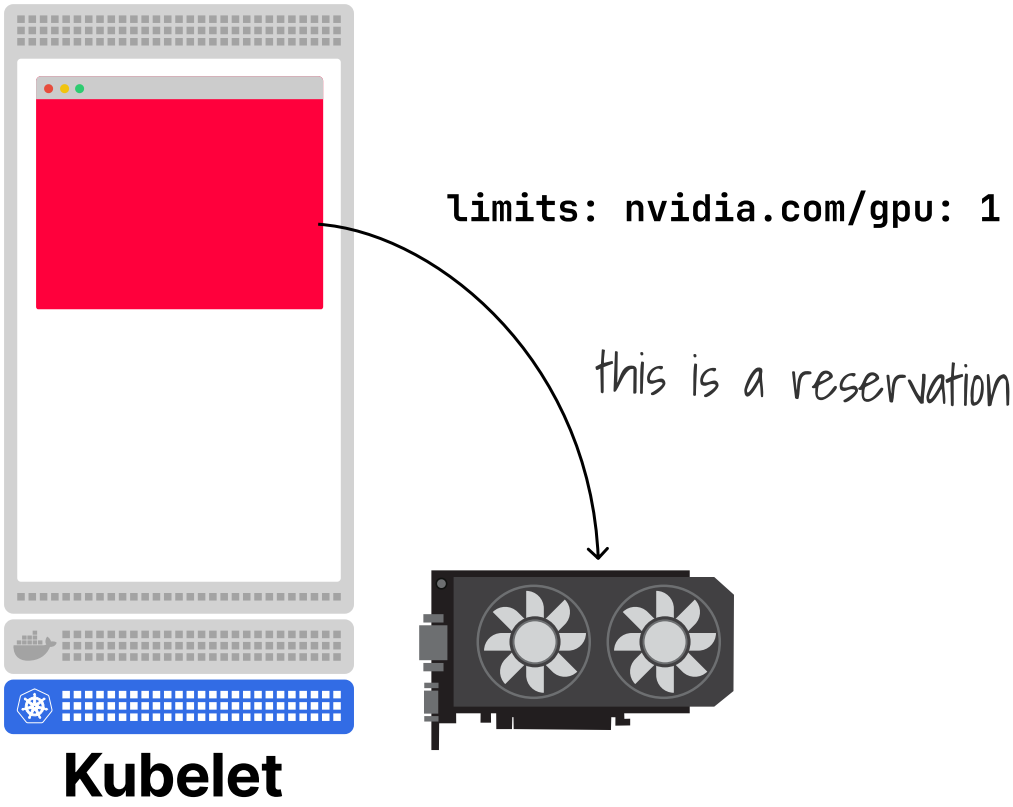


Fig. A pod on a Kubernetes node with the Kubelet sets limits nvidia.com/gpu: 1, reserving one GPU card — this is a reservation, not actual usage.

Yield tells you if the reserved hardware is actually doing useful work for your business.

Here's the tricky part: allocation and yield rarely match.

A GPU might show 100% allocation in Kubernetes, but the hardware could be running at only 2% utilization according to `nvidia-smi`.

Even if the hardware shows some activity, it might not be doing work that actually helps your business.

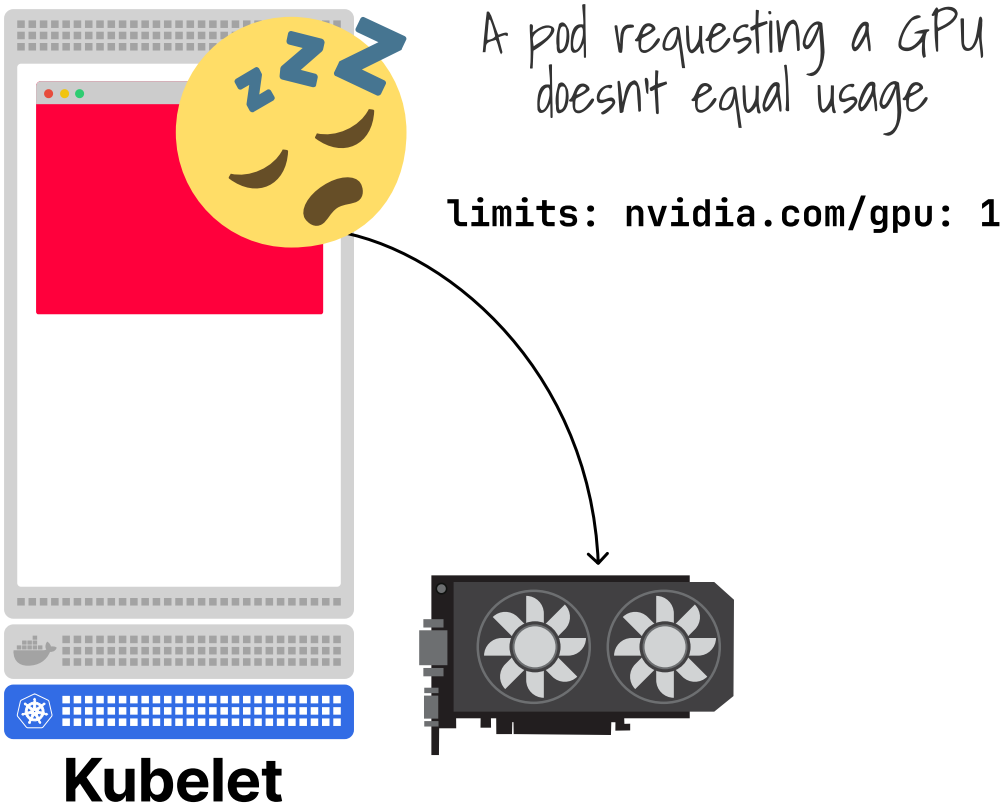


Fig. A sleeping pod on a Kubernetes node still reserves a GPU with limits nvidia.com/gpu: 1 — requesting a GPU doesn't equal usage.

So what does this look like in real life, and why does it happen so often in production?

When 100% Looks Efficient But Delivers 30%

Kubernetes reports 4 out of 4 GPUs allocated, and finance sees 100% usage on

the billing dashboard.

From those numbers, everything looks efficient.

But in reality, actual productive work often yields only 20-30%.

This creates a gap between what you pay for and what you receive.

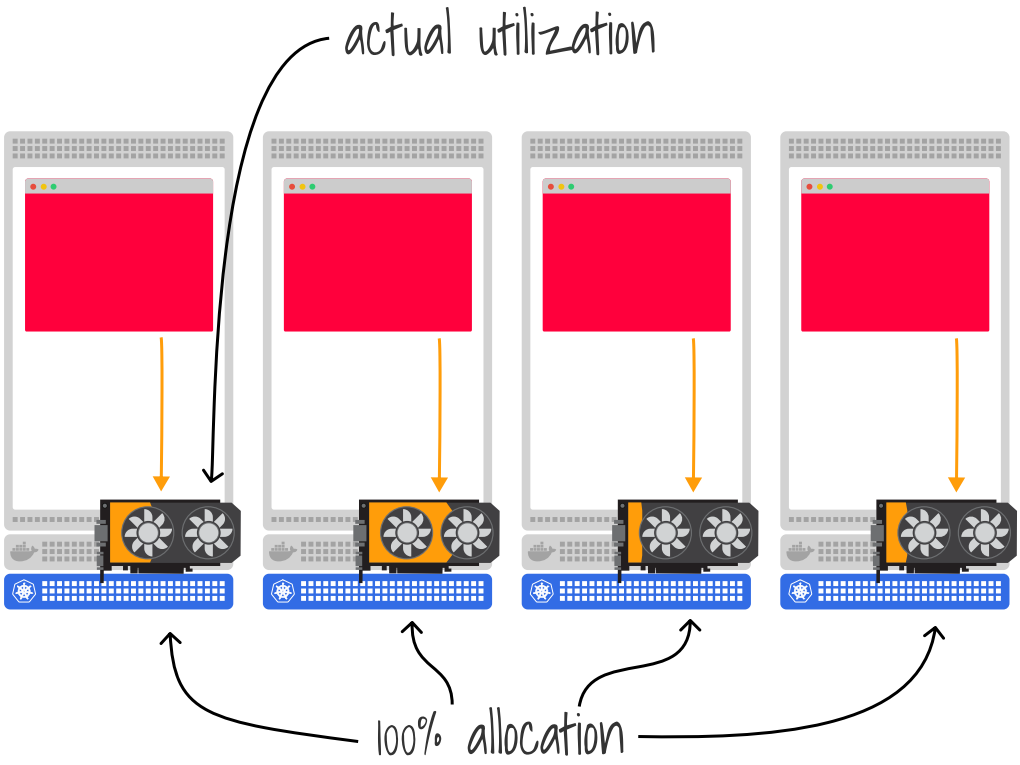


Fig. Four Kubernetes nodes each running a pod with an allocated GPU. All GPUs are at 100% allocation, but the pods vary in actual utilization — some GPUs are highlighted while others are dark, showing the gap between allocation and real usage.

Why does this gap exist?

The reasons for reserving a GPU, the conditions that make hardware busy, and the work that delivers business value all work separately.

Over time, they drift apart in predictable ways.

Let's have a look at a few examples.

A developer starts a Jupyter notebook, allocates a GPU to check if CUDA works, runs a few test cells, and then goes to lunch.

The GPU stays reserved for hours, doing nothing.

A training job finishes successfully, but doesn't end cleanly because of a bug in the exit handler.

The GPU stays allocated to a finished process that will never do more work.

Someone requests a GPU without realizing their workload doesn't need one.

Maybe they copied a template, misread the documentation, or just assumed 'ML = GPU.'

Their work runs entirely on the CPU, while the GPU remains reserved.

In every case, what you're charged for is different from what the hardware actually does, and what the hardware does is different from the business value you receive.

Let's measure this gap directly by deploying a workload that demonstrates the allocation illusion.

Measuring the Gap: A Three-Layer View

We'll start by deploying a simple inference service that exposes Prometheus metrics:

bash

```
$ kubectl create namespace demo
```

Let's create a ConfigMap with a simple Flask inference application:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: infer-app
  namespace: demo
data:
  app.py: |
    from flask import Flask
    import time

    app = Flask(__name__)
    counter = 0

    @app.route('/infer')
    def infer():
        global counter
        time.sleep(0.018)
        counter += 1
        latency = 17.5 + (hash(str(counter)) % 3)
        return f'ok latency_ms={latency:.2f}'

    @app.route('/metrics')
    def metrics():
        return f'infer_requests_total {counter}'

    if __name__ == '__main__':
        app.run(host='0.0.0.0', port=80)
EOF
```

Now deploy it with a GPU limit of 1 and expose it as a Service:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: infer
  namespace: demo
spec:
```

```
  replicas: 1
  selector:
    matchLabels:
      app: infer
  template:
    metadata:
      labels:
        app: infer
    spec:
      containers:
      - name: infer
        image: python:3.11-slim
        command: ["sh", "-c"]
        args:
          - |
            pip install -q flask && python /app/app.py
        ports:
        - containerPort: 80
        volumeMounts:
        - name: app
          mountPath: /app
        resources:
          limits:
            nvidia.com/gpu: 1
      volumes:
      - name: app
        configMap:
          name: infer-app
---
apiVersion: v1
kind: Service
metadata:
  name: infer
  namespace: demo
spec:
  selector:
    app: infer
  ports:
  - port: 80
    targetPort: 80
EOF
```

This Flask app runs only on the CPU: it doesn't use the GPU at all, but it still requires one.

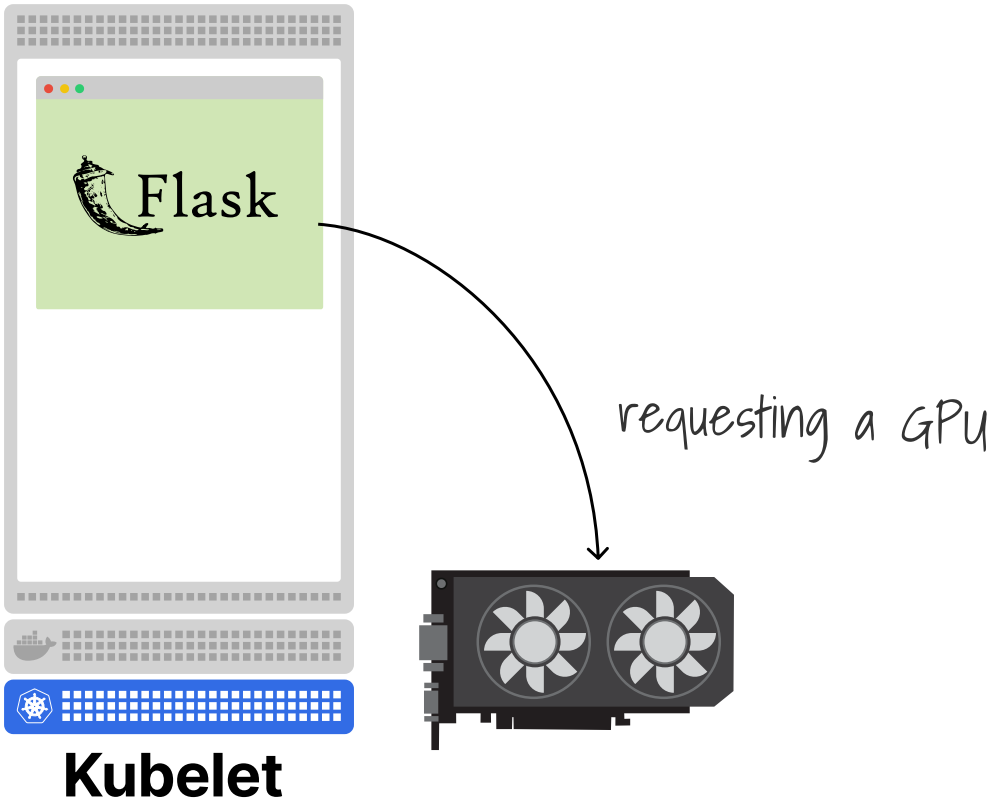


Fig. A Flask application running on a Kubernetes node with the Kubelet, requesting a GPU. The pod is deployed and ready to receive traffic.

It simulates 18 milliseconds of processing time and counts the number of requests.

Now we need to send continuous traffic to test its performance under load.

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: loadgen
  namespace: demo
spec:
```

```
  restartPolicy: Always
  containers:
  - name: loadgen
    image: curlimages/curl:8.10.1
    command:
    - sh
    - -lc
    - |
      while true; do
        out=$(curl -w ' client_latency_ms=%{time_total}' -sS -m 5 \
        http://infer.demo.svc.cluster.local/infer || echo "err")
        echo "$out"
        sleep 0.2
      done
EOF
pod/loadgen created
```

Once the pods are running, we can start measuring what's happening at different layers of the stack.

bash

```
$ kubectl get pods -n demo
NAME                     READY   STATUS    RESTARTS   AGE
infer-75f7bdd898-pv8cc   1/1     Running   0          45s
loadgen                  1/1     Running   0          12s
```

The load generator keeps sending requests to the inference service. You can see the real latency numbers by checking its logs.

bash

```
$ kubectl logs -n demo loadgen --tail=5
ok latency_ms=19.50 client_latency_ms=0.021326
ok latency_ms=17.50 client_latency_ms=0.020460
ok latency_ms=19.50 client_latency_ms=0.020558
ok latency_ms=19.50 client_latency_ms=0.025698
ok latency_ms=17.50 client_latency_ms=0.021174
```

Each request takes about 18 milliseconds to complete, and we're generating around five requests per second.

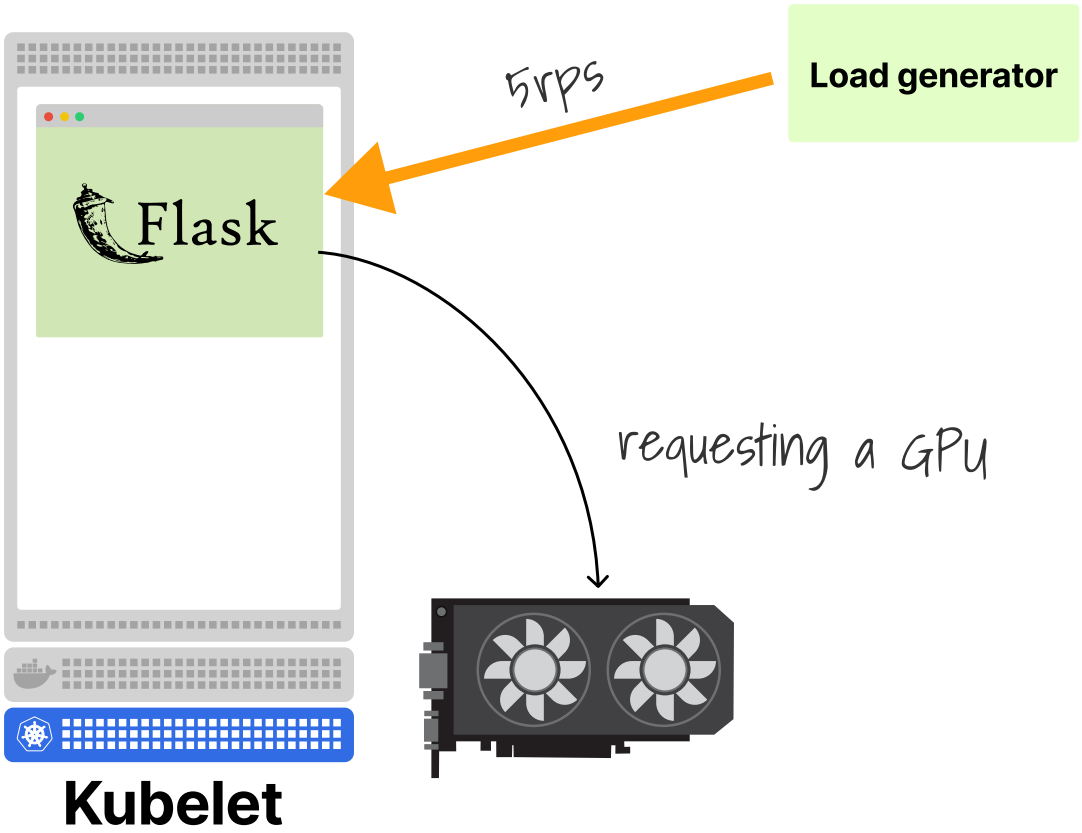


Fig. A load generator sending 5 requests per second to the Flask application on the Kubernetes node. The pod still requests a GPU from the Kubelet.

The server reports a latency of 17 to 19 milliseconds per request, while the client sees total round-trip times of 20 to 26 milliseconds.

The extra time comes from network overhead and HTTP processing.

Now let's check what Kubernetes thinks is happening on the GPU node.

bash

```
$ kubectl describe node minikube | sed -n '/Allocated
```

```
$  resources/,/^$/p'
  Allocated resources:
    (Total limits may be over 100 percent, i.e., overcommitted.)
    Resource           Requests    Limits
    --------           --------    ------
    cpu                850m (4%)   0 (0%)
    memory             370Mi (1%)  170Mi (0%)
    ephemeral-storage  0 (0%)      0 (0%)
    hugepages-1Gi      0 (0%)      0 (0%)
    hugepages-2Mi      0 (0%)      0 (0%)
    nvidia.com/gpu     1           1
```

From the scheduler's perspective, this node has 1 GPU allocated to our inference pod.

The scheduler marks it as unavailable for other workloads, even if it's not actually being used.

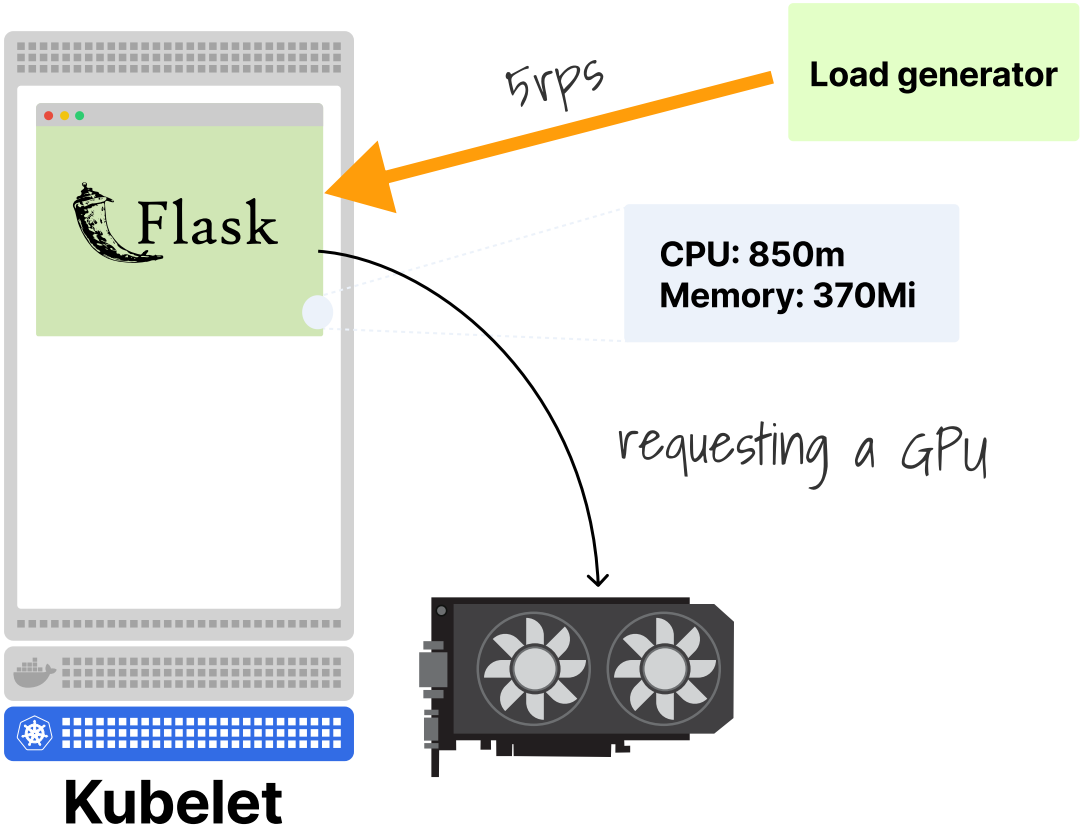


Fig. The Flask app under load showing Kubernetes resource metrics: CPU 850m and Memory 370Mi. The pod still holds a GPU reservation despite running on CPU only.

But `nvidia-smi` tells a different story.

bash

```
$ minikube ssh -- nvidia-smi
Fri Jan 16 08:51:18 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.163.01             Driver Version: 550.163.01     CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3050        Off |   00000000:01:00.0  On |                  N/A |
|  0%   42C    P2             N/A /  115W |     500MiB /   8192MiB |      2%      Default |
```

```
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
+---------------------------------------------------------------------------------------+
```

The GPU shows only 2% utilization and 500 MB of memory used out of 8 GB.

The Processes section is empty, indicating that no compute workloads are running.

The scheduler locked the GPU because the pod requested it, but the hardware is idle.

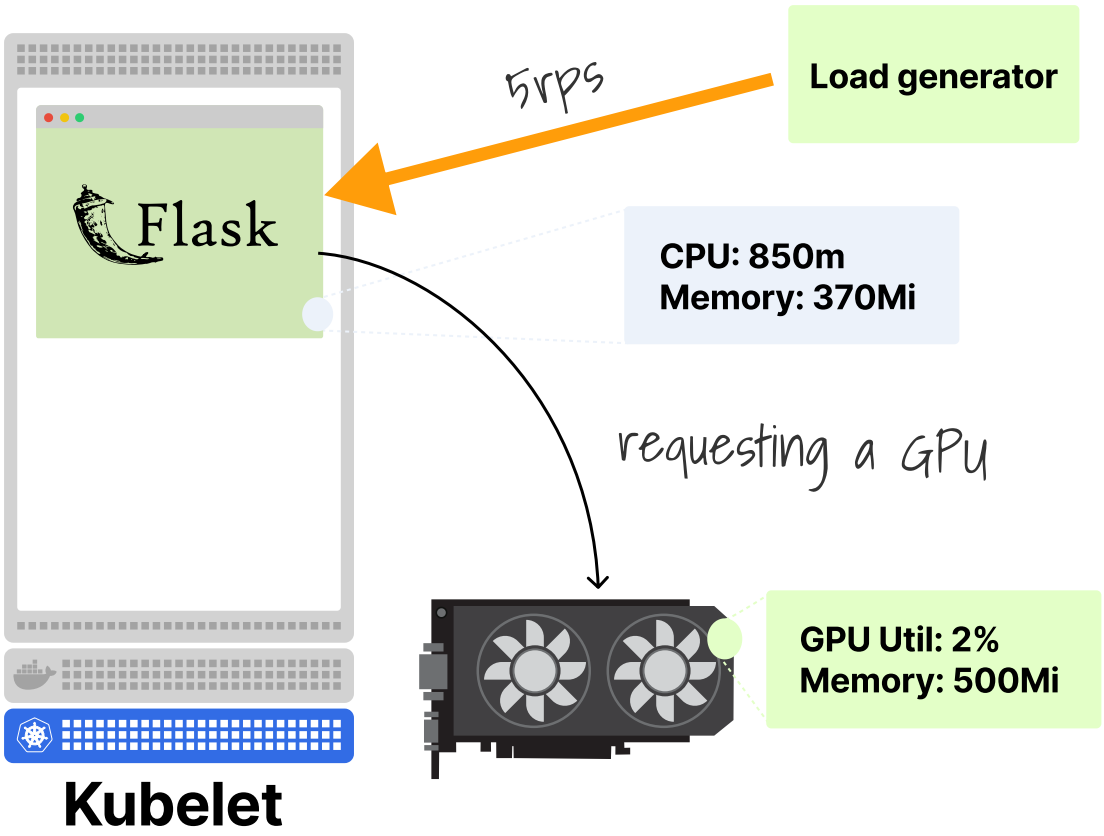


Fig. The Flask app under load with all three metrics visible: the load generator sends 5 rps, Kubernetes reports CPU 850m and Memory 370Mi, and the GPU shows only 2% utilization with 500Mi memory used.

Now let's look at the third perspective: the application metrics.

bash

```
$ kubectl exec -n demo loadgen -- curl -s http://infer.demo.svc.cluster.local/metrics
infer_requests_total 546
```

Let's check again after 30 seconds.

bash

```
$ sleep 30 && kubectl exec -n demo loadgen -- curl -s http://infer.demo.svc.cluster.local/metrics
infer_requests_total 746
```

The service processed 200 requests in 30 seconds, or 6.67 requests per second.

So, we have three very different views of the same GPU.

Kubernetes reports 100% allocation, `nvidia-smi` shows 2% utilization and 500MB used with no processes, and the application delivers 6.67 requests per second.

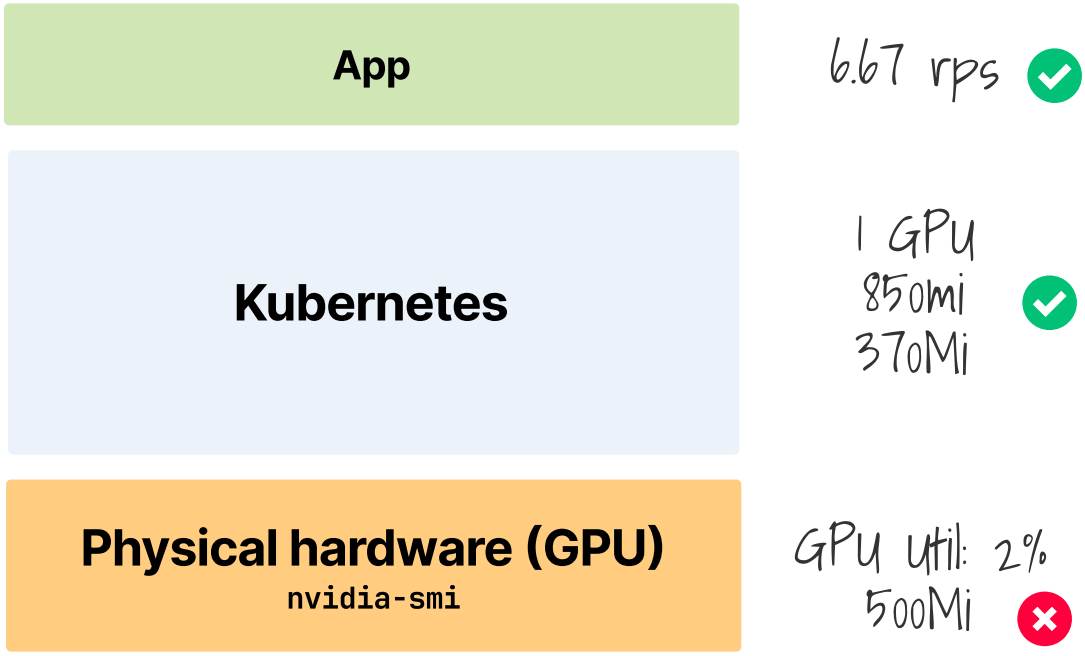


Fig. Three stacked layers showing different views of the same GPU: the App layer reports 6.67 rps (green check), the Kubernetes layer shows 1 GPU, 850m CPU, and 370Mi memory allocated (green check), but the Physical hardware layer via nvidia-smi shows only 2% GPU utilization and 500Mi used (red cross).

None of these numbers alone can tell you if the GPU is being used efficiently.

But here's the twist: the GPU isn't needed for this workload at all.

The pod claims one GPU, the hardware shows almost no utilization, and the application delivers 6.67 requests per second using only the CPU.

This isn't just low GPU yield.

It's pure allocation waste, where the GPU adds no value but is still locked away from other workloads that could use it.

But allocation waste isn't the only problem.

The metrics themselves can also be misleading.

Let's deploy a second pod that claims the GPU but does absolutely nothing, and watch what `nvidia-smi` reports.

bash

```
$ kubectl delete deploy infer -n demo
kubectl delete pod loadgen -n demo
deployment.apps "infer" deleted
pod "loadgen" deleted
```

Let's create a Pod that claims the GPU but does nothing:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: gpu-idle-claim
  namespace: demo
spec:
  restartPolicy: Always
  containers:
  - name: idle
    image: nvidia/cuda:12.4.1-base-ubuntu22.04
    command: ["bash","-lc","nvidia-smi; while true; do sleep 60; done"]
    resources:
      limits:
        nvidia.com/gpu: 1
EOF
pod/gpu-idle-claim created
```

From Kubernetes' perspective, nothing has changed.

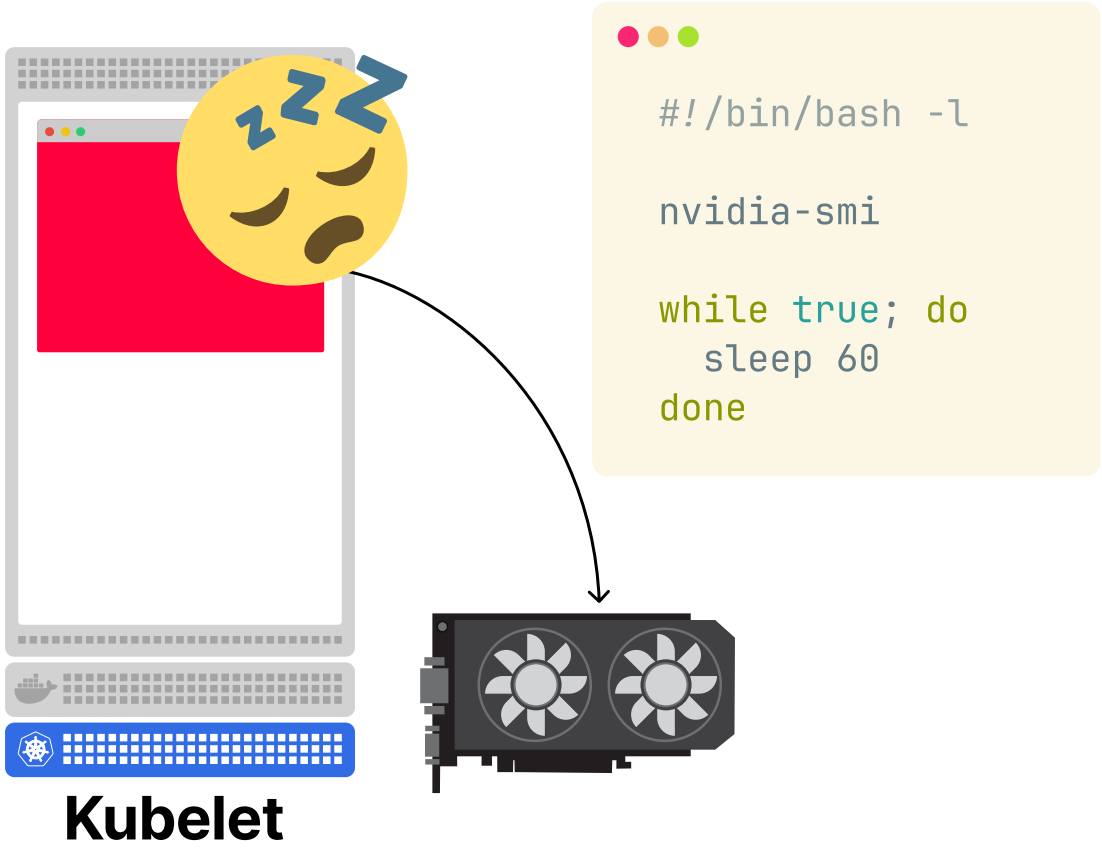


Fig. A sleeping pod on a Kubernetes node reserves a GPU while running a bash script that executes nvidia-smi once and then loops with sleep 60 forever. The pod does no useful GPU work

There's still one GPU allocated on this node.

bash

```
$ kubectl describe node minikube | sed -n '/Allocated resources/,/^$/p'
Allocated resources:
  Resource           Requests    Limits
  --------           --------    ------
  cpu                750m (3%)   0 (0%)
  memory             170Mi (0%)  170Mi (0%)
```

```
  ephemeral-storage  0 (0%)       0 (0%)
  hugepages-1Gi      0 (0%)       0 (0%)
  hugepages-2Mi      0 (0%)       0 (0%)
  nvidia.com/gpu     1            1
```

But `nvidia-smi` shows how GPU metrics can be misleading.

bash

```
$ minikube ssh -- nvidia-smi
Fri Jan 16 12:00:52 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.163.01             Driver Version: 550.163.01     CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3050        Off |   00000000:01:00.0  On |                  N/A |
|  0%   46C    P5             N/A /  115W |     605MiB /   8192MiB |     54%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+

+-----------------------------------------------------------------------------------------+
| Processes:                                                                              |
|  GPU   GI   CI        PID   Type   Process name                              GPU Memory |
|        ID   ID                                                               Usage      |
|=========================================================================================|
+-----------------------------------------------------------------------------------------+
```

The utilization is 54%, which looks productive.

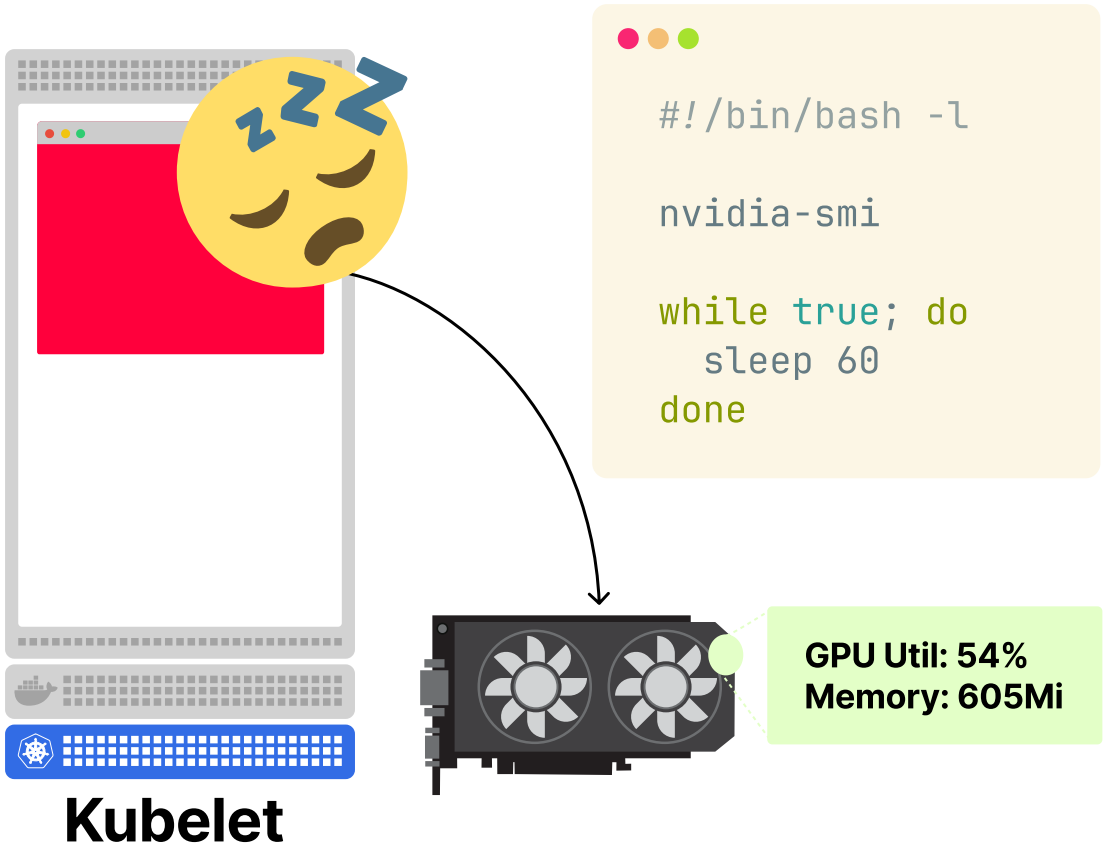


Fig. The same sleeping pod, but now nvidia-smi reports GPU Util: 54% and Memory: 605Mi — misleading metrics from a pod doing no useful GPU computation.

But `GPU-Util` is just a time-based 'busy' signal over a sampling window, not a direct measure of "useful ML work".

In NVML terms, it's reported as the percentage of time, during the last sampling period, that the GPU was active.

It's like checking if the stove is on to measure how productive your kitchen is.

The pilot light keeps the stove 'active' all the time, but that doesn't mean anyone is cooking.

On a display-attached GPU (for example, when `Disp.A` is On), that busy time can be driven by graphics/display activity and driver work, so you can observe non-trivial `GPU-Util` even when your application isn't running

CUDA kernels.

That's also why "Processes" output alone is not a reliable proof of "no activity" in every environment.

The practical rule is to treat `GPU-Util` as "something was busy recently."

Then, check yield using workload output metrics and, later, compute-focused counters.

The Three-Layer Visibility Problem

The Linux kernel sees everything that happens with CPUs and memory—which process is running, when it blocks for I/O, and how much memory it's allocated.

GPUs are different because the kernel can't see what they're doing.

A process opens `/dev/nvidia0` and gets permission to submit work.

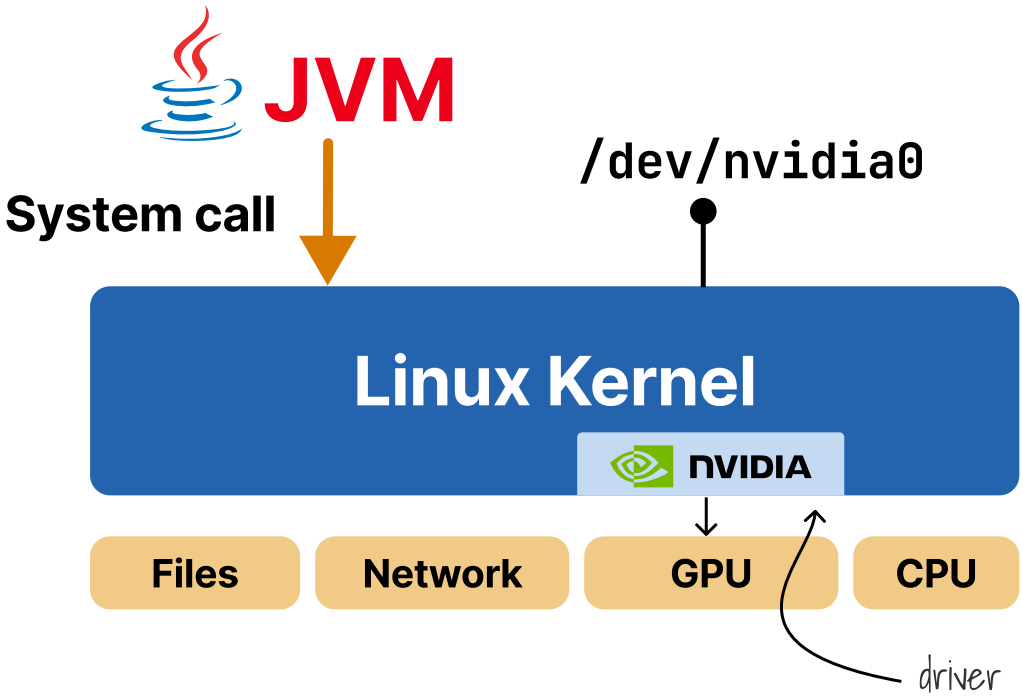


Fig. Architecture diagram showing how a JVM process makes system calls to the Linux Kernel, which manages Files, Network, GPU, and CPU. The GPU is accessed through the /dev/nvidia0 device node and the NVIDIA driver, which sits between the kernel and the GPU hardware.

After that, the kernel has no idea what's happening.

The GPU runs its own scheduler, manages its own memory, and executes kernels independently.

This creates three gaps in what you can see.

Memory works with three different numbers.

You request 8GB in your pod spec, CUDA pools 2GB from the GPU for performance, and your workload actually uses 500MB for tensors.

The scheduler decides based on the 8GB limit because that's the only number it can see.

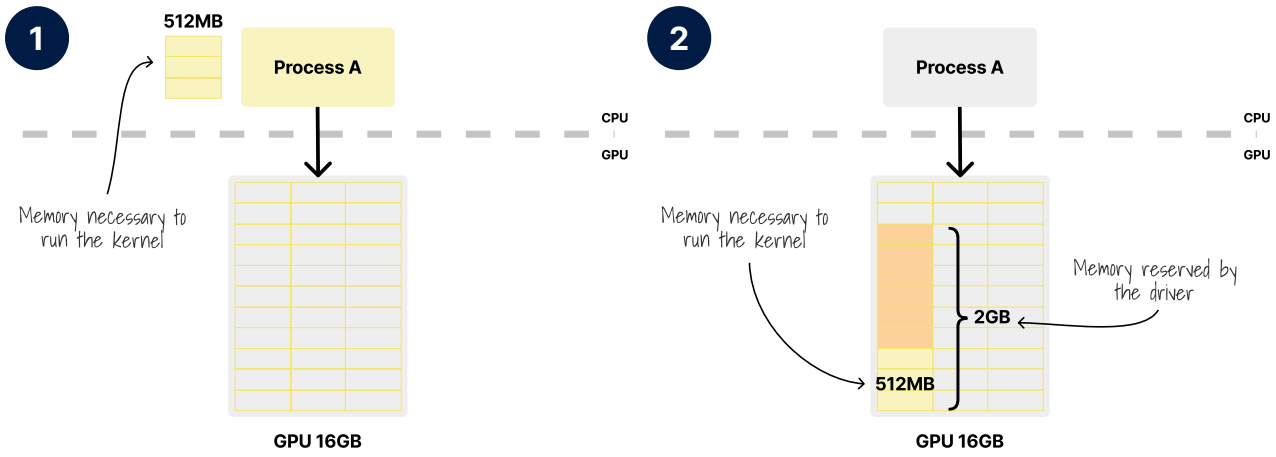


Fig. 1 Process A on the CPU sends work to the GPU, which allocates a CUDA context (Context A) with memory necessary to run the kernel.

Fig. 2 Process A's CUDA context on the GPU shows both the memory needed by the kernel and additional memory reserved by the driver — the driver allocates more than what's strictly needed.

The other layers are invisible to Kubernetes.

Execution is different from CPU time-slicing.

GPU kernels run to completion without preemption or fairness.

Once a kernel starts running, it owns the GPU until it finishes.

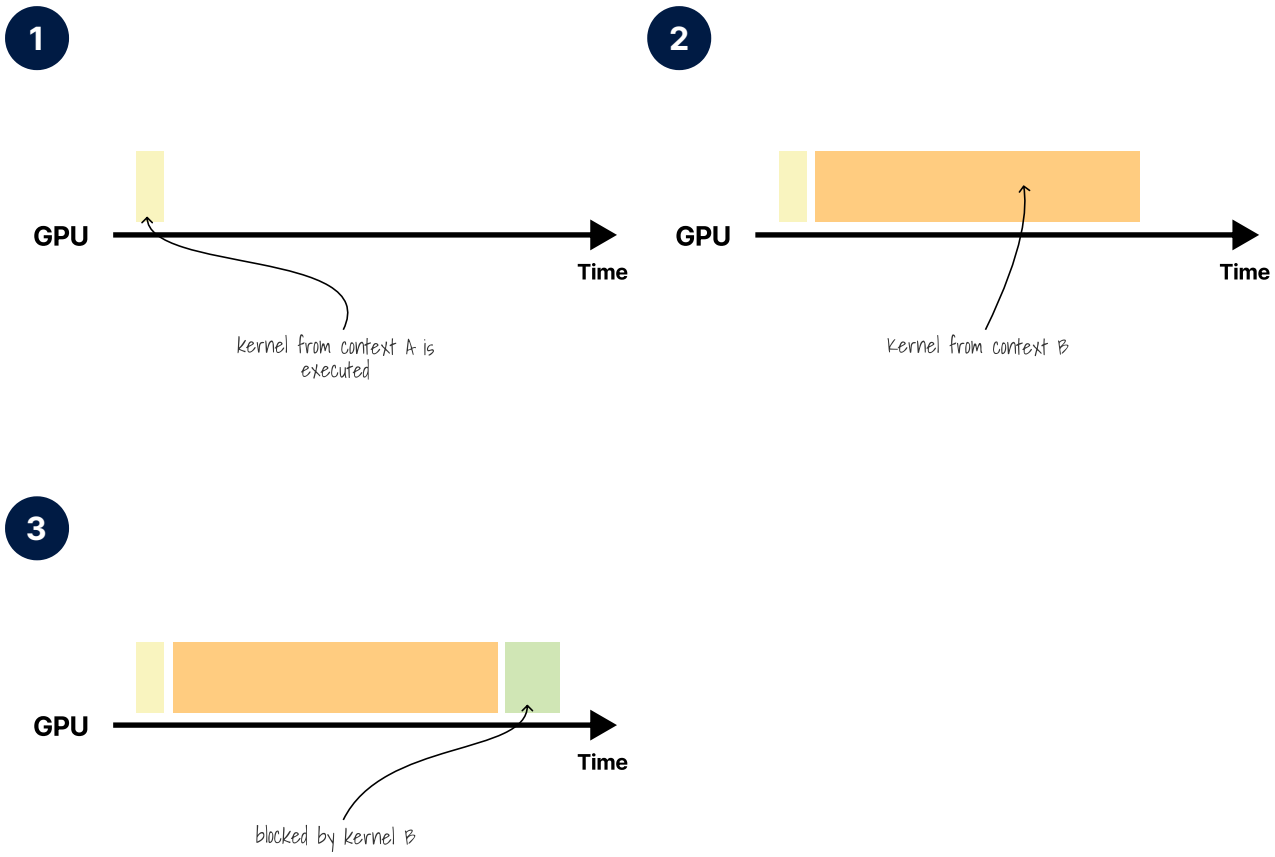


Fig. 1 A GPU timeline showing a single small kernel from Context A being executed. The rest of the timeline is idle.

Fig. 2 The GPU timeline now shows a small kernel from Context A (yellow) followed by a much larger kernel from Context B (orange) that dominates the GPU time.

Fig. 3 The GPU timeline shows Context A (yellow), then a long-running kernel from Context B (orange), and finally Context C (green) — Context C is blocked and must wait until kernel B completes.

If one workload submits a long-running kernel that takes 500 milliseconds, everything else has to wait.

You can't see inside the GPU as you can with CPUs.

Tools like `top`, `ps`, and cgroups don't work for GPUs.

The Linux kernel can't tell the difference between a workload that's using the GPU and one that's just holding a GPU allocation while running only on the CPU.

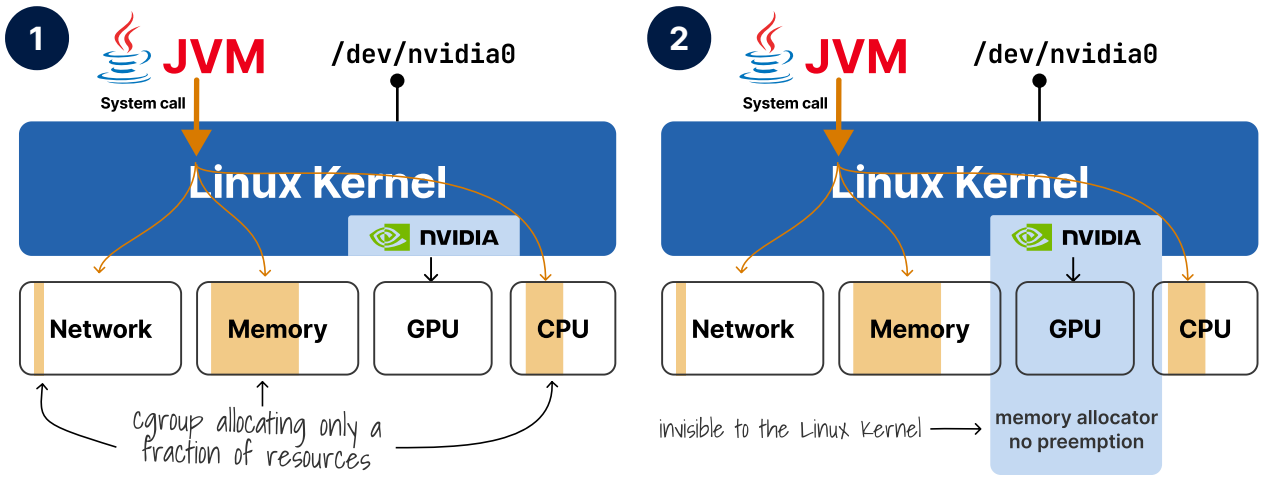


Fig. 1 A JVM process makes system calls to the Linux Kernel, which uses cgroups to allocate fractions of Network, Memory, GPU, and CPU resources. The NVIDIA driver sits between the kernel and the GPU hardware.

Fig. 2 The same architecture, but the GPU and NVIDIA driver are highlighted to show they are invisible to the Linux Kernel — the driver has its own memory allocator and no preemption, outside the kernel's control.

Stranded Resources: Where the Money Burns

The disconnect between these three layers creates waste.

GPU waste is even more expensive than CPU or memory waste because it multiplies costs.

GPUs don't come alone.

They're part of a bundle that includes a lot of CPU and memory resources. **When GPU allocation patterns strain those resources, the cost can become severe.**

Let's examine the stranded capacity on our demo node.

In production, you'd see this at a much larger scale, but the pattern is identical.

bash

```
$ kubectl describe node minikube | grep -A 10 "Capacity:"
Capacity:
  cpu:                8
  ephemeral-storage:  197834112Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             32880332Ki
  nvidia.com/gpu:     1
  pods:               110
Allocatable:
  cpu:                7820m
  memory:             30Gi
  nvidia.com/gpu:     1
```

This node has 8 CPUs, 32GB of RAM, and 1 GPU.

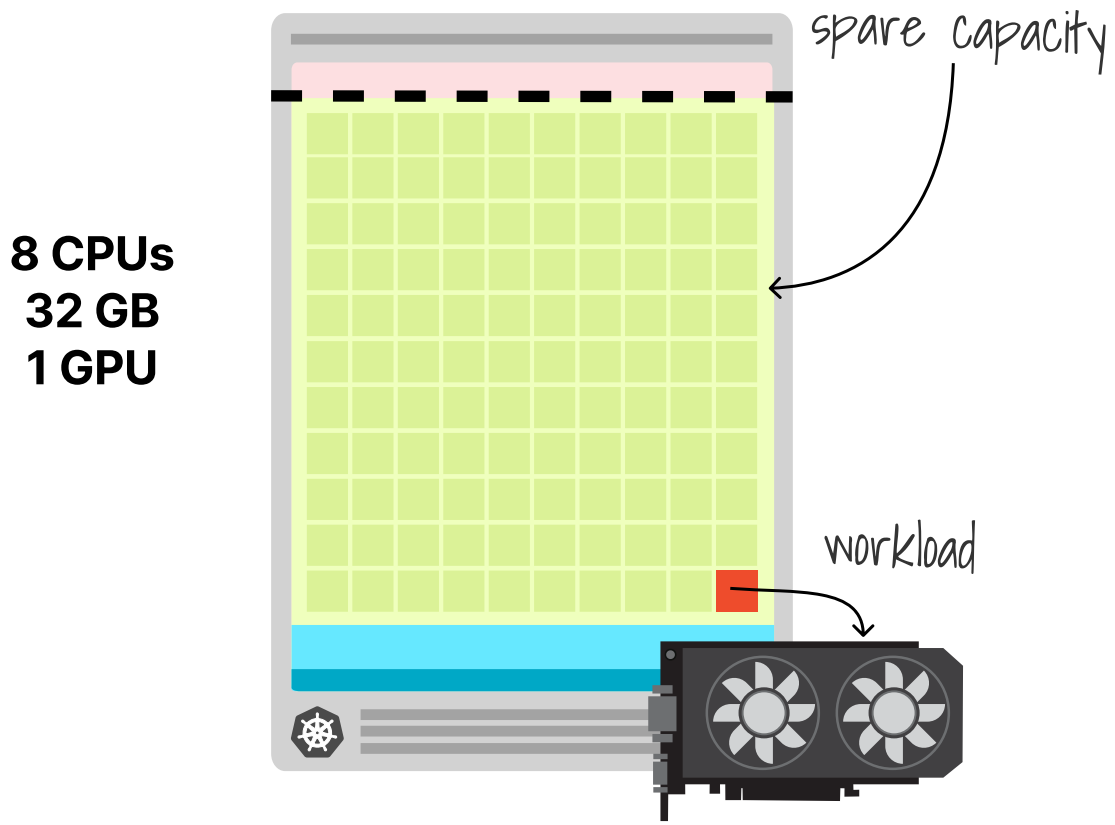


Fig. A Kubernetes node with 8 CPUs, 32 GB memory, and 1 GPU. An inference pod occupies a small corner of the available capacity, while most CPU and memory remain unused (spare capacity). The GPU is allocated to the workload.

That's a typical configuration for smaller GPU workloads where you want decent CPU and memory support for data preprocessing and batch assembly. Now let's check what our inference pod is actually consuming.

bash

```
$ kubectl top pod -n demo -l app=infer
NAME                      CPU(cores)   MEMORY(bytes)
infer-7b8f5d6c9-x4k2m     13m          72Mi
```

The pod uses 13 millicores and 72MB of memory, which is just 0.16% of the node's CPU and 0.2% of available RAM.

But because it has allocated the GPU, the scheduler considers this node "GPU-full" and won't accept more GPU workloads.

This leaves 7.8 CPU cores and 30GB of memory unused.

You could try to schedule CPU-only workloads to fill that gap, but most organizations deliberately separate GPU nodes from general-purpose compute.

They do this for cost tracking, maintenance windows, and to keep expensive GPU hardware focused on actual GPU work rather than random batch jobs.

This separation makes sense because GPU nodes are expensive, and GPU reservations can become the scheduling bottleneck, leaving the rest of the node unused.

The cascade effect gets much worse when you scale up to multi-GPU nodes.

Imagine this pattern across a node with 4 GPUs, 64 CPUs, and 256GB of RAM.

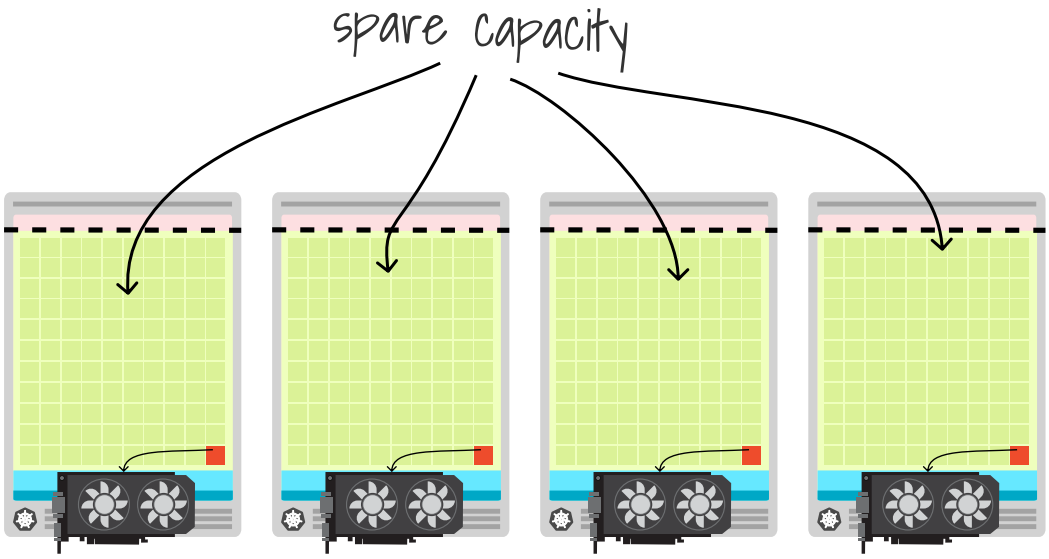


Fig. Four Kubernetes nodes, each with a GPU and large amounts of spare CPU and memory capacity. Each node runs a small workload that claims the GPU, leaving the vast majority of CPU and memory stranded and unused across all nodes.

That's a standard configuration for training or high-throughput inference.

If we had 4 GPUs and deployed 4 identical pods to our demo inference service, each pod would claim 1 GPU, leaving all 4 GPUs in use.

Actual consumption would show 52 millicores total (0.08% of 64 cores) and 288MB of memory (0.1% of 256GB), with all 4 GPUs allocated but each running at roughly 1% utilization.

At $8,000 per month for a 4-GPU node, you'd be paying for 64 CPUs while using 0.05 cores.

Paying for 256GB of RAM while using 288MB.

The rest would sit idle, not because the hardware is broken or the scheduler is inefficient, but because GPU allocation patterns created a bottleneck that leaves all the other resources unused.

You'd end up paying for large amounts of CPU and memory capacity you can't use because the GPUs are the limiting factor.

What This Means for Your Cluster

The demo showed the gap between allocation, utilization, and value.

Now you need to measure this in your production cluster across three layers.

First, track what Kubernetes has allocated by querying the API—this works from anywhere with kubectl access and shows which pods claimed GPUs.

bash

```
$ kubectl get pods -A -o json | \
  jq -r '.items[] | . as $pod | .spec.containers[] | select(.resources.limits."nvidia.com/gpu") |
  "\($pod.metadata.namespace)/\($pod.metadata.name) \(.resources.limits."nvidia.com/gpu")"'
```

Second, check what the GPU hardware is doing by running `nvidia-smi` on the GPU nodes:

bash

```
$ nvidia-smi --query-gpu=utilization.gpu --format=csv,noheader,nounits
```

Third, measure what value your applications deliver through application metrics.

If you're running Prometheus, scrape the `/metrics` endpoint; otherwise, use curl to retrieve the metrics directly from the pods.

```bash
$ curl -s http://service/metrics | grep requests_total
```

When these three views don't align, you have an allocation problem:

- High allocation with low utilization means you're paying for idle hardware.
- High allocation with high utilization but low value means the GPU is busy but not productive.

Instrument this properly with DCGM for continuous monitoring, along with the deeper metrics you need to measure real yield: GPU kernel execution time, memory bandwidth, and SM utilization.

Key Takeaways

Allocation is what you reserve and pay for, while yield is what you actually get.

A GPU can show 100% allocation while delivering only 20-30% yield, and that gap is where your money disappears.

GPUs create a three-layer visibility problem: what Kubernetes reserves, what CUDA allocates, and what applications consume are completely different numbers.

The scheduler sets limits, but actual usage is much lower.

The real cost isn't wasted GPU cycles; it's stranded resources.

A pod claiming one GPU while using 0.16% CPU and 0.2% RAM leaves the rest of the node's capacity unused.

On multi-GPU nodes, this adds up until you're burning thousands per month on idle hardware.

The Linux kernel sees everything about CPUs and memory, but is blind to GPUs.

It can't tell the difference between a workload that's GPU-bound and one just

holding an allocation while running on a CPU.

You can't improve yield until you measure it across three areas: what the scheduler allocated, what the hardware is doing, and what value your applications deliver.

Measuring What Actually Matters

A pod can claim a GPU and produce zero useful work.

Kubernetes might report 100% allocation and `nvidia-smi` could show high utilization, but neither tells you if the GPU is really doing useful work for your workload.

In this chapter, we'll measure GPU productivity by showing the difference between when the device is just active and when it's actually making progress on the job.

Most common GPU metrics, especially from `nvidia-smi`, only show activity over a short sampling window.

They don't directly measure throughput, efficiency, or job progress.

High utilization can happen if kernels launch often but use only a small part of the GPU, or if they spend a lot of time stalled waiting for memory, data transfers, or synchronization.

That's why dashboards can look healthy even when the GPU isn't delivering much useful work.

So, we focus on metrics that help answer specific questions.

- Is this workload limited by compute pipelines or by the memory system?
- Is the GPU waiting on host-to-device transfers or on peer-to-peer transfers?
- Are clocks being reduced by thermal or power constraints?
- Is memory behavior stable over time, or does it grow because of leaks, fragmentation, or allocator caching that never returns?

Each question needs different signals, so we'll build a measurement system that makes GPU productivity clear.

Before building that system, let's explain why the metrics you already use might not mean what you expect.

Why Standard Metrics Mislead

Start with the metric everyone checks first: GPU utilization in `nvidia-smi`.

In the output below, `GPU-Util` is the "67%" value.

nvidia-smi-output

```
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.163.01             Driver Version: 550.163.01     CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA A100-SXM4-40GB          On  |   00000000:00:04.0 Off |                    0 |
| N/A   32C    P0              51W / 400W |     575MiB /  40960MiB |     67%      Default |
|                                         |                        |             Disabled |
+-----------------------------------------+------------------------+----------------------+
```

`GPU-Util` doesn't mean the GPU is using 67% of its power or running at 67% of its peak performance.

It means that during the last sample period, the GPU had one or more kernels running for about 67% of that time.

The sample period is usually between 1 second and about 0.17 seconds.

In `nvidia-smi`, memory utilization is also a short-window activity metric: the percent of time during the last sample period when global device memory was being read or written.

This number shows how much time the GPU was active in that window, not how much work it actually did.

Since it only tells you if any kernel was running during that window, you can increase it by launching kernels constantly, even if each one is tiny, uses little of the GPU, or spends most of its time waiting on memory or synchronization.

It's like a camera that records 'motion detected' if anything moves in the last 60 seconds.

A fly passing by for 0.1 seconds and a person walking through for 59 seconds would look the same in the report.

`nvidia-smi` memory utilization has the same problem.

NVIDIA defines it as "the percent of time, during the last sample period, when global (device) memory was being read or written," again over that same short window.

The memory interface might be busy at times, but this doesn't tell you how much data moved, whether the accesses were efficient, or if the traffic improved overall throughput.

Memory capacity is another common source of confusion.

The "memory used" number in `nvidia-smi` corresponds to device memory

allocated by active GPU contexts, which in practice often includes CUDA context overhead and framework allocator pools (for example, PyTorch's caching allocator can keep memory reserved to speed up future allocations).

This means "reserved" memory can stay high even if the live tensors are small, so reading "memory used" as the true working set is often wrong.

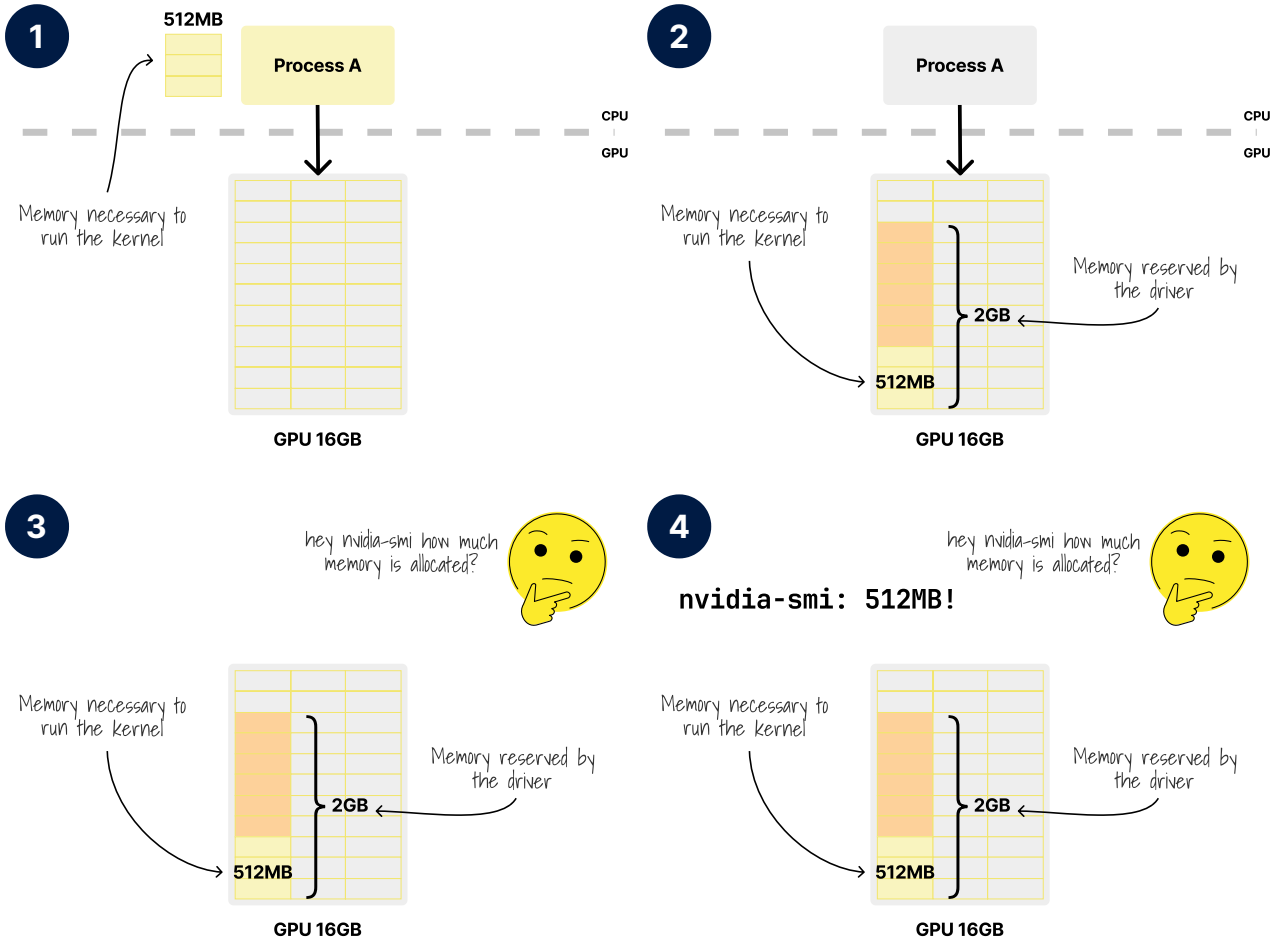


Fig. 1 Process A on the CPU sends work to the GPU, which allocates a CUDA context (Context A) with memory necessary to run the kernel.

Fig. 2 Process A's CUDA context on the GPU shows both the memory needed by the kernel and additional memory reserved by the driver — the driver allocates more than what's strictly needed.

Fig. 3 You can inspect the memory utilization via `nvidia-smi`

Fig. 4 But the answer isn't always what you expect.

Power, temperature, and throttling are still useful to monitor, but they don't measure productivity.

They only describe the hardware's condition.

A GPU can run hot or use power while doing inefficient work, stalling, or just moving data instead of computing.

These signals help you interpret what's happening, but they can't prove the workload is delivering value.

To tell the difference between "active" and "productive", you need counters that show what the GPU was actually doing, like SM activity, tensor pipeline activity, memory pressure, and transfer activity.

You can collect these metrics using DCGM if profiling or telemetry counters are available and enabled.

`dcgm-exporter` can then expose them for time-series analysis.

Now, let's deploy a workload that appears busy in `nvidia-smi` but actually does very little.

Demo 1: Busy Without Being Productive

To see why standard metrics mislead, we'll deploy a workload that can keep kernels launching continuously while doing very little useful computation.

The exact `GPU-Util` value you see may vary by system, but the main point remains: `GPU-Util` does not measure useful throughput.

bash

```
$ kubectl create namespace metrics-demo
namespace/metrics-demo created
```

Let's create a ConfigMap and Pod for a workload that launches many tiny

GPU kernels:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: fake-busy
  namespace: metrics-demo
data:
  fake_busy.py: |
    import torch
    import time

    device = torch.device('cuda:0')
    tiny_tensor = torch.ones(1, device=device)

    # Launch many tiny kernels with occasional sync
    start = time.time()
    iterations = 0
    while time.time() - start < 300:
        # Microsecond-scale kernel
        tiny_tensor = tiny_tensor + 1
        iterations += 1
        # Sync periodically to ensure work completes on the GPU and to stabilize what we observe
        if iterations % 10000 == 0:
            torch.cuda.synchronize()
---
apiVersion: v1
kind: Pod
metadata:
  name: fake-busy
  namespace: metrics-demo
spec:
  containers:
  - name: worker
    image: pytorch/pytorch:2.1.0-cuda12.1-cudnn8-runtime
    command: ["python", "/app/fake_busy.py"]
    volumeMounts:
    - name: app
      mountPath: /app
    resources:
      limits:
        nvidia.com/gpu: 1
  volumes:
  - name: app
    configMap:
      name: fake-busy
EOF
configmap/fake-busy created
pod/fake-busy created
```

This workload runs tiny kernels in a tight loop, with each kernel just adding 1 to a single-element tensor.

Depending on launch overhead and scheduling, `GPU-Util` may be low or moderate, but it can still look 'busy' even if the actual useful work per second is very small.

Wait for the pod to be ready before checking the `nvidia-smi` output:

```bash
$ kubectl wait --for=condition=ready pod/fake-busy -n metrics-demo --timeout=120s
pod/fake-busy condition met
```

Now, let's see what `nvidia-smi` reports for this workload that appears busy.

```bash
$ kubectl exec -n metrics-demo fake-busy -- nvidia-smi
Fri Jan 19 04:30:18 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.163.01             Driver Version: 550.163.01     CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3050        Off |   00000000:01:00.0  On |                  N/A |
| 31%   37C    P2             N/A /  115W |     575MiB /   8192MiB |     24%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

The GPU shows 24% `GPU-Util` *, which might look healthy at first.*

However, it only means that during the last short sample window, the GPU spent about 24% of the time running some kernel.

The 575 MiB reported by `nvidia-smi` is often dominated by CUDA context + framework runtime + the framework's GPU memory pool (caching allocator), so it can be much larger than the live tensor data.

That is why you can see hundreds of MiB used even when the live tensors are small.

So, frequent tiny kernels can keep `GPU-Util` above zero even when the amount of useful work per second is very low.

Now, let's deploy a sustained, compute-heavy workload for comparison.

Let's create a ConfigMap and Pod for a sustained, compute-heavy workload:

```bash
$ kubectl apply -f - <<'EOF'
apiVersion:
```

```
 v1
kind: ConfigMap
metadata:
  name: real-work
  namespace: metrics-demo
data:
  real_work.py: |
    import torch
    import time

    device = torch.device('cuda:0')
    # Large matrix multiplication
    size = 8192
    A = torch.randn(size, size, device=device)
    B = torch.randn(size, size, device=device)

    start = time.time()
    while time.time() - start < 300:
        C = torch.matmul(A, B)
        torch.cuda.synchronize()
---
apiVersion: v1
kind: Pod
metadata:
  name: real-work
  namespace: metrics-demo
spec:
  containers:
  - name: worker
    image: pytorch/pytorch:2.1.0-cuda12.1-cudnn8-runtime
    command: ["python", "/app/real_work.py"]
    volumeMounts:
    - name: app
      mountPath: /app
    resources:
      limits:
        nvidia.com/gpu: 1
  volumes:
  - name: app
    configMap:
      name: real-work
EOF
configmap/real-work created
pod/real-work created
```

This workload continuously runs large matrix multiplications using 8192 by 8192 matrices.

Wait for the pod to be ready before checking its GPU usage.

bash

```
$ kubectl wait --for=condition=ready pod/real-work -n metrics-demo --timeout=120s
pod/real-work condition met
```

Now, let's check what `nvidia-smi` shows for this real workload to see the difference.

bash

```
$ kubectl exec -n metrics-demo real-work -- nvidia-smi
Fri Jan 19 04:32:42 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.163.01             Driver Version: 550.163.01     CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3050        Off |   00000000:01:00.0  On |                  N/A |
| 35%   52C    P2             N/A /  115W |    1621MiB /   8192MiB |    100%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

Now, `nvidia-smi` shows 100% `GPU-Util` and 1621 MiB in use.

This means kernels were running almost the entire sampling window, and the process is using more GPU memory than in the fake-busy case.

The increase from 37°C to 52°C and the small fan speed rise fit this pattern, since sustained work usually makes the GPU hotter than running tiny, low-intensity kernels.

Even here, we still do not know what is slowing the job down.

To understand performance limits and measure yield, we need counters that separate compute activity, stalls, memory pressure, transfer throughput, and clock behavior.

Then we interpret these in the context of the workload.

How GPUs Execute Work

Before we look at the metrics that predict yield, we need to understand how GPUs actually run code, since the metrics measure specific parts of this execution model.

A GPU does not schedule threads one at a time.

Instead, threads are grouped into warps, which are bundles of 32 threads that execute together.

All 32 threads in a warp run the same instruction at the same time.

Warps are assigned to Streaming Multiprocessors (SMs), which are the main compute units on the GPU.

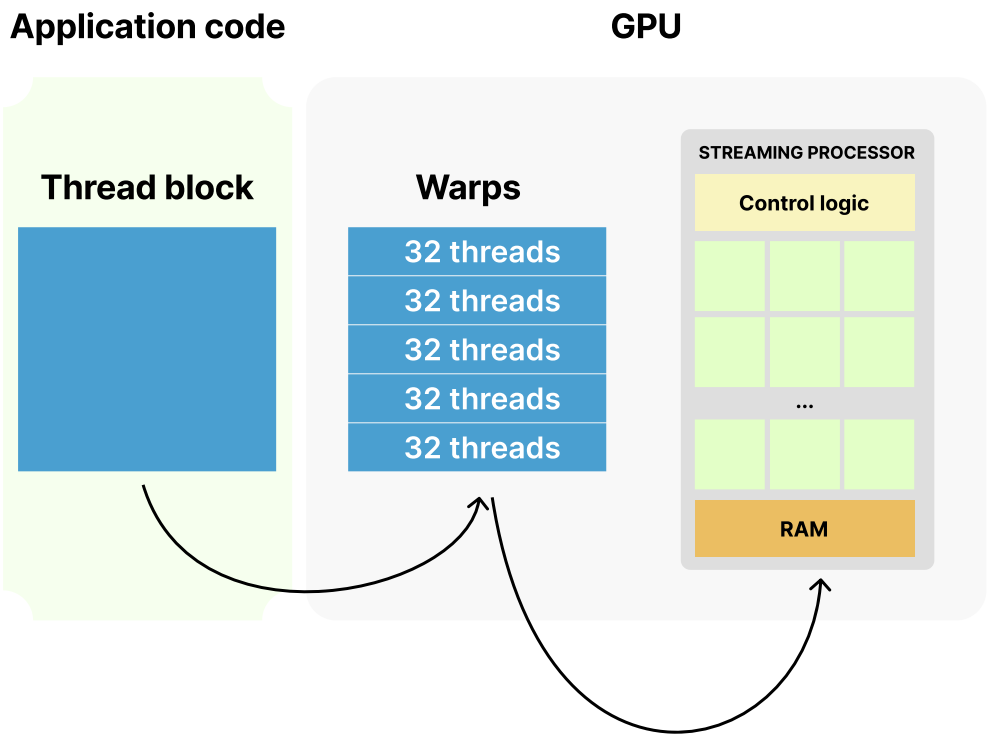


Fig. Diagram showing how application code maps to GPU hardware: a thread block from the application is split into warps of 32 threads each, which are then assigned to a Streaming Processor with control logic, CUDA cores, and RAM

Each SM contains CUDA cores (where the math happens), shared memory, registers, and warp schedulers.

An A100 has 108 SMs, while a smaller GPU might have 20 to 30.

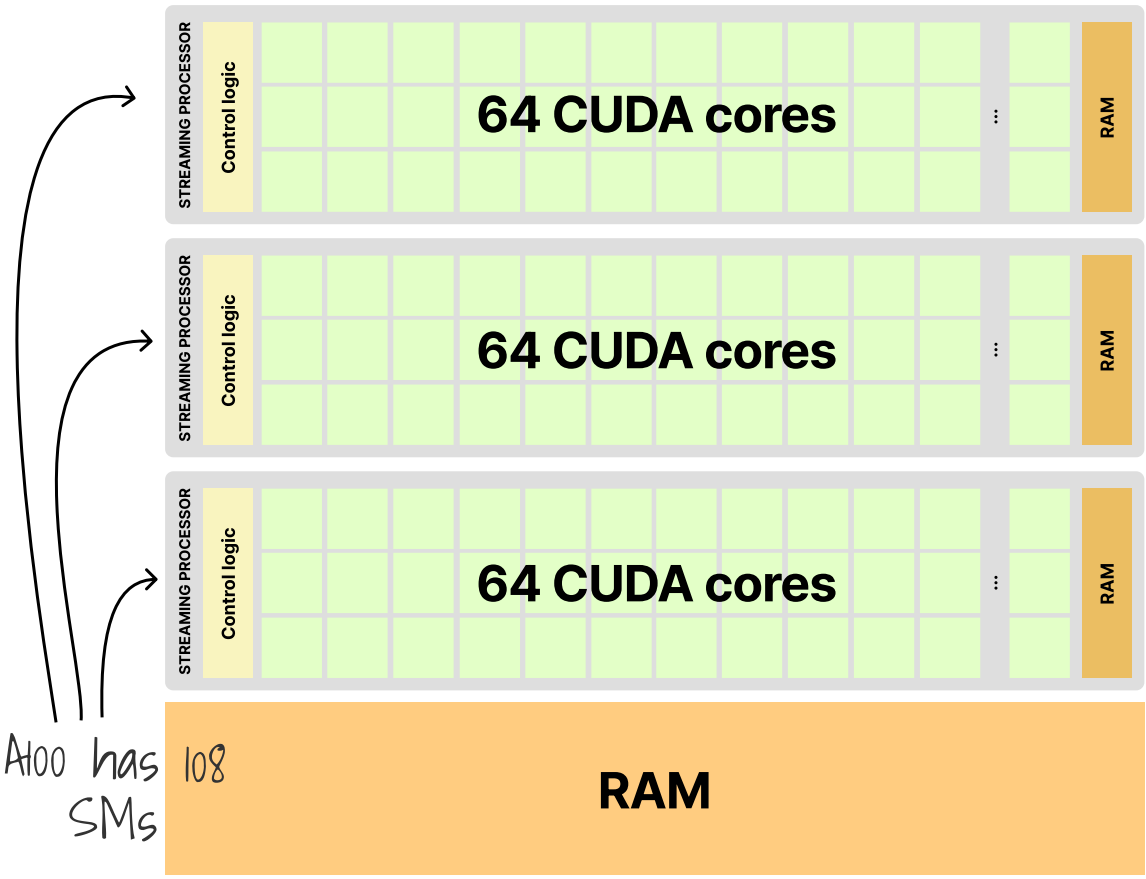


Fig. Diagram of a GPU with three Streaming Multiprocessors stacked vertically, each containing control logic, 64 CUDA cores and local RAM, with shared RAM at the bottom and a note that an A100 has 108 SMs

The GPU's job is to keep these SMs supplied with warps:

- When an SM has warps assigned and those warps are running instructions (not waiting for memory or synchronization), computation is happening.
- When SMs are empty or warps are stalled, the GPU is waiting.

That's why the following metrics focus on SM activity and warp occupancy.
They measure whether the compute units are actually doing work, not just whether "anything is happening".

The Metrics That Predict Yield

To get closer to *"is the GPU actually doing parallel work?"*, you use `SM Active` and `SM Occupancy` from DCGM or dcgm-exporter when profiling counters are supported and enabled in your driver/DCGM setup.

- `SM Active` is defined as the ratio of cycles in which a Stream Multiprocessor (SM) has at least one warp assigned.
- `SM Occupancy` tells you how full the SM is—the ratio of warps currently loaded versus the maximum the SM can hold.

If `SM Active` is high, the cores are busy.
If it's low, the GPU is spending a lot of time waiting.

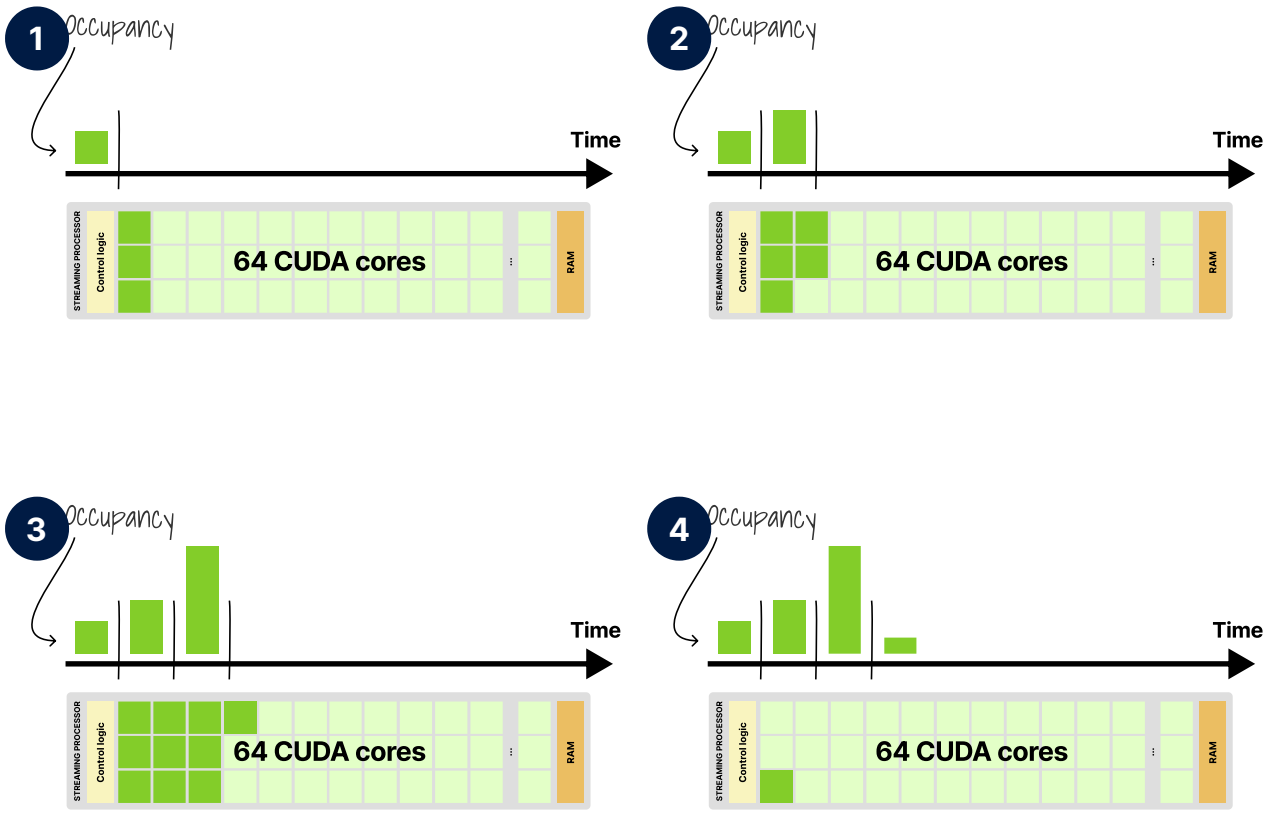


Fig. 1 One warp loaded into the SM — occupancy is low.

Fig. 2 Two warps loaded — more CUDA core slots are filled and occupancy grows.

Fig. 3 Three warps loaded — the SM is nearly full and occupancy is high.

Fig. 4 Warps drop off and most slots are empty — occupancy falls.

If occupancy is low, you're not providing enough parallel work to keep the hardware full, even if kernels are launching continuously.

Together, these two metrics help you tell the difference between just launching kernels and the GPU having enough real work to stay busy.

But they still don't guarantee high throughput, because warps can still stall, most often due to memory latency or synchronization.

That's why you can see high `GPU-Util` while `SM Active` is low.

The GPU looks busy because something is running in each sampling window, but the compute cores aren't spending most of their time doing useful math.

In that case, the GPU isn't idle.

It's blocked, and progress is slow.

To find what's blocking you, look at data movement and memory pressure, not just "memory used."

When metrics show high transfer activity but low SM activity, the GPU is waiting for data to be transferred.

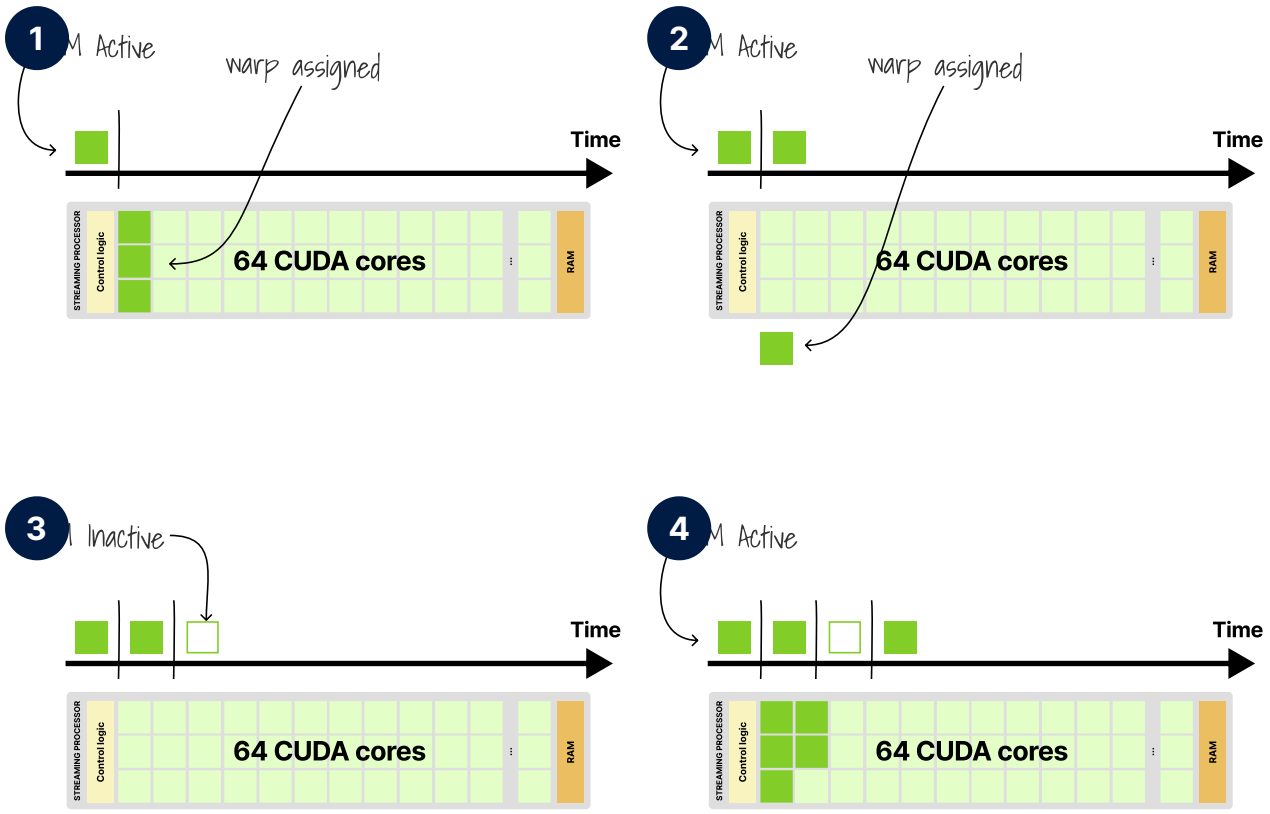


Fig. 1 One cycle with a warp assigned — the SM counts as active.

Fig. 2 Two cycles, both with warps assigned — the SM stays active across both.

Fig. 3 The third cycle has no warp — the SM is inactive for that cycle.

Fig. 4 Three out of four cycles have warps — SM Active measures this ratio.

When memory activity is high but SM activity is low, the memory system is the bottleneck.

How Data Moves

Compute happens on the SMs, but data has to get there first.

Understanding where bottlenecks can occur requires knowing the three paths data travels.

1. **Host to GPU (PCIe):** Data moves from CPU memory to GPU memory over the PCIe bus. This is the slowest link in most systems, usually 16-32 GB/s depending on PCIe generation.
2. **GPU to GPU (NVLink):** In multi-GPU systems, GPUs can transfer data directly to each other over NVLink, bypassing PCIe entirely. NVLink is much faster (300-900 GB/s depending on generation), but only matters if your workload spans multiple GPUs.
3. **GPU memory (DRAM):** The GPU has its own high-bandwidth memory (HBM on datacenter GPUs, GDDR on consumer cards). When kernels run, they read inputs and write outputs to this memory.

An A100's memory bandwidth is about 2 TB/s, which is fast but can still be a bottleneck if your workload is memory-bound instead of compute-bound.

Now that you know where data travels, here's how to tell which path is the bottleneck.

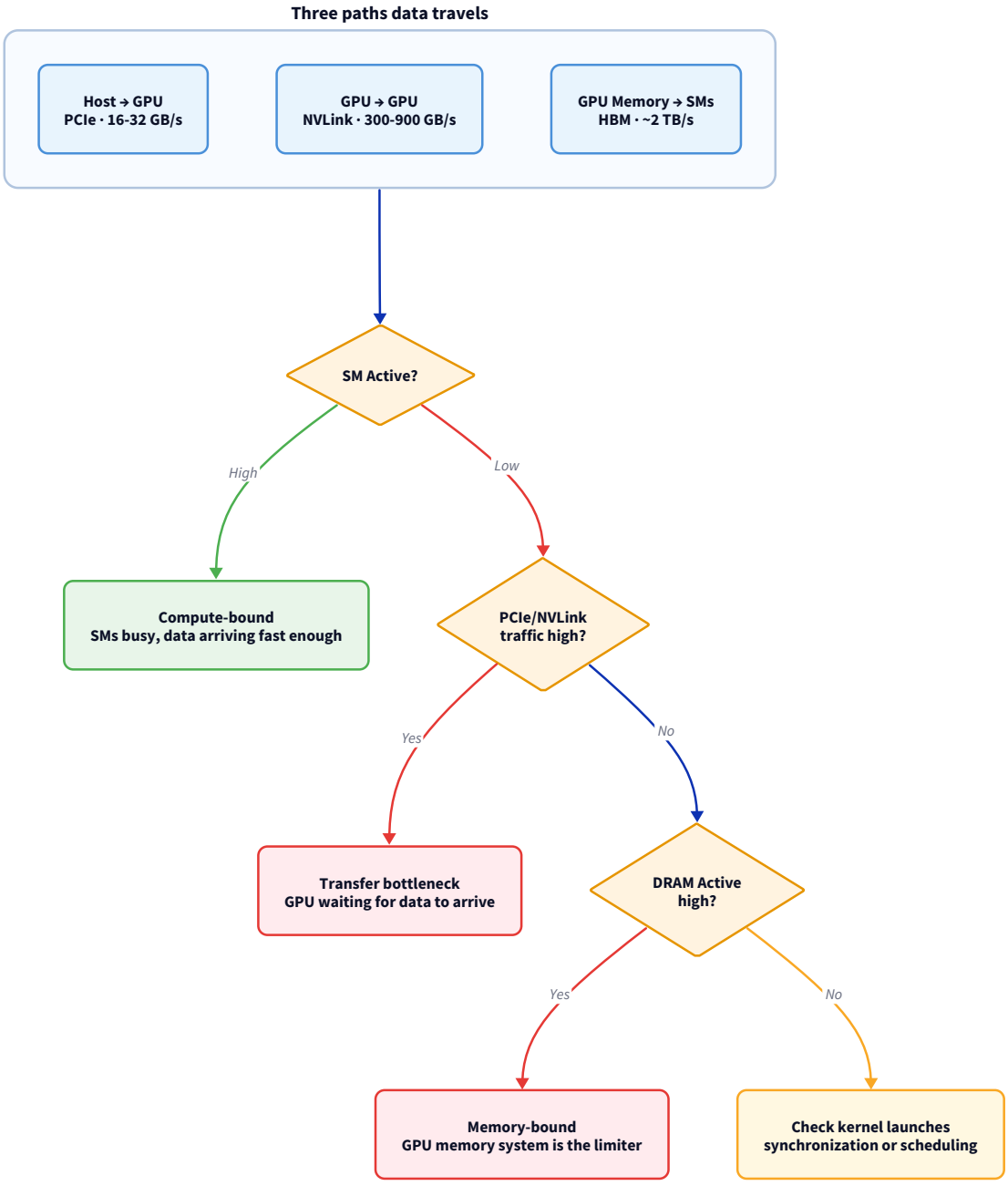


Fig. Flowchart showing the three data paths to the GPU SMs (PCIe at 16-32 GB/s, NVLink at 300-900 GB/s, and HBM at around 2 TB/s) and a decision tree for diagnosing bottlenecks: if SM Active is high the workload is compute-bound, if low check PCIe/NVLink traffic for a transfer bottleneck, then DRAM Active for a memory bottleneck

Device memory activity and achieved bandwidth tell you whether the on-GPU memory system is the limiter.

Use `DCGM_FI_PROF_DRAM_ACTIVE` to measure this.

PCIe or NVLink throughput tells you whether transfers are the limiter.

Use `DCGM_FI_PROF_PCIE_RX_BYTES` and `DCGM_FI_PROF_PCIE_TX_BYTES` for host-GPU transfers, or `DCGM_FI_PROF_NVLINK_RX_BYTES` and `DCGM_FI_PROF_NVLINK_TX_BYTES` for GPU-GPU transfers in multi-GPU systems.

If transfer traffic is high while `SM Active` is low, you are likely waiting for data to arrive.

If memory activity is high while `SM Active` is low, you are likely limited by the memory system or by inefficient access patterns.

If `SM Active` is high while transfer signals are low, you are more likely to be compute-limited.

Power and clocks are supporting signals.

Sustained heavy compute usually runs at higher clocks and draws more power, while light or heavily stalled work often sits lower, but neither by itself proves productivity.

Tensor Core activity matters only when your workload requires mixed precision.

If you expect FP16 or BF16 matrix math and tensor operations to stay low, you're probably not hitting the fast path and are leaving performance on the table.

Temperature and throttling matter because they change how fast each cycle runs.

If clocks drop due to thermal or power constraints, you can still see "100% utilization" while throughput falls.

Finally, compute engine usage tells you which hardware blocks are actually active.

A GPU can look busy while tensor cores are mostly idle or while copy engines are saturated, and those situations need different fixes.

Mapping Metrics to Workload Types

Different workloads require different metrics because they consume GPU resources differently.

Training is usually steady and long-running.

The GPU stays busy for extended periods, so you expect sustained compute activity, sustained memory system activity, and stable clocks.

In this case, low SM activity compared to `GPU Util` is a red flag because there are no natural idle gaps to explain it.

You focus on whether SM activity stays high over time windows, whether the memory system is consistently pressured in a way that matches the model, and whether power, temperature, and clocks remain stable enough to sustain throughput.

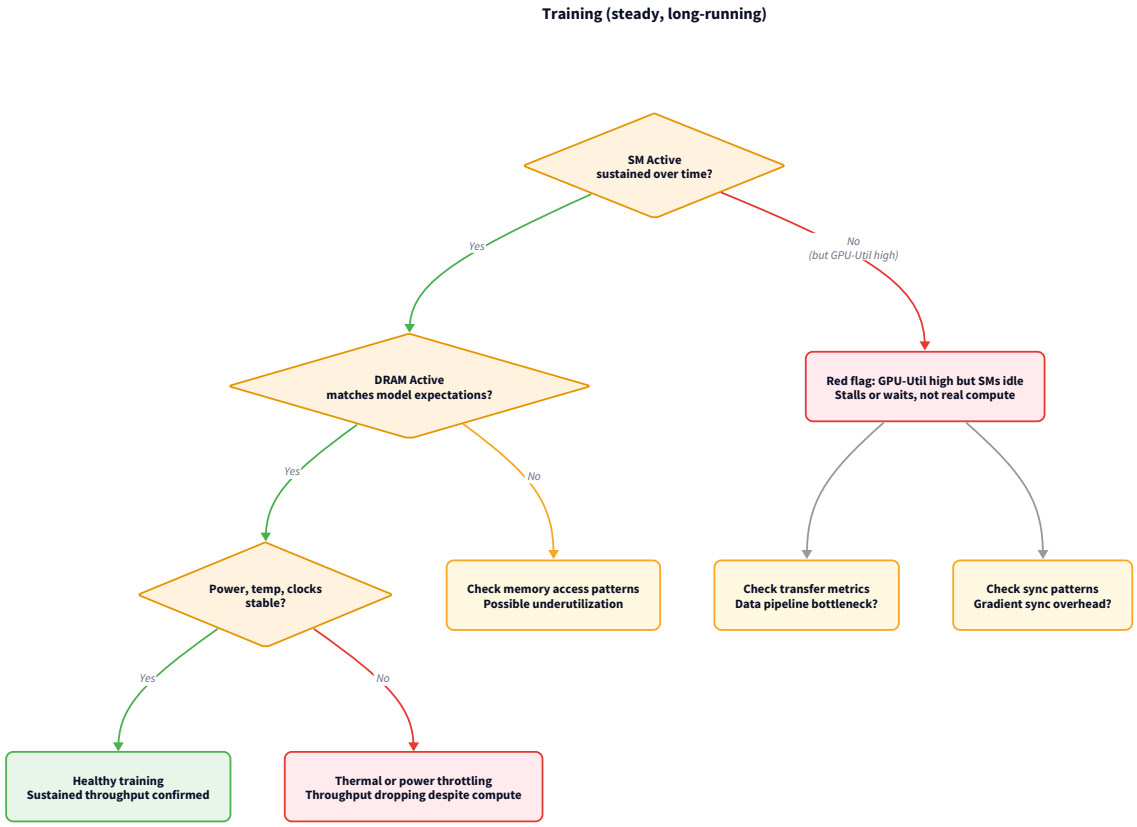


Fig. Flowchart for diagnosing training workloads: check whether SM Active is sustained over time, then whether DRAM Active matches model expectations, then whether power, temperature and clocks are stable, with red-flag paths for high GPU-Util with low SM activity pointing to transfer or sync bottlenecks

Inference behaves differently.

It is bursty and latency-sensitive, and the GPU can be idle between requests by design.

Average utilization is a weak signal here because what matters is what happens during the burst: whether SM activity is high while requests are being served, whether memory system pressure spikes as expected, and whether clocks ramp up reliably when work arrives.

You read GPU counters alongside P50, P95, and P99 latency to see if performance issues show up as tail spikes instead of lower average utilization.

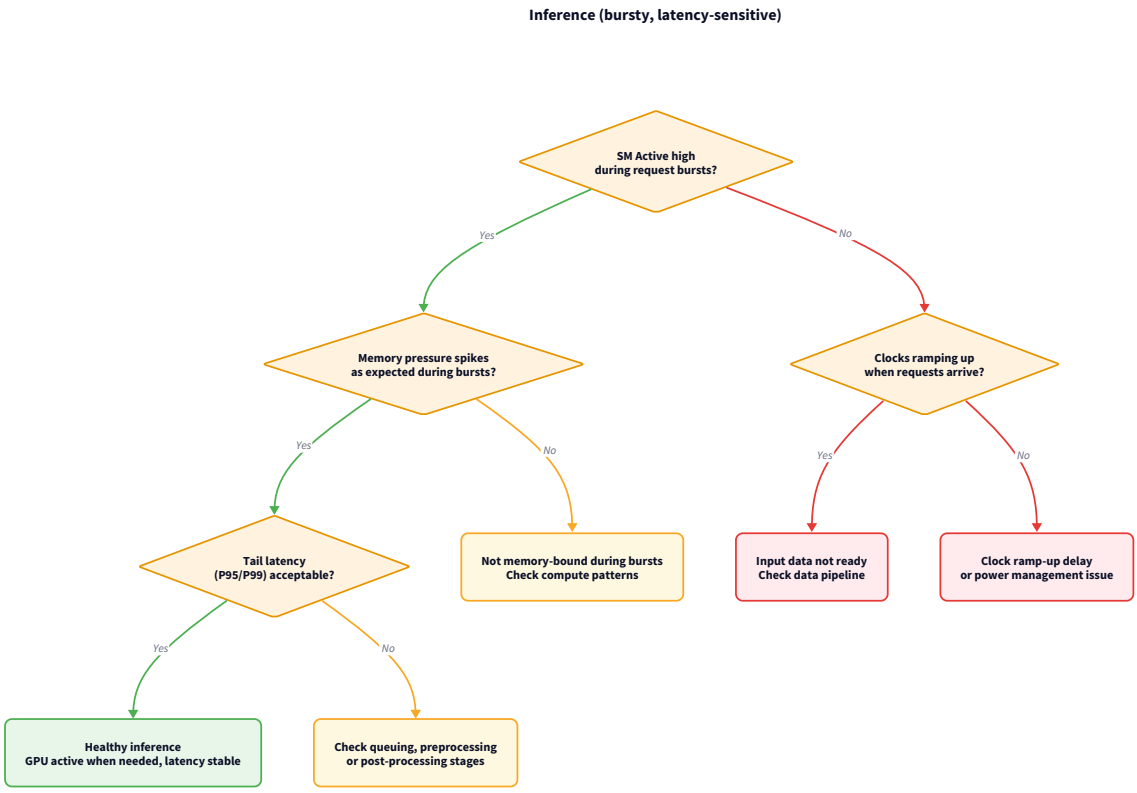


Fig. Flowchart for diagnosing inference workloads: ignore average GPU-Util, check SM Active during request bursts, then memory pressure spikes and tail latency at P95 and P99, with paths for clock ramp-up delays and data pipeline issues

Batch processing sits in between as throughput-driven work that is usually tolerant of delay, which makes it a good candidate for sharing and packing, where the goal is to finish the right amount of work on time rather than keeping the GPU busy every second.

You track throughput over meaningful windows, queue depth, and completion rates, treating instantaneous utilization as secondary.

A job that averages 60% utilization but meets its deadlines is healthier than one that averages 95% utilization but is stuck on transfers or synchronization and finishes late.

This is why you can't read a single GPU number in isolation.

The same 50% SM activity can be normal or a problem, depending on whether the workload is training, inference, or batch, because each has a different timing profile.

Another common mistake is reading the right metric from the wrong layer, since the GPU, Kubernetes, and the application all describe the same reality in different ways.

That is the three-view problem.

The Three-View Problem

There are three ways to describe what is happening on a GPU, and each one is incomplete on its own.

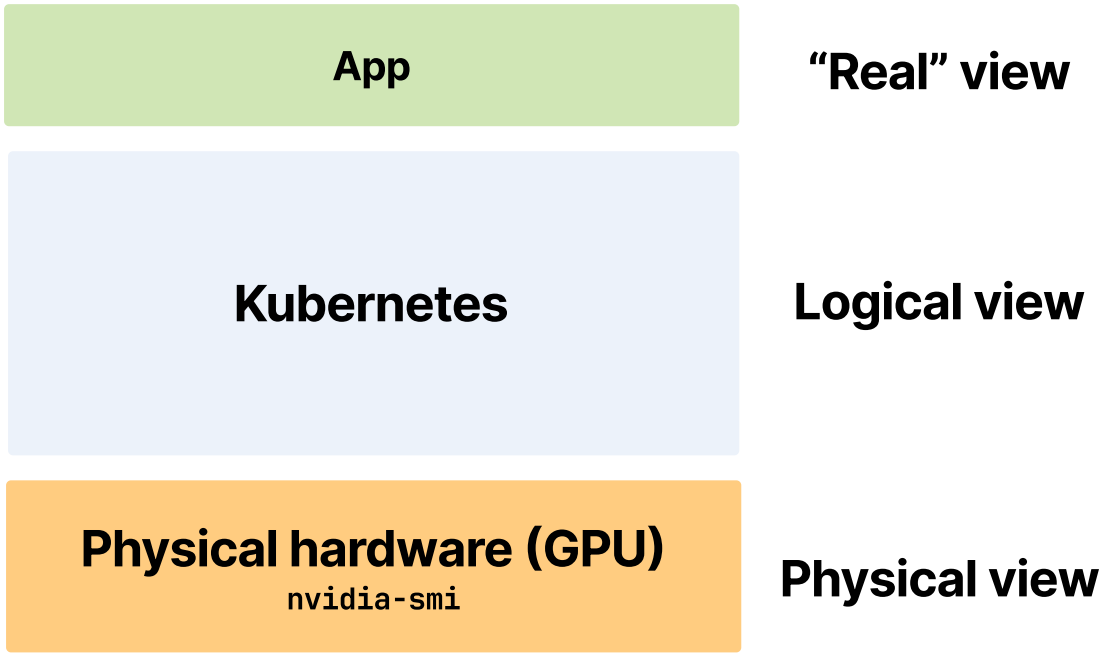


Fig. Three stacked layers representing the three views of GPU activity: the App layer at top labelled real view, the Kubernetes layer in the middle labelled logical view, and the Physical hardware GPU layer at bottom with nvidia-smi labelled physical view

`nvidia-smi` is the physical view.

It tells you what the driver sees on the device: utilization over a sampling window, memory allocated at the process level, temperature, clocks, and sometimes power.

It's useful for understanding the hardware state, but it doesn't know about pods, namespaces, or why the work exists.

It can show that memory is allocated, but it can't tell you which Kubernetes object should be accountable for it unless you connect process identity to a pod.

Kubernetes is the logical view.

It only knows what the scheduler assigned: which pod claimed which GPU.

It doesn't know whether the pod is actually using the GPU.

Once a pod gets a GPU, Kubernetes counts that GPU as "allocated," even if the pod is idle or stuck.

The workload is the real view.

It tells you what the job actually achieves: throughput, latency, step time, queue depth, accuracy, and often, framework memory behavior.

The problem is that these views don't match by default.

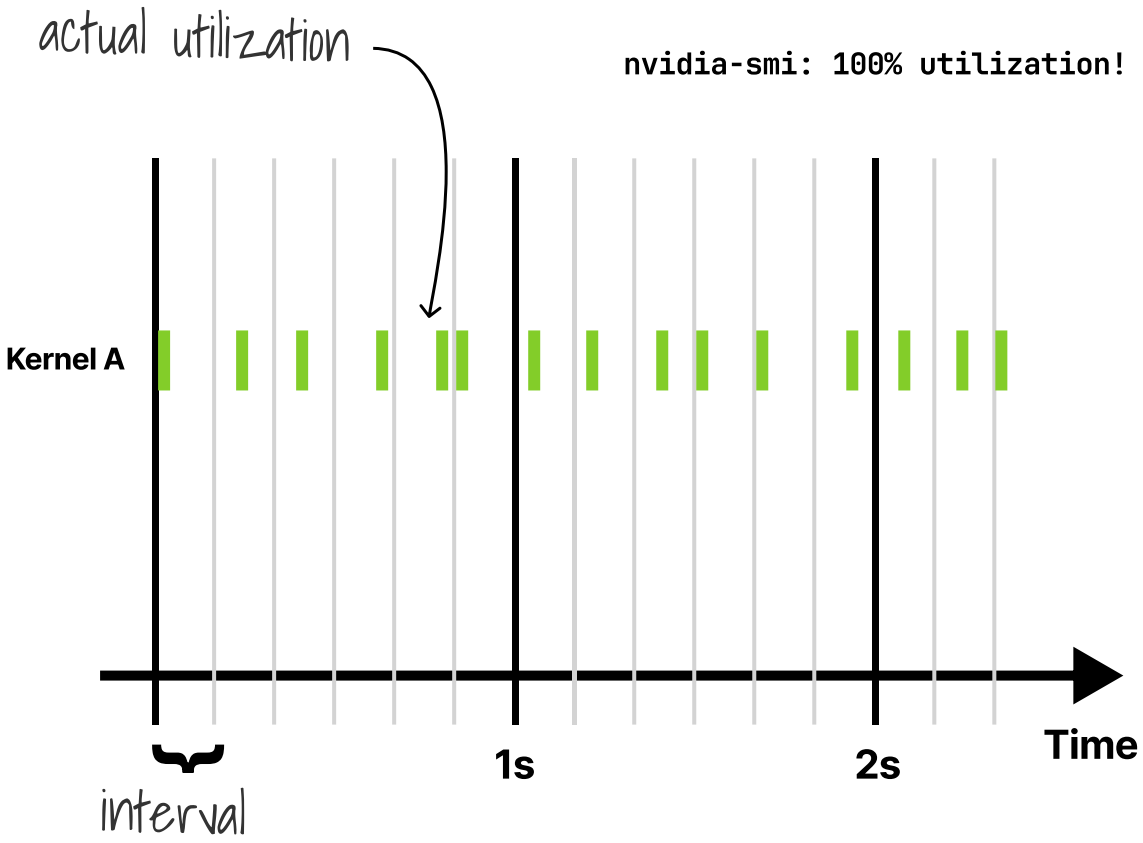


Fig. Timeline showing Kernel A running in short bursts across sampling intervals, with small green blocks of actual work scattered along the time axis but nvidia-smi reporting 100% utilization because at least one kernel ran in every sample window

A pod can own a GPU in Kubernetes even when SM activity is low, and the job makes little progress.

Or a job can reserve lots of memory, while live tensors are much smaller.

Reconciling the three views just means linking them so you can answer one question: which pod used the GPU, how much it used, and what you got out of it?

DCGM gives the GPU counters, Kubernetes exporters show what was allocated, and application metrics show the outcome.

Building the Observability Stack

Now you know the key point: **"busy" doesn't mean "productive."**

So the fix is to collect three things together:

1. GPU counters,
2. Kubernetes allocation,
3. and what the workload is actually achieving.

DCGM is a good starting point because it exposes the GPU signals you care about, and `dcgm-exporter` can publish them to Prometheus.

If pod attribution is enabled, those GPU metrics also get pod and namespace labels, so you can tell who is using what.

Tool choice is secondary.

What matters is whether you can answer basic questions like: which pod is using GPU memory right now, which namespace is holding GPUs that are mostly idle, and whether performance is getting worse over time before users notice.

`dcgm-exporter` runs as a DaemonSet on GPU nodes and serves metrics on port 9400 by default.

For it to work properly, it needs to run only on GPU nodes, it needs permission to read pods and nodes from the Kubernetes API, and it needs

access to the kubelet pod-resources data so it can map "this GPU" to "this pod." Start by creating the namespace and the RBAC resources that allow `dcgm-exporter` to read pod and node information from the Kubernetes API:

bash

```
$ kubectl create namespace dcgm-exporter
```

Let's create the RBAC resources that allow `dcgm-exporter` to read pod and node information:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ServiceAccount
metadata:
  name: dcgm-exporter
  namespace: dcgm-exporter
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRole
metadata:
  name: dcgm-exporter
rules:
- apiGroups: [""]
  resources: ["pods", "nodes"]
  verbs: ["get", "list"]
---
apiVersion: rbac.authorization.k8s.io/v1
kind: ClusterRoleBinding
metadata:
  name: dcgm-exporter
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: ClusterRole
  name: dcgm-exporter
subjects:
- kind: ServiceAccount
  name: dcgm-exporter
  namespace:
```

```
  dcgm-exporter
EOF
```

Next, create the ConfigMap that defines which GPU metrics to collect:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: dcgm-exporter-config
  namespace: dcgm-exporter
data:
  default-counters.csv: |
    DCGM_FI_DEV_SM_CLOCK, gauge, SM clock frequency (MHz)
    DCGM_FI_DEV_MEM_CLOCK, gauge, Memory clock frequency (MHz)
    DCGM_FI_DEV_GPU_TEMP, gauge, GPU temperature (C)
    DCGM_FI_DEV_POWER_USAGE, gauge, Power draw (W)
    DCGM_FI_DEV_TOTAL_ENERGY_CONSUMPTION, counter, Total energy consumption (mJ)
    DCGM_FI_DEV_FB_USED, gauge, Framebuffer memory used (MiB)
    DCGM_FI_DEV_FB_FREE, gauge, Framebuffer memory free (MiB)
    DCGM_FI_PROF_GR_ENGINE_ACTIVE, gauge, Graphics engine active (%)
    DCGM_FI_PROF_SM_ACTIVE, gauge, SM active (%)
    DCGM_FI_PROF_SM_OCCUPANCY, gauge, SM occupancy (%)
    DCGM_FI_PROF_PIPE_TENSOR_ACTIVE, gauge, Tensor pipeline active (%)
    DCGM_FI_PROF_DRAM_ACTIVE, gauge, DRAM active (%)
    DCGM_FI_PROF_PCIE_TX_BYTES, counter, PCIe transmit bytes
    DCGM_FI_PROF_PCIE_RX_BYTES, counter, PCIe receive bytes
    DCGM_FI_PROF_NVLINK_TX_BYTES, counter, NVLink transmit bytes
    DCGM_FI_PROF_NVLINK_RX_BYTES, counter, NVLink receive bytes
EOF
```

Finally, deploy the DaemonSet and expose it as a Service:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: dcgm-exporter
  namespace: dcgm-exporter
spec:
  selector:
    matchLabels:
```

```
      app: dcgm-exporter
  template:
    metadata:
      labels:
        app: dcgm-exporter
    spec:
      serviceAccountName: dcgm-exporter
      hostNetwork: true
      hostPID: true
      containers:
      - name: exporter
        image: nvcr.io/nvidia/k8s/dcgm-exporter:3.3.5-3.4.0-ubuntu22.04
        env:
        - name: DCGM_EXPORTER_LISTEN
          value: ":9400"
        - name: DCGM_EXPORTER_KUBERNETES
          value: "true"
        - name: DCGM_EXPORTER_KUBERNETES_GPU_ID_TYPE
          value: "device-name"
        ports:
        - containerPort: 9400
          name: metrics
        volumeMounts:
        - name: config
          mountPath: /etc/dcgm-exporter
        - name: pod-resources
          mountPath: /var/lib/kubelet/pod-resources
          readOnly: true
        securityContext:
          privileged: true
      volumes:
      - name: config
        configMap:
          name: dcgm-exporter-config
      - name: pod-resources
        hostPath:
          path: /var/lib/kubelet/pod-resources
---
apiVersion: v1
kind: Service
metadata:
  name: dcgm-exporter
  namespace: dcgm-exporter
spec:
  selector:
    app: dcgm-exporter
  ports:
  - port: 9400
    targetPort: 9400
    name: metrics
EOF
```

Two details here are easy to miss:

1. **The nodeSelector must match how your cluster labels GPU nodes;** if your cluster uses different labels or requires tolerations, you must adjust scheduling to match your environment.
2. **Kubernetes attribution only works when the exporter can both read the Kubernetes API through RBAC and read the kubelet's pod-resources data at the mounted path.** If either is missing, you'll still get GPU metrics but not pod or namespace labels.

First, confirm the pod and the service:

bash

```
$ kubectl get ds -n dcgm-exporter
NAME            DESIRED   CURRENT   READY   UP-TO-DATE   AVAILABLE   NODE SELECTOR                  AGE
dcgm-exporter   0         0         0       0            0           nvidia.com/gpu.present=true    4d22h

kubectl get pod -n dcgm-exporter
NAME                  READY   STATUS    RESTARTS   AGE
dcgm-exporter-r5jnh   1/1     Running   0          8s

kubectl get svc -n dcgm-exporter
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
dcgm-exporter   ClusterIP   10.100.147.130   <none>        9400/TCP   4d23h
```

Then inspect the metrics endpoint. The simplest way is port-forward:

bash

```
$ kubectl port-forward -n dcgm-exporter svc/dcgm-exporter 9400:9400
Forwarding from 127.0.0.1:9400 → 9400
```

Now check if pod attribution is working by deploying a GPU workload:

bash

```
$ kubectl create namespace metrics-demo
namespace/metrics-demo created
```

Let's create a Pod that runs a GPU-heavy workload:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: gpu-busy
  namespace: metrics-demo
spec:
  restartPolicy: Never
  containers:
  - name: worker
    image: pytorch/pytorch:2.1.0-cuda12.1-cudnn8-runtime
    command:
    - bash
    - -lc
    - |
      python - <<'PY'
      import torch, time
      x=torch.randn(4096,4096, device='cuda')
      y=torch.randn(4096,4096, device='cuda')
      end=time.time()+600
      while time.time()<end:
          z=x@y
          torch.cuda.synchronize()
      PY
    resources:
      limits:
        nvidia.com/gpu: 1
EOF
pod/gpu-busy created
```

Wait for the pod to be ready:

bash

```
$ kubectl wait --for=condition=ready pod -n metrics-demo gpu-busy --timeout=60s
pod/gpu-busy condition met
```

Query metrics to check pod attribution:

bash

```
$ curl -s localhost:9400/metrics | grep -m 5 'namespace="metrics-demo"'
DCGM_FI_DEV_SM_CLOCK{gpu="0",UUID="GPU-11ba...",device="nvidia0",\
  model="RTX-3050",host="minikube",container="worker",\
  namespace="metrics-demo",pod="gpu-busy"} 1770
DCGM_FI_DEV_MEM_CLOCK{gpu="0",UUID="GPU-11ba...",device="nvidia0",\
  model="RTX-3050",host="minikube",container="worker",\
  namespace="metrics-demo",pod="gpu-busy"} 405
DCGM_FI_DEV_GPU_TEMP{gpu="0",UUID="GPU-11ba...",device="nvidia0",\
  model="RTX-3050",host="minikube",container="worker",\
  namespace="metrics-demo",pod="gpu-busy"} 37
DCGM_FI_DEV_FB_FREE{gpu="0",UUID="GPU-11ba...",device="nvidia0",\
  model="RTX-3050",host="minikube",container="worker",\
  namespace="metrics-demo",pod="gpu-busy"} 7146
DCGM_FI_DEV_FB_USED{gpu="0",UUID="GPU-11ba...",device="nvidia0",\
  model="RTX-3050",host="minikube",container="worker",\
  namespace="metrics-demo",pod="gpu-busy"} 442
```

Pod attribution is working: metrics show `container="worker"`, `namespace="metrics-demo"`, and `pod="gpu-busy"`.
This means dcgm-exporter successfully mapped GPU 0 to pod `gpu-busy`.
Now, before we introduce more parts of the observability stack, let's look at the metrics that show real work.

Diagnosing GPU Yield

Your dashboard shows 80% GPU utilization, but the training job is slower than expected.
Let's trace the problem.

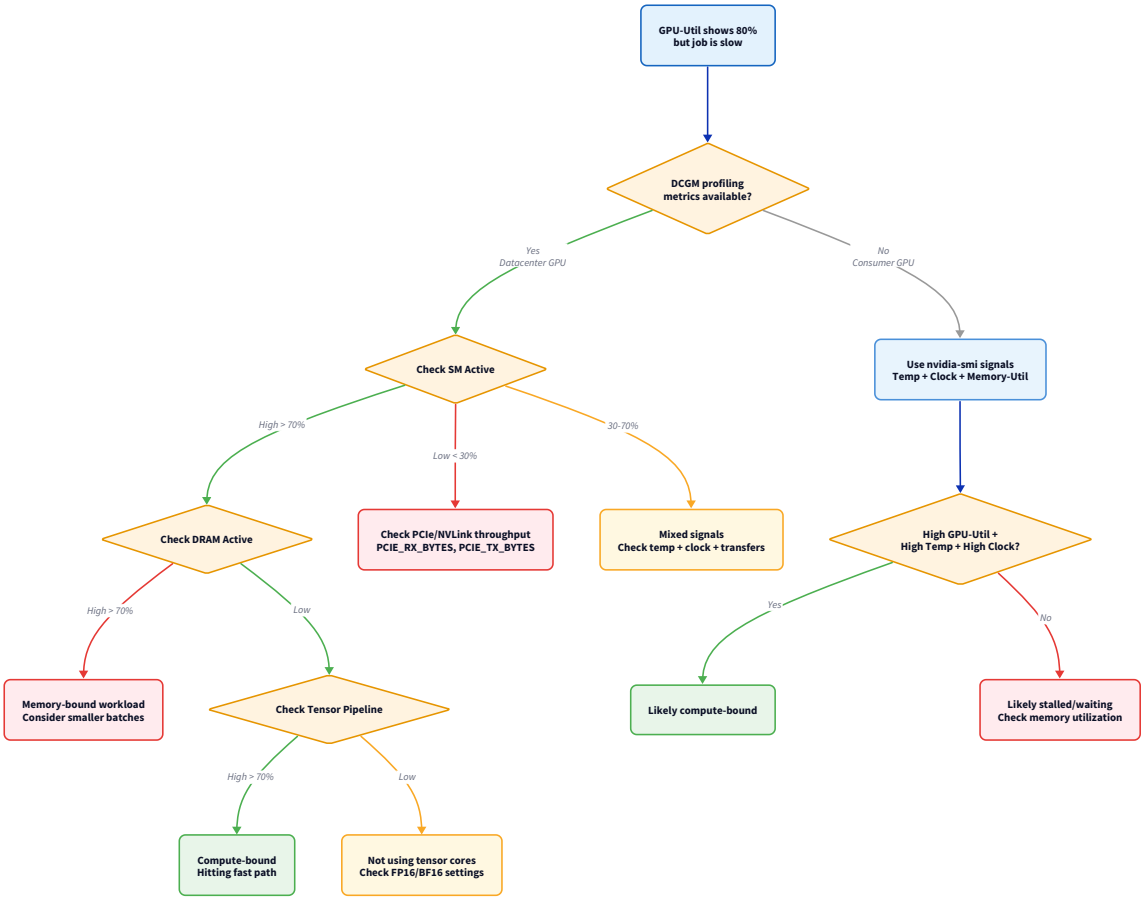


Fig. Flowchart for diagnosing GPU performance: starting from high GPU-Util but slow job, first check if DCGM profiling metrics are available, then follow a decision tree through SM Active, DRAM Active and Tensor Pipeline checks, with a fallback path using nvidia-smi temperature and clock signals for consumer GPUs

These diagnostic steps are based on datacenter GPUs with DCGM profiling metrics turned on.

If you are using consumer GPUs or working in restricted environments, you might not see `DCGM_FI_PROF_*` metrics.

If that happens, you can use `nvidia-smi` `GPU-Util`, memory usage, temperature, and clock speed as substitutes.

A steady high temperature above 60°C with high clock speeds usually means the GPU is under compute load.

On the other hand, high `GPU-Util` with low temperature can point to stalls or inefficient kernel launches.

Step 1: Is the GPU actually computing?

Check `SM Active` to see if compute cores are busy:

example-datacenter-gpu

```
$ curl -s localhost:9400/metrics | grep 'DCGM_FI_PROF_SM_ACTIVE'
DCGM_FI_PROF_SM_ACTIVE{gpu="0",pod="training-job",namespace="ml-team"} 85.2
```

You see: `SM Active` at 85%. **This means:** Compute cores are busy most of the time. **Next step:** Check if memory is the bottleneck.

You see: `SM Active` below 30%. **This means:** The GPU is mostly waiting, not computing. **Next step:** Check transfer metrics—data might not be arriving fast enough. Look for `DCGM_FI_PROF_PCIE_RX_BYTES` and `DCGM_FI_PROF_PCIE_TX_BYTES` for host-to-GPU transfers, or `DCGM_FI_PROF_NVLINK_RX_BYTES` and `DCGM_FI_PROF_NVLINK_TX_BYTES` for multi-GPU communication. High transfer activity with low `SM Active` indicates the GPU is waiting for data.

Step 2: Is memory the bottleneck?

Check DRAM Active to see if the memory system is saturated:

example-datacenter-gpu

```
$ curl -s localhost:9400/metrics | grep 'DCGM_FI_PROF_DRAM_ACTIVE'
DCGM_FI_PROF_DRAM_ACTIVE{gpu="0",pod="training-job"} 72.5
```

You see: DRAM Active above 70%. **This means:** Memory-bound workload. The GPU spends most of its time waiting for data from its own memory. **Next step:** Consider smaller batch sizes, or check for inefficient memory access patterns.

You see: DRAM Active low, `SM Active` high. **This means:** Compute-bound workload. This is usually what you want. **Next step:** Check if you're hitting tensor cores.

Step 3: Are tensor cores being used?

If your workload uses mixed-precision (FP16/BF16), check tensor pipeline activity:

example-datacenter-gpu

```
$ curl -s localhost:9400/metrics | grep 'DCGM_FI_PROF_PIPE_TENSOR_ACTIVE'
DCGM_FI_PROF_PIPE_TENSOR_ACTIVE{gpu="0",pod="bert-training"} 78.3
```

You see: Tensor pipeline above 70%. **This means:** Mixed-precision ops are hitting the fast path. **Next step:** You're in good shape—monitor for throttling.

You see: Tensor pipeline low, but expecting FP16/BF16 math. **This means:** Not hitting tensor cores, so you're leaving performance on the table. **Next step:** Check framework settings, model dtype, or whether ops are falling back to FP32.

Step 4: Confirming sustained load

A temperature of 65°C means sustained activity.

Idle GPUs usually sit at 30-40°C.

bash

```
$ curl -s localhost:9400/metrics | grep 'DCGM_FI_DEV_GPU_TEMP'
DCGM_FI_DEV_GPU_TEMP{gpu="0",pod="training-job"} 65

curl -s localhost:9400/metrics | grep 'DCGM_FI_DEV_FB_USED'
DCGM_FI_DEV_FB_USED{gpu="0",pod="training-job"} 802
```

Framebuffer usage (about 802 MiB here) describes workload size, but doesn't tell you whether you're compute-bound or memory-bound.

To recap:

- `SM Active` tells you if the GPU is computing or waiting.
- **DRAM Active** tells you if memory bandwidth is the bottleneck.
- **Tensor pipeline** tells you if mixed-precision is hitting the fast path.
- **Temperature and framebuffer** confirm activity, but don't diagnose problems.

Key Takeaways

GPU-Util is easy to misread: in `nvidia-smi`, it's a short-window activity metric, meaning the percentage of time in the last sample period when one or more kernels were running.

It's not a throughput, efficiency, or 'useful work' score.

`SM Active` reveals the truth: when GPU Util is high but `SM Active` is low, the GPU is technically "busy", but the compute cores aren't doing useful math; they're stalled, waiting.

The three views live in separate worlds: `nvidia-smi` shows the physical state, Kubernetes shows allocation, and workload shows the actual output.

But these don't match by default because a pod can claim a GPU and remain completely idle.

Use clocks as a relative signal: compare the pod's sustained SM clock to the GPU's known idle range and typical load range.

If a pod holds the GPU but the SM clock stays near the device's idle behavior for long periods, that's a strong sign of idle allocation waste.

But clocks alone still don't prove useful throughput.

DCGM with pod attribution is critical because dcgm-exporter tags GPU metrics with pod and namespace labels.

This turns "the GPU is busy" into "this pod holds 6GB but delivers zero throughput."

Dashboards make waste patterns visible with three panels: GPU clock per pod shows hoarders as flat, low lines; memory by namespace enables cost attribution; and temperature with thresholds detects throttling risk.

The Architecture Decision

You've measured your GPUs and understand what they're doing.

Now comes the tough question: which architecture will actually improve yield for your workloads?

You won't find the answer in vendor documentation or best practices guides.

Those resources assume all workloads behave the same way, but that's not true.

An inference workload that bursts for 50ms every few seconds needs a very different setup than a training job that uses the GPU nonstop for six hours.

Choosing the wrong architecture can cost real money.

Chapter 1 showed that allocation isn't the same as utilization, and Chapter 2 showed that utilization isn't the same as productivity.

Now, we'll connect these ideas to the architecture decision.

This choice determines whether high utilization leads to high yield or just causes costly contention on hardware you already own.

In this chapter, we'll test three architectures using real workloads, measure their yield, and figure out when each one is the right choice.

Three Architectures, Three Trade-Offs

You have three fundamental choices when deploying GPU workloads in Kubernetes, and each creates completely different yield characteristics by changing how workloads compete for resources:

1. Share one large GPU across multiple pods using time-slicing
2. Deploy many smaller dedicated GPUs with one pod per GPU, or
3. Use MIG to partition a large GPU into isolated slices that get hardware-level boundaries.

To make the right decision, you need three key pieces of information from your measurements.

First, find out what your workload actually uses during execution.

Look at DCGM metrics for SM active, memory bandwidth, and power draw while the workload is running.

Second, consider the usage pattern over time.

Is it steady-state compute that runs nonstop, bursty inference that spikes and idles, or batch processing that starts and finishes?

Third, think about isolation.

Can your workloads share resources without issues, or do they need hardware boundaries to avoid interference that could hurt latency or throughput?

Let's start with the first option: sharing one large GPU across multiple workloads using time-slicing.

One Large Shared GPU

Time-slicing makes a single physical GPU look like several GPUs to Kubernetes.

By configuring the device plugin to advertise replicas, the scheduler sees four GPUs even though there's only one.

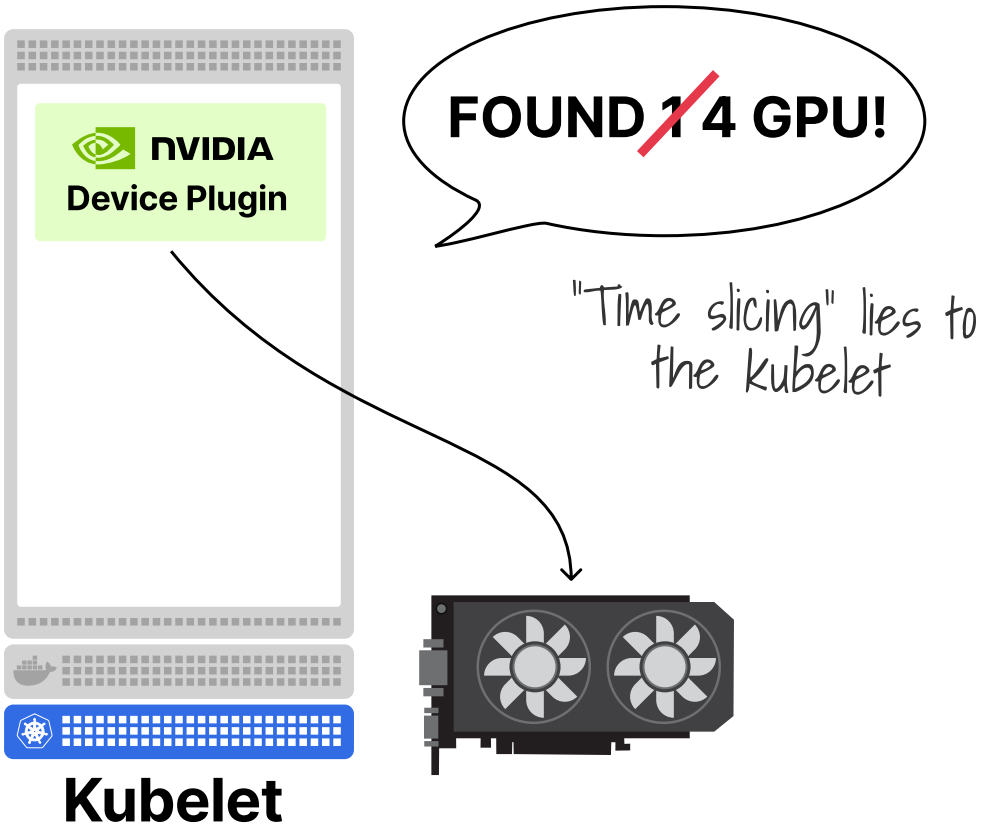


Fig. A node with the NVIDIA Device Plugin reporting 4 virtual GPUs to the kubelet instead of the physical 1

This allows you to place multiple pods on the same hardware, and each pod thinks it has exclusive GPU access.

But all the pods are still using the same physical GPU underneath, and this kind of sharing leads to specific yield patterns.

The main idea is simple: set up the device plugin to show four virtual GPUs instead of one physical GPU.

Then, schedule four pods that each request one GPU, and you're running four workloads where you used to run just one.

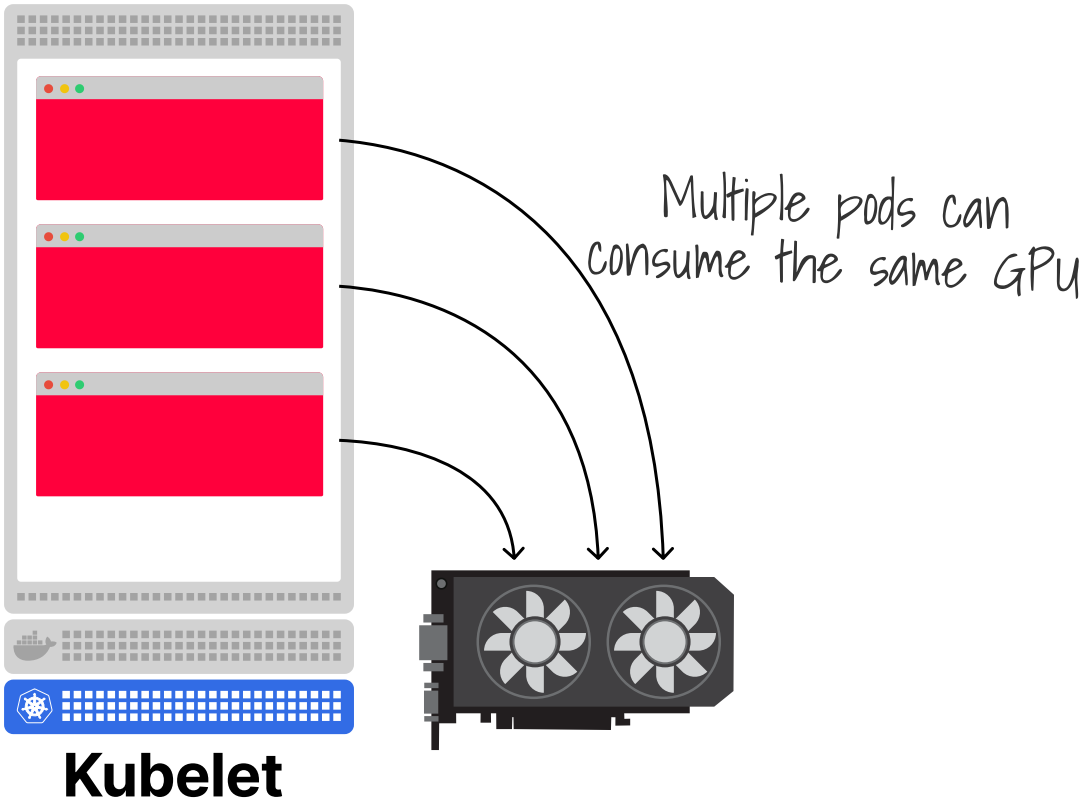


Fig. Three pods on a node all pointing to the same physical GPU, showing that multiple pods can consume the same GPU through time-slicing

The scheduler thinks there are four separate GPUs because that's what the device plugin advertises.

Kubernetes doesn't know these pods are actually sharing the same hardware.

This is where things get interesting, because time-slicing works brilliantly in some scenarios and fails catastrophically in others, and understanding which scenario you're in requires looking at actual workload behavior, not theoretical capacity numbers.

To understand why, let's look at how the GPU handles sharing.

Unlike a CPU, which can pause a process at any moment to let another run, a GPU usually runs a "kernel" (a single unit of work) until it's done.

The hardware only switches between pods at the end of each kernel.
This means there's no fine-grained or fair time-sharing.
If one pod starts a kernel that takes 500ms to finish, all other pods have to wait.

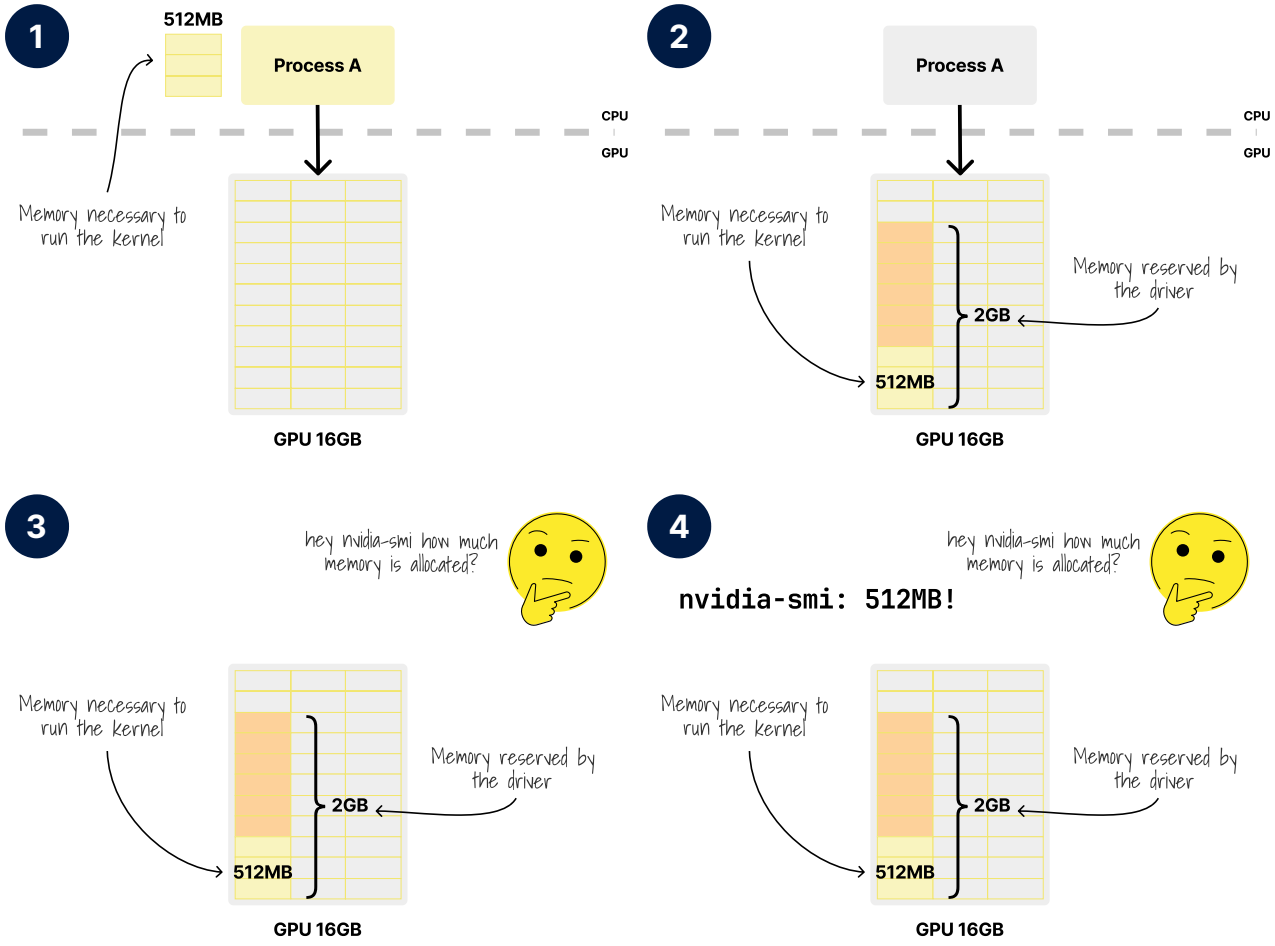


Fig. 1 Process A on the CPU sends 512MB of data to a 16GB GPU to run a kernel.

Fig. 2 The GPU allocates 512MB for the kernel plus 2GB reserved by the driver — more than the kernel strictly needs.

Fig. 3 Asking nvidia-smi how much memory is allocated — but the answer includes the driver overhead.

Fig. 4 nvidia-smi reports 512MB, which only reflects the kernel memory, not the full 2GB reserved by the driver.

There's no way to interrupt a running kernel to let another pod use the GPU.

Because of this "run-to-completion" model, time-slicing only works when workloads naturally leave gaps for each other.

Time-slicing is valuable when workloads have usage patterns that don't overlap.

One workload's idle time becomes another's execution window, so you get better utilization without losing throughput.

For example, bursty inference that runs for 50ms and then waits 200ms can share efficiently with periodic batch jobs that spike briefly and then go idle.

Their active times naturally interleave, so there's no contention.

When workloads don't naturally interleave, time-slicing can cause serious problems.

Time-slicing fails when workloads compete directly for GPU cycles.

There's no hardware isolation, so one workload's behavior can ruin another's performance.

A compute-heavy training job sharing a time-sliced GPU can greatly increase inference latency and jitter.

There's no strict QoS or hardware isolation between replicas.

In practice, this can break SLAs even if the GPU looks busy, but the exact impact depends on time-slice scheduling and each workload's kernel patterns.

Performance becomes unpredictable because one workload's execution pattern affects all others on the same GPU.

Kubernetes can't see or control this interference, since it only tracks allocation numbers, not the GPU scheduler's behavior or kernel times.

So, time-slicing works with complementary patterns and fails with competing ones.

But what do the actual yield numbers look like when you measure them?

Yield Characteristics

Time-slicing can show high utilization in dashboards because several workloads keep the GPU looking busy during the sampling window.

But it's harder to measure per-workload values because GPU-wide metrics are aggregated.

With NVIDIA's device-plugin time-slicing, the DCGM-Exporter cannot

associate metrics to containers/pods, so you typically only get GPU-level numbers unless you redesign instrumentation.

You might see overall metrics that look healthy, but individual workloads can suffer from interference and scheduling effects that standard GPU dashboards can't trace back to a specific pod.

Lost yield can come from context switching, synchronization, kernel behavior, and stalls. You need to check workload throughput and latency to confirm what's really happening.

The GPU might show 89% utilization in nvidia-smi, while SM active is only 47%.

These numbers mean different things.

SM active measures the ratio of cycles where an SM has at least one warp assigned.

The gap between utilization and SM active can be due to stalls, synchronization, kernel patterns, or sampling differences—not just a single overhead factor.

Let's prove this by deploying four different workloads on a shared GPU and watch what happens to yield:

bash

```
$ kubectl create namespace timeslice-demo
namespace/timeslice-demo created
```

Apply the time-slicing configuration:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: time-slicing-config
  namespace: gpu-operator
data:
  any: |
```

```
    version: v1
    flags:
      migStrategy: none
    sharing:
      timeSlicing:
        resources:
        - name: nvidia.com/gpu
          replicas: 4
EOF
configmap/time-slicing-config created
```

Patch the ClusterPolicy to use this configuration:

bash

```
$ kubectl patch clusterpolicy cluster-policy \
  -n gpu-operator \
  --type merge \
  -p '{"spec": {"devicePlugin": {"config": {"name": "time-slicing-config", "default": "any"}}}}'
clusterpolicy.nvidia.com/cluster-policy patched
```

Wait for the device plugin to restart and be ready before checking capacity.

bash

```
$ kubectl wait --for=condition=ready pod -n gpu-operator -l app=nvidia-device-plugin-daemonset --timeout=90s
pod/nvidia-device-plugin-daemonset-yyyyy condition met
```

Now, verify the node shows four available GPUs even though only one physical GPU exists.

bash

```
$ kubectl describe node gpu-node-01 | grep nvidia.com/gpu
  nvidia.com/gpu:     4
  nvidia.com/gpu:     4
```

The scheduler detects four available GPUs and assigns one GPU to each of

four pods that request them.

However, all four pods actually share the same physical hardware, which means their performance can differ from what simple allocation metrics suggest.

Next, consider four different workloads that are common in production clusters:

- Inference, which handles short bursts of activity and then idles.
- Training, which needs steady, continuous access.
- Preprocessing, which does lightweight data transformation with frequent pauses
- Analytics, which runs periodic batch jobs that spike and then go idle.

Each of these workloads uses resources in a different way, and we will measure how they interact when they have to share the same GPU.

First, create the ConfigMap with all four workload scripts:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: workload-configs
  namespace: timeslice-demo
data:
  inference.py: |
    import torch, time
    device = torch.device('cuda:0')
    model = torch.nn.Linear(1024, 1024).to(device)
    while True:
        x = torch.randn(32, 1024, device=device)
        y = model(x)
        torch.cuda.synchronize()
        time.sleep(0.1)  # Bursty pattern
  training.py: |
    import torch
    device = torch.device('cuda:0')
    A = torch.randn(4096, 4096, device=device)
    B = torch.randn(4096, 4096, device=device)
    while True:
        C = torch.matmul(A, B)
```

```
        torch.cuda.synchronize()  # Sustained compute
  preprocess.py: |
    import torch, time
    device = torch.device('cuda:0')
    while True:
        data = torch.randn(1024, 1024, device=device)
        normalized = (data - data.mean()) / data.std()
        time.sleep(0.05)  # Light compute
  analytics.py: |
    import torch, time
    device = torch.device('cuda:0')
    while True:
        data = torch.randn(2048, 2048, device=device)
        torch.cuda.synchronize()
        time.sleep(0.5)  # Periodic batch
EOF
configmap/workload-configs created
```

Then deploy a Pod for each workload. For example, for `inference`:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: inference
  namespace: timeslice-demo
  labels:
    workload: inference
spec:
  containers:
  - name: worker
    image: pytorch/pytorch:2.1.0-cuda12.1-cudnn8-runtime
    command: ["python", "/app/inference.py"]
    volumeMounts:
    - name: configs
      mountPath: /app
    resources:
      limits:
        nvidia.com/gpu: 1
  volumes:
```

```
  - name: configs
    configMap:
      name: workload-configs
EOF
pod/inference created
```

Repeat the same for `training`, `preprocess`, and `analytics`, changing the script names accordingly.

Check nvidia-smi to see the actual hardware state:

bash

```
$ DRIVER_POD="$(kubectl get pod -n gpu-operator -l app=nvidia-driver-daemonset -o jsonpath='{.items[0].metadata.name}')"
kubectl exec -n gpu-operator "$DRIVER_POD" -- nvidia-smi
Fri Jan 19 09:15:32 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.163.01             Driver Version: 550.163.01     CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3050        Off |   00000000:01:00.0  On |                  N/A |
| 42%   58C    P2             N/A /  115W |    3247MiB /   8192MiB |     89%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

The GPU shows 89% utilization and 3247 MB of memory consumed across all four workloads sharing the same physical device, and the temperature climbed to 58 °C, with the fan at 42% to manage the heat.

At first glance, these numbers look good: high utilization, reasonable memory use, and normal temperature.

But they hide what's really happening to your workloads.

Now check DCGM metrics to see actual compute productivity:

bash

```
$ EXPORTER_POD="$(kubectl get pod -n gpu-operator -l app=nvidia-dcgm-exporter -o jsonpath='{.items[0].metadata.name}')"
kubectl exec -n gpu-operator "$EXPORTER_POD" -- curl -s localhost:9400/metrics | grep 'DCGM_FI_PROF_SM_ACTIVE'
DCGM_FI_PROF_SM_ACTIVE{gpu="0"} 47
```

When SM active is at 47%, it means the GPU's execution resources are not being used as well as GPU-Util suggests.

This difference is not just from time-slicing overhead.

It can also come from kernel launch or synchronization overhead, memory stalls, and differences in how the metrics are measured.

Check memory bandwidth to understand if memory is the bottleneck:

bash

```
$ EXPORTER_POD="$(kubectl get pod -n gpu-operator -l app=nvidia-dcgm-exporter -o jsonpath='{.items[0].metadata.name}')"
kubectl exec -n gpu-operator "$EXPORTER_POD" -- curl -s localhost:9400/metrics | grep 'DCGM_FI_PROF_DRAM_ACTIVE'
DCGM_FI_PROF_DRAM_ACTIVE{gpu="0"} 23
```

When memory bandwidth is at 23% and SM active is at 47%, the GPU is not being fully used, even if the allocation dashboard says 100% and nvidia-smi reports 89%.

The hardware might have extra capacity, but sharing it can lower throughput because of scheduling issues, interference, and software overhead.

To understand the real impact, compare workload throughput and latency when running alone versus when shared, instead of assuming time-slicing is the only cause.

These overall numbers do not show which workload is affected.

To find out, isolate each workload and measure its performance when it has the GPU to itself.

Begin by deleting three pods so you can measure the training workload on its own.

bash

```
$ kubectl delete pod inference preprocess analytics -n timeslice-demo
pod "inference" deleted
pod "preprocess" deleted
pod "analytics" deleted
```

Wait a few seconds for the GPU state to stabilize after other workloads terminate.

```bash
$ sleep 10
```

Check nvidia-smi to see training workload behavior when running alone on the GPU.

```bash
$ DRIVER_POD="$(kubectl get pod -n gpu-operator -l app=nvidia-driver-daemonset -o jsonpath='{.items[0].metadata.name}')"
kubectl exec -n gpu-operator "$DRIVER_POD" -- nvidia-smi
Fri Jan 19 09:18:45 2026
+-----------------------------------------------------------------------------------------+
| NVIDIA-SMI 550.163.01             Driver Version: 550.163.01     CUDA Version: 12.4     |
|-----------------------------------------+------------------------+----------------------+
| GPU  Name                 Persistence-M | Bus-Id          Disp.A | Volatile Uncorr. ECC |
| Fan  Temp   Perf          Pwr:Usage/Cap |           Memory-Usage | GPU-Util  Compute M. |
|                                         |                        |               MIG M. |
|=========================================+========================+======================|
|   0  NVIDIA GeForce RTX 3050        Off |   00000000:01:00.0  On |                  N/A |
| 38%   61C    P2             N/A /  115W |    1089MiB /   8192MiB |    100%      Default |
|                                         |                        |                  N/A |
+-----------------------------------------+------------------------+----------------------+
```

Training alone achieves 100% utilization in nvidia-smi and consumes only 1089 MB of memory, which is lower than in the shared case.

This can happen because fewer processes are allocating GPU memory at the same time, but the exact difference depends on each workload's allocations and the framework's caching behavior, not just 'CUDA context overhead'.

The temperature increased slightly to 61 °C despite lower memory usage, as sustained matrix multiplication generates consistent heat.

Now check DCGM SM active to see the real difference in compute productivity.

```bash
$ EXPORTER_POD="$(kubectl get pod -n gpu-operator -l app=nvidia-dcgm-exporter -o jsonpath='{.items[0].metadata.name}')"
kubectl exec -n gpu-operator "$EXPORTER_POD" -- curl -s localhost:9400/metrics | grep 'DCGM_FI_PROF_SM_ACTIVE'
DCGM_FI_PROF_SM_ACTIVE{gpu="0"} 92
```

SM active jumped from 47% when sharing to 92% when running alone, showing a large efficiency drop under sharing.

To quantify the true cost, compare throughput/latency (and completion time)

for the same workload alone vs. shared.

The training workload achieves 92% SM active when it has dedicated access, versus 47% when sharing.

This shows that sharing significantly reduced effective compute execution.

The loss can come from interference and software overheads, so validate with throughput/latency deltas rather than attributing all loss to 'pure overhead'.

Let's see what happens when we put everything back together and measure the complete contention scenario.

Recreate the inference, preprocess analytics pods we deleted earlier, then check the shared GPU metrics one final time:

bash

```
$ kubectl wait --for=condition=ready pod --all -n timeslice-demo --timeout=120s
pod/inference condition met
pod/training condition met
pod/preprocess condition met
pod/analytics condition met
```

Now measure SM active with all four workloads competing again.

bash

```
$ EXPORTER_POD="$(kubectl get pod -n gpu-operator -l app=nvidia-dcgm-exporter -o jsonpath='{.items[0].metadata.name}')"
kubectl exec -n gpu-operator "$EXPORTER_POD" -- curl -s localhost:9400/metrics | grep 'DCGM_FI_PROF_SM_ACTIVE'
DCGM_FI_PROF_SM_ACTIVE{gpu="0"} 47
```

When all four workloads share, SM active drops to 47%.

This matches the pattern we've seen: training alone reaches 92%, but sharing lowers it for everyone to 47%.

The data shows a big drop in efficiency when sharing: SM active goes from 92% alone to 47% shared.

This points to contention or interference.

However, to understand the real impact, we need to compare the training job's steps per second or time to complete, and look at any changes in inference P95 or P99 latency, not just SM active.

Bursty inference and periodic batch jobs can share resources more efficiently because their active times don't usually overlap.

But if you add a workload that needs constant compute, like training, all jobs start to compete for resources in ways Kubernetes can't manage.

That's why time-slicing is great for development and testing, where developers want fast GPU access and flexibility is more important than overhead.

But in production, where workloads compete and you need reliable performance, it doesn't work well.

If time-slicing causes this much overhead when sharing, what happens if we do the opposite and give each workload its own dedicated GPU?

Many Smaller Dedicated GPUs

Another option is to use several smaller, dedicated GPUs instead of sharing one large GPU.

This way, each workload has its own hardware, which removes context-switching overhead and noisy neighbor issues.

However, this approach can affect how well each GPU is used and may impact cost efficiency.

For example, rather than sharing one A100 GPU among four workloads, you could use four T4 GPUs, giving each workload its own dedicated hardware.

Whether this setup is more efficient or just more expensive depends on the specific needs of your workloads.

So, when is it actually worth paying for separate hardware?

When Smaller Dedicated GPUs Win

Dedicated GPUs offer predictable performance because each workload gets steady latency without having to share resources.

Production services with SLAs benefit from this, since there are no unexpected

slowdowns when another workload runs a heavy kernel.

Failure isolation means that if one workload crashes, it does not affect the others.

On shared GPUs, however, a bad workload can destabilize the whole system through memory corruption or driver problems.

You can distribute GPUs across different regions or availability zones to meet data residency rules or lower latency for users.

Dedicated hardware can be more cost-effective when you compare smaller GPUs to larger ones.

For example, seven T4 instances often cost less than a single H100 and can deliver similar total throughput for workloads that do not need the H100's memory bandwidth or size.

You also get more flexibility, since you can add or remove T4s as needed instead of paying for a large H100 even if you do not use all its capacity.

However, dedicated hardware is not always the best choice, especially if your workloads have special requirements.

When Smaller Dedicated GPUs Fall Short

Dedicated GPUs are not enough when your workloads need a lot of memory or high-end compute that smaller GPUs cannot provide.

For example, training large language models that need 32GB of memory will not fit on 16GB T4s, no matter how many you use.

In these cases, you have to use A100 or H100 class hardware, and the cost advantage of smaller GPUs goes away.

Memory-intensive workloads face a big problem here.

If your model needs 40GB of memory and you only have 16GB GPUs, the setup will not work, and you will need to switch to larger hardware.

Dedicated hardware gives you predictable performance, but it can also lead to wasted resources.

The real question is: how do you know if you are getting good value or just wasting money?

Yield Characteristics of Smaller Dedicated

GPUs

With dedicated GPUs, it's easy to measure each workload because every GPU is assigned to just one task.

This way, you always know exactly who is using what.

DCGM metrics with pod attribution show exactly what each workload accomplishes.

Unlike time-slicing, you don't have to separate its results from others or from the total numbers.

For example, your training workload might reach 92% SM active on dedicated hardware, compared to 47% when sharing.

This clear measurement lets you calculate the real cost per unit of work, instead of estimating based on allocation.

However, there's a risk of underusing each GPU.

If you only check allocation dashboards, you might see 100% usage, but miss that SM active is actually just 12%.

If a T4 GPU averages 12% SM active, you're wasting 88% of its capacity but still paying for all of it.

Shared hardware could put those unused cycles to better use.

Having easier measurements means you need to pay attention to the data and adjust your deployments based on what you see, instead of just assuming dedicated hardware is always the best choice.

To sum up, time-slicing can boost overall utilization but adds overhead that lowers efficiency.

Dedicated GPUs offer steady performance but may not be fully used.

Both options make you pick between efficiency and predictability.

But what if you could have both?

Imagine splitting a single high-end GPU into separate, hardware-isolated slices that don't interfere with each other.

You'd get the isolation of dedicated hardware without needing to buy more GPUs.

MIG Partitions

Multi-Instance GPU allows you to divide one physical GPU into separate, hardware-isolated parts.

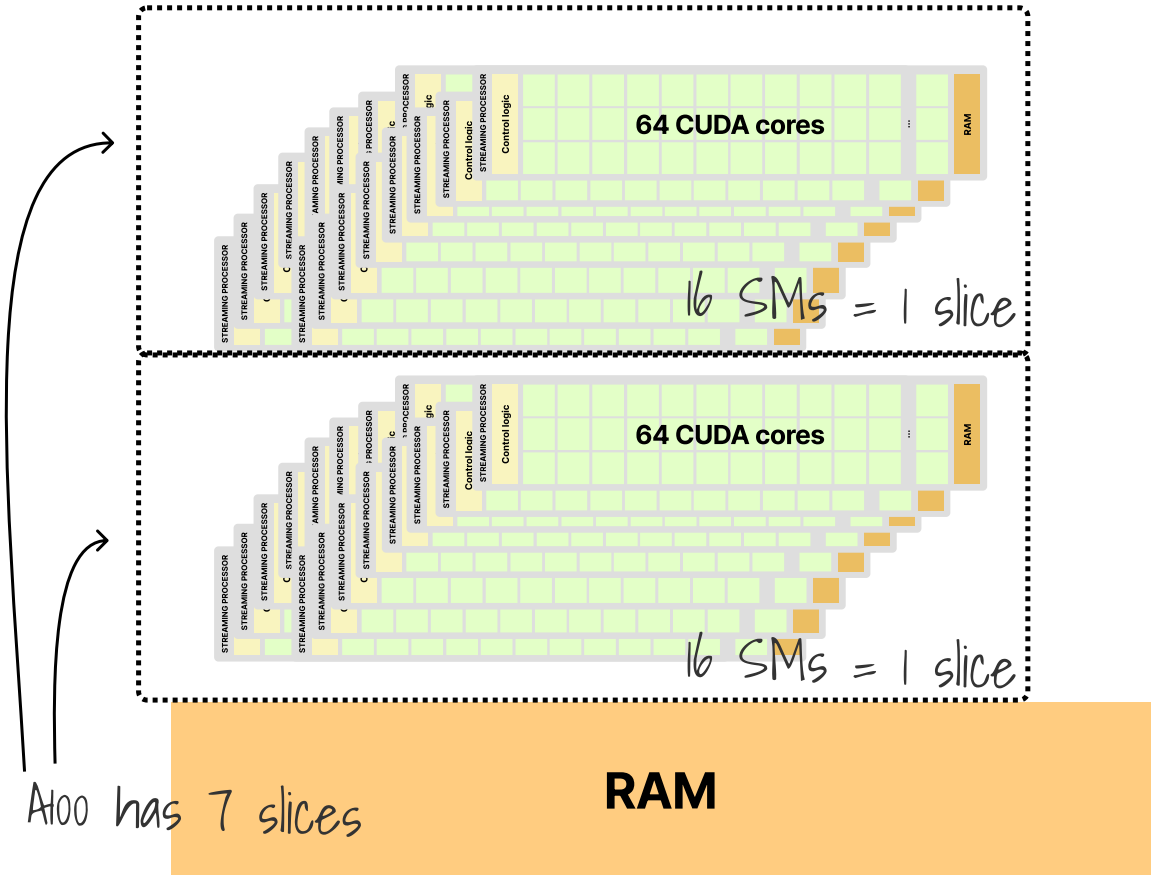


Fig. An A100 GPU divided into two groups of 16 Streaming Multiprocessors each, labelled as slices, with shared RAM at the bottom and a note that the A100 has 7 slices

This setup keeps tasks from interfering with each other and helps you get more out of a high-end GPU.

Instead of using a shared GPU, which can lead to conflicts, or four separate

GPUs that may not be fully used, you can split one A100 into three isolated parts.

For example, you could use 3g.20gb for training, 2g.10gb for inference, and 1g.5gb for preprocessing.

Each part gets its own hardware-isolated share of GPU resources, including compute and memory.

Fig. Seven MIG 1g.10gb partitions arranged in a grid, each with its own set of Streaming Processors and RAM, showing 16 Streaming Processors per slice

There is much less competition between tasks than with time-slicing.

However, you should still check performance using throughput and latency, because bandwidth is not always fully dedicated in every case.

The scheduler sees three separate GPU resources (nvidia.com/mig-3g.20gb,

nvidia.com/mig-2g.10gb, nvidia.com/mig-1g.5gb) and assigns pods to them as needed.

Still, all of these run on the same physical A100.

So, when should you use this middle-ground approach?

When MIG Wins

MIG is a good choice if you need H100 or A100-level hardware and want both isolation and the ability to match resources to your workload.

When your tasks need the GPU's memory, bandwidth, or tensor core features, MIG lets you split the device into slices so smaller jobs run separately without interfering with each other.

If you run your own hardware, MIG helps you get the most out of a GPU you already have.

In the cloud, it lets you rent just part of an H100 when you need its features but don't need a whole card.

MIG does not always save money.

If your workload fits on less expensive GPUs like T4 or L4, using dedicated, smaller cards is usually cheaper than splitting a high-end GPU.

For the cost of one H100, you can often buy several T4s, which are more cost-effective even if they offer less bandwidth and performance.

Multi-tenant environments with SLA requirements benefit because one customer's workload literally cannot affect another customer's partition—the hardware enforces boundaries that software cannot cross.

When training on a MIG slice, you might see high SM active since it avoids time-slicing interference.

However, total throughput is still limited by the slice size, which has fewer SMs and resources than a full GPU.

It's important to compare steps per second or tokens per second, not just SM active.

With MIG, you get predictable performance similar to dedicated hardware, but you only need to pay for one A100 instead of three separate GPUs.

You can also adjust each slice to fit your workload, so you don't have to over-provision whole GPUs.

However, MIG's hardware isolation also brings important limitations that can restrict how you use it.

When MIG Loses

MIG is available only on specific supported NVIDIA GPUs, and it is not supported on consumer GPUs like an RTX 3050, so it's only an option if your fleet includes MIG-capable hardware.

MIG-capable GPUs are much more expensive than other datacenter GPUs.

For example, an A100 costs several times more than a T4.

The "fraction of dedicated GPU cost" benefit only makes sense when comparing MIG slices to buying several A100s, not when comparing to cheaper GPU models.

MIG partition sizes are set by supported profiles, and the names and sizes of these profiles vary by GPU and memory SKU.

For example, the A100 40GB and 80GB use different memory proportions and profile names.

You have to pick from these set profiles instead of choosing any size you want.

If your workload needs 12GB of memory, you usually have to choose the next largest supported profile.

For example, on an A100 40GB, this might mean using a 3g.20gb class slice.

This can leave some memory unused, since MIG cannot create custom sizes like 12GB on demand.

Partitioning also means each slice gets fewer resources, such as fewer SMs, less L2 cache, and lower memory bandwidth.

Because of this, a MIG slice cannot reach the full throughput of an entire GPU, even if SM activity looks good.

MIG helps prevent interference between workloads, but it also brings limits on hardware availability and partition flexibility.

Whether these limits are important depends on what your workloads need.

Yield Characteristics of MIG Partitions

MIG provides real parallel execution without the extra cost of context switching.

This means compute-heavy workloads can reach SM activity levels close to what you get with dedicated hardware.

Training often shows higher SM activity on a MIG slice than with time-slicing.

However, SM activity by itself does not show true performance.

Since a MIG slice has fewer SMs and less capacity than a full GPU, you should compare steps per second, tokens per second, and latency—not just SM activity.

A main risk is partition fragmentation, which lowers overall GPU efficiency when workloads do not match the fixed slice sizes.

For example, if you assign a 3g.20gb slice to a workload that only needs 8GB and uses 30% of the SMs, you end up wasting 12GB of memory and most of the compute power in that partition.

MIG cannot break up slices on the fly to recover unused capacity.

As a result, any stranded resources remain idle, even though you are still paying for them.

To make the right decision, use real data from Chapter 2 about your workload's memory and compute needs instead of guessing.

If your workloads fit MIG's standard profiles, partitioning helps you use hardware efficiently.

But if your workloads need sizes that do not match the fixed profiles, you will waste capacity trying to fit them in.

Next, we will set up MIG instances on an A100, run workloads, and measure how well they use resources.

First, check if your GPU supports MIG and find out which profiles you can use.

bash

```
$ nvidia-smi -L
GPU 0: NVIDIA A100-SXM4-40GB (UUID: GPU-f8d5c3b2-4a1e-8f3d-9c7a-1b5e6d8f2a4c)

$ nvidia-smi mig -lgip
+-----------------------------------------------------------------------------+
| GPU Instance Profiles:                                                      |
| GPU   Name          ID    Instances   Memory     P2P    SM    DEC   ENC   |
|                           Free/Total   GiB              CE    JPEG  OFA   |
```

```
|=============================================================================|
|   0  MIG 1g.5gb     19     7/7        4.75       No     14     0     0    |
|      MIG 2g.10gb    14     3/3        9.75       No     28     1     0    |
|      MIG 3g.20gb     9     2/2        19.50      No     42     2     0    |
|      MIG 4g.20gb     5     1/1        19.50      No     56     2     0    |
|      MIG 7g.40gb     0     1/1        39.75      No     98     5     0    |
+-----------------------------------------------------------------------------+
```

The A100 supports multiple fixed MIG profiles.

The valid combinations are constrained by the GPU's supported layout rules for that model, but a common upper bound is that you can create up to seven 1g-class instances, or fewer larger instances.

Enable MIG mode on the GPU.

This typically requires a GPU reset to take effect (and in some environments a node reboot).

bash

```
$ nvidia-smi -mig 1
Enabled MIG Mode for GPU 00000000:00:04.0
All done.
```

Create three MIG instances that match our workload patterns—two small instances for inference and preprocessing, and one larger instance for training.

bash

```
$ nvidia-smi mig -cgi 19,19,9 -C
Successfully created GPU instance ID  1 on GPU  0 using profile MIG 1g.5gb (ID 19)
Successfully created GPU instance ID  2 on GPU  0 using profile MIG 1g.5gb (ID 19)
Successfully created GPU instance ID  3 on GPU  0 using profile MIG 3g.20gb (ID  9)
```

Verify the MIG devices exist and each has its own UUID.

bash

```
$ nvidia-smi -L
```

```
GPU 0: NVIDIA A100-SXM4-40GB (UUID: GPU-f8d5c3b2-4a1e-8f3d-9c7a-1b5e6d8f2a4c)
  MIG 1g.5gb   Device 0: (UUID: MIG-2f8a4d1c-9e3b-5a7f-8d2c-4e6b1f9a3d7c)
  MIG 1g.5gb   Device 1: (UUID: MIG-7c3e9a2f-5d1b-4f8e-9a6c-2d4f7e9b1a5c)
  MIG 3g.20gb  Device 2: (UUID: MIG-9b5d2a7f-3e8c-4f1a-9d6e-5b2c8f4a7d1e)
```

Now configure Kubernetes to expose MIG as separate allocatable resources via the NVIDIA device plugin (through GPU Operator).

Apply the MIG device plugin configuration:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: device-plugin-mig-config
  namespace: gpu-operator
data:
  any: |
    version: v1
    flags:
      migStrategy: mixed
EOF
configmap/device-plugin-mig-config created
```

Patch the ClusterPolicy so the device plugin runs with `migStrategy: mixed` and advertises MIG profiles as distinct resources.

bash

```
$ kubectl patch clusterpolicy cluster-policy \
  -n gpu-operator \
  --type merge \
  -p '{"spec":{"mig":{"strategy":"mixed"},\
  "devicePlugin":{"config":{"name":"device-plugin-mig-config","default":"any"}}}}'
clusterpolicy.nvidia.com/cluster-policy patched
```

Wait for the device plugin to restart and re-advertise resources.

bash

```
$ kubectl wait --for=condition=ready pod -n gpu-operator -l app=nvidia-device-plugin-daemonset --timeout=90s
pod/nvidia-device-plugin-daemonset-k5pjr condition met
```

Check that the node now shows MIG profiles as allocatable resources.

bash

```
$ kubectl describe node gpu-node-01 | grep nvidia.com/mig
  nvidia.com/mig-1g.5gb:   2
  nvidia.com/mig-3g.20gb:  1
  nvidia.com/mig-1g.5gb:   2
  nvidia.com/mig-3g.20gb:  1
```

Deploy the same workloads as before, but request specific MIG profiles instead of a full GPU.

bash

```
$ kubectl create namespace mig-demo
namespace/mig-demo created
```

The workload scripts are the same as before. Create the ConfigMap in the `mig-demo` namespace:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: ConfigMap
metadata:
  name: workload-configs
  namespace: mig-demo
data:
  inference.py: |
    import torch, time
```

```
    device = torch.device('cuda:0')
    model = torch.nn.Linear(1024, 1024).to(device)
    while True:
        x = torch.randn(32, 1024, device=device)
        y = model(x)
        torch.cuda.synchronize()
        time.sleep(0.1)
  training.py: |
    import torch
    device = torch.device('cuda:0')
    A = torch.randn(4096, 4096, device=device)
    B = torch.randn(4096, 4096, device=device)
    while True:
        C = torch.matmul(A, B)
        torch.cuda.synchronize()
  preprocess.py: |
    import torch, time
    device = torch.device('cuda:0')
    while True:
        data = torch.randn(1024, 1024, device=device)
        normalized = (data - data.mean()) / data.std()
        time.sleep(0.05)
EOF
configmap/workload-configs created
```

Then deploy Pods that request specific MIG profiles. For example, for `inference`:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: inference
  namespace: mig-demo
spec:
  containers:
  - name: worker
    image: pytorch/pytorch:2.1.0-cuda12.1-cudnn8-runtime
    command: ["python", "/app/inference.py"]
    volumeMounts:
```

```
    - name: configs
      mountPath: /app
    resources:
      limits:
        nvidia.com/mig-1g.5gb: 1
  volumes:
  - name: configs
    configMap:
      name: workload-configs
EOF
pod/inference created
```

Repeat for `preprocess` with `nvidia.com/mig-1g.5gb`, and `training` with `nvidia.com/mig-3g.20gb`.
Wait for all pods to reach running state.

bash

```
$ kubectl wait --for=condition=ready pod --all -n mig-demo --timeout=120s
pod/inference condition met
pod/preprocess condition met
pod/training condition met
```

Check nvidia-smi to confirm multiple MIG devices are active while workloads run.

bash

```
$ DRIVER_POD="$(kubectl get pod -n gpu-operator -l app=nvidia-driver-daemonset -o jsonpath='{.items[0].metadata.name}')"
$ kubectl exec -n gpu-operator "$DRIVER_POD" -- nvidia-smi
Fri Jan 19 10:12:45 2026
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 550.163.01             Driver Version: 550.163.01     CUDA Version: 12.4     |
+-----------------------------------------------------------------------------+
| MIG devices:                                                                |
|=============================================================================|
| GPU  GI  CI  MIG |         Memory-Usage |        Vol|      Shared         |
|      ID  ID  Dev |           BAR1-Usage | SM     Unc|     CE ENC DEC OFA  |
|                  |                      |        Err|                     |
|=============================================================================|
|  0    1   0   0  |    847MiB /  4864MiB | 12      0 |  0   0    0    0   |
|  0    2   0   1  |    512MiB /  4864MiB | 8       0 |  0   0    0    0   |
|  0    3   0   2  |  15234MiB / 20096MiB | 89      0 |  2   0    0    0   |
+-----------------------------------------------------------------------------+
```

Each MIG instance shows independent memory usage and SM utilization, and the training workload on the 3g.20gb instance (device 2) achieves 89% SM while the smaller workloads run simultaneously on their dedicated instances.

Now measure compute yield using DCGM profiling metrics to confirm the training workload maintains high SM active on its isolated partition.

bash

```
$ EXPORTER_POD="$(kubectl get pod -n gpu-operator -l app=nvidia-dcgm-exporter -o jsonpath='{.items[0].metadata.name}')"
kubectl exec -n gpu-operator "$EXPORTER_POD" -- \
  curl -s localhost:9400/metrics | grep 'DCGM_FI_PROF_SM_ACTIVE{gpu="0",GPU_I_ID="3"}'

DCGM_FI_PROF_SM_ACTIVE{gpu="0",GPU_I_ID="3"} 89
```

The training workload achieves 89% SM active on its MIG slice, which nearly matches the 92% it achieved when running alone on a full GPU during our time-slicing test, and this is dramatically better than the 47% collective SM active we saw when all four workloads shared the GPU through time-slicing.

MIG provides hardware isolation that eliminates context-switching overhead while enabling true parallel execution, so each workload gets predictable performance without interference from neighbors.

Key Takeaways

These three architectures each work differently.

With time-slicing, several pods share one GPU.

Dedicated GPUs assign each pod its own hardware.

MIG splits a single GPU into separate, isolated slices.

Time-slicing reduces compute productivity due to context-switching overhead.

For example, training can reach 92% SM active when running alone, but drops to 47% when sharing with three other workloads.

This 45-point drop means much lower GPU efficiency.

Performance loss can come from interference, scheduling overhead,

synchronization, and stalls.

To see the real impact, check changes in throughput and latency.

Dedicated GPUs provide steady performance, but they may be underused.

For example, a T4 at 12% SM active wastes 88% of its capacity, even though you pay for the whole GPU.

Dashboards showing 100% allocation can hide this waste.

MIG provides hardware-level isolation on supported GPUs.

A training job might show high SM active on a MIG slice compared to time-slicing.

However, SM active alone does not mean performance is equal, since a slice has fewer SMs and less total capacity than a full GPU.

Compare steps per second or tokens per second, and check latency, not just SM active.

Fixed partition sizes can waste capacity if a workload falls between supported profiles.

You have to round up to the next available MIG profile for that GPU model, which leaves unused memory stranded inside the instance.

Choosing the right architecture needs measurement data from Chapter 2, such as actual SM active use, usage patterns over time, and isolation needs.

These answers come from DCGM metrics, not vendor marketing or theory.

Some workloads do not need GPUs but are still scheduled on them out of habit.

Workloads with less than 5% SM active often run faster on CPUs because GPU initialization takes longer than the computation.

Sometimes, the best choice is to use no GPU architecture at all.

Full-Stack Right-Sizing

GPU nodes don't exist in isolation.

When you provision one from any cloud provider, you get a bundle with fixed ratios.

When you set up a 4-GPU training node, you get a fixed package of GPUs, CPU, and RAM, and you pay for the whole bundle.

Chapters 1-3 covered GPU metrics and architecture choices, but you can't look at GPU efficiency alone.

Wasted GPU resources also waste CPU and memory.

If your GPU usage puts pressure on other resources, your costs go up for the whole bundle, not just the GPU part.

Wasting resources on pricey GPU instances is even more expensive than on regular compute nodes.

This problem starts even before your pods begin running.

You don't get access to all the CPU and memory on the node for your workloads.

The operating system needs resources to function, the kubelet needs resources to manage containers, and the kubelet enforces eviction thresholds for memory and ephemeral storage; when a hard eviction threshold is met, the kubelet can evict pods immediately, and by default, those hard eviction thresholds are withheld from Node Allocatable (unless configured to ignore them).

It's important to know how these reservations work on GPU nodes, since unused resources on expensive GPU instances cost much more than on general-purpose nodes.

Let's look at the real impact to see how wasted GPU allocation leads to wasted money.

The Cascade Effect

Cloud providers bundle GPUs, CPUs, and memory together in fixed ratios on GPU nodes.

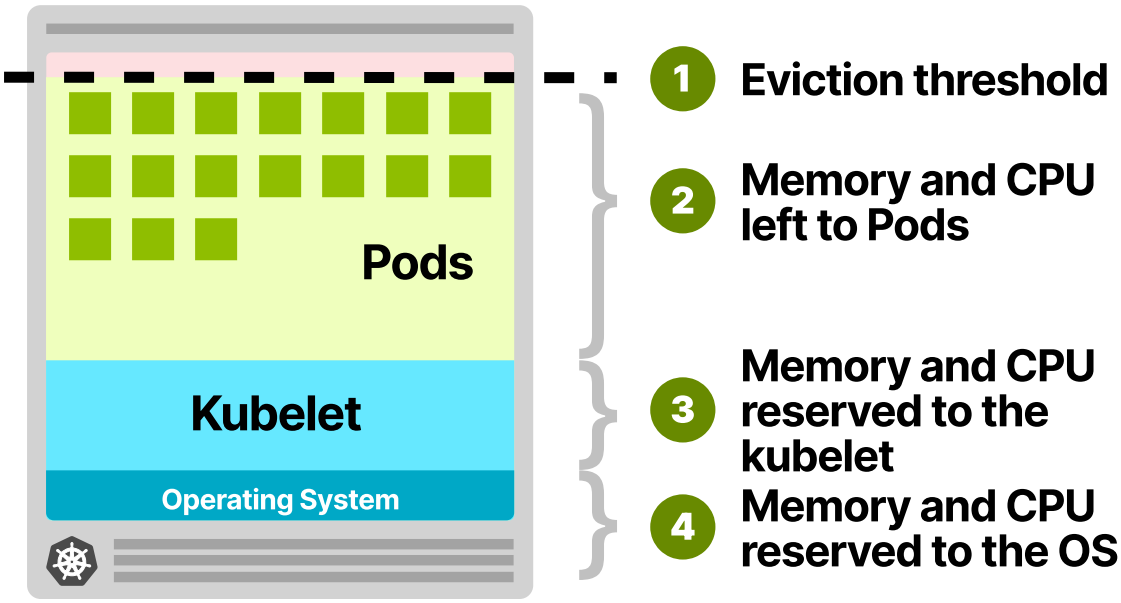


Fig. A node diagram showing four resource layers: the eviction threshold at top, memory and CPU left to pods, memory and CPU reserved for the kubelet, and memory and CPU reserved for the operating system at the bottom

On-prem DGX systems work the same way: each node comes with a set GPU, CPU, and memory ratio.

If GPUs are the bottleneck, the CPU and RAM you paid for also go unused.

Once you pick an instance type, you're stuck with its resource ratios.

If your workloads don't match what the bundle offers, you end up paying for resources you can't use.

The cascade effect happens when GPU allocation is the bottleneck, so you can't use the CPU and memory you're paying for in the bundle.

On a single-GPU node, most CPU and memory go unused once the GPU is

allocated, since the scheduler won't add more GPU workloads to that node.

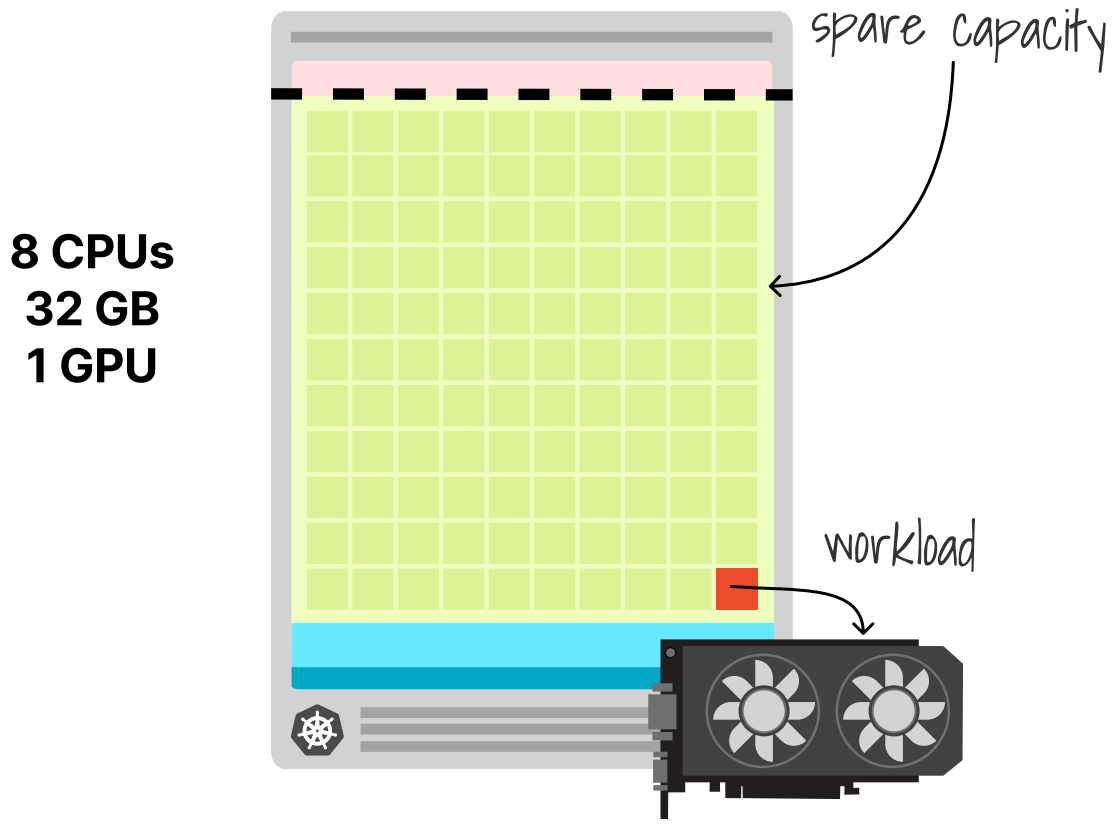


Fig. A node with 8 CPUs, 32 GB and 1 GPU showing a single small workload using the GPU while most of the CPU and memory capacity sits unused

We'll use a 4-GPU node running a simple matrix-multiplication workload to measure real resource use and see how much capacity goes unused because of GPU allocation.

bash

```
$ kubectl create namespace cascade-demo
namespace/cascade-demo created
```

Let's create four training Pods, each requesting 1 GPU, 2 CPU cores, and 8Gi of memory. For example, for `training-pod-1`:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: training-pod-1
  namespace: cascade-demo
spec:
  containers:
  - name: trainer
    image: pytorch/pytorch:2.1.0-cuda12.1-cudnn8-runtime
    command: ["python", "-c"]
    args:
    - |
      import torch
      device = torch.device('cuda:0')
      A = torch.randn(4096, 4096, device=device)
      B = torch.randn(4096, 4096, device=device)
      while True:
          C = torch.matmul(A, B)
          torch.cuda.synchronize()
    resources:
      requests:
        cpu: "2000m"
        memory: "8Gi"
      limits:
        nvidia.com/gpu: 1
        cpu: "2000m"
        memory: "8Gi"
EOF
pod/training-pod-1 created
```

Repeat for `training-pod-2`, `training-pod-3`, and `training-pod-4`, changing only `metadata.name`.

Each of these four pods requests 2 CPU cores, 8GB of memory, and 1 GPU.

Once the pods reach running state:

```bash
$ kubectl wait --for=condition=ready pod --all -n cascade-demo --timeout=120s
pod/training-pod-1 condition met
pod/training-pod-2 condition met
pod/training-pod-3 condition met
pod/training-pod-4 condition met
```

Check the node's total capacity and what Kubernetes has marked as allocatable after accounting for system reservations.

```bash
$ kubectl describe node gpu-node-01 | grep -A 10 "Capacity:"
Capacity:
  cpu:                64
  ephemeral-storage:  1921802544Ki
  hugepages-1Gi:      0
  hugepages-2Mi:      0
  memory:             263856332Ki
  nvidia.com/gpu:     4
  pods:               110
Allocatable:
  cpu:                63580m
  memory:             250Gi
```

This node has 64 CPU cores total, but the Allocatable field shows 63580m, which means about 420 millicores are not allocatable to pods because of Node Allocatable reservations (for example `system-reserved` and `kube-reserved`) configured on the kubelet.

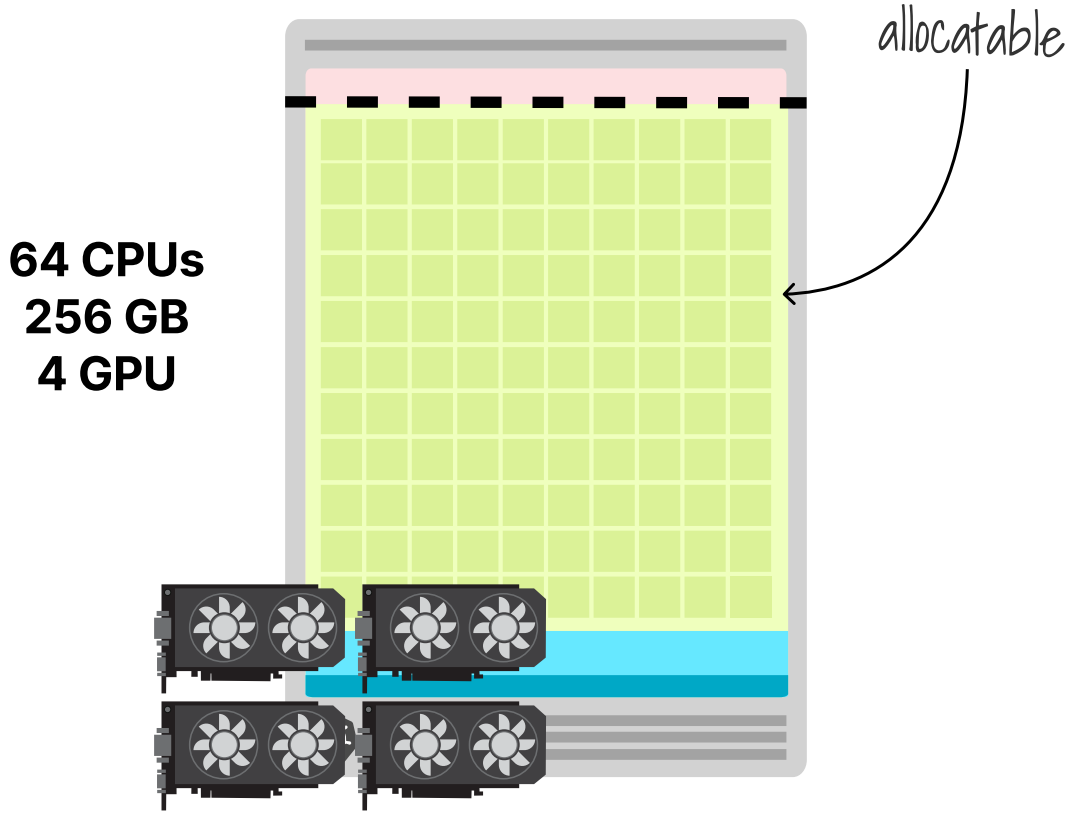


Fig. A node with 64 CPUs, 256 GB and 4 GPUs showing the allocatable resources after system reservations, with all capacity still available

For memory, Allocatable is computed from Capacity after Node Allocatable reservations (kube-reserved/system-reserved) and is normally lower than Capacity, so treat your node's `Capacity` and `Allocatable` as the authoritative numbers and don't assume fixed deltas.

Now check what the scheduler thinks these four pods have allocated from the node's total capacity.

bash

```
$ kubectl describe node gpu-node-01 | sed -n '/Allocated resources/,/^$/p'
Allocated resources:
```

```
Resource           Requests      Limits
--------           --------      ------
cpu                8000m (12%)   8000m (12%)
memory             32Gi (12%)    32Gi (12%)
ephemeral-storage  0 (0%)        0 (0%)
hugepages-1Gi      0 (0%)        0 (0%)
hugepages-2Mi      0 (0%)        0 (0%)
nvidia.com/gpu     4             4
```

This table shows what pods have requested, not what they're actually using.

From the scheduler's perspective, these four pods have allocated all 4 GPUs and requested 8 CPU cores plus 32GB of memory.

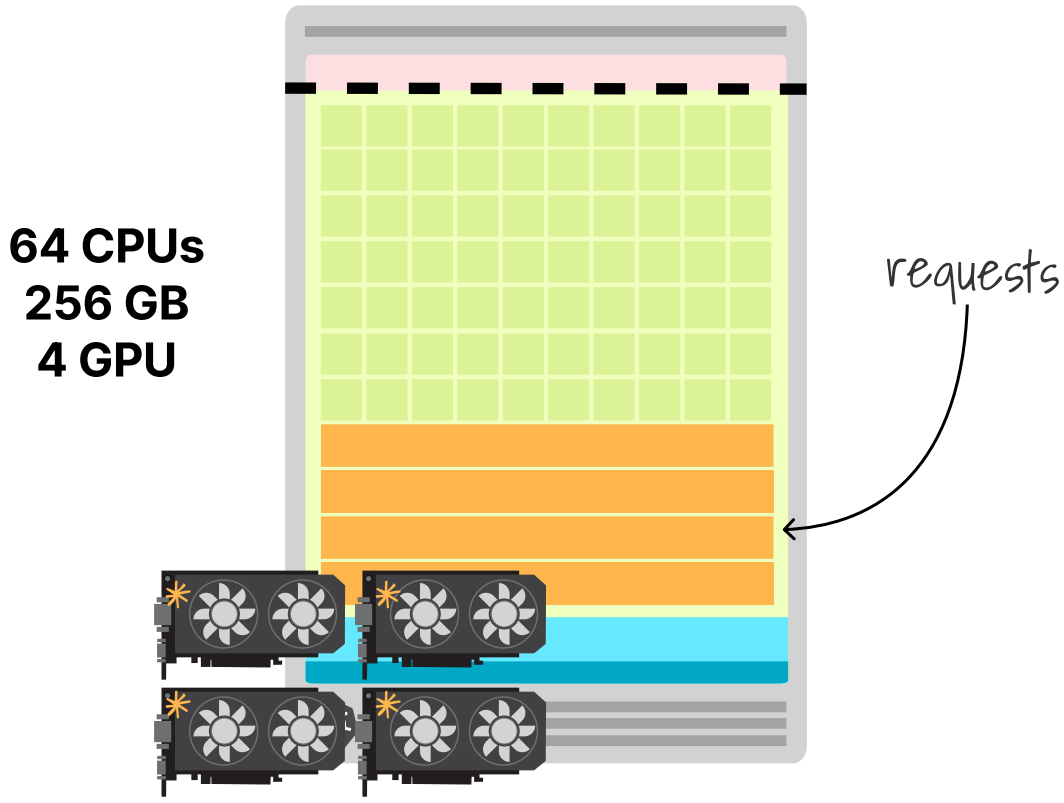


Fig. The same node now showing resource requests from four pods as orange bars filling part of the allocatable space, with all 4 GPUs allocated but only a fraction of CPU and memory requested

The percentages show 12% CPU and 12% memory, which looks efficient at first glance.

But these percentages measure requested resources against allocatable capacity, not actual consumption.

To see consumption, `kubectl top` shows recent resource usage from the Metrics Server, while Prometheus provides time-series data to capture peaks and percentiles.

Measure what the pods are actually consuming to see the gap between what they requested and what they're using.

bash

```
$ kubectl top pods -n cascade-demo
NAME               CPU(cores)   MEMORY(bytes)
training-pod-1     523m         1821Mi
training-pod-2     518m         1819Mi
training-pod-3     527m         1823Mi
training-pod-4     521m         1820Mi
```

The pods are using much less than what we asked for.

Each pod uses about 520m CPU and 1.8Gi memory, but we requested 2000m CPU and 8Gi memory for each.

To see the difference, let's compare what one pod asked for to what it actually uses.

For CPU, the pod requested 2000m but is using 520m, which means it's using 520m/2000m = 26% of its requested amount.

For memory, the pod requested 8Gi but uses 1.8Gi, which means it's using 1.8Gi ÷ 8Gi = 23% of what it requested.

Now multiply this across all four pods.

Total requested CPU is 4 pods × 2000m = 8000m (or 8 CPU cores).

Total actual CPU usage is 4 pods × 520m = 2080m (about 2.1 CPU cores).

That means 5.9 CPU cores are sitting idle, blocked by the requests.

For memory, total requested is 4 pods × 8Gi = 32Gi, but total actual usage is 4 pods × 1.8Gi = 7.2Gi.

So, 24.8Gi of memory is also sitting idle, blocked by the requests.

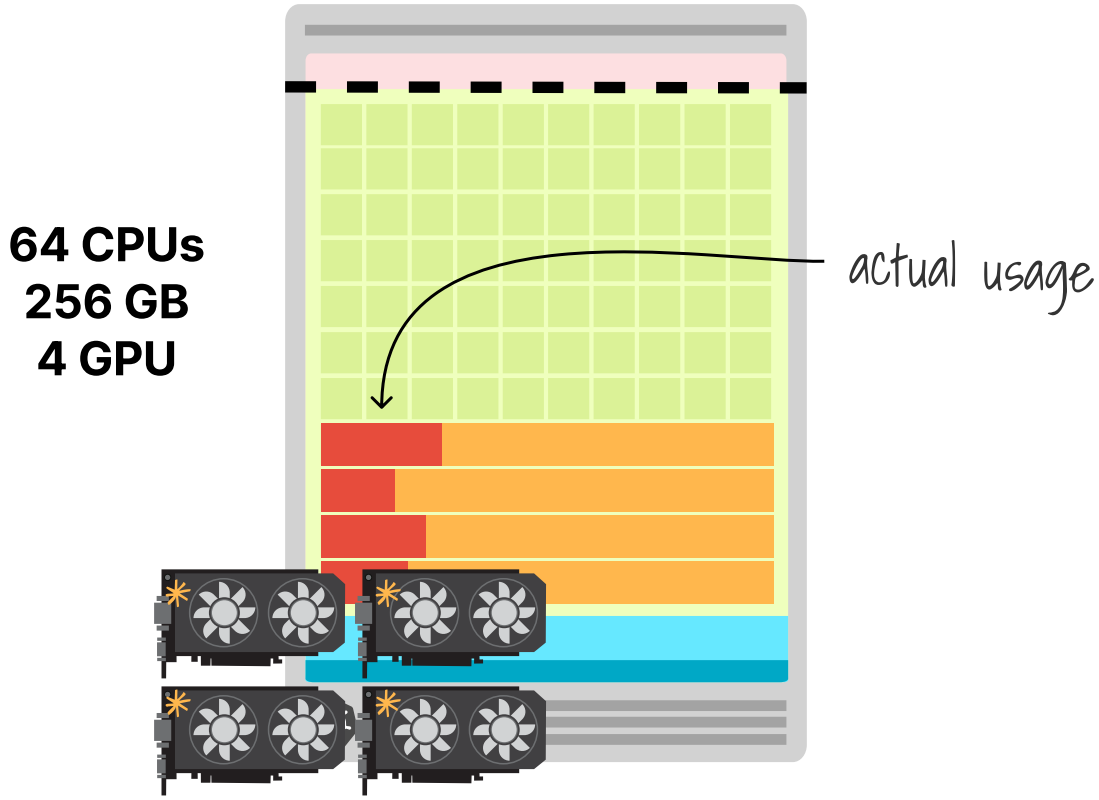


Fig. The node with requests shown in orange and actual usage shown in red, where the red bars are much smaller than the orange bars, illustrating the gap between what pods requested and what they actually consume

Let's verify these numbers at the node level to see the complete picture.

bash

```
$ kubectl top nodes
NAME          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gpu-node-01   2108m        3%     7284Mi          2%
```

The node shows a total CPU usage of 2108 m, which is about 3% of allocatable

CPU capacity.

Memory usage is 7284Mi, which `kubectl top` rounds to 2% of allocatable memory.

These numbers should be similar, but pod-level and node-level kubectl top values can differ.

The remaining CPU and memory can still be used by CPU-only pods, but they can't increase GPU workload throughput on that node because pods that request GPUs can't be scheduled once all GPU devices are allocated.

This is where costs start to pile up.

GPU nodes are expensive; the hourly price for a 4-GPU instance can range widely by GPU generation, region, and pricing model.

In this example, the GPUs are fully allocated, while CPU usage is 3% and memory is 2%.

Most of what you pay for each hour isn't being used.

The scheduler is doing exactly what it should by respecting the pod requests.

The issue is that these requests don't match what the pods really need.

Right-sizing means measuring what your workloads actually use (with Chapter 2's metrics) and then setting requests to match.

Once you know your workloads' real usage, pick the instance type that fits those needs.

Instance Selection

Cloud providers offer dozens of GPU instance types with different GPU models, different GPU counts, and different CPU-to-memory ratios.

The challenge is to find an instance that fits your workload so you don't pay for unused resources.

Consider a workload that Chapter 2's metrics show uses 8GB of GPU memory, 75% of SMs active, 4 CPU cores, and 16GB of RAM during production traffic.

You have several options:

- p3.2xlarge: 1× V100, 8 vCPU, 61 GiB RAM - overprovisions CPU and RAM significantly
- g5.xlarge: 1× A10G (24 GB GPU memory), 4 vCPU, 16 GiB RAM - matches CPU and RAM requirements perfectly
- g4dn.xlarge: 1× T4, 4 vCPU, 16 GiB RAM - typically lower cost per hour but slower GPU performance

You can't pick just by looking at hourly price, and pricing models make things even more complicated.

A cheaper instance that cuts your throughput in half can actually cost more for the same amount of work.

You should test real throughput under production load, then figure out the cost per unit of work using current prices and your performance data.

Multi-GPU instances, such as p3.8xlarge with 4 V100s, reduce per-GPU costs but come with trade-offs.

When a node fails, you lose 4 GPUs instead of 1, resulting in a greater impact on your workloads.

These instances also require that multiple workloads can actually schedule together efficiently on the same node.

Use Chapter 2's metrics to measure consumption patterns and validate performance under production conditions before committing to an instance type.

Even with good measurements, pods have strict scheduling rules, so right-sizing on GPU nodes is as much about packing resources as it is about sizing them.

Pod-Level Constraints on GPU Nodes

GPU scheduling is not a simple optimization.

It is a challenging bin packing problem with strict limits.

The pod specification requires the scheduler to place GPU, CPU, and memory

together on one node.

This constraint leads to predictable failures where GPU nodes appear full in dashboards, even though actual utilization remains low.

GPU Requests Are All-or-Nothing

A pod that requests one GPU blocks the entire device, even if it uses only 5% of it.

Kubernetes does not support partial GPU allocation unless you use MIG or time-slicing.

As a result, a single low-intensity workload can lock an expensive device and stop other GPU pods from being scheduled.

Imagine a 4-GPU node running three inference pods.

Each pod requests 1 GPU but uses only 10% of the device's compute power.

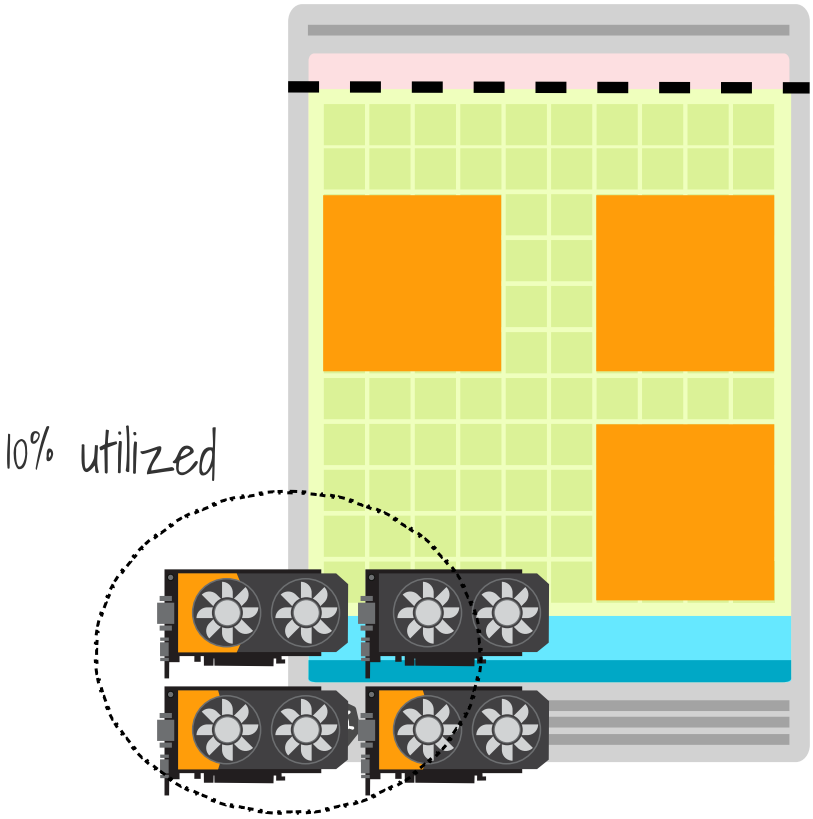


Fig. A node with 4 GPUs where three large workloads each claim one whole GPU but only use 10% of its compute, leaving the GPUs highlighted in orange as mostly idle

bash

```
$ kubectl top pods -n inference
NAME              CPU(cores)   MEMORY(bytes)
inference-pod-1   200m         2Gi
inference-pod-2   195m         2Gi
inference-pod-3   203m         2Gi
```

A new training job that could use 80% of a GPU cannot be scheduled on any of the three occupied GPUs, even though together they have spare capacity equal to 2.7 full GPUs.

The scheduler sees three GPUs as allocated and cannot split them.

As a result, the training job must wait for a completely free GPU or stay pending.

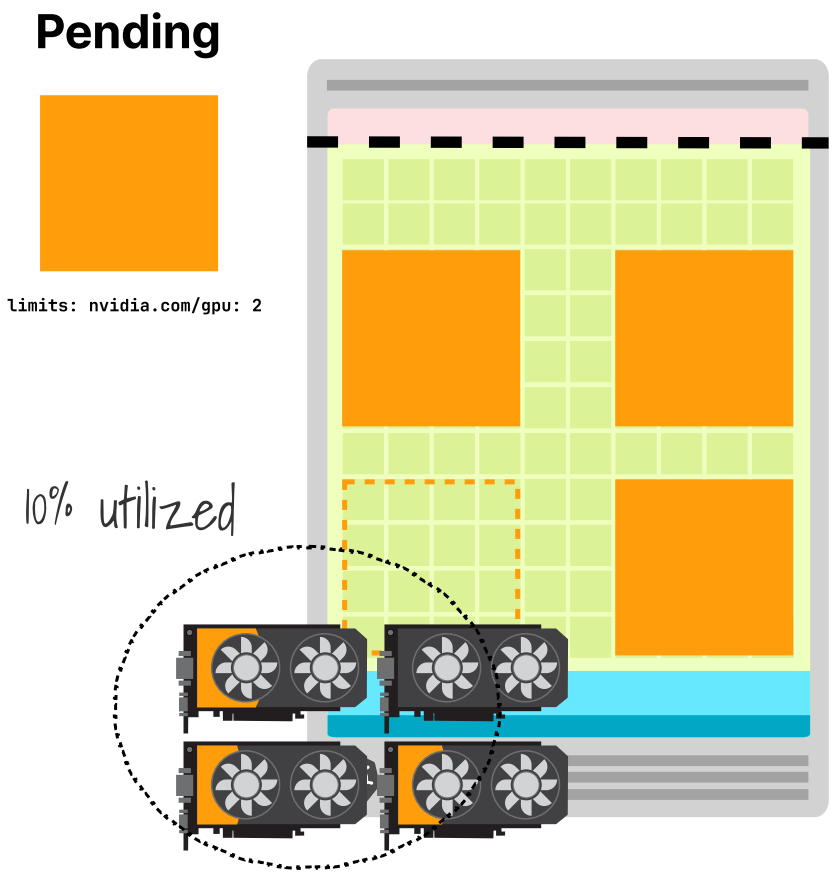


Fig. A pending pod requesting 2 GPUs that cannot be scheduled because the remaining free GPU slot is only 1, with a dashed outline showing where it would need to fit

Without MIG or time-slicing, there is no way to share a single GPU among multiple pods.

This means low-intensity workloads leave capacity unused that could support more jobs.

CPU and Memory Fragmentation

GPU nodes provide separate CPU and memory along with GPUs.

A pod that requests 1 GPU and a large amount of CPU or memory might not fit, even if GPUs are available.

This leads to a bin packing failure.

The scheduler cannot place the pod anywhere, even though GPU capacity looks available in cluster dashboards.

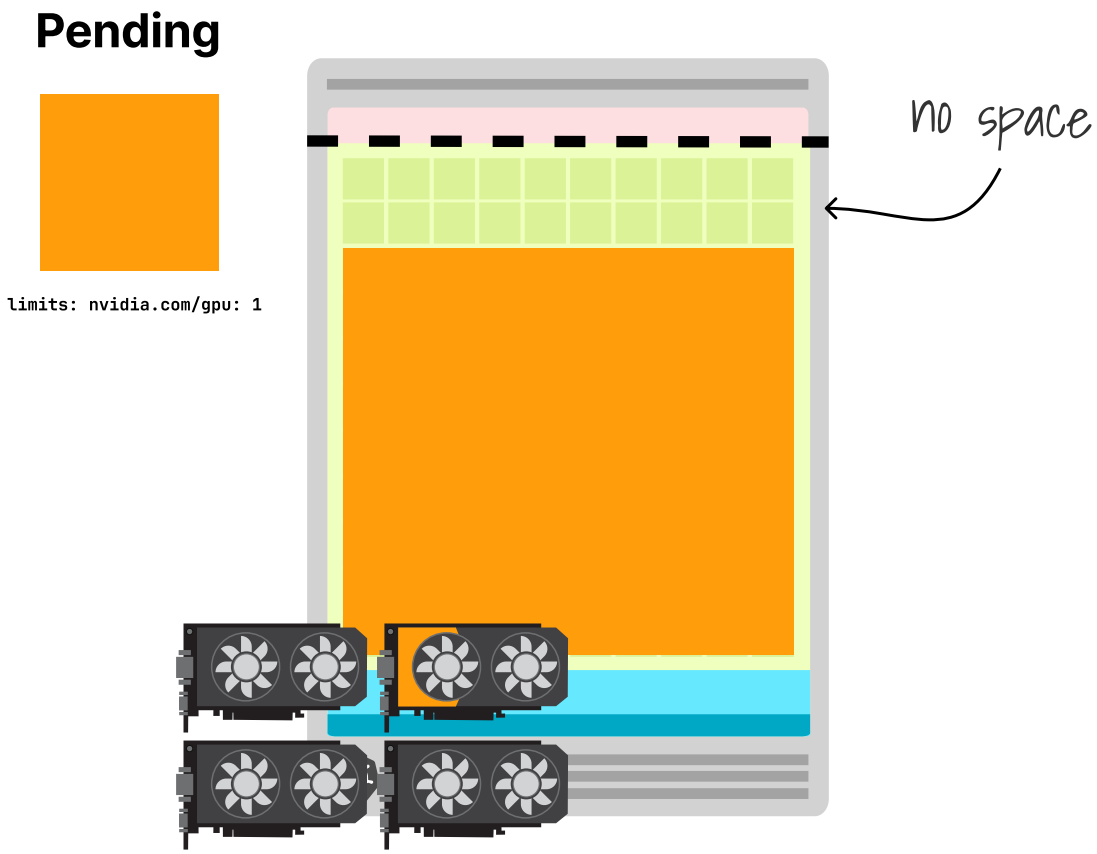


Fig. A pending pod requesting 1 GPU that cannot be scheduled because an existing workload has consumed nearly all the CPU and memory on the node, leaving no space even though a GPU is available

Start with a 4-GPU node where three pods are already running, each using 1 GPU, 4 CPU cores, and 16Gi memory.

bash

```
$ kubectl describe node gpu-node-01 | sed -n '/Allocated resources/,/^$/p'
Allocated resources:
  Resource           Requests       Limits
  --------           --------       ------
  cpu                12000m (18%)   12000m (18%)
  memory             48Gi (19%)     48Gi (19%)
  nvidia.com/gpu     3              3
```

The node has 1 GPU free, 51.58 CPU cores available (63.58 allocatable minus 12 allocated), and about 202Gi memory available (250Gi allocatable minus 48Gi allocated).

Now try to schedule a pod that requests 1 GPU, 60 CPU cores, and 8Gi memory.

bash

```
$ kubectl create namespace fragmentation-demo
namespace/fragmentation-demo created
```

Let's create a Pod that requests 1 GPU and 60 CPU cores:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: high-cpu-training
  namespace: fragmentation-demo
spec:
  containers:
  - name: trainer
    image: pytorch/pytorch:2.1.0-cuda12.1-cudnn8-runtime
    command: ["sleep", "3600"]
    resources:
      requests:
        cpu: "60000m"
        memory: "8Gi"
```

```
      limits:
        nvidia.com/gpu: 1
        cpu: "60000m"
        memory: "8Gi"
EOF
pod/high-cpu-training created
```

Check the pod status:

bash

```
$ kubectl get pod high-cpu-training -n fragmentation-demo
NAME                 STATE     RESTARTS   AGE
high-cpu-training    Pending   0          15s
```

The pod stays pending because the node has only 51.58 CPU cores available, but the pod requests 60 cores.

The free GPU cannot be allocated because the CPU request cannot be satisfied on the same node, and the scheduler cannot split the pod's resource requests across multiple nodes.

bash

```
$ kubectl describe pod high-cpu-training -n fragmentation-demo | grep -A 5 "Events:"
Events:
  Type     Reason            Age   From              Message
  ----     ------            ----  ----              -------
  Warning  FailedScheduling  20s   default-scheduler  0/1 nodes are available: 1 Insufficient cpu.
```

This is an example of CPU fragmentation.

The GPU is free, but the remaining CPU on the node is not enough for the pod's request.

The same type of failure can happen with memory.

A pod that requests 1 GPU and 210Gi memory cannot fit on a node with only 202Gi available, even if the GPU and CPU are both free.

The scheduler treats these constraints as strict limits.

It will not place the pod unless all three resources—GPU, CPU, and memory—can be met on one node.

GPU dashboards might show 25% free capacity (1 out of 4 GPUs available), but the pod cannot use that capacity because the resource bundle is incomplete.

Multi-GPU Pods and Cross-Node Fragmentation

If a pod asks for 2 or 4 GPUs, all those GPUs need to be on the same node.

Having free GPUs scattered across the cluster does not help.

Even if there is enough total GPU capacity, you might not be able to schedule the job because the GPUs are split between different nodes.

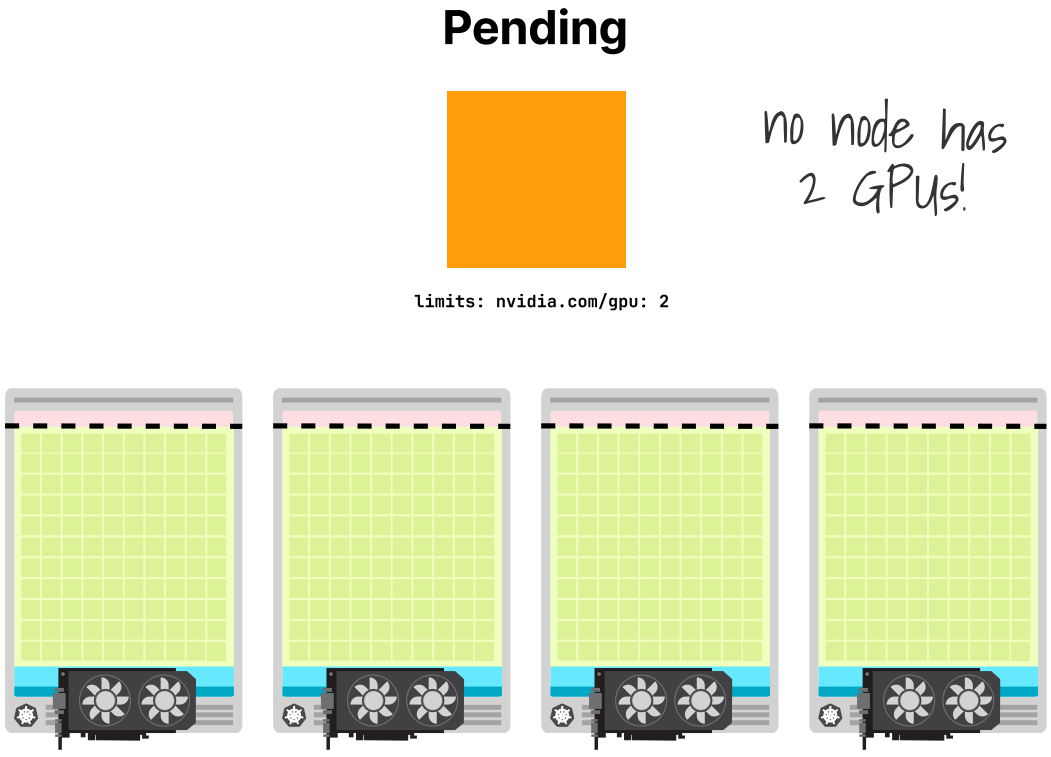


Fig. A pending pod requesting 2 GPUs with four single-GPU nodes below it, each node having only 1 GPU, so no single node can satisfy the request even though the cluster has enough GPUs in total

Imagine a cluster with three nodes.

Each node is running two pods that each use one GPU, so there is one GPU left free on each node.

bash

```
$ kubectl get nodes -l node-type=gpu -o custom-columns=NAME:.metadata.name,CAPACITY_GPU:.status.capacity.nvidia\.com/gpu
kubectl describe nodes -l node-type=gpu | grep -A 1 "Allocated resources:" | grep "nvidia.com/gpu"
NAME          CAPACITY_GPU
gpu-node-01   2
gpu-node-02   2
gpu-node-03   2

  nvidia.com/gpu     1             1
  nvidia.com/gpu     1             1
  nvidia.com/gpu     1             1
```

Across the whole cluster, there are 3 free GPUs.

That seems like it should be enough to run a pod that needs 2 GPUs.

Next, try to schedule a training job that asks for 2 GPUs:

bash

```
$ kubectl create namespace multi-gpu-demo
namespace/multi-gpu-demo created
```

Let's create a Pod that requests 2 GPUs:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: two-gpu-training
  namespace: multi-gpu-demo
spec:
  containers:
  - name: trainer
    image: pytorch/pytorch:2.1.0-cuda12.1-cudnn8-runtime
    command: ["sleep", "3600"]
    resources:
      requests:
        cpu: "8000m"
        memory: "32Gi"
      limits:
        nvidia.com/gpu: 2
        cpu: "8000m"
        memory: "32Gi"
EOF
pod/two-gpu-training created
```

Check the pod status:

bash

```
$ kubectl get pod two-gpu-training -n multi-gpu-demo
NAME                 STATE     RESTARTS   AGE
two-gpu-training     Pending   0          10s
```

The pod stays pending even though the cluster has 3 free GPUs total.

bash

```
$ kubectl describe pod two-gpu-training -n multi-gpu-demo | grep -A 5 "Events:"
Events:
  Type     Reason            Age   From               Message
  ----     ------            ----  ----               -------
  Warning  FailedScheduling  15s   default-scheduler  0/3 nodes are available: 3 Insufficient nvidia.com/gpu.
```

The scheduler cannot use GPUs from different nodes together, so the job cannot run until one node has 2 free GPUs.

This situation is called cross-node fragmentation.

There is enough total capacity, but it is not arranged in a way that fits the pod's needs.

The problem is even harder for pods that need 4 or 8 GPUs, since it is rare to find a single node with that many free GPUs as usage goes up.

Pods that use multiple GPUs also make failures more costly.

If a node with 4 GPUs fails, you lose all 4 GPUs at once instead of just one.

Any multi-GPU training job on that node has to restart from its last checkpoint.

Isolation and Blocked Backfill

GPU nodes are often marked to keep CPU-only workloads off them.

This stops other jobs from using the leftover CPU and memory when GPUs are fully used.

This isolation is done on purpose for reasons like cost tracking, security, or keeping workloads separate.

But it also means unused resources cannot be used at all, which makes the problem worse.

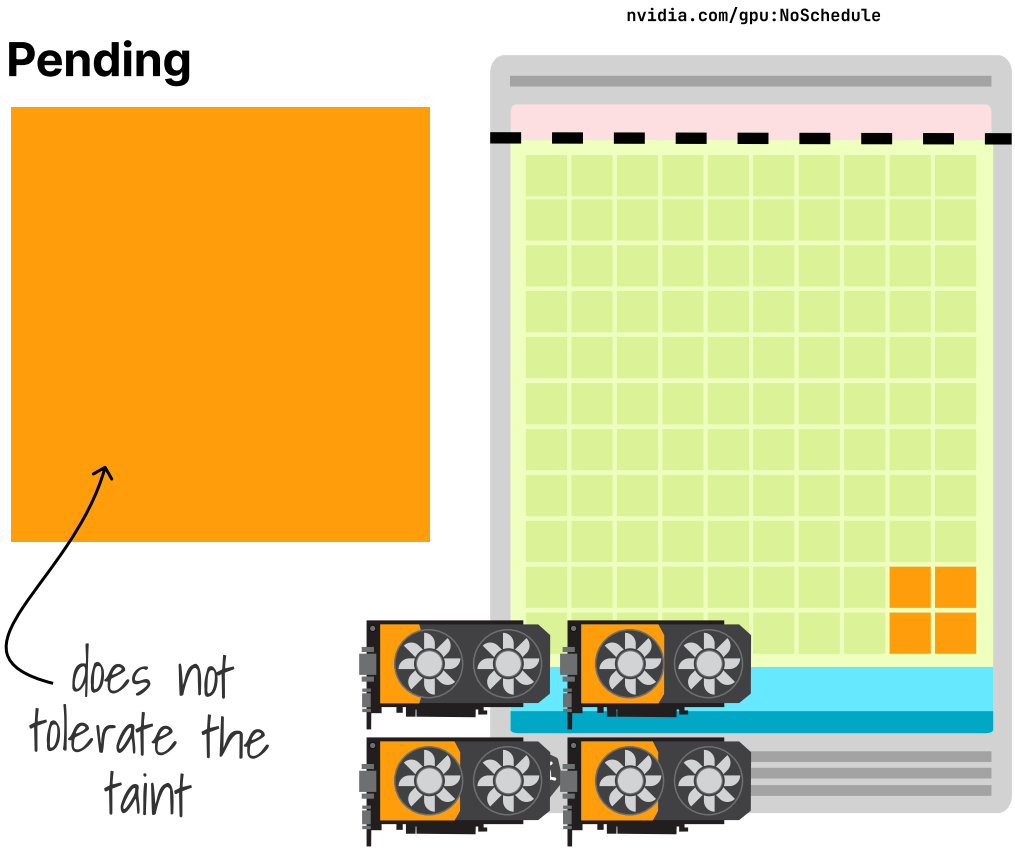


Fig. A pending pod that does not tolerate the nvidia.com/gpu NoSchedule taint, shown next to a GPU node with spare capacity and a small workload, illustrating that the pod cannot be placed despite available resources

Consider a GPU node tainted to prevent non-GPU workloads from scheduling:

bash

```
$ kubectl describe node gpu-node-01 | grep -A 3 "Taints:"
Taints:
  nvidia.com/gpu:NoSchedule
```

This taint ensures that only pods explicitly tolerating it can be scheduled on the node.

Now assume the node is running four GPU pods, each requesting 1 GPU, 2 CPU cores, and 8Gi memory.

bash

```
$ kubectl top node gpu-node-01
NAME          CPU(cores)   CPU%   MEMORY(bytes)   MEMORY%
gpu-node-01   2150m        3%     7300Mi          2%
```

The node has 97% of its CPU and 98% of its memory free, but no workload can use those resources unless it is set up to tolerate the `nvidia.com/gpu:NoSchedule` taint.

A CPU-intensive batch job that could make use of the 61 free CPU cores cannot be scheduled on this node:

bash

```
$ kubectl create namespace backfill-demo
namespace/backfill-demo created
```

Let's create a CPU-intensive batch job:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: cpu-batch-job
  namespace: backfill-demo
spec:
  containers:
  - name: worker
    image: ubuntu:22.04
    command: ["bash", "-c", "for i in {1..100}; do echo $i; done; sleep 3600"]
    resources:
      requests:
        cpu: "30000m"
```

```
        memory: "64Gi"
      limits:
        cpu: "30000m"
        memory: "64Gi"
EOF
pod/cpu-batch-job created
```

Check the pod status:

bash

```
$ kubectl get pod cpu-batch-job -n backfill-demo
NAME            STATE     RESTARTS   AGE
cpu-batch-job   Pending   0          8s
```

The pod remains pending because the GPU node is tainted, and the pod does not have the right toleration.

bash

```
$ kubectl describe pod cpu-batch-job -n backfill-demo | grep -A 5 "Events:"
Events:
  Type     Reason            Message
  ----     ------            -------
  Warning  FailedScheduling  0/1 nodes are available: 1 node(s) had untolerated taint {nvidia.com/gpu: }.
```

The 61 free CPU cores and 242Gi of memory on the GPU node go unused because the isolation policy blocks any non-GPU workloads.

If you remove the taint or add tolerations so CPU-only workloads can run, you allow backfill.

But you also lose the cost isolation and workload separation that the taint was supposed to provide.

You have to choose between leaving resources unused or mixing workloads.

Many organizations prefer to leave capacity unused to keep GPU and non-GPU costs separate.

Priority and Preemption Limitations

Priority preemption can remove lower-priority pods to make space for higher-priority ones, but it does not fix the problem when GPU, CPU, and memory requests do not fit well together.

If no node has the right combination of free resources, the high-priority pod will remain pending, even after removing as many pods as possible.

Suppose a GPU node is running four low-priority inference pods, and each one uses 1 GPU, 2 CPU cores, and 8Gi of memory.

bash

```
$ kubectl get pods -n inference -o custom-columns=NAME:.metadata.name,PRIORITY:.spec.priority
NAME              PRIORITY
inference-pod-1   100
inference-pod-2   100
inference-pod-3   100
inference-pod-4   100
```

Next, submit a high-priority training pod that asks for 2 GPUs, 16 CPU cores, and 64Gi of memory, with a priority of 1000.

bash

```
$ kubectl create namespace priority-demo
namespace/priority-demo created
```

Let's create a high-priority training Pod that requests 2 GPUs:

bash

```
$ kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: high-priority-training
  namespace: priority-demo
spec:
  priority: 1000
  containers:
  - name: trainer
```

```
    image: pytorch/pytorch:2.1.0-cuda12.1-cudnn8-runtime
    command: ["sleep", "3600"]
    resources:
      requests:
        cpu: "16000m"
        memory: "64Gi"
      limits:
        nvidia.com/gpu: 2
        cpu: "16000m"
        memory: "64Gi"
EOF
pod/high-priority-training created
```

The scheduler will remove two of the low-priority inference pods to free up 2 GPUs, but the node still does not have the right mix of resources after that.

Even if the scheduler removes `inference-pod-1` and `inference-pod-2`, freeing 2 GPUs, 4 CPU cores, and 16Gi of memory, there still are not enough resources left.

The high-priority pod needs 16 CPU cores and 64Gi of memory, but after removing the two pods, the node only has 4 free CPU cores and 16Gi of memory from those pods, plus whatever was unused before.

If the original unused capacity was minimal (because the inference pods consumed most of it), the high-priority pod cannot fit even after preemption.

bash

```
$ kubectl get pod high-priority-training -n priority-demo
NAME                      STATE     RESTARTS   AGE
high-priority-training    Pending   0          25s
```

bash

```
$ kubectl describe pod high-priority-training -n priority-demo | grep -A 5 "Events:"
Events:
  Type     Reason            Message
  ----     ------            -------
  Warning  FailedScheduling  0/1 nodes are available: 1 Insufficient memory, 1 Insufficient cpu.
```

Preemption removes pods, but it cannot change the way resources are divided on the node.

If no single node has the right combination of GPU, CPU, and memory for the high-priority pod, it will stay pending no matter how high its priority is.

Priority helps make sure that important workloads can replace less important ones, but it does not fix the main problem of fitting pods with strict resource needs onto nodes.

NUMA and Topology Constraints

On systems with more than one GPU, tying CPU and memory to specific GPUs (using PCIe lanes or NVLink) can make placement harder and lower the usable capacity.

NUMA (Non-Uniform Memory Access) means it is faster to use memory or CPU cores that are physically closer to a GPU than to use resources on another NUMA node.

High-performance workloads often need NUMA alignment to avoid slowdowns from using memory across sockets, which can reduce performance by 20-40% on some systems.

When a pod requests a GPU, the kubelet's topology manager (if enabled) tries to align CPU and memory to the same NUMA node as the GPU.

If the needed CPU or memory is not available on that NUMA node, the pod cannot be scheduled, even if those resources are free elsewhere on the same machine.

Consider a dual-socket node with 2 GPUs per socket, 32 CPU cores per socket, and 128Gi memory per socket.

A pod that needs 1 GPU, 40 CPU cores, and 64Gi of memory can only be scheduled if one NUMA node (socket) has all three resources available.

If NUMA node 0 has 1 free GPU but only 28 free CPU cores, and NUMA node 1 has 35 free CPU cores but no free GPU, the pod cannot be scheduled because the topology manager cannot satisfy the alignment requirement.

bash

```
$ kubectl describe pod numa-constrained -n topology-demo | grep -A 5 "Events:"
```

```
Events:
  Type     Reason            Message
  ----     ------            -------
  Warning  FailedScheduling  0/1 nodes are available: 1 Insufficient cpu (due to NUMA alignment).
```

Standard resource dashboards add up CPU and memory for the whole node, so operators might see 'plenty of free CPU' even though pods are still pending because of topology constraints.

NUMA alignment is important for performance, but it makes bin packing harder since it divides a node's resources into separate pools that can't be shared.

Daemonset and System Overhead

GPU nodes often run more system services than regular compute nodes.

These services include GPU drivers, DCGM exporters, monitoring agents, and log collectors.

These services use CPU and memory, which reduces the resources available for user workloads and makes bin packing harder.

The kubelet reserves resources using the `kube-reserved` and `system-reserved` flags.

These reserved amounts are subtracted from the node's total capacity to show the actual available capacity.

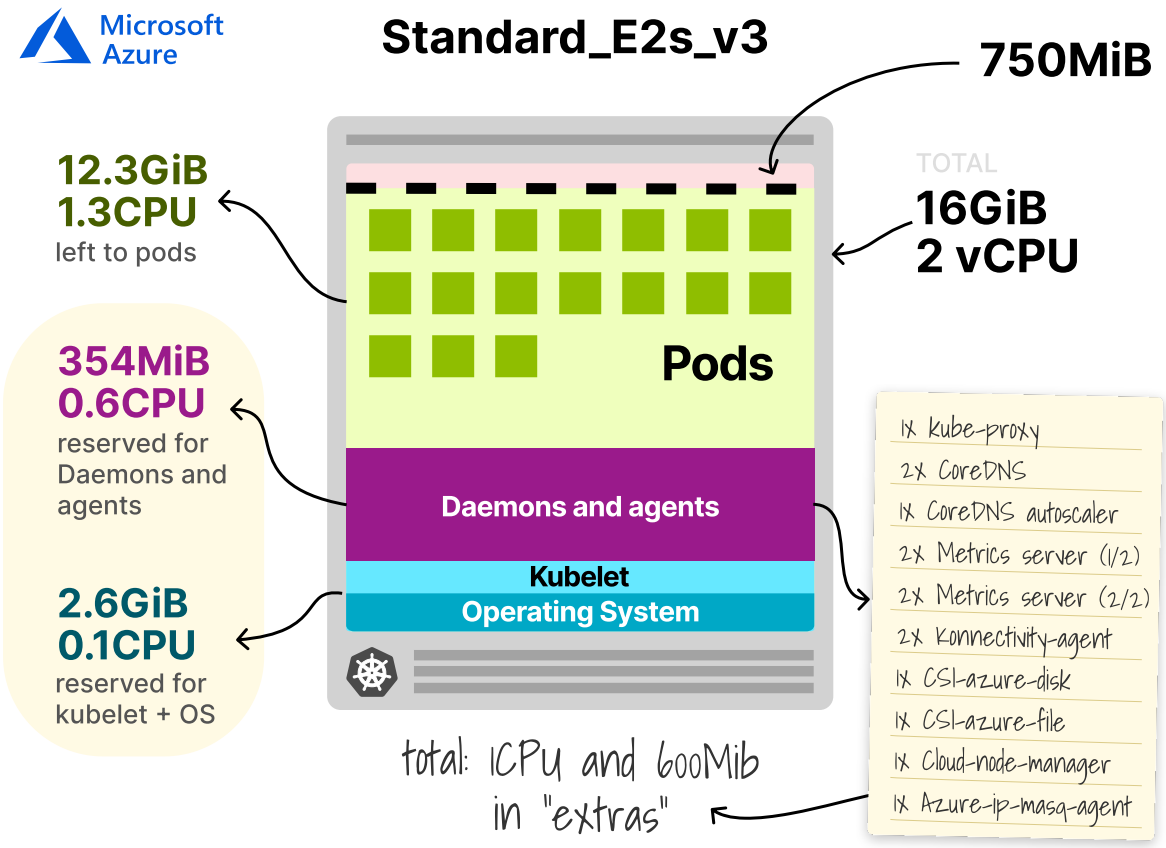


Fig. Breakdown of an Azure Standard E2s v3 node with 16 GiB and 2 vCPU total: 2.6 GiB and 0.1 CPU reserved for kubelet and OS, 354 MiB and 0.6 CPU reserved for daemons and agents, 750 MiB for the eviction threshold, and 12.3 GiB and 1.3 CPU left to pods, with a list of system DaemonSets consuming about 1 CPU and 600 MiB

For example, a GPU node with 64 CPU cores and 256Gi memory might show:

bash

```
$ kubectl describe node gpu-node-01 | grep -A 10 "Allocatable:"
Allocatable:
  cpu:                63580m
  ephemeral-storage:  1800Gi
  memory:             250Gi
```

```
nvidia.com/gpu:     4
pods:               110
```

The allocatable CPU is 63.58 cores instead of 64, and the allocatable memory is 250Gi instead of 256Gi.

System reservations and daemonsets use up the difference.

Additionally, GPU nodes often run vendor-specific daemonsets like NVIDIA's DCGM exporter, which can request 500m CPU and 1Gi memory per node.

If you run 5 such daemonsets (drivers, monitoring, logging, security scanning, and networking), the total overhead is 2.5 CPU cores and 5Gi memory before any user workloads are scheduled.

This overhead reduces the effective capacity available for packing user pods, so a node that appears to have "61 free CPU cores" in a dashboard actually has only 58.5 cores available after accounting for daemonset requests.

When you add in fragmentation from GPU allocation, this overhead makes it even harder to fit large pods onto nodes that already have some resources in use.

Mixed Workload Pools

When you share GPU nodes between inference jobs, which are short and bursty, and training jobs, which run longer and use more resources, you can run into resource shortages and unpredictable delays.

Inference workloads typically request GPUs for short bursts, use 10-30% of the device's compute capacity, and require low-latency responses.

Training jobs keep GPUs busy for hours or even days, use 80-100% of the GPU's capacity, and don't need fast response times.

When both types of workloads share the same GPU node pool, the scheduler has no way to distinguish between them or prioritize based on workload characteristics.

For example, an inference pod asking for one GPU might end up on a node where three training pods are already using their GPUs fully.

This can cause overheating or memory issues, which slow down inference.

Or, a training pod might be placed on a node with three idle inference pods.

If those inference pods start getting traffic and use their GPUs, the training job's speed drops because they all share things like PCIe bandwidth and CPU resources.

The scheduler only sees resource requests and limits, not workload behavior, so it cannot prevent these conflicts.

If you separate inference and training jobs into their own node pools using different taints or labels, you avoid resource conflicts.

However, this makes it harder to use all available resources efficiently, since each pool has less flexibility to handle changes in workload.

You have to choose between predictable performance with separate pools or higher resource use with mixed pools.

The best option depends on whether your jobs can handle slower responses and lower throughput from sharing resources.

Knowing these limits helps you design workloads and clusters that use resources more effectively, without sacrificing performance or cost control.

Key Takeaways

GPU nodes come with set amounts of CPU and memory alongside the GPUs.

If GPU allocation gets stuck, the CPU and memory you paid for can go unused.

Choosing the right instance means matching what you actually use to the specs for GPU memory, compute, CPU, RAM, and cost.

A cheaper instance that only gives you half the throughput ends up costing more for each unit of work.

Developers often ask for more resources than needed to avoid out-of-memory errors, since there's no downside to over-allocating.

But these safety margins stack over time with no feedback loop to remove them.

To profile workloads, you need to collect baseline metrics from real production

traffic using DCGM metrics from Chapter 2 and kubectl top.

To right-size well, keep your resource requests close to what you actually use most of the time, and check against peak usage (like p95 or maximum) from time-series data.

The right buffer depends on how bursty your workload is, how much failure you can tolerate, the instance types you use, and ongoing measurement as things change.